Structures in Go (structs)

Unlike traditional Object-Oriented Programming, Go does not have class-object architecture. Rather we have structures which hold complex data structures.

Uday Hiwarale Follow
Oct 19, 2018 · 11 min read

☛ What is a struct?

`struct` or a structure can be compared with `class` concept in Object Oriented Programming. If you don't know what OOP is, then look at the struct as a recipe which declares the ingredients and their amount.

A structure has different fields of the same or different type. A struct is used mainly when you need to define a **schema type** made of different individual fields (*properties*). **Like `ross` is a type of `Employee` (struct) which has `firstName`, `LastName`, `salary` and `fullTime` status (schema fields).**

☛ Declaring struct type

A **struct type** is nothing but a schema containing the blueprint of a data a structure will hold. To make things simple, we need to used **type alias** so that we can refer to struct type easily. The syntax to create a struct is

```
type StructName struct{
    field1 fieldType1
    field2 fieldType2
}
```

In the above syntax, `StructName` is a struct type while `field1` and `field2` are fields of `fieldType1` and `fieldType2` respectively.

Let's create a type `Employee` like we discussed, with some fields.

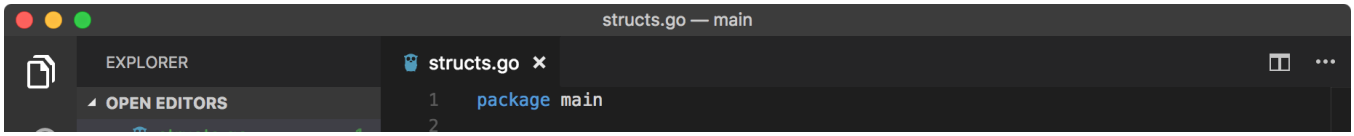
```
type Employee struct{
        firstName string
        lastName string
        salary int
        fullTime bool
}
```

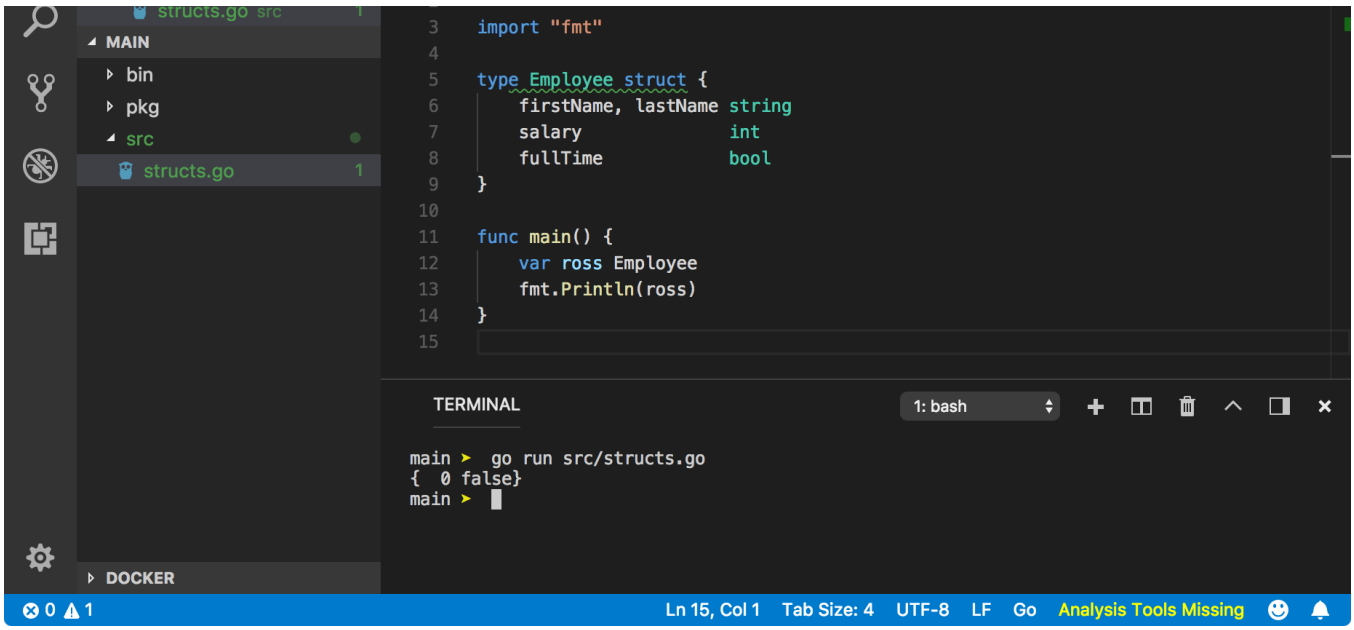
You can also combine different fields of the same time in the same line as we have seen in `variables` lesson. So we can write above program like below

```
type Employee struct {
        firstName, lastName string
        salary int
        fullTime bool
}
```

☛ Creating a struct

Since we created a struct type `Employee`, let's create a struct `ross` from it.
Since `Employee` is a type, creating a variable of that type is easy.





https://play.golang.org/p/c_Gf7YCXBJW

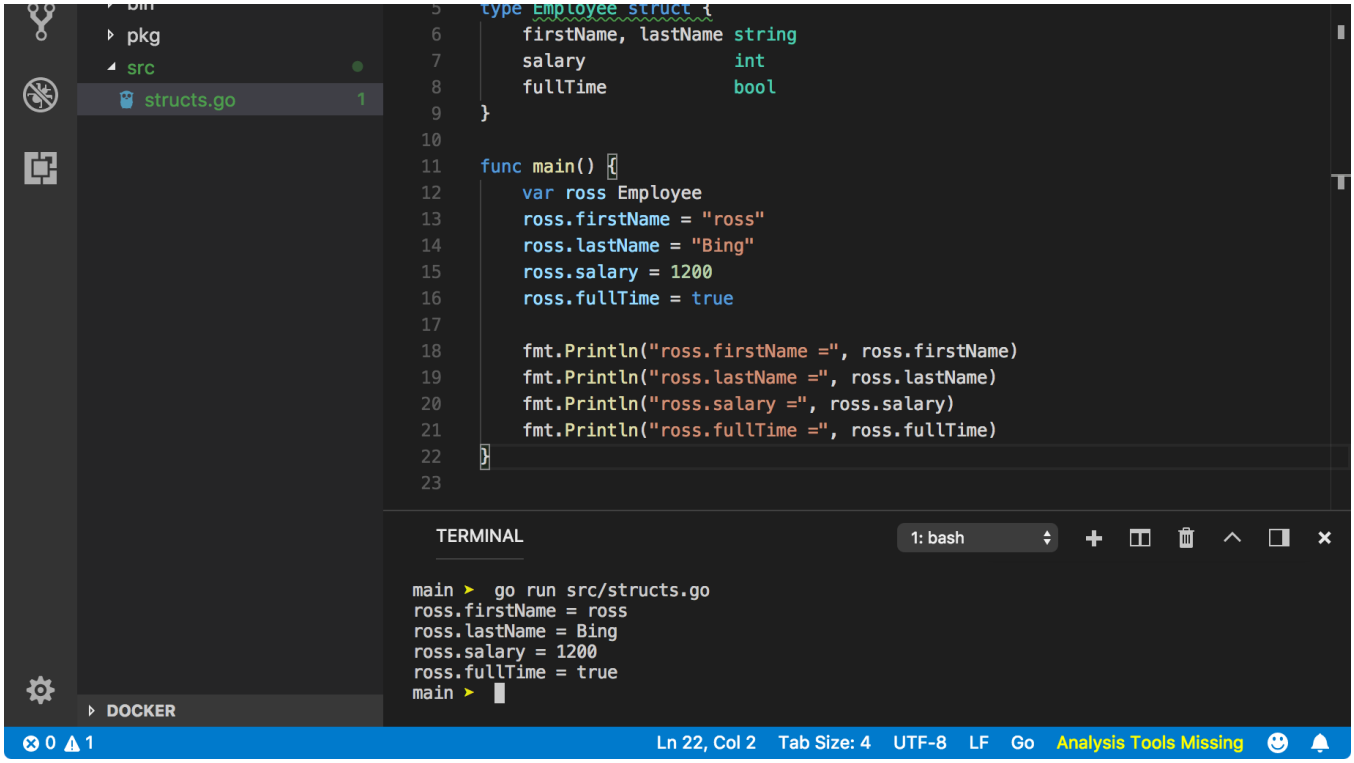
The output of the above program will look weird but it is giving **zero value** of a struct because we have defined variable `ross` of struct type `Employee` but haven't initialized it. A zero value of a struct is fields with their respective zero values. Hence `string` will have zero value of `""` (*which are printed in the result but you can't see*), `int` will have zero value of `0` and `bool` will have zero value of `false`.

> *When we are saying struct, we are referring to the variable which holds the type of `Employee`. So `Employee` is **struct type** while `ross` is **struct**, while struct is a **built-in type**. If it would be OOP, we would be calling `Employee` a class and `ross` an object while `class` a keyword.*

☛ Getting and setting struct fields

To get and set a struct field is very simple. When a struct is defined, using `.` (*dot*) notation, you can access struct fields with syntax `struct.field`.

