

Advanced programming paradigms

1. Programming paradigms

The von Neumann architecture



- John von Neumann (1903-1957)
- Idea to have the program stored with the data
- Idea of the program counter (PC)
 - ▶ Loops
 - ▶ Conditions

Source: *Wikipedia* page on von Neumann, retrieved 12.2.2014

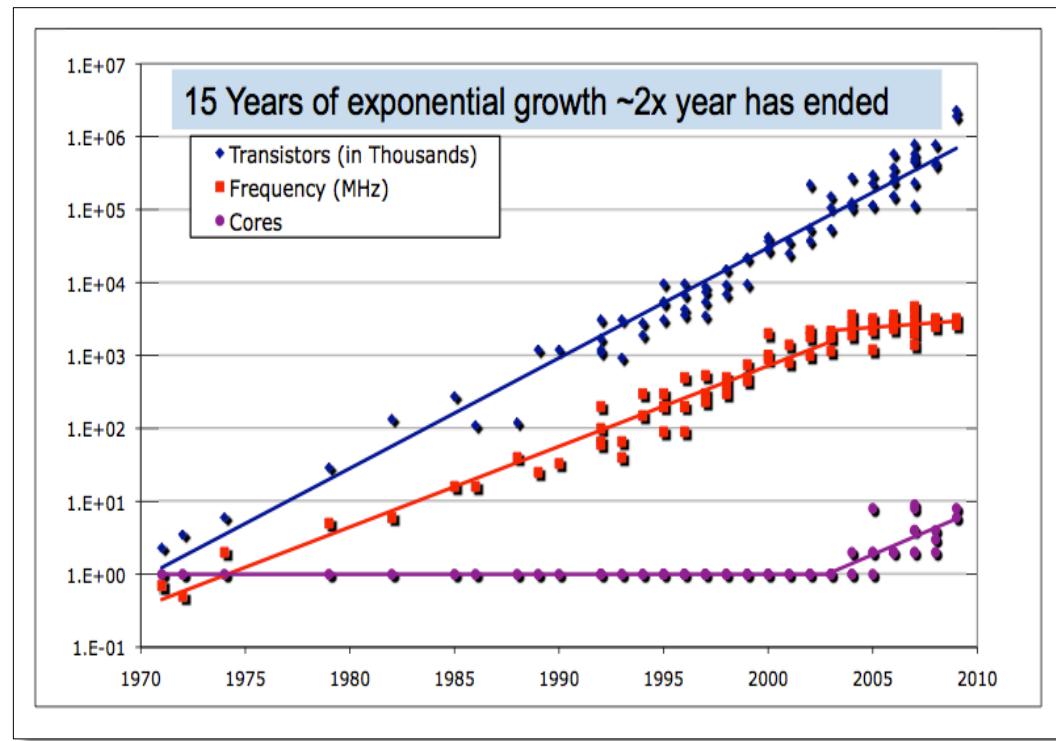
This is a sum

```
int n=100, sum=0;  
while(n!=0) {  
    sum=sum+n;  
    n=n-1;  
}  
System.out.println(sum);
```

Written in **imperative** style
programming

The world is changing

- Moore's law achieved with **cores**, no longer transistors or frequency



Data from K. Olukotun, L. Hammond, H. Sutter, B. Smith, C. Batten, and K. Asanovic, taken from Odersky M. @ DEVOXX11

Why is concurrency difficult?

shared data —> race condition

locks —> deadlock

priority inversion (pseudo deadlocks avec des threads plus prioritaires qui prennent tout le CPU, les moins prioritaires n'ont plus la chance de s'exécuter)

Concurrency vs parallelism

- Parallel programming
 - Parallel hardware gives faster programs
- Concurrent programming
 - Execute multiple threads explicitly

Root of the problem

“Non-determinism caused by concurrent threads accessing shared mutable state.”

Odersky M. @ FOSDEM09

```
var x = 0
async { x = x + 1 }
async { x = x * 2 }
```

} // can give 0, 1, 2
car lecture/écriture non atomique

A solution: stateless programming

- What does «without state» imply on a programming language?

without state = sans variable (que de l'immutable)

same parameters = same output, always

mathématiques: on exécute des fonctions, mais rien ne change, les “variables” sont des containers sans état

Lambda calculus

- Idea of stateless programming not new
 - Stems from lambda calculus in the 1930's
 - Alonso Church (1903-1995)
 - $\lambda - calc \equiv Turing\ machine$
- Algebra (and mathematics) do not have «state»
- Roots of **functional** programming

Source: Wikipedia page on Alonso Church, retrieved on 12.2.2014



Functional programming

In a **restricted** sense, functional programming (FP) means programming without mutable variables, assignments, loops, and other imperative control structures.

Functional programming

In a **wider** sense, functional programming (FP) means focusing on functions.

Functional programming

Why FP?

Multi-core
and cloud

Advanced
paradigms

Verification

Functional
programming

Course objectives

1. Provide a good understanding of functional programming (FP)
2. Apply advanced programming paradigms (DSLs, // collections, ...) to real world problems
3. Verify programs

Lesson's objectives

First steps in AdvPrPa

1. Course organization
2. Programming paradigms
3. Correctness
4. Referential transparency
5. Evaluation strategies

Administrative aspects

1. COURSE ORGANIZATION

Lecturers

- 2 people
 - First 10 lessons → me, P.-A. Mudry
 - My office: HES-SO VS, room A304
pierre-andre.mudry@hevs.ch
 - Verification part (4 lessons)
 - Edgar Lederer, Fachhochschule NWSchweiz, Windisch
edgar.lederer@fhnw.ch



Course schedule

- 3 periods per week
 - ▶ Tuesdays, from 15h00 to 17h25
 - ▶ 15 minutes break
- Course and exercises mixed
- Questions ?
 - ▶ During the course
 - ▶ Email
 - ▶ Meeting (after class, ...)



Course grading

- 1 mark for the mini-project, *Proj*
- 1 mark for final exam, *Exam*

$$\text{Final grade} = \frac{(Proj + 3 * Exam)}{4}$$

Course content

FP programming

6 weeks

- Paradigms
- Absence of state
- Lazy vs eager evaluation
- Types and type inference
- Higher-order functions
- Pattern matching
- An application

Multi-paradigms

4 weeks

- OOP with traits and mixins
- Generic types and type variance
- Infinite streams
- Parallelism & futures
- DSLs

Verification

4 weeks

- Reliability via testing
- Hoare logic
- Verification tools
- Verification condition generator
- Dafny

Exam

March

Mini-project

Beginning
of June

June

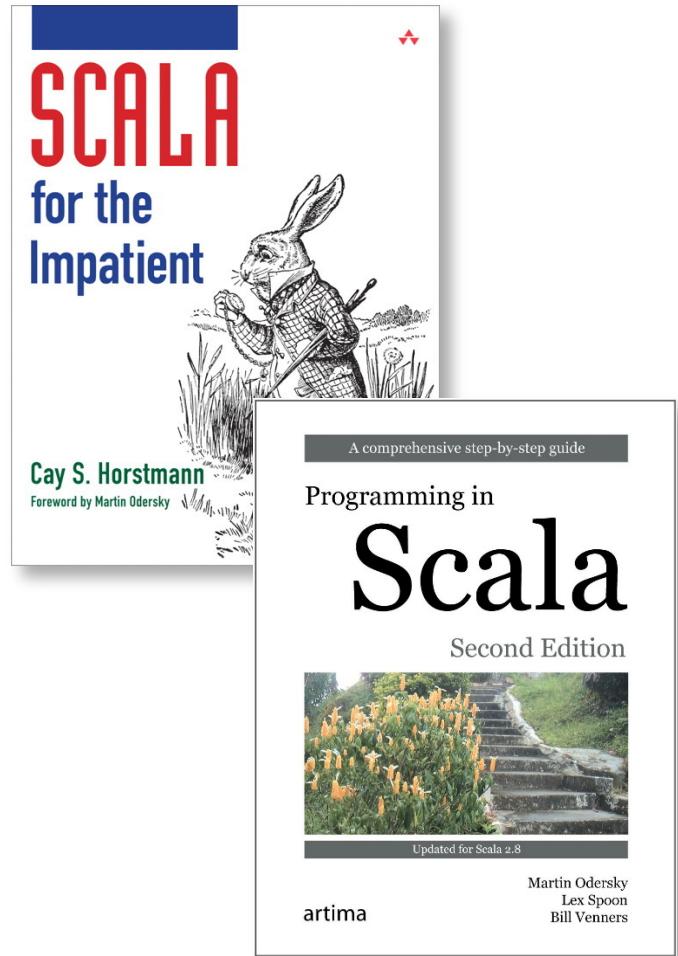
Course specifics

- Fourth edition:
 - Fine tuning.
 - Difficulty should be ok.
 - *Much more students!*
- Will (do my best to) adapt.
- (Assistant help)



Bibliography

[1] Cay S. Horstmann,
Scala for the Impatient
Addison Wesley, 2012.



[2] Martin Odersky, Lex
Spoon and Bill Venners,
*Programming in Scala 2nd
edition*, Artima, 2010.

Secondary bibliography

- [3] David Gries, *The Science of Programming*, Springer, 1981 (a classical text).
- [4] Federico Biancuzzi and Shane Warden, *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*, O'Reilly, 2009 (as leisure).
- [5] H. Abelson, G.J. Sussman & J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press, 1996

Material sources for this course

- Multiple examples and exercises from this course are taken from
 - ▶ M. Odersky, *Functional programming in Scala*, Coursera course
 - ▶ E. Lederer & D. Gruntz, *AdvPrgParadigm* course, Zürich
 - ▶ H. Abelson, G.J & J. Sussman, *Structure and Interpretation of Computer Programs*

Starting the course

2. PROGRAMMING PARADIGMS

Definition

A paradigm describes **thought patterns**, ways of thinking, in some scientific discipline.

Paradigm

- In CS, a programming paradigm can be understood as the fundamental style of programming
 - ▶ how do we reason about programs ?
 - ▶ what can we represent in a program ?
 - state
 - concurrency/parallelism
 - determinism

Main programming paradigms

programmation impérative
(Ada, C, Fortran, etc.)

programmation déclarative
(SQL, regexp)

OO
synchrone
concurrente
event-based
—> ce qu'on fait avec vs comment on le pense

programmation fonctionnelle
(Haskell, Lisp, Camel)

programmation logique
(Prolog)

Logic programming

- Based on predicate logic (1st order)
- *Concepts:*
 - Some logic formulas describe relations
 - Computer solves relations until resolution
- *Typical language:*
 - Prolog
- Use case: The cabbage/wolf/sheep problem, used for AI

Imperative programming (1)

- Assignments to modify mutable variables ≈ change state
- *Concepts:*
 - Uses control structures if-then-else, loops, breaks, goto ,continue, return
 - Abstractions with procedures and functions
- *Typical languages:*
 - C, Ada, Fortran, Pascal...

Imperative programming (2)

très utilisé parce que tellement facile: new processor, implement load, store, jump and bump,
in few weeks you can have a compiler working with all the C, gcc and tools.

- Close to von Neumann machine, where

 Mutable variables ≡ memory

 Using variable ≡ load

 Assigning variable ≡ store

 Control structures ≡ jump

- Problems:

1. Scaling up

2. Building higher-level abstractions

John Backus, Turing Award Lecture, *Can Programming Be Liberated from the Von Neumann Style?*, 1978

Backus: Fortran et les regex

Object-oriented programming

- Strongly-based on imperative programming
- *Concepts:*
 - Objects as instances: data + functions
 - Encapsulation, modularity through inheritance
 - Subtyping, polymorphism and dynamic binding
 - Genericity
- *Typical languages:*
 - C++, Java, Eiffel, Objective-C, Smalltalk...

Declarative programming

- Expresses **what** you want (the logic) without giving the control flow (**how**).
- *Characteristics:*
 - Not always Turing complete
 - Umbrella paradigm that encloses other paradigms (FP for instance)
- *Typical languages:*
 - MySQL, regular expressions, CSS, yacc

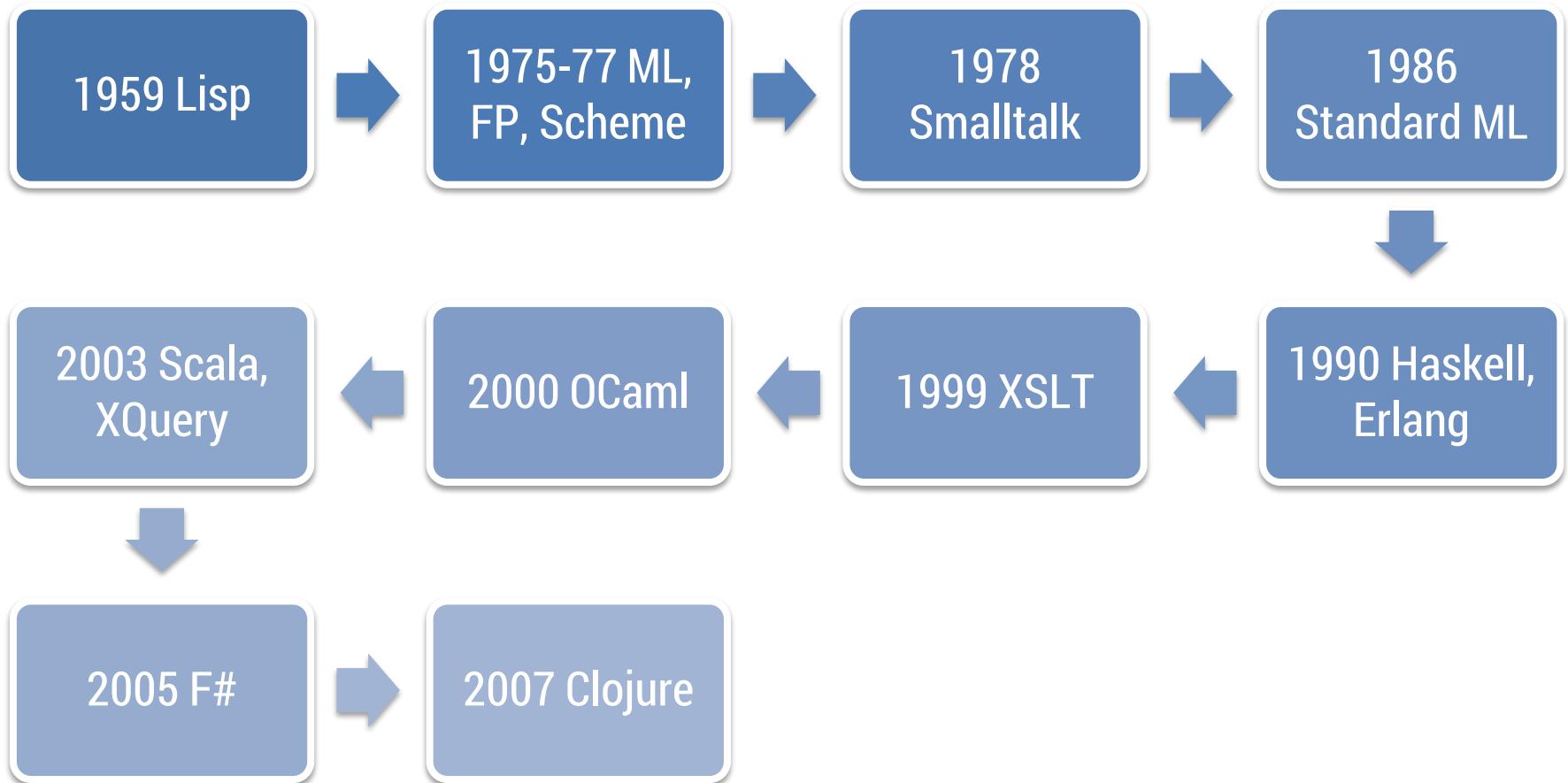
Functional programming

- Based on λ -calculus and reduction.
- Sub-expressions are replaced by simpler, but equivalent, sub-expressions until no longer possible
- Concepts:
 - No state, no commands: only expressions
 - Identifiers are values, not variables
 - No commands → no loops, just recursion
 - Functions : recursive, anonymous, ...
 - *... continued*

Functional programming (2)

- Concepts (*continued...*)
 - ▶ ...
 - ▶ Simple equational reasoning about programs
- *Typical languages:*
 - ▶ F#^{F# fonctional .NET}, Lisp, Haskell, ML, OCaml, Scala, Ruby

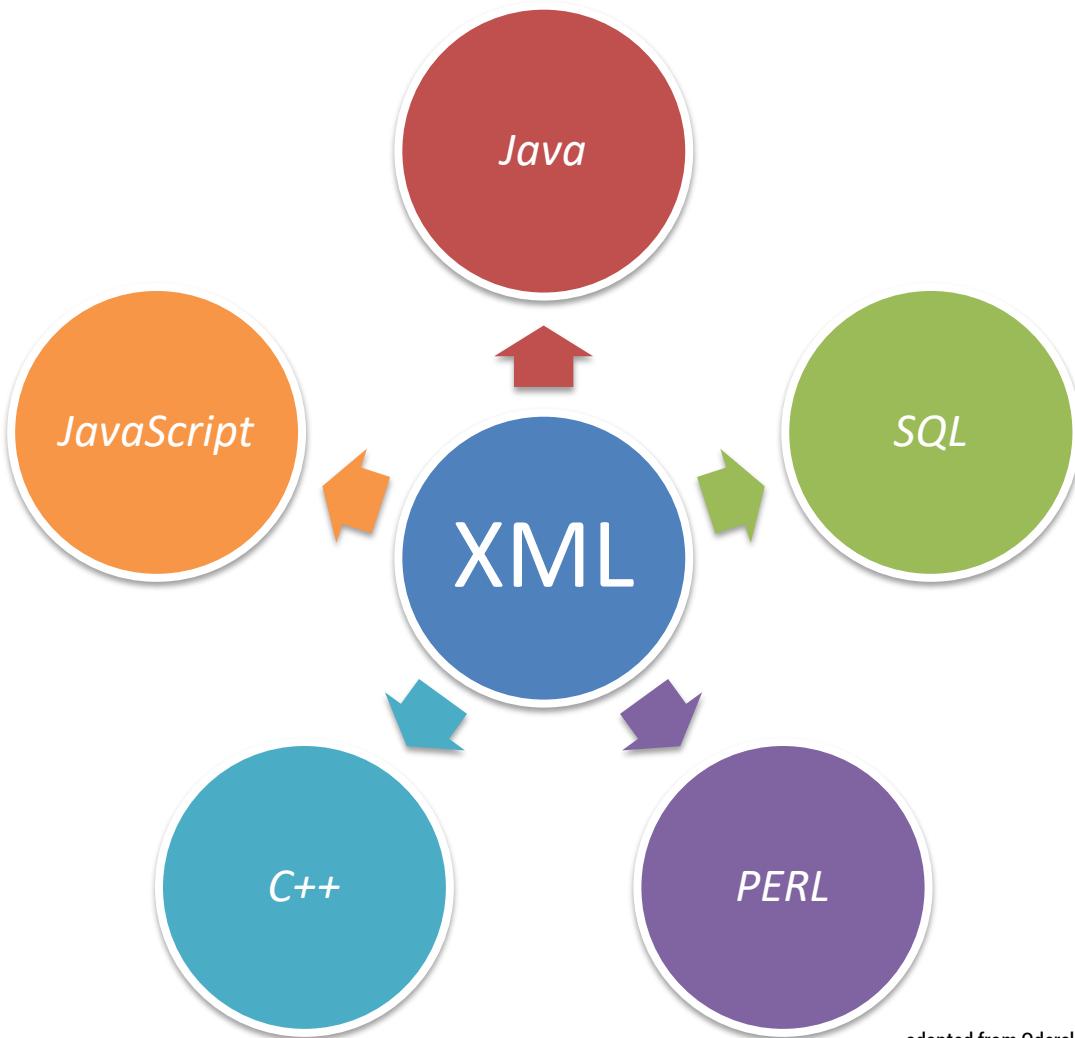
History of FP languages



Multi-paradigms

- Mix paradigms in a single language
- Examples:
 - ▶ Functional + imperative : *ML*
 - ▶ OO + functional features : *C#* features: lambdas par ex.
 - ▶ Functional + OO features : *F#, Ocaml*
 - ▶ Functional + OO : *Scala*
 - ▶ Functional + logic : *Curry* (based on *Haskell*)

Landscape of programming today...



