Code and Blog

Functional Programming

ISSUED BY

Knoldus Inc.

REPRESENTATIVE

Shubham Shrivastava

shubham.shrivastava@knoldus.com

+91-9450067694

Functional Programming

Functional programming (also called FP) is a way of thinking about software construction by creating pure functions. It avoids concepts of shared state, mutable data observed in Object-Oriented Programming.

Functional languages emphasize expressions and declarations rather than the execution of statements. Therefore, unlike other procedures which depend on a local or global state, value output in FP depends only on the arguments passed to the function.

Characteristics of Functional Programming

- The functional programming method focuses on results, not the process
- Emphasis is on what is to be computed
- Data is immutable
- Functional programming Decompose the problem into ‘functions
- It is built on the concept of mathematical functions which uses conditional expressions and recursion to do perform the calculation
- It does not support iteration like loop statements and conditional statements like If-Else

History of Functional programming

1. The foundation for Functional Programming is Lambda Calculus. It was developed in the 1930s for the functional application, definition, and recursion
2. LISP was the first functional programming language. McCarthy designed it in 1960
3. In the late 70's researchers at the University of Edinburgh defined the ML(Meta Language)
4. In the early 80's Hope language adds algebraic data types for recursion and equational reasoning
5. In the year 2004 Innovation of Functional language 'Scala.'

Functional Programming Languages

The objective of any FP language is to mimic the mathematical functions. However, the basic process of computation is different in functional programming.

Here, are some most prominent Functional programming languages:

1. Haskell
2. SML
3. Clojure
4. Scala
5. Erlang
6. Clean
7. ML/OCaml Lisp / Scheme
8..SQL

Functional Programming Design Patterns

In object-oriented development, we are all familiar with design patterns such as the Strategy pattern and Decorator pattern, and design principles such as SOLID.

SOLID is a **mnemonic device for 5 design principles** of object-oriented programs (OOP) that result in readable, adaptable, and scalable code. SOLID can be applied to any OOP program.

There are 5 principles of the SOLID principle.

1. **S**ingle-responsibility principle
2. **O**pen-closed principle
3. **L**iskov substitution principle
4. **I**nterface segregation principle
5. **D**ependency inversion principle

The SOLID word comes from all 5 principles first word's first letter.

SOLID principles were Introduced and developed by computer science instructor and author Robert C. Martin in 2000 and quickly became a staple of modern object-oriented design (OOD). The SOLID acronym became commonplace when these principles gained widespread popularity in the programming world.

1. Single-responsibility principle

The **single-responsibility principle** (SRP) states that each class, module, or function in your program should only do one job. In other words, each should have full responsibility for a single functionality of the program. The class should contain only variables and methods relevant to its functionality.

Many classes can work together to complete the task but every class can execute the function from starting to end and after executing the function class passes the output to another function.

"A class 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 class."

2. Open-closed principle

The idea of the open-closed principle is that existing, well-tested classes will need to be modified when something needs to be added. Yet, changing classes can lead to problems or bugs. Instead of changing the class, you simply want to extend it. With that goal in mind, Martin summarizes this principle, “You should be able to extend a class’s behaviour without modifying it.”

Following this principle is essential for writing code that is easy to maintain and revise. Your class complies with this principle if it is:

1. Open for extension, meaning that the class’s behaviour can be extended; and
2. Closed for modification, meaning that the source code is set and cannot be changed.

At first glance, these two criteria seem to be inherently contradictory, but when you become more comfortable with it, you’ll see that it’s not as complicated as it seems. The way to comply with these principles and to make sure that your class is easily extendable without having to modify the code is through the use of abstractions. Using inheritance or interfaces that allow polymorphic substitutions is a common way to comply with this principle. Regardless of the method used, it’s important to follow this principle in order to write code that is maintainable and revisable.

“Software entities … should be open for extension, but closed for modification.”

3. Liskov substitution principle

The **Liskov substitution principle** (LSP) is a specific definition of a subtyping relation created by Barbara Liskov and Jeannette Wing. The principle says that any class must be directly replaceable by any of its subclasses without error.

In other words, each subclass must maintain all behaviour from the base class along with any new behaviours unique to the subclass. The child class must be able to process all the same requests and complete all the same tasks as its parent class.

In practice, programmers tend to develop classes based on behaviour and grow behavioural capabilities as the class becomes more specific. The advantage of LSP is that it speeds up the development of new subclasses as all subclasses of the same type share a consistent use.

You can trust that all newly created subclasses will work with the existing code. If you decide that you need a new subclass, you can create it without reworking the existing code.

4. Interface segregation principle

The **interface segregation principle** (ISP) requires that classes only be able to perform behaviours that are useful to achieve their end functionality. In other words, classes do not include behaviours they do not use.

This relates to our first SOLID principle in that together these two principles strip a class of all variables, methods, or behaviours that do not directly contribute to their role. Methods must contribute to the end goal in their entirety.

The advantage of ISP is that it splits large methods into smaller, more specific methods. This makes the program easier to debug for three reasons:

1. There is less code carried between classes. Less code means fewer bugs.
2. A single method is responsible for a smaller variety of behaviours. If there is a problem with a behaviour, you only need to look over the smaller methods.
3. If a general method with multiple behaviours is passed to a class that doesn't support all behaviours (such as calling for a property that the class doesn't have), there will be a bug if the class tries to use the unsupported behaviour.

"Many client-specific interfaces are better than one general-purpose interface."

5. Dependency inversion principle

The **dependency inversion principle** (DIP) has two parts:

1. High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (interfaces)
2. Abstractions should not depend on details. Details (like concrete implementations) should depend on abstractions.

The first part of this principle **reverses traditional OOP software design**. Without DIP, programmers often construct programs to have high-level (less detail, more abstract) components explicitly connected with low-level (specific) components to complete tasks.

DIP decouples high and low-level components and instead connects both to abstractions. High and low-level components can still benefit from each other, but a change in one should not directly break the other.

The advantage of this part of DIP is that decoupled programs require less work to change. Webs of dependencies across your program mean that a single change can affect many separate parts.

The second part can be thought of as “the abstraction is not affected if the details are changed”. The abstraction is the user-facing part of the program.

The details are the specific behind-the-scenes implementations that cause program behaviour visible to the user. In a DIP program, we could fully overhaul the behind-the-scenes implementation of how the program achieves its behaviour without the user’s knowledge.

This means you won’t have to hard-code the interface to work solely with the current details (implementation). This keeps our code loosely coupled and allows us the flexibility to refactor our implementations later.

“One should depend upon abstractions, [not] concretions.”