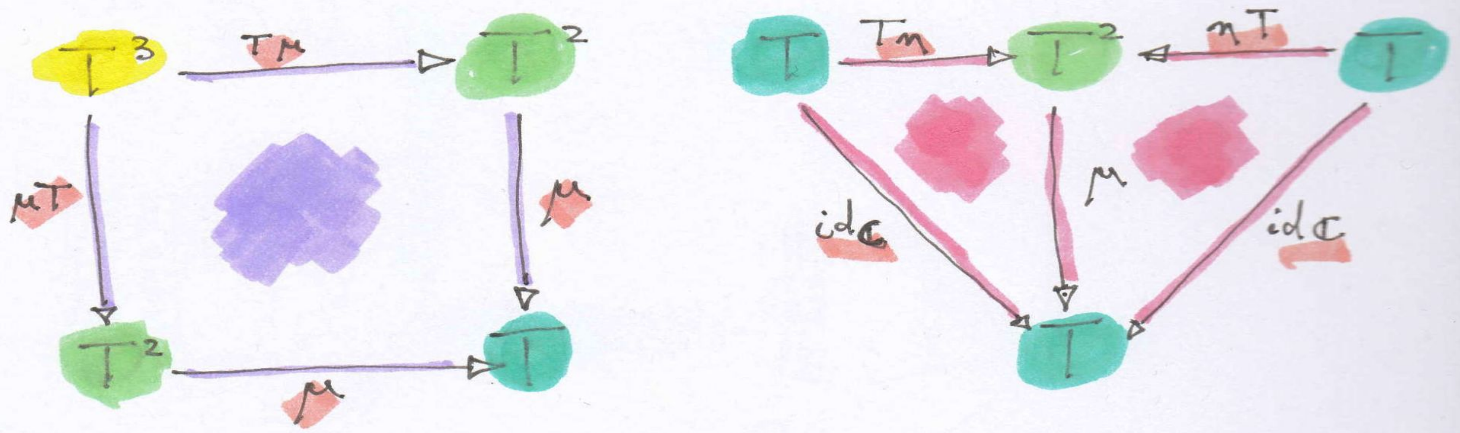
The 4 Rules of Simple Design and Category Theory

Uberto Barbini
@ramtop

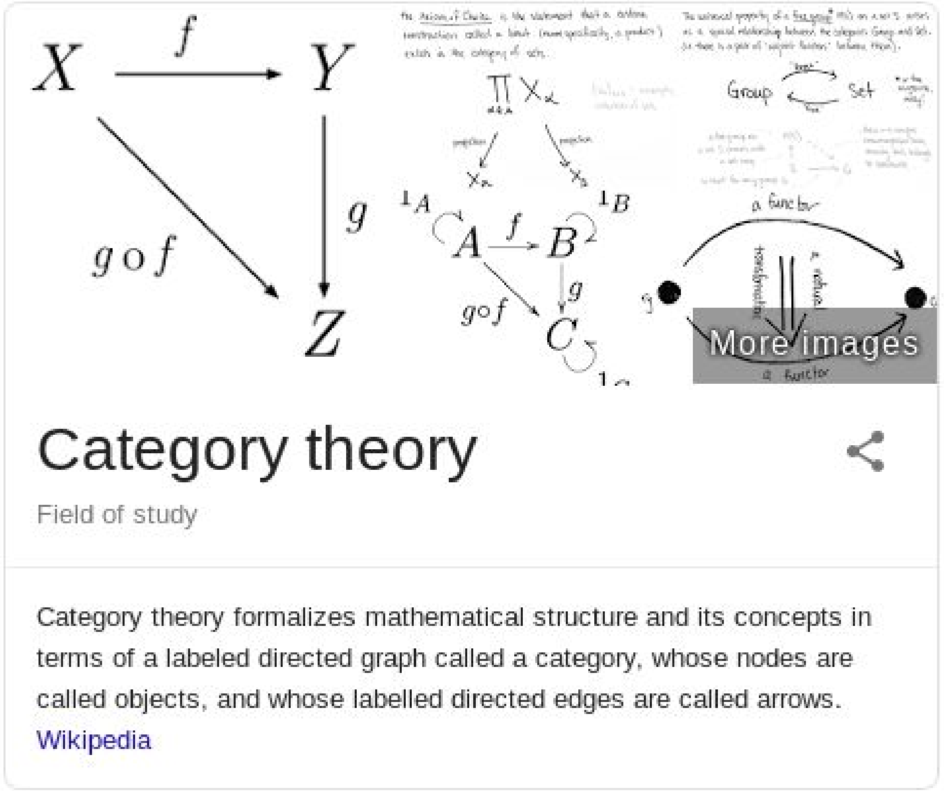


$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, "T\eta"] \arrow[dr, "id_{\mathbb{C}}"'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "\eta T"'] \arrow[dl, "id_{\mathbb{C}}"] \\
 & T &
\end{tikzcd}$$

An Innocent Question:

Why a programmer passionate about Code Quality should care about Category Theory?

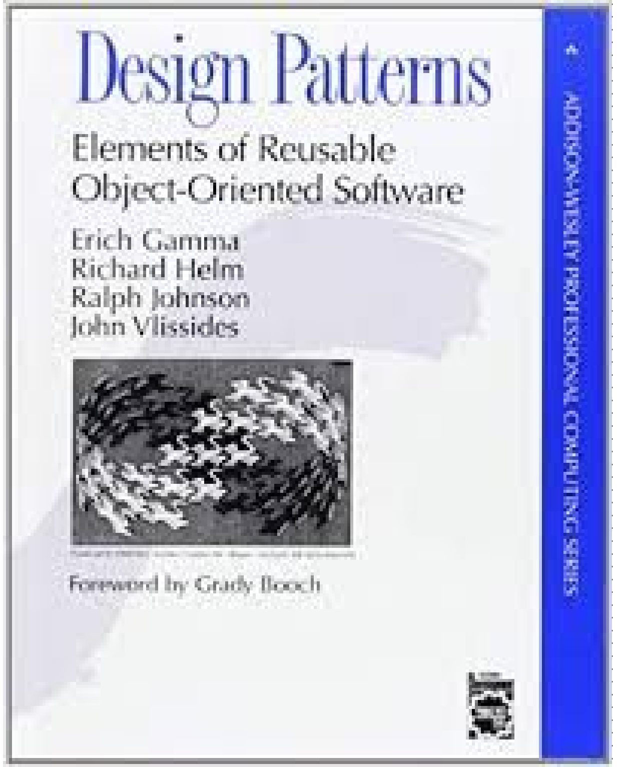
TL;DR


Category theory

Field of study

Category theory formalizes mathematical structure and its concepts in terms of a labeled directed graph called a category, whose nodes are called objects, and whose labelled directed edges are called arrows.
Wikipedia

Functional


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Object-Oriented

Code Quality and Categories

Category Theory helps us

To improve our functional design

~~Code Quality and Categories~~

Why Functional Programming?

How Categories can help?

The Alternative

Kent Beck’s
Four Rules of Simple Design:

Passes the tests
(It works)
Reveals intention
(Easy to read and understand)
No duplication
(DRY: Don’t Repeat Yourself)
Fewest elements
(remove anything that doesn't serve the three previous rules)

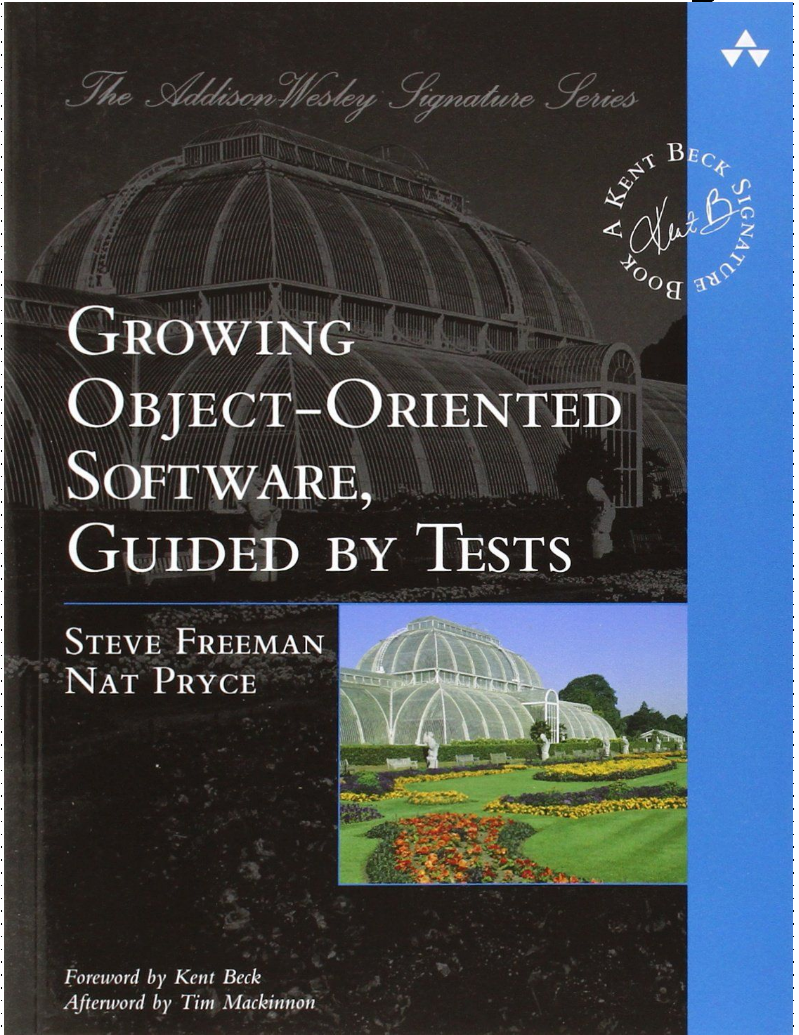
Functional Programming is a programming **paradigm** that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

