FUNCTIONAL PROGRAMMING ASSIGNMENT

Presented By:- Shashikant Tanti

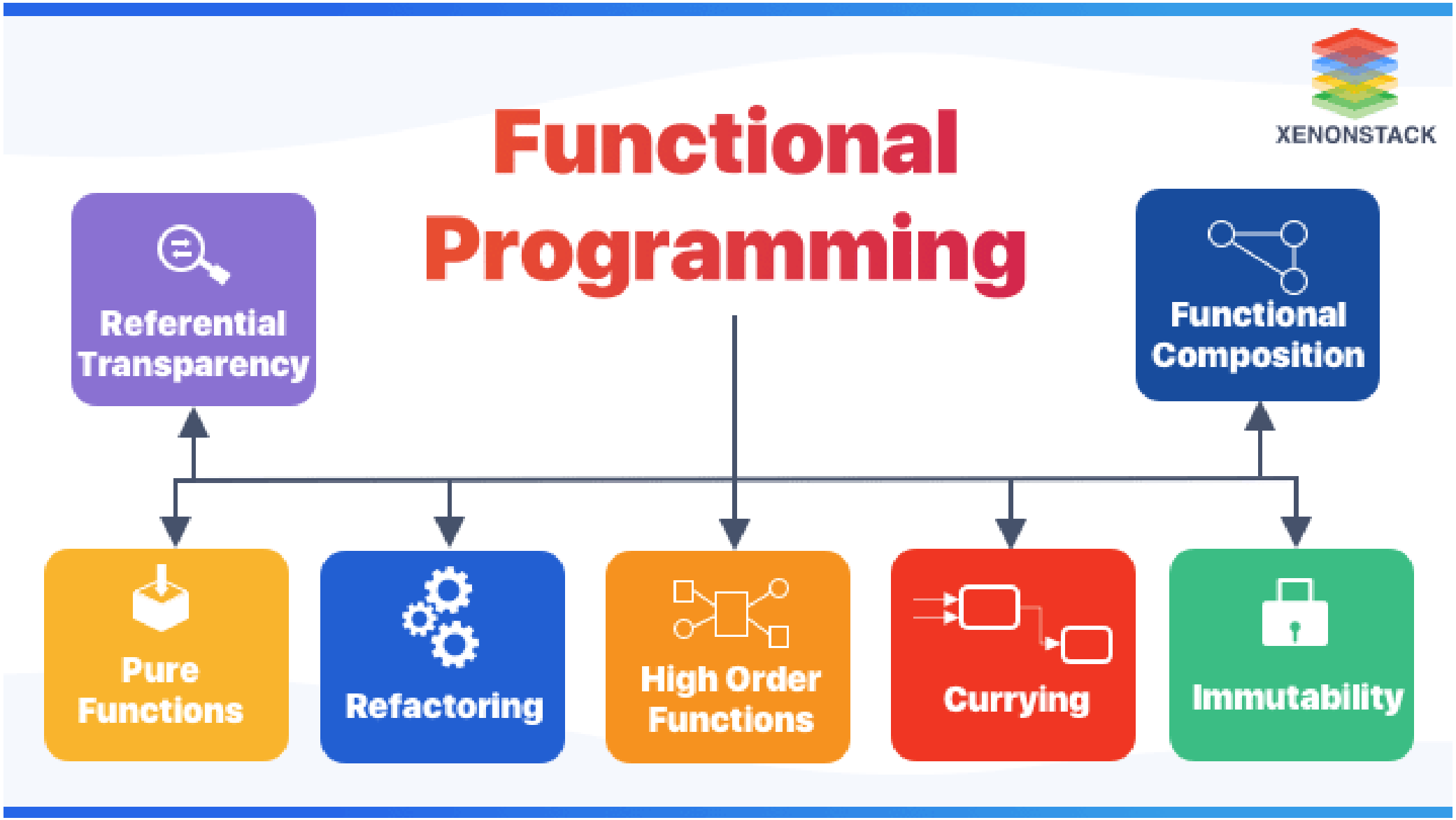
Agenda

- Functional Programming
- Advantages of FP
- Limitation of FP
- Basics of Scala
- Functional Aspect of Scala
- Conclusion

❖ Functional Programming:-

A programming style that models computation as the evaluation of expression.

