

A Tale of Two Worlds: FP and OOP

Case study



Functional vs OO programming

Table of contents

Contents

[Introduction 1](#_Toc517558558)

[Motivation 1](#_Toc517558559)

[Context 2](#_Toc517558560)

[History 3](#_Toc517558561)

[Functional languages 3](#_Toc517558562)

[Object Oriented languages 6](#_Toc517558563)

[Current state of the industry 8](#_Toc517558564)

[Elements 8](#_Toc517558565)

[Functional programming elements 9](#_Toc517558566)

[Immutability 9](#_Toc517558567)

[Higher order functions 9](#_Toc517558568)

[Recursion 12](#_Toc517558569)

[Purity and side effects 13](#_Toc517558570)

[Referential transparency and first-class functions 14](#_Toc517558571)

[Benefits/disadvantages 17](#_Toc517558572)

[Object Oriented programming elements 18](#_Toc517558573)

[Classes/Interfaces 18](#_Toc517558574)

[Inheritance 19](#_Toc517558575)

[Encapsulation 19](#_Toc517558576)

[Polymorphism 20](#_Toc517558577)

[Relations -> passing state and modifying it internally 21](#_Toc517558578)

[Conclusion 22](#_Toc517558579)

[Best Practices 22](#_Toc517558580)

[Functional Best Practices 22](#_Toc517558581)

[Pattern matching for the rescue 22](#_Toc517558582)

[Function composition 25](#_Toc517558583)

[Side effect free 25](#_Toc517558584)

[Separation of pure/impure code 26](#_Toc517558585)

[ADTs and type classes 28](#_Toc517558586)

[Object Oriented Best Practices 31](#_Toc517558587)

[SOLID 32](#_Toc517558588)

[High cohesion/low coupling 32](#_Toc517558589)

[Conclusion 33](#_Toc517558590)

[Design patterns 34](#_Toc517558591)

[Functional design patterns 34](#_Toc517558592)

[Functors 35](#_Toc517558593)

[Monads 37](#_Toc517558594)

[Pipelining 40](#_Toc517558595)

[Object Oriented Design Patterns 42](#_Toc517558596)

[Builder 43](#_Toc517558597)

[Factory 44](#_Toc517558598)

[Memento 45](#_Toc517558599)

[Observer 45](#_Toc517558600)

[Conclusion 46](#_Toc517558601)

[Demo explained 47](#_Toc517558602)

# Introduction

“A Tale of Two Worlds: FP and OOP” is a case study comparing how 2 completely different paradigms – Object Oriented paradigm and Functional paradigm – behave, code, interact with the programmer and, most importantly, how they impact the mindset and architecture of an application.

## Motivation

I personally feel there is a great confusion when it comes to OOP/FP paradigms – most of the more popular languages today implement both paradigms in a way that might be hard to understand which is which. The purpose of this work is to shed some light on what is the clear distinction between the two, how are they similar (read “how they aren’t similar at all”), but ultimately – help the reader clearly know what elements are being used when engineering a software application, when one should be favored to the other, and also, how they could be combined to produce the best of both worlds.

As a young programmer, I’ve been introduced to the mainstream paradigm – which is imperative, and more specifically, procedural.

However, it felt like it lacks discipline and organization – a lot of code is being written, but there wasn’t a bigger picture being built, but more importantly – there weren’t a lot of rules about how the code should behave or look. So it is mostly up to the programmer to do what need what needs to be done without putting much thought into the design.

Later on, I’ve been introduced to OOP – and it seemed the right approach to coding. There are a lot of best practices, and it mostly came down to classes – modelling the program as the real world, a lot of relations between objects, a lot of interactions, and most importantly – it is organized. Classes organized into packages based on relations and importance, there are some good principles that keep the programmer from harming himself, and so on.

However, as time went on, I started getting familiar with the “bad” parts of OOP – and it mostly came down to complexity under the form of lines of code – a lot of them are needed in order for something to be done “properly” – and most of that is boilerplate. The intention, the idea – which is the main focus, what is the goal – is somewhat hidden behind all the exceptions, all the edge cases, all the getters and setters – all of these elements create an unnecessarily complex system. And, most importantly, having all these relations and interactions between objects means that sometimes, the programmer doesn’t really know how the system will behave – how it will react to some input. The rules aren’t that well defined, so there is room for interpretation and debate of how something should be done “properly”.

However, this is fine, as the advantages slightly outweigh the disadvantages – but there was room for improvement.

And this is the moment I actually encountered functional programming. I initially liked the idea of minimalization – writing code that does what you want, without having to carry around all of this extra weight. However, as I dived deeper, I found that there is actually more than meets the eye: there are many rules whose role is to make sure the code does what it was intended to, that the system is stable and predictable. Generally, you don’t want to code towards a solution of a single problem – but more towards the generalization of one, and for that a higher order of abstraction is needed. All the “classes” are minimal, no getter nor setters, no mutability, and there is usually functions working **on** data, not data that has some behavior **attached** to it.

But, with all the rules, comes the complexity – and this time in terms of adaptability and learning curve. Learning how to write proper functional code takes time – a lot of time and care, and it is something that not many people want. Having to abstract over a solution sometimes is mind-bending, and the systems might get complicated – not because of the amount of code, which is usually much shorter than any OOP counterpart, but because of the abstractions that were used in order to get to a solutions. This means that the programmer has to be extremely dedicated about the work and constantly put in effort to get a better grasp of the paradigm.

## Context

There has been great debates in the last couple of years as to where programming should be heading – with the introduction of multithreaded CPUs and clusterized servers, it was rather clear that the reigning paradigm OOP is no longer suitable for the job, at least not in the form that everybody was used to.

Dealing with mutable state is incredibly difficult when multiple threads work on the same data – however, people soon realized there is a long forgotten paradigm whose core feature is immutability, and that is the functional paradigm.

With that in mind, functional features have been introduced into what otherwise would be OOP:

* Java introduced:
  + Higher order functions and Lambdas ( functions can now receive another functions as parameter and return one as well, and also Stream collection )
  + The newly released Spring 5 brought other functional features, mainly function composition for Controllers
* C# introduced:
  + LINQ – SQL-like framework which contains operations like **map**, **filter**, **reduce** , except they’re named slightly different
  + Higher order functions and lambdas
  + Extension methods – the functional way of adding functionality to data without actually altering the initial definition of the data

This moved on to people realizing that immutability is a must – and thus, people started favoring that over mutability.

As time passes by, more and more functional elements are being added to these languages – and I believe it is crucial that the programmer knows how to differentiate between the two.

# History

## Functional languages

Functional programming is a programming paradigm where a computation is treated as an evaluation of a mathematical function. Thus, building software becomes a process of composing pure functions, avoiding shared state, side-effects and mutable data, as opposed to OOP where the state is usually shared ( done so by methods which access mutable data ).

Code written in a functional matter tends to be more precise, expressive, predictable and shorted than the imperative or OO code – however, it’s a common pitfall, as it can become cryptic at times if the developer chooses not to pay attention to more self-explanatory options.

One common example in Scala would be the use of for comprehensions instead of multiple map/flatMap operations on data.

**val** possibleClusters = **for** {  
 from <- clustersWithCentroids  
 to <- clustersWithCentroids filterNot (\_ == from)  
} **yield** (from, to, computeDistance(from, to))

**val** possibleClusters = clustersWithCentroids.flatMap{ from =>  
 **val** otherClusters = clustersWithCentroids.filterNot(\_ != from)  
 otherClusters.map(to => (to, from, computeDistance(from, to)))  
}

Despite being a simple example, the first example speaks for itself – one reading and it should be clear what it does, while in the second example some deciphering and a more thorough read might be required – despite the same amount of rows. This boils down to the programmer’s desire of wanting to improve readability and scalability.

Talking about the history of functional languages, one of the most impressive aspects of the paradigm is that all of them are based on Lambda Calculus.

The history of the appearance of the first functional programming languages is as followed.

LISP

The first functional programming language ever that appeared in the late 1950’s and it was a smashing success as it is used almost 70 years later. It is seen by many as one of the simplest, yet most beautiful languages.

A few of the more notable characteristics of lisp include:

* garbage collection as a method of dealing with unused memory cells
* closures – for static scoping
* conditional expressions and use for writing recursive functions ( first ever language to do that )
* higher order operations on lists

FP (Function Programming)

It was introduced by John Backus in his 1977 lecture, "Can Programming Be Liberated from the von Neumann Style?" (!!!!), however the language wasn’t much successful outside of academia.

