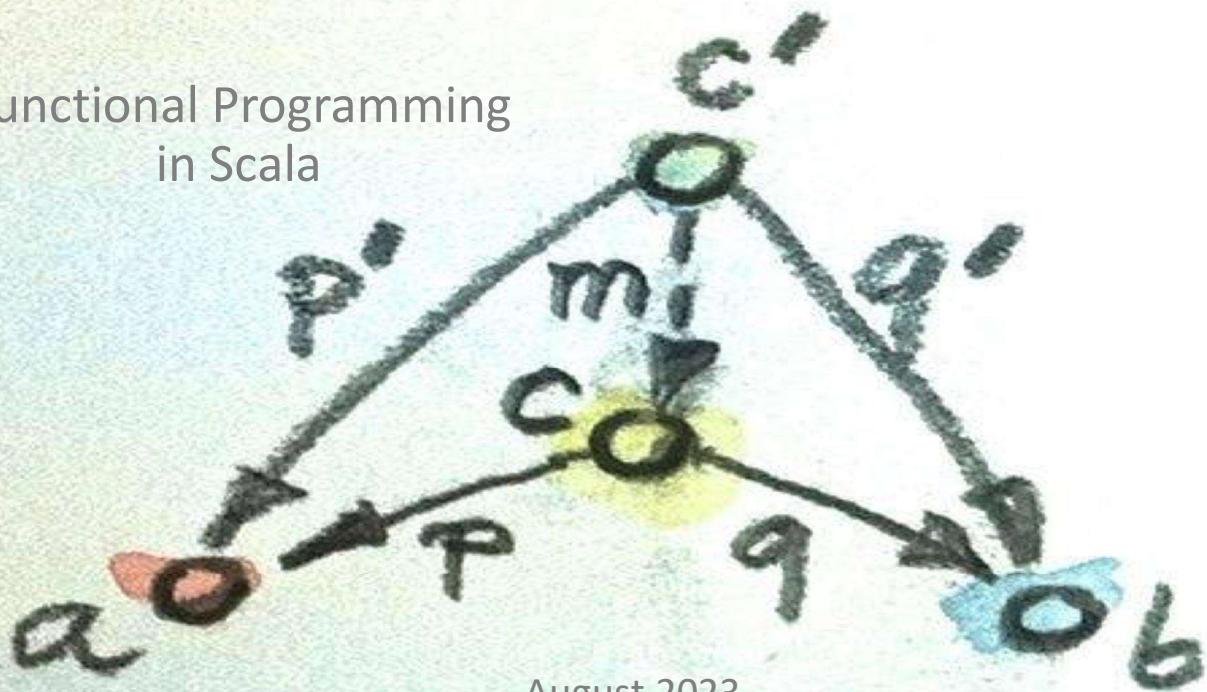
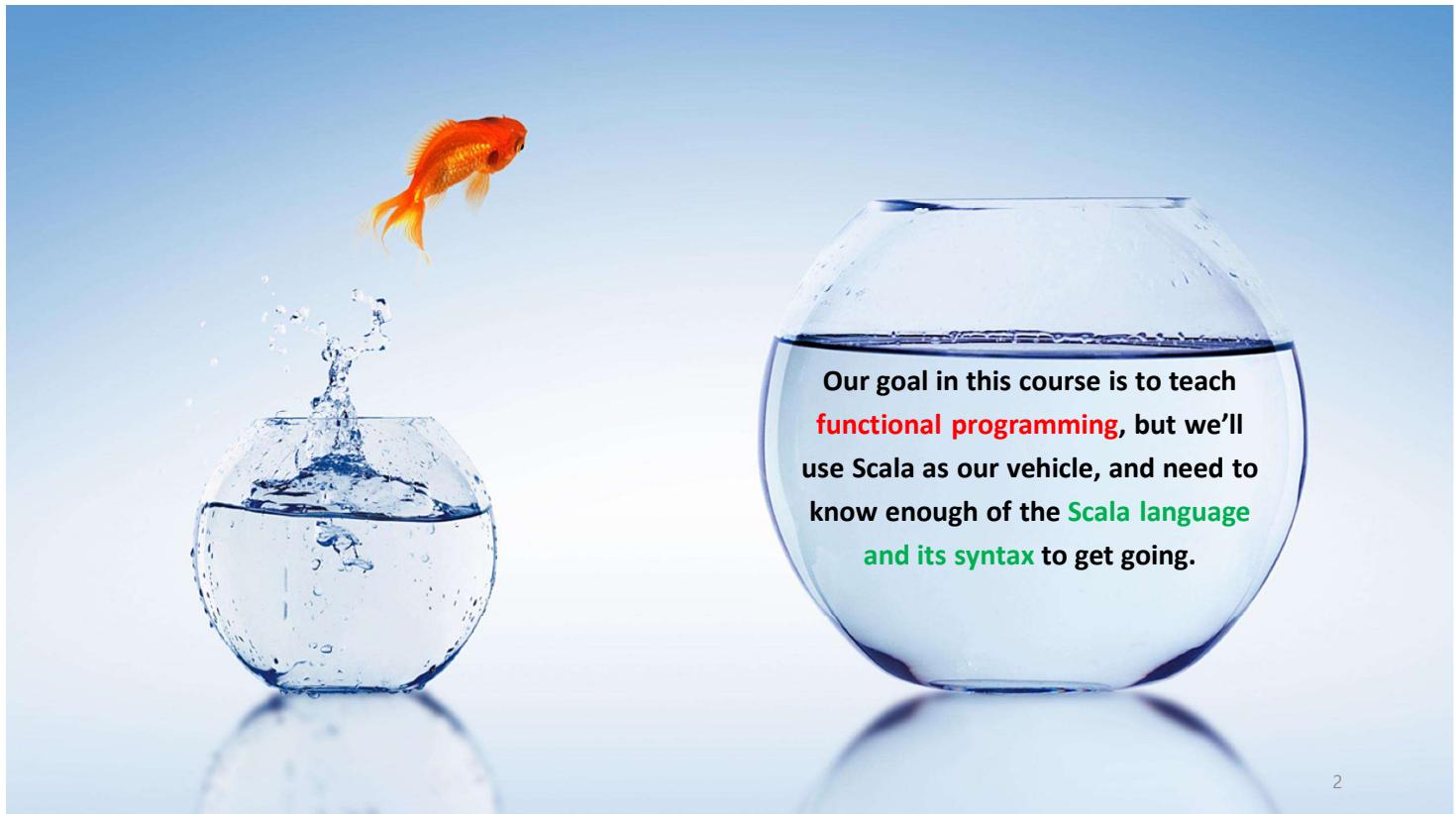


# Functional Programming in Scala



August 2023

1



2

# Prerequisites – Not Really ☺

- A reasonable background in functional programming, especially in Scala
- Recommended prerequisite course: “**Principles of Functional Programming in Scala**”, Coursera Course by Martin Odersky
- Other experiences in some other functional programming languages, such as Haskell, F#, Clojure, etc. would be fine.



