INFO0054-1 Programmation Fonctionnelle

Chapter 01: Introduction to FP (in Scala)

Christophe Debruyne
(c.debruyne@uliege.be)

References

- Chapter 1: What is functional programming?
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

- Chapter 2: Getting started with functional programming in Scala.
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

Overview

- What are programming paradigms?
  - Imperative programming (imperative, procedural, OOP)
  - Declarative programming (logic, functional)
- What is functional programming?
- Why functional programming?
- Why Scala as the programming language?
- Example
  - A Scala program with side effects
  - Removing the side effects
- Pure functions
- Referential Transparency, purity, and the substitution model
- Introducing Scala by example
  - Running Scala programs
  - Modules, object, and namespaces

Part 01
Programming paradigms

What are programming paradigms?

- Paradigm:
  - A typical example or pattern of something; a pattern or model.
  - *Source: Oxford English Dictionary.*

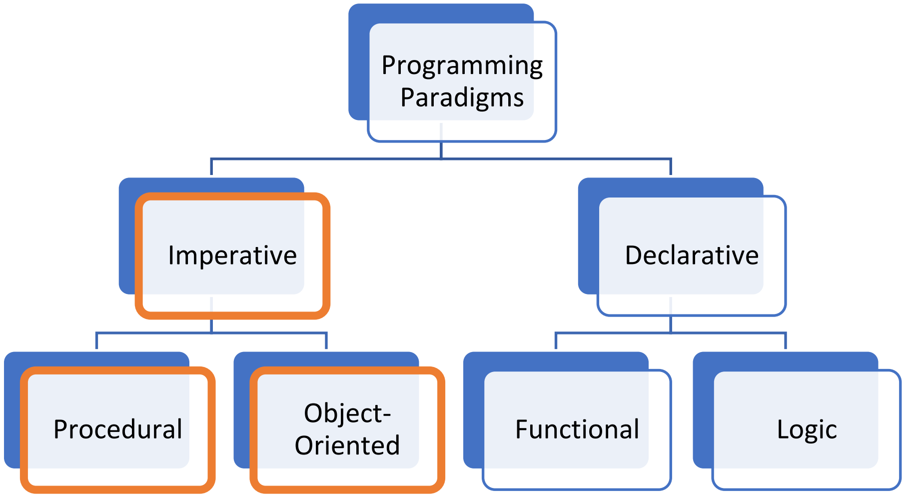
- A **programming paradigm** is an approach to solve problems using a programming language. A programming paradigm provides us a framework for thinking about and solving a problem.

- The word programming paradigm is also used for the process of classifying programming languages based on their features.
  - Programming languages often support several programming paradigms (i.e., they are multi-paradigm programming languages).
  - Programming languages may be designed for specific programming paradigms.

What are programming paradigms?

Imperative programming uses statements that change a program's state and focusses on **how to solve a problem**.

I.e., we tell the computer how to solve the problem.



Imperative programming

- **Imperative programming** also the name of a paradigm that offers little structure and abstraction.
- Imperative programming provides no support for subroutines.
- In imperative programming, the flow is controlled with GOTO statements and branch tables.

```
For examples, look
for calculating the
factorial of 5 in
assembly. :-)
```

Procedural programming

- **Procedural programming** groups instructions into subroutines or **procedures.**
- Procedural programming provides support for **control flow**.

Right: example of procedural programming in Java. Notice the use of procedures. Also notice the side effects on the variables result and n.

```java
package test;

public class Chapter01 {

    private static int ComputeFactorial(int n) {
        int result = 1;
        while(n >= 1) {
            result *= n;
            n--;
        }
        return result;
    }

    public static void main(String[] args) {
        System.out.println(ComputeFactorial(5));
    }
}
```

Object-oriented programming

- In **object-oriented programming (OOP)**, a program is structured into objects with behaviour and data, and objects pass **messages** to one another.

Right: example of OOP in Java. The example is deliberately complicated, but illustrates the point. A new Factorial object is created that manages its own state via methods. Notice that the method compute changes the object's state.

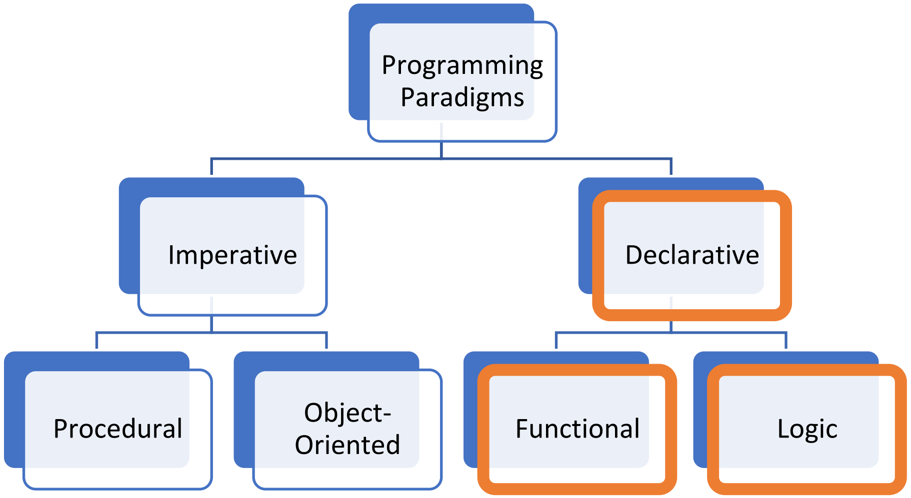
```java
package test;

public class Factorial {

    private int result = 0;

    public void compute(int n) {
        result = 1;
        while(n >= 1) {
            result *= n;
            n--;
        }
    }

    public int getResult() { return result ; }

    public static void main(String[] args) {
        Factorial f = new Factorial();
        f.compute(5);
        System.out.println(f.getResult());
    }

}
```

Covered in the 2nd semester.

