Printf and Reflection

This lesson shows how to modify printing using the reflect package and how to overload a function using an empty interface.

WE'LL COVER THE FOLLOWING

Introduction

The capabilities of the reflection package discussed in the previous section are heavily used in the standard library. For example, the function `Printf` and so on use it to unpack its ... arguments. `Printf` is declared as:

```
func Printf(format string, args ... interface{}) (n int, err error)
```

The ... argument inside `Printf` has the type `interface{}`, and `Printf` uses the reflection package to unpack it and discover the argument list. As a result, `Printf` knows the actual types of their arguments. Because they know if the argument is unsigned or long, there is no %u or %ld, only %d. This is also how `Print` and `Println` can print the arguments nicely without a format string.

Explanation

To make this more concrete, we implement a simplified version of such a generic print-function in the following example, which uses a type-switch to deduce the type. According to this, it prints the variable out.

```
package main
import (
  "os"
  "strconv"
)

type Stringer interface {

  String() string
}

type Celsius float64

func (c Celsius) String() string {
  return strconv.FormatFloat(float64(c),'f', 1, 64) + " °C"
}

type Day int

var dayName = []string{"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
"Sunday"}

func (day Day) String() string {
  return dayName[day]
}

func print(args ...interface{}) {
  for i, arg := range args {
    if i > 0 {os.Stdout.WriteString(" ")}
    switch a := arg.(type) { // type switch
      case Stringer: os.Stdout.WriteString(a.String())
      case int: os.Stdout.WriteString(strconv.Itoa(a))
      case string: os.Stdout.WriteString(a)
      // more types
      default: os.Stdout.WriteString("???")
    }
  }
}

func main() {
  print(Day(1), "was", Celsius(18.36)) // Tuesday was 18.4 °C
}
```

Print and Reflection

In the above code, at **line** 7, we define an interface `Stringer` with one method `String()` that returns a *string*. At **line 11**, we define a type `Celsius` on the basis of type *float64*. Then, we have a method `String` that can be called by the object of type `Celsius` and return a string, bringing variation to the print mechanism of Go. It is converting the `c` type object to string after parsing *float64* and rounding off to 1 digit.

At **line 17**, we define a type `Day` on the basis of type *int*. Then, at **line 19**, we make a slice of *strings* `dayName`, and add the name of seven days of the week to it. Then, we have a method `String` that can be called by object of type `Day` and return a string. It is returning the day present at the `day` index in the `dayName` slice.

Now, look at the header of the `print` function at **line 26**. It takes multiple arguments with the `interface{}` type (not sure how many arguments will it take on runtime, or of which type). To manage it, we make a *for* loop for checking each argument one by one in a single iteration. Now, we have switch cases ahead. Cases are to be judged on *type* of `arg`. The first case will be true if `arg` is of `Stringer` type. In this case, a decision will be made whether to call `String()` for `Day` or `Celsius`. The second case will be true if `arg` is of *int* type. The third case will be true if `arg` is of *string* type. What if no case is found true? In this case, we have a *default* case that prints the message of *???*.

Now, look at `main`. At **line 40**, we are calling `print` function with arguments as: `print(Day(1), "was", Celsius(18.36))`. Now, control will transfer to **line 26**. As there are three arguments, for loop will cover three iterations. For the first iteration, the first case `Stringer` will be true, and the `String()` method for type `Day` will be called (at **line 22**), and **Tuesday** will be printed on the screen. For the second iteration, the third case `string` will be true, and the `WriteString()` method from the `os` package will print **was** will be printed on the screen. For the third iteration, the first case `Stringer` will be true, and the `String()` method for type `Celsius` will be called (at **line 22**), and **18.4** will be printed on the screen.

Empty interface and function overloading:

In Chapter 4, we saw that function overloading is not allowed. In Go, this can be accomplished by using a variable number of parameters with `...T` as the last parameter. If we take `T` to be the empty interface and we know that a variable of any type satisfies `T`, then this allows us to pass any number of parameters of any type to that function, which is what overloading means. This is applied in the definition of the function

```
fmt.Printf(format string, a ...interface{}) (n int, errno error)
```

This function iterates over the slice `a`, discovering the type of its arguments dynamically. For each type, it looks if a method `String()` is implemented; if so, this is used for producing the output.

That is it much about the `reflect` package. Now in the next lesson, let's see

That is it much about the `reflect` package. Now in the next lesson, let's see how well Go handles dynamic typing.