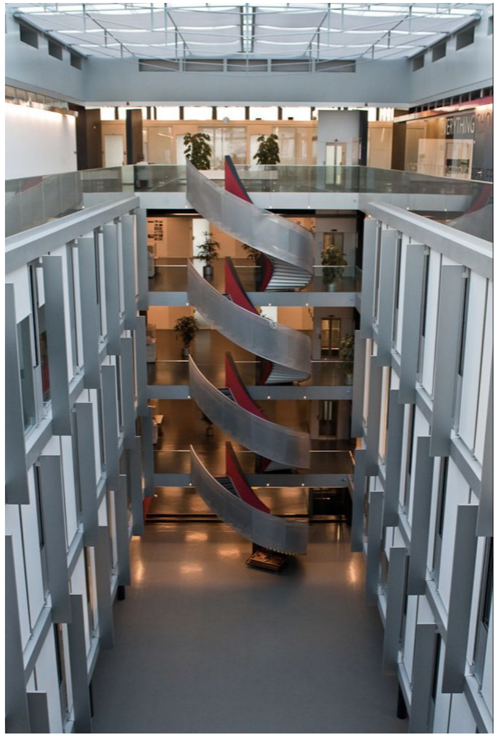
Scala

(As functional language)

Scala Introduction

Scala is a sophisticated language, it is used in large websites, applications and infrastructure:

- Created by Martin Odersky (father of javac) and influenced by Java, Ruby, Smalltalk, etc.
- The word “Scala” from Italian and the meaning is “staircase”. Actually Scala language logo is inspired in the stairs of EPFL, where it was born in 2004. (left image)
- It is said that Scala is a newer, improved Java
- Scala runs on a JVM. This implies that Scala will work wherever a Java code works.



Martin Odersky

Martin Odersky:

- German Computer Scientist. Got his PhD from ETH Zurich under supervision of Niklaus Wirth, who was designer of several programming languages including Pascal
- Postdoctoral at IBM and Yale
- Professor at EPFL: code analysis and programming languages
- Implemented the GJ compiler, that became the basis of Java compiler
- Teaches 3 courses in coursera about Scala:
  - Functional programming principles in Scala
  - Functional program design in Scala
  - Programming reactive systems

Why Use Scala (briefly)

- **Scala stands for Scalable Language:**
  - It is designed to work from small scripts to massive data processing systems.
  - It powers some of the busiest websites as Twitter, Netflix, Tumblr, Linkedin, Foursquare and more.
- **Scala is flexible and convenient:**
  - You can have customized types, control structures and collections
- **Is high-level language:**
  - Do not deal with details of the computer
- **Scala is used by:**
  - Software developers, backend engineers, data engineer and sometimes by Data Scientists and Machine Learning Engineers. Actually Apache Spark is written in Scala.

Scala in short:

Scala in short:

- It’s a high level language
- It’s statically typed:
  - The type of the variable is know at compile-time
- The syntax is concise but still readable (expressive)
- It is a pure OOP language.
  - Every variable is an object
  - Every operator is a method
- It is also a Functional Programming language.
  - Functions are also variables, you can pass them into other functions
- It has a sophisticated type inference system
- You can write code using OOP, FP or combine them.
- Scala code results in .class files that run on the JVM
- It is easy to use Java libraries in Scala

Scala as OOP and FP

Scala mix Object Oriented and Functional Programming concepts:

Scala is classified **as Object Oriented because:**

- Every value is an object
- Every operation is a method

Scala is classified **as Functional because:**

<= Focus of this pres

- Functions are first class values (integers, strings)
  - Can pass functions as arguments of other functions
  - Can return functions from functions
  - Store functions in variables
- Operations of a program should map input values to output values rather than change data in place
  - Should not have side effects

“The main force behind design of Scala is unification of the object-oriented and functional paradigms: Every function in Scala is a value, every value is an object - hence every function in Scala is an object.”

Scala Introduction

- Hello world
- Types of variables
- Input / Output

Hello world

Compiled:

Hello.scala

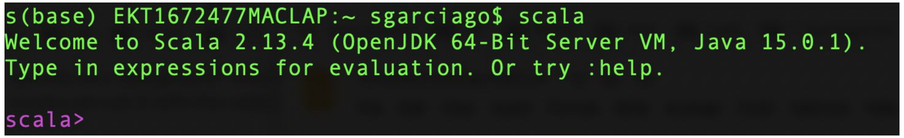
```
object Hello {
  def main(args: Array[String]) = {
    println("Hello, world")
  }
}
```

1) You compile this file with scala (equivalent to javac)
   a) Scalac Hello.scala
   b) It creates two files:
      i) Hello$.class
      ii) Hello.class
   c) Run with scala Hello

1. If defines a method named **main** inside Scala object named **Hello**
2. An **object is similar to a class**, but use it when you want a single instance of that class (static)
3. Array wraps Java array primitive
4. We can access to Command-line arguments with the variable args (it is an Array)

REPL (Read-Eval-Print-Loop):

You can run scala in the terminal and start a REPL session:



```
s(base) EKT1672477MACLAP:~ sgarciago$ scala
Welcome to Scala 2.13.4 (OpenJDK 64-Bit Server VM, Java 15.0.1).
Type in expressions for evaluation. Or try :help.

scala>
```

It is a command-line interpreter, so you can type something and the expression will be evaluated.

No semicolons!

There could be semicolons

Object vs Class

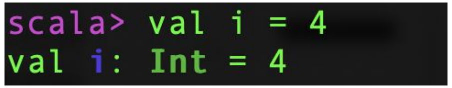
What's the difference between object and class?

In brief:

- **Class** defines a Class as Usual in Java
- **Object**: restricts the instantiation of a class to a single instance of an anonymous class. It can be used to hold static members that are not associated with instances of some class

Two Types of Variables (val and var)

val: Creates a immutable variable, like final in Java



```
scala> val i = 4
val i: Int = 4
```

Pros and Cons of using val

Pros:

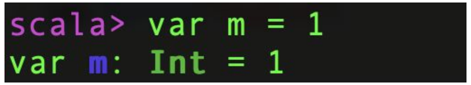
- Data do not change without notice
- Code easier to reason
- Fewer tests

Cons

- Data copying (more memory)

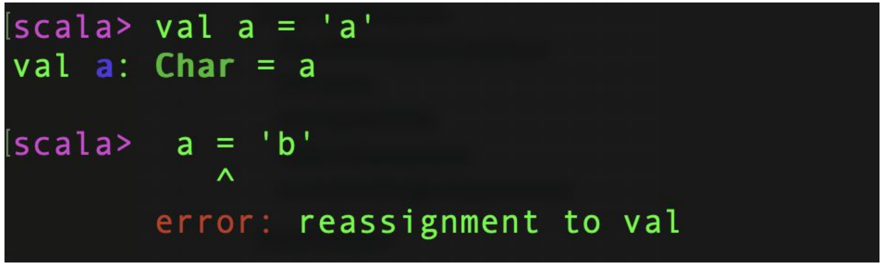
Scala General Rule: you should always use a val, unless there is a good reason not to. (Actually in functional programming all fields are immutable)

Var: creates a mutable variable and should only be used when there is a specific reason to use it



```
scala> var m = 1
var m: Int = 1
```

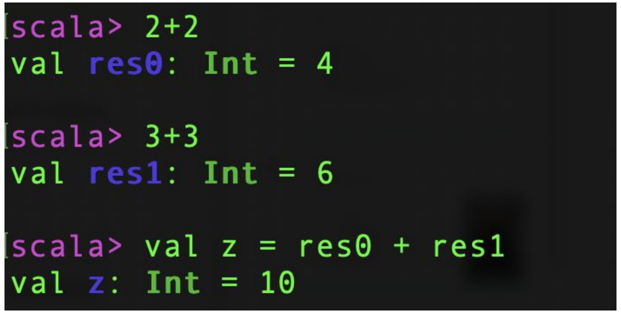
What if trying to reassign?



```
scala> val a = 'a'
val a: Char = a

scala>  a = 'b'
          ^
       error: reassignment to val
```

Note:

What if we just want to make a sum without assigning to variables?



```
scala> 2+2
val res0: Int = 4

scala> 3+3
val res1: Int = 6

scala> val z = res0 + res1
val z: Int = 10
```

It creates a variable call res0, res1 consecutively that can be used!