3

*Who am I?*

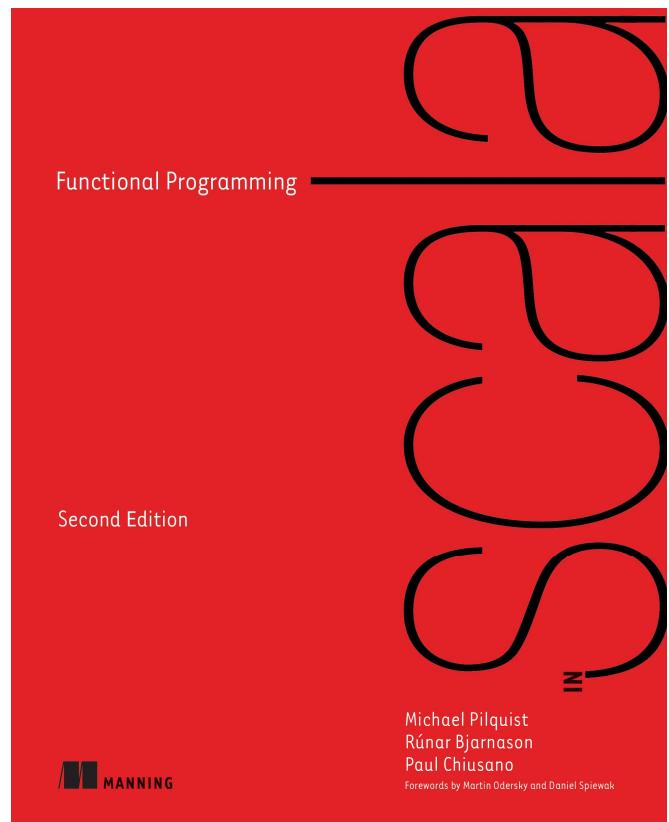


김정선 (金 正 善, Jungsun Kim)  
한양대학교 소프트웨어학부  
소프트웨어융합대학  
(College of Computing)

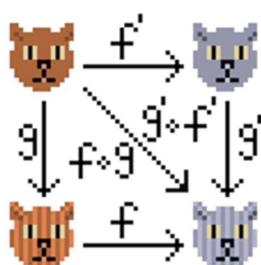
Office: 4공학관 316호  
Email: [kimjs@hanyang.ac.kr](mailto:kimjs@hanyang.ac.kr)

# Textbook

**Functional Programming in Scala,**  
2<sup>nd</sup> Edition,  
Michael Pilquist,  
Rúnar Bjarnason, and  
Paul Chiusano,  
Manning, 2023



# Course Contents



## PART 1 INTRODUCTION TO FUNCTIONAL PROGRAMMING .....1

- Focus on course*
- 1 ■ What is functional programming? 3
  - 2 ■ Getting started with functional programming in Scala 14
  - 3 ■ Functional data structures 29
  - 4 ■ Handling errors without exceptions 48
  - 5 ■ Strictness and laziness 64
  - 6 ■ Purely functional state 78

## PART 2 FUNCTIONAL DESIGN AND COMBINATOR LIBRARIES .....93

- 7 ■ Purely functional parallelism 95
- 8 ■ Property-based testing 124
- 9 ■ Parser combinators 146

## PART 3 COMMON STRUCTURES IN FUNCTIONAL DESIGN .....173

- 10 ■ Monoids 175
- 11 ■ Monads 187
- 12 ■ Applicative and traversable functors 205

*Focus on next course*

## PART 4 EFFECTS AND I/O .....227

- 13 ■ External effects and I/O 229
- 14 ■ Local effects and mutable state 254
- 15 ■ Stream processing and incremental I/O 268

# Tools

- Scala Language 3.3.0
- SBT: Build tool, v.1.9.2
- IDE: VS Code or IntelliJ

The screenshot shows the official Scala website. At the top, there's a navigation bar with links for LEARN, INSTALL, PLAYGROUND, FIND A LIBRARY, COMMUNITY, and BLOG. A small note says "New on the blog: The Scala Toolkit". The main content area features a heading "The Scala Programming Language" with a brief description: "Scala combines object-oriented and functional programming in one concise, high-level language. Scala's static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries." Below this is a code editor window titled "Functional programming with immutable collections" containing Scala code for counting 'a's in a list of fruits. A "Run in playground" button is at the top right of the code editor. At the bottom of the page, there are two large buttons: "GET STARTED" and "LEARN SCALA", with a "Scala in a Nutshell" section between them.

# Switching from OOP to Functional Programming

Why functional programming is so hard?

# Imperative Programming

- Tell **how** to do it.



- **Mutability**

*Root of all Evil!*



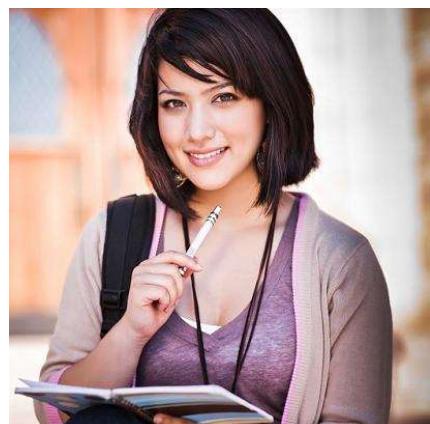
```
def qsort(xs: Array[Int], low: Int, high: Int) {  
    def swap(i: Int, j: Int) = {  
        val temp = xs(i)  
        xs(i) = xs(j)  
        xs(j) = temp  
    }  
  
    val pivot = xs((low + high) / 2)  
    var i = low  
    var j = high  
    while (i <= j) {  
        while (pivot < xs(j)) j -= 1  
        while (pivot > xs(i)) i += 1  
        if (i <= j) {  
            swap(i, j)  
            i += 1  
            j -= 1  
        }  
    }  
    if (low < j) qsort(xs, low, j)  
    if (i < high) qsort(xs, i, high)  
}
```

9

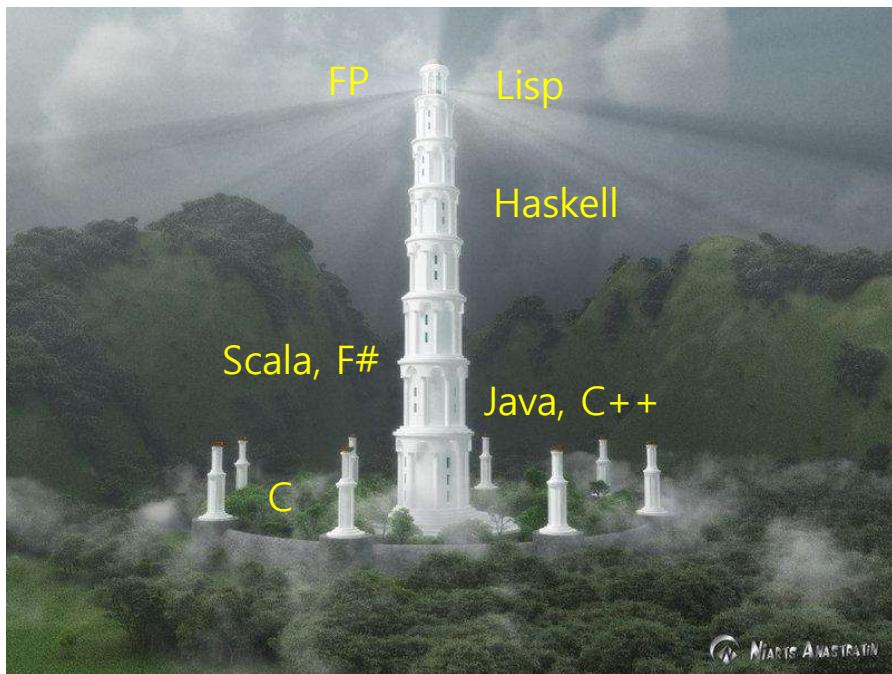
# Declarative Programming

- Tell **what** to do.
- **Immutability**

```
def qsort(xs: List[Int]): List[Int] =  
    if xs.size <= 1 then xs  
    else  
        val pivot = xs(xs.length / 2)  
        val (l, e, r) = partition(xs, pivot)  
        List.concat(qsort(l), e, qsort(r))
```

