Functional vs OO programming

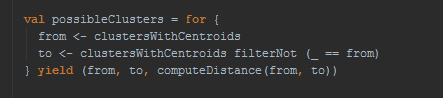
Table of contents

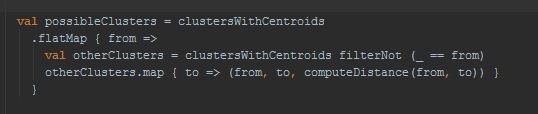
1. **History**
   1. **Functional languages**
   2. **Object Oriented languages**
   3. **Current state of the industry**
2. **Functional programming elements**
   1. **Immutability**
   2. **Functions**
   3. **Higher order functions**
   4. **Recursive functions**
   5. **Purity and side effects**
   6. **Referential transparency**
   7. **Benefits/disadvantages**
3. **Object Oriented programming elements**
   1. **Classes**
   2. **Inheritance**
   3. **Encapsulation**
   4. **Polymorphism**
   5. **Dynamic binding**
   6. **Relations -> passing state and modifying it internally**
   7. **Benefits/disadvantages**
4. **Functional Best Practices**
   1. **Function composition**
   2. **Side effect free**
   3. **Separation of pure/impure code**
   4. **ADTs and separating data from functions**
5. **Object Oriented Best Practices**
   1. **SOLID**
      1. **Single Responsibility Principle**
      2. **Open/Closed Principle**
      3. **Liskov Substitution Principle**
      4. **Interface segregation Principle**
      5. **Dependency Inversion Principle**
   2. **YAGNI**
   3. **High cohesion/low coupling**
   4. **Interfaces instead of implementation**
6. **Functional Design Patterns**
   1. **What it’s all about? Functions, Functions, Functions**
   2. **Variance/Covariance/Contravariance**
   3. **Type classes**
   4. **Functors**
      1. **Functor**
      2. **Bifunctor**
      3. **Multifunctor**
      4. **Profunctor**
      5. **Applicative Functor**
   5. **Arrow**
   6. **Monads**
      1. **Monad**
      2. **Free Monad**
      3. **IO Monad**
      4. **State Monad**
      5. **Costate Monad**
      6. **Comonad**
7. **Functional Design Patterns in Action**
   1. **Scala Cats**
   2. **Scalaz**
   3. **Haskell Standard Library**
8. **Object Oriented Design Patterns**
   1. **Builder**
   2. **Factory**
   3. **Singleton (??)**
   4. **Bridge**
   5. **Decorator**
   6. **Chain of responsibility**
   7. **Command**
   8. **Interpreter**
   9. **Memento**
   10. **Observer**
   11. **State**
   12. **Strategy**
   13. **Visitor**
9. **Object Oriented Design Patterns in Action**
   1. **Java Swing/FX**
   2. **Java Spring**
10. **Web Programming**
    1. **FRM – functional relational mapping**
       1. **What is it**
       2. **Slick**
       3. **Examples**
       4. **Why?**
    2. **ORM – objection-relational mapping**
       1. **What is it**
       2. **JDBC/JPA/Hibernate**
       3. **Examples**
       4. **Why?**
    3. **Web Services**
       1. **Functional Web services**
          1. **Scala with Akka**
          2. **Scala with Play**
          3. **Example – mail service or something similar**
       2. **Object Oriented Web Services**
          1. **Java with Spring**
          2. **Example – mail service or something similar**
    4. **Controllers** 
       1. **Functional Controllers**
          1. **Scala with Akka**
          2. **Example – how and come**
       2. **OO Controllers**
          1. **Java with Spring**
11. **Distributed systems**
    1. **History**
    2. **Current state of the industry – why is it needed**
    3. **Why choose Functional over OOP**
12. **History** 
    1. **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.**

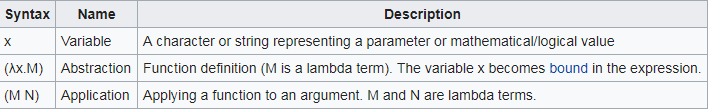




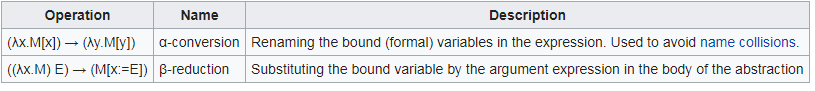
**Despite being a simple example, the first example speaks for itself – one read and it should be clear what it does, while in the second example some deciphering and a more thorough read might be required – despite the same amount of rows. This boils down to the programmer’s desire of wanting to improve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

1. **garbage collection as a method of dealing with unused memory cells**
2. **closures – for static scoping**
3. **conditional expressions and use for writing recursive functions ( first ever language to do that )**
4. **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**