Scala variable types

Scala package automatically loaded:

Double: 64 bit IEEE 754 single precision float
Float: 32 bit IEEE 754 single precision float
Long: (64 bit signed 2^63 to 2^63-1)
Int: (32 bit signed -2,147,483,648 to 2,147,483,647)
Short: (16 bit signed -32,768 to 2,767)
Byte: (8 bit signed: -128 to 127)
Char: 16 bit unsigned Unicode character (single quote)
Boolean: True/False
String: Sequence of char (double quoted)
Unit: Is used as a return statement for a function when no value is to be returned. (Can be compared to void data type of Java). Subclass of trait

For large numbers Scala also includes the types BigInt and BigDecimal:
BigInt
BigDecimal
These two support all the operators that are used with numeric types

Note that if we declare

```
val b = 1 -> defaults to Int
val b = 1.0  -> defaults to double
```

If you want to assign to specific type, it must be explicitly

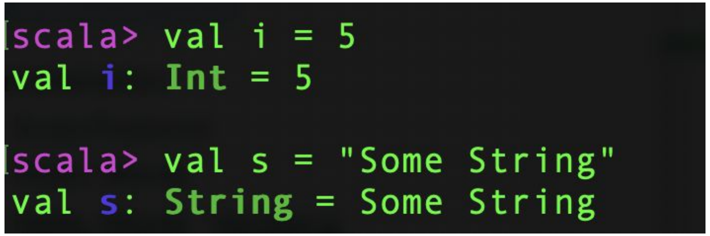
```
val b: Byte = 1
val x: Int = 1
val l: Long = 1
val s: Short = 1
val d: Double = 1.0
val f: Float = 1.0
```

String Concatenation:

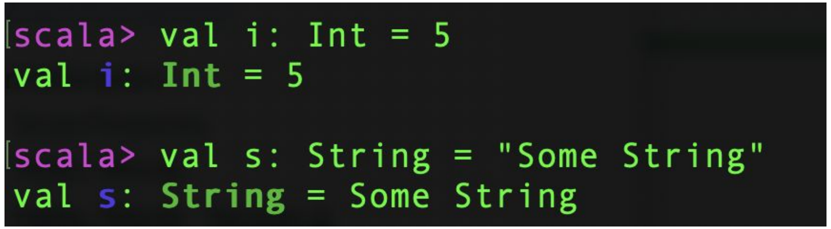
```
val firstName = "Salvador"
val lastName = "Garcia”

val name = firstName + “ “ + lastName
val name = s”$firstName $lastName”
```

Declaring variable types

In scala you typically create variables without declaring their type. Scala can usually infer the data type for you. This is known as type inference as it's great way to keep the code concise. Also, you can declare variable type explicitly, but it is not necessary



```
scala> val i = 5
val i: Int = 5

scala> val s = "Some String"
val s: String = Some String
```



```
scala> val i: Int = 5
val i: Int = 5

scala> val s: String = "Some String"
val s: String = Some String
```

Use explicit form when you need to be clear

Writing Output

We can write output using Standard out (STDOUT) using println:

It adds a newline character after the string

```
println(“output using STDOUT”):
```

Does not add the newline character

```
print(“output using STDOUT”):
```

In addition we can write output to standard error (STDERR):

It adds a newline character after the string

```
System.err.println(“an error occurs”)
```

Reading Input

Several ways to read command-line input:

readLine:

```
import scala.io.StdIn.readLine
object HelloInteractive extends App {
        print(“Enter your name: ”)
        val name = readLine()
        print(“Enter your age”)
        val age = readLine()
        println(s”Your name is: $name $age”)
}
```

Scala Control Structures

- If-else
- Match
- Try-catch
- For loops
- While, do-while

Control Structures:

if/then/else

The conditional control structure is similar to other languages:

```
if (condition1) {
    doSomethingA()
} else if (condition2) {
    doSomethingB()
} else {
    doSomethingElse()
}
```

The if/else constructor always returns a value, so you can use it as a ternary operator:+

```
val x = if (a < b) a else b
```

A common approach in functional programming is to assign the result to a variable.

Match

Scala has a match expression, that can be used with any type. For example:

```
val booleanAsString = bool match {
    case true => "true"
    case false => "false"
}
```

Match is a powerful feature of scala.

Try/catch/finally

It is similar to Java, but the syntax is consistent with match expressions.

```
try {
    writeToFile(text)
} catch {
    case fnfe: FileNotFoundException => println(fnfe)
    case ioe: IOException => println(ioe)
}
```

For loops

```
for (arg <- args) println(arg)
```

Common for loop:

```
for (i <- 0 to 5) println(i)
```

We can add a step size in the for:

```
for (i <- 0 to 10 by 2) println(i)
```

We can iterate over a list of strings:

```
val fruits = List("apple", "banana", "lime", "orange")

val fruitLengths = for {
    f <- fruits
    if f.length > 4
} yield f.length
```

While and do/while

Scala has while and do while statements:

Common while loop:

```
while(condition) {
   statement(a)
}
```

Common do while loop

```
do {
  statement(a)
  statement(b)
}
while(condition)
```

Scala as functional programming

- Introduction to functional programming
- Pure functions
    - Examples of pure functions
    - Examples of “impure” functions

Functional programming

In brief:

In Functional Programming we would like to see the code in a “math way”.

- The combination of functions as a series of algebraic equations.

We would like to use only pure functions and immutable values, because that is what you use in algebra and other forms of math.

FUNCTIONAL PROGRAMMING
A SIMPLE STEP-BY-STEP APPROACH TO LEARNING FUNCTIONAL PROGRAMMING
SIMPLIFIED
ALVIN ALEXANDER

Pure Functions

Ability to write pure functions:

- The function output depends only on its input variables
- It does not mutate any hidden state
- It does not have any back doors:
    - It does not read data from the outside world (including console, web services, databases, etc)
    - It does not write data to the outside world

As a result of this definition, any time you call a pure function with the same input value you will always get the same result. For example, you can call a double functions an infinite number of times with the input value 2, and you will always get the same result.

Examples of Pure Functions

Given that definition of pure functions, as you might imagine, methods like these in the scala.math package are pure functions:

- Abs
- Ceil
- Max
- Min

These Scala String methods are also pure functions:

- isEmpty
- Length
- Substring

Many methods on the Scala collections classes also work as pure functions, includin:

- Drop
- Filter
- Map

Examples of “imPure” Functions

An application is not very useful if it can not read or write. So as suggestion:

- The application is written using pure functions
- Write an “impure” wrapper around that to read/write.

There are some ways to make “impure” a little bit more pure.

For example:

- IO Monad for dealing with user input, files, networks and databases.

https://medium.com/walmartglobaltech/understanding-io-monad-in-scala-b495ca572174#:~:text=IO%20Monad%20is%20simply%20a,before%20it%20actually%20gets%20evaluated.