... many
(little)
languages
playing
together

adapted from Odersky M. @ FOSDEM09

This is good and bad...

- *GOOD*
 - Every language do what it is best at
- *BAD*
 - cross language communication: fragile, complicated, ...
- *Problematic*
 - type system in communication ?
 - low level representation (XML, or worse, *string* (JDBC queries))

Our language of choice



- Characteristics
 - ▶ Multi-paradigm (FP + OO)
 - ▶ Narrow language (less keywords than Java)
 - ▶ Very extensible with libraries (deeper than other languages)

Origins



Martin Odersky

Swiss language



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Runs on JVM

Since 2003

Seamlessly interoperable with Java

Making things right

3. CORRECTNESS

How can we know a program is correct ?

testing —> find bugs

proving —> prove the absence of bugs

Correctness

- A program is only correct *relatively* to its *specification*.
 - ▶ Your code computes *sin* perfectly, however you need to compute *root* → not correct
- How can we know if correct?
 - ▶
 - ▶

Testing

Assumption

The product of every successive prime numbers until p , incremented by 1, is also a prime number.

- *Examples*
 - For $p = 2, 3, 5, 7, 11, \dots, 379$

Proving

- Theorem : $(a + b)^2 = a^2 + 2ab + b^2$
- Proof:
- With a *finite* number of steps, made something that works for *infinite* number of values !

Testing vs proving

- Proving much better than testing
- Requires to think in logics / math terms
- Programming languages close to math better suited for proving
- If we prove the program is correct → increases reliability

Theme of the third part of the course!

To make things clear

4. REFERENTIAL TRANSPARENCY

Motivation

```
globalValue = 0;  
  
int function rq(int x)  
begin  
    globalValue = globalValue + 1;  
    return x + globalValue;  
end
```

```
int function rt(int x)  
begin  
    return x + 1;  
end
```

Code sample adapted from Wikipedia «referential transparency page»

Let's see what we got

```
int p = rq(x) + rq(y) * (rq(x) - rq(x));
```

Definitions

`printf()` —> impure, effet de bord

`random(seed)` —> pure, toujours le même nombre

A function is **pure** iff:

1. The function *always evaluates* to the same result for the same input.
2. Evaluation of the function does not cause any semantically observable *side-effect* (or output).

Pure fonction

- The result cannot depend on any hidden information or state that may change (IO, external input...)



Pure or impure ?

pure: sin, length, encrypt, random(seed)

1. `sin(x)`
2. `currentTime()`
3. `printf()`
4. `length(s)`, with `s` a string
5. `encrypt(k, d)` with `k` a key and `d` the message
6. `random()`
7. `random(seed)`

Referential transparency (RT)

An expression is said **referentially transparent** if it can be replaced by its value without changing the behaviour of the program (the program produces the same output).

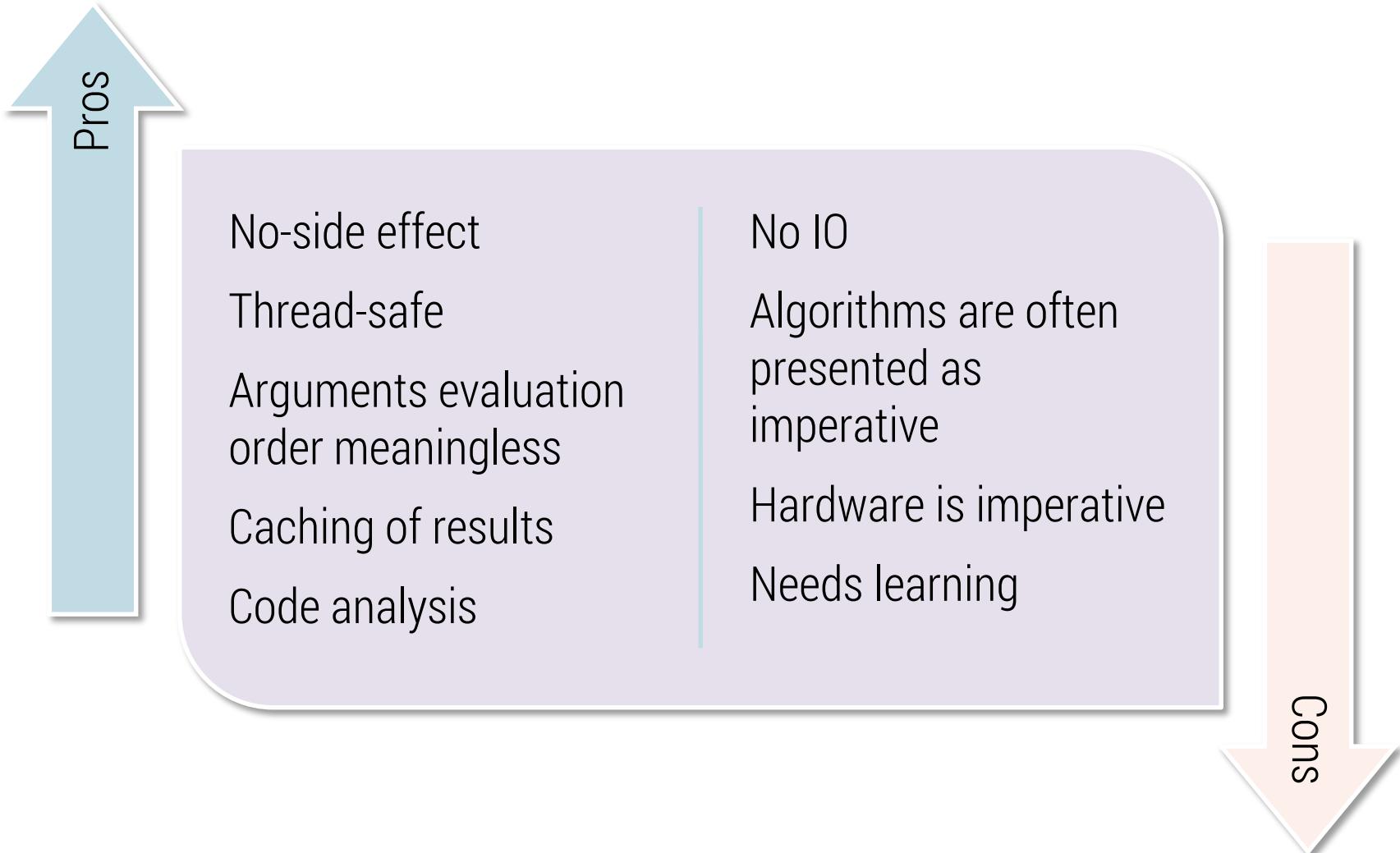
Referential transparency

- Pure functions are RT
- Math functions are RT

Why do we care?

- RT used for :
 - ▶ Program transformation (by compiler for example)
 - ▶ Proofs (by automated proof systems)
 - ▶ Evaluation (of expressions) (by runtime)
- FP is based on RT expressions and pure functions !

Pros / cons of FP, RT



Expression evaluation

5. EVALUATION STRATEGIES

Elements of programming

Each
programming
language
(non-trivial)
provides

- *Primitive expressions* for its simplest elements
- Ways to *combine* those expressions
- Ways to abstract expressions by introducing a *name* for expressions by which they can be referred to

Execution of a program

- In FP
 - program := declarations + expressions
- In FP
 - Execution means evaluation

In imperative programming

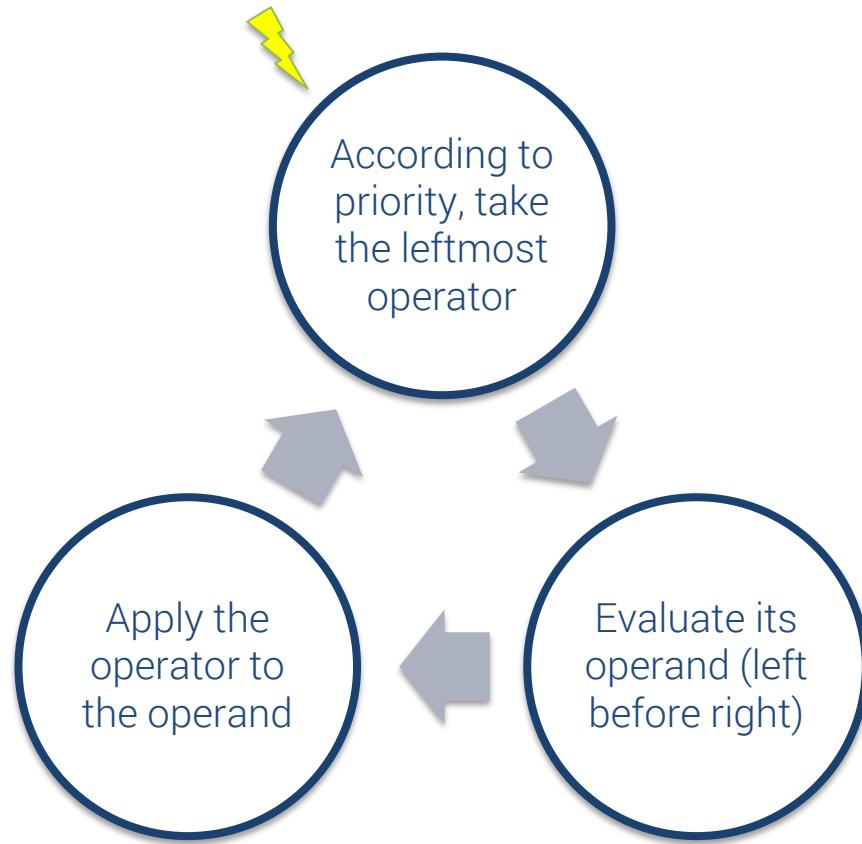
- Execution not only evaluation, e.g.

```
y:= 0; a:= 3  
function f(x)  
begin  
    y:=y+1; return x+y;  
end
```

$f(a) + f(a)$
returns $4 + 5 = 9$

- No RT, no arithmetic possible !
- Why ? Because command executions, such as **assignments**, change state.

How does evaluation work?



Rules:

- Name is evaluated by replacing it by its definition
- Evaluation stops when it results in a value
- A value is a number (for now)

Example

Declarations (definitions) in Scala:

```
def radius = 2; def pi = 3.1415
```

Expression:

```
(2*pi) * radius
```

Evaluation:

Evaluating expressions at the core

- Functional languages often have a REPL
 - *Read-Eval-Print-Loop*
 - Interactive shell that lets one write expressions and provide an answer back
- The Scala REPL can be started with
 - `scala` at the command prompt
 - within a *worksheet* in the IDE

Interactions with the REPL

```
scala> 12 + 56  
68
```

```
scala> def length = 3  
length: 3
```

```
scala> length * 5  
15
```

Definition of functions

- Definitions can have parameters, e.g.

```
scala> def sqr(x: Double) = x * x  
sqr: (x: Double)Double
```

```
scala> sqr(3)  
9.0
```

```
scala> sqr(3+4)  
49.0
```

```
scala> sqr(sqr(3))  
81.0
```

```
scala> def sumOfSqr(x: Double, y: Double) = sqr(x) + sqr(y)
```

Definition of functions

- Function parameters have a **type**
 - In Scala, types comes after the variable (and :)
- Return type (if given) after parameters, e.g.

```
def foo(x: Double, y: Int) : Boolean = ...
```

Evaluation of function applications

- Similar to operators
 1. Evaluate all arguments, left to right
(until we get a value)
 2. Replace the function by its definition
and **AT THE SAME TIME...**
 3. Replace the formal parameters of
the function by the actual
parameters

```
def sumOfSqr(x: Double, y: Double) = sqr(x) + sqr(y)
```

Example

```
scala> sumOfSqr(3, 1+4)
```

The substitution model

- This is the substitution model
- Works for every expression, as long as they are RT
- Formalized in λ - calculus

Call by-value evaluation

- In the previous example, we used call-by-value **evaluation**
 - ▶ Arguments are evaluated before being passed to the function

Call-by-name evaluation

- With call-by-value, parameters reduced before function application
- Other method: apply function to unreduced parameters → **call-by-name**

```
sumOfSqr (3, 2+4)
sqr (3) + sqr (2+4)
3*3 + sqr (2+4)
9 + sqr (2+4)
9 + (2+4) * (2+4)
9 + 6 * (2+4)
9 + 6 * 6
9 + 36
45
```

Call-by-value vs call-by-name

- Both yield the same values **iff**
 1. The reduced expression made of pure functions
 2. Both evaluations terminate
- CBV: function arguments evaluated **only once**
- CBN: function arguments **not evaluated if not used** in the function body, otherwise evaluated in each occurrence

Exercise

- Consider the following function

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

- Using CBV and CBN, how many steps are required to resolve:

- foo(2, 3)
- foo(3+4, 8)
- foo(7, 2*4)

CBV CBN

2,3: dans les deux cas, 3 étapes. (4)

3+4:

CBV, 4 étapes:

foo(3+4, 8), foo(7,8), 7,+7, 14

CBN, 5 étapes:

foo(3+4, 8), 7+(3+4), 7+7, 14

7, 2*4:

CBV: 4 steps

CBN: 3 steps

Termination

- The same values are produced by CBN and CBV iff both evaluations terminate... otherwise what ?
- Do expressions always terminate? No !!!

def loop: Int = loop (une valeur, mais au final on peut aussi dire que c'est une fonction, rajouter des () ne change rien)

scala => vs : fait du CBN

def first(x: Int, y: Int) = x

CBV ici, infinite loop

first(1, loop)

CBN pas ici!

first(1, loop)

So what does Scala do? CBV / CBN ?

