The reflect Package

This lesson will help you explore the reflect package more relative to interfaces.

WE'LL COVER THE FOLLOWING

- Methods and types in `reflect`
- Modifying (setting) a value through reflection
- Creating a type through reflection

Methods and types in `reflect`

In Chapter 8, we saw how `reflect` can be used to analyze a struct. Here, we elaborate further on its powerful possibilities. Reflection in computing is the ability of a program to examine its structure, mainly through the types; it's a form of *meta-programming*. The `reflect` can be used to investigate types and variables at runtime, e.g., their size, and methods, and it can also call these methods *dynamically*. It can also be useful to work with types from packages of which you do not have the source. It's a powerful tool that should be used with care and avoided unless strictly necessary.

Basic information of a variable is its type and its value: these are represented in the reflection package by the types *Type*, which represents a general Go type, and *Value*, which is the reflection interface to a Go value. Two simple functions, `reflect.TypeOf` and reflect.ValueOf, retrieve the Type and Value pieces out of any value. For example, if `x` is defined as:

```
var x float64 = 3.4
```

Then, `reflect.TypeOf(x)` gives *float64* and `reflect.ValueOf(x)` returns **3.4**.

Reflection works by examining an interface value; the variable is first converted to the empty interface. This becomes apparent if you look at the signatures of both of these functions:

```
func TypeOf(i interface{}) Type
func ValueOf(i interface{}) Value
```

The interface value then contains a type and value pair.

Reflection goes from interface values to reflection objects and back again, as we will see. Both `reflect.Type` and `reflect.Value` have lots of methods that let us examine and manipulate them. One important example is that Value has a Type method that returns the Type of a `reflect.Value`. Another is that both Type and Value have a `Kind` method that returns a constant, indicating what sort of item is stored: Uint, Float64, Slice, and so on. Also, methods on Value with names like Int and Float let us grab the values (like int64 and float64) stored inside. The different kinds of Type are defined as constants:

```
type Kind uint8
const (
  Invalid Kind = iota
  Bool
  Int
  Int8
  Int16
  Int32
  Int64
  Uint
  Uint8
  Uint16
  Uint32
  Uint64
  Uintptr
  Float32
  Float64
  Complex64
  Complex128
  Array
  Chan
  Func
  Interface
  Map
  Ptr
  Slice
  String
```

```
  Struct


  UnsafePointer
)
```

For our variable `x`, if `v := reflect.ValueOf(x)` then `v.Kind()` is float64, so the following is true:

```
v.Kind() == reflect.Float64
```

The `Kind` is always the underlying type: if you define:

```
type MyInt int
var m MyInt = 5
v := reflect.ValueOf(m)
```

Then, `v.Kind()` returns `reflect.Int`.

The `Interface()` method on a Value recovers the (interface) value, so to print a Value `v` do this:

```
fmt.Println(v.Interface())
```

Experiment with these possibilities with the following code:

```
package main
import (
"fmt"
"reflect"
)

func main() {
  var x float64 = 3.4
  fmt.Println("type:", reflect.TypeOf(x))
  v := reflect.ValueOf(x)
  fmt.Println("value:", v)
  fmt.Println("type:", v.Type())
  fmt.Println("kind:", v.Kind())
  fmt.Println("value:", v.Float())
  fmt.Println(v.Interface())
  fmt.Printf("value is %5.2e\n", v.Interface())
  y := v.Interface().(float64)
  fmt.Println(y)
}
```

In the above code at **line 4**, we import a package `reflect`. Look inside `main`. At **line 8**, we declare a *float64* type variable `x` and assign a **3.4** value to it. Now, in the next line, we are printing the *type* of `x` through the `TypeOf()` function of the `reflect` package which is *float64*. At **line 10**, we are reading the value of `x` in new variable `v` through `ValueOf()` function of the `reflect` package and printing it in the next line which is **3.4**.

Look at **line 12**. The type of `v` is printed just by using the `Type()` method, which is *float64*. In the next line, the `Kind()` method is called by `v`, and the returned value is printed, which is also the type of `v`, i.e., *float64*. At **line 14**, to print the value of `v`, the `Float()` method is called by `v`. At **line 15**, we recover the interface value of `v`, and print `v` in the **%5.2e** format in the next line. At **line 17**, we convert the type of recovered value to *float64* again and store it in `y`, which is later printed in the next line.

