Anatomy of methods in Go

Since there is no class-object architecture and the closest thing to class we have is structure. Function with struct receiver is a way to achieve methods in Go.

Uday Hiwarale Follow
Oct 20, 2018 · 10 min read

Since we don't have classes in Go, you can say `struct` will do a job to make objects. But in OOP, classes have properties (*fields*) and behaviors (*methods*) and so far we have seen only properties of a struct.

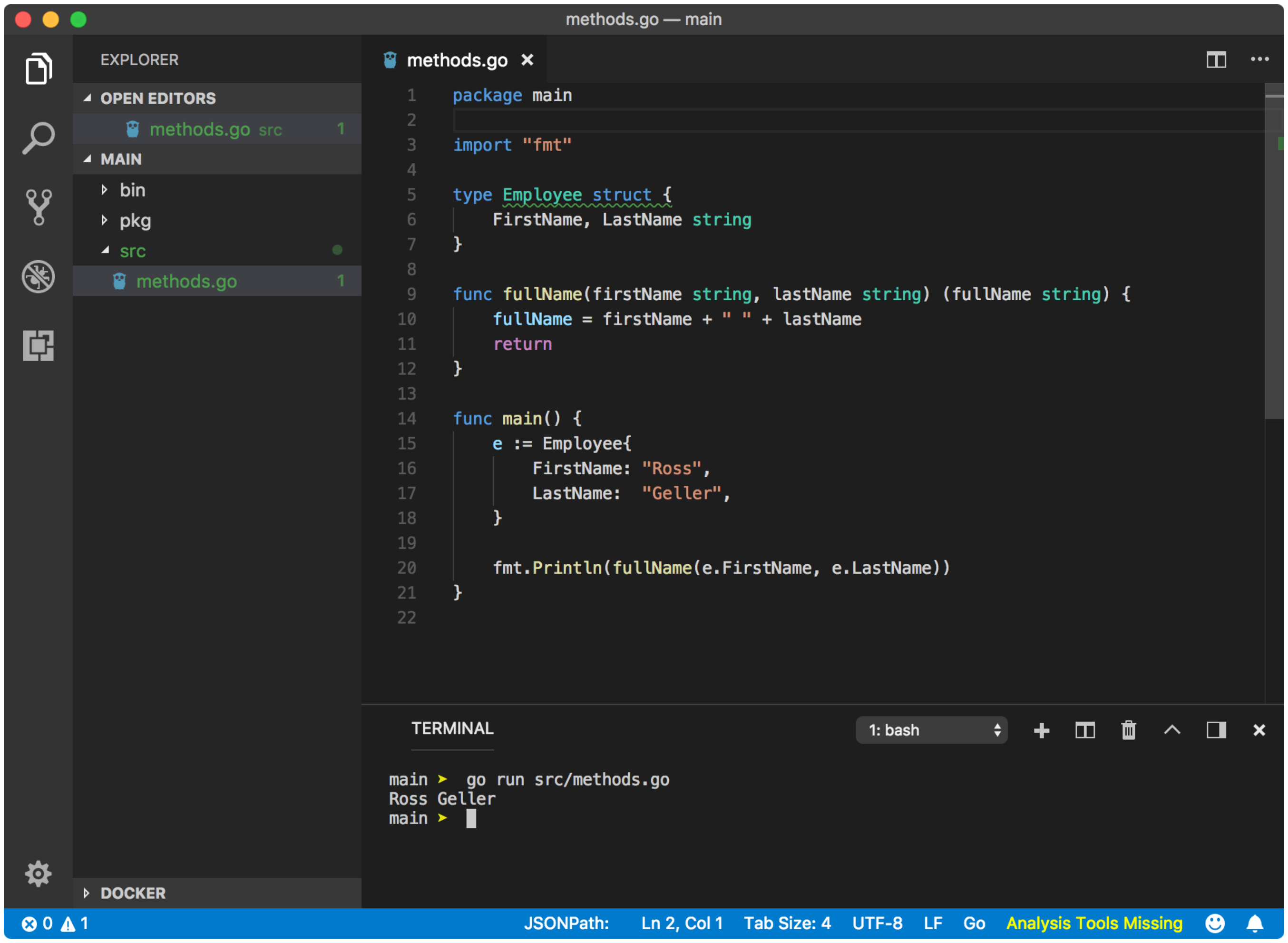
> *To all those non-OOP programmers, behavior is a action that an object can perform. For example,* `Dog` *is a type of* `Animal` *and* `Dog` *can* `bark`*. Hence barking is a behavior of animal class* `Dog`*. Hence any objects (instances) of class* `Dog` *will have this behavior.*

We have seen in structures lesson, especially in the **function field** section that, **struct field can also be a function**. We can add a `bark` field of type function which takes no arguments and returns a string `woof woof!`. But this does not adhere to the OOP concept as `struct` **fields do not have any idea of `struct` they belong to**. Hence methods come to the rescue.

☛ What is a method

As you know that struct field can also be a function, the concept of **method** will be very easy for you to understand. A method is nothing but a function, but it belongs to a certain type. A method is defined with different syntax than normal function. It required an additional parameter known as a **receiver** which is a type to which the method belongs. A method can access properties of the receiver it belongs to.

Let's see a simple program to get the full name of an `Employee` using a simple function.



```go
package main

import "fmt"

type Employee struct {
    FirstName, LastName string
}

func fullName(firstName string, lastName string) (fullName string) {
    fullName = firstName + " " + lastName
    return
}

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

    fmt.Println(fullName(e.FirstName, e.LastName))
}
```

```
main ➤ go run src/methods.go
Ross Geller
main ➤
```

https://play.golang.org/p/ANE_IN9cmk4

In the above program, we have created a simple struct type `Employee` which has two string fields `FirstName` and `LastName`. Then we defined the function `fullName` which takes two strings and returns a string. `fullname` function returns the full name of an employee by concatenating these two strings. Then we created a struct `e` and used `fullName` function to get the full name of the employee `e`.

But the sad thing is, every time we need to get the full name of an employee, we need to pass `FirstName` and `LastName` to `fullName` function manually.