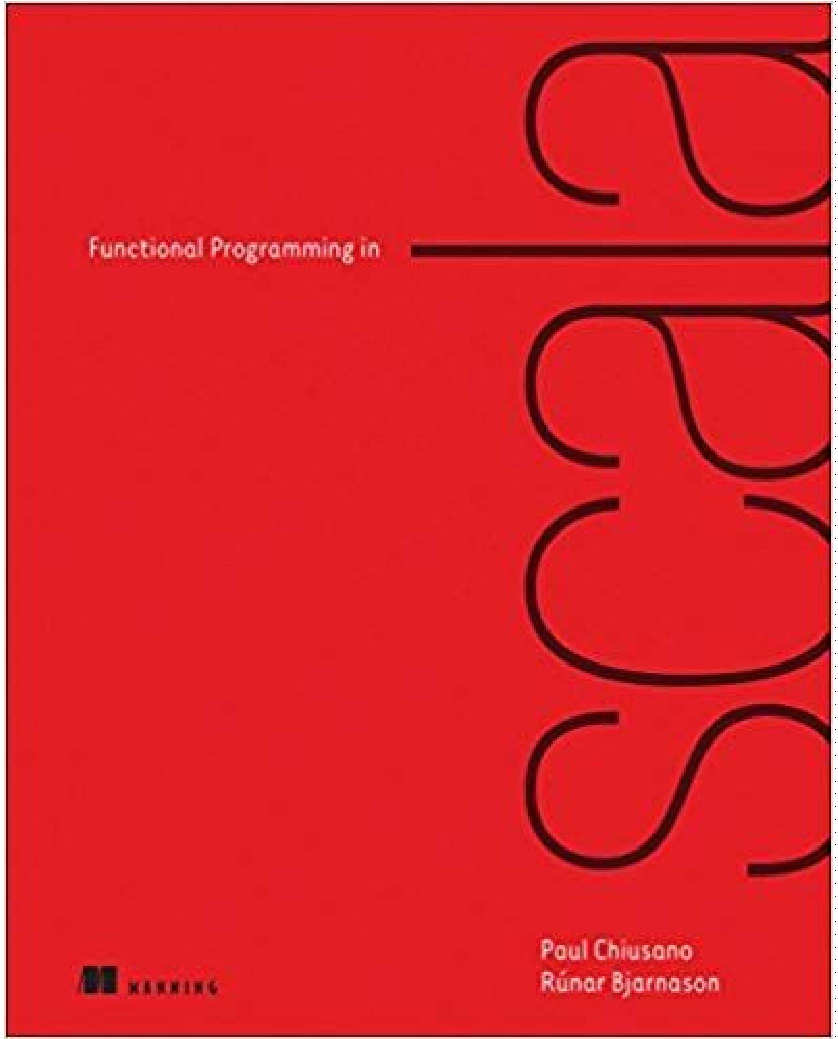
Wikipedia

Alan Kay
Smalltalk
Conversations

John McCarthy
Lisp
Transformations

A Story of Two Books


The Addison-Wesley Signature Series
A Kent Beck Signature Book
Growing Object-Oriented Software, Guided by Tests
Steve Freeman
Nat Pryce
Foreword by Kent Beck
Afterword by Tim Mackinnon



Category Tinted Glasses

The proof of the pudding is in the eating

The proof of the Functional Programmer is in the coding

What we need is:

Higher Order Functions
Immutable Data
Lazyness

But at the end…
it’s all about morphisms

“If I haven’t convinced you yet that category theory is all about morphisms then I haven’t done my job properly.”
Bartosz Milewski

https://bartoszmilewski.com/2015/11/17/its-all-about-morphisms/

Morphism examples in code

```
Date → String

User → Date

String → Int

(User → Date) → (User → String)
```

Morphism examples in code

```
Date → String
dateFormat()
User → Date
getBirthday()
String → Int
length()
(User → Date) → (User → String)
```

A **Category** is:

1) a collection of "objects"

2) a collection of "arrows"

3) each arrow works on a object domain$^{(A)}$ and a object codomain$^{(B)}$

$$\begin{tikzcd} A \arrow[r, bend left, "f"] & B \\ \text{dom} & \text{codom} \end{tikzcd}$$

4) a composition operator on arrows

$$\begin{tikzcd} A \arrow[r, bend left, "f"] & B & B \arrow[r, bend left, "g"] & C & C \arrow[r, bend left, "h"] & D \end{tikzcd}$$