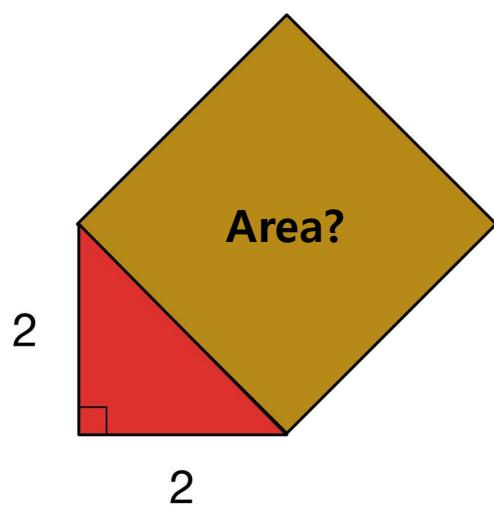


10



11

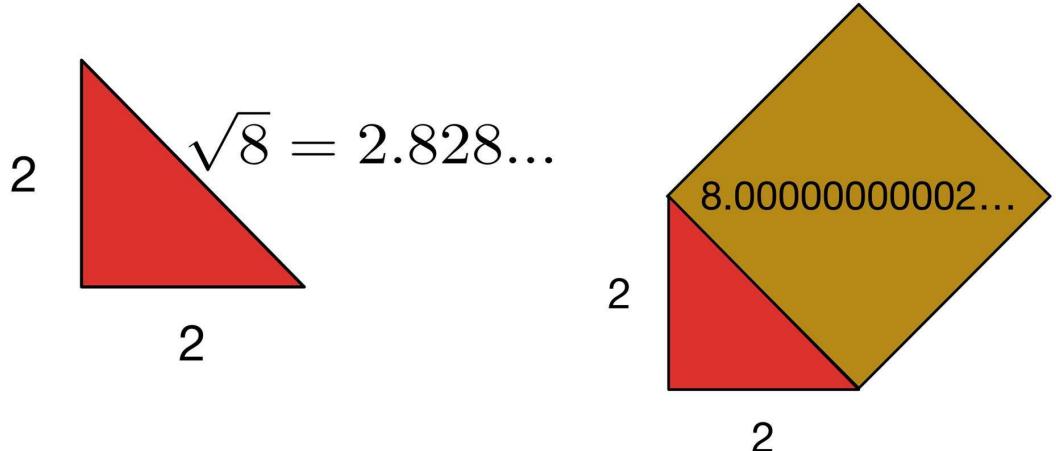
Does abstraction mean less accuracy?



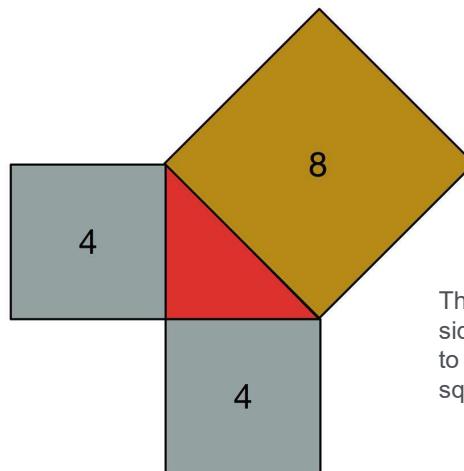
12

Is this accurate?

$$a^2 + b^2 = c^2$$



13



The area of the square whose side is the hypotenuse is equal to the sum of the areas of the squares on the other two sides.

Work with algebraic relationships.  
Go to decimals as late as possible.

14

# Common Complaints and Questions

*I've heard a lot of good things about functional programming  
but I find it very difficult to understand.*

*I have years of experience in C++/Java/C#/Javascript/etc but it  
doesn't help, it feels like learning to code from scratch again.  
Where should I start?*

15

## Switching paradigms requires a `mindset` change

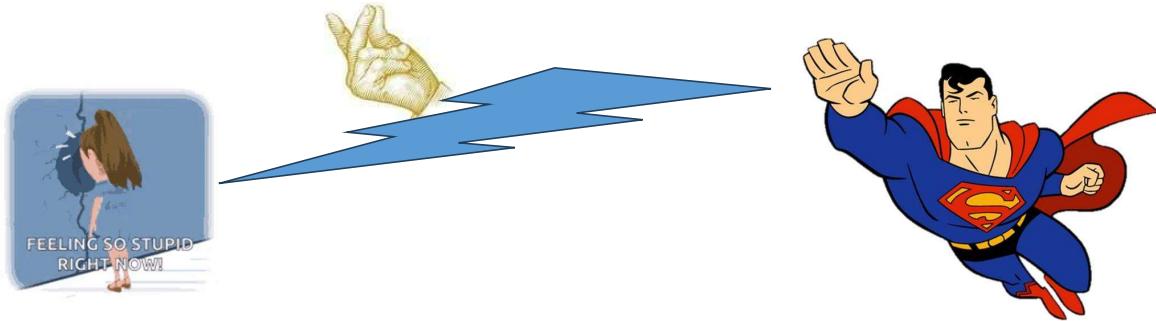
- No longer have your usual primitives, like classes, mutable variables, loops, etc.
- You will not be productive for the first couple of months, you will be stuck for hours or days on some simple things that usually took minutes.
- It will be hard and you will feel stupid.



16

# But after it clicks you will gain superpowers

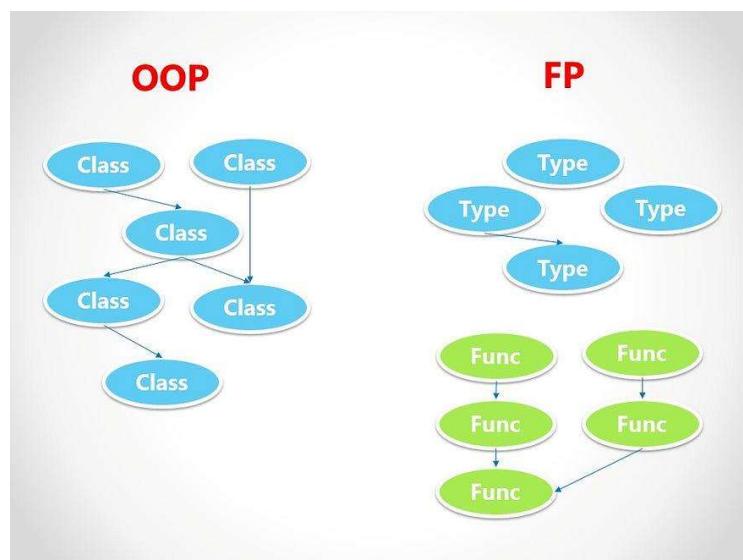
- I don't know a single person who switched back from FP to OOP after doing FP daily.
- You may switch to a language that doesn't have FP support but you'll still be writing using FP concepts, there are that good.