Functional programming (FP) is a style of software development emphasizing functions that don't depend on program state. Functional code is easier to test and reuse, simpler to parallelize, and less prone to bugs than other code. Scala is an emerging JVM language that offers strong support for FP. Its familiar syntax and transparent interoperability with Java make Scala a great place to start learning FP.


XENONSTACK
Functional Programming
Referential Transparency
Functional Composition
Pure Functions
Refactoring
High Order Functions
Currying
Immutability

Functional programming (FP) is based on a simple premise with far-reaching implications: we construct our programs using only *pure functions*—in other words, functions that have no *side effects*. What are side effects? A function has a side effect if it does something other than simply return a result, for example:

- Modifying a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

1.1. The benefits of FP: a simple example

- Let’s look at an example that demonstrates some of the benefits of programming with pure functions. The point here is just to illustrate some basic ideas that we’ll return to throughout this book. This will also be your first exposure to Scala’s syntax. We’ll talk through Scala’s syntax much more in the next chapter, so don’t worry too much about following every detail. As long as you have a basic idea of what the code is doing, that’s what’s important.

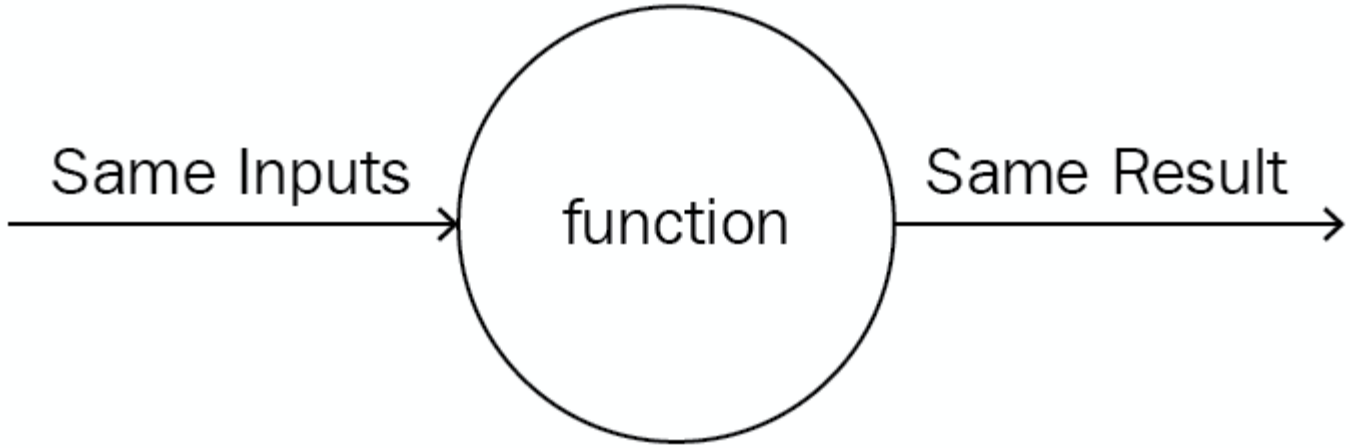
1.1.1. A program with side effects

Suppose we’re implementing a program to perform the switch at switchable things like bulb, fan,etc. We’ll begin with a Scala program that uses side effects in its implementation (also called an *impure* program).

```
package DIP

object ApplySwitch extends App {


  val electricSwitch = (client: Switchable) => {
    if (client.isOn) {
      println(client.name + " is off")
      client.isOn = false
    }
    else {
      println(client.name + " is On")
      client.isOn = true
    }
  }
  val bulbSwitch = new Switchable {
    override var isOn: Boolean = false
    override var name: String = "Bulb"
  }
  electricSwitch(bulbSwitch)
  electricSwitch(bulbSwitch)

  println()
  val FanSwitch: Fan = new Fan
  electricSwitch(FanSwitch)
  electricSwitch(FanSwitch)


}
```

