Pure Functions

06016415 Functional Programming

Outline

- Program with side effects
- Pure function

Why would we use functional programming?

“Once you get used to it, it’s self-evident. It’s clear. I look at my function. What can it work with? Its arguments.

….

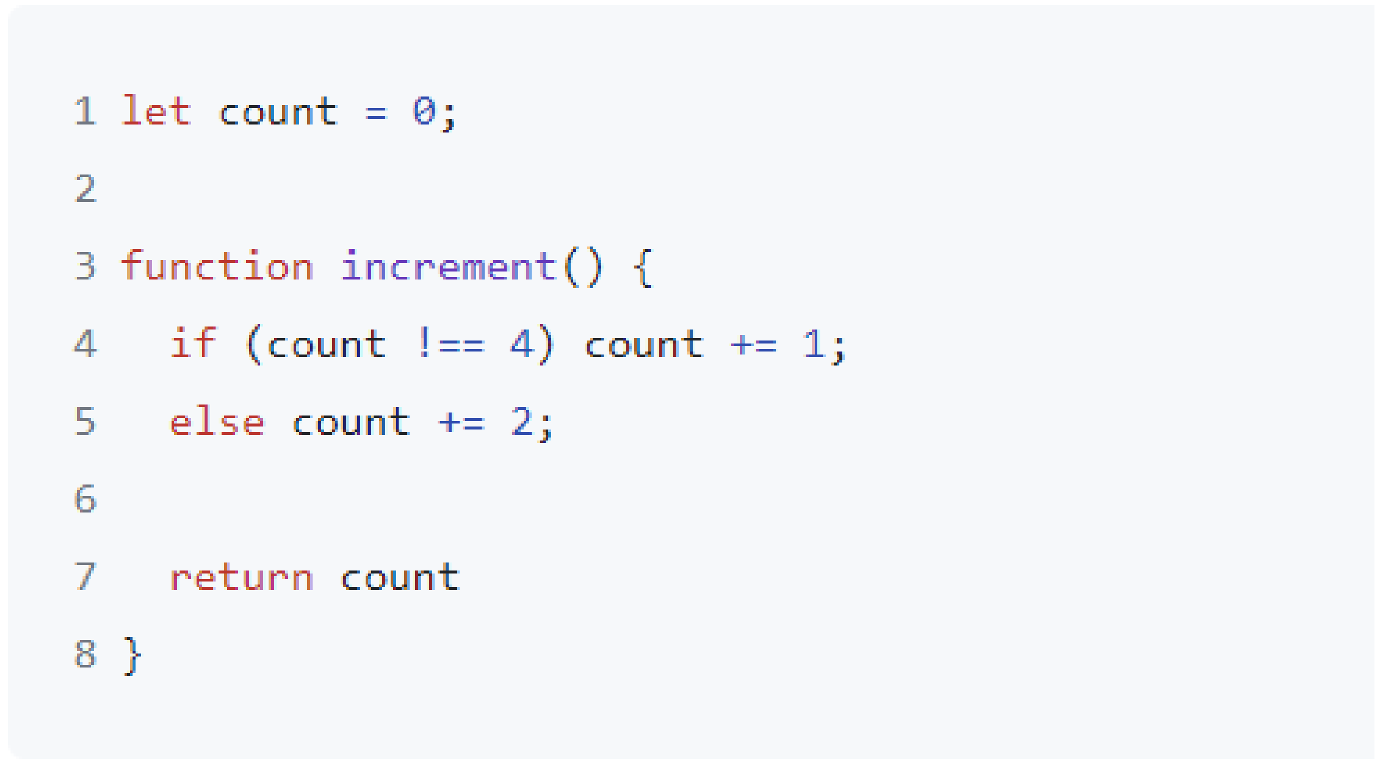
> **Anything else? ….. No.**
> **Are there any global variables? …… No.**
> **Other module data?….. No.**

