Functional Programming

September 2021

REPRESENTATIVE

UJJAWAL SINGH

[ujjawal.singh@knoldus.com](mailto:ujjawal.singh@knoldus.com)

+917017762293

Functional Programming

Functional programming is a declarative programming paradigm where programs are created by applying **sequential functions rather than statements**.

Each function takes in an input value and returns a consistent output value without altering or being affected by the program state.

These functions complete a single operation and can be composed in sequence to complete complex operations. The functional paradigm results in highly modular code, since functions can be reused across the program and can be called, passed as parameters, or returned.

Advantages of Functional programming

* **Easy debugging**: Pure functions and immutable data make it easy to find where variable values are set. Pure functions have fewer factors influencing them and therefore allow you to find the bugged section easier.
* **Lazy evaluation**: Functional programs only evaluate computations at the moment they’re needed. This allows the program to reuse results from previous computations and save runtime.
* **Modular**: Pure functions do not rely on external variables or states to function, meaning they’re easily reused across the program. Also, functions will only complete a single operation or computation to ensure you can reuse that function without accidentally importing extra code.
* **Enhanced readability**: Functional programs are easy to read because the behavior of each function is immutable and isolated from the program’s state. As a result, you can predict what each function will do often just by the name!
* **Parallel programming**: It’s easier to create parallel programs with a functional programming approach because immutable variables reduce the amount of change within the program. Each function only has to deal with user input and can trust that the program state will remain mostly the same!

Functional programming languages:

* **Haskell**: This is the clear favorite language for functional programming. It’s memory safe, has excellent garbage collection, and fast due to early machine code compiling. Haskell’s rich and static typing system gives you access to unique algebraic and polymorphic types that make functional programming more efficient and easier to read.
* **Erlang**: This language and descendent, Elixir, have established a niche as the best functional language for concurrent systems. While not as popular or widely usable as Haskell, it’s often used for backend programming. Erlang has more recently gained traction for scalable messaging apps like Whatsapp and Discord.
* **Clojure**: This language is a functional-first dialect of Lisp used on the Java virtual machine (JVM). It’s a predominantly functional language that supports both mutable and immutable data structures but is less strictly functional than others here. If you like Lisp, you’ll like Clojure.
* **F#**: F# is similar to Haskell (they’re in the same language group), but has less advanced features. It also has minor support for object-oriented constructions.

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.

The functional programming community has design patterns and principles as well.

SOLID principles

The 5 principles of SOLID are:

* **S**ingle-responsibility principle
* **O**pen-closed principle
* **L**iskov substitution principle
* **I**nterface segregation principle
* **D**ependency inversion principle

S: 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.

Classes can work together to complete larger complex tasks, but each class must complete a function from start to finish before it passes the output to another class.

Martin explained this by saying “a class should have only one reason to change”. Here the “reason” is that we want to change the single functionality this class pursues. If we do not want this single functionality to change, we will never change this class because all components of the class should relate to that behavior.

O: Open-closed principle

The **open-closed principle** (OCP) calls for entities that can be widely adapted but also remain unchanged. This leads us to create duplicate entities with specialized behavior through polymorphism.

Through polymorphism, we can extend our parent entity to suit the needs of the child entity while leaving the parent intact.

Our parent entity will serve as an abstract base class that can be reused with added specializations through inheritance. However, the original entity is locked to allow the program to be both open and closed.

The advantage of OCP is that it minimizes program risk when you add new uses for an entity. Instead of reworking the base class to fit a work-in-progress feature, you create a derived class separate from the classes currently present throughout the program.

We can then work on this unique derived class, confident that any changes we make to it will not affect the parent or any other derived class.

L: 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 behavior from the base class along with any new behaviors 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 behavior and grow behavioral 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.

I: Interface segregation principle

The **interface segregation principle** (ISP) requires that classes only be able to perform behaviors that are useful to achieve its end functionality. In other words, classes do not include behaviors 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 behaviors that do not directly contribute to their role. Methods must contribute to the end goal in their entirety.

D: 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.