What are programming paradigms?

In **declarative programming**, one declares properties of the desired result, but not how to compute it.

I.e., we tell the computer **what to solve** and not how to solve it.



Logic programming

- In **logic programming**, programs are finite sets of **facts** and **rules** that describe our world and problem. We then ask the system for answers.

Right: this is an example of a Logic Program written in Prolog.

- The first statement is a fact that captures the base case.
- The second statement is a rule declaring that the factorial of N is F if N bigger than 0; N1 is N minus 1; the factorial of N1 is F1; and F is N times F1.

```
factorial(0,1).

factorial(N,F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1,F1),
    F is N * F1.
```

We can execute the following query once the program is loaded:

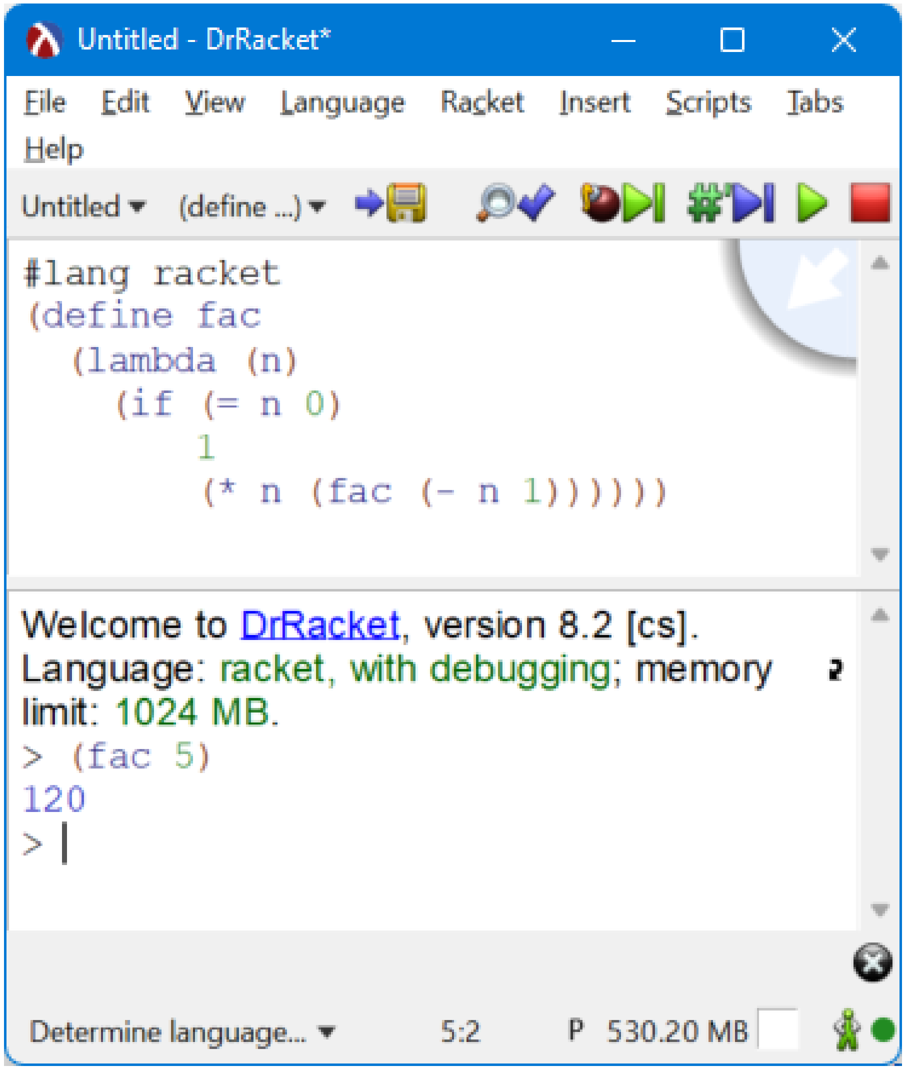
```
?- factorial(5, X).
120
```

Functional programming

- The desired result to a problem is declared as the value of a series of **functions**.
    - The mathematical notion of function.

- (!!!) We can replace function calls with their values and not change the meaning of the program.

Right: example of functional programming in Racket/Scheme.



Part 02
Functional programming

What is functional programming?

- **Functional programming** is constructing programs using (only) **pure functions**.
- Pure functions are functions that have no **side-effects**.
- So, we do no allow:
  - A modification of a variable
  - A modification of data structures in place
  - Throwing of exceptions or halting on an error
  - Printing to the console
  - Reading user input
  - Drawing on a screen
  - …

- **Functional programming is a restriction on how we program write programs, but not on what programs we can express.**
  - Functional programming is a way of programming.
  - So, one can do functional programming in any language that support procedures.
  - Programming languages designed for functional programming provides us features that make our lives easier (e.g., immutable structures, tail-recursion optimization, etc.).

Why functional programming?

- Not only is FP a great exercise for computer scientists to approach problems in a different way, but FP also has many practical advantages and uses.
- Pure functions increase the modularity of your program
  - Pure functions are easier to test
  - Pure functions are less prone to bugs (side-effects are removed)
  - Pure functions are easier to reuse
  - Pure functions are easier to "understand" and "reason about"
    - The output of a function does not depend on some local or global state producing different results at different times.
  - Pure functions are easier to parallelize
  - …
- FP is important for **data engineering** – the process of designing and building data transformation pipelines for data-driven projects (e.g., data science).
  - To read: Functional Data Engineering — a modern paradigm for batch data processing by Maxime Beauchemin. PDF also available on eCampus.

Why Scala?

There are programming languages:

- Designed solely for functional programming (rarely used in industry)

- Designed to support several programming paradigms (often academic)

- Designed primarily for other paradigms, but supporting functional programming: Rust, Java, Scala, Python,... These languages are *used in industry*.

