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).