**01.19.2023 Golang Lesson Instruction**

**1. Reflection**

Reflection is an advanced Go feature. Not it is an easy subject but because it is going to help understand how Go works with different Datatypes.

Reflection in Go is a form of metaprogramming. Reflection allows us to examine types at runtime. It also provides the ability to examine, modify, and create variables, functions, and structs at runtime. The Go reflect package gives you features to inspect and manipulate an object at runtime. Reflection is an extremely powerful tool for developers and extends the horizon of any programming language. Types, Kinds and Values are three important pieces of reflection that are used in order to find out information.

There are two question.

* Why was **Reflection** included in Go?
* When should we use **Reflection?**

Answer for the first question, Go provides *reflect* package for working with reflection. For example fmt.Println() is enough to understand the data type. The fmt package reflection to do that.

Answer for the second question, reflection allows you to handle and work with data types that do not exist at the time at which you write your code but might exist in the future. When we use an existing package with user-defined data types.

The most useful parts of the reflect package are two data types named reflect. Value and reflect.Type. Now, reflect.Value is used for storing values of any type, whereas reflect.Type is used for representing Go types. There exist two functions named reflect.TypeOf and reflect.ValueOf that return the reflect.Type and reflect.Value values, respectively. Note that reflect.TypeOf returns the actual type of variable-if we are examining a structure, it returns the name of the structure.

As structure are really important in Go, the reflect package offers the ***reflect.Numfield()*** method for listing the number of fields in a structure as well as the ***Field()*** method for getting the reflect.Value value of a specific field of a structure.

Last, the **Int()** and **String()** methods return the integer and string value of a reflect.Value.

Reflection code can look unpleasant and hard to read sometimes. Therefore, according to the Go philosophy, you should rarely use reflection unless it is absolutely necessary because despite its cleverness, it does not create clean code.

It takes a [interface{}](https://golangbot.com/interfaces-part-1/#emptyinterface) as argument. For simplicity, we will only deal with structs that contain fields of type string and int but this can be extended for any type.

***This is good resources for simply learning reflection:*** <https://golangbot.com/reflection/>

**2. The three disadvantages of reflection**

Without a doubt, reflection is a powerful Go feature. However, as with all tools, reflection should be used sparingly for three main reasons:

• The first reason is that extensive use of reflection will make your programs hard to read and maintain. A potential solution to this problem is good documentation, but developers are notorious for not having the time to write proper documentation.

• The second reason is that the Go code that uses reflection makes your programs slower. Generally speaking, Go code that works with a particular data type is always faster than Go code that uses reflection to dynamically work with any Go data type. Additionally, such dynamic code makes it difficult for tools to refactor or analyze your code.

• The last reason is that reflection errors cannot be caught at build time and are reported at runtime as panics, which means that reflection errors can potentially crash your programs. This can happen months or even years after the development of a Go program! One solution to this problem is extensive testing before a dangerous function call. However, this adds even more Go code to your programs, which makes them even slower.

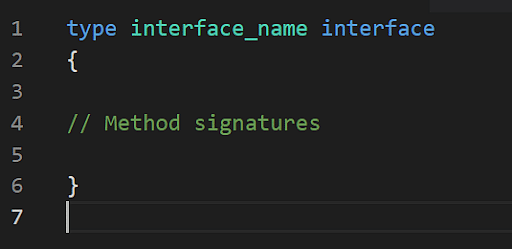
**3. Interface**

In GO, an interface is **a set of method signatures**. When a type provides definition for all the methods in the interface, it is said to implement the interface. It is much similar to the OOP world. Interface specifies what methods a type should have and the type decides how to implement these methods.

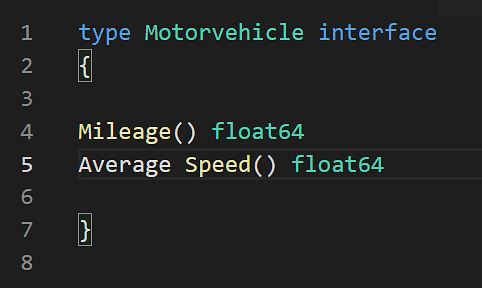
The key principle of an interface in Go is to provide method signatures for similar types of objects. Go does not have classes and inheritance to implement object orientation. Go has the interface, an abstract type to fulfill this purpose.

**How to create an Interface?**

In Go, you can create an interface using the ***type*** keyword, followed by the name of the interface and the keyword interface. And, you can specify method signatures inside curly braces.



You will get to understand the creation of interfaces with example



**Line no. 1** of the program above creates an interface type named **Motorvehicle** which has the method **Mileage()** and **Average Speed()**.