Scala is a language that fairly elegantly combines object-oriented and functional programming.

Why Scala?

Advantages

- Scala is compiled into Java bytecode, which you will learn more about this year.
  - You will learn multiple paradigms in the same "ecosystem."
  - Also runtimes in .NET and JavaScript.
- Incorporates concepts from Scheme, Haskell, and SML.
- Scala is supported by a large community (books, examples, tools,...).

Disadvantages

- Scala supports multiple paradigms and has a more complex syntax compared to languages such as Scheme.
- Scala is a less academic language.
- Optimization of tail recursion in some cases not guaranteed.
  - You can annotate a function to be tail recursive and the compiler will test this.
  - No optimization for mutual recursion yet.

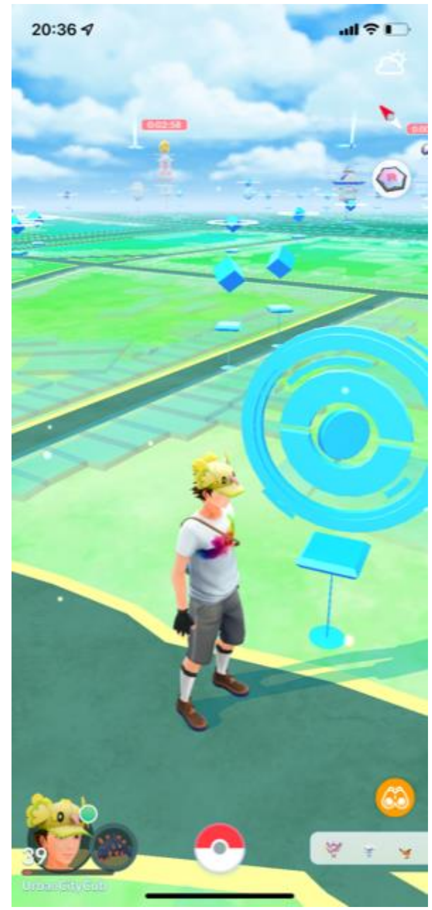
A Scala program **with side effects**

- Let's "develop" a Pokémon GO in Scala!

- One can buy Pokécoins which are charged against your credit card (via your device’s app store platform and app-store account). Pokécoins are used as an in-game currency.
- We want to support multiple purchases and our code needs to be fit for easy testing.

Disclaimer: this is by no means an accurate depiction on how the game is implemented. ;-)

In the next few slides, we will introduce some Scala syntax. The goal is to identify problems related to side effects and, step-by-step, transform the program into one that is functional.



A Scala program **with side effects**

```
/* Our first Scala example */
class PokemonGO:
    def buyPouch(a: Account): Pouch =
        val pouch = new Pouch()
        a.charge(pouch.price)
        pouch
        // we could have written:
        // return pouch
```

- Much like Java, the `class` keyword introduces a class.
- A method of a class is introduced by the `def` keyword.

- `buyPouch(a: Account): Pouch`
    - The method `buyPouch` takes as input a parameter `a` that is of the type `Account`.
    - We assume that an `Account` object contains a link with your device's app store platform which is linked to a credit card.
    - The method `buyPouch` returns a value of the type `Pouch`.
- We have 3 statements in the body of the method `buyPouch`. Notice that newlines delimit statements.
- We do not need to use the `return` keyword as the value of the last statement is returned.

A Scala program **with side effects**

```
class PokemonGO:
    def buyPouch(a: Account): Pouch =
        val pouch = new Pouch()
        a.charge(pouch.price)
        pouch
```

What is the problem?

- The second statement is an example of a side effect.
- Our function only returns a `Pouch` of Pokécoins. Should an `Account` object be concerned with contacting a credit card company, authorizing the transaction, or charging the card? No!
- ***All these things happen and change state on the side of our function.***

Consequence: our code is difficult to test. Why? We do not want our tests to contact the app store and charge a credit card.

A Scala program **with side effects**

Let's make `Account` less aware of payments and create a `Payments` object.

```
class PokemonGO:
    def buyPouch(a: Account, p: Payments): Pouch =
        val pouch = new Pouch()
        a.charge(p, pouch.price)
        pouch
```

What is the problem? (1)

While we still have a side-effect, `Payments` can be an interface allowing us to create an object implementing that interface for testing purposes. In order to test buyPouch, we need to create such an "artificial" object. Creating mock frameworks for testing purposes may be overly complex for simple functions such as buying a pouch of coins and creating a charge.

A Scala program **with side effects**

Let's make `Account` less aware of payments and create a `Payments` object.

```
class PokemonGO:
    def buyPouch(a: Account, p: Payments): Pouch =
        val pouch = new Pouch()
        a.charge(p, pouch.price)
        pouch
```

What is the problem? (2)

Reuse. What if we want to purchase multiple pouches? Now we have multiple charges that are processed via the app store. The app store withholds a commission for each transaction, so we may want to group purchases together into one transaction.

- A special class for processing batches of charges is complex.
- Two separate methods without reuse can be questionable.

Removing the side effects

- The function buyPouch returns both a *pouch* and a *charge* using a tuple.

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))
```

- Tuples allow us to group a fixed number of typed elements in an immutable data structure. They are grouped together using parentheses and separated by commas.
    - Elements can be of different types.
    - Immutable means that the values of a tuple cannot be changed.
- *Notice that the creation of a new Charge object does not require the new keyword!*

To read: https://docs.scala-lang.org/tour/tuples.html

Removing the side effects

- Charge is a new data type that we created containing an account and an amount. It also has a method for combining the charges of the same account. Charge is a case class.