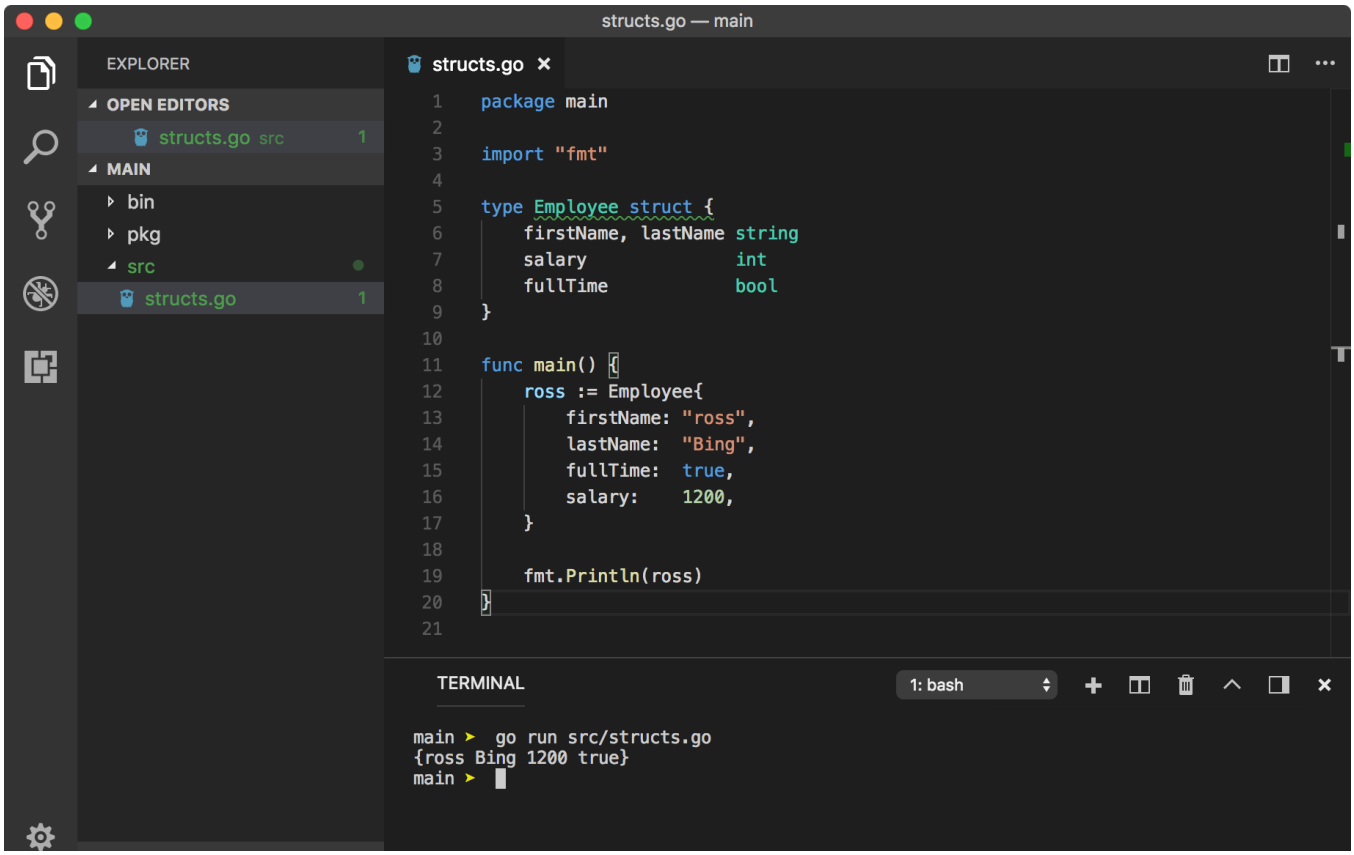
In the above program, we have defined `ross` which has 4 fields. For example, to assign a value to `firstName` field of `ross`, you need to use `ross.firstName = "ross"`. Let's give `ross` some credentials.



https://play.golang.org/p/Fw_Z6uBCinZ

☛ Initializing struct

Instead of creating a struct and then assigning values individually, we can create a struct with field values initialized in the same syntax.



https://play.golang.org/p/FGJ0Ja4WM-F

We have used shorthand notation `:=` to create a variable and Go will infer type `Employee` automatically. Order of appearance of the fields does not matter as you can see, we have initialized `fullTime` field before `salary` field.

You can also initialize some fields and leave some to zero values. In the below case, `ross` will be `{ross Bing 0 true}` as his salary at its zero value is `0`.

```
ross := Employee{
    firstName: "ross",
    lastName:  "Bing",
    fullTime:  true,
}
```

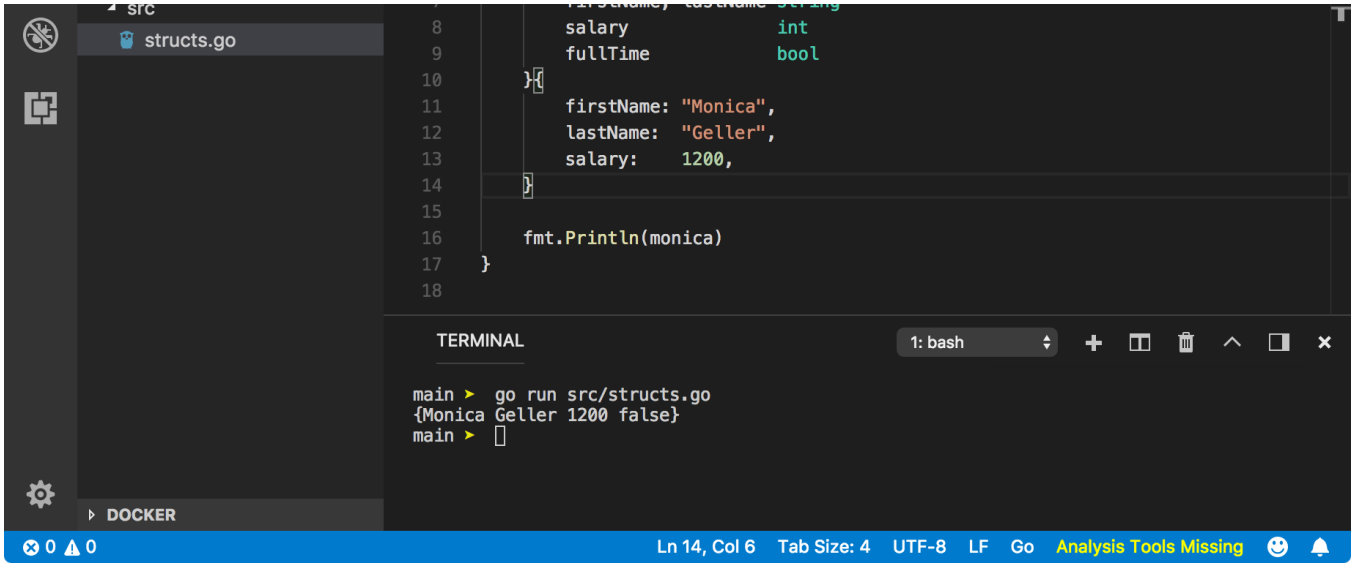
There is one another way to initialize a struct which does not include field name declarations like below.

```
ross := Employee{"Ross", "Geller", 1200, true}
```

Above syntax is perfectly valid. But when creating a struct without field names, you need to provide all field values in sequential order unlike.

☛ Anonymous struct

An `anonymous struct` is a struct with no explicitly defined **struct type alias**. So far, we have created `Employee` struct type alias which `ross` infers. But in case of an **anonymous struct**, we do not define any struct type alias and create struct from inline struct type and initial values defined in the same syntax.



https://play.golang.org/p/Np7Y8LuOdql

In the above program, we are creating struct `monica` without defining a struct type alias. This is useful when you don't want to re-use a struct type.

So you would be guessing if `ross` is of type `Employee` then what is the type of `monica`. Using `fmt.Printf` and `%T` format syntax, we got the following result

```
struct { firstName string; lastName string; salary int; fullTime bool
}
```

Looks weird? Not at all. Because this is how it would actually look like if we did not have aliased it. Now got it why we aliased it? To avoid the pain of writing above type over and over again.