17

## 1. There are no classes

*Q: No classes? How do I structure my code then?*



*Q: What about data? I often have some data and functions that change that data in one class.*

- For that we have **Algebraic Data Types (ADT)**, which is just a fancy name for a record that holds data.

```
case class Person(name: String, age: Int)
```

```
// Using type constructor  
val person = Person("Bob", 42)
```

```
// Accessing fields  
val personsAge = person.age
```

```
data Person = Person String Int
```

-- OR

```
data Person = Person{ name :: String, age :: Int }
```

19

```
case class Person(name: String, age: Int)  
  
// Using type constructor  
val person = Person("Bob", 42)  
  
// Accessing fields  
val personsAge = person.age
```

```
data Person = Person  
{ name :: String, age :: Int , address :: Address }  
  
-- Using type constructor  
person = Person "Bob" 42  
  
-- Or using records notation  
person = Person {name = "Bob", age = 42}  
  
-- Accessing fields  
personsName = name person -- "Bob"
```

*Q: Ok, but how do I change the person's age?*

```
val bob = Person("Bob", 42)  
  
// .copy provided by the 'case class'  
val olderBob = bob.copy(age = 43)
```

```
bob = Person "Bob" 42  
  
olderBob = bob { age = 43 }
```

20

# ADTs: product type and sum type

- **Product type:** a collection of fields, all have to be specified to construct a type

```
// Person is a product type consisting of 3 fields
case class Person(
  name: String,
  age: Int,
  address: Address)

// Address on its own also is a product type
case class Address(
  country: String, city: String, street: String)

// In order to create a Point
// you have to provide all 3 arguments
case class Point(x: Double, y: Double, z: Double)
```

```
data Person = Person
  { name :: String, age :: Int, address :: Address}

data Address = Address
  { country :: String
  , city :: String
  , street :: String
  }

data Position = Position
  { x :: Integer
  , y :: Integer
  , z :: Integer
  }
```

21

# ADTs: product type and sum type

- **Sum type:** *discriminated union*

```
// Scala doesn't have a nice syntax for sum types so
// it looks like a familiar OOP inheritance tree.
sealed trait Shape
case class Circle(radius: Int) extends Shape
case class Square(side: Int) extends Shape

// Scala 3 only
enum Shape:
  case Circle(radius: Int)
  case Square(side: Int)
```

-- The | can be read as 'or'  
data Shape = Circle Int | Square Int  
  
-- Alternatively  
data Shape = Circle { radius :: Int }  
| Square { side :: Int }

22

## *Q: Why sums and products are so special?*

- They are basic building blocks for modeling. Product types can be deconstructed and statically checked while sum types can be used for pattern matching.

```
enum Shape:  
  case Circle(radius: Int)  
  case Rectangle(width: Int, height: Int)  
  
// 'match' keyword allows us to pattern match on a  
// specific option of the sum type  
def area(shape: Shape): Double = shape match {  
  case Circle(r) => math.Pi * r * r  
  case Rectangle(w, h) => w * h  
  // Note how rectangle's width and height  
  // are captured in 'w' and 'h'.  
}
```

```
data Shape = Circle { radius :: Double }  
           | Rectangle { width :: Double  
                         , height :: Double }
```

```
-- In Haskell different options of a sum type can  
-- be handled with different 'functions'  
area :: Shape -> Double  
area (Circle r) = pi * r * r  
-- width and height are captured in 'w' and 'h'  
area (Rectangle w h) = w * h
```

23

## Shared Languages (Domain Driven Design)

```
enum Suit { case Spades, Hearts, Diamonds, Clubs }  
enum Rank {  
  case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King, Ace  
}  
  
case class Card(suit: Suit, rank: Rank)  
case class Deck(cards: Seq[Card])  
case class Hand(cards: Seq[Card])  
case class Player(name: String, hand: Hand)  
case class Game(deck: Deck, players: Seq[Player])  
type Deal = Deck => (Deck, Hand)  
type PickupCard = (Hand, Card) => Hand;
```

Could non-programmer  
understand this?

24

# Code should sync with Real World Vocabulary

## In the Real World

- Suit
- Rank
- Card
- Deck
- Hand
- Player
- Game
- Deal
- ShuffledDeck
- Shuffle

*The design is the code,  
The code is the design.*

## In the Code

- Suit
- Rank
- Card
- Deck
- Hand
- Player
- Game
- Deal
- ShuffledDeck
- Shuffle

Should not use  
programmer's jargon



PlayerManager  
DeckBase  
AbstractCardProxyFactoryBean

25

## 2. All you need is a function

- Meet your new best friend — a **function**.
- In FP a function is always just a function — it takes values as input and returns values as output.
  - You may know it by different names: getter, setter, constructor, method, builder, static function, etc in OOP.
- There is no need to instantiate anything to use functions, you just import the module where the function is defined and just call it.
- A functional program is just a **collection of ADTs and functions**, as in Shapes example above.

26

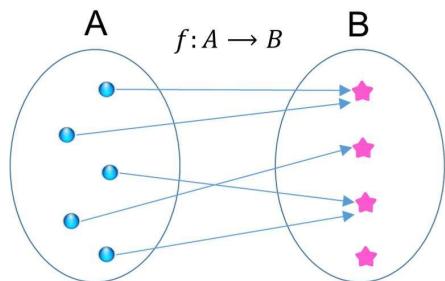
# Pure Functions

FP means programming with **pure functions**.

A pure function is one with **no side effects**.

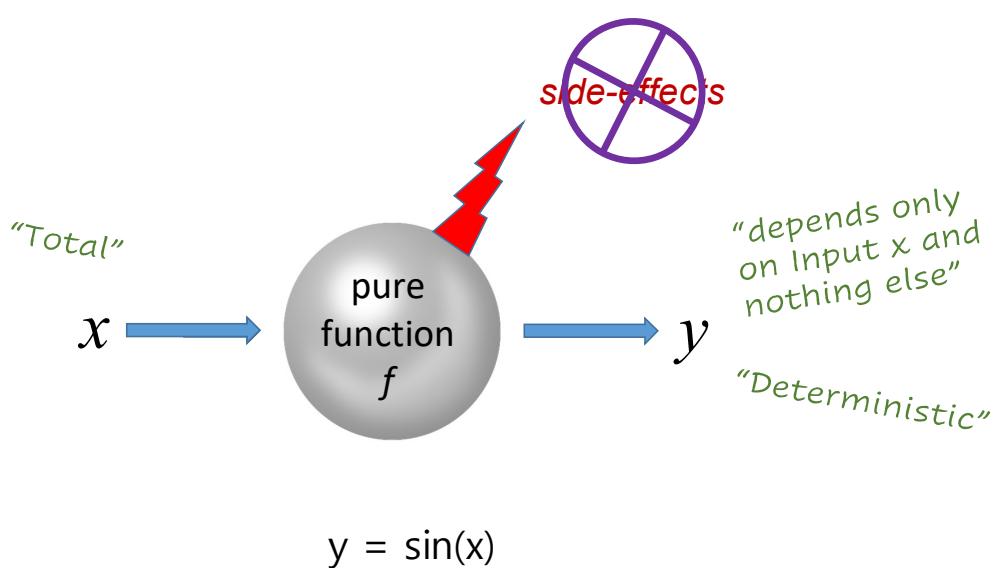
$f : A \Rightarrow B$

- 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  $a$ .



A **pure function** has **no observable effect** on the execution of the program **other than to compute a result given its inputs**.