ML

In the mid 1970’s, researchers at the University of Edinburgh needed a language to describe proof search strategies while working on a system which would automate theorem proving. So, they came up with ML (meta-language) and later figured out they could use it as a general purpose language.

Two of the most important features of the language include pattern matching and user-defined algebraic datatypes. Both features are strongly related and have played a fundamental role in defining modern programming languages.

Miranda

Designed by David turned and making its first apparition in 1985, the core feature is represented by lazy evaluation, which pretty much defined Haskell later on.

Later on, other functional programming languages emerged like:

* Haskell – 1987 – the de facto functional programming language
* Mathematica
* Scheme
* Erlang
* Elixir – runs on the Erlang Virtual Machine (BEAM)
* F#

Scala

Scala first appeared in 2004, being designed by Martin Odersky as part of a project of École Polytechnique Fédérale de Lausanne.

It is developed on the JVM platform, so there are limitations caused by that.

Regarding the paradigms used, Scala sits in a bit of a weird spot - initially, it appeared as a desire to be a better Java, with a much cleaner syntax and less boilerplate code, all while adding some functional elements. As it evolved, it started introducing more FP elements - has all elements apart from laziness by default, while still being a pure OOP language. It's one of a kind language, and it does still have its quirks. As the developers' desire to go more functional increased, a number of libraries emerged to close the gap between the functional and OOP paradigms, creating an environment for Scala to be indeed fully Functional.

## Object Oriented languages

Object Oriented Programming is today’s industry standard and is a programming paradigm whose inspiration was the real world – modelling data by inspiring from nature. Objects who have attributes and can do actions (aka they have methods), classes interacting between one another and modifying state based on some outside stimulus – they are what define OOP.

OOP code tends to be expressive if compared to the real world – there are relations between data and different ways they interact with one another, as opposed to FP where functions working on data is all there is.

The core feature that differentiate the two paradigms, but also define most of their respective features, is represented by mutability – OOP languages are designed with mutability by default, while FP languages tend to prefer immutability. This is what branches the 2 paradigms into their respective traits and what is the defining chapter making the difference.

As far as the history of the OOP languages go, terminology regarding OOP has first appeared in the late 1950s.

* Simula

Simula is widely regarded as the first Object Oriented programming language. It appeared in the 1960s and introduced important concepts that are being used even today (ex: class, object, dynamic binding, and inheritance).

Simula’s use was widely closely related to simulations (as the name suggests) of real life events and systems.

It is the basis of all OO programming languages.

* Smalltalk

First appearing in the 1980s, Smalltalk was heavily influenced by Simula and also heavily influenced other OO programming languages like Java.

Communication in Smalltalk is done via messages sent between classes – it is actually an Actor Model that can be found also in functional languages like Elixir/Erlang or Scala’s Akka framework.

* C++

After having worked with Simula for his PhD thesis, Bjarne Stroustrup went on to introduce object oriented elements in C and come up with the C++ programming language. It came from his understanding that languages needed to evolve from hardware specific, low level languages, to languages that offer high level abstractions, closer to the hardware and efficient. That was the idea of C++: to be fast as C, but also include classes.

* Java

Java is currently one of the most popular languages. Sun Microsystems, the company behind it, first released the language back in 1996, promising WORA – “write once, run anywhere” that Java is so well known for. It quickly became popular as major web browsers implemented the ability to run Java applets within web pages. Its syntax was heavily inspired by C.

The reason Java can be run on any machine is represented by the Java Virtual Machine, or JVM for short. Java code is first compiled into bytecode, which can run on any JVM available.

The current Java version is Java 10, having been released in the spring of 2018. Ever since Java 8, more and more functional elements have been introduce in order to make Java a modern, powerful, concise and overall pleasant language.

* C#

The popular language first appeared in 1988 under the ownership of Microsoft. The name is actually inspired from the musical world, where the # character means a note should be a semitone or pitch higher – and that what C# wanted to be for C.

It is often compared to Java because of how similar they are, at least as the syntax goes. But, compared to Java, C# came up with faster increments to the language, so it’s slightly ahead when it comes to how modern the language itself is. However, Java, being platform-independent, has become much more popular in the open world community, so it has the advantage of countless libraries on its side.

## Current state of the industry

The OOP paradigm has become increasing popular in the 1990’s with the rise of C++/Java/C# and has ruled a big part of the industry for almost 2 decades.

However, with the development of multi-core processors, there has been a shift to working more with threads and parallel processes - as OOP is largely based on mutating the state of the objects, OOP's domination declined a bit as it is increasingly difficult to keep track of changes while multiple threads work on the same data. And as this became a critical part of software development, a need for languages who work on immutable data emerged – and FP is the perfect fit for this, as immutability sits at its core.

Apart from this, another cornerstone is represented by the appearance of MapReduce and Hadoop used in BigData. Javascript is also another popular programming language which is mostly based on the functional paradigm.

As the need for immutability and threads increased, most OOP languages adopted some functional elements - .Net's LINQ is a wonderful example of this, while Java’s Spring 5.0 introduced a whole bunch of functional elements.

In the current state, most programming languages use functional elements to add “flavor”, make programmers’ lives easier by taking advantage of what FP brings: testability, composability and predictive behavior. However, with the rise of languages like Scala, Clojure, libraries like Akka, Spark, React Native, LINQ and so on, the future looks bright for the functional programming languages and overall the paradigm. It is close to impossible for OOP to be replaced, at least in the next couple of decades, but it is important for the programming languages to offer the best of both worlds in order for the code to be top quality.

# Elements

It is crucial for a developer to know what the defining traits of a paradigm are – they are the building blocks of the code, and it is also crucial for a programmer to build idiomatic code given a respective language – this means making sure the code does not go against what the programming language was initially built for.

For example, it would be both a mess and a nightmare to write code based on side effects in Haskell – there are special guards designed into the compiler itself that make it increasingly hard to do so. By knowing this, the programmer can get a hint of how the code should look like given a specific programming language inspired by a specific paradigm.

By being familiar with the defining blocks of a paradigm, it makes the transition easier for the programmer to a technology, and also helps the coder have a broader understanding of programming overall and how specific pieces fit the “puzzle”.

## Functional programming elements

For anybody familiar with FP, it might become apparent quite fast that the paradigm is all about functions – as the name might also suggest. When talking about defining traits of the style, it is mostly about how functions should behave and how data should be represented.

### Immutability

An immutable object represents an object whose state cannot be modified by any means - once created, it remains the same throughout its life-span, without any possibility of changing its internal state.

When it comes to representative traits of the FP paradigm, immutability is what sits at the very core of all of them. Without immutability, the paradigm wouldn't exist. As opposed to the OOP paradigm, most relations are described by applying functions over data - thus, most of those functions usually have some laws associated to them to ensure correctness.

One of the most common laws associated with these functions is indeed immutability - the insurance that the object will not be tainted after the function has been applied.

For example, one of the laws associated with a functor is represented by composition - mapping 2 functions f and g is the same as mapping f and then mapping g, which means that the following property MUST hold:

fa.map(g(f(\_))) == fa.map(f).map(g)

If fa (the object mapping over) is not immutable, then the property simply wouldn't hold for at least some cases - thus, it makes the modelling of the data unpredictable, non-deterministic.

Also, one might argue that having immutable data eases the creation of recursive functions, as it's easier to not think about what happens to your data as the recursion goes deeper and deeper, worrying only about what is the goal.

It has also become a trend in the industry to opt over immutable entities and data over mutable ones even in OOP languages - numerous articles have emerged favoring the principle, and it has become increasingly popular in Java/C#, having Builders to actually create your immutable data.

### Higher order functions

As the name probably suggests, Functional Programming is a lot about functions – they are all that paradigm is all about – all ideas and principles are centered on them.

When talking about functions, FP deals with the more mathematical definition of one – the domain, codomain, generic boundaries.

Any function which receives another function as a parameter or returns a function itself is called a higher order function. They are a defining factor when it comes to abstracting away all the logic and focusing more on what one is trying to achieve, instead of how will that happen.

Higher order functions is what defines the functional programming experience.

Thus, they enable:

* Composability – focusing on what is the goal means the programmer can easily replace functions which have the same signature
* Reusability – basically, higher order functions can be seen as templating – the programmer is only focused on what should be done, and as a final step, the pieces are just put together
* Easier testing – when all the major pieces of a flow are abstract and based on generic definitions, it enables testing to be done much easier by providing functions which return a desired behavior. Thus, mocking is now a trivial task.