Modifying (setting) a value through reflection

Continuing with the previous example, suppose we want to modify the value of `x` to be **3.1415**. Value has a number of `Set` methods to do this, but here we must be careful:

```
v.SetFloat(3.1415)
```

It produces an error: `will panic: reflect.Value.SetFloat using unaddressable value`. Why is this? The problem is that `v` is not settable (not that the value is not addressable). *Settability* is a property of a reflection Value, and not all reflection Values have it. It can be tested with the `CanSet()` method. In our case, we see that this is false, *settability of v: false*. When `v` was created with `v := reflect.ValueOf(x)`, a copy of `x` was passed to the function, so it is logical that you can't change the original `x` through `v`.

In order to change `x` through `v`, we need to pass the address of `x: v = reflect.ValueOf(&x)`. Through `Type()`, we see that `v` is now of the type **float64* and still not settable. To make it settable, we need to let the `Elem()` function work on it, which indirects through the pointer `v = v.Elem()`. Now, `v.CanSet()` gives true and `v.SetFloat(3.1415)` works!

```go
package main
import (
"fmt"
"reflect"
)

func main() {
  var x float64 = 3.4
  v := reflect.ValueOf(x)
  // setting a value:
  // Error: will panic: reflect.Value.SetFloat using unaddressable value
  // v.SetFloat(3.1415)
  fmt.Println("settability of v:", v.CanSet())
  v = reflect.ValueOf(&x) // Note: take the address of x.
  fmt.Println("type of v:", v.Type())
  fmt.Println("settability of v:", v.CanSet())
  v = v.Elem()
  fmt.Println("The Elem of v is: ", v)
  fmt.Println("settability of v:", v.CanSet())
  v.SetFloat(3.1415) // this works!
  fmt.Println(v.Interface())

  fmt.Println(v)
}
```

Setting Value through Reflection

In the above code, at **line 4**, we import a package `reflect`. At **line 8**, we declare a *float64* type variable `x` and assign **3.4** value to it. At **line 9**, we are reading the value of `x` in the new variable `v` through `ValueOf()` function of the `reflect` package.

Now, before setting a value to `x` it's better to check it's settable or not to avoid errors. Look at **line 13**. We are calling `canSet()` method on `v`, and printing the result. It will print *false*, as `v` isn't settable because it's unaddressable. Look at **line 14**, we are reading the address of `x` (pointer) in new variable `v` through `ValueOf()` function of the `reflect` package. In the next line, printing the type of `v` gives ***float64**. Again, before setting a value to `x`, it's better to check if it's settable or not to avoid errors. Look at **line 16**. We are calling `canSet()` method on `v`, and printing the result. It will print *false*, as `v` isn't settable.

Now, we try `Elem()` method on `v` at **line 17**. The **line 18** prints the *elem* of `v`, which is its value **3.4**. In the next line, we are again checking its settability. This time it works. We set the value of `v` to a floating value **3.1415** at **line 20**. At **line 21**, we recover the interface value of `v`, and print `v` at the **line 23**.

At **line 21**, we recover the interface value of `v`, and print `v` at the **line 23**, which is **3.1415** now.

Reflection Values need the address of something in order to modify what they represent.

Creating a type through reflection

In the following example a map is created from a struct:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
	"fmt"
	"reflect"
	"strings"
)

func mapToStruct(m map[string]interface{}) interface{} {
	var structFields []reflect.StructField

	for k, v := range m {
		sf := reflect.StructField{
			Name: strings.Title(k),
			Type: reflect.TypeOf(v),
		}
		structFields = append(structFields, sf)
	}

	// Creates the struct type
	structType := reflect.StructOf(structFields)

	// Creates a new struct
	structObject := reflect.New(structType)

	return structObject.Interface()
}


func main() {

	m := make(map[string]interface{})

	m["name"] = "Barack"
	m["surname"] = "Obama"
	m["age"] = 57

	s := mapToStruct(m)
	fmt.Printf("%t %[1]v\n", s)
```

```
        fmt.Printf("%t %[1]v\n", s)

        sr := reflect.ValueOf(s)
        sr.Elem().FieldByName("Name").SetString("Donald")
        sr.Elem().FieldByName("Surname").SetString("Trump")
        sr.Elem().FieldByName("Age").SetInt(72)
        fmt.Println(s)
}
```

Click the **RUN** button and wait for the terminal to start. Type `go run main.go` and press ENTER. In case you make any changes to the file, you have to press **RUN** again.

In the above code, at **line 8**, we have the header of the function `mapToStruct`. It shows that this function takes an interface of type `map[string]` and returns an interface. In the next line, we made a variable `structFields`, which is a *slice* of the type `reflect.StructFields`. At **line 11**, we have a for loop that is reading the fields and making a struct `sf` of type `reflect.structFields` in each iteration and assigning fields with the values read in that iteration. Before an iteration ends, `sf` is appended in `structFields`.

This reflection has a considerable impact on the standard library of Go. We can customize the standard functions using reflection. To learn more, move to the next lesson.