$\Rightarrow$ $f\circ(g\circ h)$

$(f\circ g)\circ h$

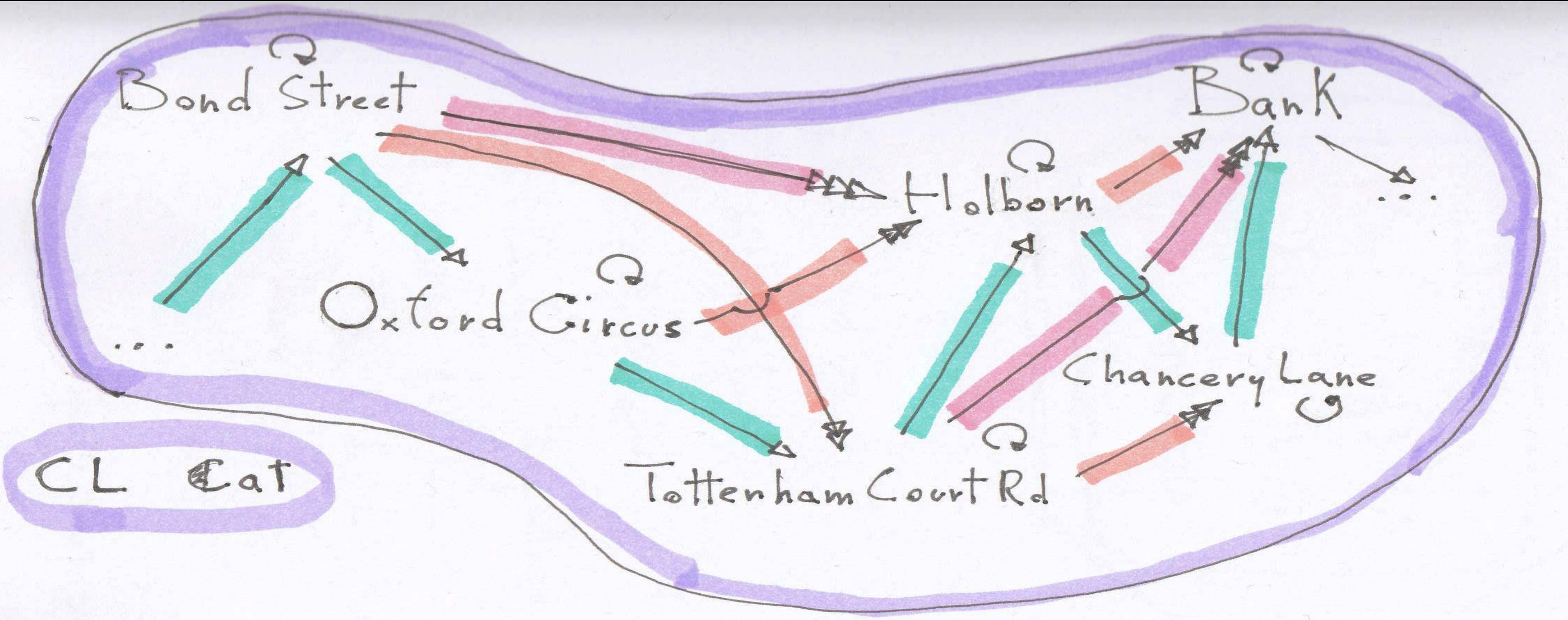
5) an identity arrow for each object

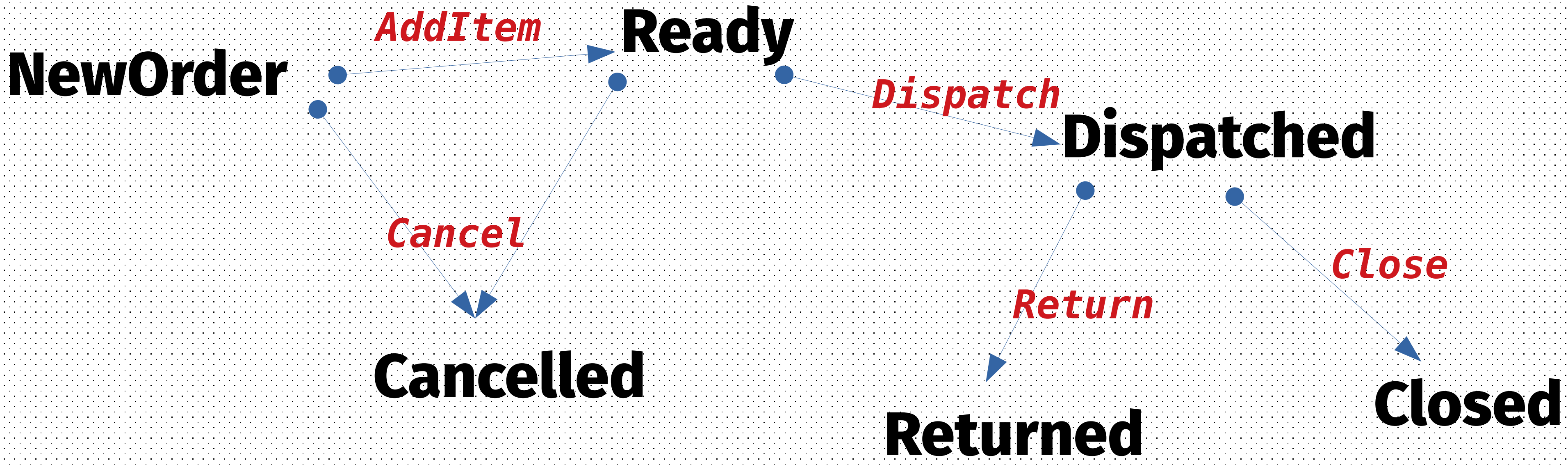
$$\begin{tikzcd} A \arrow[loop above, "idA"] & A \arrow[r, bend left, "f"] & B & B \arrow[loop above, "idB"] \end{tikzcd}$$

$\Rightarrow$ $f = f\circ id_A$

$f = f\circ id_B$

London Tube Category



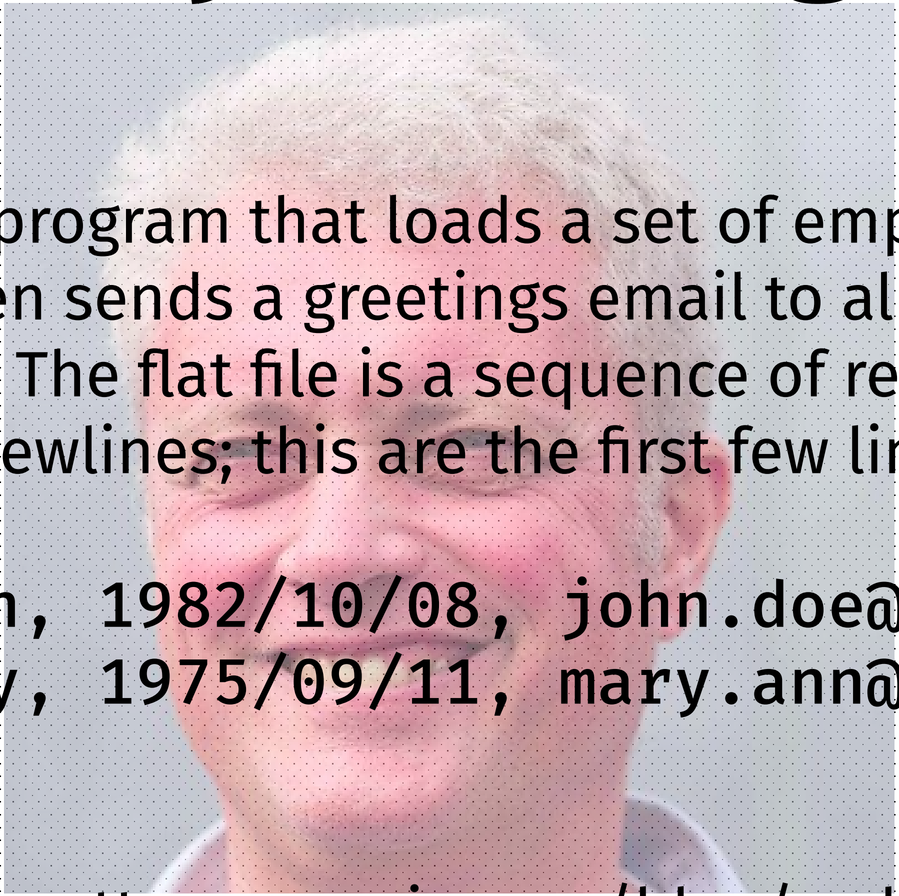


Event Source Category

for pizza delivery service

https://skillsmatter.com/skillscasts/11486-functional-cqrs

Birthday Greetings Kata



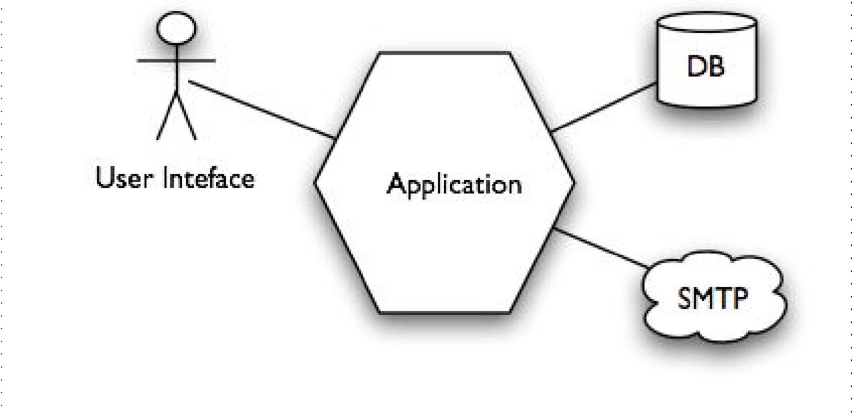
Problem: write a program that loads a set of employee records from a flat file and then sends a greetings email to all employees whose birthday is today. The flat file is a sequence of records, separated by newlines; this are the first few lines:

```
Doe, John, 1982/10/08, john.doe@foobar.com
Ann, Mary, 1975/09/11, mary.ann@foobar.com
```

http://matteo.vaccari.name/blog/archives/154

Hexagonal OO Design

Problem: write a program that loads a set of employee records from a flat file and then sends a greetings email to all employees whose birthday is today. The flat file is a sequence of records, separated by newlines; this are the first few lines:



Categories Functional Design

Our weapons:

Immutable Types
Pure Functions
Laziness
No Exceptions

Define Arrows From the Outside

```
Filename → Emails sent
```

Define Arrows From the Outside

```
Filename → Emails sent

Filename → Text → EmailData → Emails sent
```

Define Arrows From the Outside

```
Filename → Emails sent

Filename → Text → EmailData → Emails sent

… → Text → CSVrows → Employee → EmailData → …
```

Define Arrows From the Outside

```
Filename → Emails sent

Filename → Text → EmailData → Emails sent

… → Text → CSVrows → Employee → EmailData → …

… → CSVrows → Employee → isTodayBirthday → …
```

$$\begin{tikzcd}
\text{Filename} \arrow[dr] & & & & & \\
& \text{TextFile} \arrow[dr] & & & & \\
& & \text{CSVrow} \arrow[dr] & & & \\
& & & \text{Employee} \arrow[dr] & & \text{Today} \arrow[dl] \\
& & & & \text{TodayBirthday} \arrow[dd] & \\
& & & & & \text{GreetingTemplate} \arrow[dl] \\
& & & & \text{EmailData} \arrow[dr] & \\
& & & & & \text{Emails sent}
\end{tikzcd}$$

