Structs with Tags

In this lesson, we'll cover how to attach a tag with the field of a struct and how to read it.

WE'LL COVER THE FOLLOWING

- Tags: key “value” convention

A field in a struct can, apart from a name and a type, also optionally have a tag. It is a string (or a raw string) attached to the field, which could be documentation or some other important label. The tag-content cannot be used in normal programming; only the package `reflect` can access it. This package which we’ll explore deeper in the next chapter (Chapter 9), can investigate types, their properties, and methods in runtime. For example: `reflect.TypeOf()` on a variable gives its type. If this is a struct type, it can be indexed by `Field`, and then, the `Tag` property can be used.

The following program shows how this works:

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

type TagType struct { // tags
  field1 bool "An important answer"
  field2 string `The name of the thing`
  field3 int "How much there are"
}

func main() {
  tt := TagType{true, "Barack Obama", 1}
  for i:= 0; i < 3; i++ {
    refTag(tt, i)
  }
}

func refTag(tt TagType, ix int) {
  ttType := reflect.TypeOf(tt)
  ixField := ttType.Field(ix)       // getting field at a position ix
  fmt.Printf("%v\n", ixField.Tag)   // printing tags
}
```

Structs with Tags

In the above code, at **line 4**, we import a package `reflect`. At **line** 7, we are making a struct `TagType` type variable with three fields: `field1` of type *bool*, `field2` with type *string*, and `field3` with type *int*. You may have noticed with all fields, are some tags associated in *quotes* and *backticks*.

Before studying `main` let's focus on the function `refTag`. See its header at **line 20**: `func refTag(tt TagType, ix int)`. The function is taking a `Tagtype` variable `tt` and an integer variable `ix`.

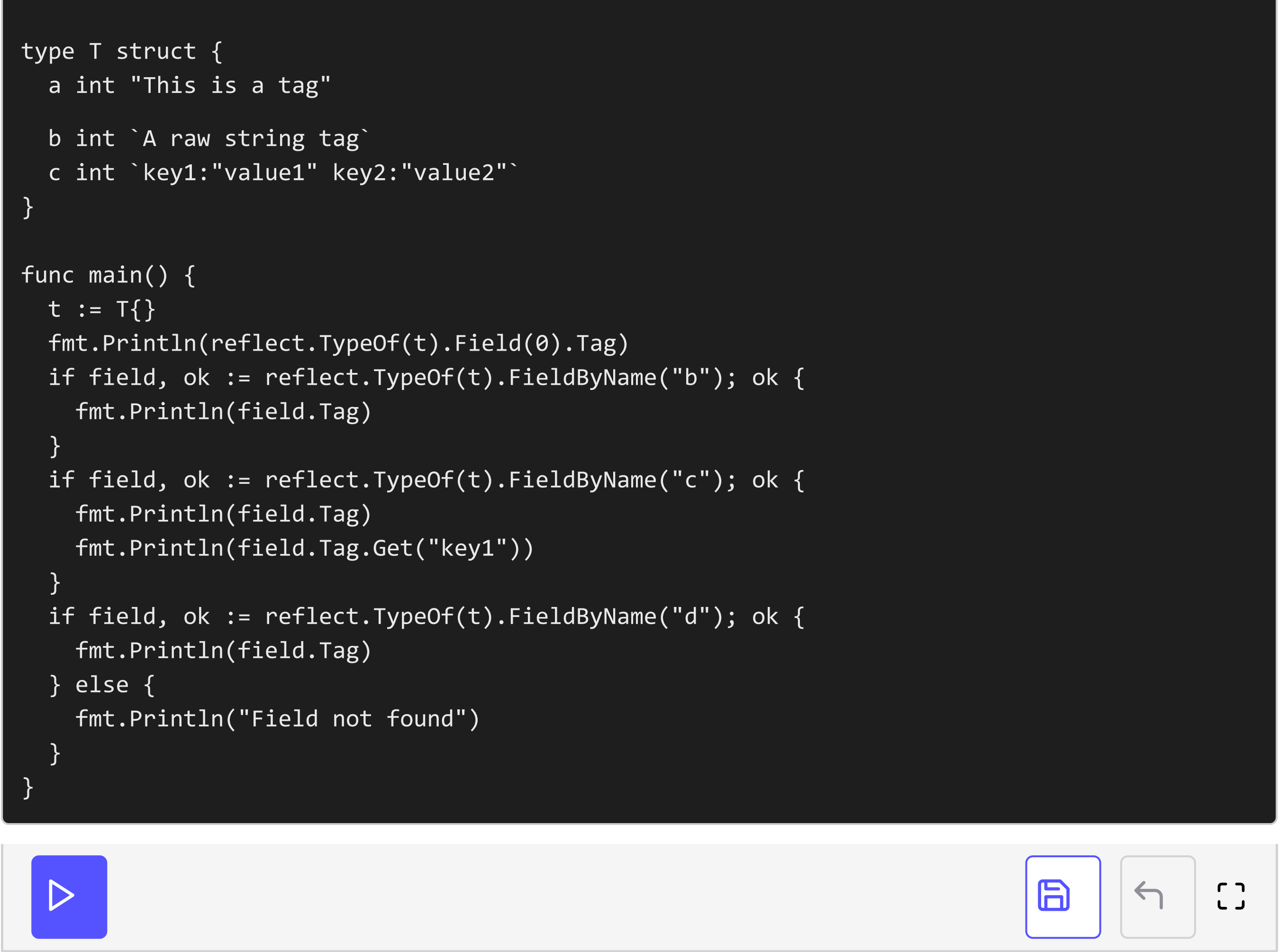
Using the `reflect` package, we are finding the type of `tt` and storing the type in `ttType` (at **line 21**). Then, we are using the `Field` function from the `reflect package` (at **line 22**) to find the field at position `ix` in `ttType` and storing it as `ixField`. At **line 23**, we are using `ixField` to find its tag, and then printing it.

Now, look at `main`. At **line 14**, we make a `TagType` variable `tt` and assign fields with values: **true**, **Barack Obama**, and **1**. Now, we wish to print *tags* associated with *fields* of `tt`. We'll use the package `refect` for this purpose. As we have three fields, we made a for loop at **line 15** with iterator `i`, starting from **0** and ending at **2**, which means it will iterate three times. In every iteration, we are calling a function `refTag`, which takes `tt` and iterator `i`. When `i` is **0**, the tag of the first field will be printed. When `i` is **1**, the tag of the second field will be printed. When `i` is **2**, the tag of the third field will be printed.

Tags: key “value” convention

Go allows the definition of multiple tags through the use of the `key: "value"` format. Look at the following example to see that once we have a tag, the value of its key can be retrieved through the `Get` method:

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