```
case class Charge(acc: Account, amount: Double):
    def combine(other: Charge): Charge =
        if (acc == other.acc)
            Charge(acc, amount + other.amount)
        else
            throw new Exception("Cannot combine charges of <> accounts.")
```

If `acc == other.acc` is true, then the value of the last statement of that branch is returned.

- Case classes are special classes that have:
  - Only one parameter list that acts as the constructor.
  - Do not require the keyword new.
  - All parameters are immutable and public.
  - Instances are compared by structure and not by reference.

To read: https://docs.scala-lang.org/tour/case-classes.html

Removing the side effects

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))

    def buyPouches(a: Account, n: Int): (List[Pouch], Charge) =
        val purchases: List[(Pouch, Charge)] = List.fill(n)(buyPouch(a))
        val (pouches, charges) = purchases.unzip
        (pouches, charges.reduce((p1, p2) => p1.combine(p2)))
```

The function `buyPouches` returns a tuple containing a list of pouches and a charge.

Lists are immutably singly linked lists of a particular type. The elements of a `List[Pouch]` are instances of pouches. The elements of a `List[(Pouch, Charge)]` are tuples containing a pouch and a charge.

Removing the side effects

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))

    def buyPouches(a: Account, n: Int): (List[Pouch], Charge) =
        val purchases: List[(Pouch, Charge)] = List.fill(n)(buyPouch(a))
        val (pouches, charges) = purchases.unzip
        (pouches, charges.reduce((p1, p2) => p1.combine(p2)))
```

`List.fill(n)(buyPouch(a))` will create a list of `n` items containing the result of `n` evaluations of the expression `buyPouch(a)`.

```
scala> List.fill(3)(scala.util.Random.nextInt(100))
val res0: List[Int] = List(77, 56, 15)
```

Removing the side effects

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))

    def buyPouches(a: Account, n: Int): (List[Pouch], Charge) =
        val purchases: List[(Pouch, Charge)] = List.fill(n)(buyPouch(a))
        val (pouches, charges) = purchases.unzip
        (pouches, charges.reduce((p1, p2) => p1.combine(p2)))
```

If we apply the function buyPouches with n equal to 5, then the variable purchases will valuations will be a list containing 5 2-tuples $List((p_1, c_1), (p_2, c_2), \ldots, (p_5, c_5))$.

Removing the side effects

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))

    def buyPouches(a: Account, n: Int): (List[Pouch], Charge) =
        val purchases: List[(Pouch, Charge)] = List.fill(n)(buyPouch(a))
        val (pouches, charges) = purchases.unzip
        (pouches, charges.reduce((p1, p2) => p1.combine(p2)))
```

The function `unzip`, which takes no arguments, allows us to split a list of pairs into a pair of lists. If we were to unzip `purchases` from the previous example, then

$$List((p_1, c_1), (p_2, c_2), \dots, (p_5, c_5))$$

would result in

$$(List(p_1, p_2, \dots, p_5), List(c_1, c_2, \dots, c_5))$$

Removing the side effects

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))

    def buyPouches(a: Account, n: Int): (List[Pouch], Charge) =
        val purchases: List[(Pouch, Charge)] = List.fill(n)(buyPouch(a))
        val (pouches, charges) = purchases.unzip
        (pouches, charges.reduce((p1, p2) => p1.combine(p2)))
```

The function `reduce` allows us to reduce a list (or array,…). It takes as input a function that implements a binary operation (that **should** be commutative and associative). The function `reduce` is thus a higher-order function. In this example, we provide an anonymous function that combines charges.

Why should? Because the order in which the elements are processed is random.

Removing the side effects

```
class PokemonGO:
    def buyPouch(a: Account): (Pouch, Charge) =
        val pouch = new Pouch()
        (pouch, Charge(a, pouch.price))

    def buyPouches(a: Account, n: Int): (List[Pouch], Charge) =
        val purchases: List[(Pouch, Charge)] = List.fill(n)(buyPouch(a))
        val (pouches, charges) = purchases.unzip
        (pouches, charges.reduce((p1, p2) => p1.combine(p2)))
```

Improvements:

- Reuse: buyPouches is written in terms of buyPouch.
- Testing: functions are easy to test.
- Testing: our PokemonGO does not "know" how charges are processed.

A more challenging example

```
class PokemonGO:
    ...
    def coalesce(charges: List[Charge]): List[Charge] =
        charges.groupBy(_.acc).values.map(_.reduce(_ combine _)).toList

----------
scala> val list = List(Charge(a, 10), Charge(a, 5), Charge(b, 10),
                       Charge(c, 5), Charge(b, 10), Charge(a, 5))
val list: List[Charge] = List(Charge(rs$line$1$Account@790d629a,10.0),
                              Charge(rs$line$1$Account@790d629a,5.0),
                              Charge(rs$line$1$Account@2b409174,10.0),
                              Charge(rs$line$1$Account@11939a9f,5.0),
                              Charge(rs$line$1$Account@2b409174,10.0),
                              Charge(rs$line$1$Account@790d629a,5.0))

scala> game.coalesce(list)
val res0: List[Charge] = List(Charge(rs$line$1$Account@11939a9f,5.0),
                              Charge(rs$line$1$Account@2b409174,20.0),
                              Charge(rs$line$1$Account@790d629a,20.0))
```

A more challenging example

```
def coalesce(charges: List[Charge]): List[Charge] =
    charges.groupBy(_.acc).values.map(_.reduce(_ combine _)).toList
```

Anonymous functions galore!

- `groupBy(_.acc)` groups the list of charges by account resulting in a map from accounts (*keys*) to lists of charges (*values*). `groupBy` takes as input a function that computes a value for each element. In this example, a function that retrieves the account of a charge.
- `values` takes the values of a map as a collection.
- `map` allows to apply a function to each value in `values`.
- We provide `_.reduce(_ combine _)` as the function. Each element in that collection is a list, so we can apply the method `reduce` on each list.
- `_ combine _` is a shorter way to write `(p1, p2) => p1.combine(p2)`.
- Lastly, the collection is transformed into a list with `toList`.