…………It’s just that.” - *Robert Virding*

Debugging

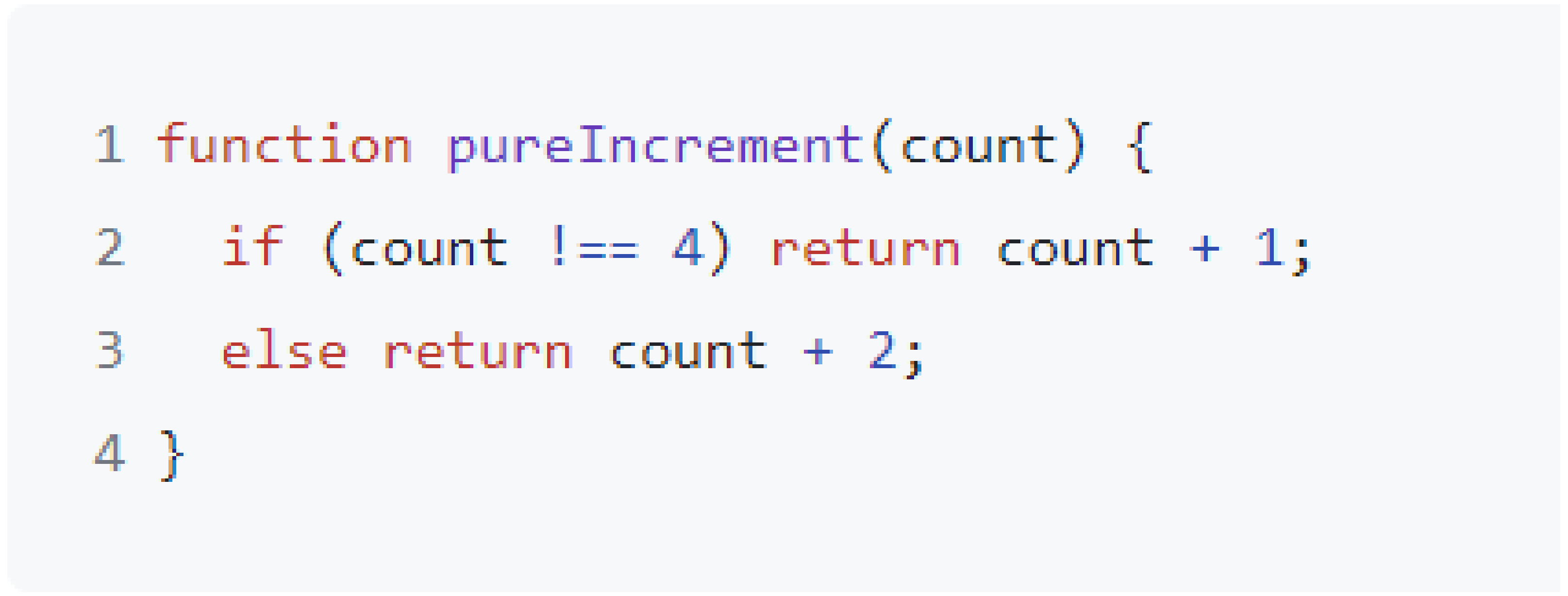
Debugging functional programming is easier than other programming paradigms because of its modularity and lack of side effects.

EX. a counter that skipped the number 5



```
let count = 0;

function increment() {
  if (count !== 4) count += 1;
  else count += 2;

  return count
}
```