Some of the more popular higher order functions include:

* Map – the function received as a parameter - function: A => B - is used to transform the data structure by applying the function over that
* Filter – the function received as a parameter – function: A => Boolean - is used to keep the parts of the data structure whose properties are compliant to the function
* Reduce – the function received as a parameter – function: (A, B) => B – is used to apply the function over all of the elements from left to right and actually reduce the result to a single element. For example, the sum of all the elements in a list is a popular example which could be done using reduce().
* Fold – similar to reduce, except one can specify which end will be used as a starting point. Usually, there are 2 implementations of fold – foldLeft, foldRight – whose names are quite self-explanatory.

In the following section, the importance of higher order functions will be briefly explained.

A simple List implementation would be the following:

//abstract definition  
**trait** List[A] {  
 **def** map[B](f: A => B): List[B]  
}  
  
//a list is either represented by an element and another List tail  
**case class** Head[A](value: A, tail: List[A]) **extends** List[A] {  
 **override def** map[B](f: A => B): List[B] = *Head*(f(value), tail.map(f))  
}  
  
//or it could be an empty list  
**case class** Empty[A]() **extends** List[A] {  
 **override def** map[B](f: A => B): List[B] = *Empty*[B]()  
}

A map operation is defined and implemented in order to illustrate the usefulness of higher order functions.

Now, let’s say there is a list defined and that the programmer wants to add 10 to every single element.

Using higher order functions, this is easily done:

**val** *example* = *Head*(1, *Head*(2, *Head*(3, *Empty*())))  
  
*println*(*example*)  
*println*(*example*.map(\_ + 10))

Without using higher order functions, this would’ve been increasingly difficult. The pure OOP way of doing this would be to define a method inside the definitions of the List – maybe named addToElements(amount: Int), or using Iterators and then using a “**for**” or “**while**” to transform it - but it is easy to see how this would’ve gone out of hand. Higher order functions enable the use of adhoc function application, which is both easier to understand for a future reader, and also reduces a lot of the boilerplate code.

By using the OOP approach, the following would happen:

1. A lot of boilerplate code expressed as methods for every single use of the List or “**for**” or “**while**” structures
2. When designing a library, it would be a major pain to try and extend the uses of the basic implementation. In OOP and imperative languages, this is done by offering Iterators and modifying a data structure using maybe a “**for”** or “**while**” structure.

Overall, higher order functions help a programmer write easy-to-read, well-structured, easy-to-test and boilerplate-free code, all of them being marks of clean coding.

### Recursion

A recursion function is any function which calls itself to yield a final result – usually, this is done over the conventional “**for**” found in imperative/OOP languages, which usually require a counter or have some mutable state – doing so would break the immutability that is desired in FP.

Apart from that, one of the main goals of functional programming is to be as close to mathematics as possible – this is usually expressed through laws that need to be met and other elements.

Recursion, as it is, if properly written, can become quite similar to a mathematical representation of a function.

A simple example is the Fibonacci sequence:

**def** fibonacci(nthNumber: Int): Int = {  
 **def** go(k: Int): Int = {  
 k **match** {  
 **case** 0 => 1  
 **case** 1 => 1  
 **case** \_ => go(k - 1) + go(k - 2)  
 }  
 }  
  
 go(nthNumber)  
}

Simple and easy to understand, close to the mathematical representation of the function.

However, an imperative/OOP approach might look something similar to this:

**def** fibonacci(nthNumber: Int): Int = {  
 **if**(nthNumber < 2)  
 1  
 **else** {  
 **var** nMinus1 = 1  
 **var** nMinus2 = 1  
  
 **var** current = nMinus1 + nMinus2  
  
 **for**( \_ <- 1 to nthNumber - 2) {  
 nMinus1 = nMinus2  
 nMinus2 = current  
 current = nMinus1 + nMinus2  
 }  
 current  
 }  
}

As easily observed, the intent of the function, the real goal of its existence is hidden behind a lot of boilerplate code – it might not seem so complicated here, but again, Fibonacci is a simple example.

Generally, when it comes to larger tasks, it is much easier to split it into multiple, smaller recursive functions and compose them to obtain the result.

However, there is a problem when it comes to recursion – as functions call other functions, they build up the stack – with enough function calls, and the famous stack overflow will creep up in one’s algorithm. Fortunately, tail recursion is here for the rescue – what it does is eliminate the intermediate function calls and only keep track of the initial call and the last one.

### Purity and side effects

In functional programming, one of the most important aspects the programmer has to keep in mind is making sure functions stay as pure as possible.

In order for a function to be categorized as pure, it has to produce no side effects.

A function is considered to have side effects if it modifies some data or variable, has some IO interaction (printing, connecting to a database, logging, etc.) or throws an exception.

In functional programming, it is extremely important for functions and also flow/behavior to be predictable – any side effects might actually affect the outcome of a function based on some external factors.

For example, if there is a function which also has some calls to a database for retrieval, the programmer can’t tell the outcome of that function given an input – is the connection to the database established, are the credentials correct, will there be a timeout, or everything will go fine and the function will output a result?

All these factors are considered evil because a function can become quite unpredictable depending on some factors that are outside of the programmer’s reach. When a function has no side effects, no matter how hard one tries to break it, it will always return the same output for the same input.

Avoidance of side effects has benefits like:

1. Less error-prone code – again, everything is input-output. Once this is true, the code is very unlikely to break.
2. Easier testing – since avoidance of side-effects is based around not having any contact with the “outside” world, this means a function will only work with its input. This, again, means testing is now a trivial task.
3. Atomic functions – making sure there are no side effects will more often than not force the programmer to break what was initially a larger function into smaller functions, each one dealing with a certain task, which together make up the same functionality the initial function had. This, of course, has a big impact on the code – easier testing, and also, easier to modify.

### Referential transparency and first-class functions

A big reason why purity is crucial to functional programming is represented by the fact that functions are treated as a first class citizen in most functional languages.

When one talks about a first class citizen in a programming language, it is usually refers to the entity which supports all operations generally available to other entities, like: access, passing around to a function, working with that as a variable, modification and being returned as a result from a function.

In terms of functional programming, functions are usually the first class citizens. This means the programmer can pass a function to another function, store it as a variable, return a function from another function, and so on.

As immutability and side-effect free are important aspects of functional programming, another notion has been introduced in order to further help predict the behavior of a program.

And that is represented by referential transparency – it represents a value’s ability to be replaced by its actual expression and the flow of the program would remain unchanged – nothing would break and the program would continue to execute, every single time, no matter the situation.

Generally, side-effects break this desirable trait, as they produce unpredictable results – the simplest example would be represented by exceptions, as they completely change the flow of a program once thrown.

Let’s have a simple example: a function which performs a division.

**def** division(a: Double, b: Double): Double = {  
 **if**( b == 0.0)  
 **throw new** IllegalArgumentException("Division by zero")  
 **else** {  
 a / b  
 }  
}

The function produces a side effect – the elephant in the room being, of course, the thrown exception.

Running this example with b as 0.0 will break the program.



Of course, this means that if one assigns the result to a value, the program will break the moment the function is ran.

Another example would be a call to a database – there is no telling whether the call will be successful or it will fail (connection timeout, no internet connection, bad credentials, etc).

That is not a desirable feature – one would want his program to continue running, and handle the division by zero as something that is to be expected and to be further treated as part of the business logic.

One of the most known methods of doing so is encapsulating the result in an Option Monad (more about this in further sections).

Simply put, the previously mentioned Monad can be explained as such: there is this box which represents the result, however, it is unknown what the result is until the box is actually opened – and there are 2 possible outcomes:

1. there might actually be a result – expressed as Some(result)
2. Or there might not be a result, meaning somewhere something went wrong – expressed as None

**def** division(a: Double, b: Double): Option[Double] = {  
 **if**(b == 0.0)  
 None  
 **else** *Some*(b)  
}

This is the function rewritten in a functional way.



And, the expected result – no result, since division by zero is impossible. Now, assigning the result to a value and working around with that will not affect our code in any way, thus enabling referential transparency.

The importance of referential transparency might be subtle at first, but it’s of great importance.

One of the caveats of most OOP/imperative languages is dealing with exceptions and generally, any side effects – this really creates a more than needed verbose code, and it might make the initial intent harder to reach to.

