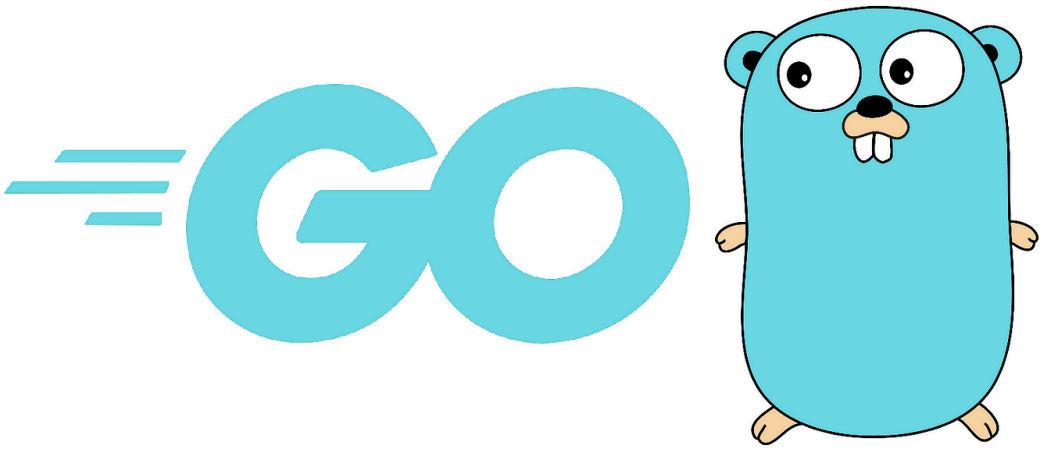

Go Programming Language
Introduction

Pedro Vinicius Messetti

30/08/2023

Introduction

Go, often referred to as Golang, is a programming language designed by Google. It was first released in 2009 and has gained popularity due to its simplicity, performance, and suitability for building scalable and efficient applications. I Basics features of Go:

- **Concurrent and Efficient:** Go was designed with concurrency and efficiency in mind. It includes features like goroutines (lightweight threads) and channels that make it easy to write concurrent programs without sacrificing performance, efficiently managing multiple tasks concurrently. This is particularly useful for building scalable and responsive applications.
- **Compiled Language:** Go is a compiled language, which means that the code is translated into machine code before execution. This contributes to its speed and efficiency.
- **Strongly Typed:** Go is statically typed, meaning that variable types are checked at compile-time, reducing the chances of type-related errors during runtime.
- **Minimalist Syntax:** Go's syntax is intentionally simple and clean, making it easy to read and write code. It has a small number of keywords and a concise standard library.
- **Garbage Collection:** Go includes automatic memory management through its garbage collection mechanism, which helps developers avoid memory leaks.
- **Package System:** Go has a well-defined package system that encourages modular and reusable code. This aids in building maintainable and organized applications.
- **Cross-Platform:** Go supports compilation for various platforms and architectures, allowing you to write code on one platform and compile it for another without significant changes.
- **Standard Library:** Go comes with a comprehensive standard library that provides essential functionality for tasks like networking, file I/O, and more. This helps to avoid relying on external libraries.
- **Open Source:** Go is an open-source language with an active community contributing to its development. This results in continuous improvement, bug fixes and the creation of useful tools and libraries.

Installation

For installing, update your packages and run:

```
sudo apt install golang
```

Running Hello World in Go is really straightforward. Create a file *(hello.go)* and use the following code:

```
package main

import "fmt"

func main() {

   fmt.Println("Hello, World!")

}
```

Compile and run using *go* command:

```
go run hello.go
```

Why Go?

Go has gained significant traction in recent years, offering a unique blend of features that make it a compelling choice for modern developers. The language strikes a balance between performance, simplicity, and efficiency, making it an attractive option for a wide range of applications. One of Go's standout features is its exceptional concurrency support through goroutines and channels. This facilitates efficient utilization of multicore processors and simplifies the development of concurrent programs, a crucial advantage for building highly responsive applications and services in today's era of parallel computing. His minimalistic syntax and standard library encourage clean and readable code, accelerating the development process and fostering collaboration among team members.

Furthermore, its strong emphasis on static typing and compile-time checks catches errors early, reducing runtime bugs and enhancing code reliability.

In a competitive job market, learning Go can be a strategic move for new developers. The language's rising popularity within industries such as web services, cloud computing, and microservices indicates a growing demand for skilled Go developers, making it an investment well worth considering.

