5 Lessons That Golang Teaches To All Programmers

You may not code in Go, but these lessons are valid for your favorite programming language

Shalitha Suranga Follow

Nov 12 · 5 min read ★

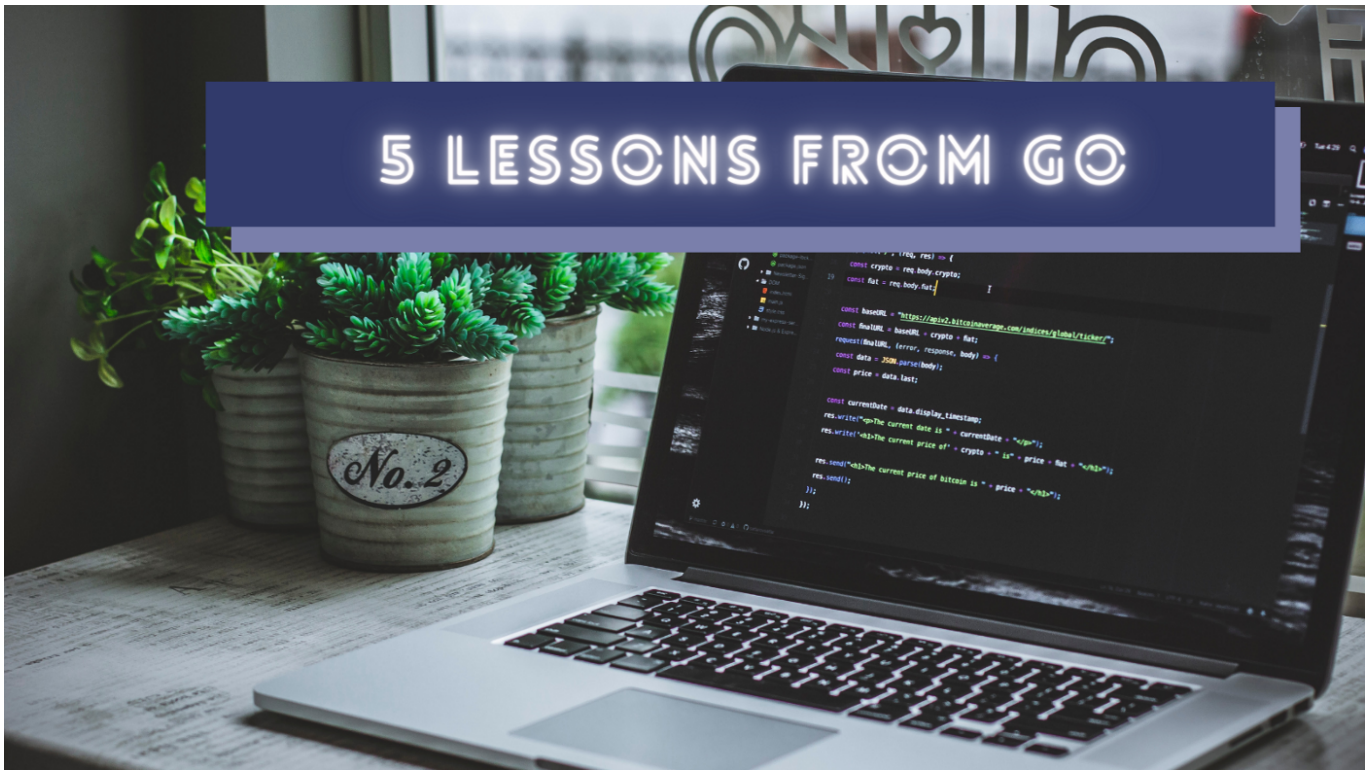

Photo by Safar Safarov on Unsplash, edited with Canva

Go (also known as Golang) is a popular programming language for building developer tooling, cloud computing projects, CLI programs, desktop applications, and web

application backends. Go gains more popularity every day due to its minimal syntax, performance, powerful concurrency support, and consistent language design. Go won't replace your favorite programming language, but undoubtedly Go is an overall good language compared to other popular programming languages.

Golang has just 25 keywords and supports object-oriented-styled programming without even having `class` as a reserved keyword. Nowadays, programmers tend to use Go in scenarios where performance and concurrency play a key role.

Like the C programming language, Go isn't a must-learn type language — but Go's design principles teach valuable lessons for all programmers. In this story, I will explain some valuable lessons from the Go programming language. These facts will ask you to rethink the way you solve software engineering problems today.

Write Code Without Overengineering

It's possible to write any software program inside one function. But, we typically split up our software programs' codebase into multiple components(modules, classes, or functions) to achieve better maintainability factors. The way you decompose your codebase defines the complexity of the codebase. The codebase may look complex and less maintainable when you create a few components. On the other hand, the codebase becomes complex again when you try to decompose code into many overengineered small components.