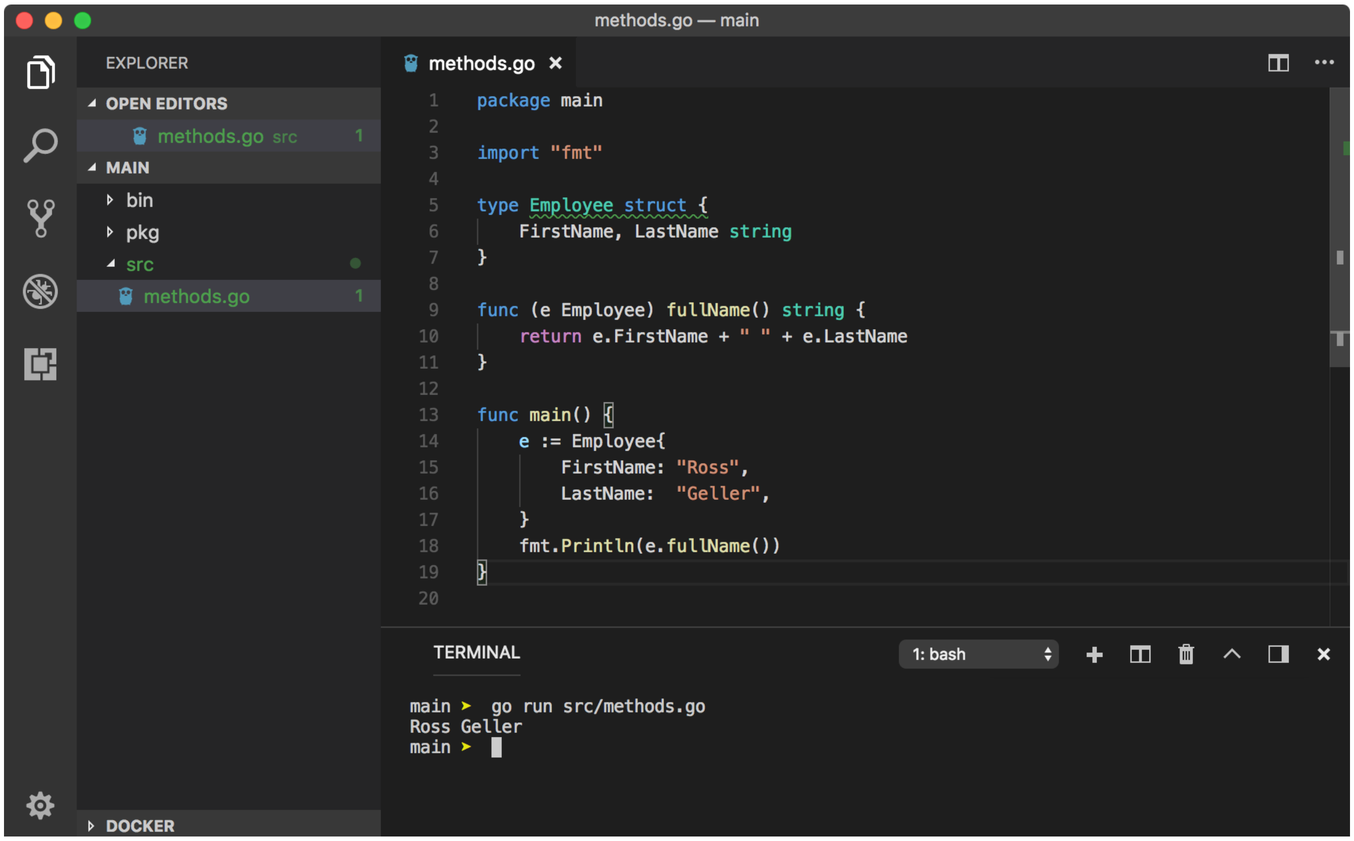
The **method** solves this problem easily. As we talked, we just need an extra receiver parameter in the function definition.

The syntax for defining a method is

```
func (r Type) functionName(...Type) Type {
    ...
}
```

From the above syntax, we can tell that method and function have the same syntax except for one input argument declaration `(r Type)` before the function name. `Type` is any legal type in Go and function arguments and return value are optional.

Let's create `fullName` method using the above syntax.



```go
package main

import "fmt"

type Employee struct {
    FirstName, LastName string
}

func (e Employee) fullName() string {
    return e.FirstName + " " + e.LastName
}

func main() {
    e := Employee{
        FirstName: "Ross",
        LastName:  "Geller",
    }
    fmt.Println(e.fullName())
}
```

```
main ➤  go run src/methods.go
Ross Geller
main ➤
```

https://play.golang.org/p/68JWFHO9Yep

In the above program, we have defined `fullName` method which does not take any arguments but returns a string. It belongs to a struct type `Employee` hence we used the argument `e` which will act as a receiver for `Employee` struct.

Receiver in the method is accessible inside the method body. Hence we can access `e` inside the method body. Using that, we can access any fields of struct. In the above program, we are concatenating `FirstName` and `LastName` and returning as the full name of the employee.

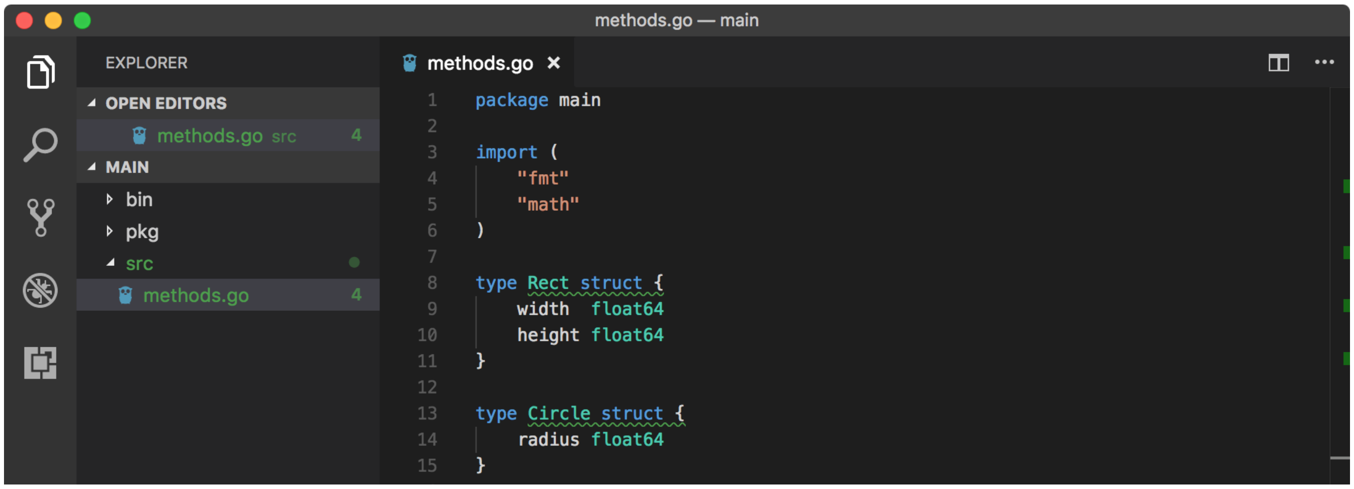
As a method belongs to a receiver type, we can call that method using `Type.methodName(...)` syntax. In the above program, we have used `e.fullName()` to get the full name of an employee as `fullName` method belongs to `e`.

