Functional vs OO programming

Table of contents

* **History**
* **Functional languages**
* **Object Oriented languages**
* **Current state of the industry**
* **Functional programming elements**
* **Immutability**
* **Functions**
* **Higher order functions**
* **Recursive functions**
* **Purity and side effects**
* **Referential transparency**
* **Benefits/disadvantages**
* **Object Oriented programming elements**
* **Classes**
* **Inheritance**
* **Encapsulation**
* **Polymorphism**
* **Dynamic binding**
* **Relations -> passing state and modifying it internally**
* **Benefits/disadvantages**
* **Functional Best Practices**
* **Function composition**
* **Side effect free**
* **Separation of pure/impure code**
* **ADTs and separating data from functions**
* **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**
* **Functional Design Patterns**
* **What it’s all about? Functions, Functions, Functions**
* **Variance/Covariance/Contravariance**
* **Type classes**
* **Functors**
* **Functor**
* **Bifunctor**
* **Multifunctor**
* **Profunctor**
* **Applicative Functor**
* **Arrow**
* **Monads**
* **Monad**
* **Free Monad**
* **IO Monad**
* **State Monad**
* **Costate Monad**
* **Comonad**
* **Functional Design Patterns in Action**
* **Scala Cats**
* **Scalaz**
* **Haskell Standard Library**
* **Object Oriented Design Patterns**
* **Builder**
* **Factory**
* **Singleton (??)**
* **Bridge**
* **Decorator**
* **Chain of responsibility**
* **Command**
* **Interpreter**
* **Memento**
* **Observer**
* **State**
* **Strategy**
* **Visitor**
* **Object Oriented Design Patterns in Action**
* **Java Swing/FX**
* **Java Spring**
* **Web Programming**
* **FRM – functional relational mapping**
* **What is it**
* **Slick**
* **Examples**
* **Why?**
* **ORM – objection-relational mapping**
* **What is it**
* **JDBC/JPA/Hibernate**
* **Examples**
* **Why?**
* **Web Services**
* **Functional Web services**
* **Scala with Akka**
* **Scala with Play**
* **Example – mail service or something similar**
* **Object Oriented Web Services**
* **Java with Spring**
* **Example – mail service or something similar**
* **Controllers**
* **Functional Controllers**
* **Scala with Akka**
* **Example – how and come**
* **OO Controllers**
* **Java with Spring**
* **Distributed systems**
* **History**
* **Current state of the industry – why is it needed**
* **Why choose Functional over OOP**
* **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?**