

**Understanding of Functional Programming**

**in Solid Principles**

# 

**Submitted By:**

**Rishi Kumar Ray**

Functional programming is a paradigm which has its roots in mathematics, primarily stemming from

lambda calculus. Functional programming aims to be declarative and treats applications as the result of

pure functions which are composed with one another.

Adopting these practices can also contribute to avoiding code smells, refactoring code, and Agile or

Adaptive software development.

"Complexity is anything that makes software hard to understand

or to modify." — John Outerhout

## The principles of Functional Programming

### **Pure Functions**

A pure function is a function which:

1. Given the same inputs, always returns the same output
2. Has no side-effects

### **No side effects**

Side effects are treated as evil by the functional programming paradigm. Side effects are things such

as I/O, logging to the console, thrown and halted errors, network calls and the alteration of an external data

structure or variable. Basically anything which makes a system unpredictable.

### **Immutability**

Immutability is at the core of functional programming. Immutability is the idea that once a value

is declared, it is unchangeable and thus makes behaviours within your programme far more predictable.

### **Referential transparency**

Referential transparency is a fancy way of saying that if you were to replace a function call with its

return value, the behaviour of the programme would be as predictable as before.

### **Functions as first-class entities**

This just means that functions are able to be passed as arguments to other functions, returned as

values from other functions, stored in data structures and assigned to variables.

### **Higher order functions**

Higher order functions are functions which do at least one of the following:

1. Takes one or more functions as arguments
2. Returns a function as its result

**In this article, we will be discussing these 3 principles.**

* Single Responsibility Principle (SRP)
* Open Closed Principle (OCP)
* Liskov Substitution Principle (LSP)

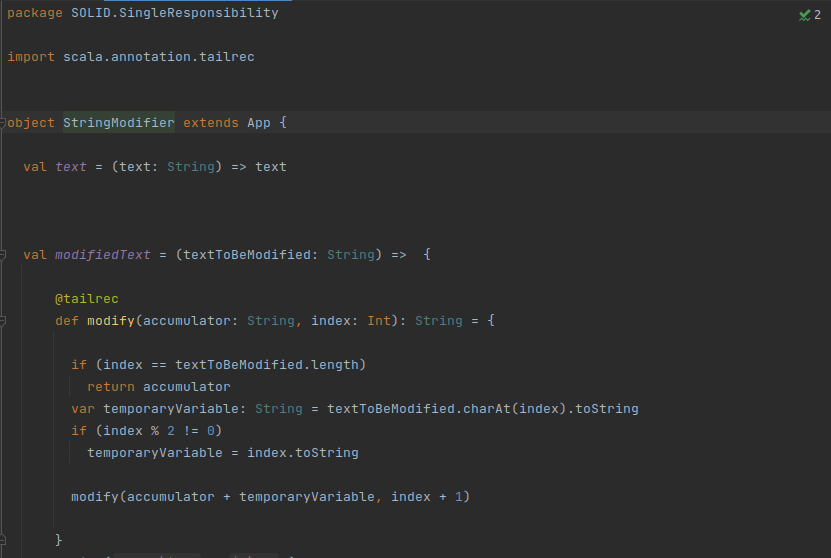
### 

Here we have Implemented functional Programming in the these.

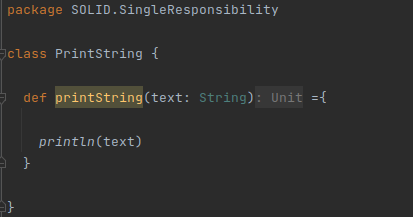
**=>Single Responsibility Principle (SRP) -**

"An object should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the object."

Although functional programming languages don't have classes the same principle holds true. Functions should be small reusable pieces of code that you can compose freely to create complex behavior.