> *This is no different than what we learned in `structs` lesson where `fullName` function was a field of struct. But in case of methods, we don't have to provide properties of struct because method already knows about them.*

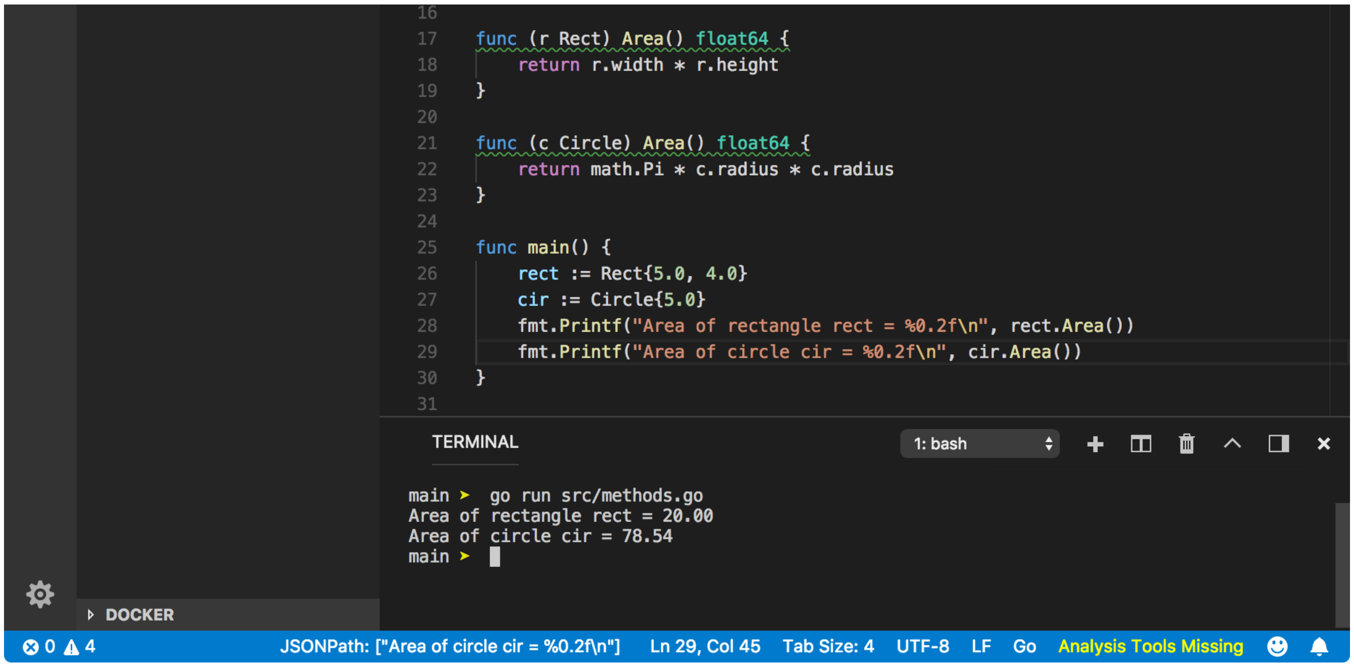
☛ Same method name

One major difference between function and method is many methods can have the same name while no two functions with the same name can be defined in a package.

Let's create two struct types `Circle` and `Rectangle` and create two methods of the same name `Area` which calculates the area of their receiver.



```go
package main

import (
    "fmt"
    "math"
)

type Rect struct {
    width  float64
    height float64
}

type Circle struct {
    radius float64
}
```



```go
func (r Rect) Area() float64 {
    return r.width * r.height
}

func (c Circle) Area() float64 {
    return math.Pi * c.radius * c.radius
}

func main() {
    rect := Rect{5.0, 4.0}
    cir := Circle{5.0}
    fmt.Printf("Area of rectangle rect = %0.2f\n", rect.Area())
    fmt.Printf("Area of circle cir = %0.2f\n", cir.Area())
}
```

```
main ➤ go run src/methods.go
Area of rectangle rect = 20.00
Area of circle cir = 78.54
main ➤
```

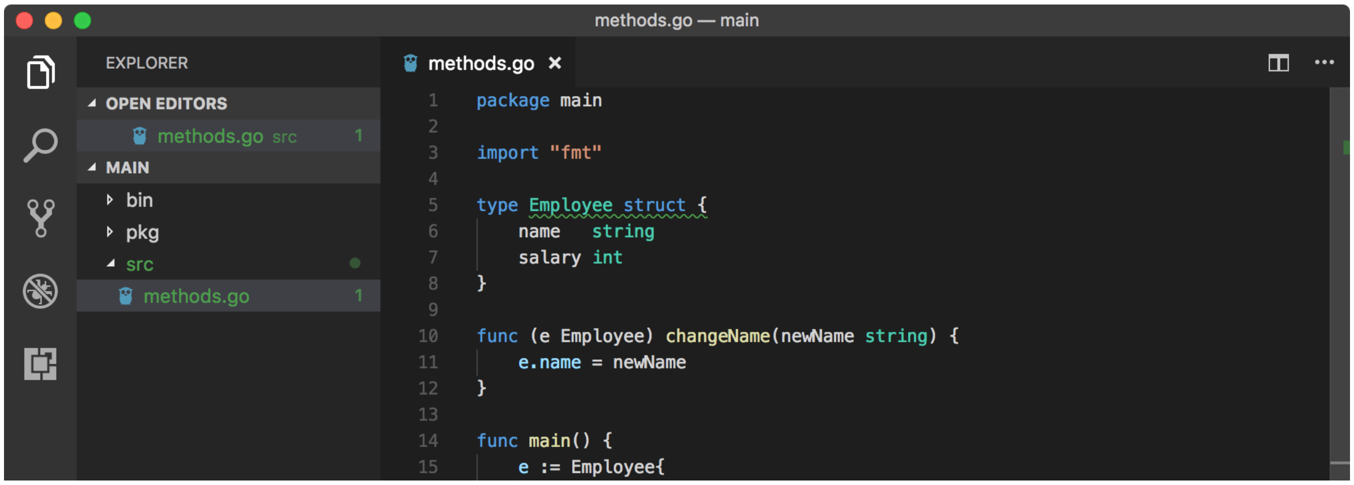
https://play.golang.org/p/3f_MScaNXUw

In the above program, we have created struct types `Rect` and `Circle` and created two methods of the same name `Area` with receiver type `Rect` and `Circle`. We are allowed to use same method names as long as method receivers are of different types. When we call `Area` method on `Rect` and `Circle`, their respective methods get executed.