In our example, if one might want to use the division() function in his code, the programmer might want to first consider those exceptions and add any throws/catch. This only adds complexity to the code without any need of having so. While, making sure that functions are side effect free, referential transparent, the programmer can further use that function call and manipulate any way he wants to, disregarding anything that might have went wrong until it is indeed needed.

And also, probably the more important aspect, it makes the code easier to reason with and prove correctness, but it also enables lazy evaluation – a core feature of the de facto functional programming language, Haskell.

### Benefits/disadvantages

As previously presented, there are some principles that are the core of functional programming:

* Immutability
* Functions/higher-order functions
* Recursion
* Purity and referential transparency

They are the basis of all the paradigm, and also the true building blocks of other more advanced concepts – it would be close to impossible to implement some of the more advanced concepts without having fulfilled these basic requirements.

When it comes to advantages, there are plenty:

* Predictability – the code behaves as one expects, as a function returns the same output for the same input
* Stability – data structures that can’t be changed, coupled with referential transparency, insure that the code won’t break very often, if ever
* Readability – recursion and higher order functions enable the programmer to breeze through the code and get a good understanding without going too deep
* Easier to modify – as most code is side-effect free and predictable, while the functions are atomic, it is easy to modify a small piece of code while still being sure of the fact that it won’t affect the overall service and it will also be predictable
* Easier testing – side-effect free code means less mocking, while referential transparency means input-output is easy to model and test for edge cases
* Less code – not having to deal with mutable state, side-effects, while benefiting from features like recursion means the core code of the app will be shorter and more concise, while being more stable/predictable than the OOP/imperative counterpart.

However, while there are plenty of advantages, there is one big disadvantage, and for some it might be the only one needed to stay away from functional programming.

The learning curve for this paradigm is rather steep compared to other paradigms, as there are a lot of rules and laws. It is overwhelming at first, and it only gets difficult while going deeper into the FP world – however, once one has become familiar with most concepts, the advantages far overweigh the disadvantages.

When it comes to these principles alone, decoupled from the whole paradigm, they are actually used a lot in imperative/OOP languages, but they are not the core of them.

It’s something that most programmers have come to realize that it’s crucial for some features to be a part of their code – like immutability – while other just make their lives easier – like higher order functions (most languages these days have some form of higher order functions), but also recursion, as some students/programmers find it easier to think about some algorithms that way.

## Object Oriented programming elements

OOP first emerged as a solution to the low level, inexpressive languages – the industry needed something that would be easier to work with and overall, more friendly. The paradigm promised higher level abstractions while still being as fast or almost as fast as their lower level counterparts.

The higher level abstractions are based on modelling the program as close to the real world as possible, using classes with attributes and methods – humans representing everything as a class, and furthermore, hierarchical relationships between classes.

When it comes to the elements representing the OO paradigm, they are closely related to classes and how they should (not) be represented.

### Classes/Interfaces

Classes are the single most important element to OOP – they are to OOP what functions are to FP.

Classes themselves have multiple elements:

* Constructors
* Attributes
* Methods

Classes are what a hold an OOP language together – without them, there would be nothing.

Interfaces are a higher abstraction of a class – they can be seen as contracts. A contract has some clauses that need to be met – it is the same with interfaces. They present a few methods that will need to be implemented if a class decides to implement the interface.

They are a powerful abstraction that decouples the code and generally enables more powerful abstraction to be created.

### Inheritance

One of the greatest features of OOP (or so was advertised) is represented by Inheritance.

The advantage of inheritance is basically the fact that the programmer can take use of code reuse.

However, as time went by, programmers realized that inheritance on itself is not that great – more than that, it’s dangerous to use. It creates tightly coupled classes – imagine 3 different classes inherit from the same base class, and the business logic changes for the base class. What will happen with the 3 inherited classes? Simply put, all this coupling creates an unstable foundation for future code.

### Encapsulation

Encapsulation represents a class’ ability to hide its attributes by using methods – this way, both of them work together and constitute the class.

Restriction to attributes is done through access modifiers, which usually are the following:

* Default – depends on the languages, for example in Java this is usually called package-private, meaning it can be accessed from any class in the current package
* Public – attributes are accessible to the whole world
* Protected – attributes are accessible through any classes from the same packages or any subclasses
* Private – this should be the default access modifier as it is the most restrictive one – access to an attribute is only done through the current class.

The reason for this is simple:

* Validating data
* There might be some logic attached to the class that changes the attribute accordingly

### Polymorphism

Polymorphism represents an object’s ability to present the same interface, but different behavior based on instantiation.

public abstract class Vehicle {  
 public abstract int wheels();  
}  
  
class Bicyle extends Vehicle {  
 @Override  
 public int wheels() {  
 return 2;  
 }  
}  
  
class Car extends Vehicle {  
 @Override  
 public int wheels() {  
 return 4;  
 }  
}

The previous example is a classic one: a vehicle’s number of wheels is dependent on the vehicle itself.

Polymorphism is important in OOP because it helps model behavior in hierarchies of classes/interfaces. This way, it better assembles real world examples, making it easier for the programmer to think in terms of how classes should interact/behave.

The power of inheritance and polymorphism is illustrated in dynamic binding.

This refers to the process of the compiler deciding what method will be called at runtime. Through inheritance and polymorphism, there might be multiple implementations for the same method.

So, in the example presented above, when calling the **wheels()** on a instantiated Vehicle, the compiler will look up the specific implementation in order for everything to work as expected.

### Relations -> passing state and modifying it internally

One of the more defining traits of OOP is represented by the interaction between objects. Through that interaction, they:

* Pass state/information
* Return state/information
* Modify internal state if needed

OOP appeared as a way to ease the programmer’s job by thinking about coding as they would think about nature – modelling the business through relations, changing state based on some trigger, passing state through references – which is how objects are passed around in most OOP languages.

Advantages to this include:

* Easier to get one’s head around
* Easy to implement
* Not having to think about creating new instances
* Mutability is what people are usually used to
* Easier to make analogies with real life

Disadvantages include:

* Behavior might be hard to predict at times
* Subtle bugs may appear – passing references around is what is dangerous
* Recursion is close to impossible – and especially on algorithms that contain calls in a “graph-ic” recursive matter => hard to keep track of states
* Working with threads is extremely dangerous and can result in deadlocks or unexpected behavior
* Any method that is not thread-safe or doesn’t control side-effects well is dangerous: for example, Java’s forEach() function from java.util.stream is extremely unpredictable when dealing with side-effects.

This is one of the most fundamental ideas programmers have to get their head around in OOP – as opposed to FP, where passing state and modifying anything is never actually done. This is one of the major differences between the paradigms – OOP is all about change, triggers and exchanging information, while FP is all about functions that give an output for an input, without changing anything.

Most of the elements from both paradigms are centered on these 2 fundamentally different ideas.

## Conclusion

As a final conclusion to this chapter, it is important for the programmer to know what the building blocks of a language are. By knowing the concepts of a paradigm and then binding that to a language belonging to that specific paradigm, it is easy relatively easy to get accustomed to a programming language.

When talking about OOP and FP, it is important to observe a crucial detail:

* FP – it deals with **functions** only, and everything is centered on that. More specifically, mathematical representations of the functions, so side-effect free.
* OOP – it deals with **classes**, relations between objects and the interaction between them.

Because of the more mathematical side of FP, immutability is a must in order to be able to express certain relations and laws, while mutability is a strong point of OOP just because there is a lot of interaction between classes.

Once this is understood, everything else falls into place.

# Best Practices

It is in everybody’s favor that there are some ground rules which specify how the programmer should write the code in order for it to be compliant, testable and more importantly, extensible.

In the following sections, best practices belonging to both paradigms will be analyzed and compared, in order to get a better feel of how each one of them impacts the mentality of the programmer.

## Functional Best Practices

When talking about functional best practices, the topic of discussion is focused on functions – on functions behave, how they should be written, and more importantly, how the programmer can and should compose functions.

### Pattern matching for the rescue

Pattern matching is the process where an entity is tested against various conditions, and if it falls through one, there will be some code executed.

It is one of the more useful and commonly met best practice, as it simplifies the code and makes it more readable.

In most functional programming languages, almost anything can be use in a pattern matching clause.

There are multiple uses to pattern matching, as a few will be demonstrated immediately.

