Lecture 1
Introduction to Functional Programming & Scala

14.3.2023

Dr. Iflaah Salman

LUT University

Functional Programming

- Imperative Programming
  - Perceived as traditional programming
  - C, C++, JAVA, C#
  - The programmer tells the computer what to do, e.g., $x = y + z$.
  - Oriented around control statements, looping constructs and assignments.
- Functional Programming
  - Aims on describing the solution
  - What the program needs to be doing (rather than how it should be done).

LUT University

Functional Programming

- It is based on ***pure functions.***
  - Functions that have ***no side effects*!**

- Side Effects: rather than simply returning the results, a function does one of the following:
  - Modifying a variable
  - Modifying a data structure in place
  - Setting a field on an object
  - Throwing an exception or halting with an error
  - Printing to the console or reading user input
  - Reading from or writing to a file
  - Drawing on the screen

LUT University

Functional Programming

"FP is a restriction on how we write programs, but not on what programs can express".

It increases modularity via programming pure functions that are easier to:

- test
- reuse
- parallelize
- generalize
- can be reasoned

Pure functions are less prone to bugs!

A Program with Side Effects

our function merely returns a **Coffee** and these other actions are happening *on the side*.

Listing 1.1. A Scala program with side effects

```
class Cafe {

  def buyCoffee(cc: CreditCard): Coffee = {

    val cup = new Coffee()

    cc.charge(cup.price)

    cup

  }
}
```

The `class` keyword introduces a class, much like in Java. Its body is contained in curly braces, { and }.

A method of a class is introduced by the `def` keyword. `cc: CreditCard` defines a parameter named `cc` of type `CreditCard`. The `Coffee` return type of the `buyCoffee` method is given after the parameter list, and the method body consists of a block within curly braces after an = sign.

Side effect. Actually charges the credit card.

No semicolons are necessary. Newlines delimit statements in a block.

We don't need to say `return`. Since `cup` is the last statement in the block, it is automatically returned.

Difficult to test because it is contacting the credit card company, and we don't want our tests to do that.

A Program with Side Effects
making it more testable

We *can* develop mocks to test *Payments* (*which can be an interface*).