- Multi-core Hardware Architecture
-

Types

Go is a statically typed language, which means that variable types are determined and checked at compile time, and they cannot change during runtime. Types determine the kind of values variables can hold, and they help catch errors at compile time while also enhancing code readability and maintainability.

Built-in Basic Types: Go provides several built-in basic types, including *int*, *float64*, *string*, *bool*, *byte* (alias for *uint8*), *int32*, and more. These basic types represent integers, floating-point numbers, strings, boolean values, and individual characters, respectively.

Type Inference: Go supports type inference, allowing the compiler to deduce the type of a variable from its initialization value. This reduces the need for explicit type declarations.

```
age := 25 // Inferred as int

name := "Alice" // Inferred as string
```

User-Defined Types: In addition to built-in types, you can define your own custom types using the *type* keyword. These types can be aliases to existing types or new struct and interface types you define. Type aliases are often used to provide more descriptive names for existing types or to improve code readability.

```
type Celsius float64 // Custom type alias

type Point struct {
```

```
    x, y int
}
```

Type Assertion: Type assertion is used with interfaces to extract a value from an interface into a specific type. It provides runtime checks to ensure the underlying type matches the asserted type.

```
var value interface{} = 42

result := value.(int) // Type assertion
```

Packages and Importing Types: Types defined in different packages can be used in your code by importing the relevant package. You can access exported types (those starting with an uppercase letter) from other packages.

```
import "package_name"

var p package_name.Point
```

Pointer Types: Go supports pointers, which are variables that store memory addresses of other variables. Pointers are used for more efficient memory management and for sharing data between different parts of your program.

Type Casting

Go has strict type compatibility rules. For example, you cannot directly assign a value of one type to a variable of another type without explicit conversion. This enhances type safety and reduces common programming errors.

```
var num int = 42
```

```
var f float64 = num // Compilation error
```

In Go, type conversion is essential when you want to assign a value of one type to a variable of a different type. To perform this action use the syntax *<Type>(expression)*, where *<Type>* is the target data type and *expression* is the value you want to convert.

```
// Convert int to float64
var x int = 42
y := float64(x)
fmt.Printf("x: %d, y: %f\n", x, y)

// Convert float64 to int
var a float64 = 3.14
b := int(a)
fmt.Printf("a: %f, b: %d\n", a, b)
```

Some considerations that is good to know about:

- **Data Loss:** Be cautious when converting between types, especially if the destination type has a smaller range or precision than the source type. Conversion can lead to data loss or unexpected results.
- **String Conversion:** Converting between strings and other types often requires explicit conversion functions. For example, you can use the *strconv* package to convert numbers to strings and vice versa.
- **Pointer Conversion:** When converting between different pointer types (e.g., **int* to **float64*), a type conversion is required, but careful when dealing with pointer conversions to avoid unexpected behavior.
- **Custom Type Conversion:** For user-defined types (structs, custom types, etc.), you can define custom conversion methods to convert between types.

Variables basics

In Go, you can declare variables using the *var* keyword followed by the variable name and type. Also you can initialize variables during declaration. For example:

```
var age int    // Declaration
var name string = "John" // Declaration with initialization
```

One of the many purpose of Go is to be simple and clean, so you can omit the type and the compiler can deduce it from the initialization value:

```
age := 25   // Declaration and initialization using type inference
```

The main advantage of the short variable declaration is its conciseness. It reduces repetitive typing and improves code readability by eliminating the need to repeat the variable's type. This reduces redundancy and makes the code more flexible to changes in the assigned value's type. Short variable declarations are typically used within a limited scope (like a function). This helps prevent accidentally reusing the same variable name in different parts of the code and promotes better code organization.

Variables have a scope, which is the region of the code where they are accessible. Go has block-level scope. Variables declared within a block (within curly braces) are only accessible within that block. Variables declared at the package level (outside functions) are accessible across multiple files within the same package.

Variables declared without an explicit initialization value are assigned a "zero value" based on their type. For example, the zero value for an integer is 0, and for a string, it's an empty string (""). You can also declare multiple variables in a single *var* statement:

```
var a, b, c int
```

Variables declared outside function scope must have *var* keyword

Constants

Named values that cannot be changed after they are assigned a value. They are used to represent fixed values in your code that should remain unchanged throughout the program's execution. declared using the *const* keyword followed by the constant's name, type (optional), and value.

```
const pi = 3.14159
const maxAttempts int = 5
```

Constants are often used to represent configuration values, mathematical constants, flags, and enumerations. They help make your code more readable, maintainable, and less error-prone by providing named values that cannot be accidentally changed.

```
const (
   monday  = 0
   tuesday   = 1
   wednesday = 2
   // ...
)
```