- Both are possible
 - because both have advantages !
 - CBN, mostly for advanced techniques (DSLs)
- Default : CBV
- CBN if argument declared with =>

Example:

```
def first(x: Int, y : => Int) = x
```

CBV vs CBN

Predicates / boolean expressions

- Example:
 - $x < y$ is called a predicate of Boolean type
 - It is also called a boolean expression
- Boolean expression are made of
 - true and false constants
 - Comparison operators
 - Boolean operators

Conditional expression

- Choice between alternatives
 - ▶ **if** and **else** keywords
- Unlike Java's and C++'s versions
 - ▶ Returns a value
 - ▶ Similar to ... ? ... : ...

```
def foo = if(3 > 4) 3 else false
```

Exercise

1. Define && and || with if

```
def andand(x : Boolean, y : Boolean) = if(x) if(y) true else false else false  
def oror(x: Boolean, y: Boolean) = if(x) if(y) true else true else false
```

encore plus simple:

```
if(x) b else false  
if(x) true else b
```

Lecture's summary



- Parallelism is becoming more prevalent
- Programming for 100's of CPUs is difficult
- FP can help notably thanks to
 - Absence of state
 - Referential transparency
- Executing a FP program is evaluating expressions

Advanced programming paradigms

2. Higher-order functions

Disclaimer

This course is freely adapted from
Martin Odersky's course
"Programmation 4" given at EPFL



Solution of assignment 1

```
def square(x: Double) = x * x
def abs(x: Double) = if (x > 0) x else -x

def sqrtIter(approx: Double, x: Double): Double =
    if (isGoodEnough(approx, x)) approx
    else sqrtIter(improve(approx, x), x)

def improve(approx: Double, x: Double) =
    approx - (square(approx) - x) / (2 * approx)

def isGoodEnough(approx: Double, x: Double) =
    abs(square(approx) - x) < 0.0001

def sqrt(x: Double) = sqrtIter(1.0, x)

sqrt(81)
```

Imbricated / nested functions

- In FP, many helper functions are used.
 - only useful in a limited scope
 - we don't want other users to access those functions
 - we don't want to pollute the name space

→ **Nested functions**

Blocks in Scala (1)

- A block is delimited by braces {}
- Contains definitions or expressions
- Last element is an expression that defines the value of the block
- Block is an expression 

Blocks in Scala (2) - Visibility

- Visibility rules: as in Java, i.e. from their definition until the end of the block
- Definitions inside a block *shadow* (override) the definitions from outside

```
def foo = 3

def bar = {
    def foo = 5
    foo + 4
}
```

Refined solution (1)

```
def sqrt(x: Double) = {
    def sqrtIter(approx: Double, x:Double): Double =
        if (isGoodEnough(approx, x)) approx
        else sqrtIter(improve(approx, x), x)

    def improve(approx: Double, x:Double) =
        approx - (square(approx) - x) / (2 * approx)

    def isGoodEnough(approx: Double, x:Double) =
        abs(square(approx) - x) < 0.001

    sqrtIter(1.0, x)
}
```

- {...} delimit a block.
- Definitions in a block are only visible there
- Definitions in a block shadow external defs
- **The end of the block is an expression that define its value**

Refined solution (2), lexical scope

```
def sqrt(x: Double) = {
    def sqrtIter(approx: Double): Double =
        if (isGoodEnough(approx)) approx
        else sqrtIter(improve(approx))

    def improve(approx: Double) =
        approx - (square(approx) - x) / (2 * approx)

    def isGoodEnough(approx: Double) =
        abs(square(approx) - x) < 0.001

    sqrtIter(1.0)
}
```

- Names defined outside blocks are also visible inside, as long as they are not masked
- The definition of the parameter `x` can be removed from all nested functions, as `x` is visible everywhere inside the block

Lesson's objectives

Recursion, higher-order functions and currying

- ▶ Tail recursion
- ▶ Higher-order functions
- ▶ Currying and partial application

Improving recursion performance

1. TAIL RECURSION

Evaluation of function applications

Reminder

- Similar to operators
 1. Evaluate all arguments, left to right, until all arguments are values
 2. Replace the function by its definition and **SIMULTANEOUSLY...**
 3. Replace the formal parameters of the function by the actual parameters

Evaluating a function application

Formally

A function $f(e_1, \dots, e_n)$ is evaluated

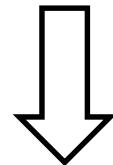
1. by evaluating the expressions $e_1 \dots e_n$ in the values $v_1 \dots v_n$
2. by replacing the application with the body of the function f in which
3. the actual parameters v_1, \dots, v_n replace the formal parameters of f

Substitution rule

def $f(x_1, \dots, x_n) = Body$

\dots

$f(v_1, \dots, v_n)$



def $f(x_1, \dots, x_n) = Body$

\dots

$[v_1/x_1, \dots, v_n/x_n] Body$

on remplace tous les $x\dots$ par les $v\dots$

Example, recursive function

```
def sum(x: Int): Int = if (x == 1) x else x + sum(x - 1)
```

```
sum(3)
-> if(3 == 1) 3 else 3 + sum(3-1)
-> 3 + sum(3-1)
-> 3 + sum(2)
-> 3 + (if(2 == 1) 2 else 2 + sum(2-1))
-> 3 + (2 + (sum(2-1)))
-> 3 + (2 + (sum(1)))
-> 3 + (2 + (if(1 == 1) 1 else 1 + sum(1-1)))
-> 3 + (2 + (1))
-> 3 + (3)
-> 6
```

Exercise : rewrite rule

```
def fact(x: Int) : Int = if(x == 0) 1 else x * fact(x-1)
```

fact(4)

```
if(4 == 0) 1 else 4 * fact(4-1)
4 * fact(4-1)
4 * fact(3)
4 * [if(3 == 0) 1 else 3 * fact(3-1)]
4 * [3 * fact(3-1)]
4 * [3 * fact(2)]
4 * [3 * [if(2 == 0) 1 else 2 * fact(2-1)]]
...
4 * [3 * [2 * [1 * [if(0 == 0) 1 else 0 * fact(0-1)]]]]
4 * [3 *[ 2 * [1 * [1]]]]
24
```

Another example, recursive as well

```
def tSum(x: Int, s: Int): Int = if (x == 0) s else tSum(x-1, s+x)
```

```
tSum(3, 0)
-> if (3==0) 0 else tSum(3-1, 0+3)
-> tSum(3-1, 0+3)
-> -> tSum(2, 3) double flèche: deux étapes en une
-> if (2==0) 3 else tSum(2-1, 3+2)
-> tSum(2-1, 3+2)
-> -> tSum(1, 5)
-> if (1==0) 5 else tSum(1-1, 5+1)
-> tSum(1-1, 5+1)
-> -> tSum(0, 6)
-> if (0 == 0) 6 else tSum(0-1, 6+0)
-> 6
```

vs sum: pas besoin de garder
quoi que ce soit sur la stack.
Au dernier appel récursif, on ne fait que remonter le résultat.

What is the difference ?

- First example
 - Perform recursive calls until the end of recursion, then return and calculate the result.
 - Result at the last recursive call
- 2nd example
 - Calculation **then** recursive call.
 - Pass the results to the next recursive step.
 - Return value is always of the same type



Tail recursion

- The second example is said to be tail-recursive
 - **Last action** is the recursive call
 - Optimized by the compiler to loops, iff they are private or final, or appear in another function
- @tailrec annotation to check if correctly done

Not clear?

```
recsum(5)           accumulateur vs tail recursion
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + 1)))
15
```

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
tailrecsum(0, 15)
15
```



Assignment 2

Exercise 1a, tail recursion

Defining values (1)

- The definition

`def f = expr`

- defines a name for the expression
`expr`
- `expr` will be **evaluated every time** it is used
- `f` is actually a function without parameters

Defining values (2)

- However

val v = expr

- defines a name for the **value** of the expression **expr**
- **expr** will be **evaluated once**
- **v** is not a variable !



Defining values

Having fun with functions

2. HIGHER-ORDER FUNCTIONS

First class citizens

Privileges of 1st class citizens

- To be passed as arguments to functions.
- To be named in a variable.
- To be returned as values of functions.
- To be incorporated into data structures.

Higher-order functions

A higher-order function is a function that takes functions as **argument** or that **returns** a function.

Higher-order functions

```
def square(x: Double) = x * x
def twice(x: Double) = 2.0 * x
```

```
def deriv(f: Double ⇒ Double, x: Double, dx: Double) = {
  (f(x + dx) - f(x)) / dx
}
```

$$f'(x_0) = \lim_{\substack{h \rightarrow 0 \\ h \neq 0}} t_{x_0}(h) = \lim_{\substack{h \rightarrow 0 \\ h \neq 0}} \frac{f(x_0 + h) - f(x_0)}{h}$$

```
val dx = 0.0000000001
deriv(square, 3, dx)
deriv(twice, 3, dx)
```

Storing functions in values

```
val incr = (x: Int) => x + 1
val times2 = (x: Int) => x * 2
val x = 3
val y = incr(x)
val z = times2(incr(x))
```

```
def square(x: Int) = x * x
// Invalid assignment !
val w = square
```



Functions with no names

- FP uses lots of (small) functions
- Sometimes, giving names to every function can be tedious
 - **anonymous functions**
 - like literal values for the functional type, i.e. like «foo» for the *String* type

```
(x : Double) => x * x
                  //> res0: Double => Double = <function1>
(x: Double, y : Int) => (x+y) * 2.0
                  //> res1: (Double, Int) => Double = <function2>
```

Remark

- Anonymous functions are **syntactic sugar**:

$$(x_1 : T_1, \dots, x_n : T_n \Rightarrow E)$$

\equiv

$$\{\mathbf{def} \ f(x_1 : T_1, \dots, x_n : T_n) = E; f\}$$

! Le compilateur doit faire attention à donner un nom unique à la fonction non nommée.
f doit donc être un “nom frais”

Sums with higher-order functions

$$\Sigma_a^b f(n)$$



Assignment 2

Exercise 2a

Storing functions in vals

3. PARTIAL FUNCTION APPLICATION & CURRYING

Partial application of function

en Scala, tout est object behind the hood, mais pas les définitions de fonctions.

- This is ok

```
val incr = (x: Int) => x + 1
```

- This is not:

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

 val w = square

- One should write

```
val w = square(_ : Int)
```

```
val w = square _
```

Partial application : usage

- Later binding of arguments

```
def adder(x: Int, y: Int) = x + y
  > adder: (x: Int, y: Int)Int
val incr = adder(_ : Int, 1)
  > incr  : Int => Int = <function1>
val add2 = adder(2, _ : Int)
  > add2  : Int => Int = <function1>

add2(5)
  > res0: Int = 7
incr(5)
  > res1: Int = 6
```

Motivation

- Some languages don't have multiple arguments functions (*Haskell* for instance)
- How to evaluate ?
 - ▶ `def mul(x: Int, y: Int) = x * y`
`def mul1(x: Int) = (y: Int) => x*y`
- Scala provides automatic way for this:
 - ▶ `def mul1(x: Int)(y: Int) => x * y`

Currying

- Turns a function with multiples argument into **a chain of functions**, each with a single argument.

def $f(args_1) \dots (args_n) = E$, with $n > 1$

def $f(args_1) \dots (args_{n-1}) = (\text{def } g(args_n) = E ; g)$

def $f(args_1) \dots (args_{n-1}) = (args_n \Rightarrow E)$

by iterating n times

def $f = (args_1 \Rightarrow (args_2 \Rightarrow \dots (args_n \Rightarrow E) \dots))$

Currying (2)

- Names come after *Haskell B. Curry* (1900-1982)
- Gave its name to three (!) programming languages:
 - *Haskell*
 - *Curry*
 - *Brooks*
- First citation of the currying idea : *Moses Schönfinkel*
 - schönfinkelisation ?



Source: Haskell programming language wiki, retrieved 24.2.2014

Example of currying

```
def addNormal(a: Int, b: Int) = a + b  
def addOneAtATime(a: Int) = (b: Int) => a + b
```

addNormal(3, 4)

addOneAtATime(3) returns a fn —> (b: Int) => 3 + b

addOneAtATime(3)(4)



```
def addCurry(a: Int)(b: Int) = a + b  
addCurry(3)(4)
```

```
val inc = addCurry(1) _  
inc(3)
```

Currying, in practice for us?

- ▶ 1-argument function calls possible with {}
- ▶ Curry → new control abstractions (that look like the language)

```
def repeat(it: => Int) (body: => Unit): Unit = {  
    if (it > 0) {    rappel:=> call by name  
        body  
        repeat(it - 1) (body)  
    } body: call by name oblig.  
    } ici, it pas oblig par name. Mais si on veut faire un while(condition), la condition doit  
absolument être by name, sinon elle est évaluée une fois et on se retrouve avec un  
while(true) ou while(false)  
    repeat(10) {  
        println("Hello")  
    }
```

More information during the DSLs lecture!

Currying vs partial application

Partial application

- Sets some values and reduces the arity of the base function
- Partial application evaluates immediately

a way of using functions

Currying

- Transform a n-ary function to n unary function calls
- Each call returns a function

a way of defining functions

Lecture's summary



- Functions are useful abstractions
 - General computation methods as explicit elements of the language
- Functions can be combined with higher-order function to create new abstractions
- Various language mechanisms help us: anonymous functions, partial application...

What have we done?

4. SYNTAX ELEMENTS SO FAR

EBNF grammar of elements seen so far

- Non contextual form as **EBNF** (extented Backus-Naur form)
 - More information there:
 - <http://www.garshol.priv.no/download/text/bnf.html#id2.3>.
 - <http://www.ics.uci.edu/~pattis/misc/ebnf.pdf>
 - Reminder
 - | denotes an alternative
 - [...] denotes an option (0 or 1)
 - {...} denotes a repetition (0 or more)
 - Capital letter means non terminal

Types

```
Type      = SimpleType | FunctionType
FunctionType = '(' [Types] ')' '=>' Type | SimpleType '=>' Type
SimpleType   = Byte | Short | Char | Int | Long | Double |
              Float | Boolean | String
Types        = Type {',' Type}
```

- A type can be:
 - a numerical: Int, Double, Byte, Short, Char, Long, Float
 - a boolean
 - a string
 - a function: Int => Int, (Double, Int) => Int
- More types later

Expressions

```
Expr      = InfixExpr | FunctionExpr | if '(' Expr ')' Expr else Expr
InfixExpr = PrefixExpr | InfixExpr Operator InfixExpr
Operator   = ident
PrefixExpr = ['+' | '-' | '!' | '~'] SimpleExpr
SimpleExpr = ident | literal | SimpleExpr '.' ident | Block
FunctionExpr = Bindings '=>' Expr
Bindings    = ident [':' SimpleType] | '(' [Binding {',' Binding}] ')'
Binding     = ident [':' Type]
Block       = '{{' {Def ';' } '}'}
```

- An expression can be:
 - An identifier : x, sqrt
 - A literal : 0, 0.1, "hello"
 - A function application: sqrt (x)
 - An operator application: -x, y+3
 - A selection: math.abs
 - A conditional expression : if (x >0) y else x
 - A block: {x * 2}
 - An anonymous function: (x => x + 1)

Definitions

```
Def      = FunDef | ValDef
FunDef   = def ident {'(' [Parameters] ')'} [':' Type] '=' Expr
ValDef   = val ident [':' Type] '=' Expr
Parameter = ident [':' ['=>'] Type
Parameters = Parameter {',', Parameter}
```

- A definition can be:
 - ▶ a function definition:
`def inc(x: Int) = x + 1`
 - ▶ a value definition:
`val y = inc(3)`

Advanced programming paradigms

3. Functional data structures

Disclaimer

This lecture is based on M. Odersky's
reference book "Scala by example",
chapter 6

<http://www.scala-lang.org/docu/files/ScalaByExample.pdf>

Lesson's objectives

Introduction to classes / multiparadigm programming

- ▶ Class parameters and constructors
- ▶ Methods and operators overloading
- ▶ Trees
- ▶ Classes as extensions
 - ▶ Defining sets of numbers

Rational numbers \mathbb{Q}

- Number expressed as a ratio n/d , where
$$n \in \mathbb{Z} \text{ and } d \in \mathbb{Z}^*$$
- Compared to floats, fractions are represented without rounding errors or approximations
- Today → a class for working with rational numbers in Scala

Operations on rational numbers

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1}{d_1} \div \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \Leftrightarrow n_1 d_2 = d_1 n_2$$

A first look at classes in Scala

1. CLASS PARAMETERS

1st attempt at representing rationals

- Feasible with functions only:

```
def sumNum(n1: Int, d1: Int, n2: Int, d2: Int) = {...}
```

```
def sumDenom(n1: Int, d1: Int, n2: Int, d2: Int) = {...}
```

- Not very practical
 - Have to manage every part of the rational number "manually"

Embedding in data structure

```
class Rational(n: Int, d: Int) {  
    def num = n  
    def denom = d          n et d sont des attributs privés de la classe (auto)  
}  
}
```

- Creates a new **type**, named Rational
- Gives a constructor for building rational numbers
- This type contains two members, num and denom
- Multiple others implementations possible (later...)

Classes and objects

- Classes are blueprints (or molds) for building **objects**
- Objects store data in a single abstraction
- Objects can be created with a specific operator: new

```
val r1 = new Rational(2, 3)
```

```
val r2 = new Rational(4, 3)
```

Objects members

- Objects of the class Rational possess the **members** num and denom
- Members can be selected with the '.' operator

```
val r1 = new Rational(2, 3)
r1.num          //> res0: Int = 2
r1.denom        //> res1: Int = 3
```

```
val r2 = new Rational(4, 3)
r2.num          //> res2: Int = 4
r2.denom        //> res3: Int = 3
```

First try at implementing rational arithmetics

Implementing arithmetics

```
def add(x: Rational, y: Rational) = {
    new Rational(x.num * y.denom + y.num * x.denom,
                 x.denom * y.denom)
}

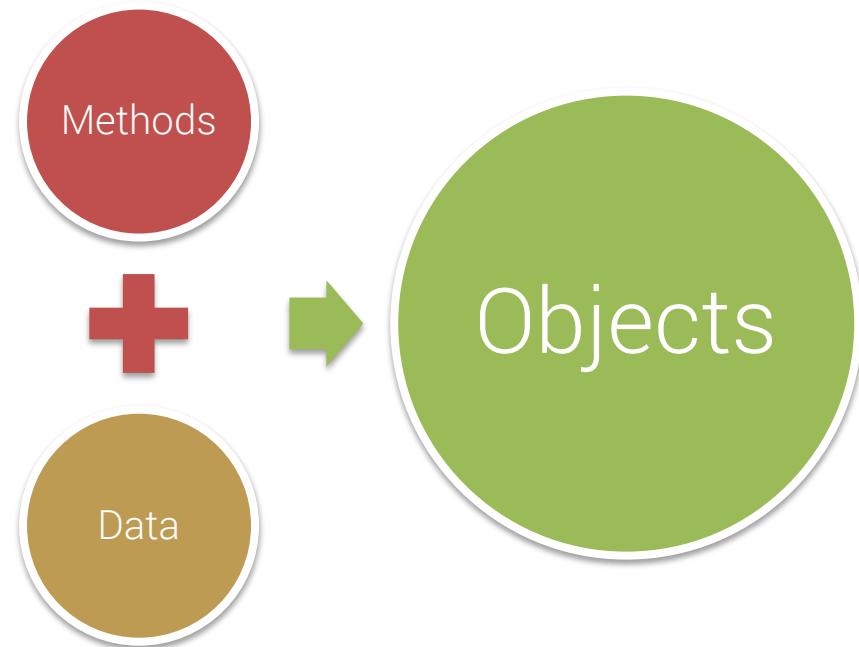
def stringVersion(x: Rational) = {
    x.num + "/" + x.denom
}

val r3 = add(r1, r2)
r3.num
r3.denom

stringVersion(r3)
```

Storing methods in classes

- Principle of OO languages, store methods where they belong
- Pack the functions with the class
 - such functions are called **methods**
- Example:
 - store numerator, denominator and the methods add minus times divided equal toString



Implementing methods

- Added in the class body
- Can be applied to the object
- Have access to the members of the class (**self-reference**)
 - ▶ this inside the class represents the object on which the method has been called
 - ▶ this can be omitted

2nd try at implementing rational arithmetics

Some remarks

pk utiliser def pour num/denum ?
permet de mettre une fonction à la place de num/denum si on le souhaite.
En outre, permet d'être surchargée dans une sous-classe.
Mais: appel de fonction un poil plus lourd.