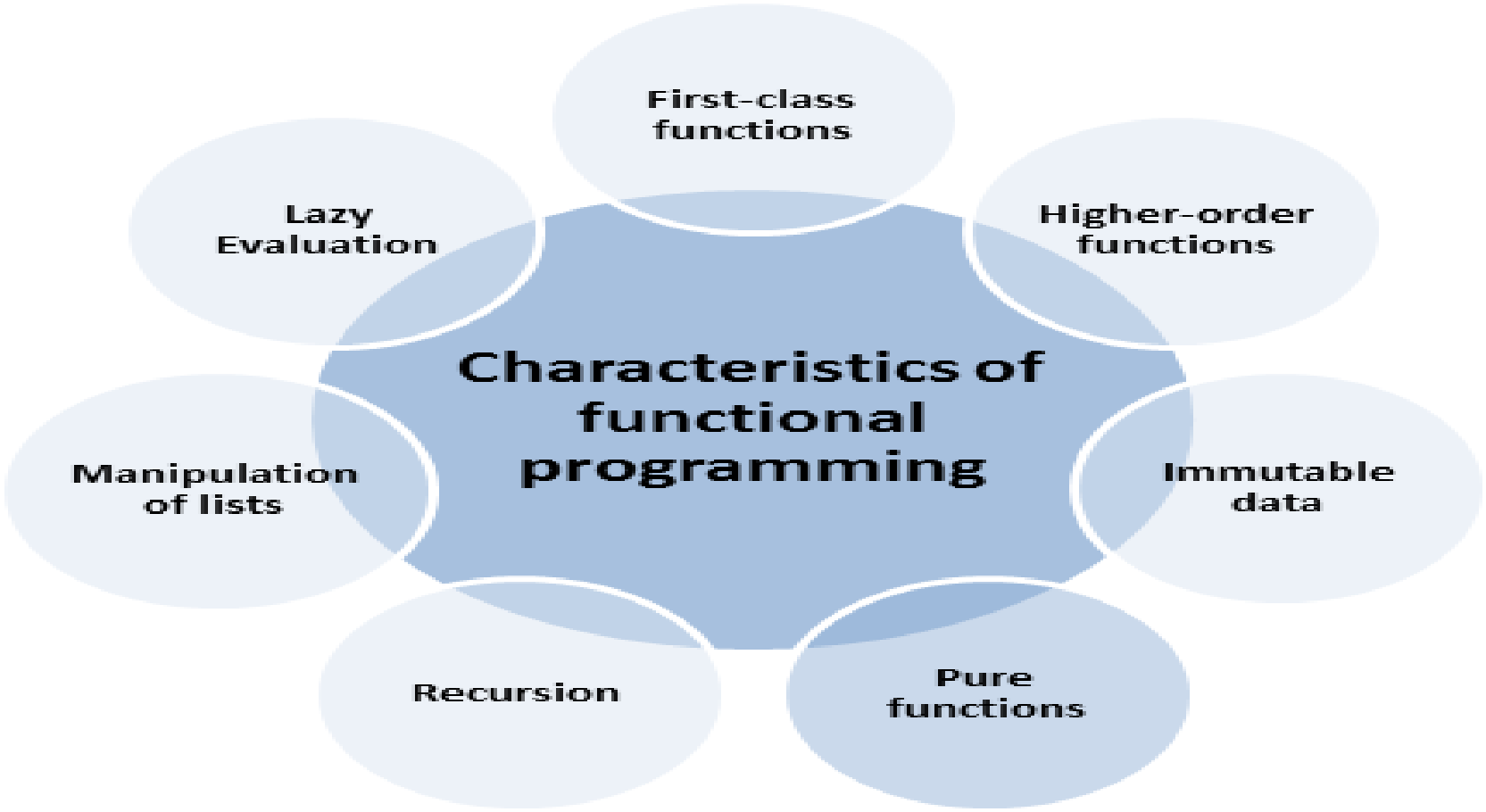
As a result of this side effect, the code is difficult to test. We don’t want our tests to actually to switch the bulb or fan! This lack of testability is suggesting a design change

Here, is taken another example of Pure Function for understanding about the Functional Programming:-



Scala: Pure function

Pure Functions produce no side effects and do not depend on global variables or states.