In the end, Functional Programming applications have a core of pure functions combined with other functions to interact with the outside world.

But before continue...

A little bit of Scala OOP

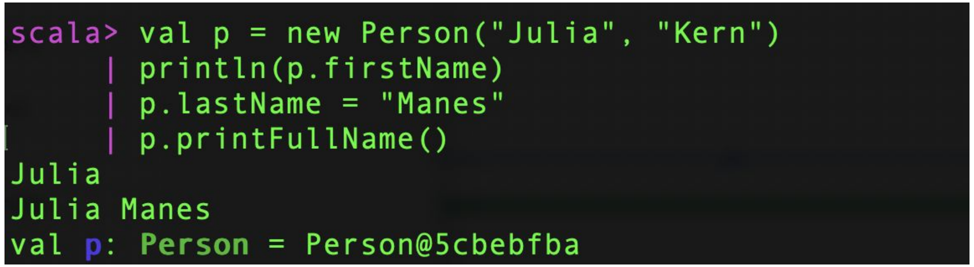
- Scala Classes
- Scala Methods
- Scala Traits
- For loops
- Scala Collection classes

Scala Classes

Examples class are similar to Java:

Person class recibes as input the first name and last name and has one method that print full name

```
class Person(var firstName: String, var lastName: String) {
  def printFullName() = println(s"$firstName $lastName")
}
```



```
scala> val p = new Person("Julia", "Kern")
     | println(p.firstName)
     | p.lastName = "Manes"
     | p.printFullName()
Julia
Julia Manes
val p: Person = Person@5cbebfba
```

As notices it is easy to access the fields in the class

Scala Methods

Using the expression def we can create functions:

Def Declaration:
The return type is optional, you can write it or omit it

```
def sum(a: Int, b: Int): Int = a + b
def concatenate(s1: String, s2: String): String = s1 + s2
```

Def Invocation:

```
val x = sum(1, 2)
val y = concatenate("foo", "bar")
```

Traits

Traits allows to break the code into small, modular utis. For example:

```
Trait speaker has 1 def: speak()
trait Speaker {
    def speak(): String
}
Trait TailWagger has 2 def: startTail(), stopTail()
trait TailWagger {
    def startTail(): Unit = println("tail is wagging")
    def stopTail(): Unit = println("tail is stopped")
}
Trait Runner  has 2 def: startRunning(), stopRunning()
trait Runner {
    def startRunning(): Unit = println("I’m running")
    def stopRunning(): Unit = println("Stopped running")
}
```

Traits (extends and override)

We can **extends** all those traits:

```
Trait speaker has 1 def: speak()
class Dog(name: String) extends Speaker with TailWagger with Runner {
    def speak(): String = "Woof!"
}
```

We can **override** multiple trait methods:

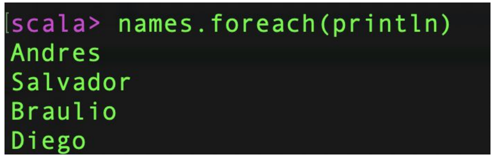
```
class Cat extends Speaker with TailWagger with Runner {
    def speak(): String = "Meow"
    override def startRunning(): Unit = println("Yeah ... I don’t run")
    override def stopRunning(): Unit = println("No need to stop")
}
```

Collections Classes

Lists, there are many ways to populate lists:

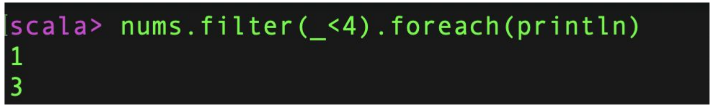
```
val nums = List.range(0, 10)
val nums = (1 to 10 by 2).toList
val letters = ('a' to 'f').toList
val letters = ('a' to 'f' by 2).toList
val names = List(“Andres”, “Salvador”, “Braulio”, “Diego”)
```

Foreach method:



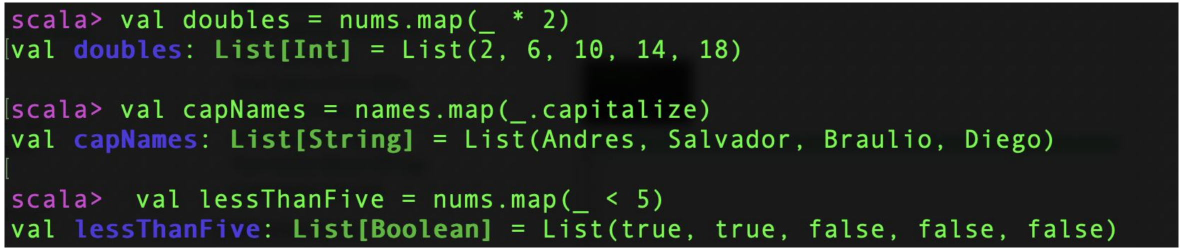
```
[scala> names.foreach(println)
Andres
Salvador
Braulio
Diego
```

Filter method:



```
[scala> nums.filter(_<4).foreach(println)
1
3
```

Map method:

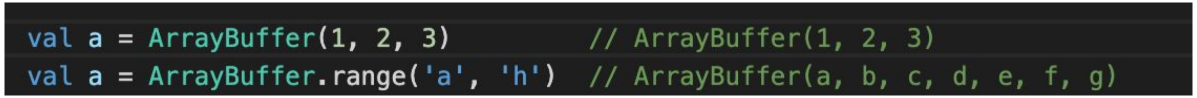


```
scala> val doubles = nums.map(_ * 2)
[val doubles: List[Int] = List(2, 6, 10, 14, 18)

[scala> val capNames = names.map(_.capitalize)
val capNames: List[String] = List(Andres, Salvador, Braulio, Diego)
[
scala>  val lessThanFive = nums.map(_ < 5)
val lessThanFive: List[Boolean] = List(true, true, false, false, false)
```

ArrayBuffer

ArrayBuffers produces mutable sequences that are indexable:

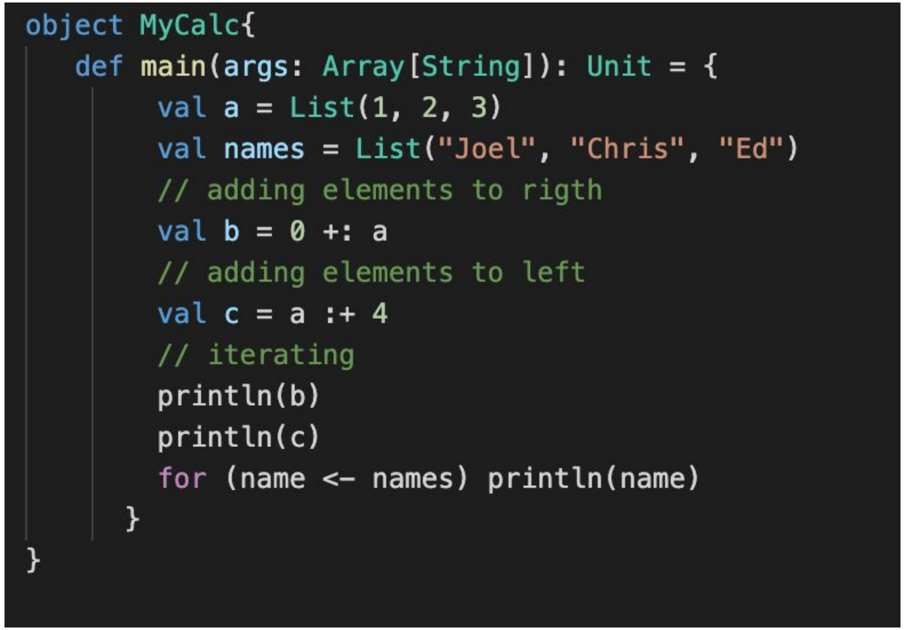
.append: append given element to buffer
.clear: clear content of arrayBuffer (does not resize)
.insert: inserts new element at a given index
.insertAll: insert new elements at a given index
.prepend: prepend a single element at the front
.prependAll: prepend multiple elements at the front
.remove: remove element at given index



```
val a = ArrayBuffer(1, 2, 3)          // ArrayBuffer(1, 2, 3)
val a = ArrayBuffer.range('a', 'h')   // ArrayBuffer(a, b, c, d, e, f, g)
```