****

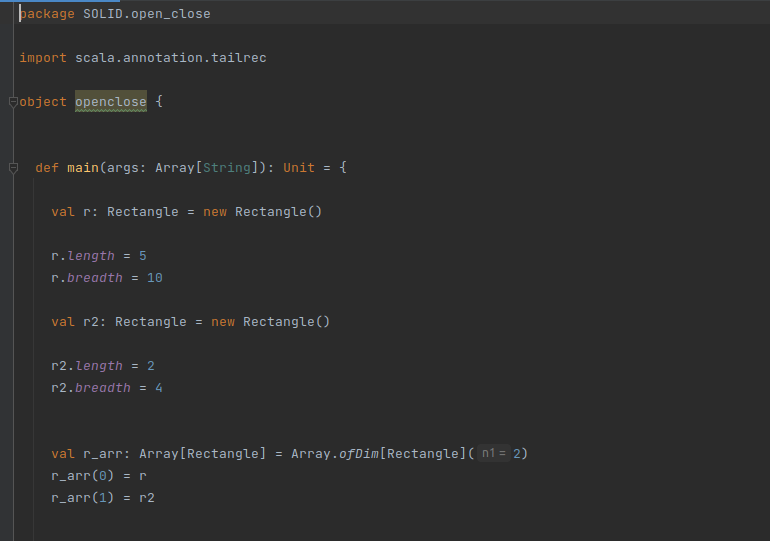


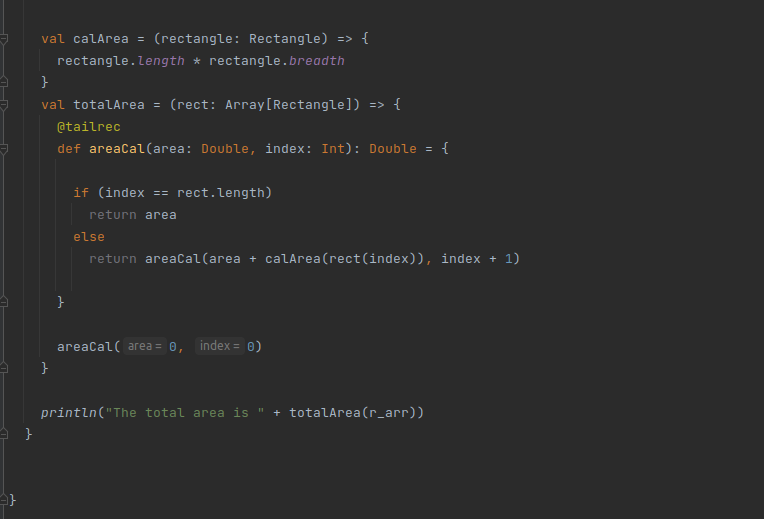


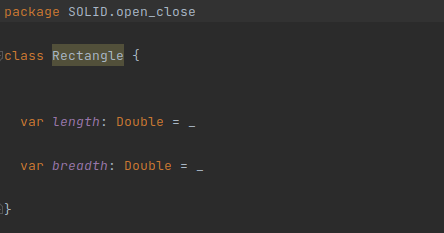
## =>Open-Closed Principle

"Software entities ... should be open for extension, but closed for modification."

Instead of using inheritance, Functional Programming achieves this by using two tools.





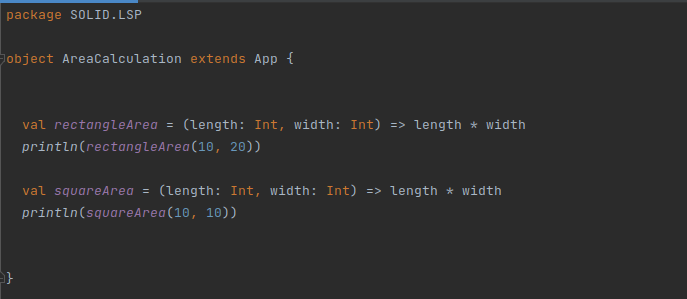


## =>Liskov Substitution Principle

"Objects in a program should be replaceable with instances of their subtypes without

altering the correctness of that program."

Again when people generally think about this principle the first idea that comes to their head is that if the parent class has some behavior, their children should not break that behavior, but this is not the only applicable case, LSP also applies in case we use generic or parametric programming where we create functions that work on a variety of types, they all hold a common truth that makes them interchangeable.



## Conclusions

Functional programming gives us some principles which make our code more readable, predictable and testable. This allows us to have code which contains less bugs, easier onboarding.