Imparative Programming:-

Imparative programming is a programming paradisgm that uses statements that change a program's state.

Example :-

- C #
- C++
- Java

Functional Programming:-

A programming Style that models computation as the evaluation of expression.

Example :-

- Haskell
- Erlang
- Lisp
- Clojure

History of Functional Programming

- Mathematical abstraction Lambda Calculus was the foundation for functional Programming
- Lambda calculus was developed in 1930s to be used for functional definition, functional application and recursion.
- It states that any computation can be achieved by sending the input to some black box, which produces its result.
- This black box is lambda functions, and in FP it's functions.
- LISP was the first functional programming language. It was defined by McCarthy in 1958

Imperative vs Functional Programming

Characteristics	Imperative	Functional
State changes	Important	Non-existent
Order of execution	Important	Low importance
Primary flow control	Loops, conditions and method (function) calls	Function calls
Primary manipulation unit	Instances of structures or classes	Functions

Characteristics of FP

- Immutable data
- Lazy evaluation
- No side effects
- Referential transparency
- Functions are first class citizens

Both Functional programming and object-oriented programming uses a different method for storing and manipulating the data. In functional programming, data cannot be stored in objects and it can only be transformed by creating functions. In object-oriented programming, data is stored in objects.

Immutable data

Mutable	Immutable

var collection = [1,2,3,4]
 	
for (i = 0; i<collection.length; i++) {
collection[i]+=1;
 	
}
 	
Uses loop for updating
 	
collection is updated in place
	val collection = [1,2,3,4]
	
	val newCollection = collection.map(value => value +1)
	
	Uses transformation for change
	
	Creates a new copy of collection. Leaves
	
	collection intact

❑ Unbeatable Advantages of Functional Programming:

1. Use of Pure Functions. Pure functions always produce the same output and have no external values affecting the end result. ...
2. Optimum Transparency.
3. Lazy Evaluation.
4. Enhanced Readability. .
5. Static Variables.
6. Seamless Parallel Programming.
7. Validating Functional Signatures.
8. Conclusion.

Drawbacks/Downsides of Functional Programming

- Recursion
- Input/output (IO)
- Terminology problems
- The non-functionality of computers
- The difficulty of stateful programming

Four Strengths of Functional Programming

1. **Abstraction is powerful**
2. **It's inherently parallel**
3. **It's easily testable/debuggable**
4. **Development is faster**

Conclusion:

When all is said and done, I think the strengths of functional programming outweigh the weaknesses. Functional programming makes a lot of sense for much of general-purpose programming, but you do need to be careful about its limitations in the areas of state management and input/output.

❑ History of Scala

Scala is a general purpose programming language. It was created and developed by Martin Odersky. Martin started working on Scala in 2001 at the Ecole Polytechnique Federale de Lausanne (EPFL). It was officially released on January 20, 2004.

Scala is not an extension of Java, but it is completely interoperable with it. While compilation, Scala file translates to Java bytecode and runs on JVM (Java Virtual machine).

Scala was designed to be both object-oriented and functional. It is a pure object-oriented language in the sense that every value is an object and functional language in the sense that every function is a value. The name of scala is derived from word scalable which means it can grow with the demand of users.

Scala History

- Martin Odersky is the creator
- Curious about compilers from beginning
- Created first compiler in 1995 - **Pizza**
  - Generics,Function pointers, Case class and pattern matching
- Led to ***javac*** compiler and **generics** eventually
  - Java is a strict language and hard to implement these new ideas

Why Scala?