Good programmers tend to minimize the project's complexity when they write code by managing a minimal number of modules.

Golang motivates us to prevent overengineering via its minimal design. Go has a few reserved keywords — but offers all features you need from a modern language. Go typically doesn't provide multiple ways to do the same thing. For example, Golang doesn't offer ternary operators and traditional while loop syntax because it already has if-else and for control flows. Go defines a minimal syntax without using more characters than the language need. See the following while loop.

```
for x < 10 {
  sum += x
}
```

Golang motivates all programmers to write clean code without overengineering. The following story guides you further about clean code practices:

Stop Extending the Software Cores with Features

Even though your software system doesn't follow the plugins-based architecture, you may have a module that contains your software system's main logic. The software system core refers to the main code section that drives the entire software system. For example, a web application's core module consists of database connectivity, server code, authentication flow, and core business logic.

If programmers don't modify the software system's core frequently, the whole software system becomes stable. On the other hand, the entire software system becomes more error-prone, unstable, and slow if programmers frequently modify the core with features. Most software development project teams typically lockdown (or minimize) core modifications due to this reason.

Golang teaches us this lesson via its standard API design and built-in features. Go's reference core spec is fixed, but the development team keeps extending the standard library with features and improvements. This approach indeed helped Golang to become more stable as a programming language within a short time.

Exception-Based Error Handling Is For Exceptional Cases

Programmers use various strategies to handle errors in their software systems. Programmers who work with C/C++ languages often tend to use error codes for error handling. C# and Java developers always use exception-based error handling strategies. Meanwhile, some programmers use exceptions only for critical error handling flows.

Almost all modern programming languages support exceptions with traditional try-catch-like syntaxes. Using exceptions is indeed okay, but exceptions make some unavoidable problems. Exceptions are for exceptional cases, but most programmers even try to avoid if-else validations with exceptions. Also, exceptions complicate the flow of the program because try-catch blocks create another logical flow to handle the error.

Golang doesn't offer traditional exceptions and try-catch blocks as a language feature. Instead, it lets programmers return the error as the second return value of functions. This approach allows programmers to handle the error within the same flow of the program without creating another logical flow. For example, check how we typically detect JSON parsing errors in Go.

```
err := json.Unmarshal([]byte(jsonString), &myStruct)

if err != nil {
    fmt.Println("JSON decode error!")
    return
}
```

Think twice before throwing an error. Check whether you can set a default value or pre-check input with an if condition.

Traditional OOP Is not the Best For Every Project

Nowadays, programmers often work with software projects that follow the Object-Oriented Programming (OOP) paradigm. The OOP paradigm offers us several principles to split large codebases into small and well-manageable components called classes. OOP is not the only programming paradigm to build software systems — we can use

procedural, functional, and logical paradigms too. We can indeed use OOP to build any software system by identifying several manageable objects. However, OOP comes with some disadvantages.

Every programmer typically starts coding with the procedural programming paradigm — remember how you wrote code to get a user input via the command line. Also, the procedural programming pattern is natural and easy to understand because it motivates to separate data models and logic. On the other hand, OOP motivates to write logic inside data models. Therefore, OOP principles can make simple projects complex and bloaty. Also, reusability patterns in OOP can lead to overengineering sometimes.

Even though Go offers ways to create classes with methods, it doesn't fully support all traditional OOP concepts. Golang doesn't support traditional classes and inheritance (there are some workarounds). Golang offers most features to write codes in the natural procedural style. Think about procedural and functional paradigms before drawing class diagrams for your next software project.

Consider All Possibilities Of Solving a Coding Problem

As programmers, we solve various software engineering problems by writing code. Some problems have straightforward solutions. But, most software engineering problems that we meet have no straightforward solutions — and we need to construct the optimal solution from a pool of candidate solutions. Some programmers tend to use easy shortcuts to solve a problem and create technical debt that needs time-consuming code refactoring.

Premature optimization is the root of all evil, but it's worth investing time to find the optimal solution from all possibilities because the undo process is hard in programming.

Golang solves programming language design problems in very different and impressive ways — that other programming languages didn't consider. See how Golang supports creating enumerations without even having `enum` as a reserved keyword.

```
const (
    Red int = iota
    Green
    Blue
)
```

Golang answered the parallel programming problem with Goroutines. Goroutines are managed lightweight threads and not native operating system-level threads. See how easy to turn a function into a thread compared to other programming languages.

```
go func() {
    fmt.Println("I'm async!")
}()
```

Mastering computer science knowledge is the key to solving computing problems better. The following story explains more about it:

Programming Technology Software Development Go Software Engineering