27



$$y = \sin(x)$$

28

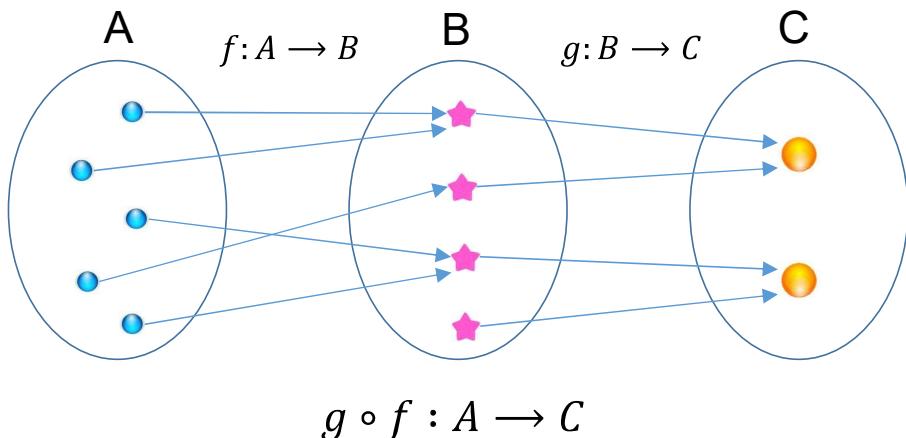
# 3 main properties a function should have

1. **Pure:** *no side effects.* Functions are not allowed to do more than their type definition says.
  - cannot change global variables, access filesystem, do network requests, etc
2. **Total:** *returns values for all inputs.* Functions that crash or throw exceptions on some inputs are not total or partial.
  - e.g., integer divide by zero => partial function
3. **Deterministic:** *always returns the same result for the same input.* For deterministic function it doesn't matter how and when it's called
  - Functions that depend on a current date, clock, timezone or some external state are not deterministic.

*It's programmer responsibility to satisfy those properties.*

29

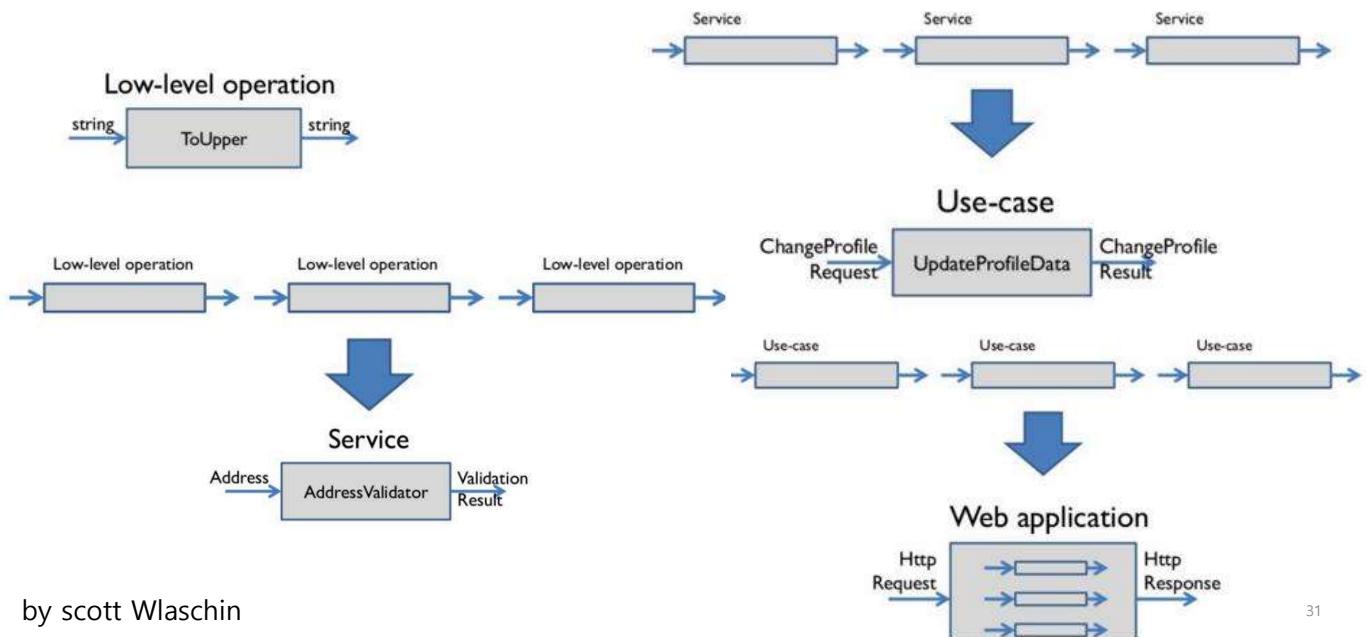
## Functions and Types (Morphisms and Sets)



30

# Modular and Scale

Composition is everywhere!



by scott Wlaschin

31

*Q: Why do I care if a function has these properties or not?*

- If a function satisfy those properties you get “**referential transparency**”.
- In short, you’ll get the ability to look at the function type definition and know exactly what it can and cannot do.
- You can refactor your code *fearlessly* because RT guarantees you nothing will break.
- RT is basically what allows us to control complexity of our software.

32

### 3. No, you can't change a variable

*Q: This is the weirdest part, how do I make anything useful without changing variables?*

- **Variables are immutable** and used only to alias or label values, not as a physical reference or a pointer to the actual data.

33

*Q: Why not just change it in place?*

- Because it breaks referential transparency and referential transparency is the key to FP.
  - It will make your life so much easier while not having mutable variables is a fair price to pay.
  - Besides, no mutation means you get **thread safe code for free**, no more weekends wasted on ‘happens only on a Tuesday evening’ concurrency bugs.
- Immutability is a simple concept but it’s hard to adopt after years of OOP experience.
  - It’s common to see people reverting to **vars** in Scala just ‘to get this thing working’..
  - Besides, there is no such ‘hack’ in Haskell so you have to be immutable from day 1.

34

## 4. No, you can't do 'for' loops

*Q: Our bread and butter — the 'for' loop — you say FP doesn't have it as well? How do you iterate over an array?*

- We have other means of achieving the same — **recursion** and **higher order functions (HOFs)**.

```
def sum(as: List[Int]): Int = as match
  // Nil is a type constructor for an empty list
  case Nil => 0
  case x :: xs => x + sum(xs)

sum(List(1,2,3,4,5)) // 15
```

```
-- [] is a type constructor for an empty list
mySum :: [Int] -> Int
mySum [] = 0
mySum (x : xs) = x + (mySum xs)
```

35

## Higher Order Functions

- Higher order functions take other functions as an argument.

```
// Pass a function to transform every item in the list
List(1,2,3,4,5).map(n => n + 1) // List(2,3,4,5,6)

// Or shorter syntax
List(1,2,3,4,5).map(_ * 2) // List(2,4,6,8,10)

// Fold example. Sum all the values in a list
List(1,2,3,4,5).fold(0)((l, r) => l + r) // 15

// Multiply each number by 2, convert to string and
// produce a total string appending results together
List(1,2,3,4,5).foldLeft("")((acc, number) =>
  acc ++ (number * 2).toString) // "246810"
```

```
-- Add 1 to every item in the list
map (+1) [1,2,3,4,5] -- [2,3,4,5,6]

-- Sum all the values
foldl' (+) 0 [1,2,3,4,5] -- 15

-- same as left in Scala
foldl' (\acc n -> acc ++ show (n * 2)) "" [1,2,3,4,5]
-- "246810"
```

36

## 5. On nulls and exceptions

- **null** is a lower-level details often used to represent *absence* or some sort of internal *failure* that prevents function from returning a proper value.
- **In FP, it is possible to represent absence or failures as an ordinary value.**
- The same is true with exceptions.
- In your pure program it's possible to *represent failures and exceptions with ordinary values*, too.
- Throwing exceptions makes function *partial* which again breaks referential transparency and creates problems.