Lists

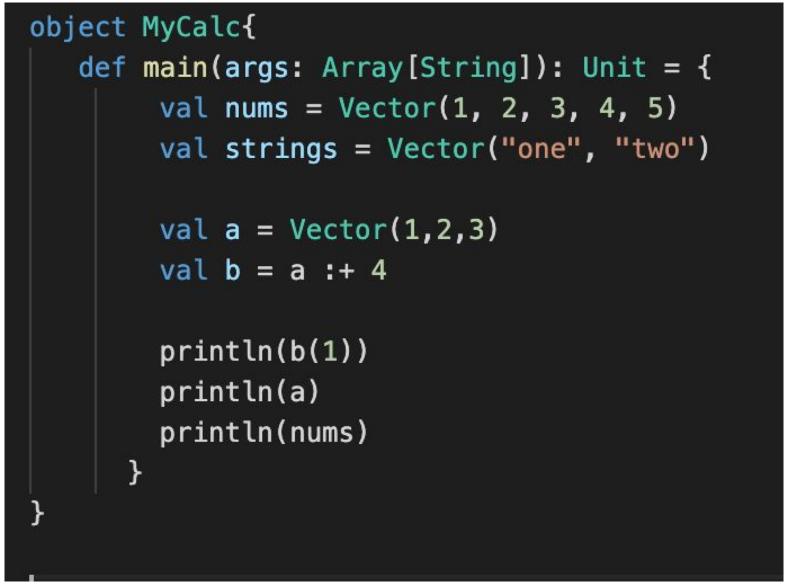
Immutable sequence. That is a linked list that you can not modify. If you want to add or remove elements, you create a new List



```scala
object MyCalc{
   def main(args: Array[String]): Unit = {
       val a = List(1, 2, 3)
       val names = List("Joel", "Chris", "Ed")
       // adding elements to rigth
       val b = 0 +: a
       // adding elements to left
       val c = a :+ 4
       // iterating
       println(b)
       println(c)
       for (name <- names) println(name)
     }
}
```

Vector

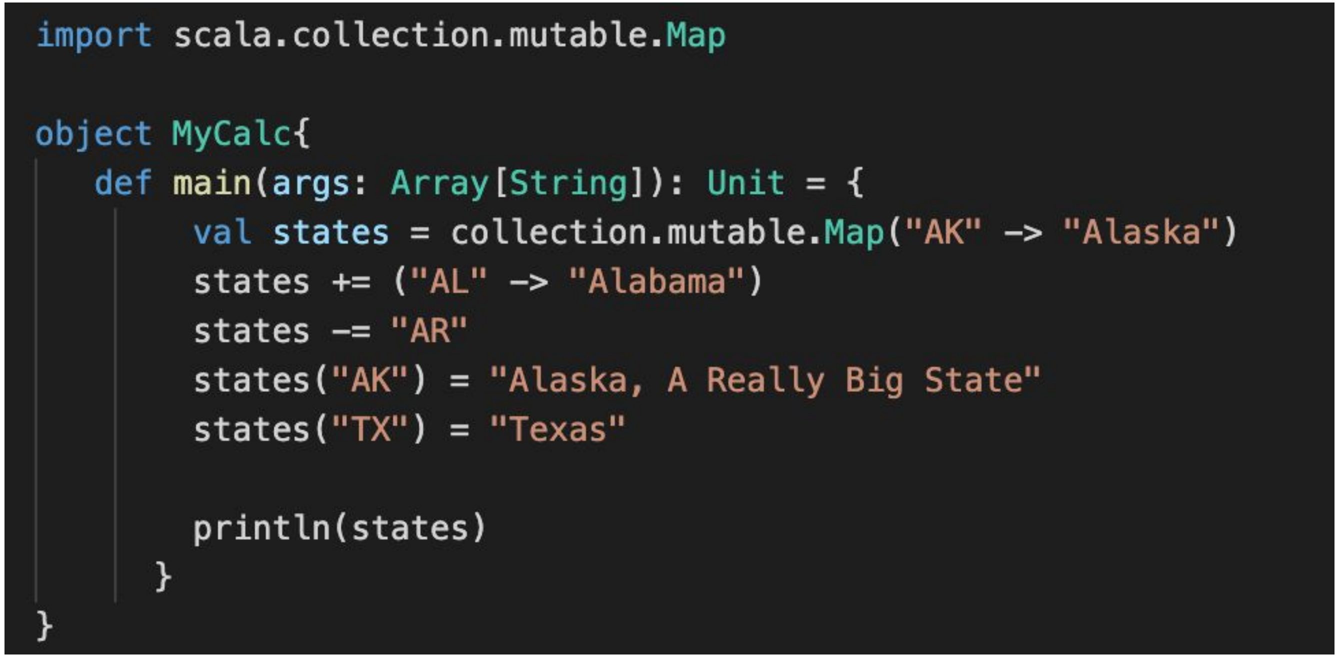
Vector is indexed and also inmutable. You can access elements rapidly by their index value. Is different of List that Vector is indexed



```scala
object MyCalc{
    def main(args: Array[String]): Unit = {
        val nums = Vector(1, 2, 3, 4, 5)
        val strings = Vector("one", "two")

        val a = Vector(1,2,3)
        val b = a :+ 4

        println(b(1))
        println(a)
        println(nums)
      }
}
```

Map

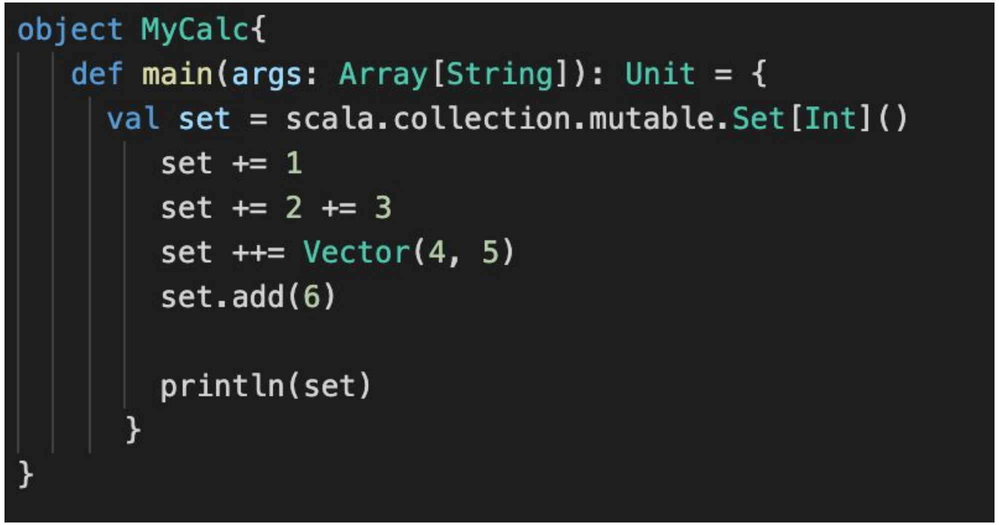
Map is a iterable sequence that consists of keys and values. Is indexable and mutable (also has an immutable version)



```scala
import scala.collection.mutable.Map

object MyCalc{
   def main(args: Array[String]): Unit = {
        val states = collection.mutable.Map("AK" -> "Alaska")
        states += ("AL" -> "Alabama")
        states -= "AR"
        states("AK") = "Alaska, A Really Big State"
        states("TX") = "Texas"

        println(states)
      }
}
```