Removing the side effects

- So far, we <u>moved side-effects to outer layers of our program</u>.

- In this course, we will learn how we can apply to "transformation" process to any function. In other words, we will try to find some functional analogue.

- In practice, we will implement a:
  - A pure "core" of the programming (the actual functional programming)
  - A thing layer of code for handling side-effects
    - Writing to files
    - Printing results on a screen
    - Exception handling
    - …

Part 03
Pure functions

Pure functions

- A pure function is
  - "a function f with input type A and output type B is a computation that relates every value a of type A to exactly one value b of type B so that b is determined solely by the value of a."
  - Any internal or external change of state is irrelevant, and the function should not do anything else.
  - I.e., a function has no observable effect on the execution other than computing the result given a particular input.

- Is there a more formal way to describe pure functions?
  - Yes, with the concept of referential transparency.

Referential Transparency and Purity

- "An expression e is referentially transparent if, for all programs p, all occurrences of e in p can be replaced by the result of evaluating e [(or any expression yielding the same result)] without affecting the meaning of p.“

- "A function f is pure if the expression $f(x)$ is referentially transparent for all referentially transparent x."

- To read: What Is Referential Transparency? by Pierre-Yves Saumont. PDF also available on eCampus.

Referential Transparency: Example

- Is buyPouch pure?

```
def buyPouch(a: Account): Pouch =
    val pouch = new Pouch()
    a.charge(pouch.price)
    pouch
```

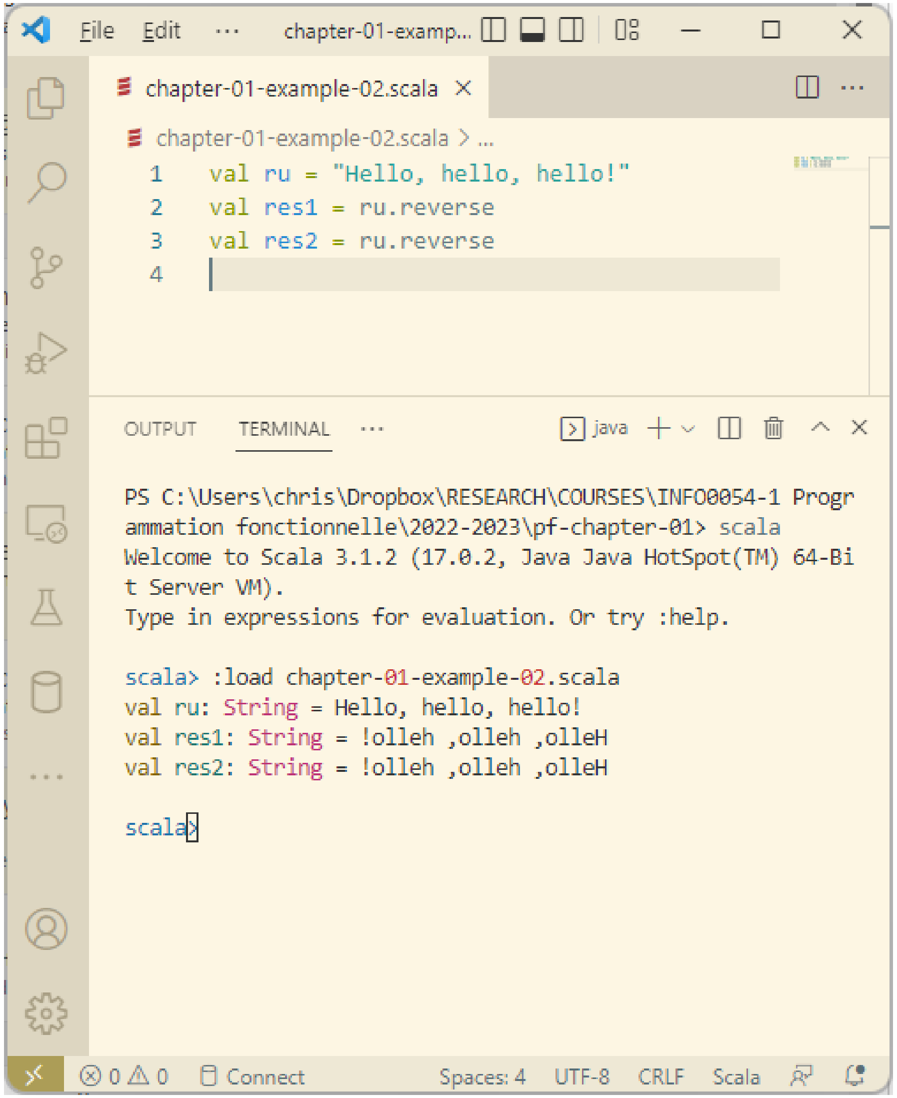
- We have already seen that buyPouch is not pure but let us analyze it using the notion of referential transparency.
  - We know that buyPouch returns a new pouch. If buyPouch is pure, then for any program `p`, the behavior of `p(buyPouch(a))` should be the same as `p(new Pouch())`.
  - That is not the case!
    - `new Pouch()` does not do anything, and
    - `buyPouch(a)` charges the account.

Referential Transparency, Purity, and Substitution Model

- Referential transparency forces the *invariant* (a condition that is always true) that everything a function does is represented by the value it returns.