**def** skipOne[G](l: List[G]): List[G] = {  
 l **match** {  
 **case** *Nil* => *List*()  
 **case** h*::Nil* => *List*(h)  
 **case** h*::*\_*::*t => *List*(h) ::: *skipOne*(t)  
 }  
}



Here, pattern matching is used against a list in order to find out information about the current state of that – is it empty, does it contain only one element, or 2 elements and a tail? By doing so, it is possible to filter every other element out of the list – the second picture is the result of a call on a list from 1 to 10.

**def** factorial(n: Int): Option[Int] = {  
 n **match** {  
 **case** badInput **if** n < 0 => None  
 **case** one **if** n == 1 || n == 0 => *Some*(1)  
 **case** x => *factorial*(n - 1).map(result => result \* x)  
 }  
}

Pattern matching can also be used in recursive functions, but also testing whether the input is not expected or a result cannot be returned given that input. Here, it is known that factorial is defined for natural numbers, so it is tested whether the n is less than 0. Otherwise, the definition remains unchanged.

Furthermore, there is another aspect that makes pattern matching attractive to functional programmers – the ability to easily express relations close to their mathematical representation.

**trait** X  
  
**case class** A(x: Int) **extends** X  
**case class** B(x: Int, y: Int) **extends** X  
**case class** C(x: Int, y: Int, z: Int) **extends** X  
**case object** D **extends** X  
  
**def** process(input: X): Unit = {  
 input **match** {  
 **case** *A*(x) => *println*(s"""A: **$**x""")  
 **case** *B*(x, y) => *println*(s"""B: **$**x, **$**y""")  
 **case** *C*(x, y, z) => *println*(s"""C: **$**x, **$**y, **$**z""")  
 **case** D => *println*("D")  
 }  
}

One final use for pattern matching is the processing of hierarchies of classes, and having a different flow depending on what’s the input.

**def** processGeneric[G](input: G): Unit = {  
 input **match** {  
 **case** \_: String => *println*("It's a string!")  
 **case** \_: Int => *println*("It's an int!")  
 **case** x: X => *process*(x)  
 }  
}

Or simply processing generic parameters at runtime. The function receives any parameter type and it tries to process it. The last case is representative of the case when there is no match, because otherwise, it would throw an error: 

Generally, pattern matching is used for clearing out the intent of the code, make it more readable, easier to extend and modify.

### Function composition

One of the more interesting features of functional programming is represented by function composition – the idea of pipelining results from one function to the other, creating a whole new function with the aggregated transformations of multiple functions.

This greatly enables modularity and atomicity, as it enables the programmer to easily chain function without have to write a lot of extra boilerplate code.

In order to do compose two functions, the return type of the first function must match the argument type of the second function – the newly created function will receive as a parameter what the first function receives and returns what the second function returns.

In Scala, this can be done in 2 ways: compose and andThen. Both will be briefly explained.

**def** f(i: String): String = "f(" + i + ")"  
  
**def** g(i: String): String = "g(" + i + ")"  
  
**val** *composition* = *f* \_ compose *g***val** *andThen* = *f* \_ andThen *g*

*println*(*composition*("test"))  
*println*(*andThen*("test"))



As easily seen, they both do the same job – compose functions – but differently. It is important to know the difference between them, and apply them accordingly.

### Side effect free

When it comes to the way the programmer writes the code, it is important that a major part of it is indeed side-effect free.

As previously talked, side-effects represent any interaction with the “outside” world, modifying the state of a variable or generally any IO operations (reading a file, connecting to a database, retrieving data from a database and so on).

In a paradigm where the correctness and reasoning are what define it, it is trivial to come up with reasons why side effects are considered evil, and therefore, are to be avoided at all costs: they produce unstable, unpredictable behavior, and thus, all the work will be going down the drain.

In order for the code to be “pure” (aka have no side effects), there are some general tips which might lead to such traits:

* Avoid mutable state at all costs
* If a state has to be changed, produce a new value which is a copy of the previous one with the needed updates – with this an important issue appears, and that is represented by the amount of memory used. However, this is easily fixed by modern garbage collectors.
* Use recursion to express a flow – ideally, all recursion functions should be tail optimized. It is a technique where all intermediate calls of the function are not held onto the stack anymore, except the first and last one. This makes any update method on a value trivially simple for the garbage collector to remove from memory, making sure there is as little memory as possible used.
* Using higher order functions – using map/filter/reduce/etc. instead of the classic “for” and “while” insure the fact that new, immutable and side-effect free values are produced – this, of course, if the operations are properly defined and all the laws are respected
* Isolation of side effects – when it comes to FP, it is important to have a clear border between the pure and impure code. There are languages where this is required by the compiler (in Haskell, all side effects are contained in the IO Monad), but there are languages where this doesn’t happen (for example, Scala). It becomes the programmer’s job to make sure that all the side effects are properly managed and marginalized.

By making sure the code is side-effect free, it becomes easier to test, modify and also reason about it.

### Separation of pure/impure code

There is no real world application that does not have any side-effect – they are what bring meaning to some computations – be it storing in a DB, printing it out on the screen, reading/writing to files, etc.

This means that side-effects can’t be avoided – they are part of the development cycle, and cannot be eliminated.

In order to contain, isolate and better control side effects, there is a clear separation between the pure and impure code – this is generally done by making sure the core of an application is pure, and there is an impure layer which works with all the pure functions, giving them meaning.

By doing so, the programmer is sure that the most of the business logic is pure – meaning less error-prone, easy to test, easy to modify and easy to reason with, while all the “dirty” code won’t interfere with the pure one.

Let’s have an example: a function which reads from a file the content, and counts how many times each word comes up.

An initial implementation might look something similar to this:

**def** count(file: String): Unit = {  
 **val** content = Source.*fromFile*(file).getLines.toList  
  
 **val** words = content.flatMap(line => line.split(" "))  
 **val** groupedWords = words.groupBy(w => w).values  
 **val** counter = groupedWords.map(w => (w.head, w.length))  
 counter.foreach(word => *println*(s"""Word: **$**{word.\_1} Count: **$**{word.\_2}"""))  
}

It get the job done, but:

* What happens if the file doesn’t exist?
* What happens if the file can’t be read/isn’t text, generally not the wanted format?
* What happens if the file is empty?
* The function does more than it promises:
  + It reads the file
  + It splits the lines by empty space
  + It actually counts the words
  + Prints them out
* **def** readFromFile(file: String): Try[List[String]] = *Try* {  
   Source.*fromFile*(file).getLines().toList  
  }  
    
  **def** getWords = (lines: List[String]) => lines.flatMap(line => line.split(" "))  
  **def** groupWords = (words: List[String]) => words.groupBy(w => w).values.toList  
  **def** countWords = (groupedWords: List[List[String]]) => groupedWords.map(g => (g.head, g.length))  
  **def** prettyPrinter(wordsCount: List[(String, Int)]): Unit =  
   wordsCount.foreach(word => *println*(s"""Word: **$**{word.\_1} Count:**$**{word.\_2}"""))  
    
  **def** countFromFile(file: String): Unit = {  
   **def** counter = *getWords* andThen *groupWords* andThen *countWords* andThen *prettyPrinter  
    
   readFromFile*(file) **match** {  
   **case** *Success*(lines) => counter(lines)  
   **case** *Failure*(ex) => *println*(ex)  
   }  
  }

Now, having split the initial function into smaller, atomic function which deal with only 1 task, it is easy to compose them in Scala and get the final result.

Now, any IO related business is handled outside of the pure functions, and those pure functions only deal with input-output stuff, not having to worry about anything else.

This, in turn, has several benefits:

* Modularity – each function can be easily changed/adjusted so it does something different
* Easier to test – each function can be tested independently
* Easier to understand – it’s enough to look at the counter definition and it’s easy to understand the flow

This is a relatively simple example, but the more complexity, the harder it is to manage that.

Splitting that complexity into smaller, simpler task, might be the way to do it.

### ADTs and type classes

Algebraic data types (or simply put ADTs) are a mathematical representation of data.

As opposed to OOP, where data also has behavior added to it, ADTs only contain the definition of the data, no behavior defined whatsoever.

This way, there is a clear separation between data and functions operating on data – there is only a formal definition of how data looks like.

Using ADTs also enables to programmer to use pattern matching on that specific data, further enhancing the experience.

In the following section, there will be a demonstration of an ADT (trees) using Scala.



A formal definition of a tree is:

* Empty node
* Node with a value, a left tree, and a right tree

Simple as that. Now, a prettyPrinter for a tree will be defined.



For a given tree, the result will be similar to this:



While this works, there is indeed a better way of doing this by using a higher level abstraction – and that is expressed through type classes.

