### **Single-Responsibility Principle**

"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.

### **Open–Closed Principle**

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

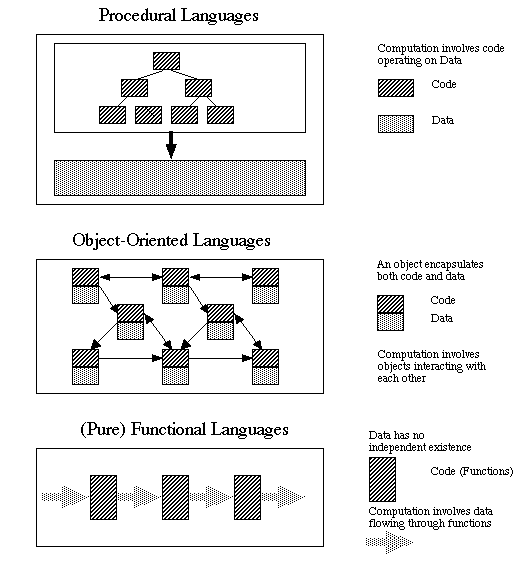
This principle is usually instantly related to inheritance. A well-defined parent class that holds functionality and children of this class extend or reuse the mentioned functionality. In reality, it just means that we should be able to reuse and extend code without having to modify the original implementation.

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.

### **LISP Belongs to the Functional Language Paradigm**

LISP is the earliest representative of the functional programming language paradigm. Unlike procedural and object-oriented languages - whose theoretical model of computation is the Turing Machine, LISP's theoretical model of computation is the Lambda calculus developed by Alonzo Church. (It can be shown that the two models of computation are equivalent in power - that is, any algorithm that can be expressed in one model can also be expressed in the other).

This difference might be understood in terms of the following diagram:



In procedural languages, code operates on data; in object-oriented languages, objects encapsulate code and data and interact with one another; in functional languages data flows through functions but has no separate existence of its own. That is, in procedural languages data is passive; in OO languages it is active; in pure functional languages it is ephemeral.

However, most functional languages (including LISP) provide a mechanism for keeping certain data in existence even when it is not flowing through functions. In LISP, this takes the form of what Common Lisp calls "special variables" - essentially equivalent to what in other programming languages are called "global variables". That is to say, though LISP is a functional language, it is not a pure functional language (though it could be used as one by avoiding imperative constructs.)