The *iota* identifier is used in constant declarations to create a sequence of related values, incrementing by one each time.

```
const (
   apple = iota // 0
   banana       // 1
   cherry       // 2
)
```

Functions

Functions can be defined in any package and can be used within the same package or in other packages by importing them. Declare a function using the *func* keyword followed by the function name, a parameter list in

parentheses, an return type (optional) and the function body in curly braces. Functions can take zero or more arguments. The return type indicates the type of value that the function will return. If a function doesn't return any value, the return type can be omitted. The following function example accepts two integers parameters and returns another integer:

```
func add(a, b int) int {
   return a + b
}
```

Go functions can return multiple values, particularly useful for returning both a result and an error, for example:

```
func divide(a, b float64) (float64, error) {
   if b == 0 {
            return 0, errors.New("division by zero")
   }
   return a / b, nil
}
```

When a variable (without pointer) is passed into a function, the function receives a copy of the variable and is unable to mutate the caller's data. In the following example, an integer variable is created and assigned the value of 5, then passed to the function *increment* but when printing the variable after the function, the value still is 5 because the increment function receives a copy of *x* and doesn't return his value to be stored again in the same variable *x*.

```
func increment(x int) {
   x++
}

func main() {
   x := 5
   increment(x)

   fmt.Println(x)
```

```
}
```

To change the actual value of the variable, you can pass it as a pointer, so changes to the parameter *x* inside *increment* directly affect the original *x* in *main*. Like this:

```
func increment(x *int) {
   *x++
}

func main() {
   x := 5
   increment(&x)
   fmt.Println(x)
}
```

When you call *increment(&x)*, you are passing a reference (memory address) to the *x* variable. Inside the *increment* function, you dereference the pointer using **x* and increment the value it points to. This change directly affects the original *x* variable. In Go, when you pass an argument by reference (as a pointer), changes made to the parameter within the function affect the original variable outside the function.

A function can return a value that the caller doesn't care about. Do this by explicitly ignoring variables by using underscore _ , is not a conventional name, it is the way for the compiler to understand that this value will be ignored. Is crucial to ignore what will not be intended to use, otherwise the Go compiler will throw an error if there are any unused variable declarations in the code. In the following example, even though *getPoint()* returns two values, we can capture the first one and ignore the second by using the underscore:

```
func getPoint() (x, y int) int {
   return 3, 4
}

x, _ := getPoint() // ignore y value
```

Named Return Values

A return statement without arguments returns the named return values. This is known as a “naked” return. Naked return statements should be used only in short functions because they can harm readability in longer ones.

```
func getCoords() (x, y int) {
   // x and y are initialized with zero values
   return // automatically returns x and y
}
```

The function above has the same behavior as the following one:

```
func getCoords() (int, int) {
   var x int
   var y int
   return x, y
}
```

In the first example, *x* and *y* are the return values. At the end of the function, we could simply write *return* to return the values of those two variables, rather than writing *return x, y*.

Anonymous Function

Anonymous function is a function which doesn’t contain any name. They are often used for short, one-off tasks where defining a separate named function is unnecessary and are defined using the *func* keyword. It is useful when you want to create an inline function. Below is a simple example just for understanding his syntax.

```
func(){
   fmt.Println("Hello world!")
}()
```

In Go language, you are allowed to assign an anonymous function to a variable. When you assign a function to a variable, the type of the variable

is of function type and you can call that variable like a function call as shown in the below example.

```
value := func(){
  fmt.Println("Hello World!")
}
value()    // Output: Hello World!
```

Function as Parameters

In Go, functions are first-class citizens, meaning you can pass functions as parameters to other functions, which can lead to more flexible and modular code. This programming technique is often referred to as "higher-order functions" and enables implementing strategies, callbacks, and more.

```
func applyOperation(a, b int, op func(int, int) int) int {
   return op(a, b)
}
```

Anonymous functions are often used as arguments to other functions, especially when the function's behavior needs to be customized. They are functions without a name defined using the *func* keyword and are useful when the function has to be short and one-off tasks. They can be assigned to variables, passed as arguments to other functions and used directly.

This is useful when you want to capture and use variables from the surrounding scope inside the function.