Set

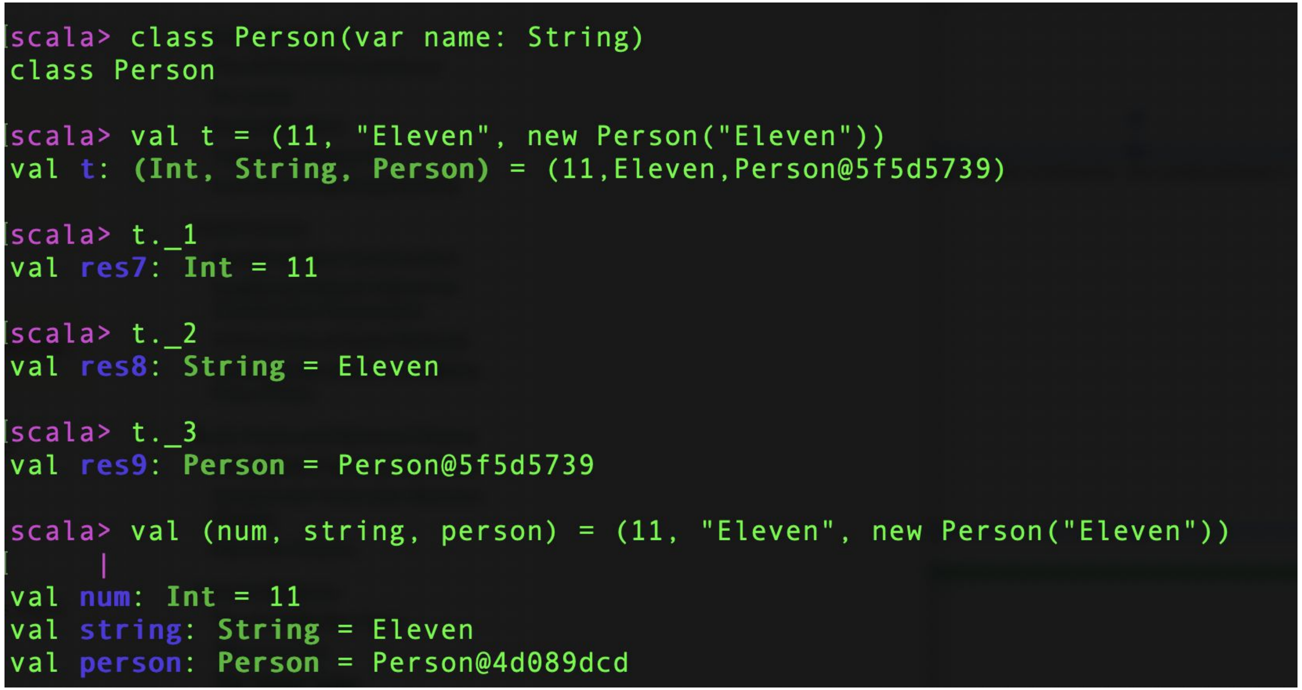
Sets are iterables and contain no duplicate elements.



```
object MyCalc{
    def main(args: Array[String]): Unit = {
      val set = scala.collection.mutable.Set[Int]()
        set += 1
        set += 2 += 3
        set ++= Vector(4, 5)
        set.add(6)

        println(set)
      }
}
```

Tuples

Tuples allows to use a heterogeneous collection of elements. Can contain between 2 -22 values (all can be different types)



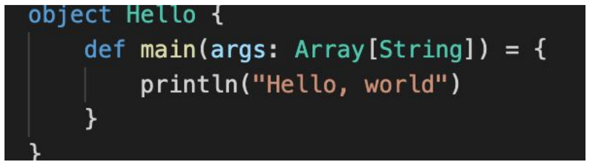
```
scala> class Person(var name: String)
class Person

scala> val t = (11, "Eleven", new Person("Eleven"))
val t: (Int, String, Person) = (11,Eleven,Person@5f5d5739)

scala> t._1
val res7: Int = 11

scala> t._2
val res8: String = Eleven

scala> t._3
val res9: Person = Person@5f5d5739

scala> val (num, string, person) = (11, "Eleven", new Person("Eleven"))
     |
val num: Int = 11
val string: String = Eleven
val person: Person = Person@4d089dcd
```

Scala as functional programming

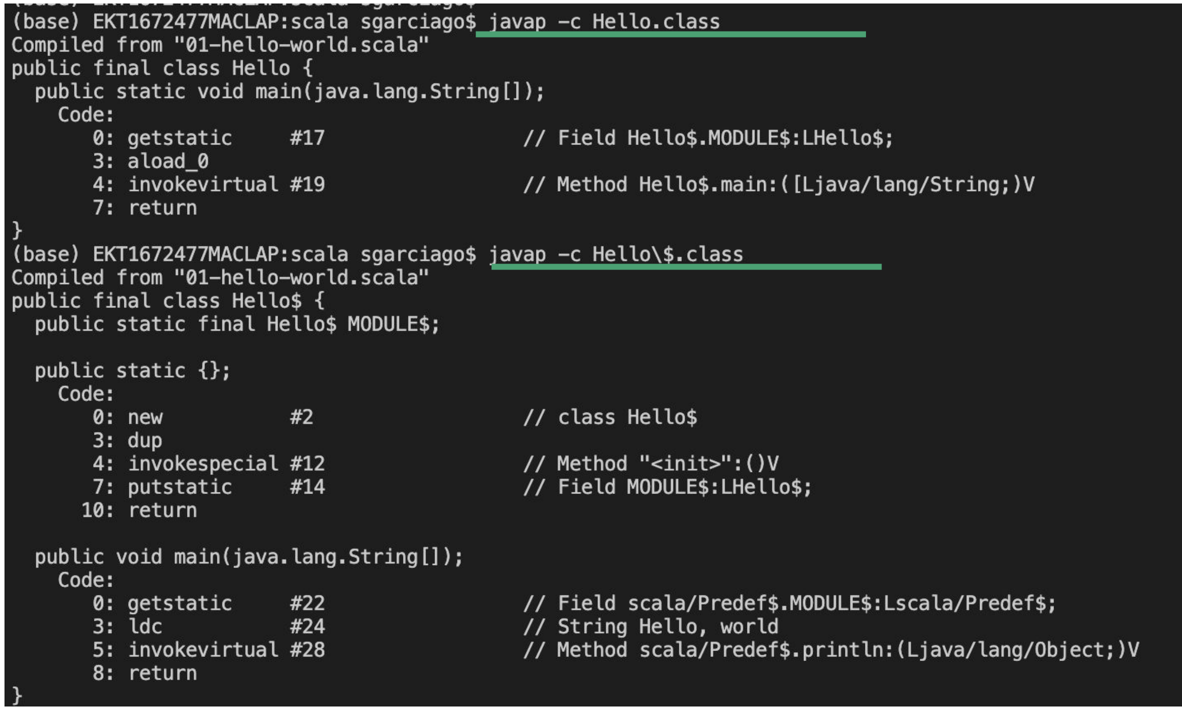
- Scala style guide

Last class doubts

Why creates multiple files (for instance: Hello.class and Hello$.class



```scala
object Hello {
    def main(args: Array[String]) = {
        println("Hello, world")
    }
}
```



```
(base) EKT1672477MACLAP:scala sgarciago$ javap -c Hello.class
Compiled from "01-hello-world.scala"
public final class Hello {
  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #17                 // Field Hello$.MODULE$:LHello$;
       3: aload_0
       4: invokevirtual #19                 // Method Hello$.main:([Ljava/lang/String;)V
       7: return
}
(base) EKT1672477MACLAP:scala sgarciago$ javap -c Hello\$.class
Compiled from "01-hello-world.scala"
public final class Hello$ {
  public static final Hello$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class Hello$
       3: dup
       4: invokespecial #12                 // Method "<init>":()V
       7: putstatic     #14                 // Field MODULE$:LHello$;
      10: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #22                 // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: ldc           #24                 // String Hello, world
       5: invokevirtual #28                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
       8: return
}
```