There are many benefits to using Scala, and Scala 3 in particular. It's hard to list every benefit of Scala, but a "Top Ten" list might look like this:

1. Scala embraces a fusion of functional programming (FP) and object-oriented programming (OOP)

2. Scala is statically typed, but often feels like a dynamically typed language

3. Scala's syntax is concise, but still readable; it's often referred to as *expressive*

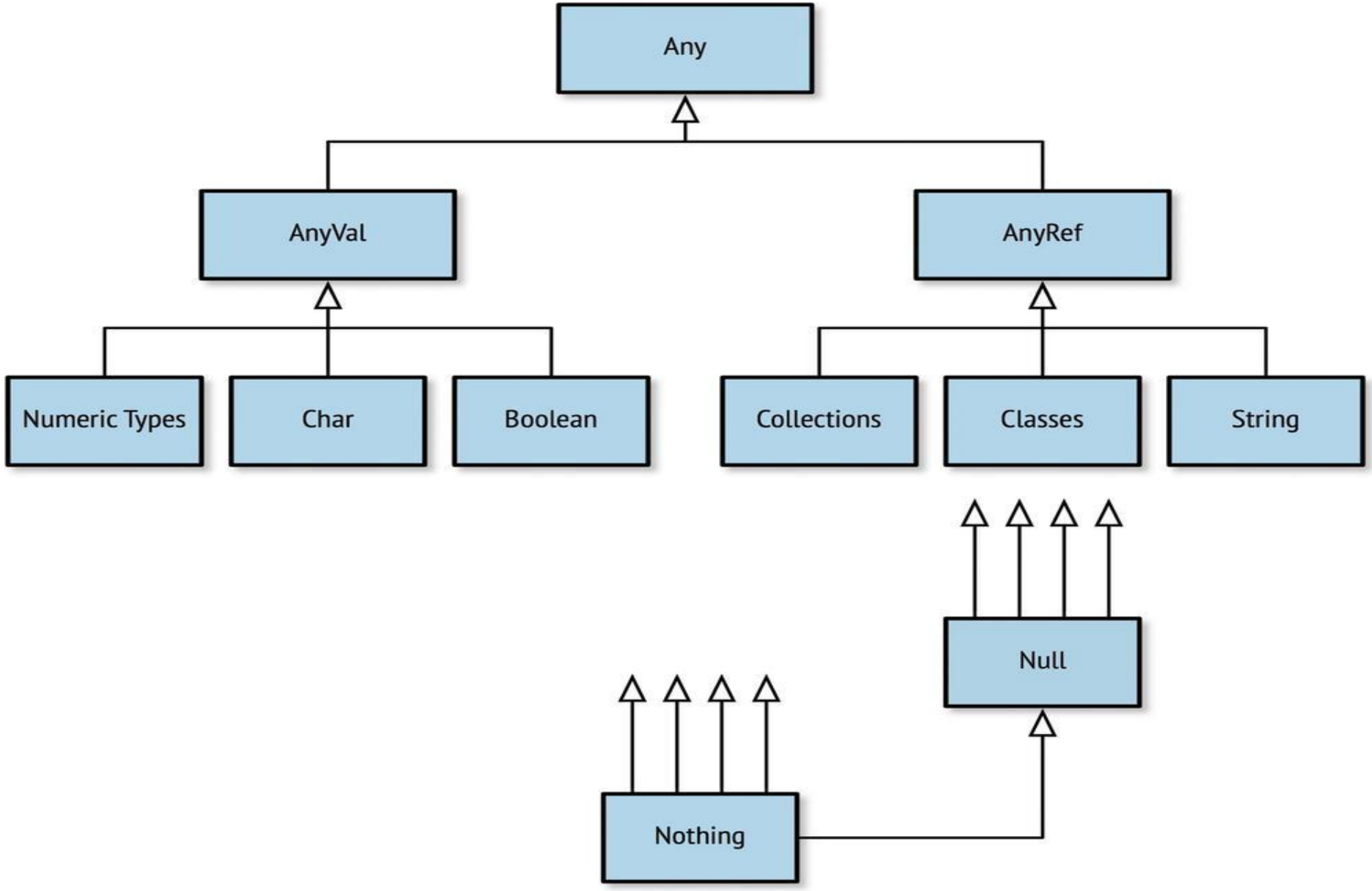
4. *Implicits* in Scala 2 were a defining feature, and they have been improved and simplified in Scala 3