In a functional way

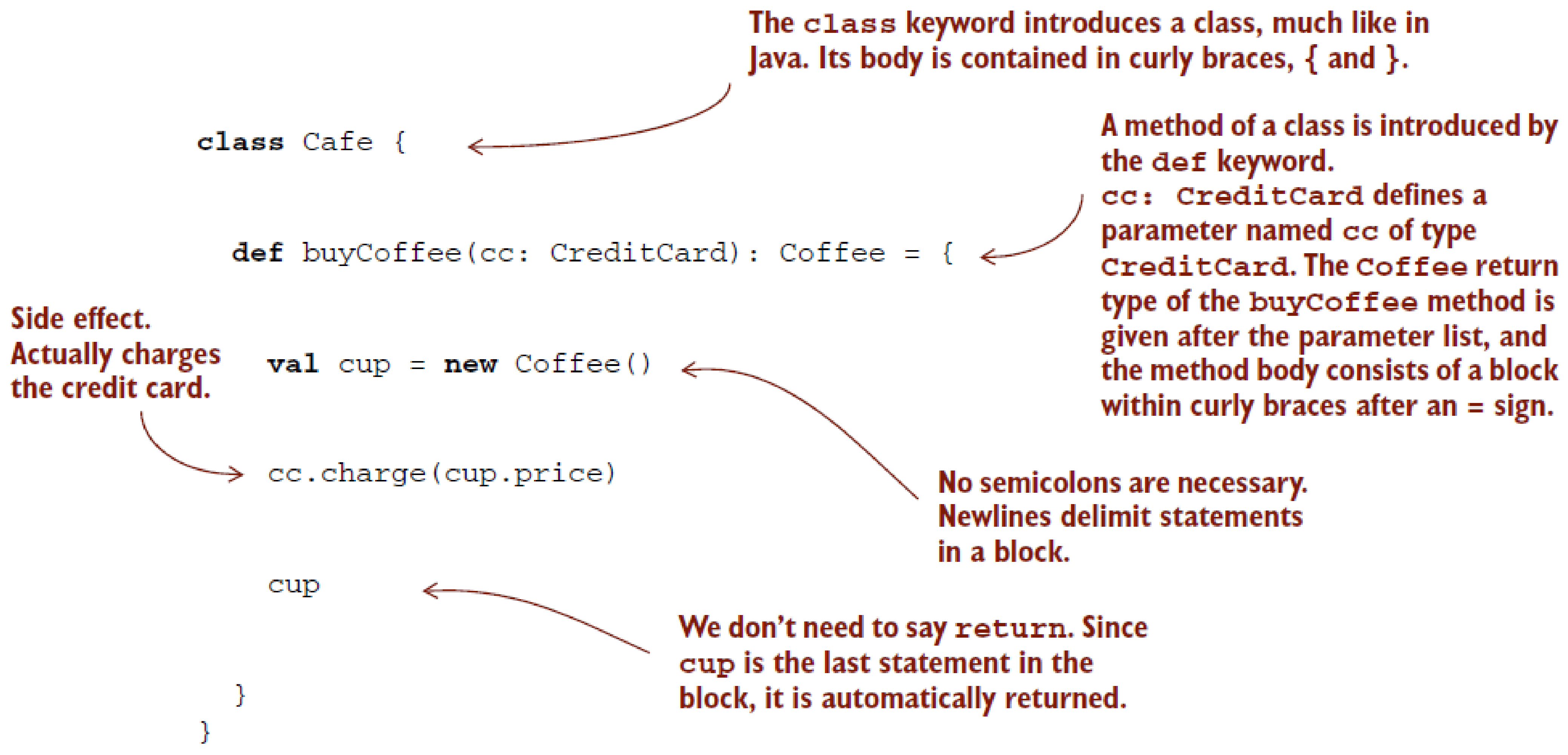


```
function pureIncrement(count) {
  if (count !== 4) return count + 1;
  else return count + 2;
}
```

Program with side effects

- In FP, the programs is constructed using only *pure functions* — functions that have no *side effects*.
- **But what are side effects?**
  - A function has a side effect if it does something other than simply return a result. This includes, for example, the following cases:
    - 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

Example: 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.