Quick Scala style guide hints

- If you code properly the code, then it is more readable
- In functional programing, it is suggested to avoid cast and type tests (you should not from beginning the type of the variable)
- Be careful of the line length and whitespaces for readability
- Use local values to simplify complex expressions
- Choose meaningful names for methods and values
- Avoid unnecessary invocations of computational expensive methods
- Do not copy paste code (you should think in parameterization instead in just copy and paste)
- Scala does not require semicolons (only when writing multiple statements on the same line)
- Do deliver final code with print lines
- Avoid mutable local variables
- Avoid redundant if statements

Scala style guide (Indentation and naming conventions)

Indentation:

- **Indent with 2 spaces**: contrary to other languages tabs are not used:
- **Line length:** Usually 80 characters, if not desirable, then you can use multiple lines and 2 space indentation
  - Methods with numerous arguments (in the range of 5 or more) use multiple lines

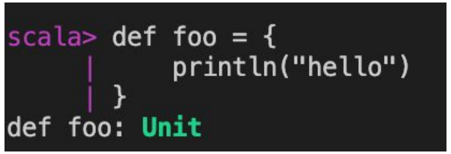
Naming conventions:

- **Classes and traits:** Classes should be named in *upper camel case*
- **Objects:** Object names are like class names *(upper camel case).*
- **Package:** Follow the Java package naming conventions: *for instance: package com.novell.coolness*
- **Methods:** Textual (alphabetic) names for methods should be in *lower camel case*
- **Constant, variables:** Constant names should be in *upper camel case*.
- **Type parameters:** a single *upper-case letter* (from the English alphabet) should be used, starting with A

Scala style guide (Indentation and naming conventions)

Curly Braces {}:

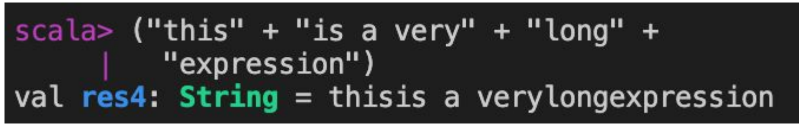
- Curly braces { must be on the same line as the declaration they represent.



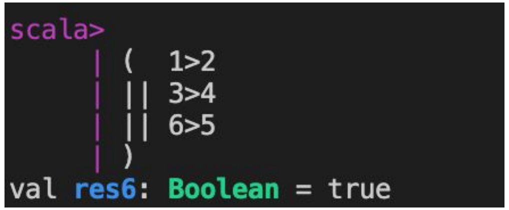
```
scala> def foo = {
     |     println("hello")
     | }
def foo: Unit
```

Parenthesis ():

- When blocks wrap access lines, opening and closing should be unspaced and on the same lines of content
- Also serves to disable semicolon inference and start lines with operators



```
scala> ("this" + "is a very" + "long" +
     |    "expression")
val res4: String = thisis a verylongexpression
```

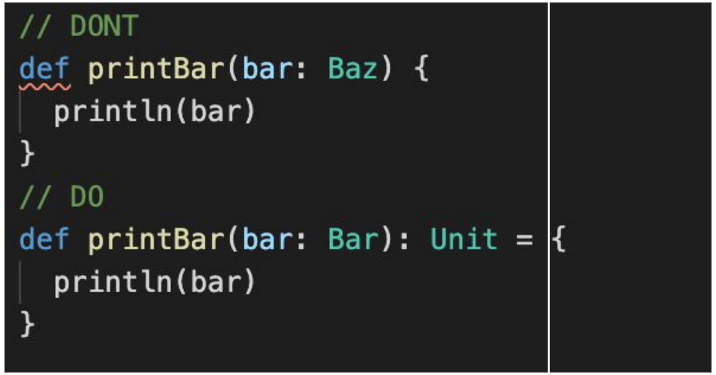


```
scala>
     | (  1>2
     | || 3>4
     | || 6>5
     | )
val res6: Boolean = true
```

Scala style guide (methods)

Methods:

- **def foo**(bar: **Baz**): **Bin** = expr
  - Common pattern
- **def foo**(x: **Int** = 6, y: **Int** = 7): **Int** = x + y
  - If want a default value, can be specified
  - You should specify a return type for all public members
- **private def foo**(x: **Int** = 6, y: **Int** = 7) = x + y
  - Local methods or private methods may omit their return type



```
// DONT
def printBar(bar: Baz) {
  println(bar)
}
// DO
def printBar(bar: Bar): Unit = {
  println(bar)
}
```

Body:

Body with less than 30 characters should be given on a single line: (if it is more, but can be in one line, can be in next line:)

- **def add**(a: **Int**, b: **Int**): **Int** = a + b
- **def sum**(ls: **List**[**String**]): **Int** =
-   ls.map(_.toInt).foldLeft(0)(_ + _)

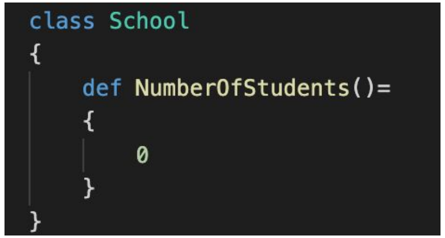
Otherwise must be with braces

More on scala style:
https://docs.scala-lang.org/style/scaladoc.html#classes-objects-and-traits

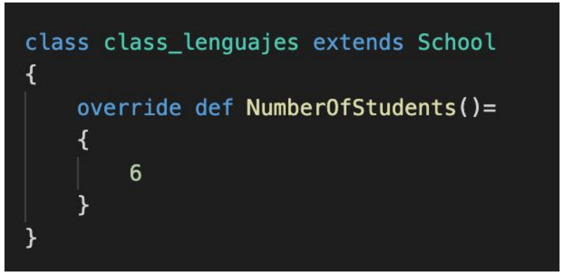
Modifiers: overriding

Overriding is identical to overriding in Java. In scala methods, vars and vals can be overwritten.