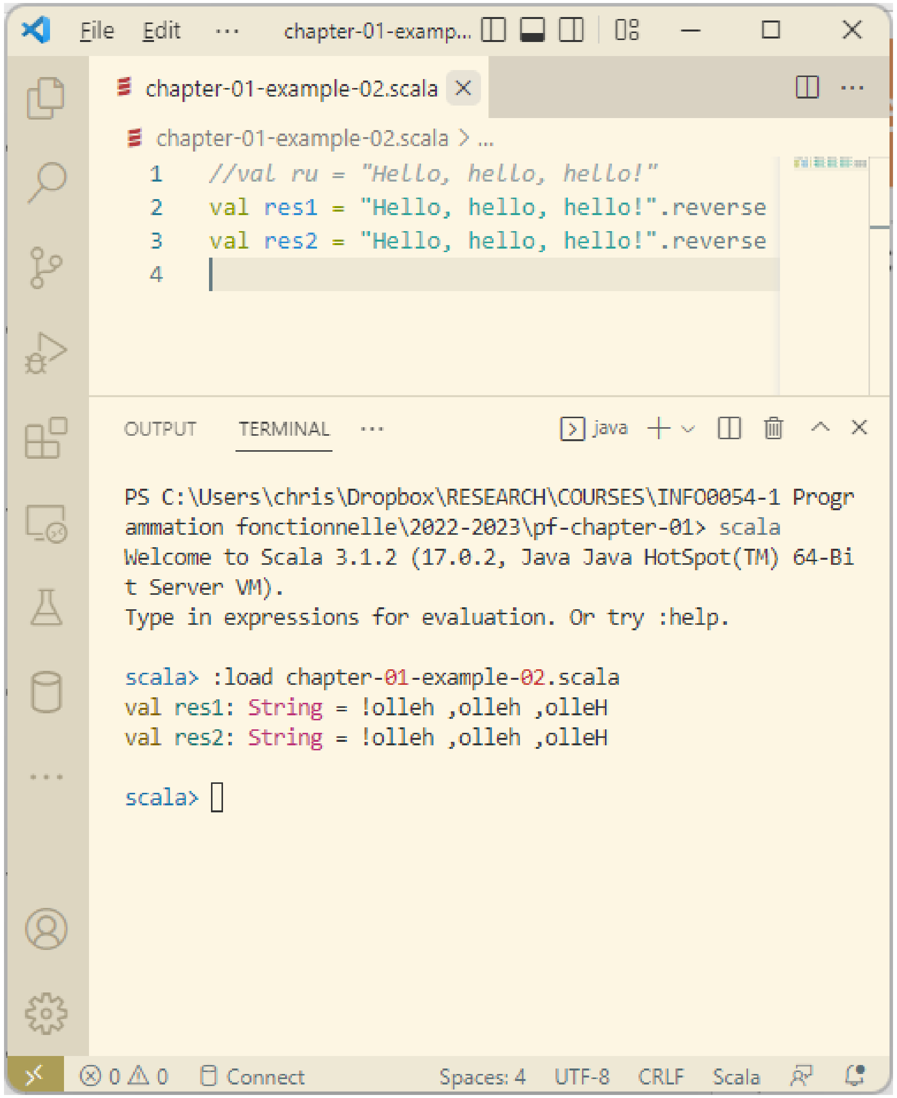
- This allows us to reason about functional programs in a particular way, we can substitute expressions with equivalent expressions (e.g., the value they evaluate to).
  - The substitution model allows us to reason about program evaluation.
  - Since we substitute expressions with equivalent expressions, referential transparency allows to reason about the equivalence of programs – equational reasoning.

Referential Transparency, Purity, and Substitution Model



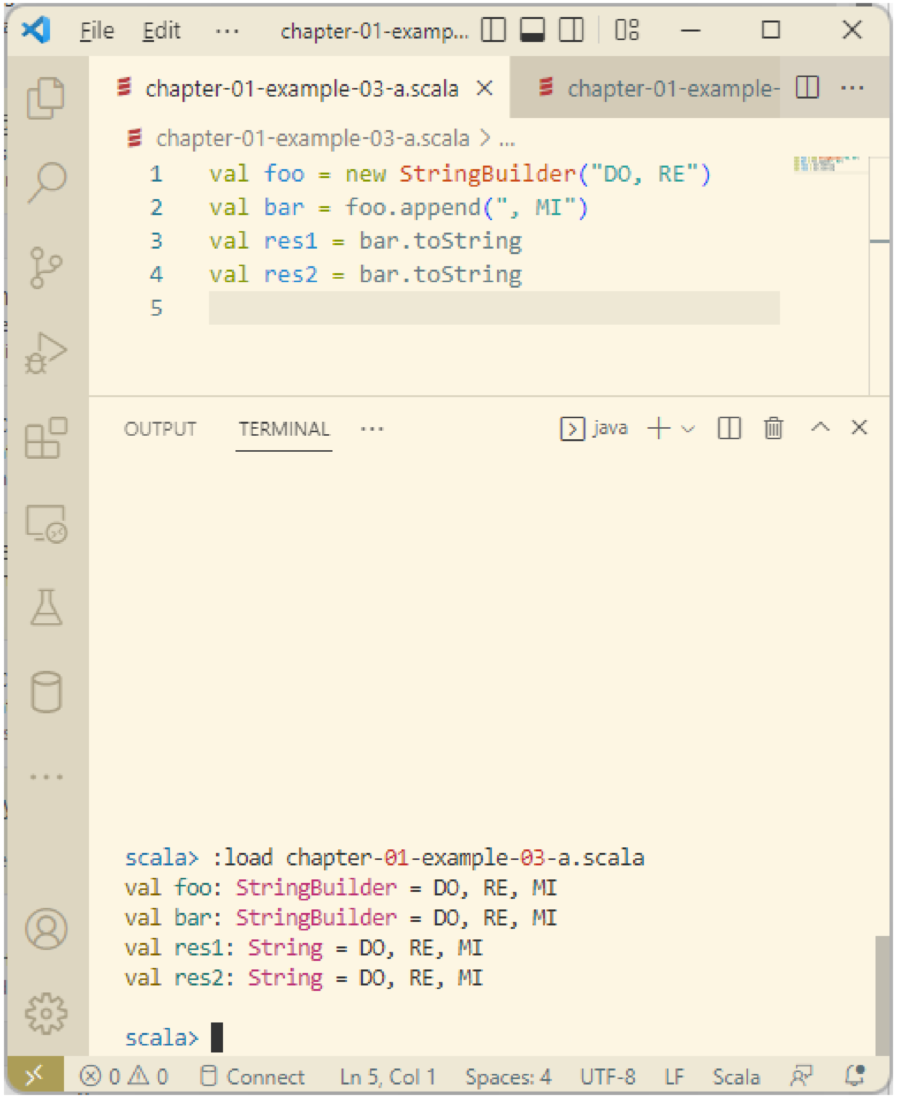
- Let's start with a simple program manipulating strings.
  - In Scala, strings are immutable; whenever we "modify" a string, the function returns a new string instead of changing the original one.
- When executing the program, we see that the values of `res1` and `res2` are the same.
- Now let's replace all occurrences of `ru` by the expression referenced by `ru`.