```
class Rational(n: Int, d: Int) {  
  
    def num = n  
    def denom = d  
  
    def add(that: Rational) = {  
        new Rational(num * that.denom + that.num * denom,  
                     denom * that.denom)  
    }  
  
    def minus(that: Rational) = {  
        new Rational(num * that.denom - that.num * denom,  
                     denom * that.denom)  
    }  
  
    def times(that: Rational) = {  
        new Rational(num * that.num, denom * that.denom)  
    }  
  
    def divided(that: Rational) = {  
        new Rational(num * that.denom, denom * that.num)  
    }  
  
    def equal(that: Rational) = {  
        num * that.denom == denom * that.num  
    }  
  
    override def toString() = num + "/" + denom  
}
```

- override means "redefines an existing method"
- num and denom as def, could be val as well. Is that good?
- Example usage:

```
val r1 = new Rational(2, 3)  
       //> r1 : ... = 2/3  
val r2 = new Rational(1, 3)  
       //> r2 : ... = 1/3  
val r3 = r1.add(r2).times(r2)  
       //> r3 : ... = 9/27
```

Exercise 1

1. Implement the method neg
2. Implement the method less
3. Implement the method max,
using the less method

Building better data abstractions

- The result $9/27$ is not intuitive
 - ▶ Not simplest form
 - ▶ Why?
- How to implement this?
 - ▶ dans le constructeur
 - ▶

3rd iteration, simplification of rational numbers

$$gcd(a, b) = \begin{cases} a & \text{if } b = 0 \\ gcd(b, a \% b) & \text{otherwise} \end{cases}$$

euclide

Preconditions

- How to make sure the denominator is never 0 ?
- Use the predefined function `require`
 - `require(predicate)`
 - `require(predicate, message)`
 - if predicate is `false`, exception is thrown, including the message if present

```
class Rational(n: Int, d: Int) {  
    require(d != 0)  
    ...  
}
```

require() vs assert()

```
def square(x: Int) = x * x  
assert(square(2) == 4)
```

- assert (predicate) also checks that something must be true, otherwise exception is thrown
- Intent is different:
 - ▶ require for checking precondition on the caller
 - ▶ assert to check if a function is working as expected

Auxiliary constructors

- We have defined a way to create rational numbers with two parameters using a **constructor**
- How to add new (auxiliary) constructor(s)?

```
class Rational(a: Int, b: Int) {  
    def this(a: Int) = this(a, 1)  
    ...  
}
```

- Using the new constructor:

```
val r1 = new Rational(2)  
val r2 = new Rational(1, 2)  
val r3 = r2.times(r1)
```

Auxiliary constructors

Scala vs Java

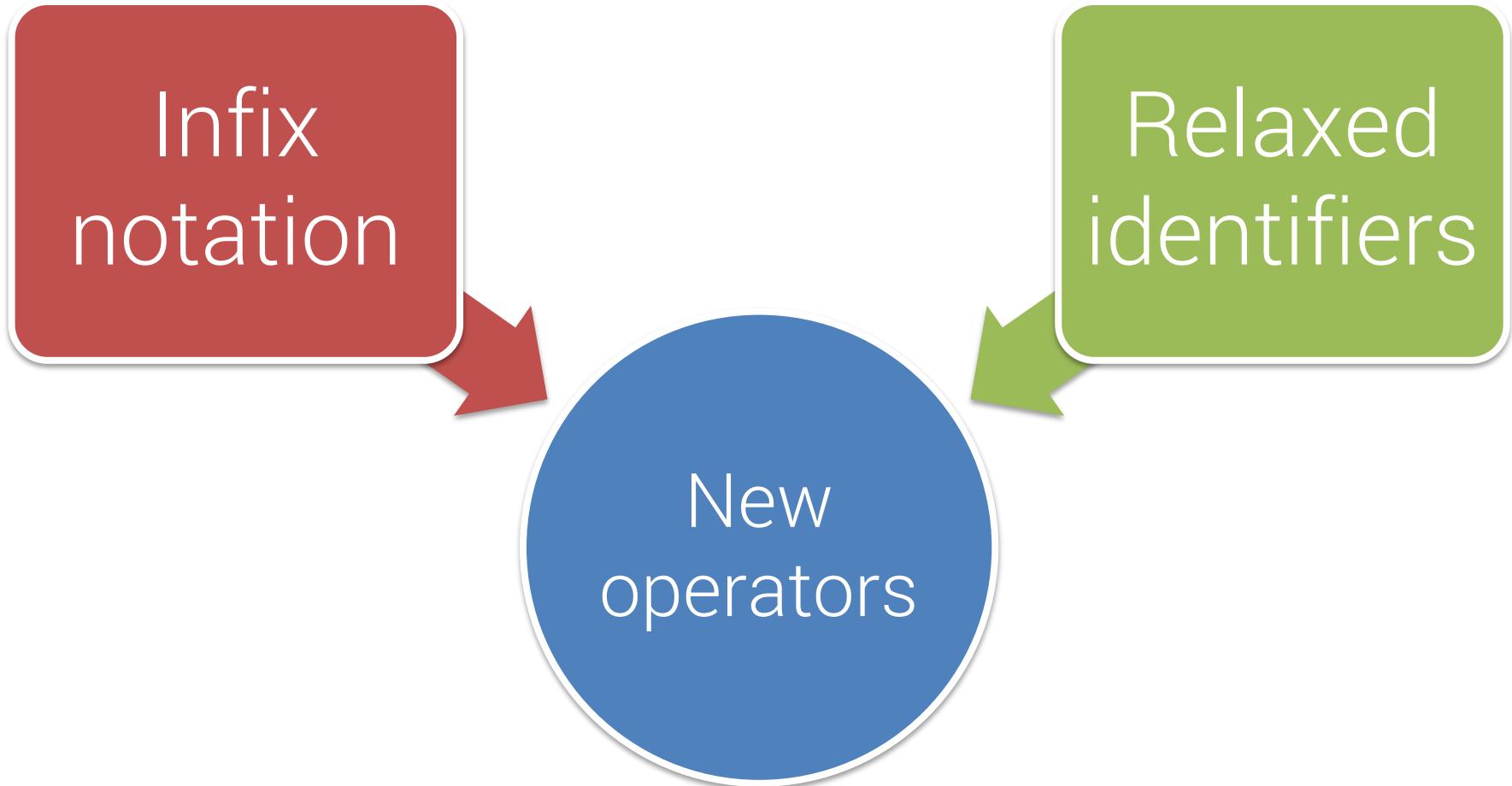
- In Scala
 - 1st action of auxiliary constructor must call primary constructor
 - only the primary constructor can invoke superclass constructor
 - brings conciseness, more details later
 - note that all the code in the class body which is not definitions is executed at construction time, once



Defining operators

- With integers and floating values
 $x + y$ yields a valid result
- Are Int or Float "special" classes?
 - not really...
 - backed up with Java int and float but otherwise normal classes
- Why then `x.add(y)`?
 - No reason why rational numbers must be different...

Implementing new operators



Implementing new operators

1. Infix notation

- Any method with a **single** parameter can be used as an **infixed operator**
- Principle: removes the `.` and the `()`
- *Example:* que pour les méthodes (vs functions, i.e. méthodes dans les méthodes)

$$\begin{aligned}x.\text{add}(y) &\equiv x \text{ add } y \\x.\text{less}(y) &\equiv x \text{ less } y\end{aligned}$$

RT (ref. transparency): can be replaced by its value without changing the behaviour of the program

- Should be used only for RT methods



Two steps required

2. Relaxed identifiers

- Operators can be used as identifiers
(not the case in Java)
- Identifiers can be:
 - **Alphanumeric**: start with a letter, followed by a sequence of letters or numbers
 - **Symbolic**: start with an operator symbol, followed by other operator symbols
 - _ is a letter

4th iteration, making Rational look like other numerical type

Method overloading

- Replace with standard operators name
- Unary operators must be prefixed with `unary_`
- Operators are actually method calls
 - ▶ $x + y = x.+(\textcolor{violet}{y})$
 - ▶
- What about precedence rules?



Precedence rules

- The precedence of an operator is determined by its first characters
- If same precedence, left associativity (some exceptions, `::` for instance)

Precedence rule table
All letters
I
^
&
< >
= !
:
+ -
* / %
Every other special characters

Lowest priority
Highest priority

Exercise 2

1. Parenthesize the following expressions

a + b ^? c ?^ d less a ==> b | c

((a + b) ^? (c ?^ d)) less ((a ==> b) | c)

$r * 2$ 

What about $2 * r$?

Implicit conversions

- Make a conversion from Int to Rational automatically
- New keyword: implicit
- Example:

```
implicit def intToRat(x: Int) = new Rational(x)
```

- Where to implement this? Must be in scope... *object companion*, more explanations later!

A word of caution

depuis 2.11, on doit dire explicitement qu'on veut les implicites.

- All this helps you creating rich libraries
 - Easy to use
 - Concise
- Operators must make sense, be obvious
- Implicits are not explicitly written (!)
- Must remain readable, not always easy...

Putting data in trees

TREES AS AN ABSTRACT DATA STRUCTURE

Tree as an ADS

Binary search trees (BST)

A BST is a tree where the value of a node is bigger than the values of the nodes in its left sub-tree and smaller than the values in its right sub-tree. In addition, each node in a BST can have at most 2 children (1 left, 1 right).

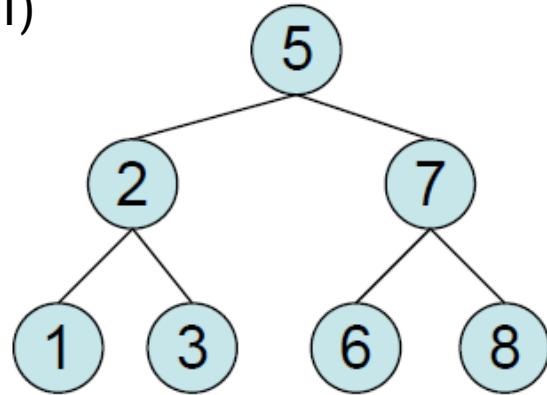
Binary search trees

- Sub-category of trees
 - 2 children per node
 - Always sorted!

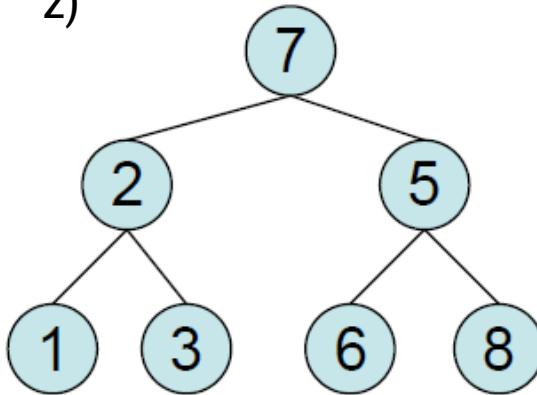
Exercise : which are BSTs ?



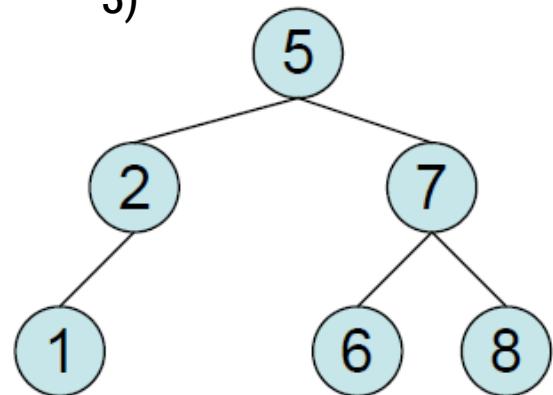
1)



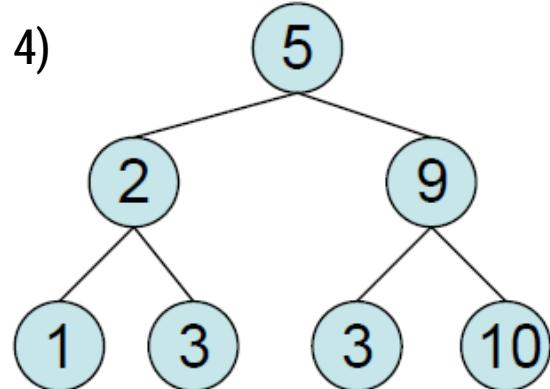
2)



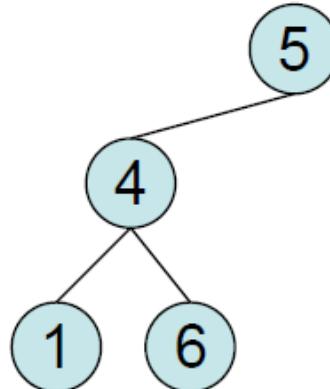
3)



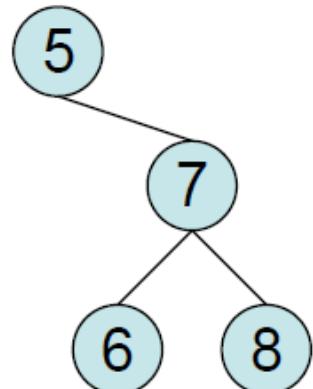
4)



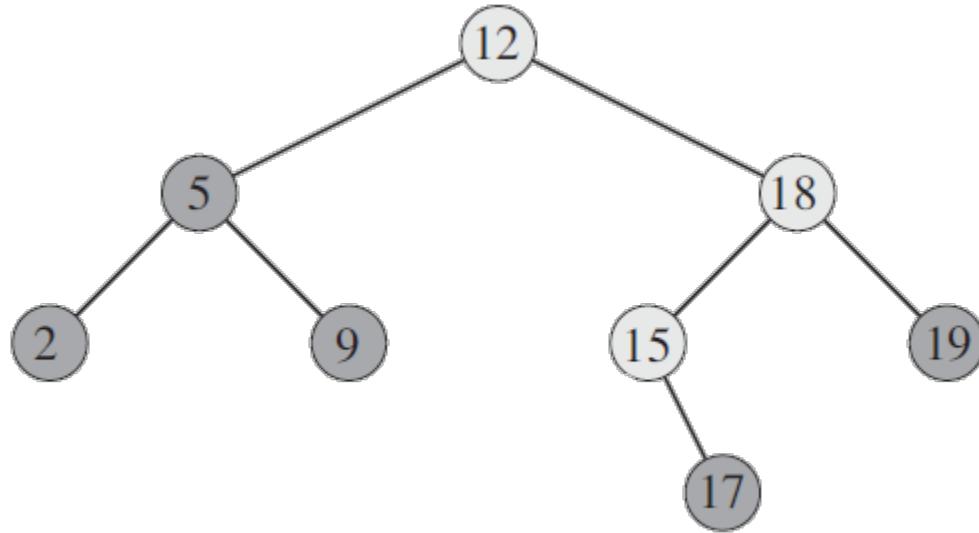
5)



6)



Node insertion



More OOP

2. ABSTRACT CLASSES & EXTENSIONS

Abstract classes

- We want to write a class for sets of integers. Let's call it IntSet
- An IntSet should possess two operations:

‣ **def** add(x: Int) : IntSet
‣ **def** contains(x: Int) : Boolean



We don't have state !

An abstract set

```
abstract class IntSet() {  
    def add(x: Int): IntSet  
    def contains(x: Int): Boolean  
}
```

- No implementation of some (or all) members (here add and contains)
- Abstract class means that no instance of this class can be built with the operator new

Extending classes with inheritance

- Let's say we want the set as a binary sorted trees (BST)
- Two possible tree types
 - A tree for the empty set
 - A tree consisting in an element and two sub-trees

Live-coding: first steps of set implementation

Code given in assignment 3

OOP terms

Text

Implementation and overriding

possible to redefine a defined method in a subclass with override

override mandatory if method already defined (vs abstract)

Object/singleton definitions

- For IntSet, there should be only one single Empty
 - overkill to have many instances
 - single instance at anytime = **singleton object**
 - replace class by object
- Effects:
 - no instance can be created
 - there exists only one instance
 - singleton pattern for free

```
object IntSetApp extends App { ... }
```

Lecture's summary



- A class defines a type and a function to create objects
- Objects have members that can be accessed with the . operator
- Classes can be extended
- Classes can be abstract, i.e. without implementation
- If A **extends** B then A **conforms** to B, i.e. type A objects can be used where B objects are required

A function can be invoked with a list of arguments to produce a result. A function has a parameter list, a body, and a result type.

Functions that are members of a class, trait, or singleton object are called methods.

Functions defined inside other functions are called local functions.

Functions with the result type of Unit are called procedures.

Anonymous functions in source code are called function literals. At run time, function literals are instantiated into objects called function values.

Advanced programming paradigms

4. Pattern matching and lists

Cheatsheet from OOP (last week)

```
class Sample{  
    private final int x;  
  