Type classes offer adhoc polymorphism – think of this as overloading. While OOP languages offer this feature as subtyping, functional programming do this by through the use of generics and specific implementations.

In the next section, a type class will be created in order to offer a generic pretty printer, and then, the type class will be used in order to write a specific implementation for the Tree ADT.

**trait** PrettyPrinter[A[\_]] {  
 **def** prettyPrinter(t: A[\_]): Unit  
}  
  
**object** PrettyPrinter {  
 **object** ops {  
 **implicit class** PrettyPrinterOps(t: Tree[\_]) {  
 **def** prettyPrinter(): Unit = {  
 *PrettyPrinter*[Tree].prettyPrinter(t)  
 }  
 }  
 }  
  
 **def** apply[A[\_]](**implicit** sh: PrettyPrinter[A]): PrettyPrinter[A] = sh  
  
 **implicit val** *treePrinter*: PrettyPrinter[Tree] = (t: Tree[\_]) => {  
 **def** go(curr: Tree[\_], depth: Int = 0): Unit = {  
 curr **match** {  
 **case** *Node*(v, l, r) =>  
 *println*("\t" \* depth + "Value: " + v)  
 go(l, depth + 1)  
 go(r, depth + 1)  
 **case** Nill => *println*("\t" \* depth + "Empty Node")  
 }  
 }  
  
 go(t)  
 }  
}

The Scala way of using type classes is through implicits.

Implicits can be seen as the dependency injection mechanism used in most OOP languages: through them, the compiler can look up definitions of methods/parameters when those are not found, and use the specific implementations found.

In the example shown above, there is a treePrinter implicit defined.

The apply function is what actually pulls out the specific implicit which is required.

Now, in order to have an idiomatic use of the prettyPrinter() function, an implicit class will be defined which receives a tree and has a single function, prettyPrinter(), which pulls out the implicit definition of PrettyPrinter for trees, and uses that function in order to print the tree.

## Object Oriented Best Practices

When talking about best practices in OOP, people usually talk about how the interaction between objects should take place – how specific classes exchange information and how they mutate state depending on some external stimulus.

### SOLID

When it comes to OOP principles, there is close to nothing that is more popular than the famous SOLID principles. They are general guidelines which help the programmer architect the flow of the application, how classes should look like, their hierarchy and relations to one another, their functionality and role in the flow.

SOLID stands for:

* Single Responsibility Principle – every class should deal with only 1 responsibility and 1 only. That should be the only reason a class should change.
* Open-closed Principle – this refers to a class’s ability to have extensible behavior, without modifying it. This is done through abstractions – the most basic example would be a program working with geometrical shapes. The system should be able to support adding new shapes without the rest of the algorithms suffer in any way. So, an interface should be created that is implemented by all shapes (Square, Round, etc.), and then use the interface in the specific algorithms, leaving dependency injection to deal with all of the details.
* Liskov’s Substitution Principle – the principle states that all derived classes should be replaceable by their base classes. For example, a Square can’t derive from a Rectangle, as they aren’t interchangeable.
* Interface Segregation Principle – an interface’s contract should only be tied to a specific functionality. This means the client should not be forced to implement something that is not going to be used. In other words, better have more finely grained interfaces than a few large ones.
* Dependency Inversion Principle – the flow of a program should be controlled by abstractions (read “interfaces”). Let’s suppose there is an algorithm which depends on a few factors – were it to be implemented using concrete implementations, there should be implementations for every possible combination of factors. The OOP way of doing this is by using interfaces – the algorithm utilizes interfaces only, and the concrete implementation is defined separately and given as input, injecting different factors into the algorithm.

These principles are the basis of a scalable, maintainable and reusable.

### High cohesion/low coupling

As previously mentioned, OOP is all about interaction, change, triggers and actions between classes.

This, indeed, can become problematic as complexity increases, and, if let unsupervised, the code base can become a nightmare for the programmer to work with – the relations between classes are too tight, changes to one single class might impact a dozen others, and it’s a domino effect that could potentially lead to an impossible to extend codebase.

In order to prevent this unwanted scenario, the principle of “high cohesion/low coupling” came up and it suggests the following:

* High Cohesion – every single element of a class should be closely tied together. This means that attributes are closely related to the method attached to the object, and it doesn’t deal with anything apart from that. There are several advantages:
  + Modularity – every single class deals with a single, small part of a flow
  + Extensibility – since classes are almost atomic, this means it is slightly easier to add functionality as compared to classes which do more than they should
  + Easier to read – this means that most classes do what they specify, without having to “select” information based on what one wants to achieve
* Low Coupling - the relations between classes should be as few, and as weak as possible. This basically translates to low/no dependencies between classes, how much a class knows about another class. The benefits to doing so are the following:
  + Easier to modify – since there aren’t a lot of dependencies between objects, it means changing the behavior of one class will not be too impactful on the application.
  + Readability – a lot of relations and dependencies means a lot of reading until one is able to fully understand a flow.

## Conclusion

By now, it should be rather clear just how the two paradigms expect their programmers to think about the piece of code they have to design.

There are major differences between the mentalities imposed OOP and FP, and these would be:

* The key/core of the paradigm:
  + FP – functions
  + OOP – classes
* What best practices put emphasis on:
  + FP – how functions should behave/interact without mutating state
  + OOP – how classes should behave/interact in order to change
* What should be avoided (restrictions/bad practices):
  + FP – mutable state, mixing side effects with pure functions
  + OOP – too many dependencies between classes, classes doing too much
* What should be done:
  + FP – atomic functions, a side effect free core which the side effects can use
  + OOP – atomic classes, with low dependencies and working mostly with interfaces (which are basically an abstraction)

# Design patterns

In OOP, when people discuss about design patterns, the subject of the talk is mainly about the interactions between classes – how does one class integrate in relation to the other classes, how they communicate and what they “exchange” in terms of data, how they mutate their internal state based on some external stimulus. However, when it comes to the actual implementation of a design pattern, it is largely based on what the programmer wants to achieve – so the notions of design patterns in OOP is more of a hint on how the interaction between data should be.

However, when it comes to functional programming, the idea of design patterns is more about what should be the actual result, without much care about how that should take place. And that is done through the use of functions – for a given input, there should be a given output – no emphasis on any relations - if any - and since mutability is more of a “coding smell”, it is not based on the idea of how data interacts, but more about how a specific object will be changed to produce a new result – based around the idea of input-output. More than that, since purity is a must, most design patterns – which sound more like mathematical functions – also have some laws that need to be met in order for an operation to be valid. This, of course, enhances predictability and creates a rather stable developing environment.

The main difference between the mentalities of OOP vs FP design patterns is the following:

* OOP – how data interacts to change some state, and how it should be done, with little to no care about the final result
* FP – how a function should change the input into an output, with little to no care about the actual implementation.

## Functional design patterns

There is also another mentality shift when it comes to functional programming – getting accustomed to the design patterns also includes thinking about programming as a dual world – there is the normal world, the world where one knows what there is in terms of data available, and there is another world – the functional world, a world of possibility.

The functional world can mostly be seen as a box which might or might not contain something – the programmer keeps applying functions and logic over the “box”, but it is never really known what’s inside the box until it is opened up. As a functional programmer, one wants to lift some data up into the functional world and keep applying logic and functions over that until the result is needed for some IO interaction – that’s when the data “returns” to the real world.

There are plenty of examples where this concept is easily explained, mapping over Option/Maybe Monads in Scala/Haskell being the most usual one. The mentioned Monad has 2 possible outcomes:

* Some(a) / Just(a) – in the case of a value being present, more commonly seen as the actual result
* None / Nothing – in case something went wrong

It is indeed difficult at first to think in terms of 2 different worlds, and it does take some exercise to get used to it, but getting accustomed to it meaning one will be able to think on a higher level of abstraction about what the goal is.

### Functors

While many programmers might not actually be familiar with the functional design patterns, they are more often than not used.

Functors on their own are among the easier to understand functional abstractions, but they are of great importance when talking about something slightly more complex like Monads or Applicative Functors.

Basically, a functor is anything with a **map** method – countless examples emerge, from List, Set, Map, all the way to Option and so on, and maybe the lesser known functors like Scala’s Try and/or Either.



What is useful about map has been also previously mentioned: when the programmer lifts a data of type A into a functor like List[A], it doesn’t matter how many **maps** have been done until the result is actually evaluated and it is “returned” into the real world. Until then, it’s only a black box.