5. Scala integrates seamlessly with Java, so you can create projects with mixed Scala and Java code, and Scala code easily uses the thousands of existing Java libraries

6. Scala can be used on the server, and also in the browser with Scala.js

7. The Scala ecosystem offers the most modern FP libraries in the world

8. Strong type system


Any
AnyVal
AnyRef
Numeric Types
Char
Boolean
Collections
Classes
String
Null
Nothing

1. Single-Responsibility Principle

Single-responsibility Principle (SRP) states:

A class should have one and only one reason to change, meaning that a class should have only one job

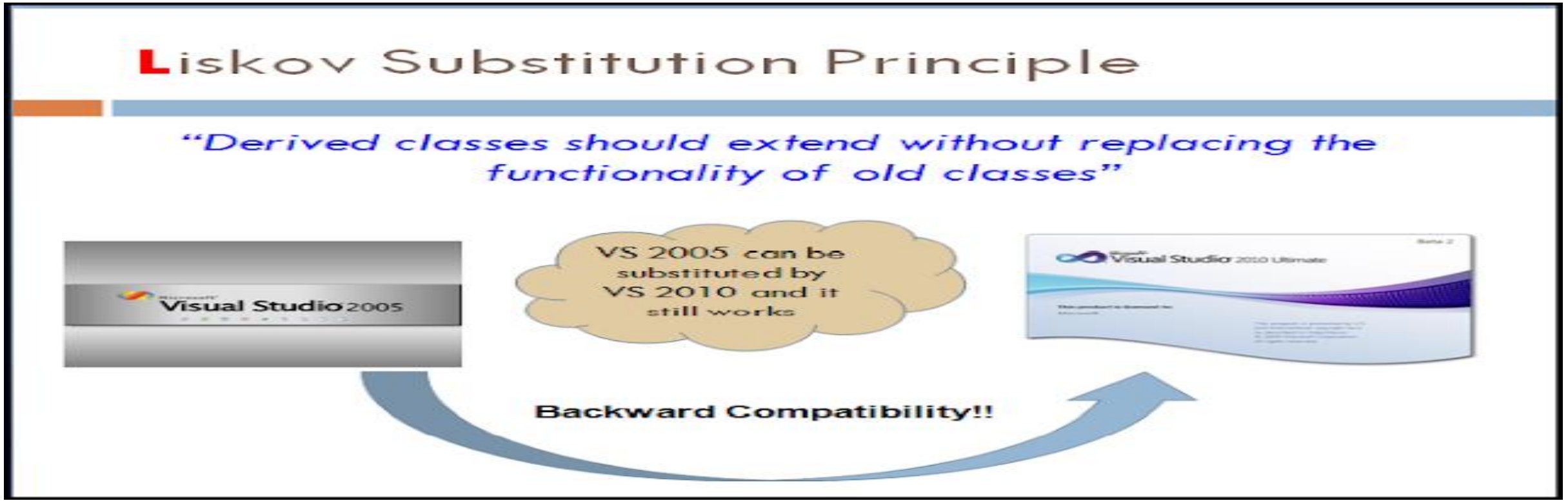
The single-responsibility principle (SRP) is a computer-programming principle that states that **every module, class or function in a computer program should have responsibility over a single part of that program's functionality**, and it should encapsulate that part. ... Hence, each module should be responsible for each role.

```scala
package SRP
import scala.annotation.tailrec

object StringModify extends App {
  val text = (text: String) => text

  /**.
   * modification is used for text modifier or reversing the string
   */

  val modified_Text = (text_To_BeModified: String) =>  {

    @tailrec
    def modify(accumulator: String, index: Int): String = {

      if (index == text_To_BeModified.length)
        return accumulator
      var temp_Var: String = text_To_BeModified.charAt(index).toString
      if (index % 2 != 0)
        temp_Var = index.toString

      modify(accumulator + temp_Var, index + 1)
    }
    modify( accumulator = "", index = 0)
  }
  /**
   * Result as a output of reverse text and modified text.....
   */

  val reverse= (text: String) => text.reverse

  (new StringPrint).StringPrint(reverse(text("Kites ")))
  (new StringPrint).StringPrint(modified_Text(text("Shashikant Tanti")))
}
```