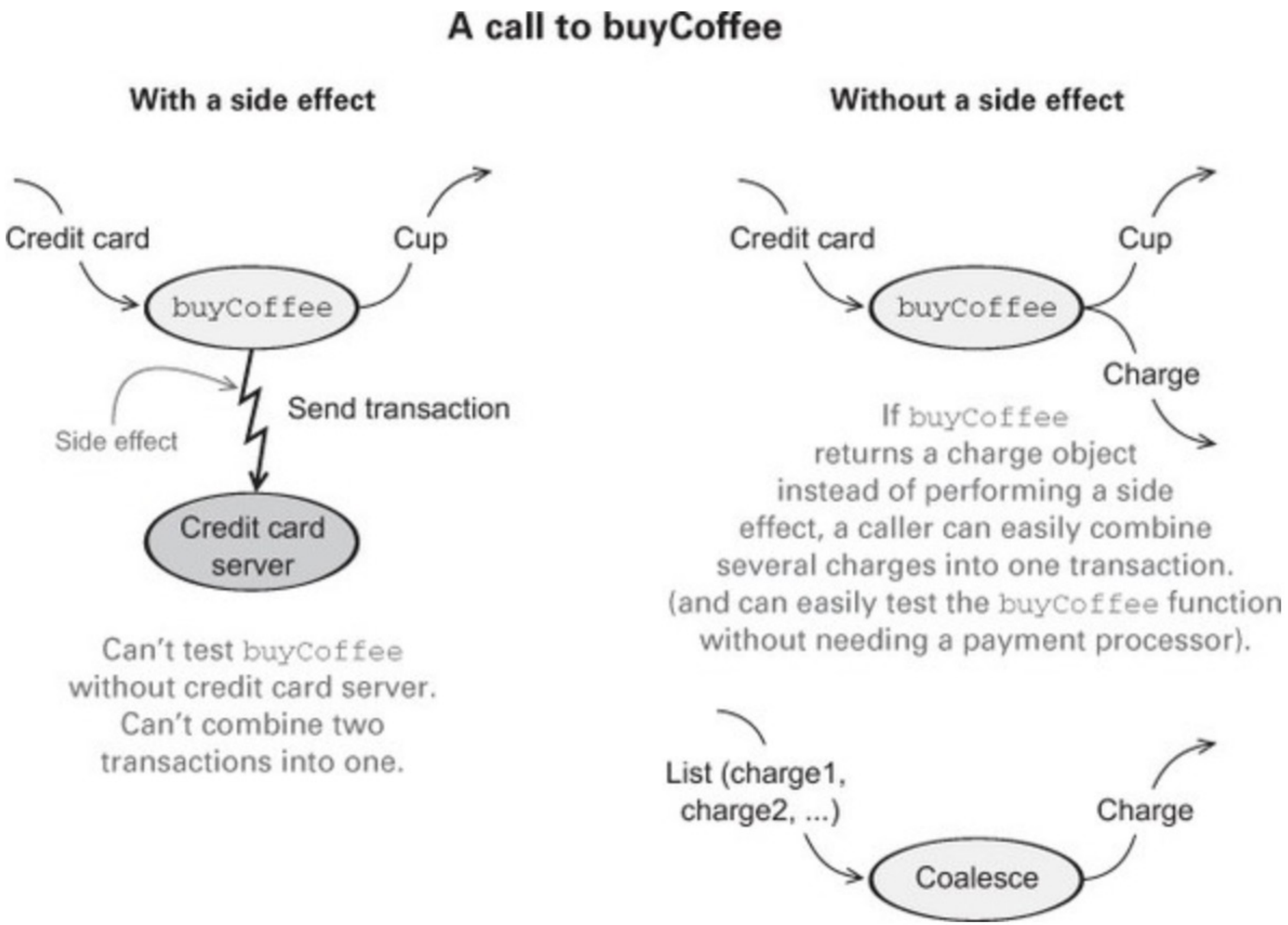
Listing 1.2. Adding a payments object

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

We have gained some level of testability.

A Program with Side Effects
reusability

The code was still difficult to reuse.
Scenario: In the case of buying more than one coffee, calling the function that many times (in a loop) would contact the bank that many times and that many processing charges.



A Program with Side Effects

removing side effects

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    (cup, Charge(cc, cup.price))
  }
}
```

buyCoffee now returns a pair of a Coffee and a Charge, indicated with the type (Coffee, Charge). Whatever system processes payments is not involved at all here.

To create a pair, we put the cup and Charge in parentheses separated by a comma.

```
case class Charge(cc: CreditCard, amount: Double) {
        def combine(other: Charge): Charge =
        if (cc == other.cc)
        Charge(cc, amount + other.amount)
        else
        throw new Exception("Can't combine charges to different cards")
}
```

A case class has one primary constructor whose argument list comes after the class name (here, Charge). The parameters in this list become public, unmodifiable (immutable) fields of the class and can be accessed using the usual object-oriented dot notation, as in other.cc.

```
case class Charge(cc: CreditCard, amount: Double) {

  def combine(other: Charge): Charge =

    if (cc == other.cc)

      Charge(cc, amount + other.amount)

    else

      throw new Exception("Can't combine charges to different cards")

}
```

An if expression has the same syntax as in Java, but it also returns a value equal to the result of whichever branch is taken. If cc == other.cc, then combine will return Charge(..); otherwise the exception in the else branch will be thrown.

A case class can be created without the keyword new. We just use the class name followed by the list of arguments for its primary constructor.

The syntax for throwing exceptions is the same as in Java and many other languages. We'll discuss more functional ways of handling error conditions in a later chapter.

LUT University

A Program with Side Effects

Listing 1.3. Buying multiple cups with `buyCoffees`

```scala
class Cafe {

  def buyCoffee(cc: CreditCard): (Coffee, Charge) = ...