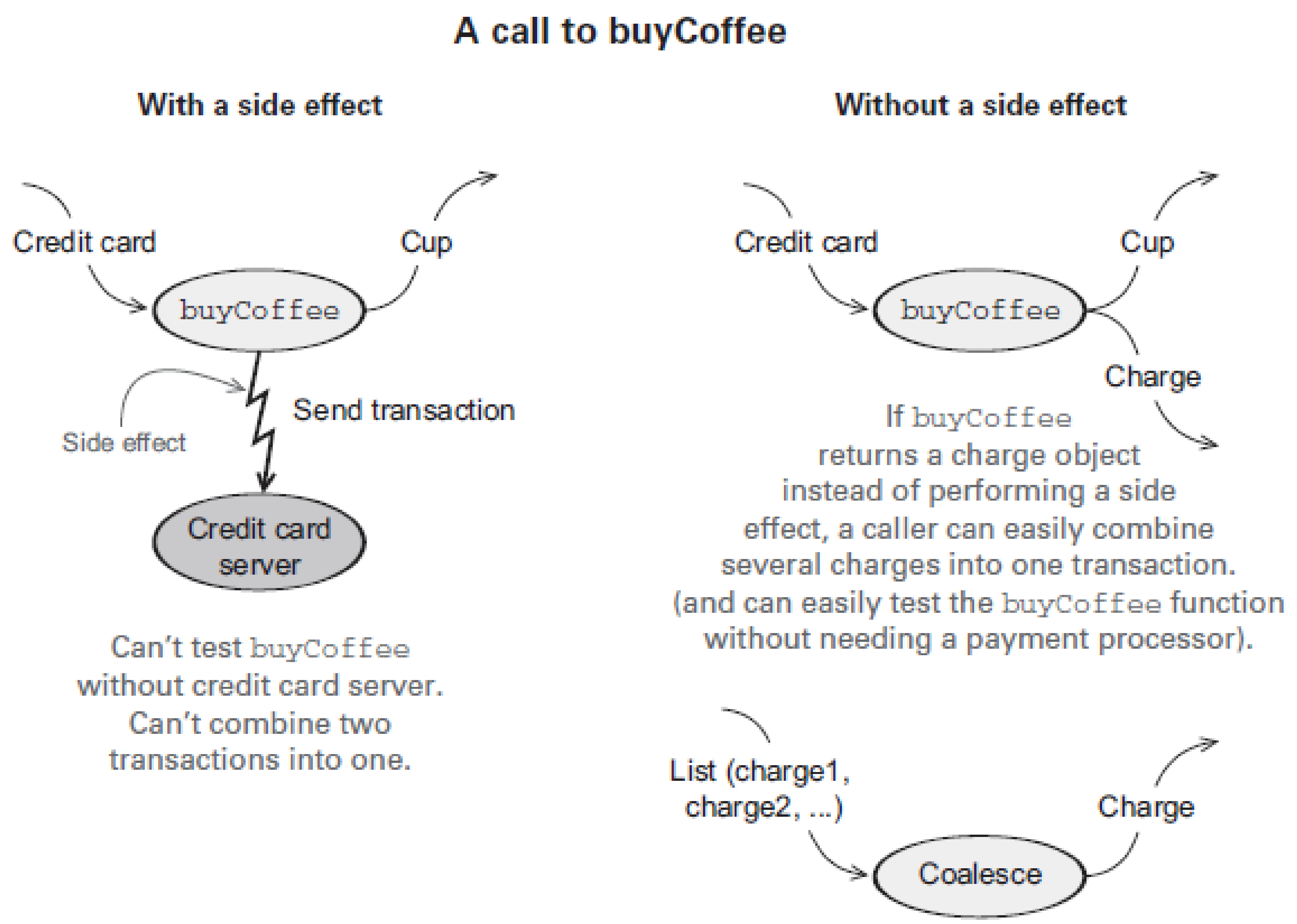
Side effect. Actually charges the credit card.