2.The Liskov Substitution Principle:-

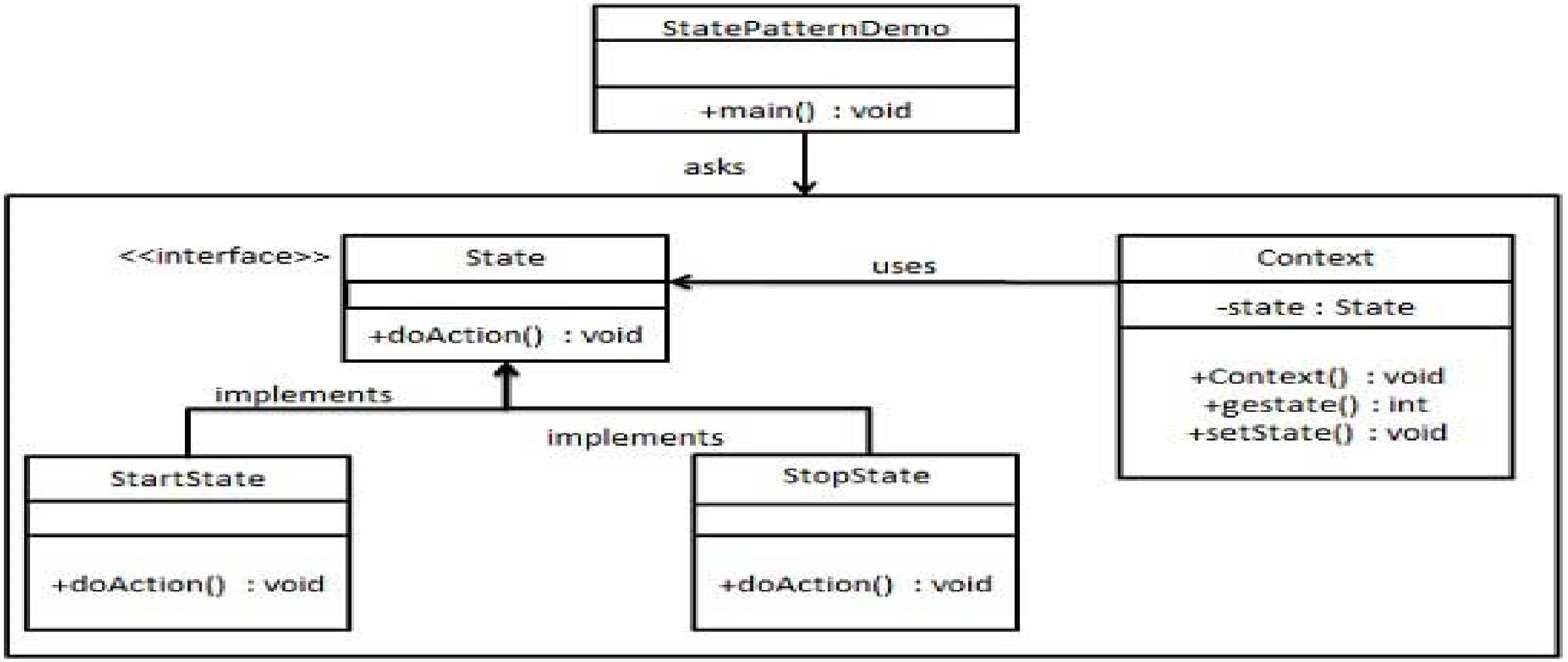
The principle says that subtypes must be substitutable for their base types.:-