The formal definition of a Functor is the following:

**trait** Functor[F[\_]] {  
 **def** map[A, B](f: A => B, data: F[A]): F[B]  
}

Some elements will be explained:

* F[\_] – is actually what lifts the A into the other world, it is the box that contains the result.
* The map() function receives 2 arguments:
  + A function which transform an A into a B, and some data that is actually the “**box**” of type A, and returns the transformed box

In the introduction about design patterns, it has been mentioned that most design patterns have some laws associated with them.

When it comes to functors, there are 2 laws that need to be met:

1. Identity law – calling map with the identity function is the same as doing nothing



1. Composition law – calling map with an f() function and then with the g() function is the same as calling map over g(f()).



The laws are meant to enforce that things will not get “weird” and non-deterministic.

There are numerous types of functors, among which:

* Bifunctors
* **trait** BiFunctor[F[\_, \_]] {  
   **def** biMap[A, B, C, D](fab: F[A, B])(f: A => C)(g: B => D): F[B, D]  
   **def** first[A, C](fab: F[A, \_])(f: A => C): F[C, \_  
   **def** second[B, D](fab: F[\_, B])(f: B => D): F[\_, D]  
  }

As seen, a bifunctor is a functor that contains 2 possible results.

The most popular Scala example is Either[A, B] with two implementations: Left[A], Right[B], which is usually used to hold an exception in the left side if there is any, and the actual result in the right side.

The laws remain the same, except they apply to both first() and second() functions.

* Multifunctor – is basically a generalization of the bifunctor – so for example a bifunctor is a multifunctor of type 2.
* Applicative Functor – this is actually somewhere in the middle ground between a functor and a monad
* **trait** ApplicativeFunctor[F[\_]] {  
   **def** apply[A, B](fa: F[A])(f: F[A => B]): F[A] => F[B]  
  }

What this Functor actually does is create a pipeline of functions that need to be applied, and those will be applied only when the “unboxing” is done.

* Endofunctors – these are just regular Functors that map to themselves.

### Monads

The Monad is one of the most famous term when it comes to functional programming, and it’s been the reason why a lot of programmers generally fear FP – lots of articles have been written about it, and there is still some confusion when it comes to what it actually does and what does it mean.

Most Scala/Haskell programmers probably used Monads at least once, but they didn’t know it.

Simply put, a Monad is anything with a **flatMap**() method attached to it.



So here, clustersWithCentroids is a List[A] – does this make List a Monad? Yes, it does – same with Option.

Apart from that, Scala has better support for Monads through the **for comprehension** structure control.



More formally put, a Monad is a mechanism which enables sequencing computations.

But, this is mainly what a Functor also is – a bunch of **map**s applied one after the other.

However, a Functor doesn’t account for any complications that may arrive on the “way”, while the Monad does. The Functor doesn’t care whether the function attached to it produces another List, or another Option.

The Monad does indeed.



This is a small example in order to illustrate what a Functor produces compared to what a Monad produces.

Once again, the divide function will be used in order to produce an Option[Double].

**def** division(a: Double)(b: Double): Option[Double] = {  
 **if** (b == 0.0)  
 None  
 **else** *Some*(a / b)  
}

HEREHEHRHERHEHRHERHEHREHRHERH

So, our value divisionOf10 waits for input in order to divide 10 by some Double, and returns an Option result.

The result for the following computations will be printed:



And they are:



As easily seen, the flatMap() method accounted for any complications that emerged during computations, while the map() method just went along with it.

The definition of a Monad is as followed:

**trait** Monad[F[\_]] {  
 **def** pure[A](value: A): F[A]  
  
 **def** flatMap[A, B](value: F[A])(func: A => F[B]): F[B]  
}

Compared to the Functor, there is one small change when it comes the function that it receives as a parameter: the function receives an A which is returned as a Monad of B.

The **pure** method is just a way to “lift” normal values into the Monadic world.

And just as Functors have laws, so do Monads have a few:

1. Left identity: calling **pure** on an A and flat-mapping that to a function should be the same as applying that function over A.



1. Right identity: passing **pure**  to a flatMap function should have no effect



1. Right and left associativity: flatMapping using 2 functions f() and g() is the same as flat-mapping over f and then over g



As with Functors, there are a lot of monad types, but the more basic and easier to understand ones would be:

* State Monad – as the name suggests, it is a control monad where there is usually an initial state, and the goal is to get the a final state, while keeping track of some changes
* IOMonad – using this monad is the only way to produce any side effects in a language like Haskell. It uses the monad to actually capture and control any side effects there might be. Haskell has some form of syntactic sugar in order to make this a more pleasant experience (under the form of **do**).

More advanced monads include: comonads, gonads, free-monad, co-state monad.

### Pipelining

The reason behind why most functional design patterns focus on functions is simple – pipeling through composition.

One interesting analogy is thinking about the flow of the program as a rails – a function represents a rail which might split into 2 other rails, depending on the result – for example, for **Option**, those 2 rails might be Some or None.

Ideally, one might want to compose the function returning an Option[] with something else – through composition, the flow might continue on Some, but do something else on None. This should go on and on until the flow is completed.



This is closely related to working in the functional world, previously mentioned. This way, the flow will disregard dealing with any errors until the result needs to be extracted out.

Compared to OOP, where most methods might throw errors and the programmer has to deal with them in every single method utilizing that, pipelining reduces a lot of the boilerplate produced by error handling.

And since purity is a must, the flow produced by using this pattern will be stable and predicting.



In the following section, there will be a type class implementation for some operations on the previously implemented tree.

It deals with doing a DFS on a tree, and the reason behind this is in order to find out the common ancestors between 2 elements.

**implicit val** *treeOps*: TreeOps[Tree] = **new** TreeOps[Tree] {  
 **override def** DFS[A](t: Tree[A], el: A): Option[List[A]] = {  
 **def** go(curr: Tree[A], road: List[A]): Option[List[A]] = {  
 curr **match** {  
 **case** Nill => None  
 **case** *Node*(v, l, r) =>  
 **if** (v == el)  
 *Some*(road :+ v)  
 **else** {  
 **val** left = go(l, road :+ v)  
 **val** right = go(r, road :+ v)  
  
 left orElse right  
 }  
 }  
 }  
  
 go(t, *List*.*empty*)  
 }  
  
 **override def** commonAncestor[A](t: Tree[A], el1: A, el2: A): Option[A] = {  
 **def** roads = **for** {  
 firstDFS <- **this**.DFS(t, el1)  
 secondDFS <- **this**.DFS(t, el2)  
 } **yield** firstDFS.zip(secondDFS)  
  
 roads.map(r => r.takeWhile(e => e.\_1 == e.\_2).last.\_1)  
 }  
}

The DFS function is pretty straight forward – remember the road until the current Node, and if that Node is Empty, then it means it doesn’t exist in the tree => None. Otherwise, return the road.

Now, the DFS can be seen as a track:



Given this, now let’s talk about the commonAncestor function: receiving 2 elements, find their common ancestor.

Using pipelining, there is a call to get both DFS for both elements – returning a track for both of them.

Now, if one of those tracks is None, then nothing else happens and the function returns a None. But, if there is a result, the logic is continued – zipping both roads, another track is created.

And finally, using the previous track, all is glued together and it is tried to extract the last common element of the 2 lists, which might not exist – thus, returning another track. This can continue on forever until an actual result needs to be evaluated.

Compared to the OOP approach, where multiple exceptions should have been thrown if the element is not found, if there was no common ancestor and so on.

This way, all the useless and heavy code caused by exceptions is thrown out the window, and only the relevant one remains – having a clean, easy to both read and understand flow.

## Object Oriented Design Patterns

As opposed to the FP paradigm, the OOP design patterns are all about state and interaction – triggering actions based on some stimulus received or created. This way, the program feels like it is coming alive and it might initially be easier for the programmer to better understand a project.

There are 3 categories of design patterns in OOP:

1. Creational – trying to deal with object instantiation in an elegant, simple way. Examples include: Builder, Singleton, Factory, and Abstract Factory.
2. Behavioral – trying to deal with common interactions between classes and encapsulate them in a more generic way. Examples include: Memento, State, and Observer.
3. Structural – closely resembling behavioral patterns, it deals with easing the relationships between classes. Examples include: Adapter, Bridge, Decorator, and Façade.

There won’t be much detail about this in the paper since it is out of scope, but it is worth mentioning.