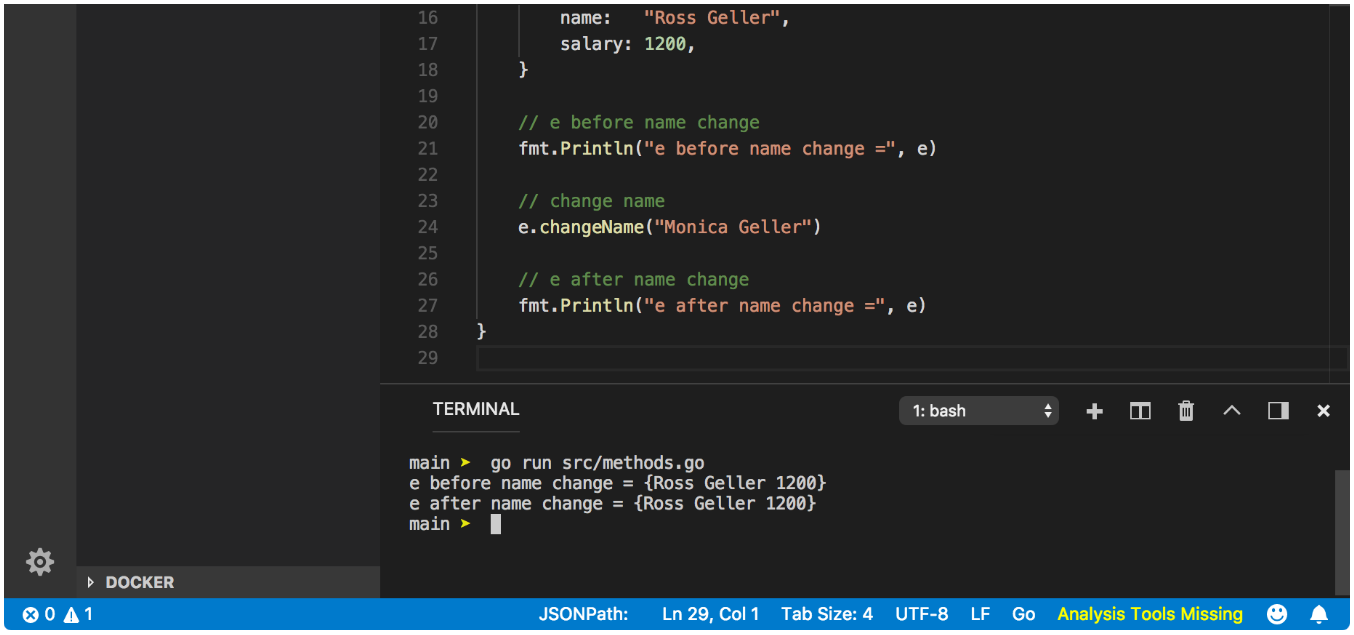
☛ Pointer receivers

Until now we have seen methods which receive the value of the receiver that means methods only receives a copy of the **Type** they belong to, for example, a struct in previous examples.

To verify that, we can create a method that changes the value of a struct field. Let's create a method name `changeName` that changes the name of Employee struct.



```go
package main

import "fmt"

type Employee struct {
    name   string
    salary int
}

func (e Employee) changeName(newName string) {
    e.name = newName
}

func main() {
    e := Employee{
```



```go
        name:   "Ross Geller",
        salary: 1200,
    }

    // e before name change
    fmt.Println("e before name change =", e)

    // change name
    e.changeName("Monica Geller")

    // e after name change
    fmt.Println("e after name change =", e)
}
```

```
main ➤  go run src/methods.go
e before name change = {Ross Geller 1200}
e after name change = {Ross Geller 1200}
main ➤
```

https://play.golang.org/p/cV-W27SGcnl

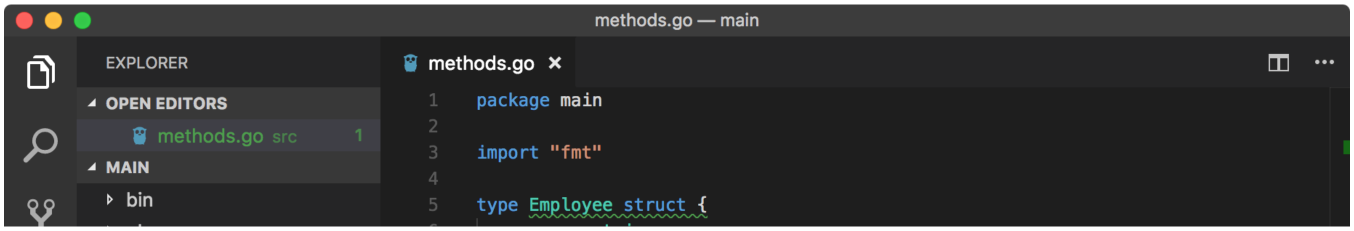
In the above program, we have called method `changeName` on struct `e` of type `Employee` while in the method, we are assigning a new value to the name field.

From the above result, it seems like the method `changeName` did not do anything with `name` property of `e`. This proves that receiver `e` in method definition was just a copy of the actual struct `e` (*from* `main` *method*), hence any changes made to did not affect the original struct.