    Sample(int x) {  
        this.x = x;  
    }  
  
    int foo(int y) {  
        return x + y;  
    }  
  
    static int staticF(int x, int y) {  
        return x * y;  
    }  
}
```

```
class Sample(x: Int) {  
    def foo(y: Int) = x+y  
}  
  
object Sample{  
    def staticF(x: Int, y: Int) = x*y  
}
```



Lesson's objectives

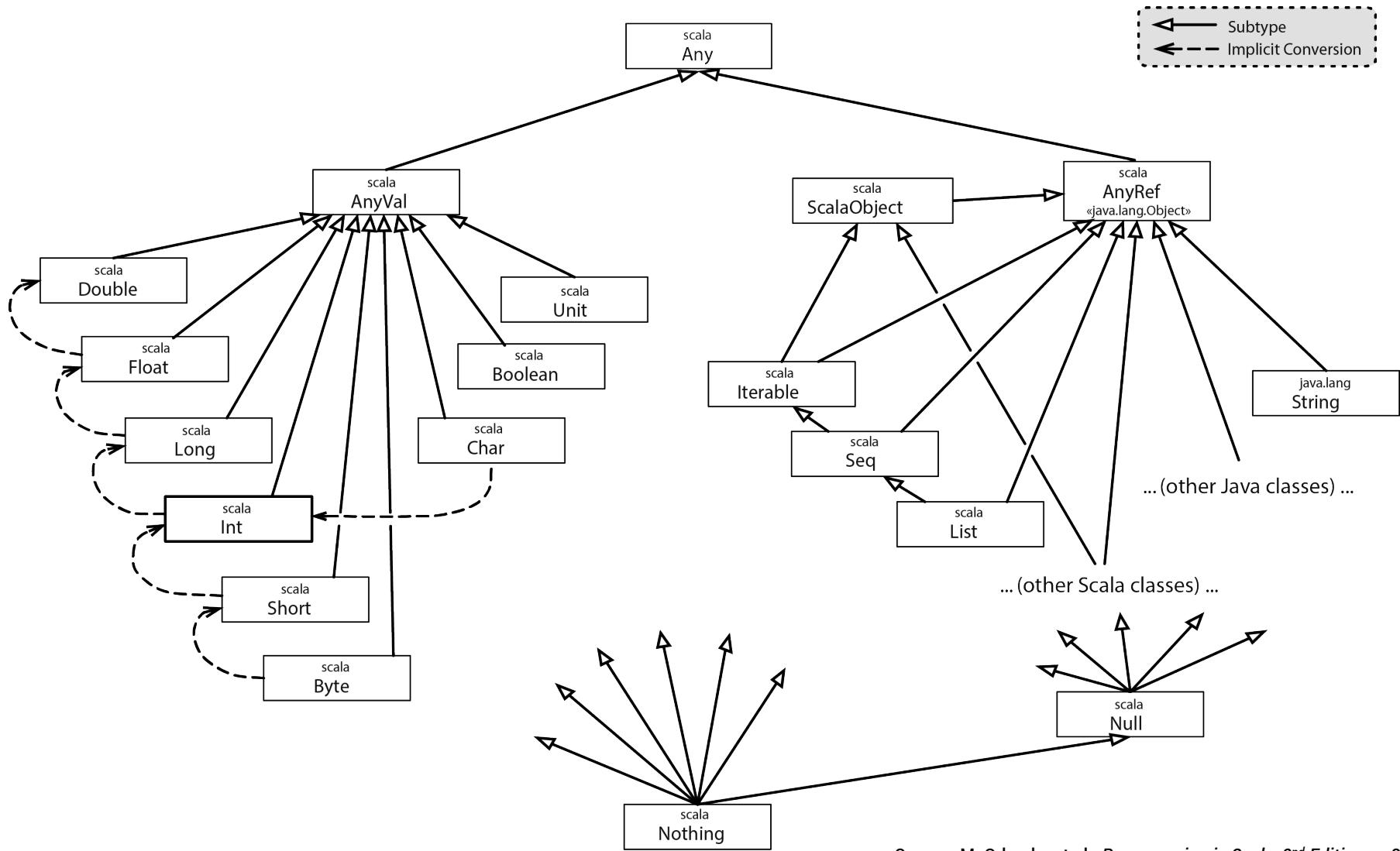
Understand pattern matching and discover the power of List

- ▶ Scala types
- ▶ Pattern matching
 - ▶ Different types of matching
 - ▶ Case classes
- ▶ List in FP

Diving into the depths of the language

1. SCALA TYPES

Scala type hierarchy



Source: M. Odersky et al., *Programming in Scala, 2nd Edition*, p. 252

Top types : Any, AnyRef and AnyVal

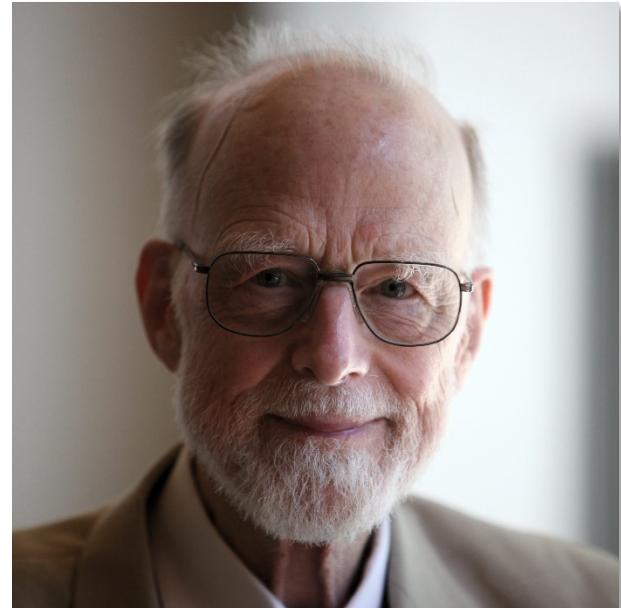
- Any base for every other type. Methods == and !=, equals, ## (hashcode), isInstanceOf, asInstanceOf
- AnyRef alias for java.lang.Object: eq/ne (object equals)
- AnyVal

The Null type

- Corresponds to Java null
 - Is a type
 - Rarely used in Scala
- Subtype of every reference type
 -
 -
 -

A bit of history, the invention of null

- Sir Tony Hoare
 - Maker of quicksort
 - Hoare logic (verification part of the course)
- Was developing type system for an OO language:



Source: Wikipedia page on Tony Hoare, retrieved on 9.3.2014

"[...]But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

T. Hoare, "[Null References: The Billion Dollar Mistake](#)"
March 2009, QCon London.

Note: some useful Scala functions

- In `scala.Predef` companion object
 - `assert`, `require`
 - `println` (and variants `printf`, `print...`)
 - `readInt` (and variants `readLine...`)
 - `sys.error(message: String)`
 - `sys.exit()`
 - `???`, for marking non implemented functions

Switch / case on "steroids"

2. PATTERN MATCHING

Motivation for pattern matching

- Let's say we want to design an interpreter for computing simple arithmetic expressions
 - Numbers and sum
- Model
 - numbers are leaves in the tree

Solution 0

```
abstract class Expr {  
    def isNumber: Boolean  
    def isSum: Boolean  
  
    def numValue: Int  
    def leftOp: Expr  
    def rightOp: Expr  
}  
  
class Number(n: Int) extends Expr {  
    def isNumber = true  
    def isSum = false  
  
    def numValue: Int = n  
    def leftOp: Expr = error("Number, not a leftOp")  
    def rightOp: Expr = error("Number, not a rightOp")  
}  
  
class Sum(e1: Expr, e2: Expr) extends Expr {  
    def isNumber = false  
    def isSum = true  
  
    def numValue: Int = error("Calling num on a sum")  
    def leftOp: Expr = e1  
    def rightOp: Expr = e2  
}
```

Is that good?

- Evaluation of expressions:

```
def eval(e: Expr): Int = {  
    if (e.isNumber) e.numValue  
    else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)  
    else error("Problem with expression" + e)  
}
```

- Problem?



- How can we solve this?

Solution 1

Using OOP

```
abstract class Expr {  
    def eval: Int  
}  
  
class Number(n: Int) extends Expr {  
    def eval = n  
}  
  
class Sum(e1: Expr, e2: Expr) extends Expr {  
    def eval(): Int = e1.eval + e2.eval  
}
```

- Good, but what about adding new methods (for instance displaying expressions)?
 - Add `def show(): String` and add body in every sub-class... okay

Can we solve everything?

- Not really! Let's say we want to simplify expressions such as:

$$a \times b + a \times c \rightarrow a \times (b + c)$$

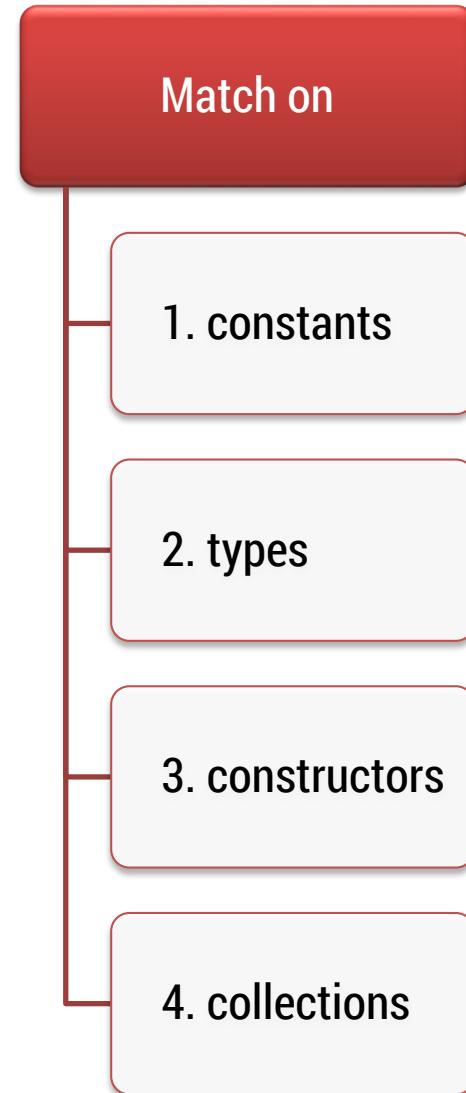
- Problem:
- Access methods for the sub-classes... again!

Decomposition with pattern matching

- Access and test functions are there to reverse the constructor mechanism
 1. Which sub-class was really used
 2. What were the arguments of the constructors
- Very common in programming → **pattern matching**

Pattern matching

- Generalization of switch case
- Java: switch limited to int values and String (since Java 7)
- New keywords:
 - ▶ match / case
- If no match, exception is thrown



1. Matching on constants

```
def foo(x: Int) = {  
    x match {  
        case 1 => "one"  
        case 2 => "two"  
        case 3 | 4 => "many"  
        case _ => "other value"  
    }  
}
```

```
def bar(x: Any) = {  
    x match {  
        case 1 => "one"  
        case 5.0 => "5 double"  
        case "two" => 2  
        case 'c' => "Letter c"  
        case _ => "something else"  
    }  
}
```

2. Matching on types

```
def typeMatch(x: Any) = {
  x match {
    case a: Int => "Got an int, " + a
    case b: String => "The string " + b
    case _ => "Don't know what it was"
  }
}

def guardMatch(x: Any) = {
  x match {
    case a: Int if (a % 2 != 0) => "Odd int, " + a
    case b: String if (b.length > 4) => b + " is long"
    case c: String if (c.length <= 4) => c + " is short"
    case _ => "Don't know what it was"
  }
}
```

Exercise 1

- Implement a function `patFoo` that returns `true` if the parameter is
 - an integer which can be divided by 4 without rest
 - an uppercase letter
 - a boolean value
- Otherwise, it should return `false`

Case classes (1)

- A case class is like a normal class but its definition starts with the modifier **case**, i.e.

```
case class Number(n: Int) extends Expr  
case class Sum(e1: Expr, e2: Expr) extends Expr
```

- This defines two subclasses
- Base for *algebraic* data types

<http://merrigrove.blogspot.ch/2011/12/another-introduction-to-algebraic-data.html>

Case classes (2)

1. Short constructor

- `no new` (`new` becomes optional)
- `apply()` method in companion object

2. Arguments as public fields

- `implicit val` in front of parameters

3. Creates methods for you

- `equals` the proper one, the one which compares values vs references !
- `hashCode`
- **handy copy method**
- **nice `toString`** using the constructor. Example: `Number(5)` vs `Number#e5eb3`

```
case class Foo(x: String)
val a = Foo("Hello")
val b = Foo("Hello")
a.x
a == b
a.hashCode
b.hashCode
```

3. Matching on constructors

Case class pattern

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

- Extract values from **constructor** directly
- Here, extracts the type of the Expr

Case classes – sealing a class

```
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2) // If absent compiler warns
}
```

- Compiler warns if match is not exhaustive
- Subclasses must be in same file
- Can't add new subclasses elsewhere *i.e. in other files*



Adding the show method

Pattern matching and methods

```
sealed abstract class Expr {  
    def eval: Int  
}  
  
case class Number(n: Int) extends Expr {  
    override def eval = n  
}  
  
case class Sum(e1: Expr, e2: Expr) extends Expr {  
    override def eval = e1.eval + e2.eval  
}
```

“expression problem”:

si on ajoute souvent des sous-classes -> mieux dans les enfants
si on ajoute souvent des méthodes -> mieux dans la superclasse

```
sealed abstract class Expr {  
    def eval: Int = this match {  
        case Number(n) ⇒ n  
        case Sum(e1, e2) ⇒ e1.eval + e2.eval  
    }  
}  
  
case class Number(n: Int) extends Expr  
case class Sum(e1: Expr, e2: Expr) extends Expr
```

- Eval function was outside the classes...
- Perfectly fine to define eval in the super class or subclass!
- Which solution is best?

Let's meet the star



3. LISTS IN FP

The importance of lists in FP

- Plays a key role
 - Speed vs arrays for insertion
 - Recursive data type
- Typical operations on lists:
 -
- Remember: RT means we have immutable lists
 - Implications?

Validity of our approach

- We use  **Scala**
- Matching and lists are key in every functional language
 - ▶ *Haskell, F#, Scheme...*
- *Python* ?
 - ▶ No pattern matching, no tail recursion, no extensive list functions library...

The List type

- Lists are homogeneous, meaning that all the elements have the same type.

Examples:

```
val names : List[String] = List("Roger", "Ana", "Paul")
val nums : List[Int] = List(1, 2, 3, 4)
val diag3 : List[List[Int]] = List(List(1, 0, 0),
                                    List(0, 1, 0),
                                    List(0, 0, 1))
```

List constructors

- Every list is built from:
 - ▶ empty list Nil
 - ▶ operator :: (cons) := right associative
- For instance:
 - ▶ `val fruit = "apples"::("kiwi)::Nil`
 - ▶ `var nums = 1::(2::(3::(4::Nil)))` ou simplement `1::2::3::4::Nil`
- Notes
 - ▶ Every Scala operator ending with : is right associative
 - ▶ `4 :: Nil` is actually `Nil.::(4)`
 - ▶ `::` is contained in the `List` class
 - ▶ `A::B::C::Nil` is `A::(B::(C::Nil))`

Equivalent list builders

- `1 :: 2 :: 3 :: 4 :: Nil` builds a list equivalent to `List(1, 2, 3, 4)`

Basic List operations

- Every list operation can be expressed with three operations:
 - ▶ head
 - ▶ tail
 - ▶ isEmpty
- Defined as methods from the List type

```
val nums = 1::2::3::4::Nil
```

Example: sorting a list

- To sort List (7, 3, 9, 2)
 1. sort List (3, 9, 2) to obtain List (2, 3, 9)
 2. insert 7 at the correct position
- The idea of insertion sort:

```
def isort(xs: List[Int]): List[Int] =  
  if (xs.isEmpty) Nil  
  else insert(xs.head, isort(xs.tail))
```

Exercise 2

- Implement the `insert` function, using recursion.

Pattern matching with lists

- Nil are is a case class
- `p :: ps` a pattern that matches a list with a head matching `p` and a tail matching `ps`
- The `insert` function from previous exercise can also be done using pattern matching!

Live coding: implementing insert with pattern matching

A simple list function: length

```
def lengthInt(l: List[Int]): Int = {
  l match {
    case Nil => 0
    case x :: xs => lengthInt(xs) + 1
  }
}

def lengthString(l: List[String]): Int = {
  l match {
    case Nil => 0
    case x :: xs => lengthString(xs) + 1
  }
}
```

Genericity

type erasure: pas de types à l'exécution !!

- **Genericity** makes the type as a parameter
- Is done at **compile time**
 - Scala is statically typed
 - Genericity is not available at run-time (*type erasure*)
- Functions (and classes) can be tailored to different types
- Notation : [T] as parameter

pk List[T] et pas List[Any] ? on le verra plus tard

Example:

```
def length[T](l: List[T]): Int = {
    l match {
        case Nil => 0
        case x :: xs => length(xs) + 1
    }
}
```

Concatenation of Lists

- How to append two lists?
 - ▶ the `++` operator (legacy `:::`)
- `List(1, 2, 3) ++ List(4, 5, 6)`
 - ▶
- `++` might be written with basic operations, how?
 - ▶ You'll find out in today's assignment...

Other list functions

- Very useful functions on lists
 - `length`, returns number of elements in the lists
 - `last`, returns the last element
 - `init`, returns all the elements but the last
 - `reverse`
 - `take (n)`, returns the first n elements
 - `drop (n)`, returns everything but the first n elements
 - `apply (n)`, returns the nth element
- Will be implemented in assignment 4 (but already exist in Scala ☺)

Lecture's summary



- Pattern matching reverses the construction process
 - Very practical way of doing complex things
- Lists play a major role in FP
- Working with lists helps understanding FP
- Next week: better functions with lists!

Advanced programming paradigms

5. Advanced lists and for comprehensions

Disclaimer

The functions definitions are taken from the online Scala API

<http://www.scala-lang.org/api/2.11.8/>

The tuple part comes from M. Odersky course, *Functional programming in Scala*, available on Coursera, slides 5-3

Lesson's objectives

Higher-order functions on lists and for comprehensions

- ▶ 1st lecture
 - ▶ Higher-order functions on lists
- ▶ 2nd lecture
 - ▶ Tuples
 - ▶ *For* comprehensions

Thinking with lists

1. HIGHER-ORDER FUNCTION ON LISTS

Lists so far

- ▶ `length`, returns number of elements in the lists
- ▶ `last`, returns the last element
- ▶ `init`, returns all the elements but the last
- ▶ `reverse`
- ▶ `take (n)`, returns the first n elements
- ▶ `drop (n)`, returns everything but the first n elements
- ▶ `apply (n)`, returns the n^{th} element

Creating lists, from scratch

Using the companion object

- Scala API is rich, very rich
- Companion object methods:
 - **def** empty[A] : List[A]
 - **def** fill[A] (n: Int) (elem : => A) : List[A]
 - **def** range[A] (start: A, end: A) : List[A]
 - ...
- Using Range
 - 1 to 5 → Range(1, 2, 3, 4, 5)
 - 0 until 5 → Range(0, 1, 2, 3, 4)
 - 1 to 7 by 2 → Range(1, 3, 5, 7)

Other useful functions

- indexOf
- contains
- distinct
- sum
- union, intersect, diff
- mkString
- Plenty of others, see the official Scala API

<http://www.scala-lang.org/api/current/#scala.collection.immutable.List>

Higher-order functions (reminder)

A higher-order function is a function that takes functions as **argument** or that **returns** a function.

Higher-order functions

- Applicable on lists?



Higher-order functions on lists

The most used ones

map

filter

reduceLeft, reduceRight

foldLeft, foldRight

flatMap

zip

The map function