### Builder

The builder is one of the most stumbled upon OOP design pattern, and it deals with one important answer to a question: how does one go about creating a complex object?

The “traditional” way of creating objects is through constructors. However, as object complexity increases, so do the number of parameters. This way, the class blows out of proportion and the code becomes hard to maintain and understand.

The builder comes in and separates the “construction” part from the representation part – this way, the code is easier to understand, maintain, and most importantly, control.

In the following section, a builder for a Person (with 4 attributes: first name, last name, birthdate and nationality) will be created.

public class PersonBuilder {  
 public String firstname;  
 public String lastname;  
 private Date birthdate;  
 public Nationality nationality = Nationality.*Ro*;  
  
 public PersonBuilder with(Consumer<PersonBuilder> builderFunction) {  
 builderFunction.accept(this);  
 return this;  
 }  
  
 public Person createPerson() {  
 return new Person(firstname, lastname, birthdate, nationality);  
 }  
  
 public void setBirthdate(String birthdate) throws InvalidValue, ParseException {  
 SimpleDateFormat sdf = new SimpleDateFormat("dd-M-yyyy");  
 Date d = sdf.parse("01-01-1950");  
 Date date = sdf.parse(birthdate);  
 if (date.after(d))  
 this.birthdate = date;  
 else throw new InvalidValue("Invalid date!");  
 }  
}

Without getting too much into a detail, a Consumer in Java is the way of “grouping” actions. Now, all the validation logic can be moved into the Builder, so that the actual class will only have a constructor and getters. This, again, encourages immutability in OOP, which might be good in some cases where Threads are popular.

In order to create a Person, one can do the following:

public class Main {  
 public static void main(String[] args) {  
 Person person = new PersonBuilder()  
 .with(personBuilder -> {  
 personBuilder.firstname = "Mr.";  
 personBuilder.lastname = "John";  
 personBuilder.nationality = Nationality.*En*;  
 try {  
 personBuilder.setBirthdate("01-01-1900");  
 } catch (InvalidValue | ParseException invalidValue) {  
 invalidValue.printStackTrace();  
 }  
 })  
 .createPerson();  
  
 System.*out*.println(person);  
 }  
}

Running this will, of course, break the program because of the restriction on the birthdate.

Builders offer quite a few advantages:

* Separation of data from the actual logic of validating it
* Flexibility in terms of instantiation
* Cleaner code
* Easily maintainable
* Enables immutability in the class it’s building for

When it comes to the negative parts of a Builder, there are close to none – it really comes down to the programmer to evaluate and decide whether it’s worth having Builders:

* Is it beneficial to easily enable immutability?
* Is that immutability required?
* Is the class complex?
* Is it worth it, considering it adds quite a lot of code to the project?

### Factory

When it comes to interaction with the user, or anything related to instantiating objects at runtime, the Factory is the best option for doing so. It is used in order to provide an interface for creating objects, while hiding the specific instantiation details.

The Factory design pattern is an elegant solution to the previously mentioned problems, creating an easy to maintain and read code base.

In the following example, it’s been put up a roulette game that every 3 seconds gives the client a new gift.

public class Roulette {  
 public void run() throws InterruptedException {  
 while (true) {  
 System.*out*.println("Let's see what the russian roulette will give you");  
  
 System.*out*.println(generateGift(GiftType.*randomGift*()).message());  
  
 Thread.*sleep*(3000);  
 }  
 }  
  
 public Gift generateGift(GiftType gift) {  
 switch (gift) {  
 case *Drink*:  
 return new Drink();  
 case *Car*:  
 return new Car();  
 case *Nothing*:  
 return new Nothing();  
 case *Toy*:  
 return new Toy();  
 case *Error*:  
 return new Nothing();  
 default:  
 return new Nothing();  
 }  
 }  
}

This way, all the instantiation-related code is kept in one place, easy to maintain and extend.

### Memento

As OOP is mostly based on mutating state, there are cases when it is crucial for the business logic to keep track of changes throughout the history of an object – 2 basic uses would be the undo/redo functionalities.

In this case, the Memento design pattern is the solution – the flow is designed so that it keeps track of changes throughout the object’s lifetime.

### Observer

The Observer Design pattern is useful in defining dependencies between a list of Subjects and a list of Observants – thus being useful in building even-driven systems.

It is again dependent on mutable state, as notifications are usually pushed when something changes in an object’s internal state.

At its core, the design is rather easy to understand: there are 2 main participants, the Subject and the Observer.

public interface Subject<T extends String> {  
 void subscribe(Observer o);  
 void unsubscribe(Observer o);  
 void notifyObservers();  
 String getLastNotification();  
 void addNotification(T notification);  
}

public interface Observer {  
 void receiveNotification(Subject from);  
}

It is quite a simple and elegant solution to keeping track of changes across multiple classes. The Observer design pattern is, again, dependent on mutable state.

## Conclusion

Following the best practices section, design patterns further accentuate what the focus of either paradigms is: functions, for FP, and classes, for OOP.

The general differences between the 2 paradigms when it comes down to design patterns are:

* Emphasis:
  + FP – functions/input-output
  + OOP – classes – how they exchange state/data and how they mutate
* Restrictions:
  + FP – clearly defined laws whose importance is to make sure there won’t be any unwanted behavior
  + OOP – no laws, just overall a definition of the design patterns
* Difficulty:
  + FP – quite difficult to understand and get a better grasp of them
  + OOP – easy to understand and apply, only downfall is that there are a lot of them

# Demo explained

The purpose of the demo is to give insights on how code is being written under the two paradigms – OOP and FP.

In order for this to be clearly visible, I had to come up with a demo that meets the requirements:

* The problem is well known
* The problem has to be easy to understand – so the idea is placed on how the code is written, rather than what the problem actually is
* The problem has to have some good amount of details and complexity – this is necessary in order to underline how the paradigms behave. Anything that is not too complex, and there won’t be much to compare to.

After much thought, I believe that the Hierarchical Clustering Problem is a good fit – there are some relations between data, and quite a good amount of logic applied on them.

The Hierarchical Clustering Problem is a Machine Learning problem which deals with aggregation of points in different clusters, in order to build a hierarchy.

The input is represented as a set of Points, and the end result is a dendogram (tree) where points are clusterized on different levels.

The implementations should underline the following:

* How a paradigm impacts the architecture
* How data is being used and logic is applied to them
* How extensibility is offered and under which conditions
* How readable the code is
* How much code is being written

Conclusion

The purpose of this case study was to give insights to 2 of the more popular programming paradigms: OOP and FP.

As most programming languages mix FP into what would otherwise be an OOP language, I feel there is great confusion among programmers as to what elements belong to which paradigm. Over the past years, the industry required multithreading more and more with the development of multi-core CPUs, so new programming best practices, libraries and frameworks emerged that basically are founded on functional grounds:

* C#’s LINQ – SQL-like wrapper which uses higher order functions to process data
* Spark – a superset of Hadoop which uses the Map/Reduce programming model. The model is used to process large amounts of data in parallel, so it’s written in a functional style ( the JVM Spark is actually written in Scala )
* Immutability which is a desirable trait in most apps
* Java’s Stream collection – contains higher order functions like map, reduce, filter, etc.
* Java’s Spring 5 – introducing new functional elements, mainly function composition for controllers

More than this, FP is infamous for the harder to grasp concepts and mind bending required to write anything substantial, or so some programmers say. Through this work, I tried to shed some light on some functional concepts that might seem difficult, but in fact they are not – indeed, the learning curve is steeper, but it’s not something that should fear the programmer – it should intrigue him, leaving him wanting to find out more.

Because of all of this, I personally believe it’s important for the coder to know the origins, history and how the 2 paradigms differentiate one another, and more importantly, how to combine them in order to produce the best quality code possible.

Apart from this, I believe it’s important to be familiar with the terminology, uses and specific details about the tools that are being used – best practices, design patterns, etc. – are terms the programmer should actively think about when designing a piece of code, not something that comes intuitively. In order to aid this, the paper tries to explain as good as possible most of the terminologies found in both paradigms, as well as how they are useful, and also how they differ from their counterparts.

Ultimately, either paradigm WILL get the job done – however, since working smart is better than working hard, it is essential to be familiar with the available tools, and more importantly, know what tools are better appropriate to certain tasks compared to others.

In conclusion, the paper approaches two different paradigms and they are both studied in order to find out differences/similarities between them, with the goal of offering the programmer a broader view of what is accomplishable with either one of them, but ultimately familiarize and present what each paradigm offers.