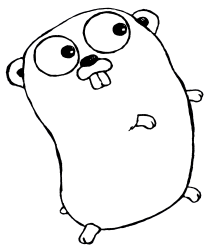
Characteristics of Go

This lesson discusses the aspects that make Go a successful language in the programming world.

WE'LL COVER THE FOLLOWING

Features of Go



Go is essentially an **imperative** (procedural, structural) language, built with **concurrency** in mind. It is **not truly object-oriented** like Java and C++ because it doesn't have the concepts of classes and inheritance. However, it does have a concepts of **interfaces**, with which much of the polymorphism can be implemented. Go has a clear and expressive type-system, but it is lightweight and without hierarchy. So in this respect, it could be called a **hybrid language**.

Some features of modern OOP languages were intentionally left out. Because, object orientation was too *heavy* often leading to cumbersome development constructing big type-hierarchies, and so not compliant with the speed goal of the language. As per the decision made by the Go-team, the following OOP features are missing from Golang. Although, some of them might still be

features are missing from Golang. Although, some of them might still be implemented in its future versions.

- To simplify the design, no *function* or *operator overloading* was added.
- *Implicit conversions* were excluded to avoid the many bugs and confusion arising from this in languages like C/C++.
- No *classes* and *type inheritance* is supported in Golang.
- Golang does not support *variant types*. However, almost the same functionality is realized through interfaces.
- *Dynamic code loading* and *dynamic libraries* are excluded.
- *Generics* are not included (but this is a possible feature for **Go 2.0**).
- *Exceptions* are not included (although recover/panic often goes in that direction).
- Assertions are not included.
- Immutable (unable to change) variables are excluded.

A discussion around these features by the Go-team itself can be found in the Go FAQ.

Golang is a **functional language**, meaning that functions are the basic building blocks, and their use is very versatile. There is more information on functions in Golang in Chapter 4.

Go is **statically typed**, making it a safe language that compiles to native code and has a very efficient execution. It is also strongly typed, which means according to the principle *keep things explicit*. Implicit type conversions (also called castings or coercions) are not allowed. An important thing to note is that Go does have some features of **dynamically typed** languages (using `var` keyword). That's why Go also appeals to programmers who left Java and the .Net world for Python, Ruby, PHP, and JavaScript.

Last but not least, Go has support for **cross-compilation**, for example, developing and compiling on a Linux-machine for an application that will execute on Windows. It is one of the first programming languages that can use *UTF-8* not only in strings but also in program code. Go is truly an **international language** because Go source-files are also in UTF-8.

Uses of Go

There are many uses of Go. Following are some main uses of this language:

- Used as a system programming language
- Used as a general programming language
- Used as an internal support

Used as a system programming language

Go was originally conceived as a systems programming language for the heavy server-centric (Google) world of web servers, storage architecture, and the like. For certain domains like high performance distributed systems, Go has already proven to be a more productive language than most others. Golang shines in and makes massive concurrency and event-processing easy. So it should be a good fit for the game server and IoT (Internet of Things) development.

Used as a general programming language

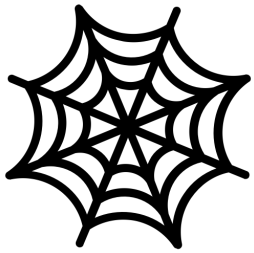
Go is also a general programming language, useful for solving text-processing problems, making frontends, or even scripting-like applications. However, Go is not suited for real-time software because of the garbage collection and automatic memory allocation.

Used as an internal support

Go is being used for some time internally in Google for heavy-duty distributed applications, e.g., parts of *Google Maps* run on Go.

Before moving any further, if you are interested in exploring some interesting real-life usage of Golang in other organizations around the world, click here. The web also brings you a lot of Go success stories.

Beware of the trap



If you come to Go and have a background in other contemporary (mostly class or inheritance-oriented languages like Java, C#, Objective C, Python, or Ruby), then beware! You can fall in the *trap* of trying to program in Go as you did in your previous language. However, Go is built on a different model. So if you decide to move code from

different model. So if you decide to move code from

language X to Golang, you will most likely produce non-idiomatic code that works poorly overall.

You have to start over by *thinking in Go*. If you take a higher point of view and start analyzing the problem from within the Go mindset, often a different approach suggests itself, which leads to an elegant and idiomatic Go solution.

Guiding design principles

Go tries to reduce typing, clutter, and complexity in coding through a minimal amount of keywords (25). This, together with the clean, regular, and concise syntax, enhances the compilation speed because the keywords can be parsed without a symbol table as its grammar is LALR(1).

These aspects reduce the number of code lines necessary, even when compared with a language like Java. Additionally, Go has a *minimalist approach*: there tends to be only one way of getting things done, so reading other people's code is generally pretty easy, and we all know the code's readability is of the utmost importance in software engineering.

The design concepts of the language are *orthogonal* because they don't stand in each other's way, and they don't add up complexity to one another.

Golang is completely defined by an explicit specification that can be found here; it is not defined by an implementation as is Ruby, for example. An explicit language specification was required for implementing the *two different compilers* **gc** and **gccgo**, and this in itself was a great help in clarifying the specification.

We have completed building the context of the Go language. A quiz awaits you in the next lesson.