Referential Transparency, Purity, and Substitution Model



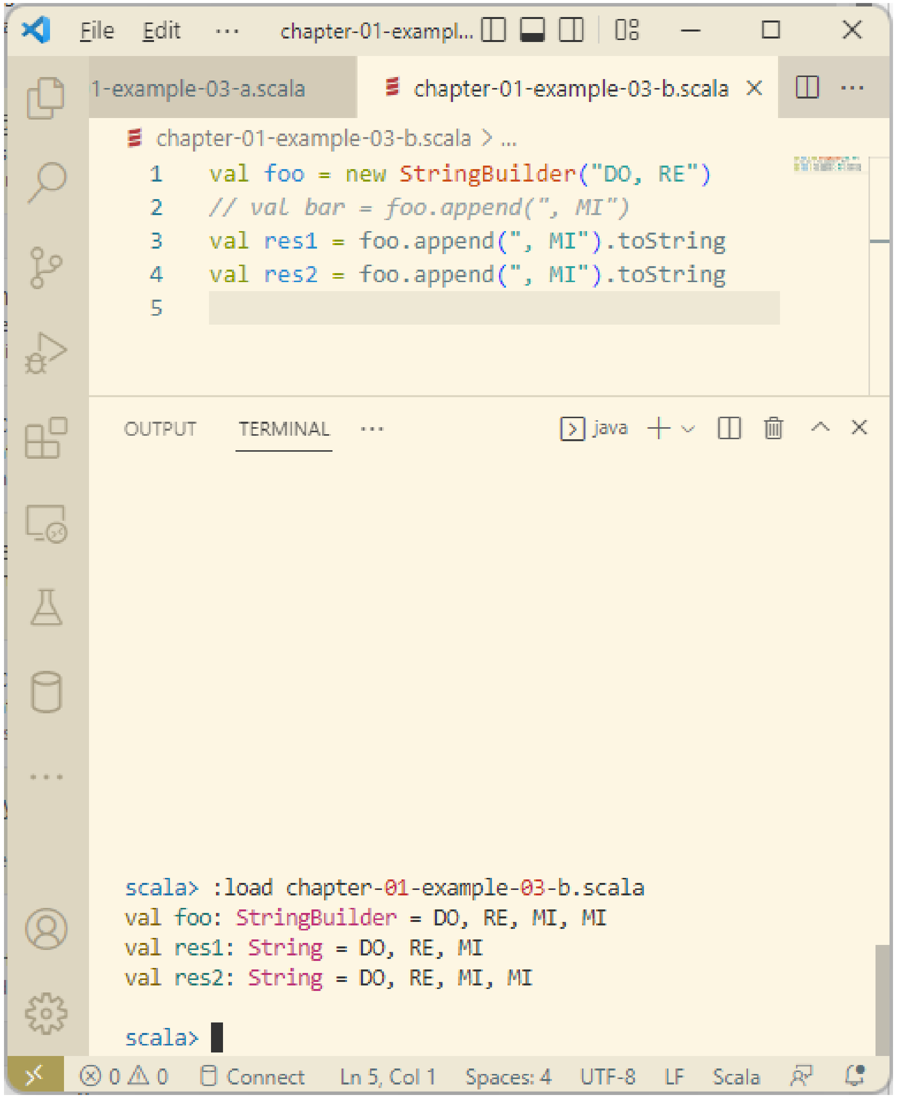
- Notice that the values of `res1` and `res2` remained the same.
- The changes we did to this program did not affect the behavior of our program, so we can deduce that `ru` was referentially transparent.
- `res1` and `res2` are referentially transparent as well. If they were to occur in a larger program, we can replace their occurrences with `"Hello, hello, hello!".reverse`

Referential Transparency, Purity, and Substitution Model



- Now an example that is NOT referentially transparent.
- Here, we are using the StringBuilder class, which provides us a mutable sequence of characters.
- When executing the program, we see that the values of `res1` and `res2` are the same.
- Now let's replace all occurrences of `bar` by the expression referenced by `bar`.

Referential Transparency, Purity, and Substitution Model



- Notice that the values of `res1` and `res2` are now different!
- Every time we call `foo.append(", MI")`, we change the internal state of the StringBuilder object referred by `foo`. StringBuilder's `append` function <u>is not a pure function!</u>
- The changes do affect the behavior of our program, so we can deduce that `foo` is not referentially transparent.
- `res1` and `res2` are not referentially transparent as well. Can you explain why?

By the way…

- You will notice that in the book, the authors entered the statements one by one. Each statement is evaluated, and its result printed on screen. append not only changes the state of the StringBuilder object, but it also returns itself. Both foo and bar refer to the same StringBuilder object.