The principle defines that **objects of a superclass shall be replaceable with objects of its subclasses without breaking the application**. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.



```scala
package LSP

object Test {

  val withdraw: (Account, Double) => String = (account: Account, amount:Double)=>{
    account.amount = account.amount - amount
    "ALERT: You've withdrawn Rs. " + amount + " Available Bal Rs.  " + account.amount
  }


  val deposit: (Account, Double) => String = (account: Account, amount:Double)=>{
    account.amount = account.amount + amount
    "UPDATE: Rs. " + amount + " has been deposited into your account. Avl Bal INR " + account.amount
  }


  def main(args: Array[String]): Unit = {

    val savingAccount = new AccountSaving( name = "Shashikant Tanti",  amount1 = 4500.45)
    val currentAccount = new AccountCurrent( name = "Sankata Jee",  amount1 = 8000.66)

    // The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

    Printer(withdraw(savingAccount, 1000)).printMessage()
    Printer(withdraw(currentAccount, 500)).printMessage()

    Printer(deposit(savingAccount, 100)).printMessage()


  }
}
```

State Transition Flow Diagram on Liskov Substitution Principle:



3.Dependency Inversion Principle:

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.
Based on this idea, Robert C. Martin's definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

An important detail of this definition is, that high-level **and** low-level modules depend on the abstraction. The design principle does not just change the direction of the dependency, as you might have expected when you read its name for the first time. It splits the dependency between the high-level and low-level modules by introducing an abstraction between them. So in the end, you get two dependencies:

1.the high-level module depends on the abstraction, and
2.the low-level depends on the same abstraction.

Lets take an example of Dependency Inversion Principle where there is a car class having different functionality like – DieselEngine, PetrolEngine which having Data inversion of classes:-

```scala
package DIP
object ApplySwitch extends App {

  val electricSwitch = (client: Switchable) => {

    if (client.isOn) {
      println(client.name + " is off")
      client.isOn = false
    }
    else {
      println(client.name + " is On")
      client.isOn = true
    }
  }
  val bulbSwitch = new Switchable {
    override var isOn: Boolean = false
    override var name: String = "Bulb"
  }

  electricSwitch(bulbSwitch)
  electricSwitch(bulbSwitch)

  println()
  val FanSwitch: Fan = new Fan
  electricSwitch(FanSwitch)
  electricSwitch(FanSwitch)

}
```

6. Conclusion

In this paper, we've argued that modularity is the key to successful programming. Languages that aim to improve productivity must support modular programming well. But new scope rules and mechanisms for separate compilation are not enough — modularity means more than modules. Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To support modular programming, a language must provide good glue. Functional programming languages provide two new kinds of glue — higher-order functions and lazy evaluation. Using these glues one can modularize programs in new and useful ways, and we've shown several examples of this. Smaller and more general modules can be reused more widely, easing subsequent programming.

This explains why functional programs are so much smaller and easier to write than conventional ones. It also provides a target for functional programmers to aim at. If any part of a program is messy or complicated, the programmer should attempt to modularize it and to generalize the parts. He or she should expect to use higher-order functions and lazy evaluation as the tools for doing this.

Of course, we are not the first to point out the power and elegance of higher order functions and lazy evaluation. For example, Turner shows how both can be used to great advantage in a program for generating chemical structures.

Refrences :

https://www.coursera.org/learn/scala-functional-programming

http://www.dipmat.unict.it/~barba/PROG-LANG/PROGRAMMI-TESTI/READING-MATERIAL/ShortIntroFPprog-lang.htm

Functional Programming in Scala:-

https://www.slideshare.net/DamianJureczko/functional-programming-in-scala-68115326?qid=176c0161-b55d-4b81-acf8-0291274986e6&v=&b=&from_search=5

Thank You!