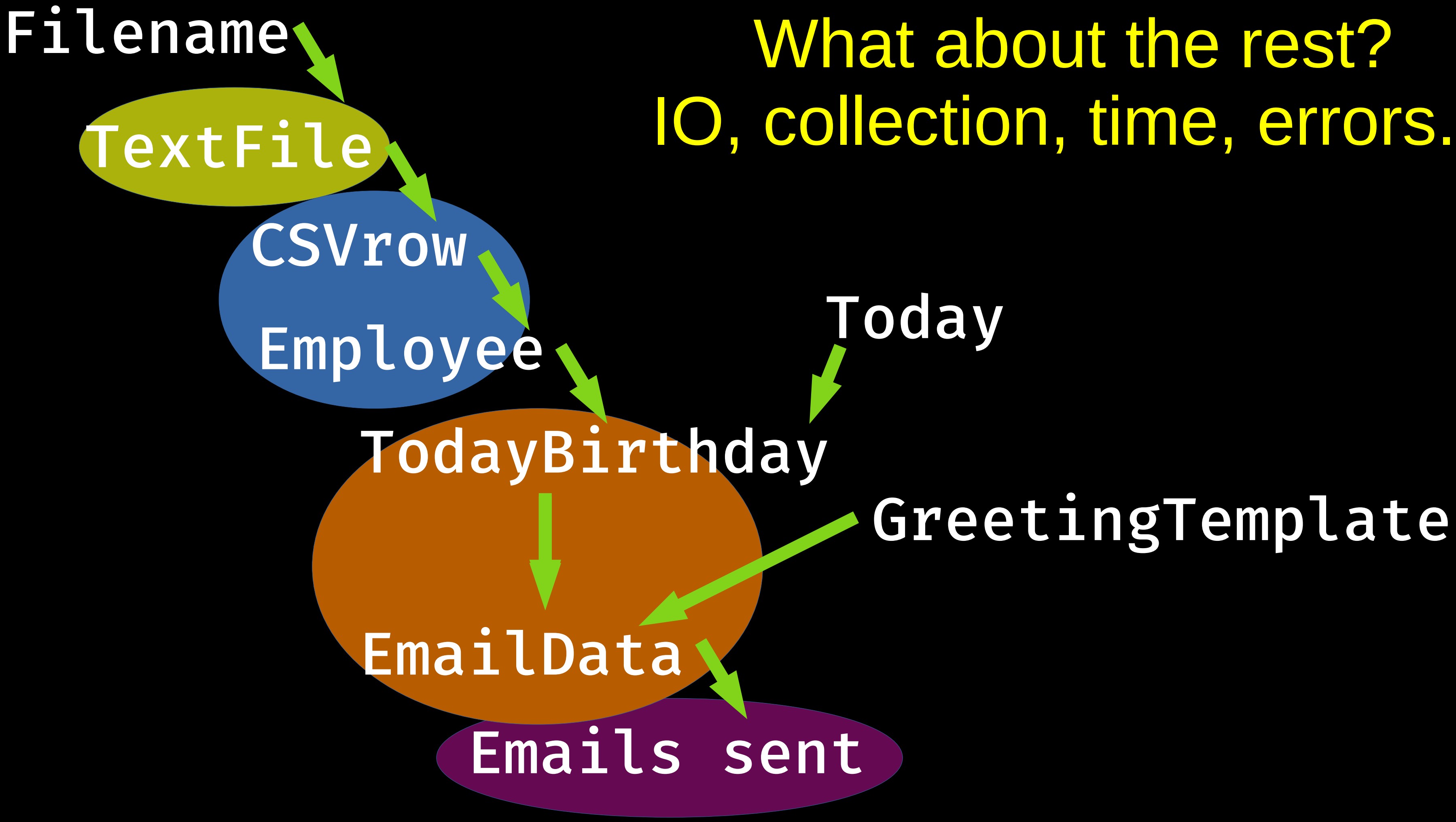
Objects → Types

```
data class Employee(val firstName: String,
                    val lastName: String,
                    val dateOfBirth: LocalDate,
                    val email: EmailAddress)


data class Email (val recipient: EmailAddress,
                  val subject: String,
                  val text: String)

inline class EmailAddress(val raw: String)

inline class CsvRow(val raw: String)
```

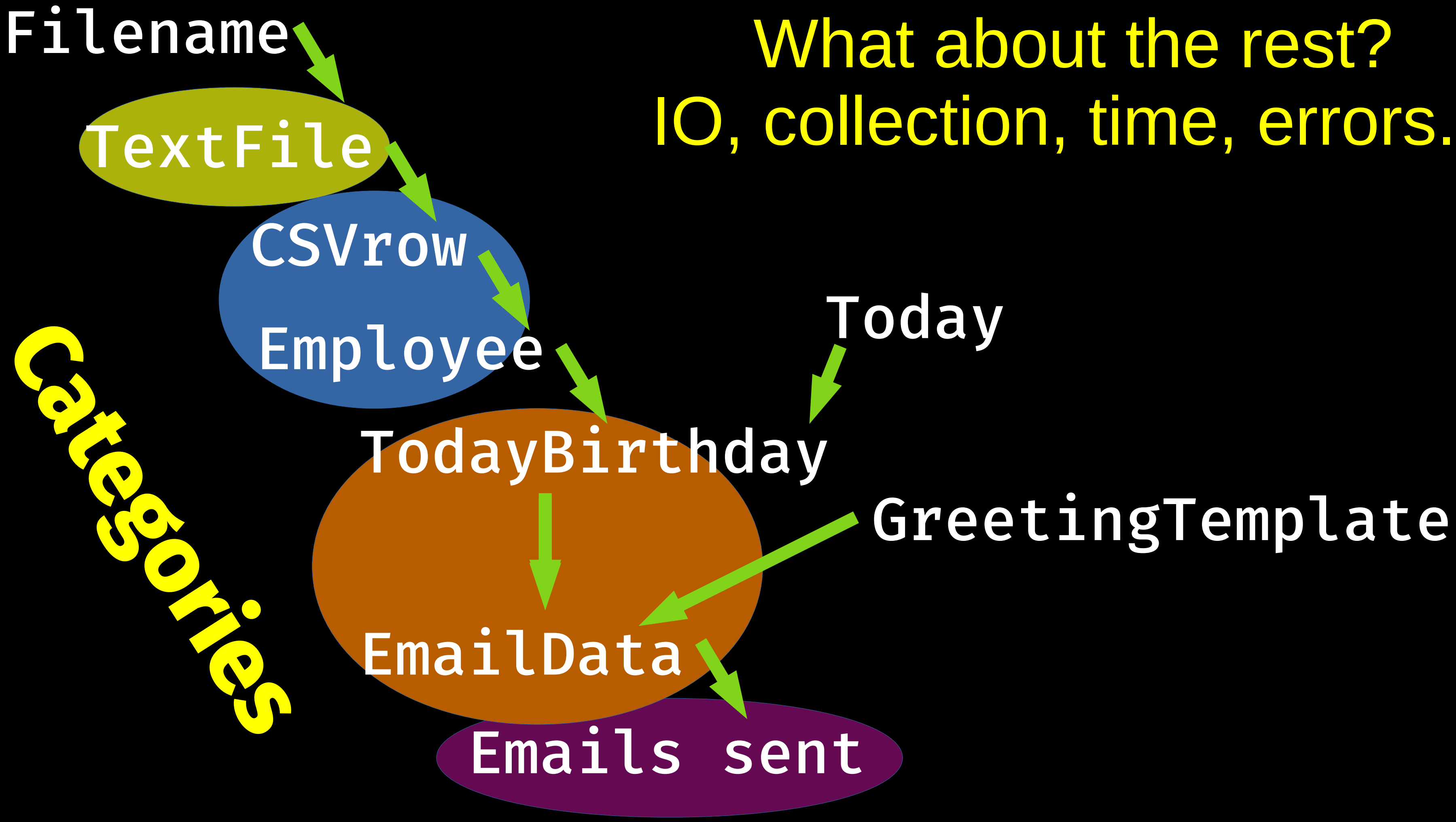
Morphisms → Pure Functions

```kotlin
typealias EmployeeToEmail = (Employee) -> Email

class EmailTemplate(val msgTemplate: String): EmployeeToEmail {
    override fun invoke(e: Employee): Email  =
                  Email(e.email, "Greetings",
                  msgTemplate.replace("%", e.firstName))
}

fun rowToEmployee(csv: CsvRow): Employee =
        csv.raw.split(",").let{
            Employee(
                  firstName = it[1].trim(),
                  lastName = it[0].trim(),
                  email = EmailAddress(it[3].trim()),
                  dateOfBirth = LocalDate.parse(it[2].trim(), LOCAL_DATE))
}
```

What about the rest?
IO, collection, time, errors...

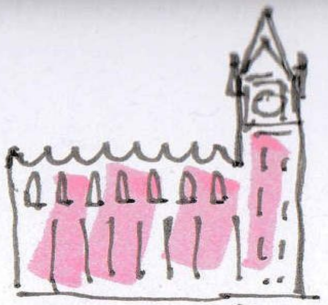


What about the rest?
IO, collection, time, errors...



