

Week 1

Call by value first resolves the “values” before calling the function, while call by name first calls the function, gets the results and then resolves them.

Week 2

Take a function as a parameter:

```
def sum(f: Int => Int, a: Int, b: Int): Int =  
  if a > b then 0  
  else f(a) + sum(f, a + 1, b)
```

We can also generate functions using functions:

```
def sum(f: Int => Int): (Int, Int) => Int =  
  def sumF(a: Int, b: Int): Int =  
    if a > b then 0  
    else f(a) + sumF(a + 1, b)  
  sumF  
  
def sumInts = sum(x => x)  
def sumCubes = sum(x => x * x * x)  
def sumFactorials = sum(fact)  
  
sumCubes(1, 10) + sumFactorials(10, 20)
```

Generalization:

It's the same as creating a function that takes the n-1 arguments, and one takes the last one:

```
def f(ps1)...(psn-1) = (psn => E)
```

Types

```
Type = SimpleType | FunctionType  
FunctionType = SimpleType '=>' Type  
| '( ' [ Types ] ') ' '=>' Type  
SimpleType = Ident  
Types = Type { ' , ' Type }
```

Several ways of writing functions that return functions

```
def isGreaterThanBasic(x: Int, y: Int): Boolean =  
  x > y  
val isGreaterThanAnon: (Int, Int) => Boolean =  
  (x, y) => x > y  
val isGreaterThanCurried: Int => Int => Boolean =  
  x => y => x > y // Same as `x => (y => x > y)`  
def isGreaterThanCurriedDef(x: Int)(y: Int): Boolean =  
  x > y
```

▷ Curried signifie que la fonction prend ses arguments un par un ! (en fait elle renvoie une nouvelle fonction à chaque fois) C'est utile si on veut appliquer des transformations partielles (fixer le premier argument et retarder l'application du second).

Week 3

Classes and Substitutions

Now suppose that we have a class definition,

```
class C(x1, ..., xm) { ... def f(y1, ..., yn) = b ... }
```

where

- ▶ The formal parameters of the class are x_1, \dots, x_m .
- ▶ The class defines a method f with formal parameters y_1, \dots, y_n .

(The list of function parameters can be absent. For simplicity, we have omitted the parameter types.)

Question: How is the following expression evaluated?

```
C(v1, ..., vm).f(w1, ..., wn)
```

Classes and Substitutions (2)

Answer: The expression $C(v_1, \dots, v_m).f(w_1, \dots, w_n)$ is rewritten to:

```
[w1/y1, ..., wn/yn][v1/x1, ..., vm/xm][C(v1, ..., vm)/this] b
```

There are three substitutions at work here:

- ▶ the substitution of the formal parameters y_1, \dots, y_n of the function f by the arguments w_1, \dots, w_n ,
- ▶ the substitution of the formal parameters x_1, \dots, x_m of the class C by the class arguments v_1, \dots, v_m ,
- ▶ the substitution of the self reference *this* by the value of the object $C(v_1, \dots, v_m)$.

```
extension (r: Rational)
  def min(s: Rational): Rational = if s.less(r) then s else r
  def abs: Rational = Rational(r.numer.abs, r.denom)
```

Using Extension Methods

Extensions of a class are visible if they are listed in the companion object of a class (as in the code above) or if they defined or imported in the current scope.

Members of a visible extensions of class C can be called as if they were members of C . E.g.

```
Rational(1/2).min(Rational(2/3))
```

Caveats:

- ▶ Extensions can only add new members, not override existing ones.
- ▶ Extensions cannot refer to other class members via *this*

Step 2: Infix Notation

An operator method with a single parameter can be used as an infix operator.

An alphanumeric method with a single parameter can also be used as an infix operator if it is declared with an `infix` modifier. E.g.

```
extension (x: Rational)
  infix def min(that: Rational): Rational = ...
```

It is therefore possible to write

<code>r + s</code>		<code>r.+(s)</code>
<code>r < s</code>	<code>/* in place of */</code>	<code>r.<(s)</code>
<code>r min s</code>		<code>r.min(s)</code>

Déterminée en fonction du caractère qui démarre l'opérateur:

Precedence Rules

The *precedence* of an operator is determined by its first character.

The following table lists the characters in increasing order of priority precedence

```
(all letters)
|
^
&
< >
= !
:
+ -
_
* / %
(all other special characters)
```

Functions and Methods

Note that a method such as

```
def f(x: Int): Boolean = ...
```

is not itself a function value.

But if `f` is used in a place where a Function type is expected, it is converted automatically to the function value

```
(x: Int) => f(x)
```

or, expanded:

```
new Function1[Int, Boolean]:
  def apply(x: Int) = f(x)
```