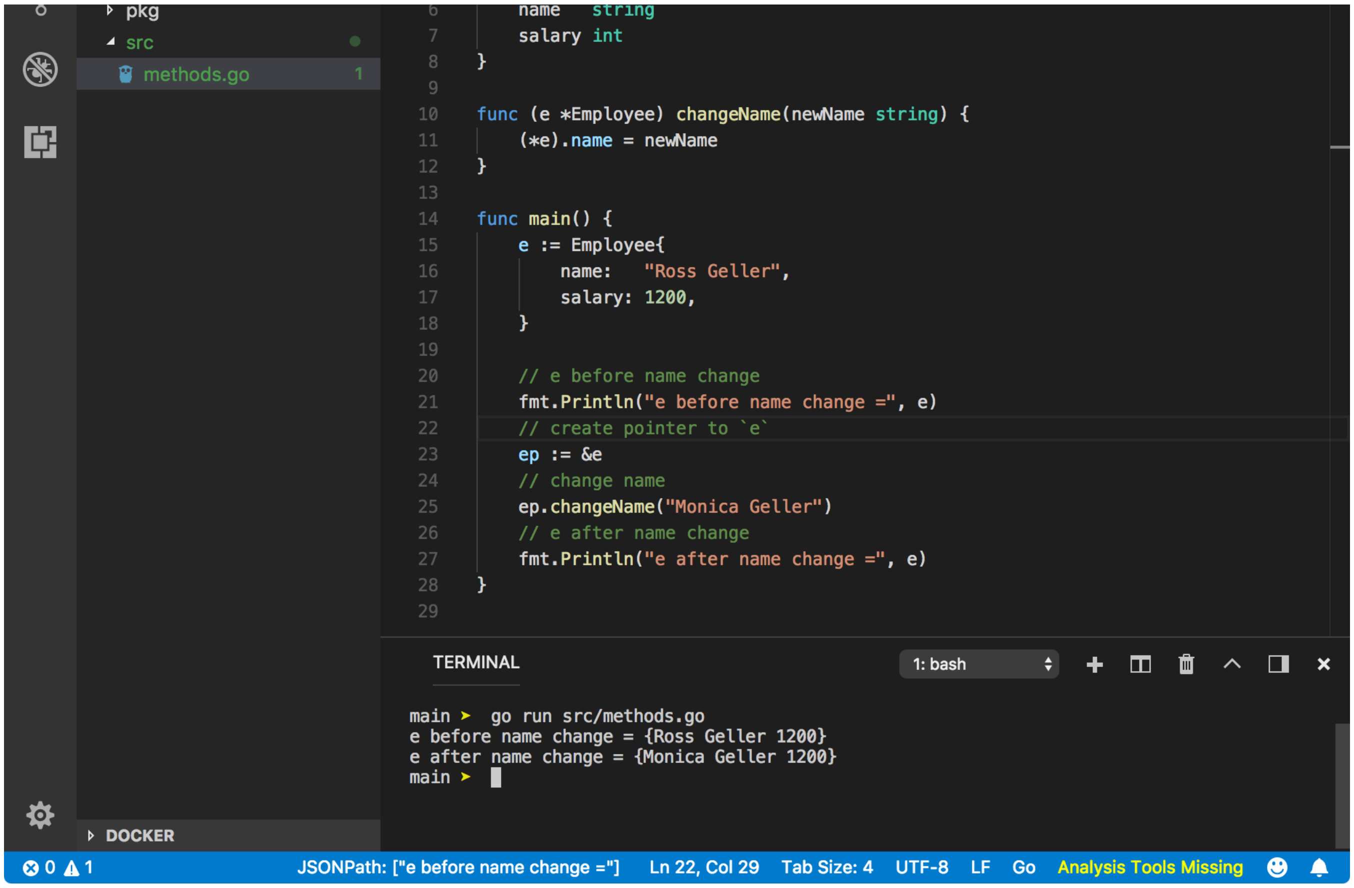
But a method can also accept `pointer` value of the receiver. Syntax to define a method with `pointer` receiver is very similar to the normal method. In the below definition, we instructed Go that this method will receive a pointer receiver.

```go
func (r *Type) methodName(...Type) Type {
    ...
}
```

Let’s re-write the previous example with a method that receives pointer receiver.



```go
package main

import "fmt"

type Employee struct {
```



```go
    name   string
    salary int
}

func (e *Employee) changeName(newName string) {
    (*e).name = newName
}

func main() {
    e := Employee{
        name:   "Ross Geller",
        salary: 1200,
    }

    // e before name change
    fmt.Println("e before name change =", e)
    // create pointer to `e`
    ep := &e
    // change name
    ep.changeName("Monica Geller")
    // e after name change
    fmt.Println("e after name change =", e)
}
```

```
main ➤  go run src/methods.go
e before name change = {Ross Geller 1200}
e after name change = {Monica Geller 1200}
main ➤
```

pkg
src
methods.go
DOCKER
TERMINAL
1: bash
JSONPath: ["e before name change ="] Ln 22, Col 29 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