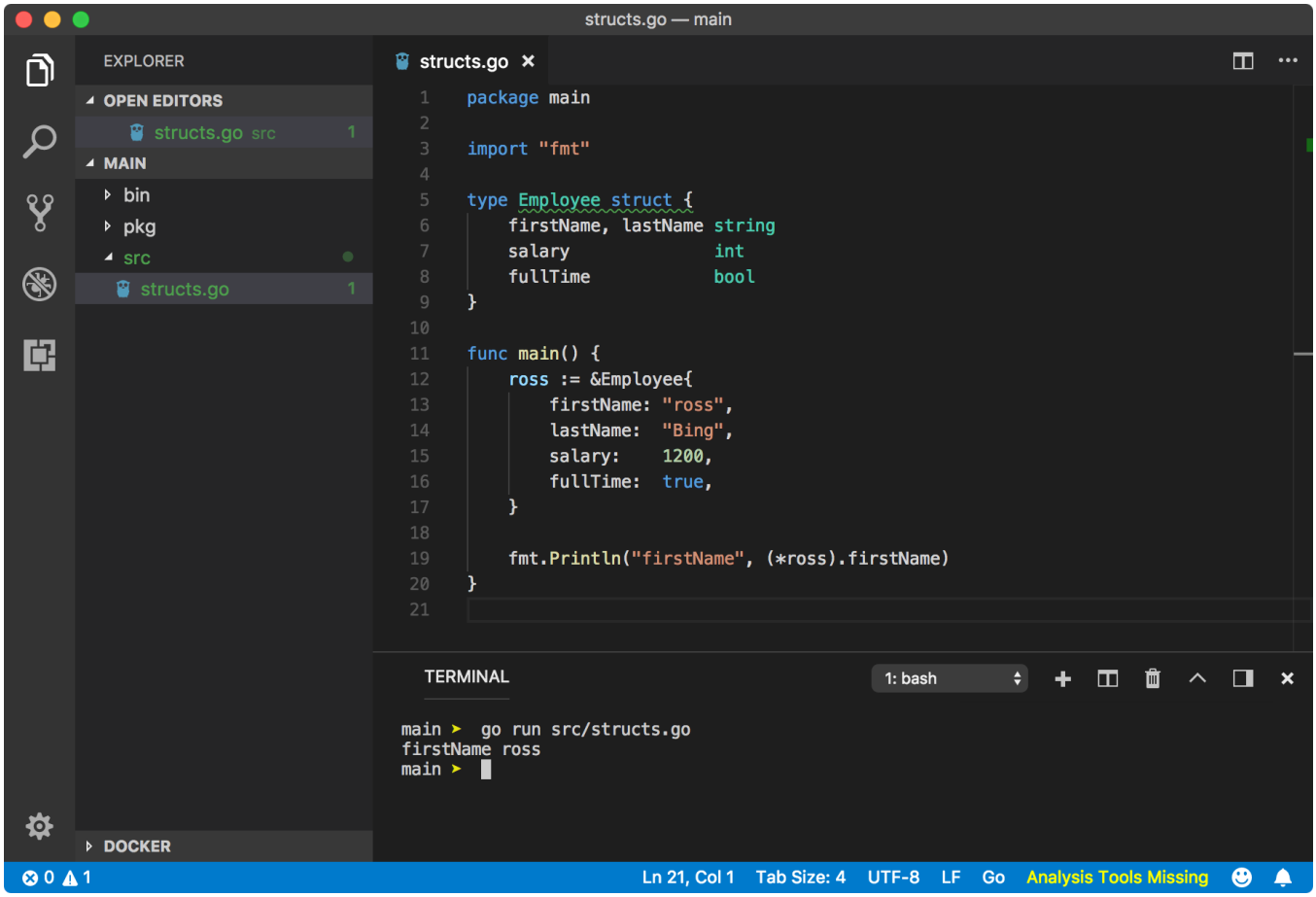
☛ Pointer to struct

Instead of creating a struct, we can create a pointer which points to the struct in one statement. This saves one step to create a struct (*variable*) and then create a pointer to that variable.

The syntax to create a pointer struct is

```
s := &StructType{...}
```

Let's create `ross` as a pointer to the struct



https://play.golang.org/p/fph03X-T-bu

In the above program, since `ross` is a pointer, we need to use `*ross` to get actual value and use `(*ross).firstName` to access `firstName` so that compiler doesn't get confused between `(*ross).firstName` and `*(ross.firstName)`.

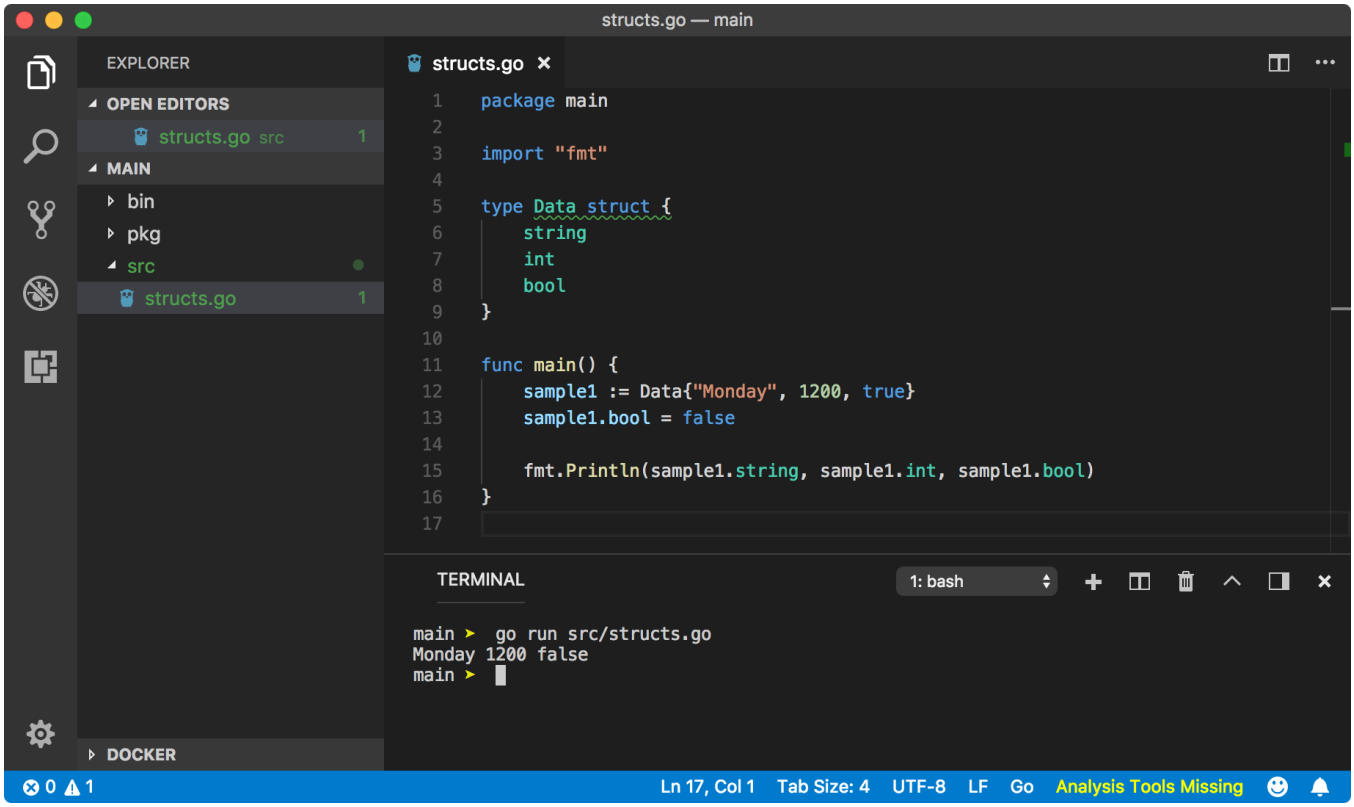
But Go provide another syntax to access fields like in case of `array` pointer. Without using `*`, we can use directly pointer to access fields.

```
ross := &Employee{
    firstName: "ross",
    lastName:  "Bing",
    salary:    1200,
    fullTime:  true,
}

fmt.Println("firstName", ross.firstName)
```

☛ Anonymous fields

You can define a struct type without any field names. You can just mention the field types and Go will use type as a field name.