  def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))
    val (coffees, charges) = purchases.unzip
    (coffees, charges.reduce((c1,c2) => c1.combine(c2)))
  }
}
```

`List.fill(n)(x)` creates a `List` with `n` copies of `x`. We'll explain this funny function call syntax in a later chapter.

`List[Coffee]` is an immutable singly linked list of `Coffee` values. We'll discuss this data type more in chapter 3.

`unzip` splits a list of pairs into a pair of lists. Here we're destructuring this pair to declare two values (`coffees` and `charges`) on one line.

`charges.reduce` reduces the entire list of charges to a single charge, using `combine` to combine charges two at a time. `reduce` is an example of a *higher-order function*, which we'll properly introduce in the next chapter.

Functional Programming

"FP is merely a discipline that takes what many consider a good idea to its logical endpoint, applying the discipline even in situations where its applicability is less obvious."

A Pure Function

is easier to reason about

A function **f** with input type **A** and output type **B**

(written in Scala: A => B, pronounced “A to B” or “A arrow B”)

is a computation that relates every value **a** of type **A** to exactly one value **b** of type **B** such that **b** is determined solely by the value of **a**.

Any changing state of an internal or external process is irrelevant to computing the result **f(a)**.

Example:

function: intToString

Int => String

“IF it really is a *function*, it will do nothing else!”

function: length function of a String in Java, Scala

Returns only length; the same length is always returned.

strings are not modified (*immutability*)

LUT University

Referential Transparency
pure functions

Referential transparency (RT) is a property of *expressions* and not just functions.

2 + 3 = 5

2, 3 are expressions; + is the pure function. This has *no side effects*.

The answer is always 5 OR always evaluates to 5.

If in a program we replace 2 + 3 with 5, no behaviour or meaning will change.

> “A function is *pure* if calling it with RT arguments is also RT.”

> “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** without affecting the meaning of **p**. A function **f** is pure if the expression **f(x)** is referentially transparent for all referentially transparent **x**.”

Referential Transparency

purity and substitution model

```scala
def buyCoffee(cc: CreditCard): Coffee = {
  val cup = new Coffee()
  cc.charge(cup.price)
  cup
}
```

buyCoffee(customerCreditCard) will evaluate to **cup**
cup is **new Coffee()**

NOW

p(**buyCoffee(customerCreditCard)**) != p(**new Coffee()**)

Therefore, it does not held RT and purity.

RT forces the invariance that everything a function *does* is represented by the *value* that it returns, according to the result type of the function.

Referential Transparency

purity and substitution model

- RT enables a simple and natural mode of reasoning about program evaluation called the substitution model.
- Computation proceeds like an algebraic equation.
- RT enables equational reasoning about programs.

In an algebraic equation, every part of an expression is expanded, replacing all variables with their referents, and then, reducing it to its simplest form. At each step, a term is replaced with an equivalent one; computation proceeds by substituting equals for equals.

LUT University

Referential Transparency
purity and substitution model

```
scala> val x = "Hello, World"
x: java.lang.String = Hello, World

scala> val r1 = x.reverse
r1: String = dlroW ,olleH

scala> val r2 = x.reverse
r2: String = dlroW ,olleH
```

r1 and r2 are the same.

replace all occurrences of x with the expression referenced by x.

```
scala> val r1 = "Hello, World".reverse
r1: String = dlroW ,olleH

scala> val r2 = "Hello, World".reverse
r2: String = dlroW ,olleH
```

r1 and r2 are still the same.

Replacement/expansion didn’t affect the result.
Therefore, **x** was referentially transparent.

Referential Transparency
not RT example

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val y = x.append(", World")
y: java.lang.StringBuilder = Hello, World

scala> val r1 = y.toString
r1: java.lang.String = Hello, World

scala> val r2 = y.toString
r2: java.lang.String = Hello, World
```

r1 and r2 are the same.

now we substitute the call to append(), replacing all occurrences of y with the expression referenced by y

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val r1 = x.append(", World").toString
r1: java.lang.String = Hello, World