But, but...
how can we switch category?

Functors



$\mathbb{C}$ MILANO $\quad\quad$ $\mathbb{C}$ LONDON

$$\begin{tikzcd}
\text{DUOMO} \arrow[r, "\text{ML FUNCTOR}"] \arrow[d, "fm"] \arrow[dd, bend left, "hm"] & \text{ST. PAUL} \arrow[d, "fl"] \arrow[dd, bend left, "hl"] \\
\text{CASTELLO} \arrow[r, "\text{MLF}"] \arrow[d, "gm"] & \text{TOWER OF LONDON} \arrow[d, "gl"] \\
\text{PALAZZO REALE} \arrow[r, "\text{MLF}"] & \text{BUCKINGHAM PALACE}
\end{tikzcd}$$

$$fm \circ gm = hm \longrightarrow fl \circ gl = hl$$

Simplest functor in our code:

```
string.length()
```

Simplest functor in our code:

`string.length()`

Unfortunately it doesn’t compose

Generics are **type builders**

`List<T>` is not a type but you can build types

`List<String>`

`List<Double>`

`List<Employee>`

Etc.

Generics are **type builders**

`List<T>` is not a type but you can build types

`List<String>`

`List<Double>`

`List<Employee>`

Etc.

`Employee → List<Employee>`

is a Functor to the Category of List

List is said to have a Functor instance

`Employee → List<Employee>`

is a Functor to the Category of List

List is said to have a Functor instance

map is how you translate functions (morphisms)

```
val employees: List<Employee> = …
val names: List<String> =
    employees.map { emp → emp.name }
```

```
List<T>.map(f: (T) -> R): List<R>
```

```
val employees: List<Employee> = …
val names: List<String> =
    employees.map { emp → emp.name }
```

```
List<T>.map(f: (T) -> R): List<R>
```

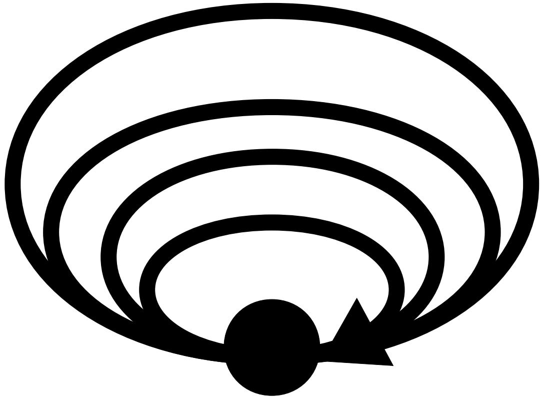
Functors are like Fork lifts, they lift a function from

`Employee → String`

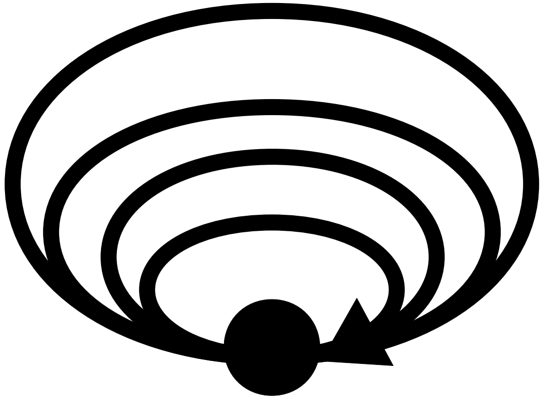
to

`List<Employee> → List<String>`

A Category with only one Object is called a …



A Category with only one Object is called a Monoid

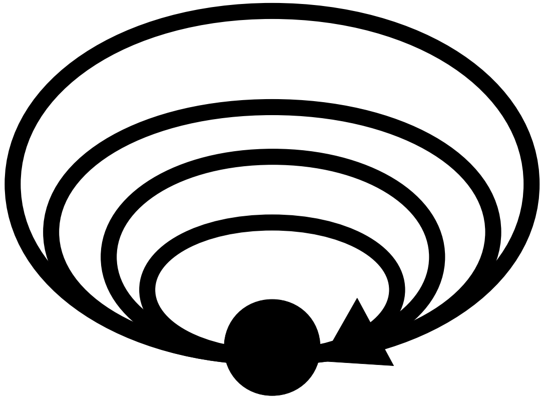


(A, A) → A

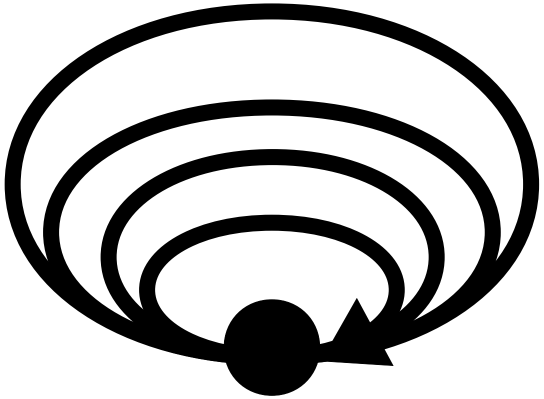
String, String → String   concat

Int, Int → Int   add

A Category of all EndoFunctors Is called a ...



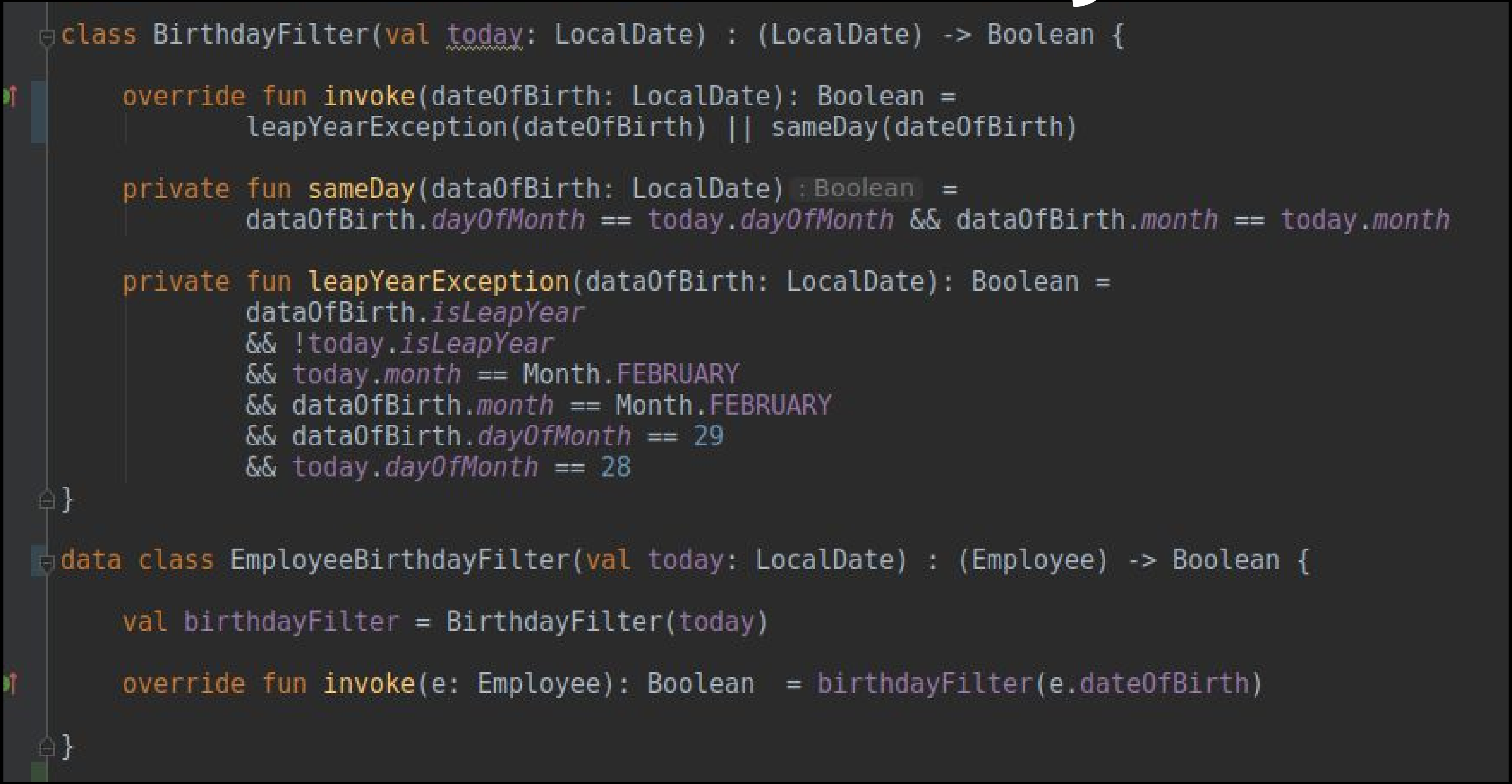
A Category of all EndoFunctors Is called a Monad