```
def map[B](f: (A) => B): List[B]
```

- Builds a new collection by applying a function to all elements of this list.

Examples:

```
val l = List(1, 2, 3, 4, 5)
l map ((x: Int) => x * 2)
l map ((x) => x * 2)
l map (x => x * 2)
l map (_ * 2)

l map ((x) => x * 2.0)
l map ((x: Int) => x.toString)
l map ((x: Int) => if (x % 2 == 0) true else false)
```

The filter function

```
def filter(p: (A) => Boolean): List[A]
```

- Selects all elements of this list which satisfy the predicate.

Examples:

The `reduceLeft` function (1)

```
def reduceLeft[B >: A](f: (B, A) => B): B
```

- Applies a binary operator to all elements, going left to right.
- Returns the result of inserting `op` between consecutive elements of this list, left to right :

$$\begin{aligned} \text{List}(x_1, \dots, x_n).reduceLeft(op) \\ \equiv \\ op(op(\dots op(x_1, x_2) \dots, x_{n-1}), x_n) \end{aligned}$$

where x_1, \dots, x_n are the elements of the list

The reduceLeft function (2)

Examples:

reduceLeft demo

The reduceRight function

```
def reduceRight [B >: A] (f: (A, B) => B) : B
```

- Goes from right to left
- Accumulates the result in the second argument, not the first

Examples:

$$op(x_1, op(x_2, \dots, op(x_{n-1}, x_n) \dots))$$

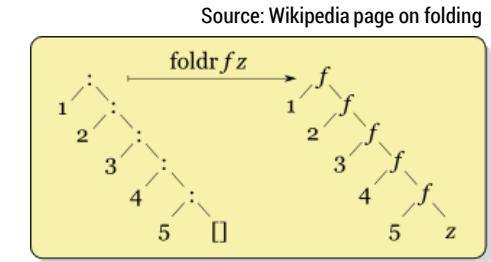
where x_1, \dots, x_n are the elements of the list

The foldRight function

```
def foldRight[B](z: B)(f: (A, B) => B): B
```

- Applies a binary operator to all elements, going right to left, starting with an initial value
- Very powerful function

Examples:



The foldLeft function

```
def foldLeft [B] (z: B) (f: (B, A) => B) : B
```

- Applies a binary operator to all elements, going left to right, starting with an initial value
- Parameters order for accum.



Examples:

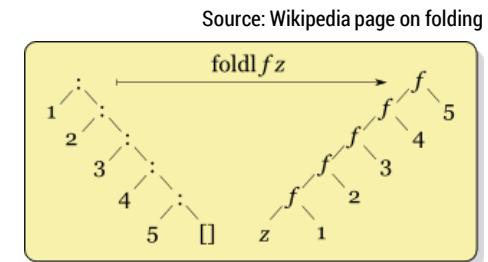
attention:

list.foldRight(0)(_-_-)

list.foldLeft(0)(_-_-)

ne donnent pas la même chose.

avec foldRight, on a (a,b) => b, donc on va faire 5 - 0 la première fois, puis 4 - 5, etc.



Exercise 1 : Compute

1. `List(1, 2, 3).foldRight(2)((x, y) => x - y)`

```
3 - 2 = 1  
2 - 1 = 1  
1 - 1 = 0  
=> 0
```

2. `List("a", "b", "c").foldLeft("0")((a, b) => a + ", " + b)`

“0, a, b, c”

3. `List(1, 2, 3).foldLeft(List(4))((list, e) => e :: list)`

3 :: 2 :: 1 :: 4

foldLeft vs foldRight

- Results of those methods on the same list are equal iff the binary operator is
 - associative
 - commutative

Example:

Consider the List (1, 2, 3, 4) with seed 0

Associative	Commutative	Op	foldR	foldL
No	No	$f(x, y) \Rightarrow x - y$		
No	Yes	$f(x, y) \Rightarrow (x+y) / 2$		
Yes	No	$f(x, y) \Rightarrow x$		
Yes	Yes	$f(x, y) \Rightarrow x + y$		

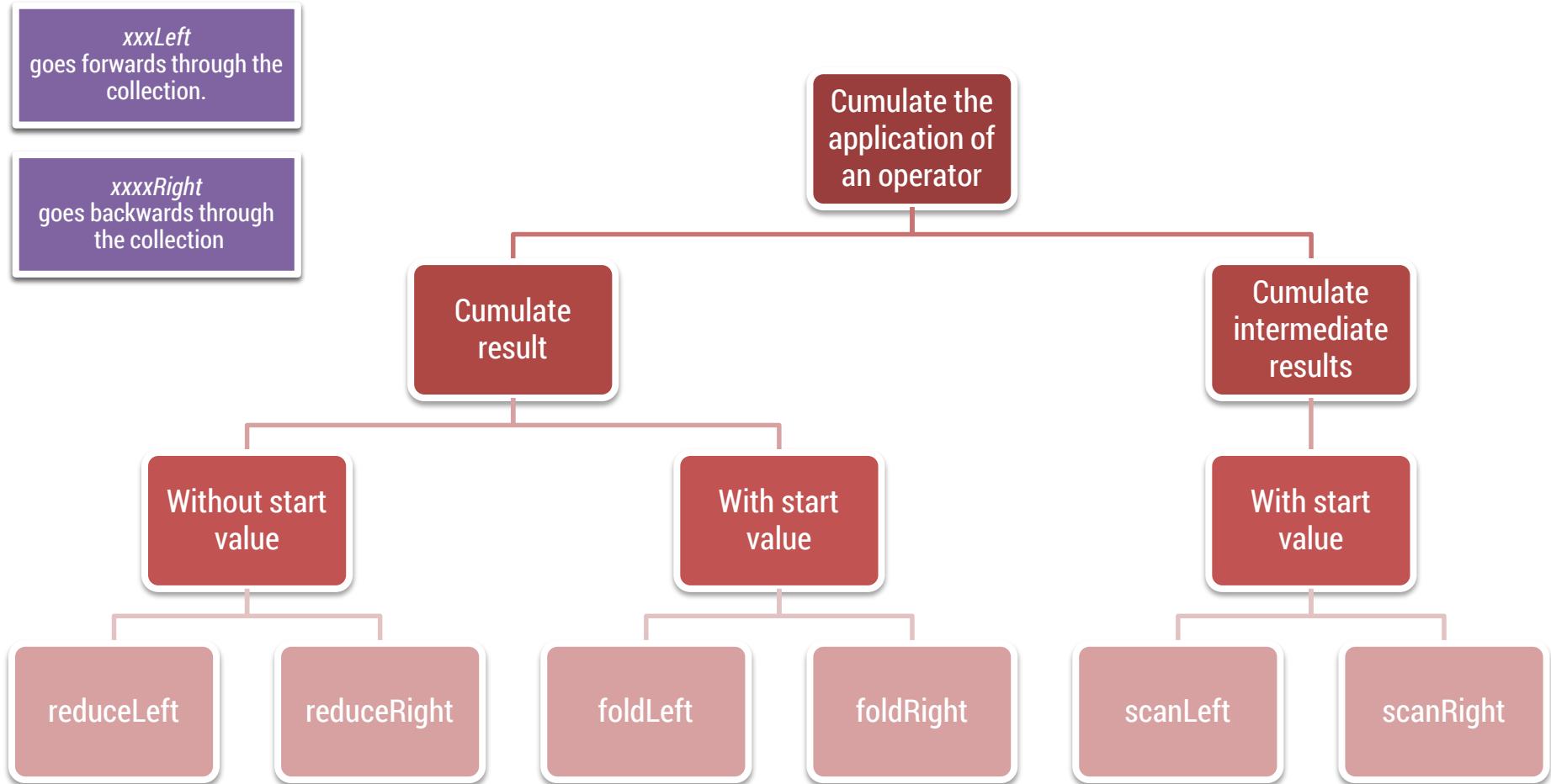
The scanLeft function

```
def scanLeft[B, That](z: B)(op: (B, A) => B) : That
```

- Produces a collection containing cumulative results of applying the operator going left to right.
- Also scanRight, similar

Examples:

Summary reduce / fold / collect



Exercise 2

- Complete the following snippets:

```
def length[A] (x : List[A]) : Int = {  
  (x foldRight (0)) (???)  
}  
                                (elt, acc) => acc+1
```

```
def map[A, B] (x: List[A], f: A => B) : List[B] = {  
  (x foldRight List.empty[B] ()) (???)  
}  
                                (elt, acc) => f(elt)::acc
```

Flatmap, zip and the rest

- Many HOF in list class
 - flatten → flattens a List (List () ...) into a List
 - flatMap → maps and flatten
 - zip → Returns a list formed from this list and another list by combining corresponding elements in pairs
 - sortWith → sorts the element given a comparison function (`lt: (A, A) => boolean`)

HOF examples, putting it all together

dup(List(1, 2, 3)) → List(1, 1, 2, 2, 3, 3)

```
def dup[A] = (_: List[A]).foldRight(List.empty[A]) {  
    (elt, acc) => elt :: elt :: acc  
}
```

Multiple values in one

2. TUPLES

A detour on pairs and tuples

- The *pair* consisting of x and y is written (x,y)

Example

- The type of this pair is
 - ▶
- Extractor pattern
 - ▶
- Also works for tuples (pair is Tuple2)

Translation of tuples

- A tuple (T_1, \dots, T_n) is a short form of
 - ▶ `scala.TupleN[T1, ..., Tn]`
- A tuple expression (e_1, \dots, e_n) is a function application
 - ▶ `scala.TupleN(e1, ..., en)`
- A tuple pattern (p_1, \dots, p_n) is equivalent to constructor pattern
 - ▶ `scala.TupleN(p1, ..., pn)`

Tuple class

```
case class Tuple2[T1, T2](_1: +T1, _2: +T2) {  
    override def toString = "(" + _1 + "," + _2 + ")"  
}
```

- The fields of a tuple can be accessed with `._1` and `._2`, ...
- Instead of pattern

```
val (name, value) = pair
```

- Pattern matching form is generally preferred

Tuple example: merging two sorted lists

```
val n1= List(1,2,4,6)          //> n1  : List[Int]
                           = List(1, 2, 4, 6)
val n2= List(2,3,5,7)          //> n2  : List[Int]
                           = List(2, 3, 5, 7)
merge(n1, n2)                //> res0: List[Int]
                           = List(1, 2, 2, 3, 4, 5, 6, 7)
```

Exercise 3

A bit longer

- Implement the function filter that filters the elements of a list (without using the existing filter function), i.e.

```
val b = (1 to 10).toList
          //> b  : List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
filter(_ % 2 == 0, b)
          //> res0: List[Int] = List(2, 4, 6, 8, 10)
```

- Implement the partition function (without using the existing function), you can traverse only once

```
partition(_ % 2 == 0, b)
//> res1: (List[Int], List[Int]) = (List(2, 4, 6, 8, 10), List(1, 3, 5, 7, 9))
```

Like a database access

3. FOR COMPREHENSIONS

Iterating in Java

```
List<Person> persons = ...  
for (int i = 0; i < persons.size(); i++) {  
    Person p = persons.at(i);  
    ...  
}  
  
for (Person p : persons) {  
    ...  
}
```

Iterating in Scala – For-loops

```
case class Person(name: String, age: Int)
val persons = List(Person("John", 23),
                   Person("Mary", 30),
                   Person("Alex", 22),
                   Person("Cindy", 40))

for (p <- persons)
  println(p.name)
```

- Implemented using HOF

- ▶ using `list.foreach{ ... }` under the hood

Filtering

```
case class Person(name: String, age: Int)
val persons = List(Person("John", 23),
                  Person("Mary", 30),
                  Person("Alex", 22),
                  Person("Cindy", 40))

for (p <- persons
      if p.age > 25)
  println(p.name)
```

- Sometimes, we want to filter out some values given a predicate
- Alternative using HOF:
 - ▶ `persons.foreach{ (p : Person) => println(p) }`

For-loops syntax

- A For-loop is of the general form

for (s) e

Generator

- Is of the form $x \leftarrow e$, where e is a list-valued expression. It binds x to successive values in the list.

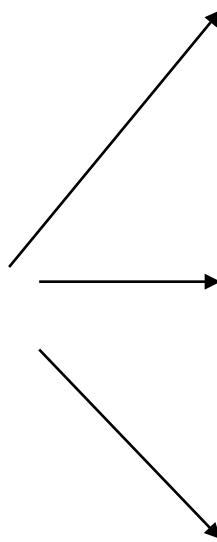
Definitions

- Is of the form $\text{val } x = e$. It introduces x as a name for the value of e in the rest of the comprehension.

Filter

- Is an expression $\text{if } f$ of type Boolean. It omits from consideration all bindings for which f is false.

s is a sequence of



For-loops, remarks

M. Odersky, *Scala by example*, chapter 10, available online at <http://www.scala-lang.org/docu/files/ScalaByExample.pdf>

1. The sequence **s** must start with generator. If there are several generators, later generators vary more rapidly than earlier ones.
2. The expression **e** is executed for each element generated from the sequence of generators and filters **s**.
3. It is always translated into `foreach`

Producing a collection

Introducing *For-comprehensions*

- Most of the time, we want to create a new collection
- New keyword : `yield`
- Is the general form of the for-comprehensions
 - ▶ For-loop is just syntactic sugar to mimic imperative languages loops

Syntax of For comprehensions

- A for expression is of form:

for (s) yield e

- Where s is a sequence of **generators** and **filters**
 1. A generator is of form $p \leftarrow e$ where p is a pattern and e an expression whose value is a list. It binds the values of the pattern p to the values of the list
 2. A filter is of form **if** f where f is a boolean expression. It removes all the bindings for which f is false.
 3. The sequence must start with a generator
 4. If there are multiple generators, the latest ones vary more quickly than the firsts.
 5. e is an expression whose value is returned by an iteration

Examples

```
val persons = List(Person("John", 23), Person("Mary", 30),  
Person("Alex", 22), Person("Cindy", 40))
```

```
val names = for (  
    p <- persons if p.age > 25  
) yield p.name
```

```
val names2 = for (  
    p <- persons;  
    if p.age > 25;  
    if p.name.startsWith("C"))  
) yield p.name
```

Translating comprehensions

Using map and filter

- Possible to translate comprehensions to map, flatMap and filter
- A few simple examples:
 - ▶ `for(x <- e) yield e'`
→ `e.map(x => e')`
 - ▶ `for(x <- e if f; s) yield e'`
 - with if f a filter and s a sequence of generators and filters
→ `for(x <- e.filter(x => f); s) yield e'`

Translating for to map and filter (2)

- ▶ `for (x<-e; y<-e', s) yield e''`
 - where s is a sequence (possibly empty) of generators and filters
 - `e.flatMap(x => for (y<-e'; s) yield e'')`
- ▶ (and the translation goes on)

Example of translation

```
for (x <- 1 to 2  
     y <- 'a' to 'b') yield (x, y)
```

```
(1 to 2).flatMap(x=>('a' to 'b')).map(y=>(x, y)))
```

Comprehensions generalization

- Not restricted to lists or any other collection
- Rely only on map, filter and flatMap
- Possible to extend to other types
 - has to define map, filter and flatMap
- Useful for arrays, databases, XML data, syntactic analyzers...

An example with databases

```
Database[Person] persons = ...  
  
for (p <- persons  
      if p.age > 20) yield p.name  
  
persons.filter(p => p.age > 20).map(p => p.name)
```

Finding everything



4. EXHAUSTIVE COMBINATIONS

Generating exhaustive search

- Possible with other techniques
- For comprehensions make it easier

Example

- ▶ generating all the couples of integer values under 100 for which the sum of those numbers is prime

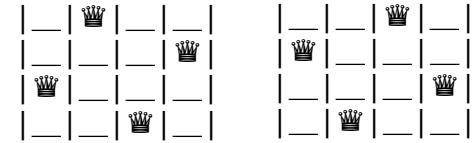
Possible implementations

```
def primeSum(max: Int): List[(Int, Int)] =  
  for {  
    i <- (1 to max).toList  
    j <- (1 to max).toList  
    if (isPrime(i + j))  
  } yield  
  (i, j)  
  
// A possible prime function, using matching  
def isPrime(i: Int): Boolean =  
  i match {  
    case i if i <= 1 => false  
    case 2 => true  
    case _ => !(2 to (i - 1)).exists(x => i % x == 0)  
  }
```

COMMENTS FOR ASSIGNMENT #5

Reading exercise

- The "n queens" problems
 - Placing n queens on a n by n chessboard so that no two queens attack each other
- Typical recursive problem
 - Place correctly a single queen to any solution of the problem of placing $n-1$ queens on an nxn chessboard
 - Recursions stops when we have to place 0 queens



Read solution page 80 in

<http://www.scala-lang.org/docu/files/ScalaByExample.pdf>



Advanced programming paradigms

7. Typing and streams

Disclaimer

The material on types is adapted from chapter 12 from *Programming in Scala*

The stream part is adapted from M. Odersky, "*Functional Programming in Scala*", online Coursera course

Lesson's objectives

See the advanced types, understand mixins and use lazy evaluation

- ▶ Traits
- ▶ Type system, bounds and variance
- ▶ Streams, lazy evaluation and infinite sequences

Interfaces supercharged

1. TRAITS

Traits

- Like Java interfaces, but allow for methods implementation and val declarations
- Traits
 - ▶
 - ▶
 - ▶
 - ▶

Traits (2)

- Trait definitions
 - ▶ Similar to classes, keyword **trait**
 - ▶ No constructor parameters and single constructor
 - ▶ Can be extended using inheritance
- A class can **mix** several traits
 - ▶ Solves diamond inheritance problem

Traits as interfaces

- In its simplest form, *trait* = *interface*