37

## FP style of null and/or Exceptions

```
// A simple sum type that has two options
enum Option[A]:
    case Some(value: A)
    case None

// Instead of throwing an exception or crashing
// the function will return a value.
def divide(dividend: Int, divisor: Int): Option[Int] =
    if divisor == 0 then None
    else Some(dividend / divisor)
```

```
-- Simple sum type with two options
data Maybe a = Just a | Nothing

-- Safe total function that never throws
divide :: Int -> Int -> Maybe Int
divide dividend divisor =
    if divisor == 0 then Nothing
    else Just (dividend / divisor)
```

38

## 6. Your code is not a sequence of statements anymore

- In imperative language you write a program using **statements**.
- Statements are “**actions**” to do something including side-effects.
- Once you call such program — it’s done, completed, executed.

```
def main: Unit = {  
    println("Good morning, what is your name?")  
  
    val name = scala.io.StdIn.readLine()  
  
    println(s"Nice to meet you, $name!")  
}
```

39

## Statements vs. Expressions

- In functional programs, a program consists of **expressions**.
- Each expression evaluates to a (pure) **value**., no side-effecting allowed.
- Therefore, a program is merely a “**description**” of what should happens.
- Nothing is executed until the very last moment.

40

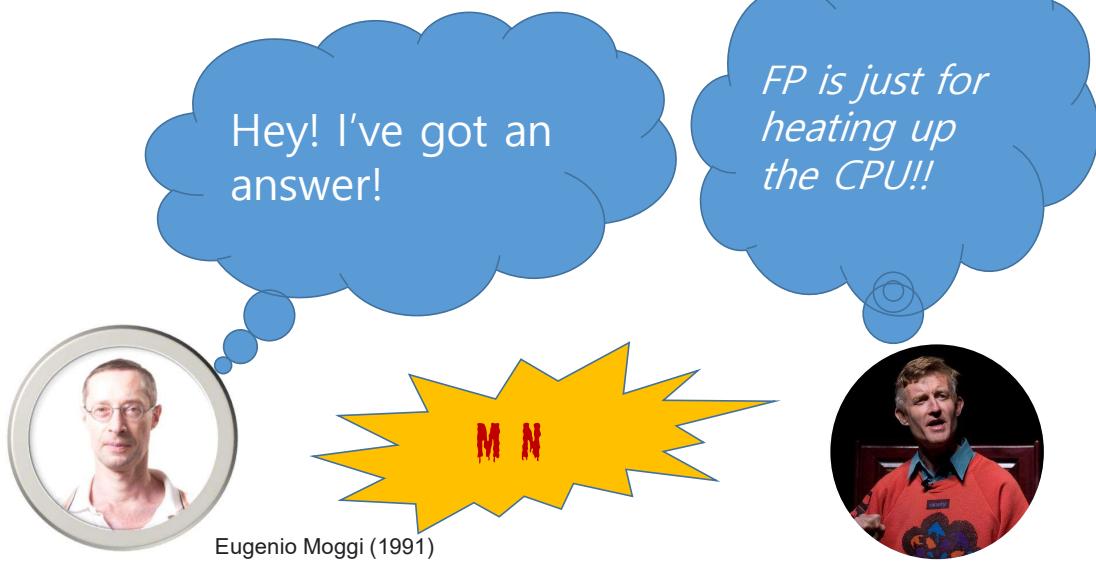
# Side Effects

- 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
- etc.



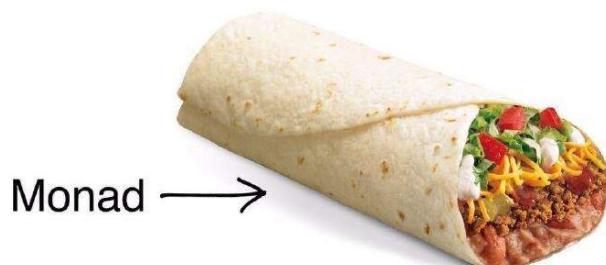
41

# Side Effects



42

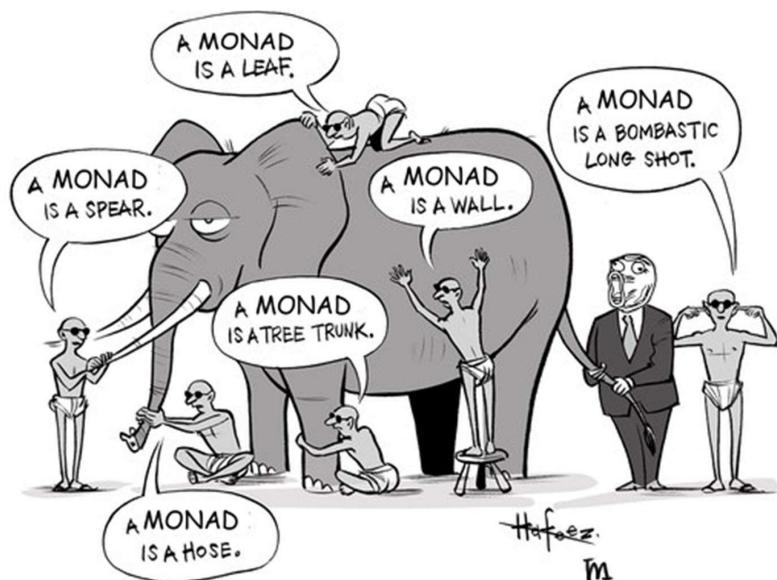
# Monads are like Burritos



From <https://twitter.com/monadburritos/status/915624691829587968>

43

Let's face it: all monad tutorials suck. So here's another one:



44

A monad is just a monoid  
in the category of endofunctors.



What's the problem?

45

## Turn Procedural Effect to Functional Effects

$\text{IO}[\text{A}]$



A description of an I/O effect that when unsafely interpreted, will succeed with a value of A

- Functional effects are immutable data structures that merely describe sequence of operations.
- At the end of the world, the data structure has to be impurely “interpreted” to real world effects.