(A →F<B>, B → F<C>) → A → F<C>

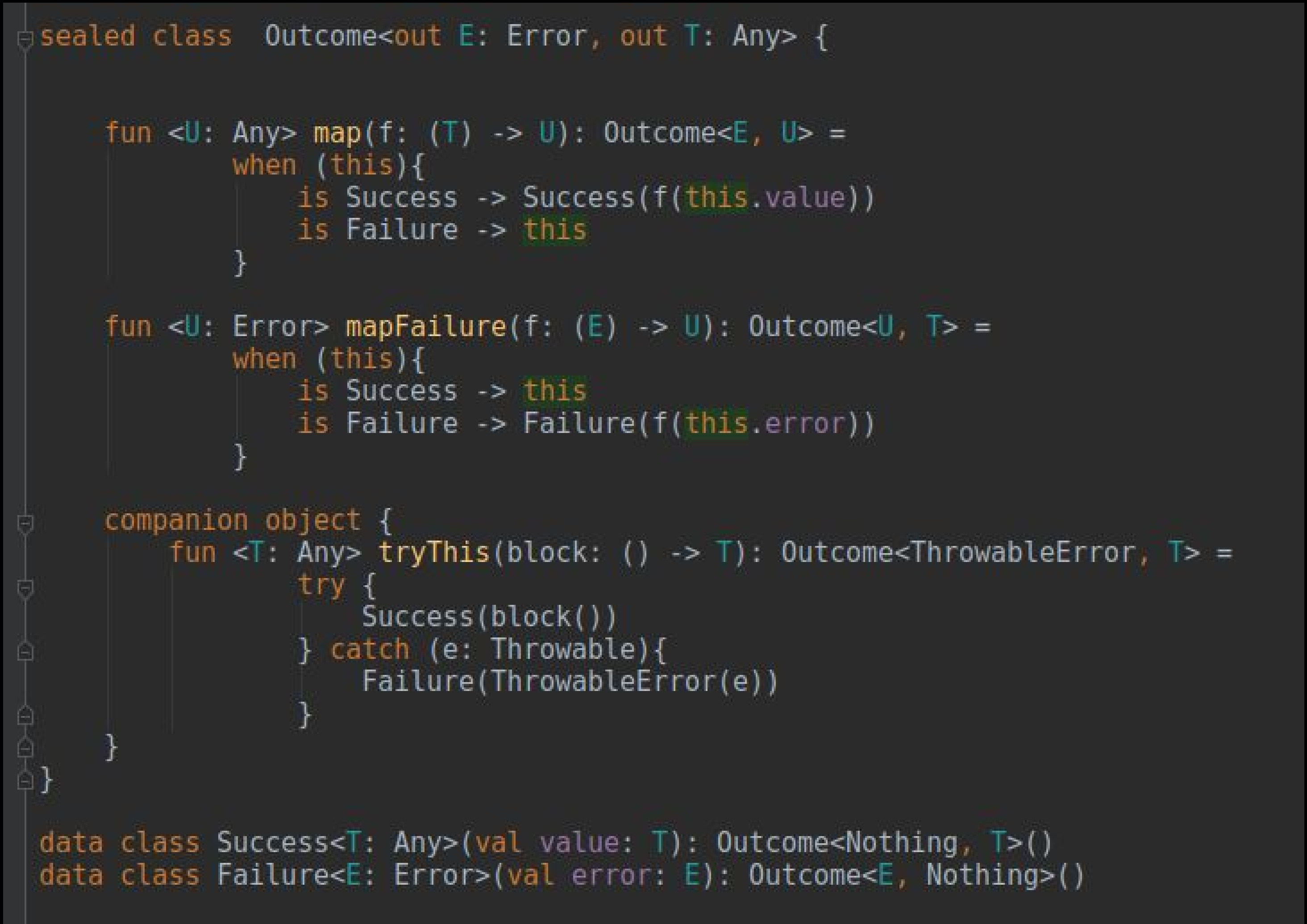
Monads are needed to compose Functors

Functor for Birthday Filter



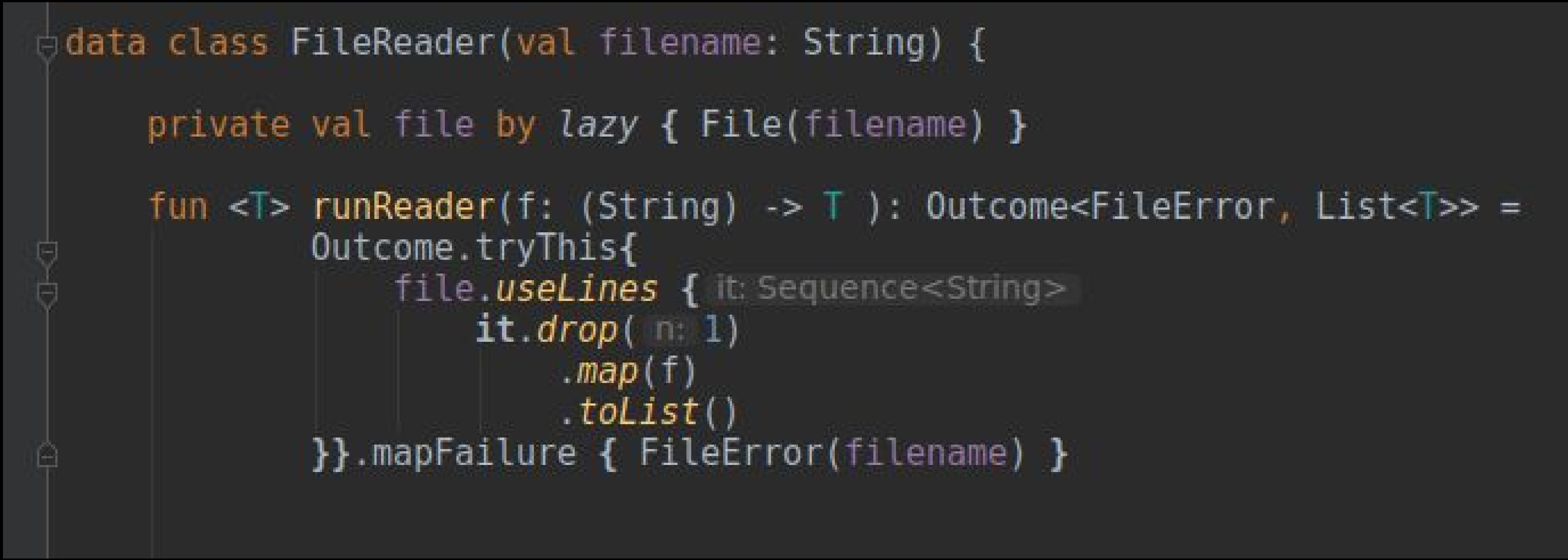
```
class BirthdayFilter(val today: LocalDate) : (LocalDate) -> Boolean {

    override fun invoke(dateOfBirth: LocalDate): Boolean =
            leapYearException(dateOfBirth) || sameDay(dateOfBirth)

    private fun sameDay(dataOfBirth: LocalDate) : Boolean  =
            dataOfBirth.dayOfMonth == today.dayOfMonth && dataOfBirth.month == today.month

    private fun leapYearException(dataOfBirth: LocalDate): Boolean =
            dataOfBirth.isLeapYear
            && !today.isLeapYear
            && today.month == Month.FEBRUARY
            && dataOfBirth.month == Month.FEBRUARY
            && dataOfBirth.dayOfMonth == 29
            && today.dayOfMonth == 28
}

data class EmployeeBirthdayFilter(val today: LocalDate) : (Employee) -> Boolean {

    val birthdayFilter = BirthdayFilter(today)

    override fun invoke(e: Employee): Boolean  = birthdayFilter(e.dateOfBirth)

}
```