```
trait Logged {  
    def log(msg: String)  
}  
  
class ConsoleLogger extends Logged {  
    override def log(msg: String) = println("[LOG] " + msg)  
}
```

- All Java interfaces can be directly used as Scala traits

Traits with implementations

- Traits can also provide methods implementations

```
trait ConsoleLogger extends Logged {  
    override def log(msg: String) = println("[LOG] " + msg)  
}
```

A complete example: logging factory

Example explained

```
class Customer(name: String) extends Person(name) with ConsoleLogger{  
    log(s"Person $name created")  
}
```

```
class Customer(name: String) extends Person(name) with Logged {  
    log(s"Person $name created")  
}
```

```
val x = new Customer("Patrick Jane") with ConsoleLogger  
val y = new Customer("Teresa Lisbon") with FileLogger
```

Rules: **extends** or **with** ?

- Use the keyword **extends** to mix a trait into a class that doesn't explicitly inherit from another class
- Use the **with** keyword to mix a trait into a class that already extends another class
- Use **with** repeatedly if required
- If multiple traits, the rightmost wins
 - For diamond-like problems

Traits in Scala's libraries

Thin vs Rich interfaces

- Rich interface
 - ▶ many methods, more convenient for the caller
 - ▶
- Thin interface
 - ▶ few methods, easier to implement, less functionalities
 - ▶
 - ▶

Traits in Scala's libraries

- Libraries make heavy use of traits
 - Add methods to a class
 - Traits can enrich a thin interface, making it rich
- Adding a concrete method is a one-time effort
 - Rich interfaces require less work in Scala

Recipe



- Define a trait with few abstract methods (the thin part) and add large number of concrete methods
- Mix the trait into a class and implement the thin portion

An example: the Ordered[T] trait

- Provided in Scala's libraries

```
trait Ordered[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  
    def compareTo(that: A): Int = compare(that)  
}  
                                         renvoyer un int → utile pour dire plus que juste pas égal
```

- Can be used for comparing using standard operators (<, >, <=, >=)
- Beware ==

Exercise 1: On Ordered[T] trait

Make the following code possible:

```
val a = Foo(2); val b = Foo(3); val c = Foo(2)
```

```
a < b
```

```
a > b
```

```
a == b
```

```
a == c
```

```
a >= c
```



Beyond generics

2. ADVANCED TYPING

Type parameters

Reminder

- Classes, traits and methods can be **type-parametrized**
- Use square brackets after identifier to define type parameter.

Examples

List(1,2) => Int

List("a", "b") => String

List("a", 2) => Any

Convention

- type parameters are a single capital starting with A

With type inference?

```
def getMiddle[A] (x: List[A]) = x(x.length / 2)
```

- Compiler infers the actual type depending on the compile-time information:
 - ▶
 - ▶
 - ▶
- Scala is **statically** typed, all types are determined at compile-time



Type bounds: motivation

- From lesson 3

```
abstract class IntSet() {  
    def add(x: Int): IntSet  
    def contains(x: Int): Boolean  
}
```

- First try:

```
abstract class Set[A] {  
    def incl(x: A): Set[A]  
    def contains(x: A): Boolean  
}
```

- Implementation as binary search trees, problem ? 

-
-

Type upper bound example

- We can solve this problem with upper bounds:
 - Our class must be a subclass (or mixin) of **another** class or trait
- Already seen in reduce functions on Lists, the other way around
- Operator < :

Lower bounds on set

Type bounds

Used with type parameters (T) to force a relation of the parameter type to another type Y

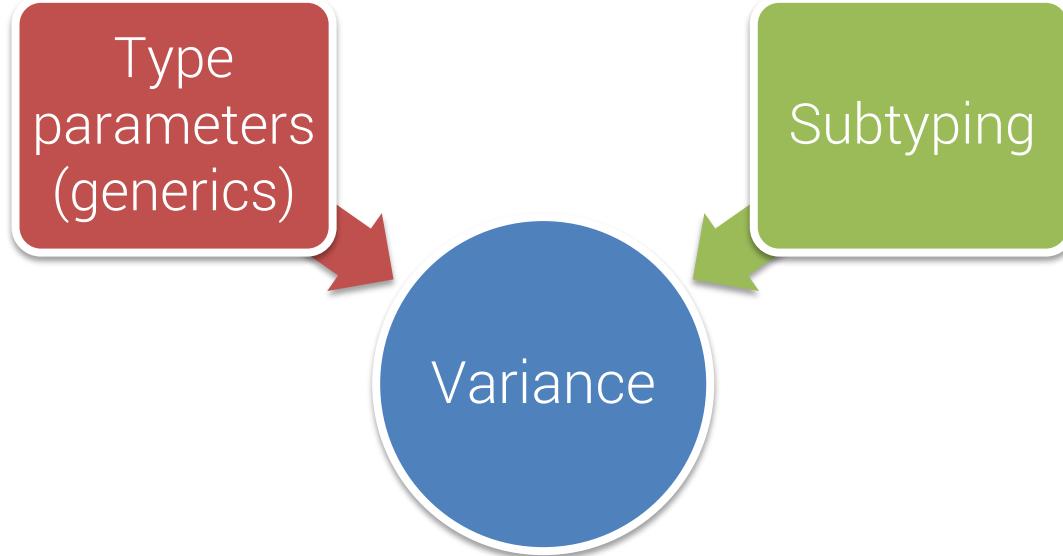
Type bounds

- Lower bound →
 - the parametric type T must be a superclass of Y
 - Written $T >: Y$
- Upper bound →
 - the parametric type T must be a subclass of Y
 - Written $T <: Y$

View bounds

- Our `IntSet` solution works with classes that explicitly extends *Ordered*
- What about `String`, `Int` etc... ?
- Possible with *implicit conversion* (later in DSL course)
- Requires the use *view bounds*
 - `<%`, means that there must exist a conversion to that type

Variance



- Is Stack [String] a subtype of Stack [AnyRef] ?
- E.g., if T is a subtype of S, then Stack [T] should be a subtype of Stack [S] → this is *co-variance*

Sorts of variance

- *Covariance*

Enables you to use a more specific type than originally specified.

- *Contravariance*

Enables you to use a more generic (less derived) type than originally specified.

- *Invariance (or non-variance)*

Means that you can use only the type originally specified; so an invariant generic type parameter is neither covariant nor contravariant.

Variance annotations

- By default, in Scala, generic types are non-variant
 - E.g., stacks with different elements types never have subtype relations
- One can enforce
 - Co-variance:
 - Contra-variance

Covariance and contra-variance

covariance: un tableau de chat est un tableau d'animaux

contra-variance: un tableau d'animaux est un tableau de chats

- class Stack [-A] means
 - ▶ If T subtype of S,
 - Stack [S] subtype of stack [T] ???
- In FP (pure), all types could be covariant
- Variance is not inherited
 - ▶ One should annotate **every** class



Variance in methods declarations

- A bit tricky, but **the compiler knows the rules**
- To be safe for **covariance**
 - type T must appear only in "producer" (positive) positions in methods
- To be safe for **contravariance**
 - type T must appear only in "consumer" (negative) positions in methods

```
abstract class Container[A] {  
    def get : A          // + covariant position  
    def put(elem: A)    // - contravariant position  
    val elem: A         // + covariant position  
}
```

Exercise for variance

Which are correct?

```
abstract class Container[+A] {  
    def get : A  
    def put(elem: A)          put ne va pas  
    val elem: A  
}
```

```
abstract class Container[-A] {  
    def get : A  
    def put(elem: A)          get et elem ne vont pas  
    val elem: A  
}
```

```
abstract class Container [+A] {  
    def get : A  
    def put[B >: A](elem: B)   ok  
    val elem: A               B>A ?? what ?? —> trick pour faire fonctionner le compilateur.  
}                                Side-effect: on peut mettre des animaux dans des collections de chats...
```

Don't do it if you don't need it

5. LAZY EVALUATION

Definition

Do things as late as possible and never do them twice

Lazyness in Scala

- Sound optimization, as in FP each expression produces the same result each time it's evaluated!

Why lazy evaluation ?

- Was used in *Anagrams* for loading dictionary



Lazyness in Scala

- Powerful, avoids unnecessary computations
- Haskell uses this as default
- In Scala, default is strict evaluation
 - why not lazy by default?
 - allows lazy values

Exercise 3

- What gets printed as side-effect?

```
def expr = {  
    val x = {print("x"); 1}  
    lazy val y = {print("y"); 2}  
    def z = {print("z"); 3}  
    z + y + x + z + y +x  
}
```

xxyz

Computing with infinite sequences

4. STREAMS

Motivation

- Using HOF, easy to find the second prime number between 1000 and 10000
 - ▶
- Performance, however, is bad. Why ?
 - ▶

Motivation : what can we do?

- 1.
- 2.

Streams

- Similar to List
 - Tail evaluated only *on demand*
- Creating a Stream
 - Factory like other collections
 - Stream(1, 2, 3)
 - Calling `toStream` on any collection will turn it into a Stream

```
(1 to 1000).toStream  
  > res0: Stream[Int] = Stream(1, ?)
```

Methods on Streams

- Almost all methods of List
 - ((1000 to 10000).toStream filter isPrime) (1)
- Major exception is ::
 - ▶ `x :: xs` always produces a list
- Alternative is # :: which produces a Stream
 - ▶ `x# :: xs` \equiv `Stream.cons(x, xs)`
 - ▶ `# ::` can be used in patterns and expressions

Implementation of Streams

```
trait Stream[+A] extends Seq[A] {  
    def isEmpty: Boolean  
    def head : A  
    def tail : Stream[A]  
    ...  
}
```

```
object Stream{  
    def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {  
        def isEmpty = false  
        def head = hd  
        lazy val tail = tl  
    }  
  
    val empty = new Stream[Nothing] {  
        def isEmpty = true  
        def head = throw new NoSuchElementException("empty.head")  
        def tail = throw new NoSuchElementException("empty.tail")  
    }  
}
```

Streams are lazy

```
def cons[T] (hd: T, tl: => Stream[T]) = new Stream[T] {  
    def isEmpty = false  
    def head = hd  
    lazy val tail = tl  
}
```

- More efficient because
 - ▶ computation only if required
 - ▶ computation done once
 - ▶ results saved if reused

Explanation on how the magic works

Definition of filter in Streams

```
def filter(p: T => Boolean) : Stream[T] =  
  if(isEmpty) this  
  else if(p(head)) cons(head, tail.filter(p))  
  else tail.filter(p)
```

Exercise 2

- Consider the following code:

```
def range(low: Int, high: Int): Stream[Int] = {  
    println(s"Calling range with $low")  
    if (low >= high) Stream.Empty  
    else low #:: range(low + 1, high)  
}
```

- What gets printed for
 - range(1,10)(0) calling range with 1
 - range(1,10).map(_+1) calling range with 1
 - (range(1,10).map(_+1)).toList

Infinite streams

- All elements except the first are computed only when needed
 - We can define **infinite streams!**

Examples:

- The stream of all integers starting from a given number:

```
def from(n: Int) : Stream[Int] = n #::from(n+1)
```

- The stream of all natural numbers:

```
val nats = from(0)
```

- The stream of all multiples of 4:

```
nats map (_ * 4)
```

Exercise 4

- Consider two ways to express infinite streams of multiples of a given number n

```
val xs = from(1) map (_ * n)  
val ys = from(1) filter (_ % n == 0)
```

- Which one is the faster and why?

Si après on a un

take 3 toList

filter —> il faut parcourir les 8 premiers éléments pour en trouver 3. On génère/consume donc plus
map —> juste les trois premiers

Conclusion



- This concludes the FP part of the *AdvPrPar* course

What have we seen?

higher-order functions

case classes and pattern matching

immutable collections

absence of mutable state

flexible evaluation strategies
strict/lazy/by name



A useful toolkit for every programmer
A different way of thinking

What remains to be covered

- FP with state
 - what does it mean?
 - what does it change?
- *Lesson 8* : Parallelism, how to exploit immutability
 - the *Actors* model
 - parallel collections
 - *Futures*
- *Lesson 9* : Domain-specific languages
 - implicit conversions
 - high-level libraries as embedded DSLs
 - interpretation techniques for external DSL
- What else could be done ?
 - Data analytics with Scala, Web framework (*Play!*, *Akka*), parser combinators...

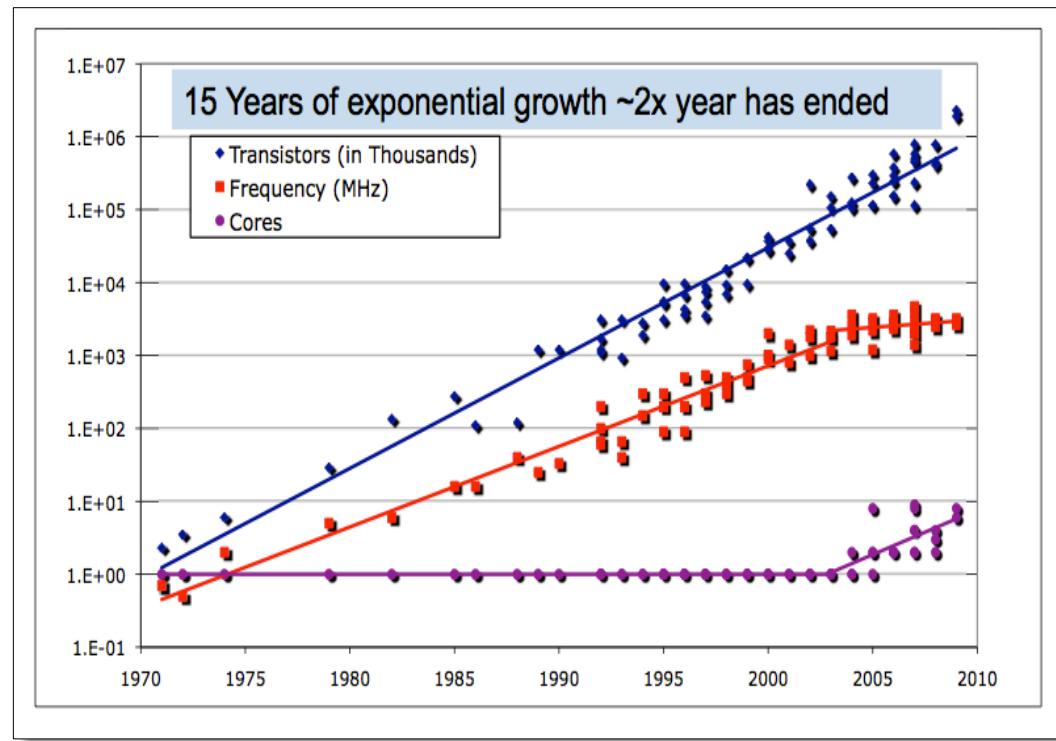


Advanced programming paradigms

8. Parallelism

The world is changing

- Moore's law achieved with **cores**, no longer transistors or frequency



Data from K. Olukotun, L. Hammond, H. Sutter, B. Smith, C. Batten, and K. Asanovic, taken from Odersky M. @ DEVOXX11

Why is FP crucial for parallel programming?

- Because it's hard!
 - races, deadlocks, memory effects...
- Reasons:
 - non-deterministic threads interleaving
 - observable because of shared state
- If we eliminate shared state, we eliminate the problem

In FP, we don't have state!

Scala parallel toolbox



ACTORS

PARALLEL GRAPH
PROCESSING

STM

PARALLEL DSLS

FUTURES

PARALLEL
COLLECTIONS
DISTRIBUTED

Lesson's objectives

**Discover and apply two tools for
tackling parallelism in Scala**

- ▶ Parallel collections
- ▶ Futures

Leveraging parallelism on collections

1. PARALLEL COLLECTIONS

Parallel collections

1. Add `.par` to your collection

2. Your code is now parallel (i.e. runs on all multiple cores)

3. There is no point 3.

```
val a = new Array[BigInt] (1000000)  
  
for (i <- 0 until a.size) a(i) = i  
  
val t1 = time {  
    val sum = a.sum  
}  
  
val t2 = time {  
    val sum = a.par.sum  
}  
  
println(s"$t1 ms and $t2 ms")
```

299 and 120 ms

biensur, si on loope sur 5000, on ne gagne pas vraiment. Il faut du temps pour mettre en place le parallélisme.

Pros and cons

<http://docs.scala-lang.org/overviews/parallel-collections/configuration.html>

Pros

easy to use

can make your code
faster

easy to use

Cons

overhead requires
large collection to be
interesting

control is hidden

granularity can't be
changed

cores, nb threads, etc.
not customizable



Parallel collections – Beware

```
List(1, 2, 3, 4, 5) foreach println _
```

ordre pas nécessairement maintenu !