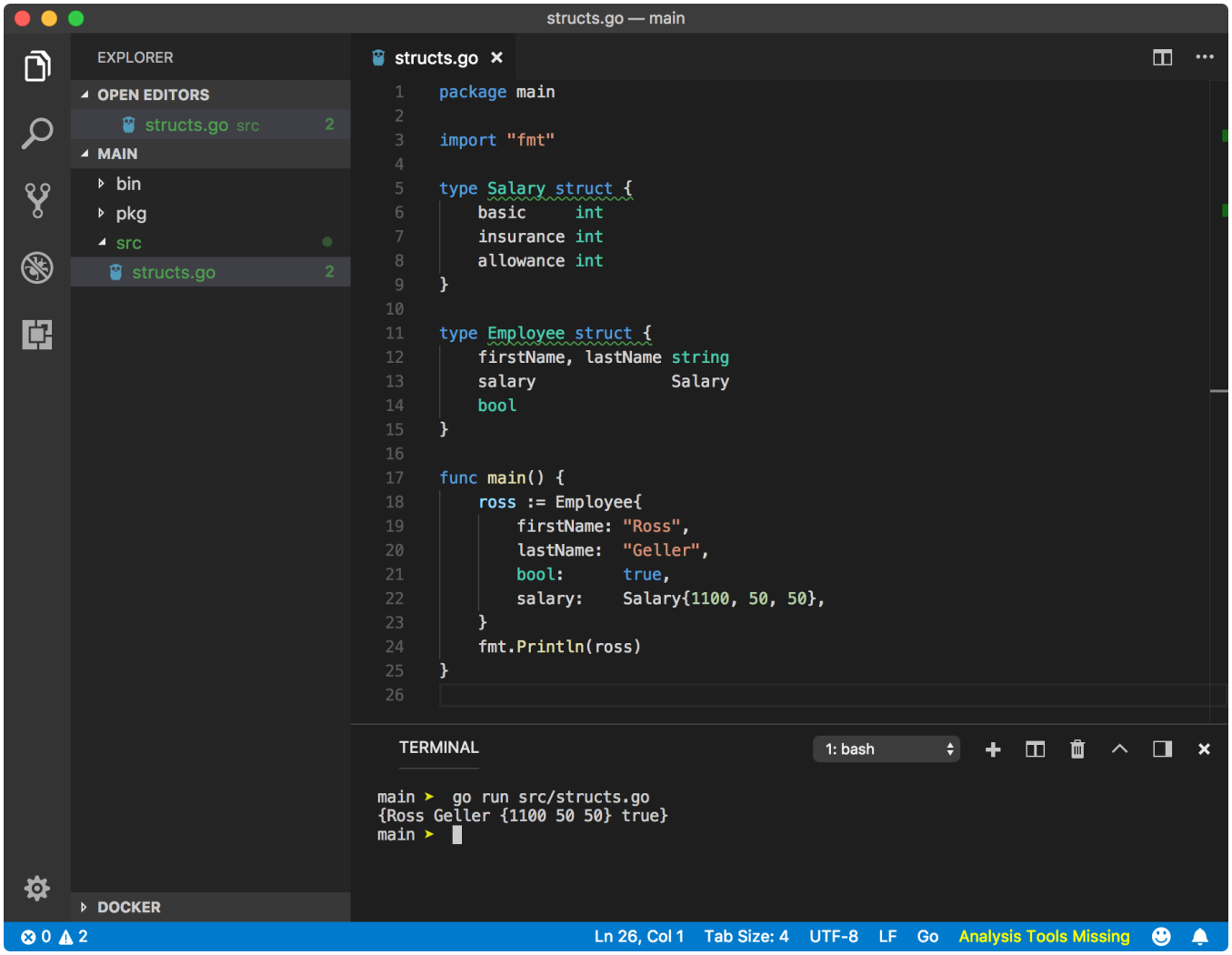
https://play.golang.org/p/oIIlvnrTWdQ

In the above program, we have defined struct type `Data` with 3 fields with field names and types `string`, `int` and `bool`. There is no special case for `Data` struct type when compared to others. Just, in this case, Go helped us creating field names automatically. You are allowed to mix some anonymous fields within named fields like below.

```
type Employee struct {
        firstName, lastName string
        salary              int
        bool
}
```

☛ Nested struct

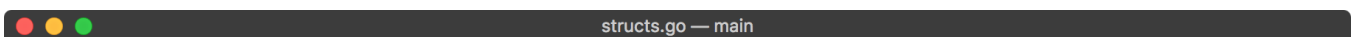
A struct field can also be a struct. A nested struct is a struct that is a field of another struct. Let's see an example

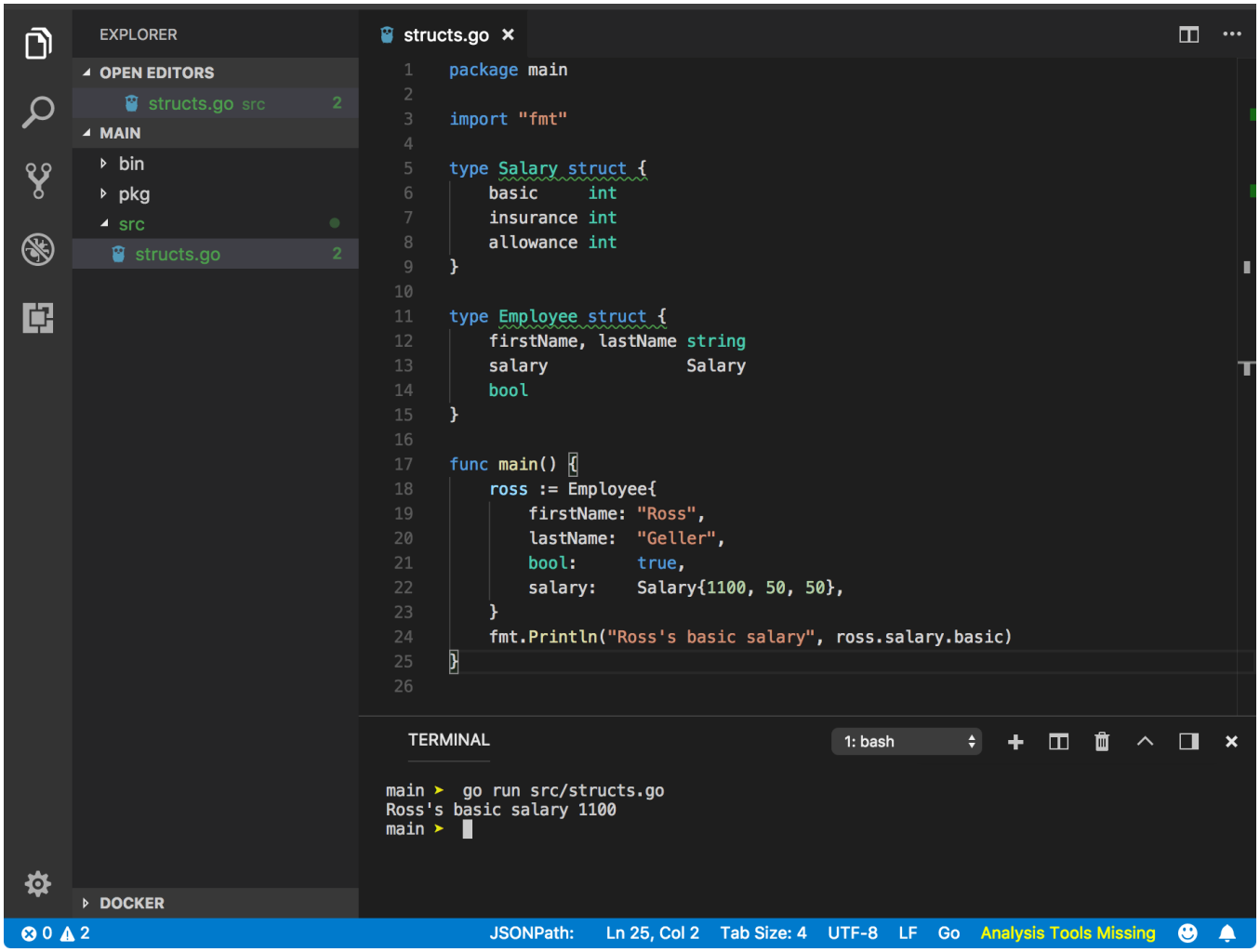


https://play.golang.org/p/uGsHK2ztQ5o

As you can see in the above example, we have created a new struct type `Salary` which defines an employee's salary. Then we modified `Employee` struct type with instead of `int` type, `salary` field is now a struct of type `Salary`. When creating `ross` struct of the type `Employee`, we initialized all fields even `salary` field. We have used the short approach of excluding field names while initializing `salary` struct.

Normally, you would access a field of a struct like `struct.field` as we have seen. You can access `salary` struct in the same manner like `ross.salary` which will give your `salary` struct. You can then access (*or update*) fields of `salary` using the same approach like `ross.salary.<field>`. Let's see that in action.