46

# In FP, a program is a description of effects

```
def main: IO[Unit] =  
  // Nothing happens  
  val program = getDescription()  
  
  // When interpreted, effects materialized  
  program.unsafeRun()  
}  
  
def putStrLn(line: String): IO[Unit] = IO(println(line))  
  
def getStrLn: IO[String] = IO(scala.io.StdIn.readLine())  
  
def getDescription: IO[Unit] =  
  for  
    _   <- putStrLn("Good morning, what is your name?")  
    name <- getStrLn  
    _   <- putStrLn(s"Nice to meet you, $name!")  
  yield ()
```

47

```
def getDescription: IO[Unit] =  
  for  
    _   <- putStrLn("Good morning, what is your name?")  
    name <- getStrLn  
    _   <- putStrLn(s"Nice to meet you, $name!")  
  yield ()
```



**for-comprehension** is a syntactic sugar  
for writing programs in imperative style

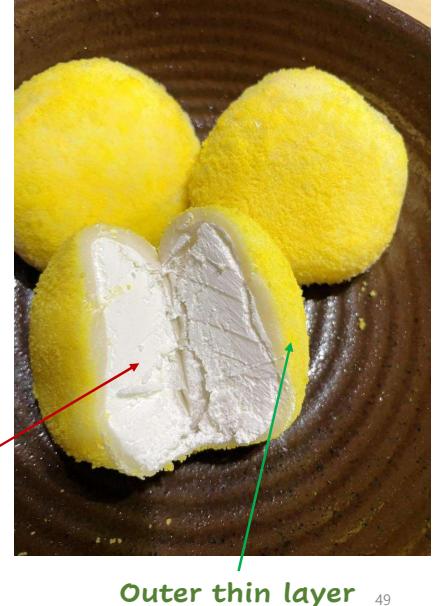
```
def getDescription: IO[Unit] =  
  putStrLn("Good morning, what is your name?").flatMap { _ =>  
    getStrLn.flatMap { name =>  
      putStrLn(s"Nice to meet you, $name!").map { _ =>  
        ()  
      }  
    }  
  }
```

48

# Functional Programming

생크림 합쌀떡 프로그래밍

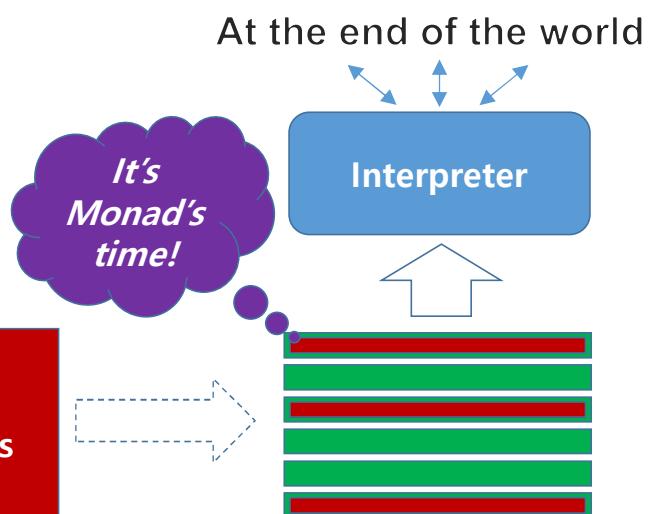
- Over the course of “next” course, we’ll learn how to express all of our programs without side effects, and that includes programs that perform I/O, handle errors, and modify data.
- **General guideline:** Implement programs with a pure core and a thin layer on the outside that handles effects.



49

## Pits of Success

- Functional Programming
- Domain Driven Design
- Hexagonal Architecture



## 7. Functors, Monads, Applicatives?

*Q: I hear FP people talk about this things constantly but they don't make any sense to me. Is there an easy explanation?*

- **Functors, Monads** and **Applicatives** are pretty powerful and **common abstractions** borrowed from *Category Theory*.
- You will see them everywhere.
- 1. But, get comfortable with function composition, higher order functions and polymorphic functions.
- 2. Then learn **type classes**.
- 3. After that **Functors** and **Monads** should come naturally.

51

## Functional programming

- FP is a *declarative-style* programming.
- Functional programs are composed of nested *pure functions*.
- A pure function is one with *referential transparency* (and therefore *no side effects*).
- Everything is basically *immutable*.
  - Use *recursions* instead of loops
- FP is based on *lambda calculus* and *category theory*.

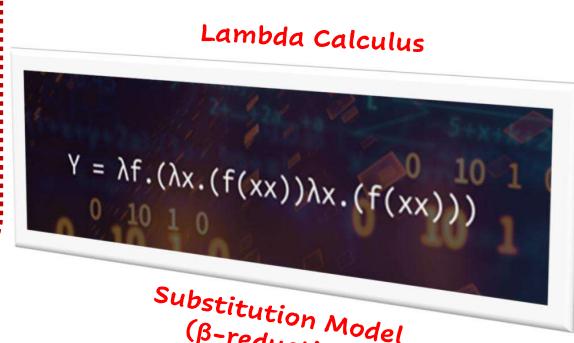


Alonzo Church  
(1903-1995)

52

# Core Concepts of Functional Programming

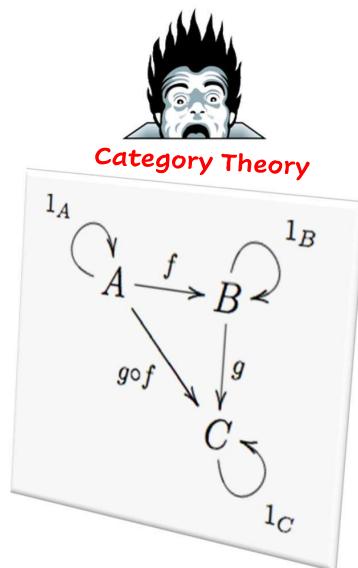
Pure Functions  
Referential Transparency  
Composition  
High-Order Functions  
Immutability  
Currying  
Recursions  
Parametric Polymorphism  
Algebraic Data Types  
Higher Kind Types  
Lazy evaluation



53

## Functional Patterns

Semigroup/Monoid  
Functor  
Applicative  
Monad  
Natural Transformation  
 $\{\text{Cata, Ana, Hylo}\}$  morphisms  
Lens  
Adjunction  
 $F$ -Algebra



54

# A Scala Program with Side Effects

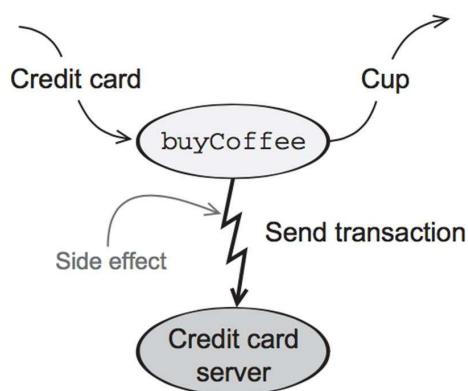
```
class Cafe {  
    def buyCoffee(cc: CreditCard): Coffee = {  
        val cup = Coffee()  
        cc.charge(cup.price)  
        cup  
    }  
}
```

**Side effect.**  
**Actually charges the credit card.**

Charging a credit card involves some interaction with the outside world.

55

## Problem Due to Side Effects



Can't test `buyCoffee` without credit card server.

Can't combine two transactions into one.

The code is difficult to test.  
It's difficult to reuse `buyCoffee`.

56

# Adding a Payment Object ... but ...

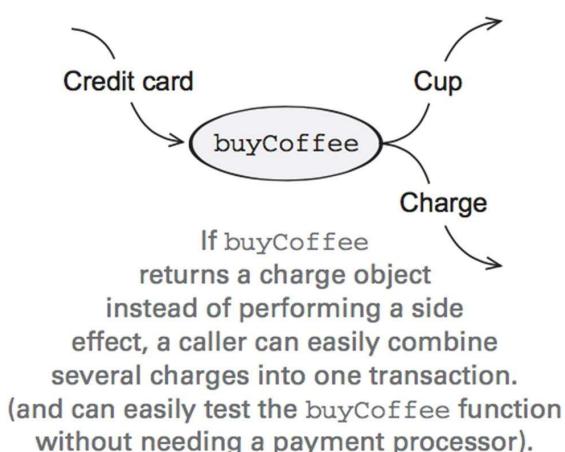
```
class Cafe {  
    def buyCoffee(cc: CreditCard, p: Payments): Coffee = {  
        Side effect. val cup = Coffee()  
        p.charge(cc, cup.price)  
        cup  
    }  
}
```

It's still difficult to reuse `buyCoffee`.