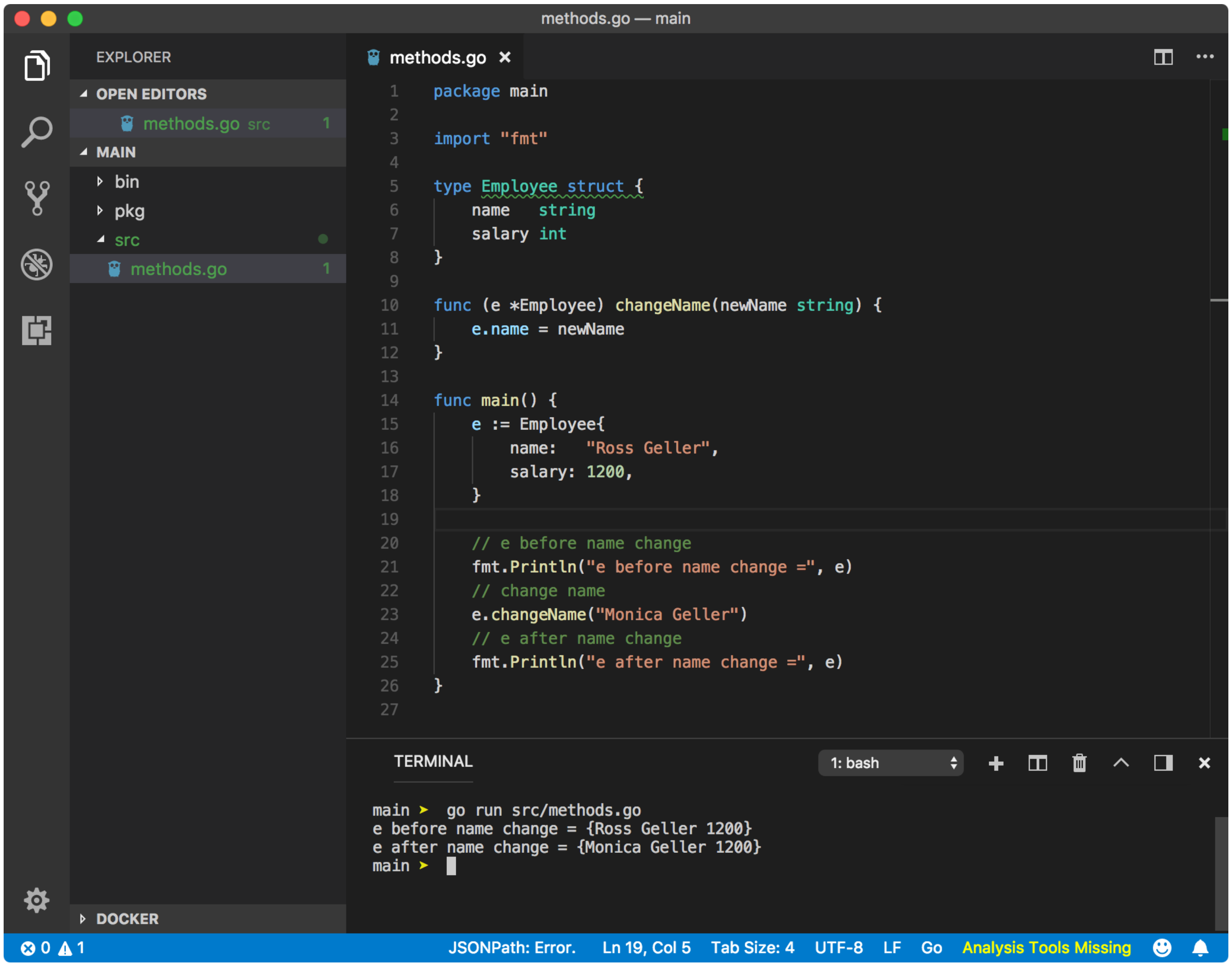
https://play.golang.org/p/PfSQ_GP-GPN

Let's see what changes we made.

- We changed the definition of the method to receive a pointer receiver using `*`. Now method `changeName` will receive pointer of the receiver, that means original value. Inside method body, we are converting pointer of the receiver to the value of the receiver using `*`, hence `(*e)` will be the actual value stored in the memory. Hence any change made on it will be reflected in the original value of the receiver struct.
- Then we create a pointer `ep` which points to struct `e`.
- While calling `changeName` method on struct `e`, we need to call it on the pointer of it which will be `ep`. Since method belongs to pointer of `e` rather than the value of `e`. This will pass the pointer `ep` to the method instead of value `e`.

> *In above program, we created pointer `ep` to call method on it, but you can use `(&e).changeName("Monica Geller")` syntax instead of creating new pointer.*

This all sound little complex. But don't worry, Go makes it simple by providing some shortcuts. Let's rewrite above programming using Go's shortcuts.



```go
package main

import "fmt"

type Employee struct {
    name   string
    salary int
}

func (e *Employee) changeName(newName string) {
    e.name = newName
}

func main() {
    e := Employee{
        name:   "Ross Geller",
        salary: 1200,
    }

    // e before name change
    fmt.Println("e before name change =", e)
    // change name
    e.changeName("Monica Geller")
    // e after name change
    fmt.Println("e after name change =", e)
}
```

```
main ➤  go run src/methods.go
e before name change = {Ross Geller 1200}
e after name change = {Monica Geller 1200}
main ➤
```

https://play.golang.org/p/D2zm7lpSme4

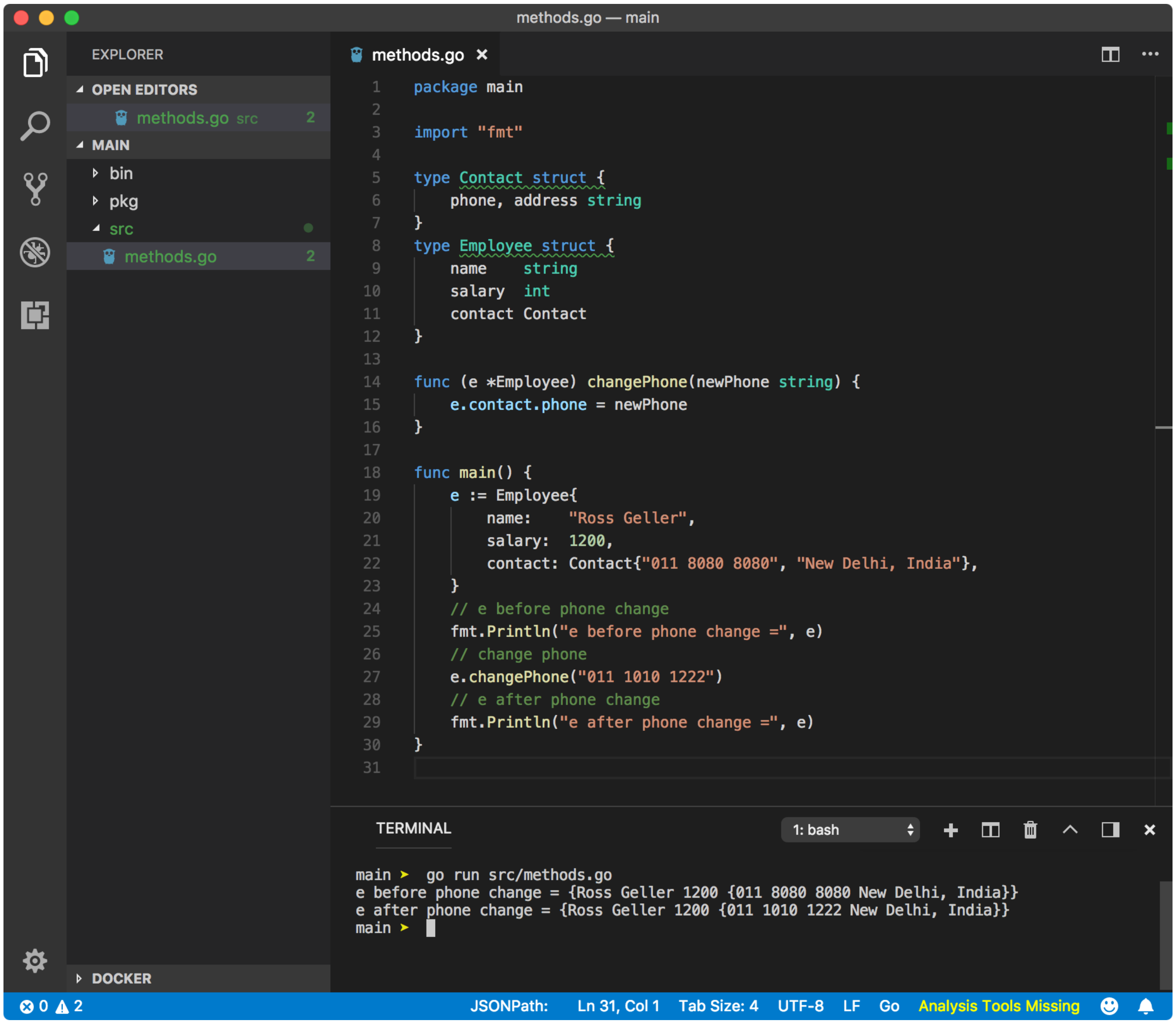
Above program will work just fine like before. So what changed.