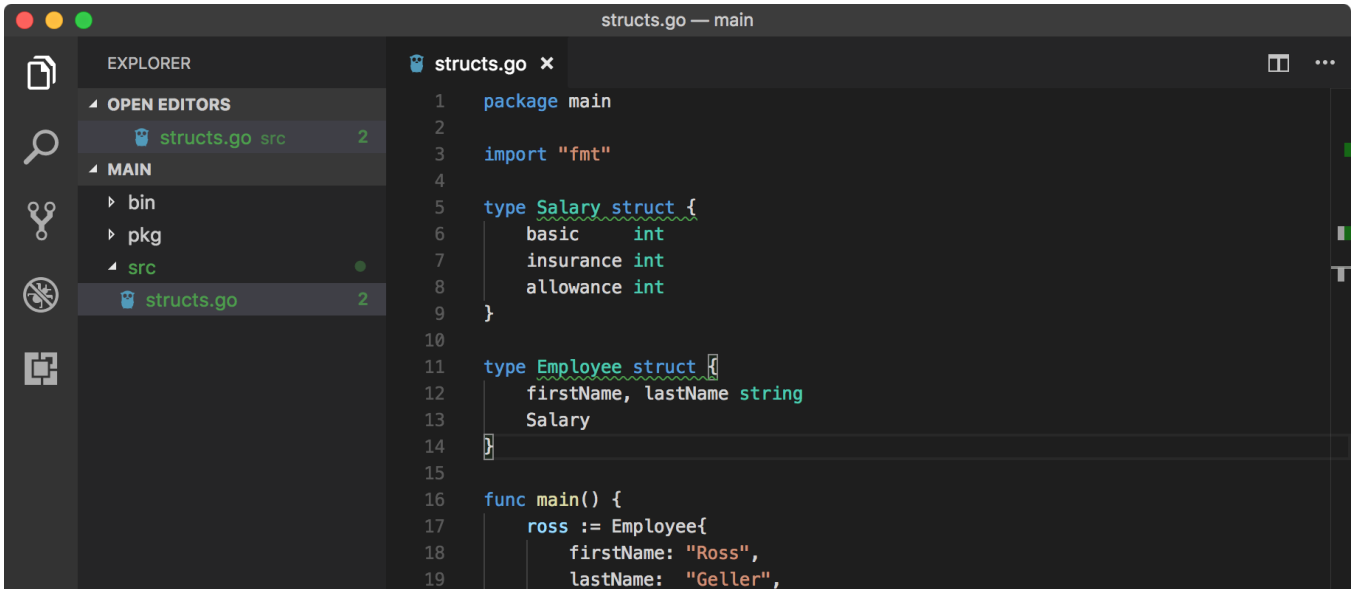


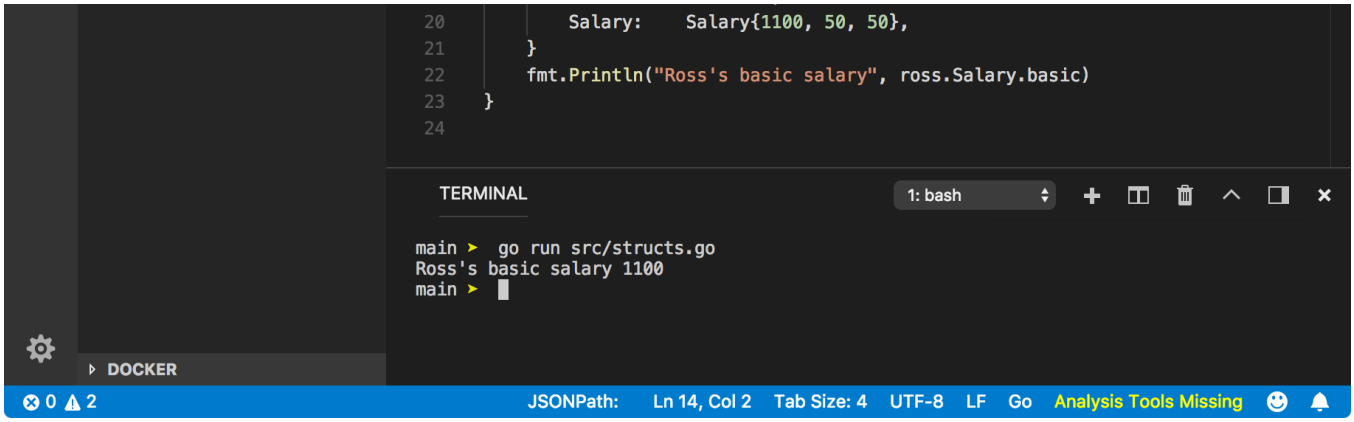


https://play.golang.org/p/jVs385qTwZo

☛ Promoted fields

As we have seen anonymous fields where a type can actually be a field name, we can also use this approach in our nested structs.

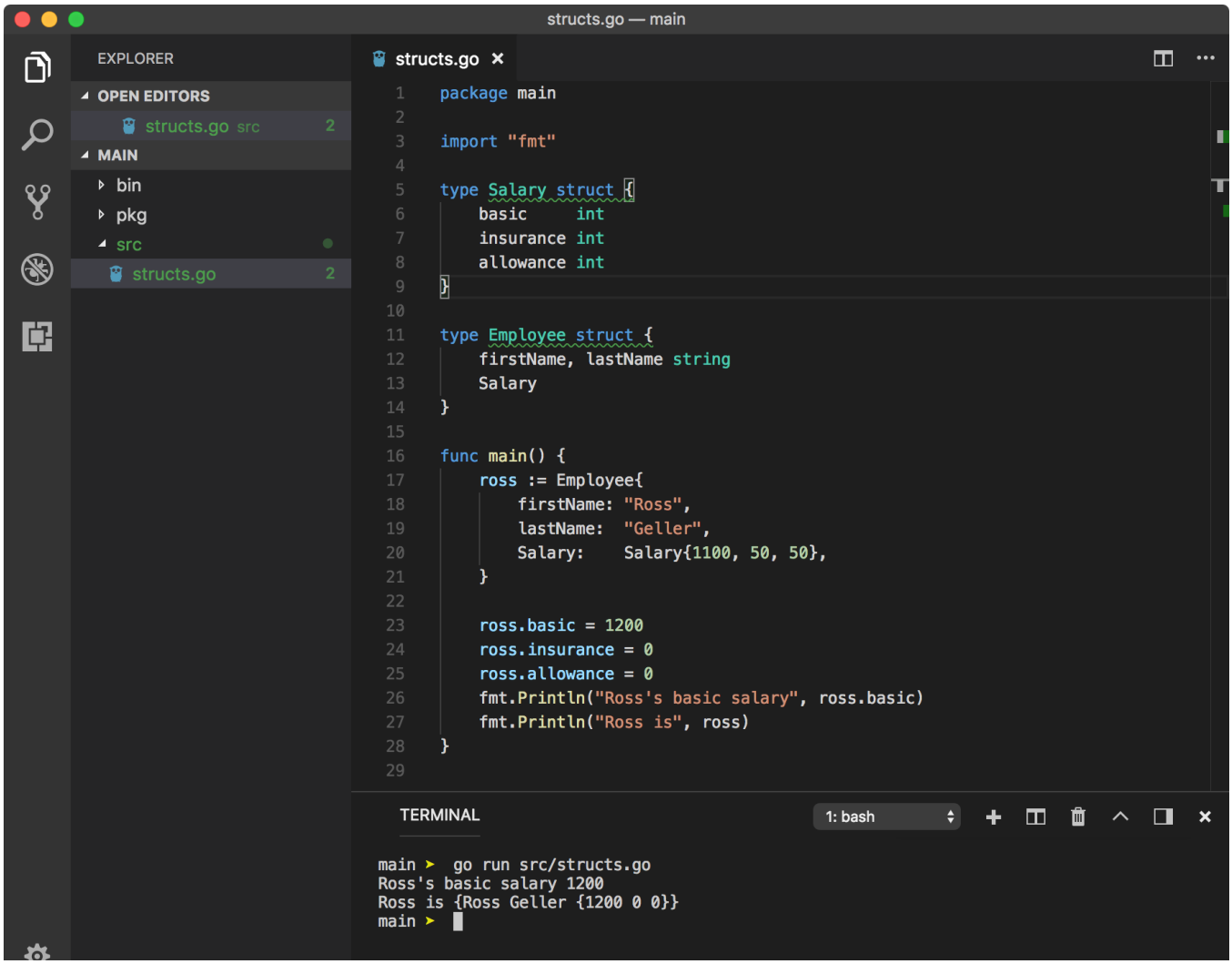




https://play.golang.org/p/wzFIlOZkoqT

Using the previous example of the nested struct, we removed `salary` field name and just used `Salary` struct type to create anonymous fields. Then you can use `.` approach to `get` and `set` nested salary fields.

But the cool thing about Go is when we use an anonymous nested struct, all the nested struct fields are automatically available in parent struct.