Though side effects still occur when we call `p.charge(cc, cup.price)`, at least regained some testability via **Payment interface** or mock objects.

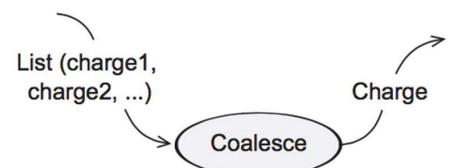
57

# Functional Solution



The functional solution is to eliminate side effects and have **buyCoffee return the charge as a value** in addition to returning the **Coffee**.

Separated the concern of *creating* a charge from the *processing* or *interpretation* of that charge.



58

# Functional Solution

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

59

## Charge as a First-Class Value

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

60

# Additional Benefit of First-Class Value

Making **Charge** into a first-class value has other benefits:

- more easily assemble business logic for working with these charges

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

61

# 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** without affecting the meaning of **p**.

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

62

```

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    ← r1 and r2 are the same.
r2: String = dlrow ,olleH

```

Suppose we 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    ← r1 and r2 are still the same.
r2: String = dlrow ,olleH

```

What's more, r1 and r2 are referentially transparent as well.

63

```

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.

```

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    ← r1 and r2 are no longer the same.

```

We conclude that **StringBuilder.append** is  
not a pure function.

64

## Non RT Example

For `buyCoffee` to be pure, it must be the case that

$$p(\text{buyCoffee}(\text{aliceCreditCard})) == p(\text{Coffee}())$$

for any  $p$ .

```
class Cafe {  
    def buyCoffee(cc: CreditCard): Coffee = {  
        val cup = Coffee()  
        cc.charge(cup.price) // return value is ignored here  
        cup  
    }  
}
```

65

## RT and Substitution Model

- Referential transparency forces the invariant that **everything a function does is represented by the value that it returns**, according to the result type of the function.
- This constraint enables a *simple and natural mode of reasoning* about program evaluation called the **substitution model**.

66

# Benefits of Purity, Substitution Model

- A pure function is modular.
  - because it separates the logic of the computation itself from “*what to do with the result*” and “*how to obtain the input*”; it’s a black box.
- Modular programs allows local reasoning and independent reuse.
  - the meaning of the whole depends only on the meaning of the components and the rules governing their composition
- Hence, pure functions are **composable**.

67



- Rich Hickey
- Creator of Clojure

“Simplicity is prerequisite  
for reliability”  
- Edsger W. Dijkstra



## Why Functional Programming matters?

*Referential transparency that allows you to modularize your program in new ways, and helps answer these questions by making things as simple as possible!*

69

### OO patterns/principles

- Single Responsibility Principle
- Open Closed Principle
- Interface Segregation Principle
- Dependency Inversion Principle
- Factory Pattern
- Strategy Pattern
- Decorator Pattern
- Visitor Pattern
- ...

### FP equivalents

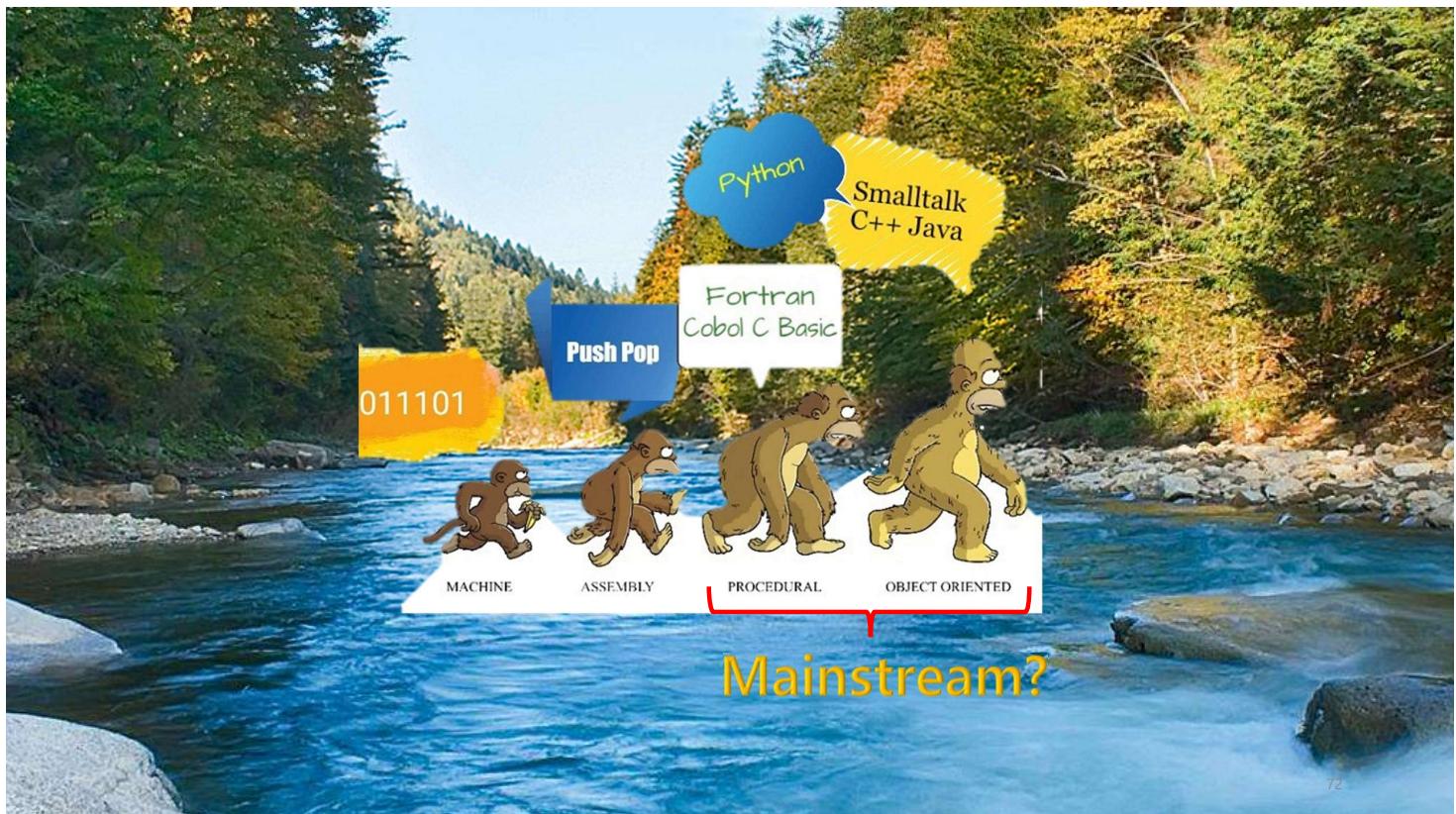
- Functions
- Functions
- Functions also
- Functions
- You will be assimilated!
- Function again
- Functions
- Resistance is futile!
- ...

# Summary: Benefits of Functional Programming

Enables **local reasoning** and is more **modular** and **composable**.

- Easier to test
- Easier to reuse
- Easier to **parallelize**
- Easier to optimize
  - Compile-time optimization, Lazy evaluation, Memoization, etc.
- Less prone to bugs
- Easier to generalize

71



72

## But unfortunately it's often more like this



How many FP  
people see OOP



How many OOP  
people see FP

↑  
And that's where we are ☺

Odersky's slide

73



74