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

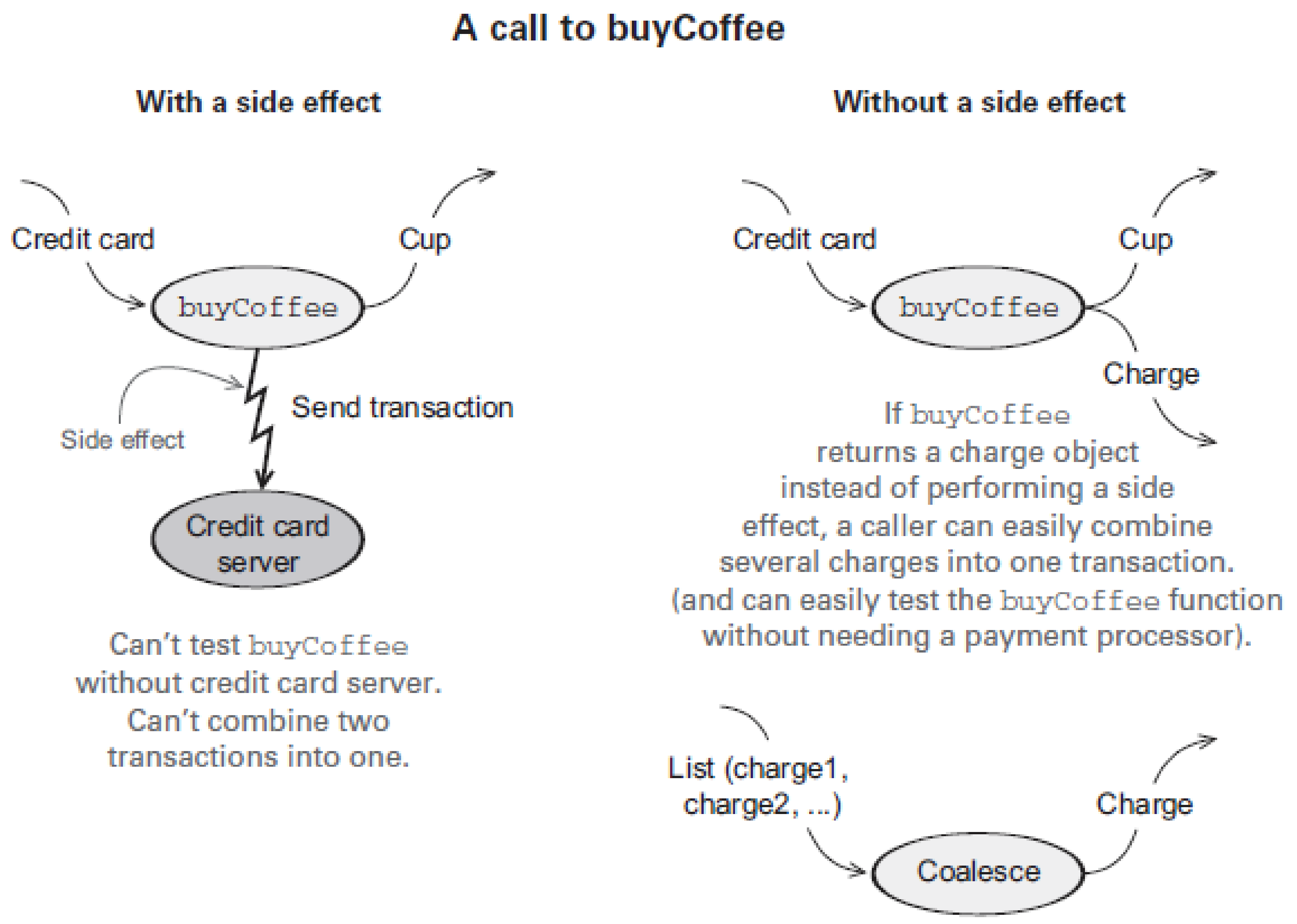
We don't need to say return. Since cup is the last statement in the block, it is automatically returned.

```
class Cafe:
  def buyCoffee(cc: CreditCard): Coffee =
    val cup = Coffee()
    cc.charge(cup.price)
    cup

class CreditCard:
  def charge(price: Double): Unit =
    println("charging " + price)

class Coffee:
  val price: Double = 2.0

val cc = CreditCard()
val cafe = Cafe()
val cup = cafe.buyCoffee(cc)
```

Removing the side effects


A call to buyCoffee
With a side effect
Credit card
Cup
buyCoffee
Side effect
Send transaction
Credit card server
Can’t test buyCoffee without credit card server. Can’t combine two transactions into one.
Without a side effect
Credit card
Cup
buyCoffee
Charge
If buyCoffee returns a charge object instead of performing a side effect, a caller can easily combine several charges into one transaction. (and can easily test the buyCoffee function without needing a payment processor).
List (charge1, charge2, ...)
Charge
Coalesce

Removing the side effects


A call to buyCoffee
With a side effect
Without a side effect
Credit card
Cup
buyCoffee
Side effect
Send transaction
Credit card server
Can't test buyCoffee without credit card server. Can't combine two transactions into one.
Credit card
Cup
buyCoffee
Charge
If buyCoffee returns a charge object instead of performing a side effect, a caller can easily combine several charges into one transaction. (and can easily test the buyCoffee function without needing a payment processor).
List (charge1, charge2, ...)
Charge
Coalesce

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

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

class CreditCard:
  def charge(price: Double): Unit =
    println("charging " + price)