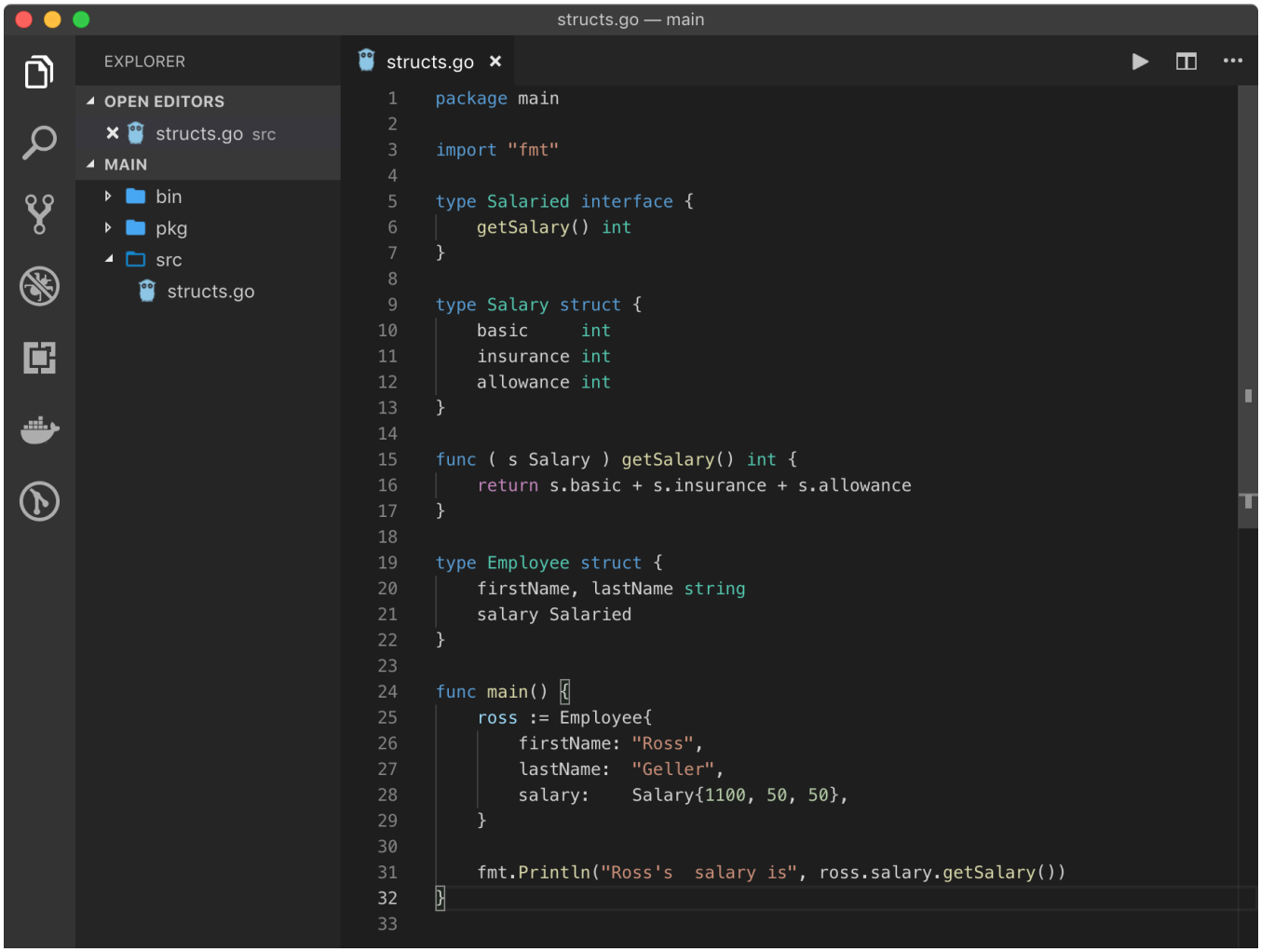
https://play.golang.org/p/XKn6-rKvPg9

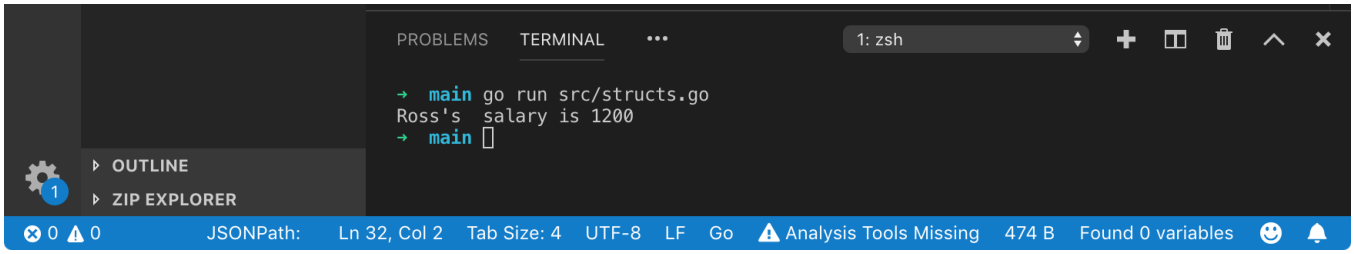
> *If a nested anonymous struct has a same field (field name) as in the parent struct, then inner struct field won't get promoted.* ***Only*** `non-conflicting` ***fields will get promoted.***

☛ Nested interface

Like a struct, an interface can also be nested in a struct. Please read **interfaces** lesson before you read this section. Since we know that, an interface type is a declaration of the method signatures. **Any type that implements an interface can also be represented as a type of that interface (*polymorphism*)**.

Using this polymorphism principle, we can have a struct field of an interface type and its value is anything that implements that interface. Let's see that in action using an earlier example.





https://play.golang.org/p/eV-1Nlx7Jm-

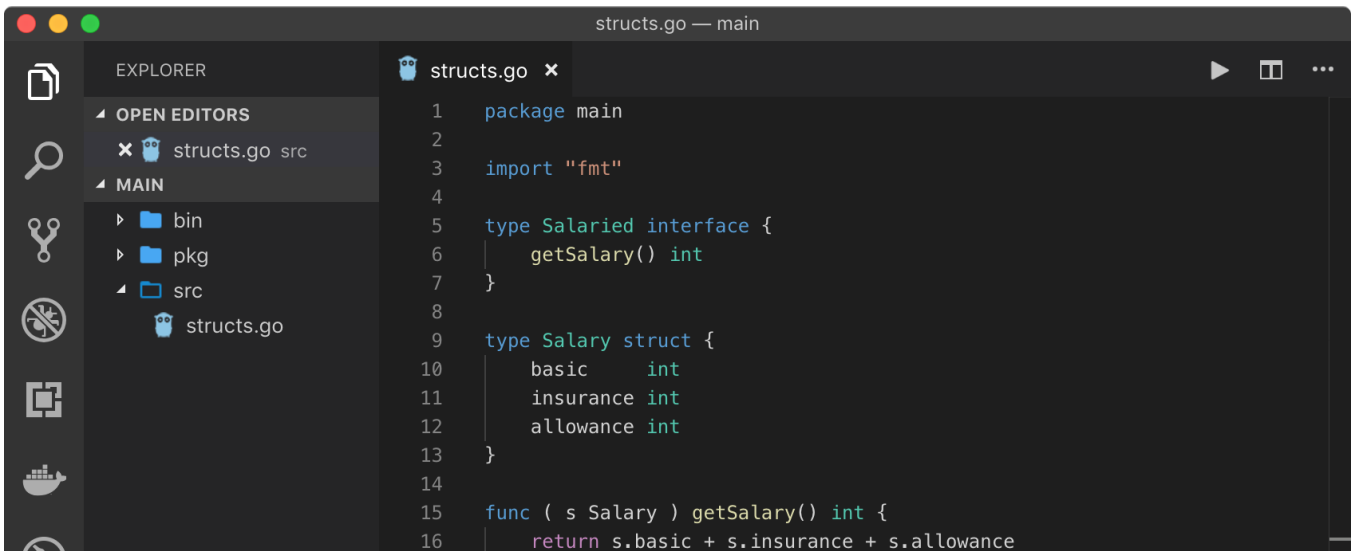
In the above example, we have created `Salaried` interface with `getSalary` method signature. Since `Salary` struct implements this method, it implements `Salaried` interface. Hence, we can store an instance of `Salary` struct type in a variable of `Salaried` type.

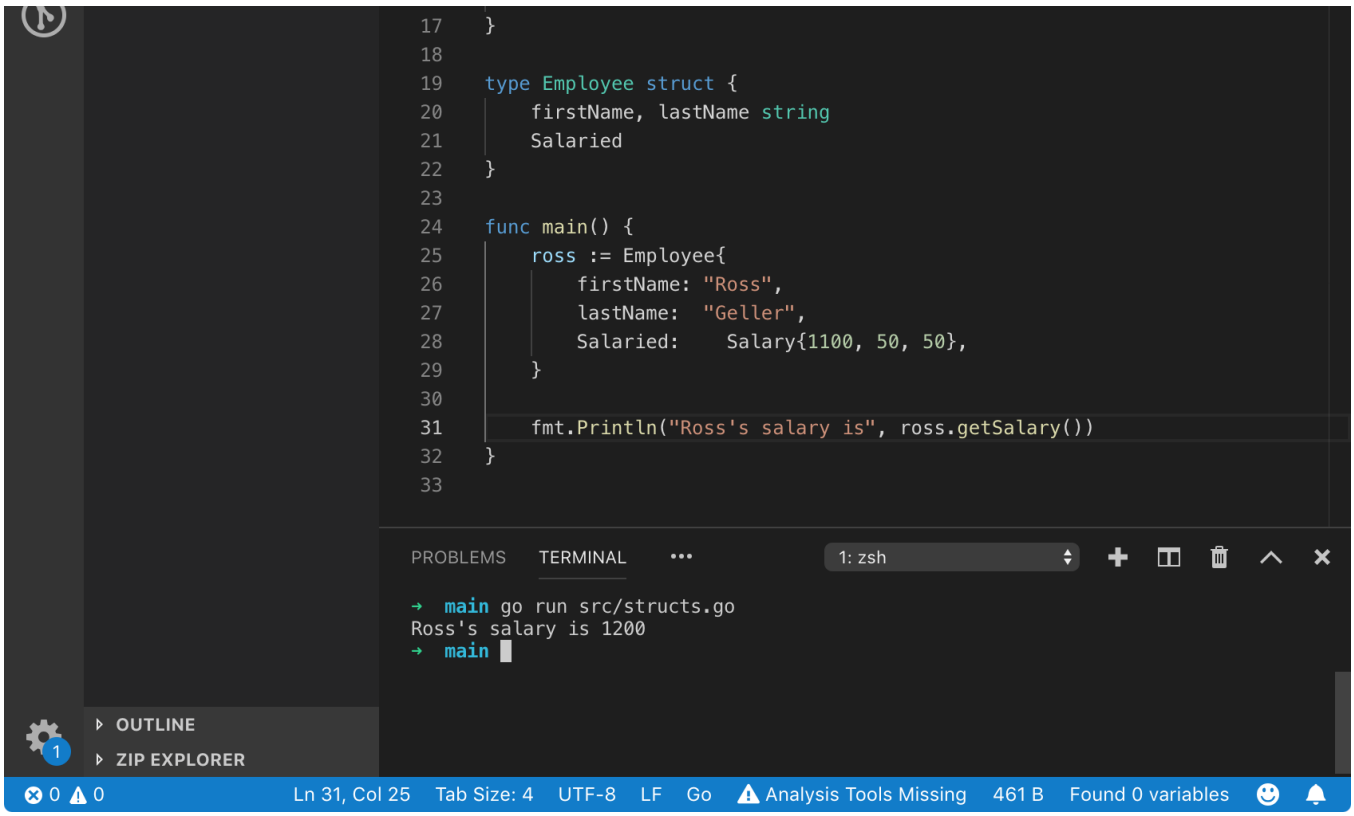
This is what we have done in line no. 28 by assigning an instance of `Salary` struct in `salary` field of `Employee` struct which is of `Salaried` type.