Functor for Errors



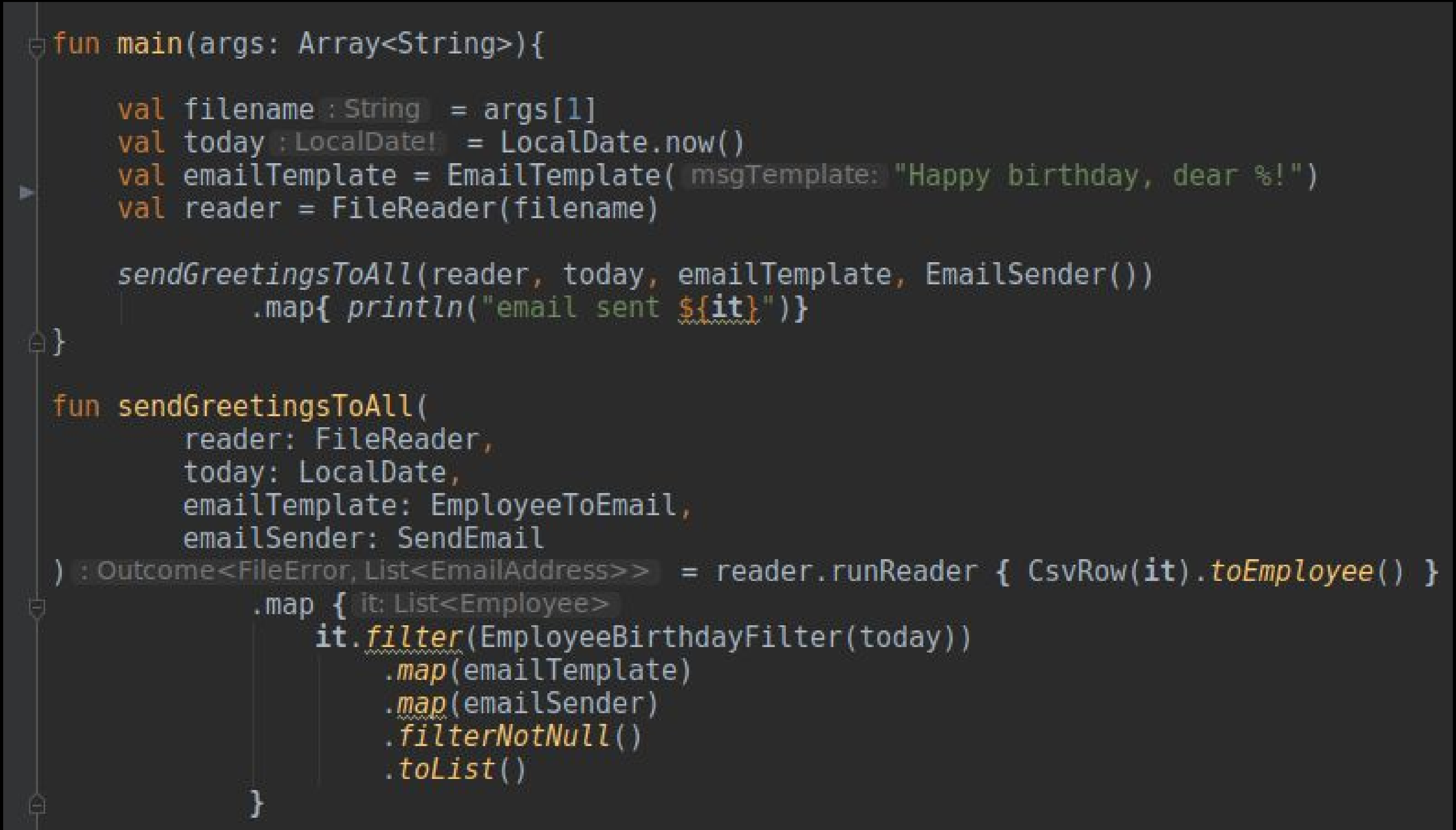
```kotlin
sealed class  Outcome<out E: Error, out T: Any> {


    fun <U: Any> map(f: (T) -> U): Outcome<E, U> =
            when (this){
                is Success -> Success(f(this.value))
                is Failure -> this
            }

    fun <U: Error> mapFailure(f: (E) -> U): Outcome<U, T> =
            when (this){
                is Success -> this
                is Failure -> Failure(f(this.error))
            }

    companion object {
        fun <T: Any> tryThis(block: () -> T): Outcome<ThrowableError, T> =
                try {
                    Success(block())
                } catch (e: Throwable){
                    Failure(ThrowableError(e))
                }
    }
}

data class Success<T: Any>(val value: T): Outcome<Nothing, T>()
data class Failure<E: Error>(val error: E): Outcome<E, Nothing>()
```

Functor to Read from IO



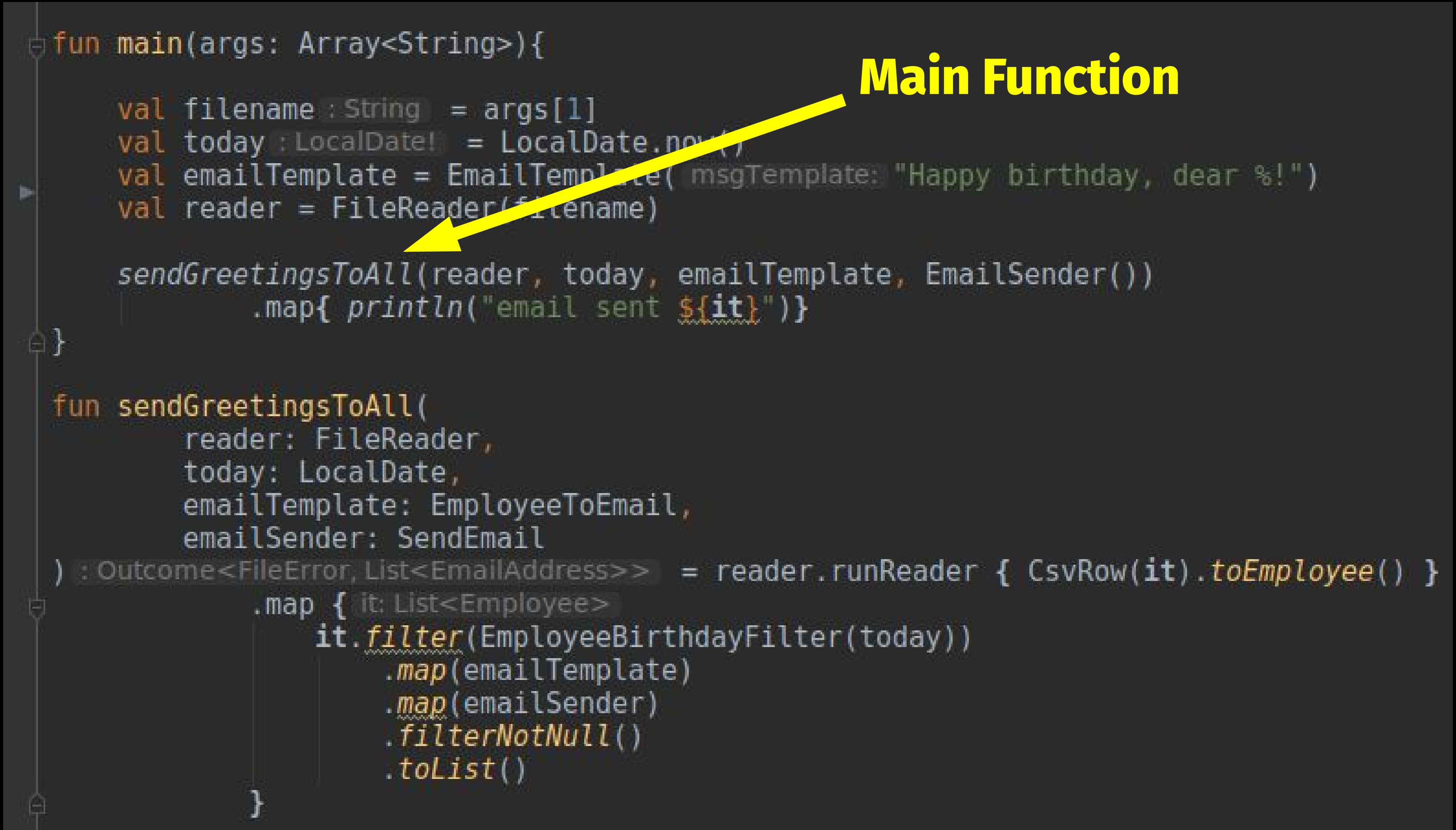
```
data class FileReader(val filename: String) {

    private val file by lazy { File(filename) }

    fun <T> runReader(f: (String) -> T ): Outcome<FileError, List<T>> =
            Outcome.tryThis{
                file.useLines { it: Sequence<String>
                    it.drop( n: 1)
                        .map(f)
                        .toList()
            }}.mapFailure { FileError(filename) }
```