scala> val r2 = x.append(", World").toString
r2: java.lang.String = Hello, World, World
```

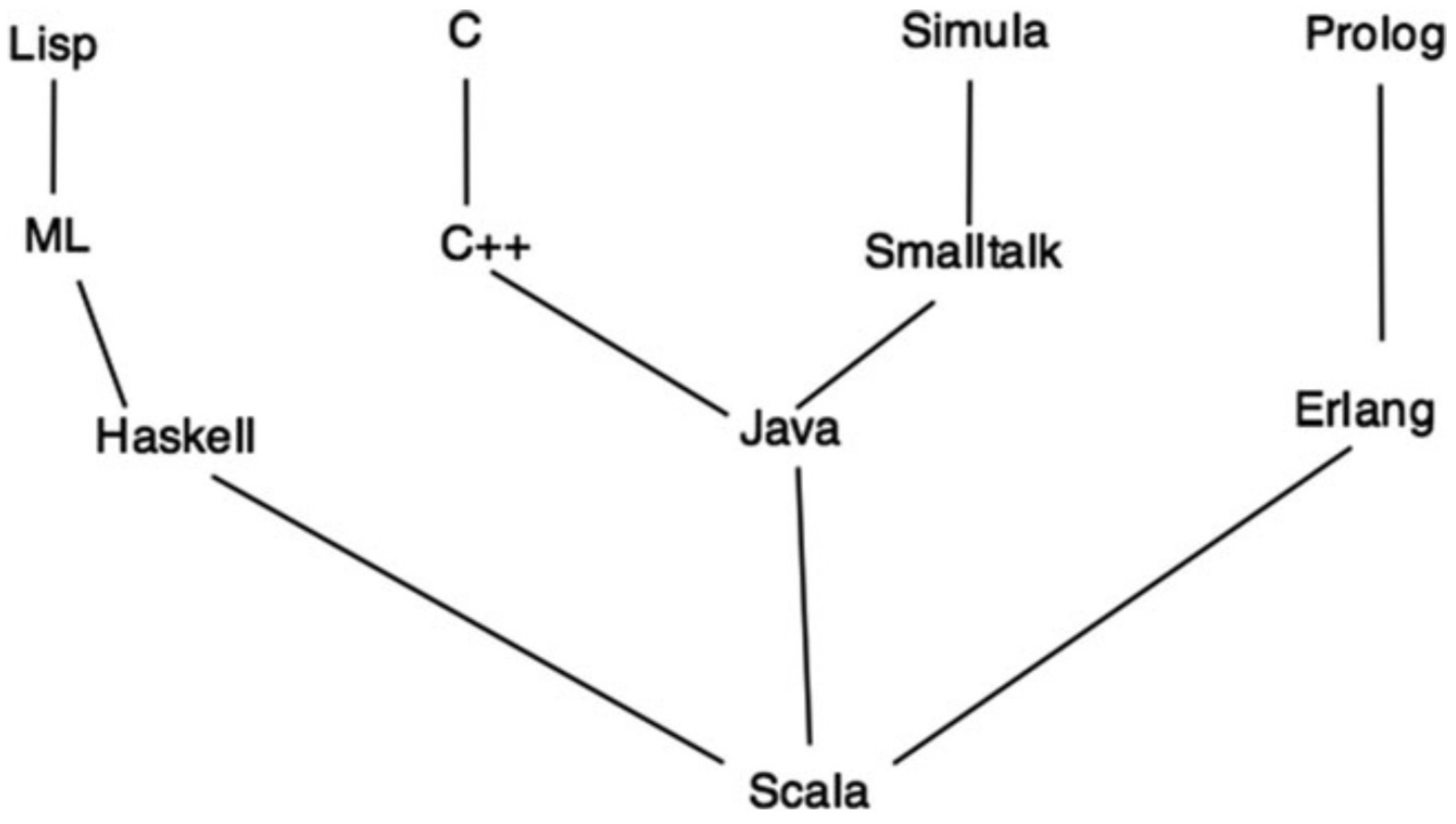
r1 and r2 are no longer the same.

By the time r2 calls x.append(), r1 had already *mutated* the object referenced by x.

The function is not referentially transparent. The value returned depends on x which may be changed by another process. If you think of this in terms of Object Oriented programming, x could be a class member variable and plus a class method. Such operations are referentially opaque and are common in the OO paradigm.

Scala

Scala


Lisp
C
Simula
Prolog
ML
C++
Smalltalk
Haskell
Java
Erlang
Scala

Scala

- It is a multi-paradigm (a hybrid) language: OOP + FP.
- From the OOP perspective, it is quite like JAVA or C++.
- It also enables functional programming like Haskell.
- It can be compiled into a JAVA byte code – runs on JVM.
- It has interoperability with JAVA.
- JRE allows Scala to exploit its libraries.

References

- Chiusano, P., & Bjarnason, R. (2014). *Functional Programming in Scala*. Manning publications.
- Hunt, J. (2018). A Beginner's Guide to Scala, Object Orientation and Functional Programming. In *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer International Publishing. https://doi.org/10.1007/978-3-319-75771-1