Entering the statements one by one

```
scala> val foo = new StringBuilder("DO, RE")
val foo: StringBuilder = DO, RE

scala> val bar = foo.append(", MI")
val bar: StringBuilder = DO, RE, MI

scala> val res1 = bar.toString
val res1: String = DO, RE, MI

scala> val res2 = bar.toString
val res2: String = DO, RE, MI
```

- You may wonder why, in our example, foo and bar have the "same values" in the output. When loading a file, all statements are executed as a whole and then the values are returned. The call to append of the second statement took place before the value of foo is printed.

Loading the Scala file

```
scala> :load ex.scala
val foo: StringBuilder = DO, RE, MI
val bar: StringBuilder = DO, RE, MI
val res1: String = DO, RE, MI
val res2: String = DO, RE, MI
```

Composition and modularity

- With RT providing us a formal way to describe pure functions, we can understand why pure functions are more modular.
  - Modular programs consists of components that can be understood and reused independently, and
  - The meaning of the whole depends only on the meaning of the components and they way they are composed.

- A pure function is modular and composable because it separates two concerns:
  - The computational logic, and
  - How input is provided, and the output returned.

Part 04
Examples in Scala

Introducing Scala by example

- Here's an example with curly braces. You may be familiar with curly braces to group statements into blocks from other programming languages such as C.
- Curly braces may help some developers with the readability of their code.
- Statements are separated by a semicolon ';' or a newline.

```scala
object FactorialModule {
  def fac(n: Int): Int = {
    if(n == 0) 1
    else n * fac(n - 1)
  }

  private def formatFac(n: Int) = {
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))
  }

  def main(args: Array[String]): Unit =
    println(formatFac(5))
}
```

Introducing Scala by example

```scala
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Introducing Scala by example

- The `object` keyword allows to declare a singleton type. A singleton type is a class that that only has one instance, and that instance is referred to by that class's name.
- *If you are familiar with [software design patterns](), this idea corresponds with the [singleton]() pattern.*

```scala
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Note: (pure) functions are special cases of methods. I may refer to methods when talking about Scala.

Introducing Scala by example

- In Scala, all expressions return a value (unless something goes wrong).
- The pure function fac takes an integer as input and returns an integer. The returned value is the factorial of the input.
- The ": Int" can be read as "has type."
  - n has type Int
  - fac has type Int
  - …

```
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Introducing Scala by example

- The body of a method comes after a single '=' sign.
- The left-hand side is also called the signature of a method.
- The right-hand side is called the definition of a method.

```scala
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Introducing Scala by example

- `formatFac` is private, which means that only members of the class can access that method.
- Also notice that we have not explicitly declared the return type of `formatFac`. Scala is usually able to infer the return type of a method.

```
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Introducing Scala by example

- Scala will look for `main` method with a specific signature (accepting an array of strings and returning *nothing*) and use that as an entry to start the program.
- `main` does not return anything and its return type is therefore `Unit`. `Unit` is also written as `()`, which can be seen as an empty tuple. The return type of `println` is `Unit`.

```
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

By the way…

- `val` is used to declare immutable variables. I.e., we cannot reassign them.
- *`var`* is used for mutable variables*, but this is to be avoided in this course. ;-)*

```
scala> val x = 1
val x: Int = 1

scala> var y = x
var y: Int = 1

scala> y = 2
y: Int = 2

scala> x = 5
-- [E052] Type Error: ----------------------------------------------------------
1 |x = 5
  |^^^^^
  |Reassignment to val x
  |
  | longer explanation available when compiling with `-explain`
1 error found
```

Overwriting the value of a variable.

Running our module

1) By compiling the source code into bytecode and run the latter

```
$ scalac FactorialModule.scala
$ scala FactorialModule
The factorial of 5 is 120
```

2) We can "skip" compilation for simple programs

```
$ scala FactorialModule.scala
The factorial of 5 is 120
```

3) Using the Scala interpreter and its Read-Eval-Print-Loop (REPL)

```
$ scala
Welcome to Scala 3.1.2 (17.0.2, Java Java HotSpot(TM) 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> :load FactorialModule.scala
// defined object FactorialModule

scala> FactorialModule.fac(7)
val res0: Int = 5040

scala> :quit
```

4) Using Integrated Development Environments (IDEs), but that's up to you to configure.

Modules, object, and namespaces

- In Scala, every value is an object. Each object *may* have one or more members.
- An object whose primary purpose is to give its members a namespace is called a module. Member *include*:
  - Methods declared with `def`
  - Objects declared with `val`
  - Objects declared with `object`
  - …
- In our example, `FactorialModule` is considered a module. We accessed its members with a dot (as we do in Java).

```
scala> FactorialModule.fac(7)
val res0: Int = 5040
```

Modules, object, and namespaces

- Within an object, the members do not need to be qualified.
- The method `formatFac` does not need to call `FactorialModule.fac(n)`.
- *If necessary*, members have access to their enclosing object via the keyword `this`.

```
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    //message.format(n, fac(n))
    message.format(n, this.fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Importing

- We can import members into scope and then use them in an *unqualified* member from then onwards.

```
scala> import FactorialModule.fac

scala> fac(8)
val res0: Int = 40320
```

- We can also import all the public members of an object into scope by using an underscore.

```
scala> import FactorialModule._

scala> fac(8)
val res0: Int = 40320
```

Homework

- Read chapters 1 and 2 of the book.
- Install Scala on your system and try running some of the code.
- Prepare some of the first exercises.

Lexicon

- Accumulator – accumulateur
- Composable – composable
- Functional programming – programmation fonctionnelle
- Higher-order function – fonction d'ordre supérieur
- Higher-order programming – programmation d'ordre supérieure
- Immutable – immuable
- Modular – modulaire
- Mutual recursion – récursion mutuelle / récursion croisée
- Referential transparency – transparence référentielle
- Seperation of concerns – séparation des préoccupations
- Side effect – effet de bord
- Tail recursion – récursion terminale
- Tuple – tuple