When we call a method on a variable of an interface type, the method will be executed on the **dynamic value** that interface represents, which is at the moment is an instance of `Salary` struct from line no. 28.

> ⚠ *If we did not assign any value to `salary` field while creating an `Employee` struct as we did on line no. 25, Go will panic in a runtime error as we are trying to call a method on a `nil` value which is the default dynamic value of an interface. Hence, try to avoid having a struct field of an interface type.*

Similar to the field promotions we saw earlier when a field is an anonymous struct, methods are promoted when a struct field is an anonymous interface.

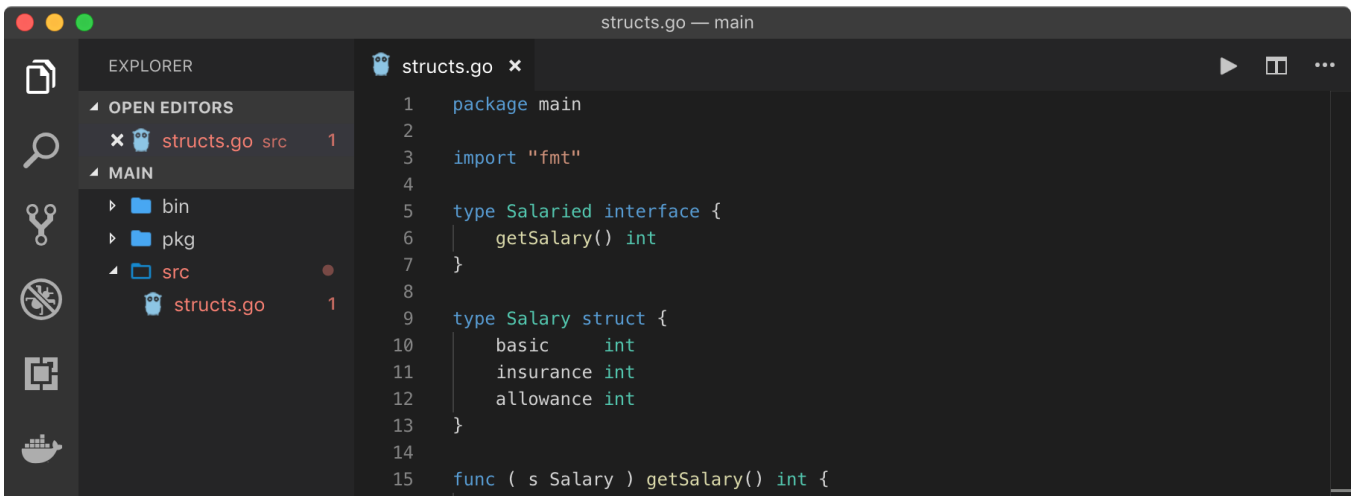


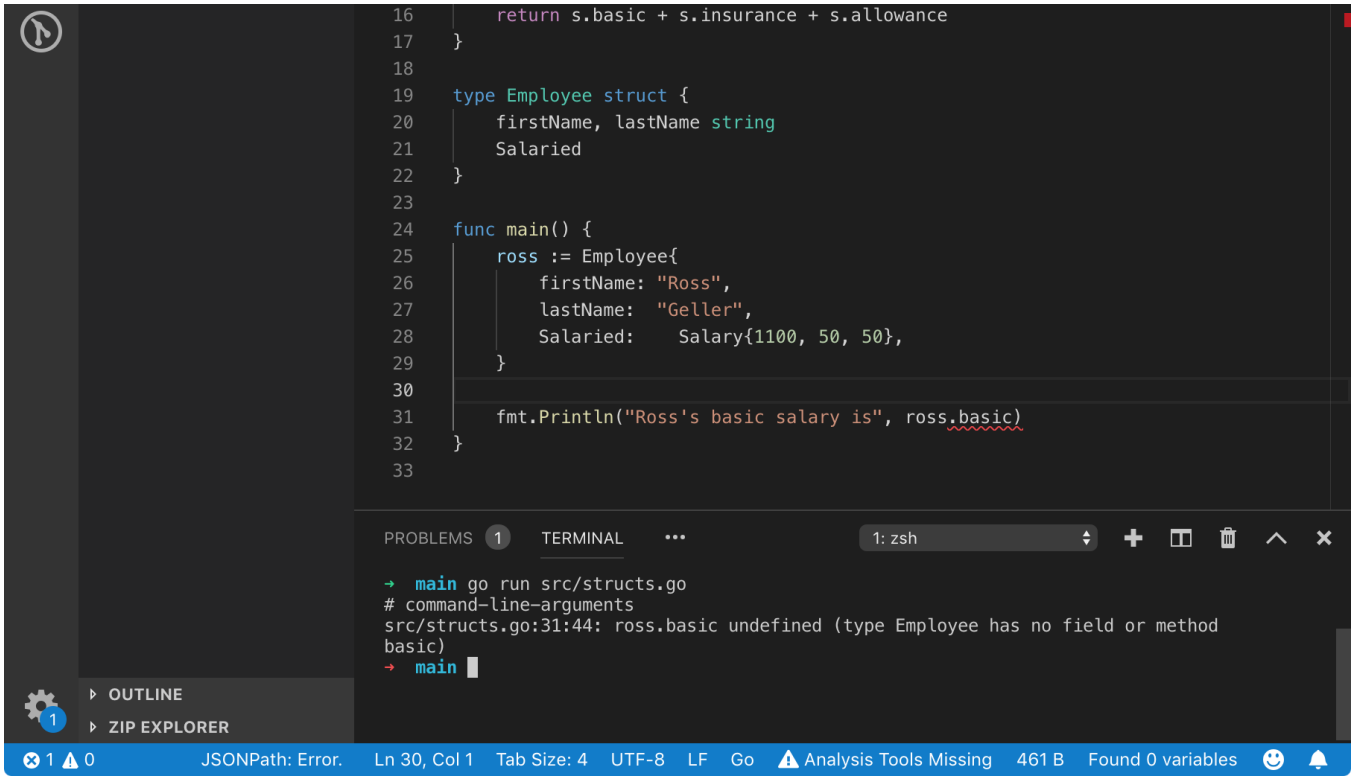


https://play.golang.org/p/XzQcglQw2RM

As we can see in the above example, we renamed `salary` field of `Employee` struct to `Salaried` which is also its type. This will promote all the methods `Salaried` interface provides. Hence, we are able to call the `getSalary` method on `Employee` instance as done on line no. 31.

The most important thing to remember here is, in constant with field promotion as seen earlier in the case when the anomymous field is a struct type, **only methods are promoted when the anomoyous field is an interface type**. This can be verified by the below example.





https://play.golang.org/p/JbwpKbNMsm2

☛ Exported fields

As we have seen in `packages` lesson, any variable or type which starts from the capital letter is exported from that package. In the case of a struct, we made sure that all our structs used in this lesson are exported, hence they start with capital letter viz. `Employee`, `Salary`, `Data` etc. But the really cool thing about struct is, we can also control which fields of a struct are exported. This follows the same `uppercase letter` approach.

```
type Employee struct {
    FirstName, LastName string
    salary int
    fullTime bool
}
```

In the above struct type `Employee`, `FirstName` and `LastName` are only two fields which are exported.

Let's create a simple package organization with the package name `org`. We can create a file `WORKSPACE/src/org/employee.go` and place the following code

```
// employee.go
package org

type Employee struct {
        FirstName, LastName string
        salary              int
        fullTime            bool
}
```

In the main package, we can import `Employee` struct like below.

```
// main.go
package main

import (
        "fmt"
        "org"
)

func main() {
        ross := org.Employee{
                FirstName: "Ross",
                LastName:  "Geller",
                salary:    1200,
        }

        fmt.Println(ross)
}
```

Above program won't compile and the compiler will throw below error.

```
unknown field 'salary' in struct literal of type org.Employee
```

This happens because `salary` field is not exported. We need to use `org.Employee` as struct type as `Employee` type came from `org` package. We could also create type alias in the main package to make it simpler.

```
// main.go
package main
```

```
import (
        "fmt"
        "org"
)

type Employee org.Employee

func main() {
        ross := Employee{
                FirstName: "Ross",
                LastName:  "Geller",
        }

        fmt.Println(ross)
}
```

Above program yields below result

```
{Ross Geller 0 false}
```

Looks weird? Because we did not expect values of `salary` and `fullTime` fields. When we import any struct from another package, we get struct as is, just that we don't have any control over them. This is useful when you want to **protect some fields but still make them useful for an outsider**.