```
type T struct {
  a int "This is a tag"

  b int `A raw string tag`
  c int `key1:"value1" key2:"value2"`
}

func main() {
  t := T{}
  fmt.Println(reflect.TypeOf(t).Field(0).Tag)
  if field, ok := reflect.TypeOf(t).FieldByName("b"); ok {
    fmt.Println(field.Tag)
  }
  if field, ok := reflect.TypeOf(t).FieldByName("c"); ok {
    fmt.Println(field.Tag)
    fmt.Println(field.Tag.Get("key1"))
  }
  if field, ok := reflect.TypeOf(t).FieldByName("d"); ok {
    fmt.Println(field.Tag)
  } else {
    fmt.Println("Field not found")
  }
}
```

Getting Value from Tag

In the above code, at **line 4**, we import a package `reflect`. At **line** 7, we are making a struct `T` type variable with three fields: `a` of type *int*, `b` with type *int*, and `c` with type *int*. You may have noticed with all fields, are some tags associated, in *quotes* and *back ticks*.

Now, look at `main`. At **line 14**, we make a `T` variable `t` without assigning fields with values. At **line 15**, we are printing the tag of field **0** of `t` as: `fmt.Println(reflect.TypeOf(t).Field(0).Tag)`. In the next line, we are checking whether we have a field with the name `b` in `t`, using the `reflect` package as: `if field, ok := reflect.TypeOf(t).FieldByName("b"); ok`. If `b` exists then `ok` becomes **true**, and the field of `b` is stored in `field`. If `ok` is *true*, only then **line 17** will be executed. In this case, `ok` will be true because `t` has the field `b`. So the tag of field `b` will be printed on the screen.

Similarly, at **line 19**, we are checking whether we have a field with the name `c` in `t`, using the `reflect` package as: `if field, ok := reflect.TypeOf(t).FieldByName("c"); ok`. If `c` exists, then `ok` becomes **true**, and the field of `c` is stored in `field`. If `ok` is *true*, only then **line 24** will be executed; otherwise, **line 26** will be executed. In this case, `ok` will be *false* because `t` has field `d`. So, the tag of field `d` will not be printed on the screen (**line 24**). **Line 26** will be executed, and **Field not found** will be printed.

Now that you’re familiar with tag’s convention, in the next lesson, you’ll study the concept of anonymous fields and embedded structs.