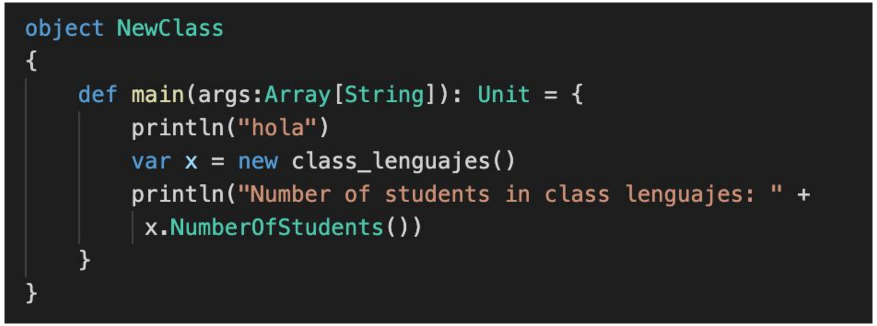
Method Overriding: If a subclass has the method name identical to the method name in the parent class and changes its features



```scala
class School
{
    def NumberOfStudents()=
    {
        0
    }
}
```



```scala
class class_lenguajes extends School
{
    override def NumberOfStudents()=
    {
        6
    }
}
```



```scala
object NewClass
{
    def main(args:Array[String]): Unit = {
        println("hola")
        var x = new class_lenguajes()
        println("Number of students in class lenguajes: " +
         x.NumberOfStudents())
    }
}
```

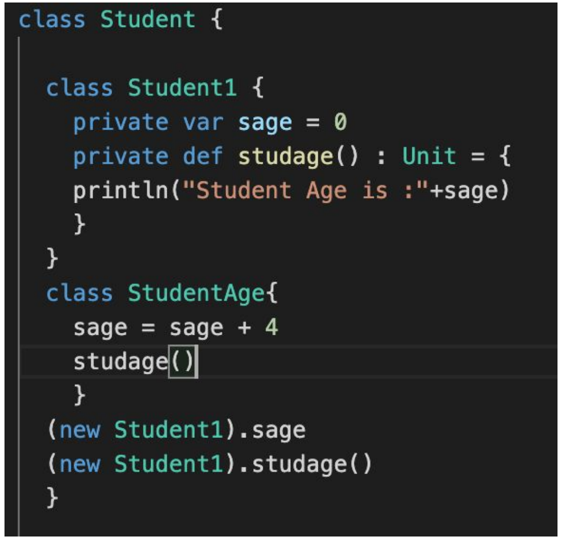
Modifiers: access

Private, public and protected are class modifiers used in scala.

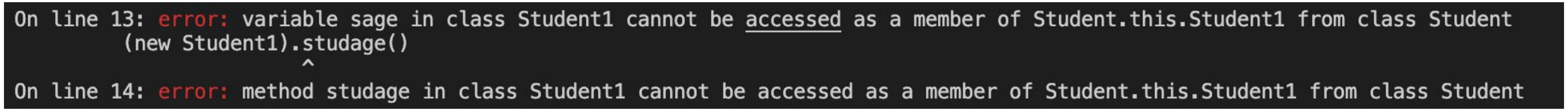
Private members: only available inside the class or to the object

Protected members: can be accessed from subclasses of the class

Public member: The default access modifier is public.



```
class Student {

  class Student1 {
    private var sage = 0
    private def studage() : Unit = {
    println("Student Age is :"+sage)
    }
  }
  class StudentAge{
    sage = sage + 4
    studage()
    }
  (new Student1).sage
  (new Student1).studage()
  }
```



```
On line 13: error: variable sage in class Student1 cannot be accessed as a member of Student.this.Student1 from class Student
         (new Student1).studage()
                        ^
On line 14: error: method studage in class Student1 cannot be accessed as a member of Student.this.Student1 from class Student
```

Functional programming & substitution model

Strictly talking functional programming means no mutable variables, assignments, loops and other imperative control structures. In a broad sense: focusing on functions. From previous slides we

Functions can be values that are produced, consumed and composed:

- Functions are first class values (integers, strings)

Scala model of expression evaluation is based on the **principle of substitution model**:

- Variable names are replaced by the values they are bound to
- All evaluation reduce an expression to a value, which is a term that does not need further evaluation.
- Is formalized in the lambda-calculuus which gives foundation for functional programming

Functional programming & substitution model

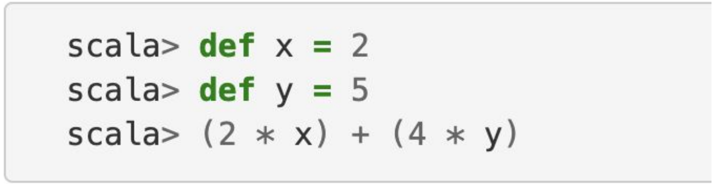
Program expressions are evaluated in the same way we would evaluate a mathematical expressión:

(2*2)+(3*4) will evaluate first 2*2 and 3*4 and finally 4 + 12. Scala works whis way

Scala non primitive expression is evaluated as follows:

- Take the operator with highest precedence
- Evaluate its operands from left to right
- Apply the operator to operand values

During the process Scala evaluator rewrites the program expression to another expression.



```
scala> def x = 2
scala> def y = 5
scala> (2 * x) + (4 * y)
```

$\rightarrow (2 * 2) + (4 * y)$

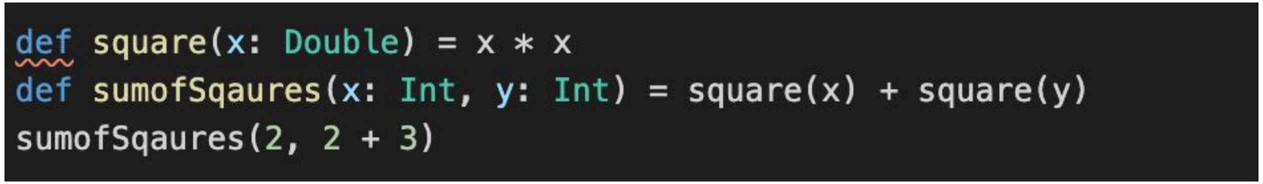
$\rightarrow 4 + (4 * 5)$

$\rightarrow 4 + 20$

$\rightarrow 24$

Function evaluation strategy

There are two evaluation strategy for function with parameters namely call-by-value and call-by-name. For expressions that only use pure function and can be reduced with substitution mode, both yields to same final results:



```
def square(x: Double) = x * x
def sumofSqaures(x: Int, y: Int) = square(x) + square(y)
sumofSqaures(2, 2 + 3)
```

Call-by-name: avoid evaluation of parameters if not used in the function body. Example

sumofSquares(2,5)
square(2)+square(2+3)
2*2 + square(2+3)
4+(2+3)*(2+3)
4+5+(2*3)
4+5*5
4+25
29

Call-by-value: evaluates every function argument only once: avoid the repeated evaluation of arguments. Example

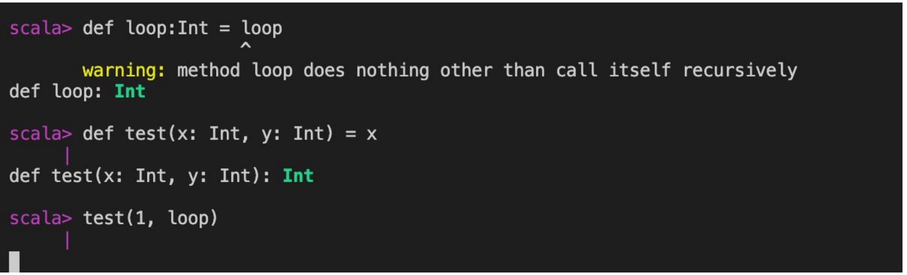
sumofSquares(2,5)
square(2)+square(5)
2*2 + 5*5
4+25

Function evaluation strategy

Call by value is more efficient. If call by value evaluation terminates then call by name also terminate. **Call by value** might loop infinitely but call by name would terminate:

