

Functional vs OO programming

Case study



Functional vs OO programming

Table of contents

Contents

[History 1](#_Toc516562922)

[Functional languages 1](#_Toc516562923)

[Object Oriented languages 4](#_Toc516562924)

[Current state of the industry 4](#_Toc516562925)

[Functional programming elements 4](#_Toc516562926)

[Immutability 4](#_Toc516562927)

[Functions 4](#_Toc516562928)

[Higher order functions 5](#_Toc516562929)

[Recursion 7](#_Toc516562930)

[Purity and side effects 9](#_Toc516562931)

[Referential transparency and first-class functions 10](#_Toc516562932)

[Benefits/disadvantages 12](#_Toc516562933)

[Object Oriented programming elements 13](#_Toc516562934)

[Classes 13](#_Toc516562935)

[Inheritance 13](#_Toc516562936)

[Encapsulation 13](#_Toc516562937)

[Polymorphism 13](#_Toc516562938)

[Dynamic binding 13](#_Toc516562939)

[Relations -> passing state and modifying it internally 13](#_Toc516562940)

[Benefits/disadvantages 13](#_Toc516562941)

[Functional Best Practices 13](#_Toc516562942)

[Pattern matching for the rescue 13](#_Toc516562943)

[Function composition 15](#_Toc516562944)

[Side effect free 16](#_Toc516562945)

[Separation of pure/impure code 17](#_Toc516562946)

[ADTs and type classes 19](#_Toc516562947)

[Object Oriented Best Practices 22](#_Toc516562948)

[SOLID 22](#_Toc516562949)

[Single Responsibility Principle 22](#_Toc516562950)

[Open/Closed Principle 22](#_Toc516562951)

[Liskov Substitution Principle 22](#_Toc516562952)

[Interface segregation Principle 22](#_Toc516562953)

[Dependency Inversion Principle 22](#_Toc516562954)

[YAGNI 22](#_Toc516562955)

[High cohesion/low coupling 22](#_Toc516562956)

[Interfaces instead of implementation 22](#_Toc516562957)

[Functional Design Patterns 22](#_Toc516562958)

[What it’s all about? Functions, Functions, Functions 22](#_Toc516562959)

[Variance/Covariance/Contravariance 23](#_Toc516562960)

[Functors 23](#_Toc516562961)

[Arrow 23](#_Toc516562962)

[Monads 23](#_Toc516562963)

[Pipelining 23](#_Toc516562964)

[Object Oriented Design Patterns 23](#_Toc516562965)

[Builder 23](#_Toc516562966)

[Factory 23](#_Toc516562967)

[Bridge 23](#_Toc516562968)

[Chain of responsibility 23](#_Toc516562969)

[Command 23](#_Toc516562970)

[Memento 23](#_Toc516562971)

[Observer 23](#_Toc516562972)

[State 23](#_Toc516562973)

[Strategy 23](#_Toc516562974)

[Visitor 23](#_Toc516562975)

[Demo explained 23](#_Toc516562976)

[Distributed systems 23](#_Toc516562977)

[History 23](#_Toc516562978)

[Current state of the industry – why is it needed 23](#_Toc516562979)

[Why choose Functional over OOP 23](#_Toc516562980)

# 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.





Despite being a simple example, the first example speaks for itself – one read 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.

First introduced in the 1930s by the mathematician Alonzo Church, lambda calculus consists of constructing terms and performing operations on them. More than that, there are only 3 rules that are used to build terms:



Reductions consist of the following operations:



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 in turned later pretty much defined Haskell.

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
* Current state of the industry

Ever since Java has emerged as a programming language, the industry has been ruled by the OOP paradigm.

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 what better than FP, whose core is based on immutability?

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.

## Object Oriented languages

## Current state of the industry

# Functional programming elements

## Immutability

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

## Functions

## Higher order functions

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:



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:



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 written properly, can become quite similar to a mathematical representation of a function.

A simple example is the Fibonacci sequence:



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

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



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.

Scala’s approach to tail recursion is slightly different – at compile time, scalac (the compiler) will write some optimization, basically rendering the recursion into a “**while**” loop.

## 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 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.

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.



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.

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



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, he 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 implements some of the more advanced concepts without having fulfilled these basic requirements.

When it comes to advantages, there are plenty:

* Predictability – the code behaves at 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

## Classes

## Inheritance

## Encapsulation

## Polymorphism

## Dynamic binding

## Relations -> passing state and modifying it internally

## Benefits/disadvantages

# Functional Best Practices

## 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.





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.



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.



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



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 return type of the second function – the newly found 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.





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
* 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:



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



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.



The Scala way of using type classes is through implicits.

Implicits can be seen as the dependency injection mechanism used in most OOP languages: though them, the compiler can look up for 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 prin the tree.

# Object Oriented Best Practices

## SOLID

## Single Responsibility Principle

## Open/Closed Principle

## Liskov Substitution Principle

## Interface segregation Principle

## Dependency Inversion Principle

## YAGNI

## High cohesion/low coupling

## Interfaces instead of implementation

# 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.

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 he has 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 he’s trying to achieve.

## Variance/Covariance/Contravariance

## Functors

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



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



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

The most popular Scala example is Either[A, 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



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 – thousands 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, clusterseWithCentroids 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** applied one after the other.

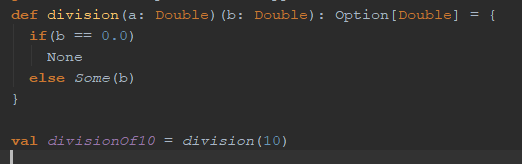
However, a Functor doesn’t account for any complications that may arrive on the “way”, while the Monad does. The Functor doesn’t care whether the function attached to it produces another List, or another Option.

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].



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:



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 laws:

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.

## Arrow

## 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.



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 exception is thrown out the window, and only the relevant one remains – having a clean, easy to read and understand flow.

# Object Oriented Design Patterns

## Builder

## Factory

## Bridge

## Chain of responsibility

## Command

## Memento

## Observer

## State

## Strategy

## Visitor

# Demo explained

# Distributed systems

## History

## Current state of the industry – why is it needed

## Why choose Functional over OOP