```
List(1, 2, 3, 4, 5).par foreach println _
```

Alternatives



<http://scala-blitz.github.io/>

- Overheads like boxing or use of iterators have to be eliminated
- Task-based preemptive scheduling used in *Scala Parallel Collections* does not handle certain kinds of irregular data-parallel operations well

Measuring performance on the JVM

- Not straightforward (at all)
 - ▶ JIT —> compilation à la volée
HotSpot compiling —> JVM peut optimiser des parties de code qui prennent du temps et sont souvent appelées... Change complètement le temps mesuré !
GCC, cold start, ...
- Some pointers to look at
 - ▶ <http://scalameter.github.io/>
 - ▶ <https://code.google.com/p/caliper/>
 - ▶ <https://github.com/alno/sbt-caliper>

Computing without blocking

2. FUTURES

Futures

Abstraction over a value that **might or might not** be defined

Future

- Non-blocking
 - Execution
 - Composition
 - Transformation
- Fully standalone library

Motivation

- A Future is a handle for a value not yet available
- Asynchronous API
 - Does not wait for a result before returning
 - Slow API return a Future right away and «fill in» when it resolves.

Example

HTTP request: launch, do something else, .get later

Motivation

- Several important libraries have their own Futures/Promises implementations.
 - `java.util.concurrent.Future`
 - `scala.actors.Future`
 - `com.twitter.util.Future`
 - `akka.dispatch.Future` akka : version commerciale de scala
 - `scalaz.concurrent.Promise`
 - `net.liftweb.actor.LAFuture`
- This makes it clear that...
 - Futures are important, popular and powerful
 - Fragmentation in the Scala ecosystem

Motivation (2)

- The standard Java `java.util.concurrent.Futures` is neither efficient nor composable.
- Scala *Futures* more powerful (composable) by taking advantage of higher-order functions
 - `scala.concurrent.future`

et un peu plus efficace

Java Future

- The standard Java `java.util.concurrent.Futures` neither efficient nor composable

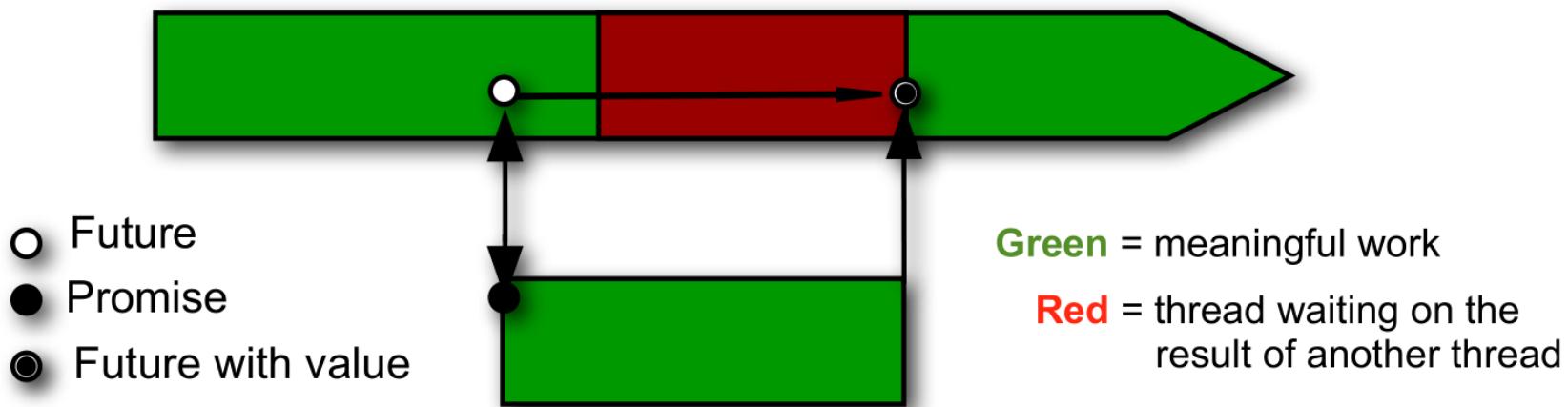


Image from M. Odersky, H. Miller, P. Haller,
Actors reloaded, DEBS2012 conference,
Berlin.

Java Future (2)

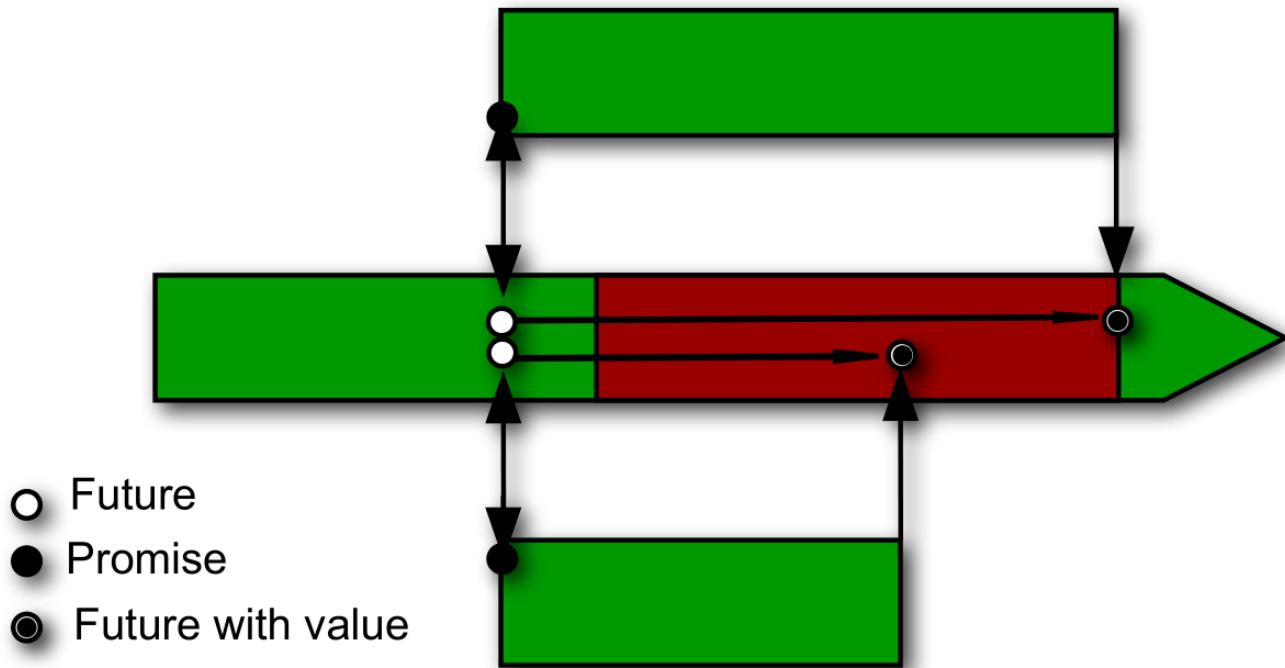


Image from M. Odersky, H. Miller, P. Haller,
Actors reloaded, DEBS2012 conference,
Berlin.

Scala Future, what we want to do

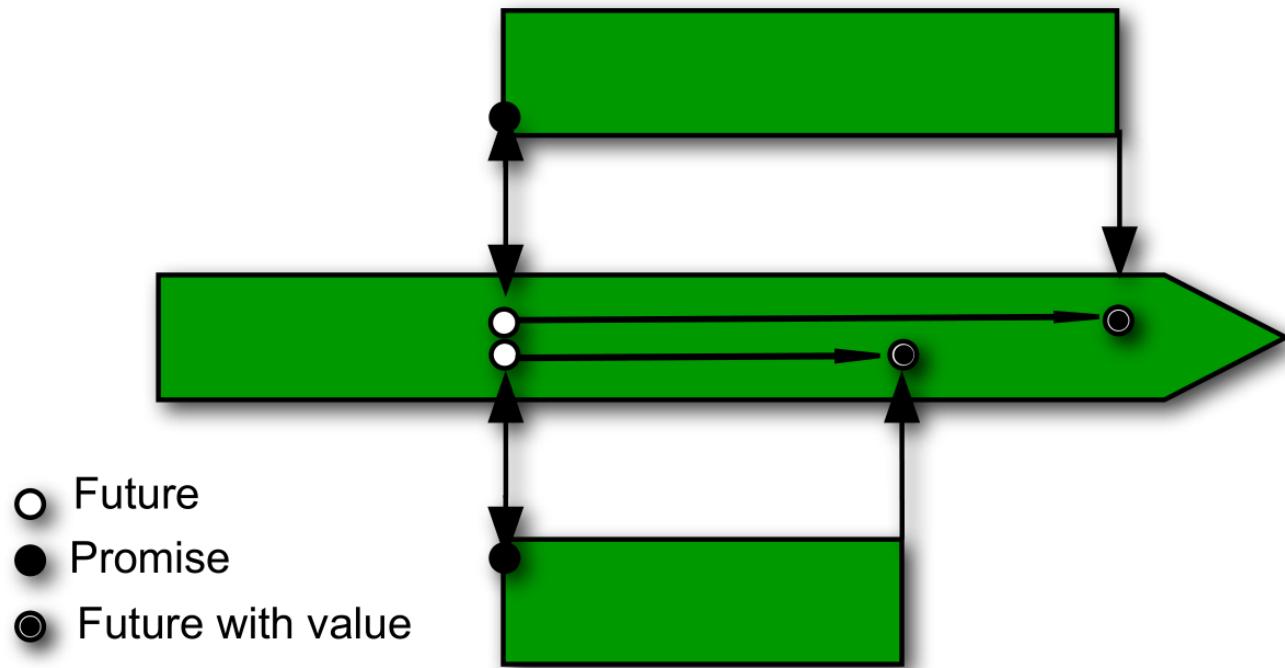


Image from M. Odersky, H. Miller, P. Haller,
Actors reloaded, DEBS2012 conference,
Berlin.

Operations on Futures

- Start async computation
- Assign result value
- Wait for result
- Compose futures

1. Start async computation

Example

```
val f = Future(3 + 4)
```

- Can also return Future from a function

```
def longComputation(x: Int): Future[Int] = {  
    Future(Thread.sleep(500) ; 5)  
}
```

2. Assign result value

Using callback

- Methods callbacks when Future resolves, asynchronous
 - ▶ onComplete, onSuccess, onFailure

Futures callbacks

Callback examples

```
val f: Future[Int] = Future(3 + 4)
```

```
f onComplete {  
  case Success(result) =>  
  case Failure(t) => println("Error occurred" + t.getMessage)  
}
```

```
f onSuccess {  
  case x: Int => println("Computation done, result is " + x)  
}
```

```
f onFailure {  
  case t => println(t)  
}
```

3. Wait for result

- Normally, code should be non-blocking
 - We should not wait for result
- Sometimes we want to wait for Future to be resolved before continuing
- Use Await object
 - `Await.ready`
 - `Await.result`

Blocking for completion



Blocking is bad!

```
val f = Future {  
    Thread.sleep(1000);  
    println("This is the future")  
    5  
}
```

```
val result : Int = Await.result(f, 3 seconds)  
println(result)
```

```
f onComplete { case _ => println("Job done") }  
Await.ready(f, 3 seconds)
```

4. Compose future with transformers

Because Futures are non-blocking

def foreach[U] (f: T => U) : Unit *thread du future réutilisé à la fin pour le foreach*

def map[S] (f: T => S) : Future[S]

def flatMap[S] (f: T => Future[S]) : Future[S]

def filter(p: T => Boolean) : Future[T]

Example: *Text*

Futures composition

Composition with comprehensions

```
val f1 = Future {  
    Thread.sleep(10 + Random.nextInt(500))  
    println("F1 finished")  
    42  
}  
  
val f2: Future[String] = future {  
    Thread.sleep(10 + Random.nextInt(500))  
    println("F2 finished")  
    " is the answer"  
}  
  
// Combine the result of two futures together  
val comb = for {  
    r1 <- f1  
    r2 <- f2  
} yield (r1 + r2)
```

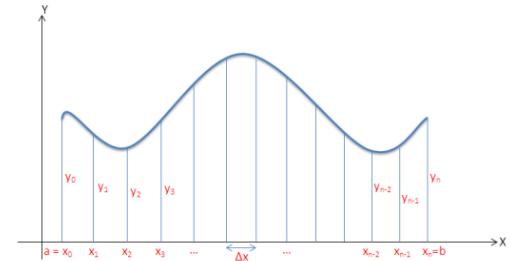
- Other way to compose Futures
- Sometimes easier to express

Today's summary



- Many tools for parallelism in Scala
- Parallel collections easy to use
 - Need big collection to be efficient
- Futures allow
 - To do things asynchronously
 - Use results without locking
 - Focus on computation
- Other interesting paradigm: *Actors*

Assignment 8



- Parallel function integration
- Bitcoin to CHF conversion
 - REST API calls with Futures
 - Future composition
 - Using JSON



Order of operations

```
println("> Before the future")

val f = Future {
    "[FUTURE] Hello from the future!"
}

println("> After the future")

// Completely asynchronous
f.onSuccess { case s => println(s) }

println("> Doing something else, long")
...
println("> Done doing long thing")

// Blocks until the future is ready, blocking is bad
Await.result(f, Duration.Inf)
```

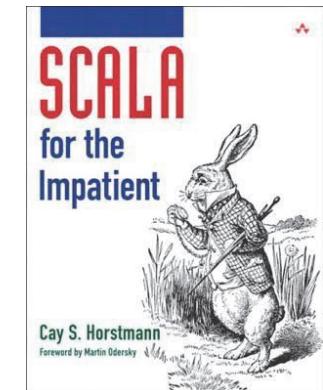
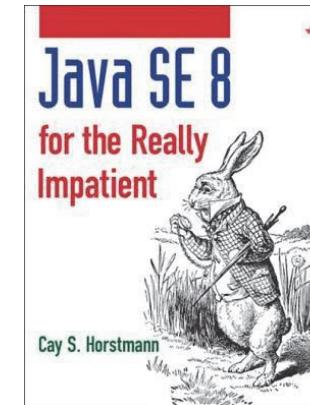
Advanced programming paradigms

9. Domain Specific Languages (DSL)

Disclaimer

The material for this chapter is adapted from Cay Horstmann's lecture given in the course in 2015.

<http://horstmann.com/>



Lesson's objectives

**Understand what are DSLs and
develop a simple one**

- ▶ Introduction
- ▶ Implicits
- ▶ Interactive lab

What's a DSL

1. INTRODUCTION

Domain Specific Languages

- Programming language for a specific domain
 - ▶ SQL
 - ▶ PHP/Rails/Lift
 - ▶ Maxima/Octave/Matlab
 - ▶ Dot / Processing
 - ▶ SBT / Rake
 - ▶ Embedded systems

External DSL

- *Standalone language*, with constructs adapted to the programming domain (e.g. mixing HTML and code)
- Disadvantages
 - Need to learn many different languages
 - Languages often designed by amateurs (DOS shell, PHP)

Internal DSL

- Embedded into a regular programming language
- *Disadvantage*: most programming languages have little capability for expressing domain-specific syntax
- Examples:
 - *Ruby* metaprogramming (can redefine what `x.foo` means)
 - *Scala* : operators, code blocks, implicits

Operators in Scala, reminders (1)

- Operators without parentheses

numbers map square reverse

≡

numbers.map(square).reverse

- Used for «fluent APIs» (for instance in testing)

```
it should "yield the original when applied twice" in
{
    assert(x.reverse.reverse == x)
}
```

Operators in Scala, reminders (2)

- Operators can be any sequence of *operator characters* (including Unicode)

`==>`

`⇒`

- Define a method with `==>` or `⇒`
 - ▶ Unary prefix `+` `-` `!` `~` are converted to calls to `unary_operator`
 - ▶ Rvalue `()` calls `apply`
 - ▶ Operators ending in `:` are right-associative

Code blocks

```
// Simple block passing, note the by name call
def time(f: => Unit) = {
    val start = System.currentTimeMillis
    f // Execute the block
    val duration = System.currentTimeMillis - start
    duration
}

// With curried argument
def timeMsg(s: String)(f: => Unit) = {
    println(s);
    f
}

time{
    (1 to 100000).reverse.sorted
}
```

Making conversions automatic

3. IMPLICITS

Implicit conversion functions

- Is a method or function that automatically converts a type to another
- Its name is irrelevant
- Marked with the **implicit** keyword

Example

```
implicit def intToRat(x: Int) = new Rational(x)
```

```
val f1: Rational = 2  
3 * Rational(3, 4)
```

on ne va pas modifier la classe int (primitif), mais plutôt donner une fonction au compilateur pour convertir un int en un rationnel

"Pimp my library" pattern

When are conversions made

Implicit conversion are made

- If the type of an expression differs from expected type
- If a non existent member is accessed
- If object invokes a method with different type than expected

Pt2: "HAL".increment
string n'a pas increment, mais
PimpedString oui =>
conversion de string à
PimpedString

- Additional rules:
 - No conversion made if code compiles without it
 - Multiple conversion are NEVER done
 - Ambiguous conversion trigger an error

Scope of implicits functions / methods

- Implicits must be in scope
 - Current scope, without prefix
 - Implicit scope: member of companion objects
- Precedence rules
 - Local implicits
 - Imported implicits
 - Companion objects of types
 - Companion object of the type arguments of the type
- Remark: starting with Scala 2.10
 - **import** language.implicitConversions

Since Scala 2.10, implicit classes

```
implicit class PimpedString(s: String) {  
    def increment = s.map(c => (c + 1).toChar)  
}  
  
val c: PimpedString = "HAL"  
println("HAL".increment)
```

- Reduce boilerplate
 - no need for companion object
 - no need for import
- Makes the class' primary constructor available for implicit conversions when the class is in scope, with some restrictions

la classe ne peut être dans un fichier seul + un seul paramètre possible

Implicit parameters

- A function / method can have one or more implicit parameters → context provide default value
- Must be curried

```
def read(path: Path)(implicit cs: Charset) =  
  new String(java.nio.file.Files.readAllBytes(path, cs))
```

- Not specified in the call
- A unique implicit object type must be «in the right place»
- Useful for global defaults

WAIT... There's more

- For many DSLs, it is necessary for operators to build up intermediary tree structures that are then translated to some target, such as SQL. The *Lightweight Modular Staging* project provides useful plumbing if you need to do that.
- Macros are compiler plug-ins that allow you to define new language constructs and rules for their compilation.
- Check out some real-life DSLs:
 - The simple build tool (SBT)
 - The ScalaTest and [specs2](#) testing frameworks
 - A somewhat obsessive [Regex DSL](#)
 - For fun if not profit, [Baysick](#), a vintage Basic interpreter
 - An [image processing library](#) that is waiting to be turned into a DSL
- Next...your very own!

Applying this stuff

3. INTERACTIVE LAB