Variadic Functions

Go allows the creation of functions that can accept a variable number of arguments of a specific type. This can be very useful when you want to work with a flexible number of values without explicitly specifying their count. A variadic function is declared by adding ... after the type of the last parameter in the parameter list. This parameter acts as a slice that collects

all the arguments passed to the function. When you call a variadic function, you can pass any number of arguments of the specified type, including zero arguments. The arguments are automatically collected into a slice within the function.

```
func sum(numbers ...int) int {
    total := 0
    for _, num := range numbers {
        total += num
    }
    return total
}

func main() {
    result := sum(1, 2, 3) // Pass any number of arguments
    fmt.Println(result)    // Output: 6
}
```

Variadic parameters can be combined with regular parameters in a function, but the variadic parameter must be the last parameter in the list of parameters. Variadic functions are powerful tools that make the code more flexible and versatile.

Defer Statement

Used to schedule a function call to be executed after the current function completes, but before it returns. Imagine you have a series of tasks to complete in a program, and you want to make sure that a particular task happens just before you finish something, regardless of whether everything else goes smoothly or not. This is where *defer* comes in.

Basically you are telling the compiler to remember that function and its arguments but not to execute it right away. Instead, put it on hold until the surrounding function is finished. Here are a pretty simple and basic example:

```
func main() {
    defer fmt.Println("This will be executed last.")
```

```
    fmt.Println("This will be executed first.")
}
```

Defer is helpful because it ensures that important tasks are never accidentally skipped, even if an error occurs. For example, if you're working with files, you can use *defer* to close the file when you're done, no matter how the function exits (due to successful completion or an error).

Defer can be especially handy in complex functions, ensuring that resources are cleaned up, temporary files are deleted, or final logs are recorded, no matter what path your code takes.

Structures

A struct in Go is a composite data type that groups together variables (fields) under a single name. It's similar to a dictionary in Python or object literals in JavaScript. Structs allow you to define custom data structures to represent complex data. To declare it, use the *type* keyword followed by the structure name and a list of field names and their types enclosed in curly braces.

```
type Person struct {
    FirstName string
    LastName  string
    Age       int
}
```

You can create instances (objects) of a struct by specifying values for its fields. This is similar to creating objects from classes in object-oriented programming.

```
person := Person{
    FirstName: "John",
    LastName:  "Doe",
    Age:       30,
}
```

```
fmt.Println(person.FirstName) // John
```

The *Person* syntax creates a new *Person* instance and inside the curly braces, you specify each field and its corresponding value as a key-value pair. You can specify values for struct fields in any order, as long as you label each value with the correct field name, allowing flexibility and readability when initializing structs with many fields. To access the fields of a struct use dot notation.

OBS: If you create a struct instance without specifying values for its fields, the compiler assigns "zero values" to each field based on its type.

Just like functions, Go allows the creation of anonymous structs, which are the same as the normal structs but it is defined without a name and therefore cannot be referenced elsewhere in the code. This is useful for small, one-time use data structures and also prevents re-using a struct definition that was never intended to be re-used.

Go supports a concept called struct embedding, a powerful and flexible feature that allows you to compose complex data structures by embedding one struct within another. It's similar to the concept of inheritance in object-oriented programming but offers some distinct advantages. It facilitates cleaner code by reusing fields and methods from embedded structs, supports encapsulation to control data visibility, enables method delegation, and fosters polymorphism. This is helpful to model real-world relationships intuitively, making the code more expressive and readable. It also avoids boilerplate code by allowing you to embed reusable functionality, such as logging or behavior interfaces, within multiple structs. This feature enhances code organization and maintainability, making it a fundamental for building efficient and modular programs.

```
type Contact struct {
   Email   string
   Phone   string
}

type Person struct {
   FirstName string
   LastName  string
```

```
    Contact    Contact // Embedded struct
}
```

Methods on Structs

While Go is **NOT** object oriented, it does support methods which can be defined on structs. Methods are functions that have a receiver, that is a special parameter that goes before the name of the function. The receiver parameter associates the method with a specific type, the convention is to use a short name related to the type. Methods enable you to define behavior that is specific to a particular type, like the example below:

```
type rect struct {
    width, height int
}

func (r *rect) area() int {
    return r.width * r.height
}

func main() {
    r := rect{
        width: 10,
        height: 5,
    }
    fmt.Println(r.area()) // 50
}
```

In the example, we declare a method named *area* that operates on a *rect* struct. It has a receiver parameter *(r *rect)*, which means it can be called on instances of the *rect* struct. The method calculates and returns the area of the rectangle by multiplying *width* and *height* fields. We could have stored the area as a separate field in the *rect* struct, but if we update the height and forget to update the area, we have a bug in the code. To avoid these situations we use methods to have a one line assessor to get the area, but without actually storing that area in the structure.

