**SOLID Principles With Functional Programming**

SOLID is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin. These principles establish practices that lend to developing software with considerations for maintaining and extending as the project grows. Adopting these practices can also contribute to avoiding code smells, refactoring code, and Agile or Adaptive software development.

SOLID Principles:-

The following five concepts make up our SOLID principles:

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

These five software development principles are guidelines to follow when building software so that it is easier to scale and maintain. They were made popular by a software engineer, Robert C. Martin.

**BENEFITS OF SOLID PRINCIPLES-**

Some of the benefits SOLID Principle holds are as follows:-

* Loose Coupling
* Code Maintainability
* Dependency Management

# **The SOLID Principles:**

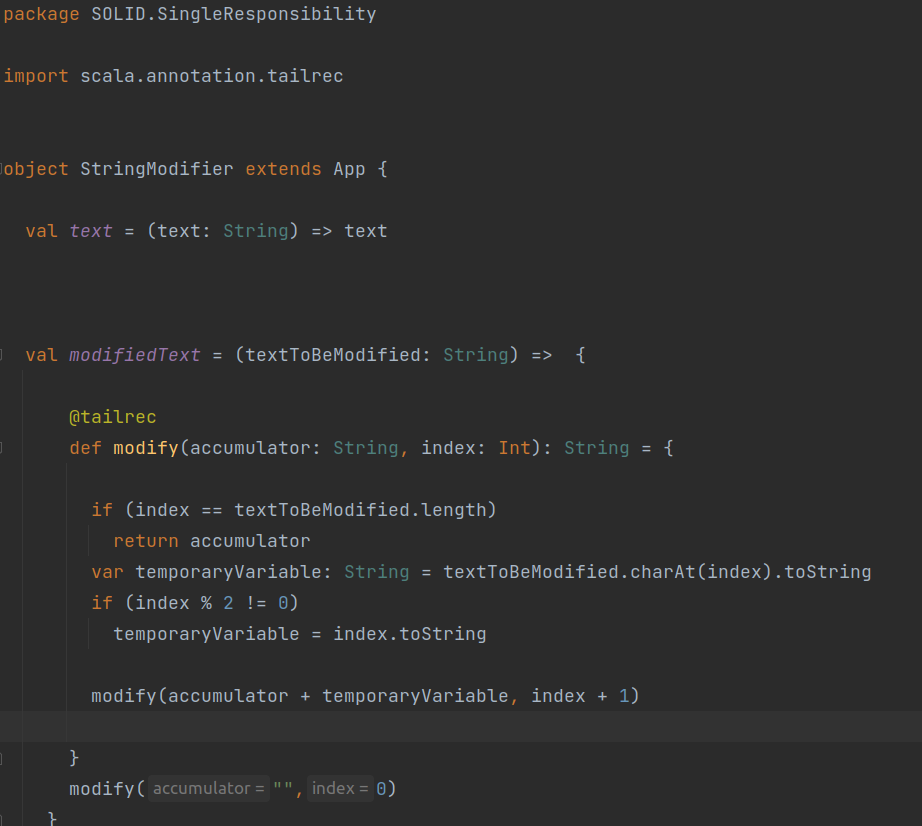
## **1.Single Responsibility:**

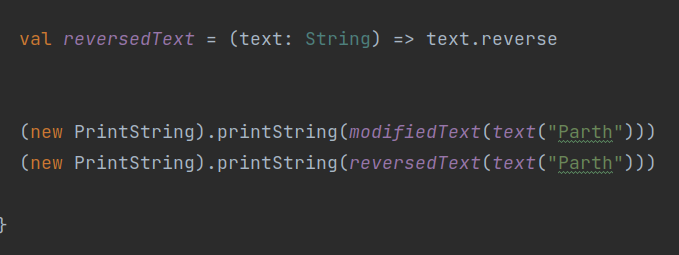
"An object should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the object."

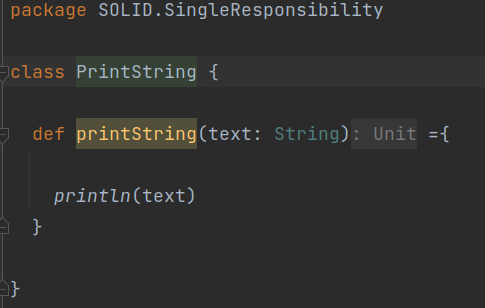
Traditionally when people talk about this principle they think about classes (although the original idea comes from UNIX development), they think about extracting behavior into multiple classes and handling a proper separation of concerns.  
Although functional programming languages don't have classes the same principle holds true. Functions should be small reusable pieces of code that you can compose freely to create complex behavior.

This can be extracted to almost anything, once your functions are small, the modules where they are located they should also form a cohesive closure that does only one thing and does it well.

As long as your function or class or module has only one reason to change then you are applying this principle.

****

****

****

**2.Open–Closed Principle**

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

This principle is usually instantly related to inheritance. A well-defined parent class that holds functionality and children of this class extend or reuse the mentioned functionality. In reality, it just means that we should be able to reuse and extend code without having to modify the original implementation.Instead of using inheritance, Functional Programming achieves this by using two tools. Composition to create new behaviors from previously defined functions and higher-order functions to change functionality at runtime, btw if you are interested in reading more about these topics you can check my series [Functional Programming for the object-oriented developer](https://dev.to/patferraggi/functional-programming-for-the-object-oriented-developer-part-0-1bgj).



****

**3.Liskov Substitution:**

"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.

This pattern is super common in functional programming, where you create functions that embrace polymorphic types (aka generics) to ensure that one set of inputs can seamlessly be substituted for another without any changes to the underlying code.



**Summary:-**

So far, we have discussed these three principles and highlighted their goals. They are to help you make your code easy to adjust, extend and test with little to no problems.

**SUBMITED BY-**

**PARTH**