- If a method receives a pointer receiver, you don't need to using `(*e)` syntax to deference pointer or get the value of the pointer. You can use simple `e` which will be the address of the value that pointer points to but Go will understand that you are trying to perform an operation on the value itself and under the hood, it will make `e` to `(*e)`.
- Also, you don't need to call a method on the pointer if the method receives a pointer receiver. Instead, you can use value as it is, `e` and call a method on it. Go will pass the pointer of `e` under the hood if the method expects pointer receiver.

> *You can decide between method with pointer receiver or value receiver depending on your use case. But generally, even if you do not wish to change data of the receiver, methods with pointer receiver are used as no new memory is created for operations. Which happens when passing copy of the receiver, which happens in case of methods with value receiver.*

☛ Methods on nested struct

As a struct field also can be a struct, we can define a method on main struct and access nested struct to do anything we want.

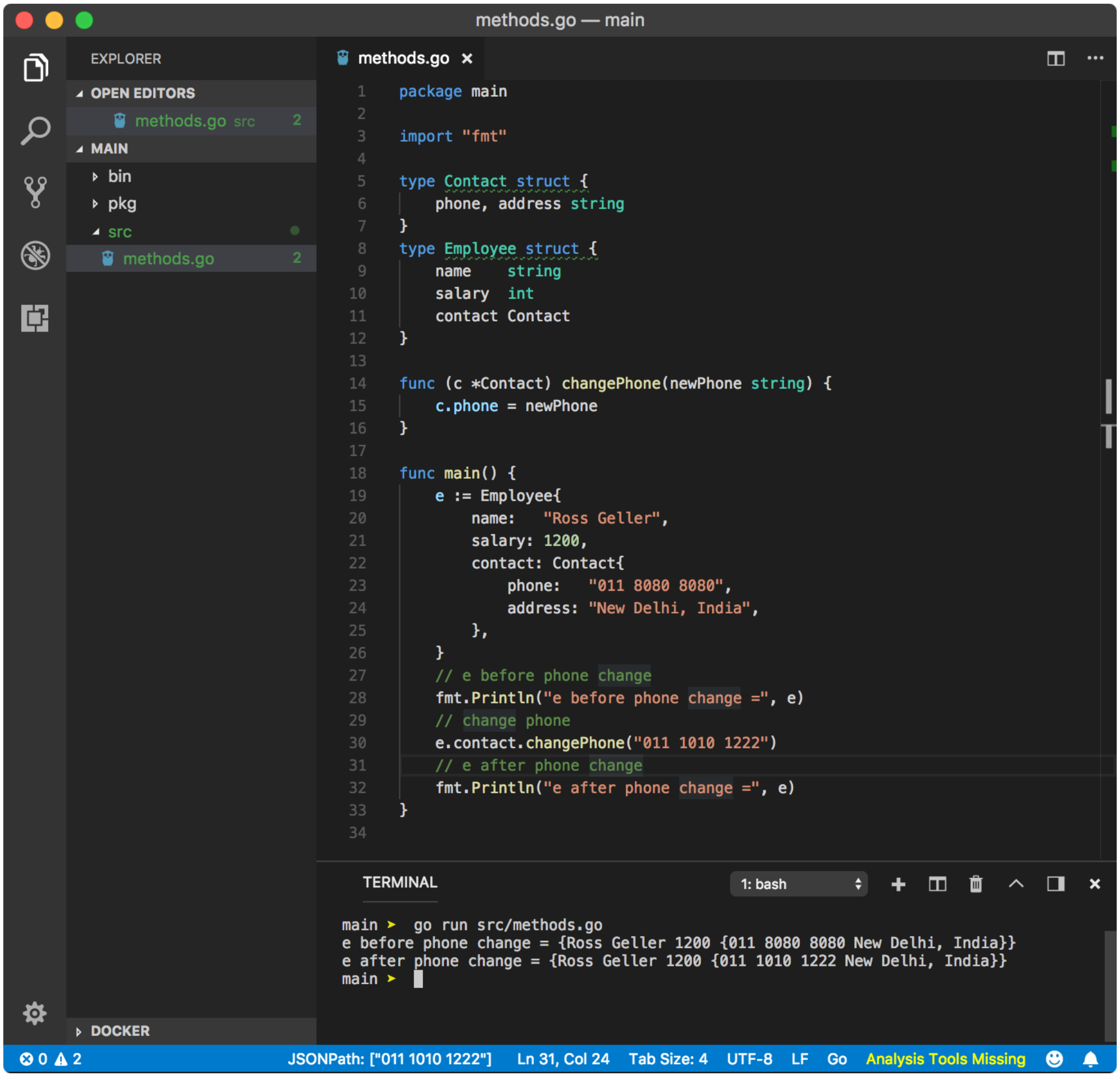


```go
package main

import "fmt"

type Contact struct {
	phone, address string
}
type Employee struct {
	name    string
	salary  int
	contact Contact
}

func (e *Employee) changePhone(newPhone string) {
	e.contact.phone = newPhone
}

func main() {
	e := Employee{
		name:    "Ross Geller",
		salary:  1200,
		contact: Contact{"011 8080 8080", "New Delhi, India"},
	}
	// e before phone change
	fmt.Println("e before phone change =", e)
	// change phone
	e.changePhone("011 1010 1222")
	// e after phone change
	fmt.Println("e after phone change =", e)
}
```

```
main ➤  go run src/methods.go
e before phone change = {Ross Geller 1200 {011 8080 8080 New Delhi, India}}
e after phone change = {Ross Geller 1200 {011 1010 1222 New Delhi, India}}
main ➤
```