Interfaces

They are collections of method signatures. It provides a way to specify a contract that certain types must adhere to, allowing different implementations to be used interchangeably as long as they fulfill the defined method signatures. A type “implements” an interface if it has all of the methods of the given interface defined on it. In the following example, a shape must be able to return its area and perimeter.

```
type shape interface {
   area() float64
   perimeter() float64
}

type rect struct {
   width, height float64
}

type circle struct {
   radius float64
}

func (r rect) area() float64 {
   return r.width * r.height
}

func (r rect) perimeter() float64 {
   return 2 * (r.width + r.height)
}

func (c circle) area() float64 {
   return math.Pi * math.Pow(c.radius, 2)
}

func (c circle) perimeter() float64 {
   return 2 * math.Pi * c.radius
}
```

In this example, we can see that the *rect* and *circle* structures implement both *area* and *perimeter* methods as specified by the *shape* interface. If both types have methods that match the method signatures defined in the interface, both *rect* and *circle* types automatically satisfy (“implements”) the *shape* interface.

A type automatically implements an interface if its methods match the interface's method signatures, without any explicit declaration. This implicit implementation promotes clean code and reduces the need for explicit inheritance. Any type that satisfies the interface contract can be used wherever that interface is expected, enabling dynamic behavior and easy swapping of implementations. Go's interfaces are "satisfied implicitly" rather than requiring explicit declarations, enhancing simplicity and flexibility in designing code that can adapt to different implementations.

A type can implement any number of interfaces. For example, the empty interface *(interface{})* is always implemented by every type because it has no requirements. An interface should define what is necessary for other types to classify as a member of that interface. They should not be aware of any types that happen to satisfy the interface at design time.

```
type car interface {
   Color() string
   Speed() int
   IsFireTruck() bool
}
```

The above example is an interface to describe the components necessary to define a car. *Color()* and *Speed()* make perfect sense, they are methods confined to the scope of a car. *IsFireTruck()* is an anti-pattern, because we are forcing all cars to declare whether or not they are firetrucks.

Errors

Syntax best practices

- **Files:** The convention suggests using lowercase and short, meaningful names for Go files. If a filename consists of multiple words, it's common to use underscores to separate them. Choose short yet descriptive names that convey the purpose or content of the file. Avoid overly long names, but make sure the name is clear enough for other developers to understand. The filename should generally match the package name declared in the file. For example, a package named mypackage, the filename should be *mypackage.go*

- **Variables:** The convention is using camelCase and choosing names that clearly describe the purpose and content of the variable. This makes the code self-explanatory and reduces the need for extensive comments, striking a balance between clarity and brevity is essential.

  Variables declared at the package level (i.e., variables accessible outside functions) should start with a lowercase letter. This signifies that the variable's scope is limited to the package.If a variable needs to be accessible outside the package (exported), start its name with an uppercase letter. This indicates that the variable is part of the package's public interface.
- **Functions:** Typically use camelCase. If they are intended to be exported (public functions), they will start with an uppercase letter. Functions starting with lowercase letters are treated as private to the package.

- **Structures:** Naming structs in Go follows similar conventions to naming variables. Use CamelCase and keep concise while still conveying their purpose. While structs often represent a single entity, sometimes will represent a collection, configuration of related items. In such cases, using the plural might be appropriate.

- **Interfaces:** Important to use names that clearly convey the role and purpose of the interface. Choose names that succinctly describe the behavior or role of the interface. Should give a clear idea of what methods it defines or what functionality it provides. A common convention for naming interfaces is to use the "er" suffix because it

represents something capable of performing an action or exhibiting a behavior. For example: *Reader, Writer, Formatter*.

Packages

In Go, the *package main* declaration is used to indicate that the current Go code file is intended to be the entry point of a standalone executable program. The main package contains the *main* function, which serves as the starting point for program execution. When the program is compiled and executed, the code within the main function is executed, allowing developers to define the primary logic and functionality of the executable program. This package is crucial for creating executable programs in Go, and it distinguishes the entry point from other packages that might be used to organize and modularize code in larger projects.

Slices

Operators Table

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
=	Assignment
+=	Addition assignment

-=	Subtraction assignment
***=**	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
&&	Logical AND
!	Logical NOT
&	Bitwise AND
^	Bitwise XOR
<<	Left shift
>>	Right shift

- Main function does not support passing arguments. They are passed directly from os.Args

- Variables declared inside an if short live

Switch Control Structure

- **Break statement:** Go's switch statement a
- **Default Case:**
- **Tag-less Switch:**
- **Type Switch:**

For range

Perfomance