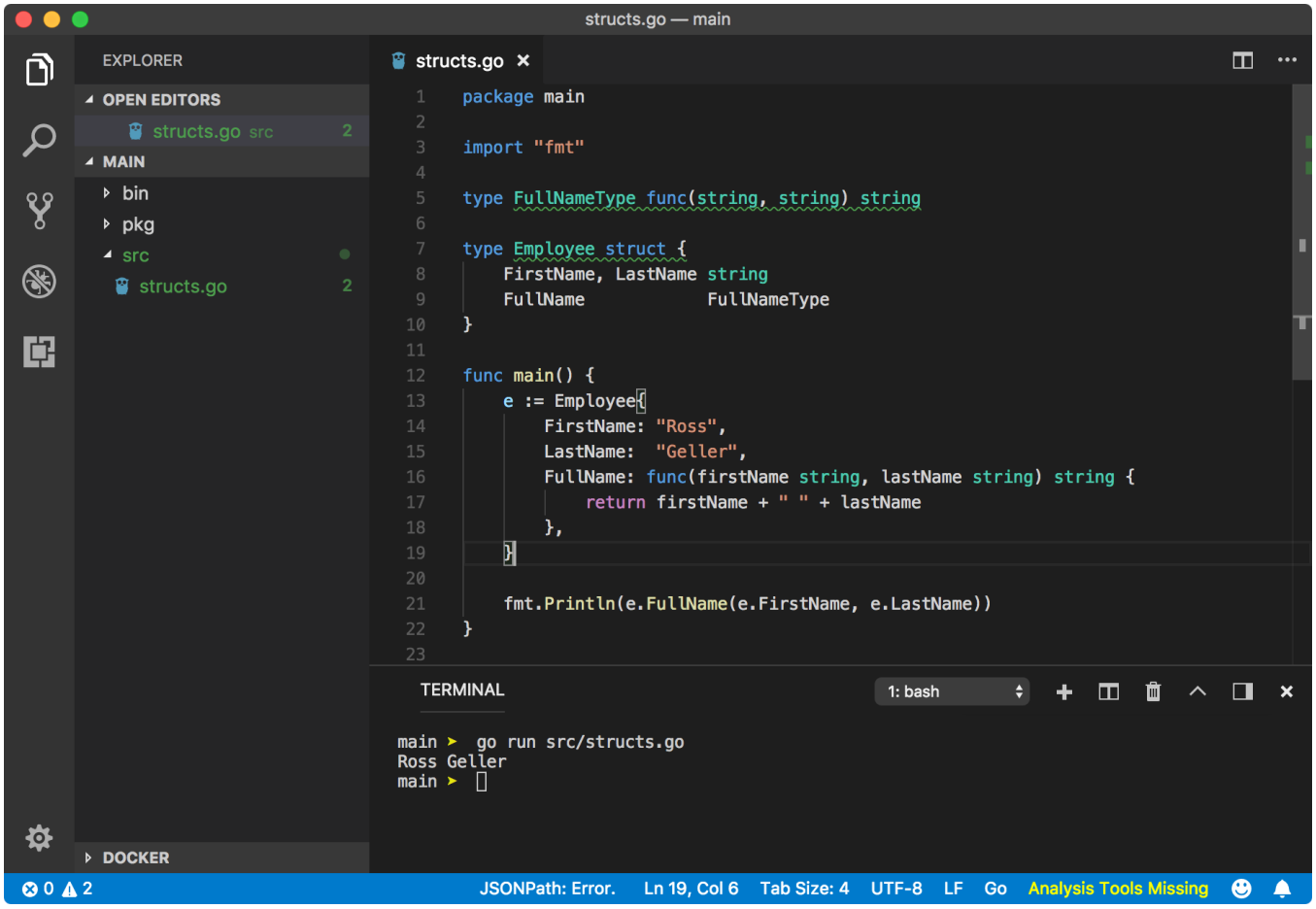
What will happen in case of a nested struct?

- A nested struct must also be declared with an uppercase letter so that other packages can import it.
- Nested struct fields starting with an uppercase letter are exported.
- If a nested struct is anonymous, then its fields starting with an uppercase letter will be available as promoted fields.

☛ Function fields

If you remember `function as type` and `function as value` topics from `functions` lesson, you can pretty much guess that `struct` fields can also be functions.

So let's create a simple struct function field which **returns the full name of an employee**.



https://play.golang.org/p/U73YGQcWkwa

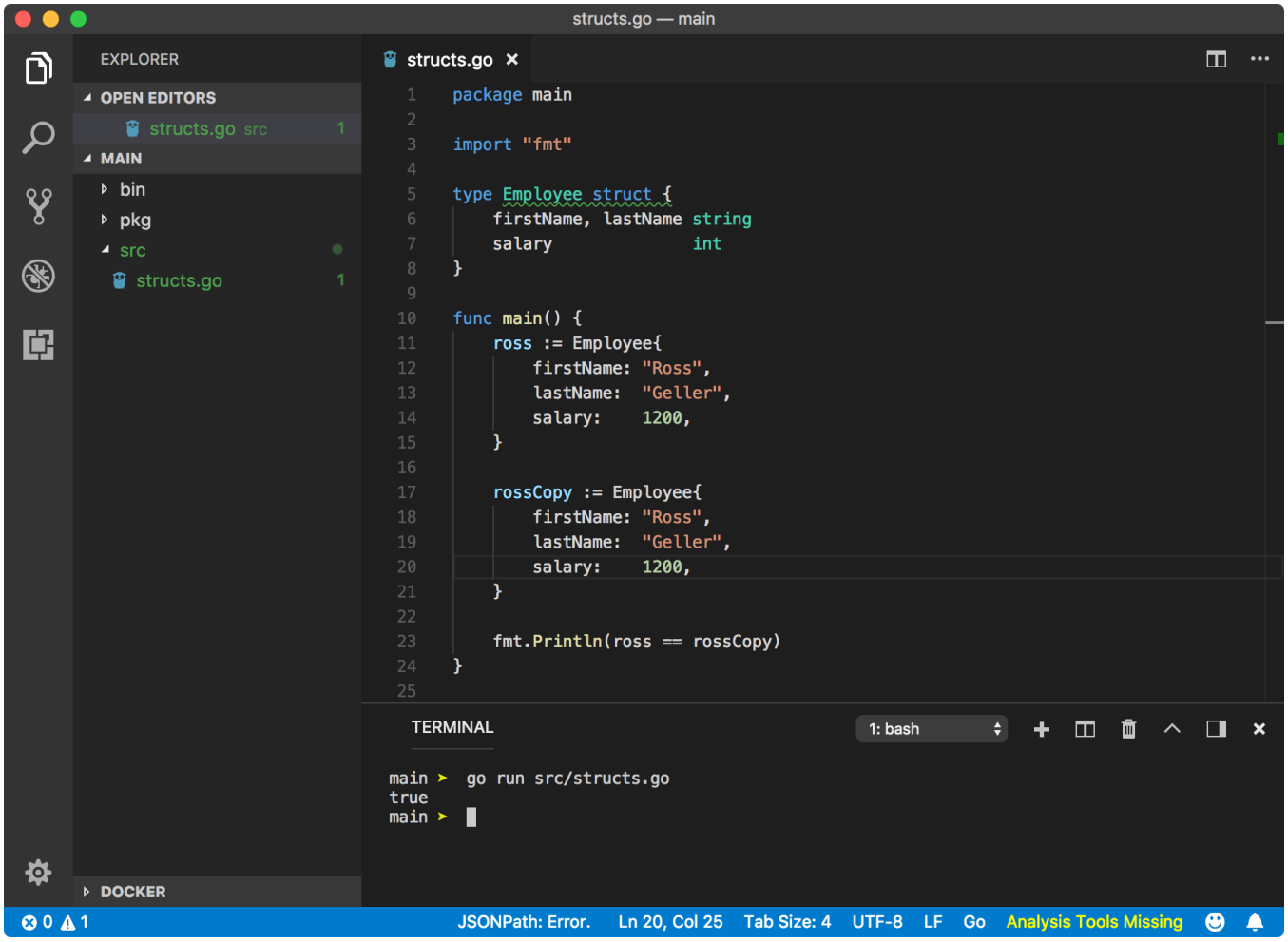
In the above program, we have defined struct type `Employee` which has two string fields and one function field. Just for simplicity, we have created a function type alias `FullNameType` to function type which takes two strings and return one string. While creating a struct `e`, we need to make sure the field `FullName` follows the function type syntax. In the above case, we assigned it with an anonymous function.

Then we simply executed the function `e.FullName` with two string arguments `e.FirstName` and `e.LastName`.

> *In case you are wondering, why we need to pass properties of `e` (viz. `e.FirstName` and `e.LastName`) to `e.FullName` as `FullName` field belongs to the same struct `e`, then you need to see `methods` lesson (upcoming).*

☛ struct comparison

Two structs are comparable if they belong to the same type and have the same field values.



https://play.golang.org/p/AFkN-AxDSk5

Above program prints `true` as both `ross` and `rossCopy` belongs to the same struct type `Employee` and have the same set of values.

However, if a struct has field type which is not comparable, for example, `map` which is not comparable, then the struct won't be comparable. For example, if `Employee` struct type had `leaves` field of type `map`, we could not do the above comparison.

```
type Employee struct {
        firstName, lastName string
        salary              int
        leaves              map[string]int
}
```

☛ struct field meta-data

Struct gives one more ability to add meta-data to the fields. Usually, it is used to provide transformation info on how a struct field is encoded to or decoded from another format (*or stored/retrieved from a database*), but you can use it to store whatever meta-info you want to, either intended for another package or for your own use.

This meta information is defined by string literal (*read* `strings` *lesson*) like below

```
type Employee struct {
        firstName string `json:"firstName"`
        lastName  string `json:"lastName"`
        salary    int    `json: "salary"`
        fullTime  int    `json: "fullTime"`
}
```

In the above struct, we are using struct type `Employee` for JSON encoding/decoding purpose. You will learn more struct meta-data in upcoming lessons like `json`, `validation` and others.