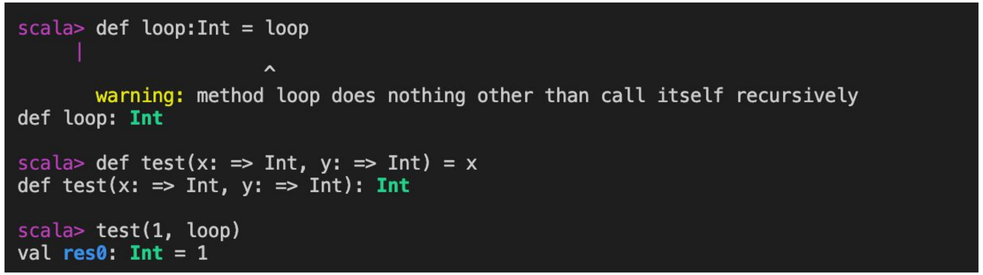
Scala uses **call by value by default** because often is more efficient. We can **enforce use call by name** by preceding the parameters types by => :

Call by value



```
scala> def loop:Int = loop
                         ^
       warning: method loop does nothing other than call itself recursively
def loop: Int

scala> def test(x: Int, y: Int) = x
     |
def test(x: Int, y: Int): Int

scala> test(1, loop)
     |
```

Call by name



```
scala> def loop:Int = loop
     |
                         ^
       warning: method loop does nothing other than call itself recursively
def loop: Int

scala> def test(x: => Int, y: => Int) = x
def test(x: => Int, y: => Int): Int

scala> test(1, loop)
val res0: Int = 1
```

Higher order functions and anonymous functions

As mentioned before,

- Higher order functions let pass functions as arguments and return them as results (first class values) :
- These functions are called higher order functions (opposite to first order functions)

Passing functions as parameters leads to the creation of many small functions:

- It is not necessary to define and name all functions
- We can create functions literals, which let us write a function without giving it a name: these are anonymous functions
- For example:

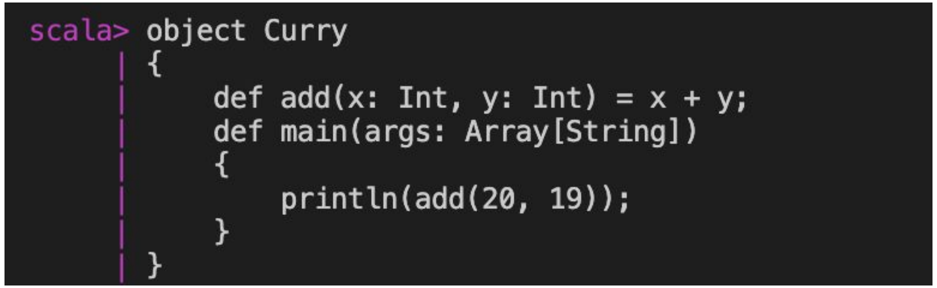
(x: Int) => x*x*x

(x: Int) is the parameter of a function and x*x*x is the body

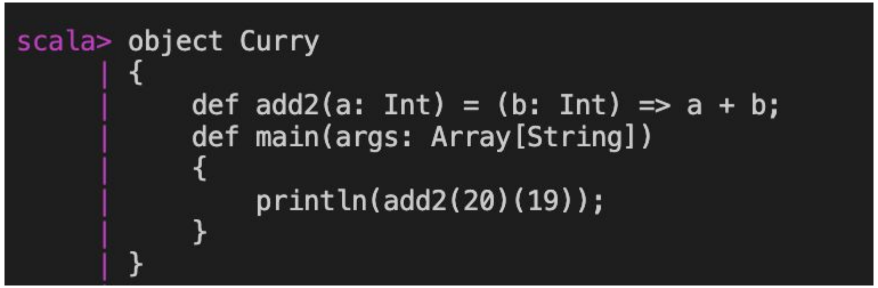
The type of the parameter can be omitted if it can be inferred

Currying

Is a process for transforming a functions. This functions takes multiple arguments into a function that takes single argument:



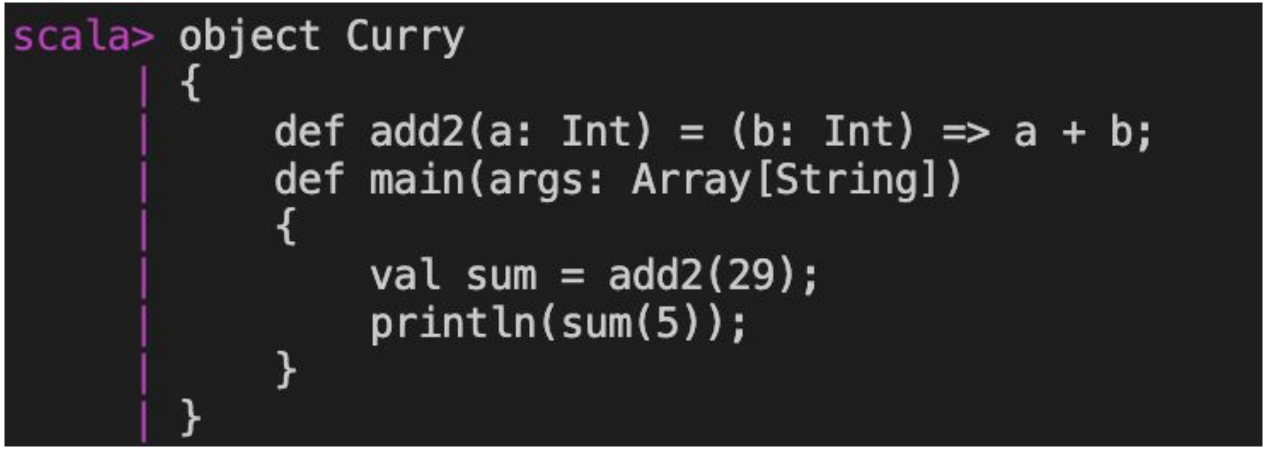
```
scala> object Curry
     | {
     |     def add(x: Int, y: Int) = x + y;
     |     def main(args: Array[String])
     |     {
     |         println(add(20, 19));
     |     }
     | }
```



```
scala> object Curry
     | {
     |     def add2(a: Int) = (b: Int) => a + b;
     |     def main(args: Array[String])
     |     {
     |         println(add2(20)(19));
     |     }
     | }
```

Currying (Partial Application)

Partially applied functions: we can only pass one argument and assign the function to the value, the second argument is passed with the value and these arguments are added and result is printed:



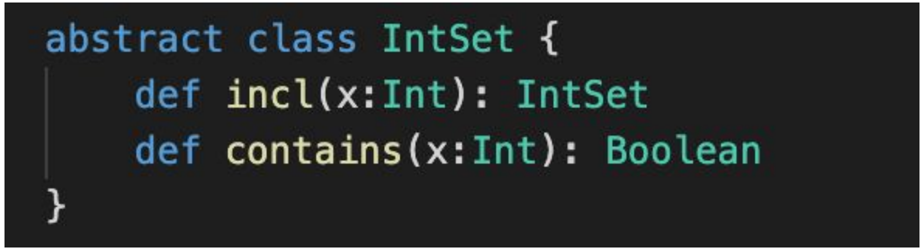
```
scala> object Curry
     | {
     |     def add2(a: Int) = (b: Int) => a + b;
     |     def main(args: Array[String])
     |     {
     |         val sum = add2(29);
     |         println(sum(5));
     |     }
     | }
```

Class hierarchies

An important aspect of class hierarchies is the model of evaluation

The actual method called might depend on the runtime type of t

Abstract Classes:



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

The equal is also missing and the body too. (only permissible in abstract classes) ->

Can contain members which are missing an implementation -> cannot be instantiated