class Coffee:
  val price: Double = 2.0
```

Brake down the functions into smaller functions

Let's say that you want to solve a math problem, like:

(6 * 9) / ((4 + 2) + (4 * 3))

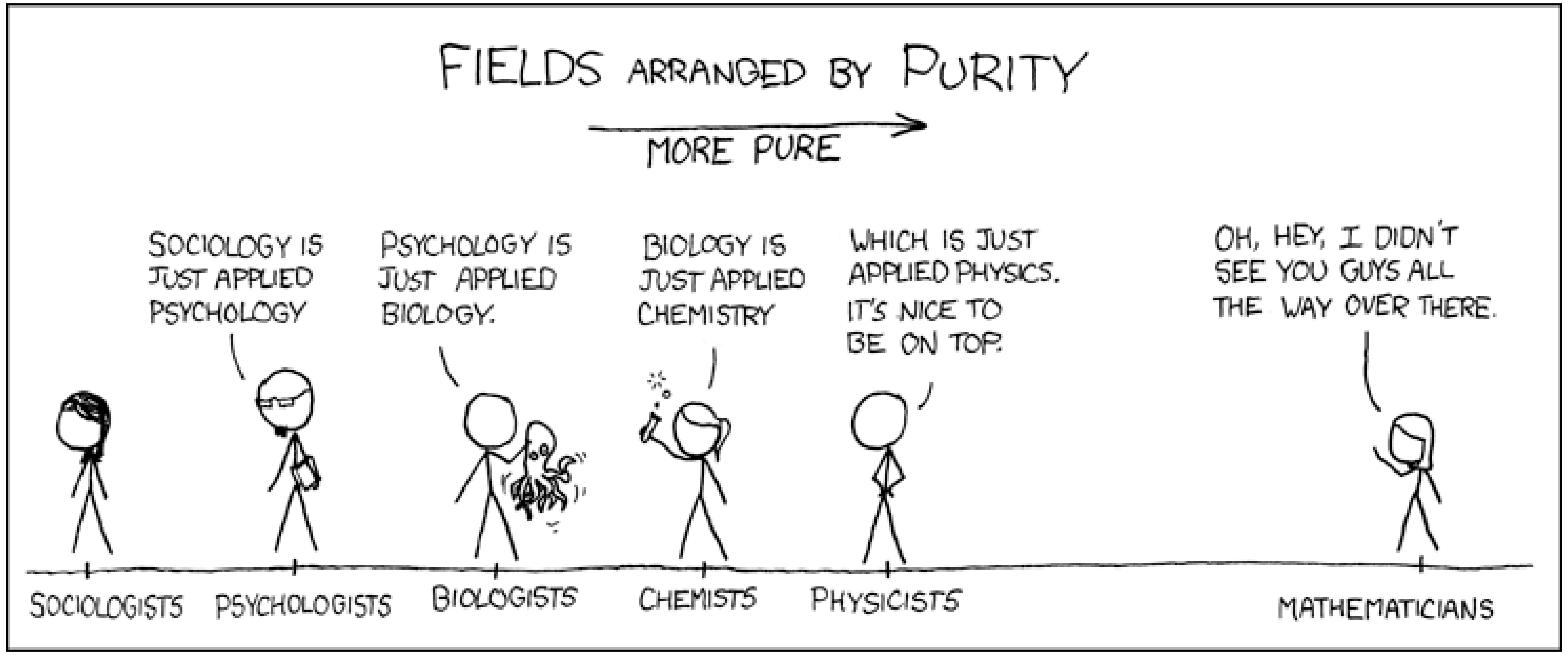
In a functional way

```
(define (mathexample)
  (/
    (* 6 9)
    (+
      (+ 2 4)
      (* 4 3)
    )
  )
)
```

This is why functional programming is often referred to as **"pure programming!"**

Functions run as if they are evaluating mathematical functions, with no unintended side effects.

Purity


FIELDS ARRANGED BY PURITY
MORE PURE
SOCIOLOGY IS JUST APPLIED PSYCHOLOGY
PSYCHOLOGY IS JUST APPLIED BIOLOGY.
BIOLOGY IS JUST APPLIED CHEMISTRY
WHICH IS JUST APPLIED PHYSICS. IT'S NICE TO BE ON TOP.
OH, HEY, I DIDN'T SEE YOU GUYS ALL THE WAY OVER THERE.
SOCIOLOGISTS
PSYCHOLOGISTS
BIOLOGISTS
CHEMISTS
PHYSICISTS
MATHEMATICIANS

Example

“output depends on input”

$$f(x) = \sum_{n=0}^{100} n + x$$

Exactly what is a (pure) function?

“output depends on input”

- Pure function will always get the same result.
- The return is solely dependent on the parameter list.

Example

```
A => B
```

- 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 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).

```
intToString()
```

- A function intToString having type Int => String will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

Reusability

- The small pure functions can often reuse them much more easily than your traditional object-oriented program.
- In OOP, the class can reuse by add a feature
  - Typically you add conditionals and parameters, and it will get larger.
  - The abstract classes and interfaces get pretty robust. It require to pay careful attention to the larger application architecture because of side effects and other factors that will affect.
- In FP, it’s the opposite in that your functions get smaller and much more specific to what you want.
- **One function does one thing, and whenever you want to do that one thing, you use that one function.**