The Result



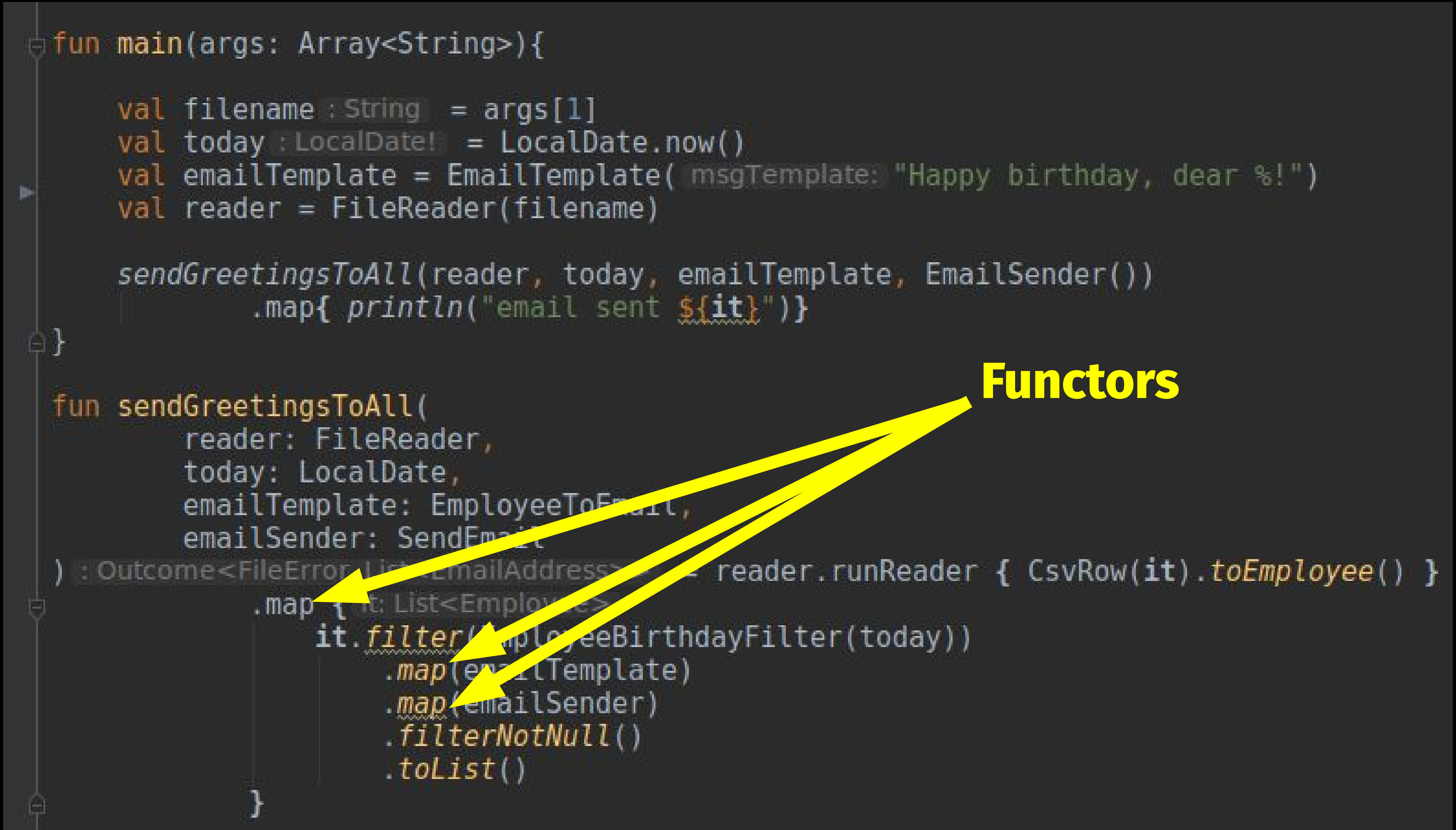
```kotlin
fun main(args: Array<String>){

    val filename : String  = args[1]
    val today : LocalDate!  = LocalDate.now()
    val emailTemplate = EmailTemplate( msgTemplate: "Happy birthday, dear %!")
    val reader = FileReader(filename)

    sendGreetingsToAll(reader, today, emailTemplate, EmailSender())
            .map{ println("email sent ${it}")}
}

fun sendGreetingsToAll(
        reader: FileReader,
        today: LocalDate,
        emailTemplate: EmployeeToEmail,
        emailSender: SendEmail
) : Outcome<FileError, List<EmailAddress>>  = reader.runReader { CsvRow(it).toEmployee() }
            .map { it: List<Employee>
                it.filter(EmployeeBirthdayFilter(today))
                    .map(emailTemplate)
                    .map(emailSender)
                    .filterNotNull()
                    .toList()
            }
```

The Result



```
fun main(args: Array<String>){

    val filename : String  = args[1]
    val today : LocalDate!  = LocalDate.now()
    val emailTemplate = EmailTemplate( msgTemplate: "Happy birthday, dear %!")
    val reader = FileReader(filename)

    sendGreetingsToAll(reader, today, emailTemplate, EmailSender())
            .map{ println("email sent ${it}")}
}

fun sendGreetingsToAll(
        reader: FileReader,
        today: LocalDate,
        emailTemplate: EmployeeToEmail,
        emailSender: SendEmail
) : Outcome<FileError, List<EmailAddress>>  = reader.runReader { CsvRow(it).toEmployee() }
            .map { it: List<Employee>
                it.filter(EmployeeBirthdayFilter(today))
                    .map(emailTemplate)
                    .map(emailSender)
                    .filterNotNull()
                    .toList()
            }
```

The Result



```
fun main(args: Array<String>){

    val filename : String  = args[1]
    val today : LocalDate!  = LocalDate.now()
    val emailTemplate = EmailTemplate( msgTemplate: "Happy birthday, dear %!")
    val reader = FileReader(filename)

    sendGreetingsToAll(reader, today, emailTemplate, EmailSender())
            .map{ println("email sent ${it}")}
}

fun sendGreetingsToAll(
        reader: FileReader,
        today: LocalDate,
        emailTemplate: EmployeeToEmail,
        emailSender: SendEmail
) : Outcome<FileError, List<EmailAddress>> = reader.runReader { CsvRow(it).toEmployee() }
            .map { it: List<Employee>
                it.filter(EmployeeBirthdayFilter(today))
                    .map(emailTemplate)
                    .map(emailSender)
                    .filterNotNull()
                    .toList()
            }
```

CHECK!

Passes the tests
Reveals intention
No duplication
Fewest elements

Takeouts

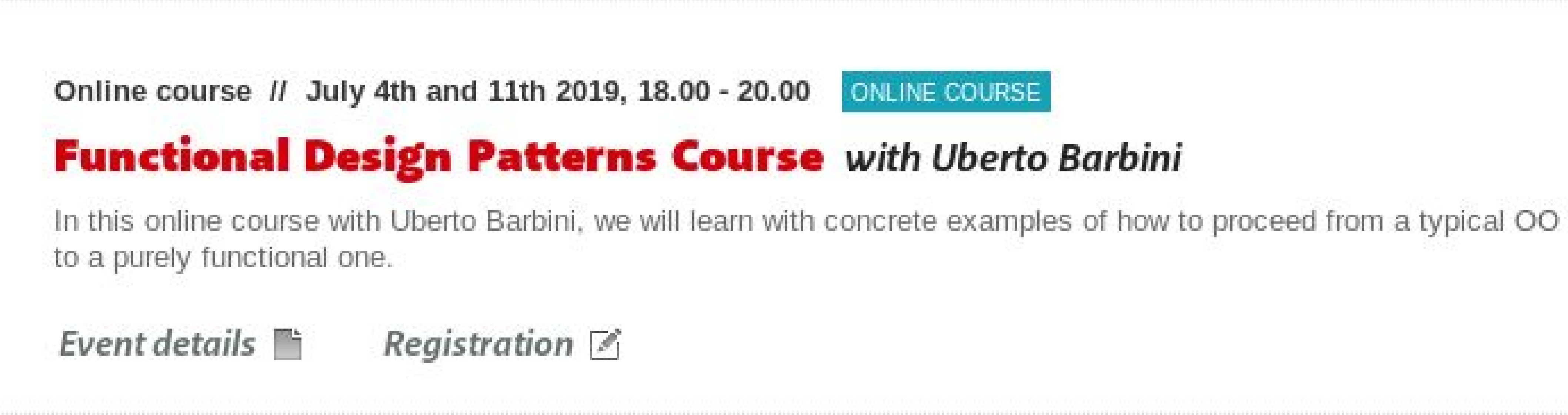
Functional Programming goals are the same than Object-Oriented Programming

You don’t need a special language or library

You need to study and practice a different paradigm

github.com/uberto/birthdaykata

Uberto Barbini


Online course // July 4th and 11th 2019, 18.00 - 20.00 ONLINE COURSE

Functional Design Patterns Course *with Uberto Barbini*

In this online course with Uberto Barbini, we will learn with concrete examples of how to proceed from a typical OO
to a purely functional one.

Event details *Registration*

Blog: **`medium.com/@ramtop`**

Twitter: **`@ramtop`**