https://play.golang.org/p/8_QhLqqb9Ot

If the inner struct implements a method, then you can call a method on it using `.` (*dot*) accessor.



```go
package main

import "fmt"

type Contact struct {
	phone, address string
}
type Employee struct {
	name    string
	salary  int
	contact Contact
}

func (c *Contact) changePhone(newPhone string) {
	c.phone = newPhone
}

func main() {
	e := Employee{
		name:   "Ross Geller",
		salary: 1200,
		contact: Contact{
			phone:   "011 8080 8080",
			address: "New Delhi, India",
		},
	}
	// e before phone change
	fmt.Println("e before phone change =", e)
	// change phone
	e.contact.changePhone("011 1010 1222")
	// e after phone change
	fmt.Println("e after phone change =", e)
}
```

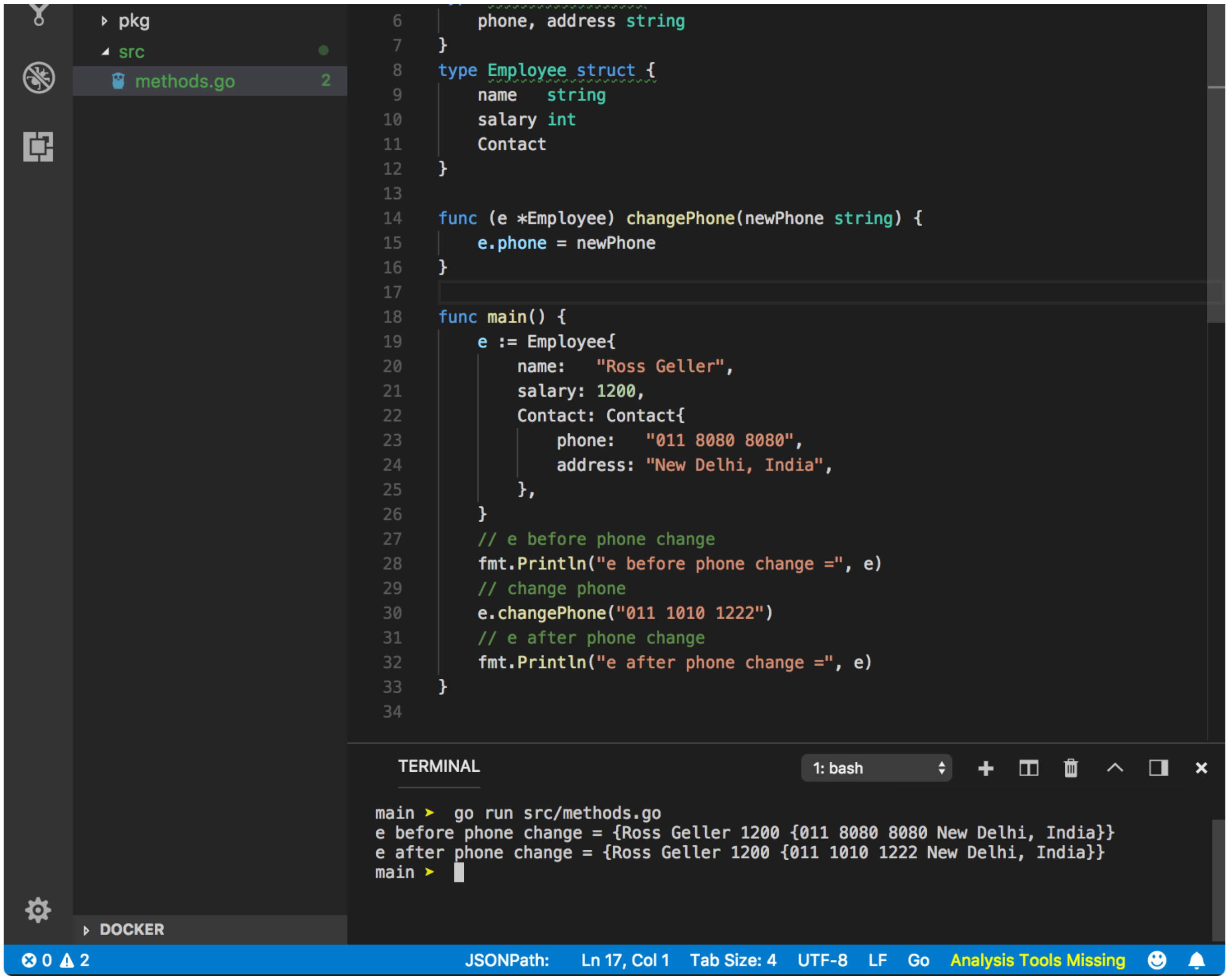
```
main ➤  go run src/methods.go
e before phone change = {Ross Geller 1200 {011 8080 8080 New Delhi, India}}
e after phone change = {Ross Geller 1200 {011 1010 1222 New Delhi, India}}
main ➤
```

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

But what will happen if the nested field is anonymous? You remember anonymous fields where a field has no name or name of the field is derived from its type. In case if the field is of a struct type, nested struct fields will be promoted.

Let’s see what will happen to the methods in a case when struct field is anonymous.

```go
package main

import "fmt"

type Contact struct {
```


pkg
src
methods.go
DOCKER
TERMINAL
1: bash
JSONPath: Ln 17, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

```go
    phone, address string
}
type Employee struct {
    name   string
    salary int
    Contact
}

func (e *Employee) changePhone(newPhone string) {
    e.phone = newPhone
}

func main() {
    e := Employee{
        name:   "Ross Geller",
        salary: 1200,
        Contact: Contact{
            phone:   "011 8080 8080",
            address: "New Delhi, India",
        },
    }
    // e before phone change
    fmt.Println("e before phone change =", e)
    // change phone
    e.changePhone("011 1010 1222")
    // e after phone change
    fmt.Println("e after phone change =", e)
}
```

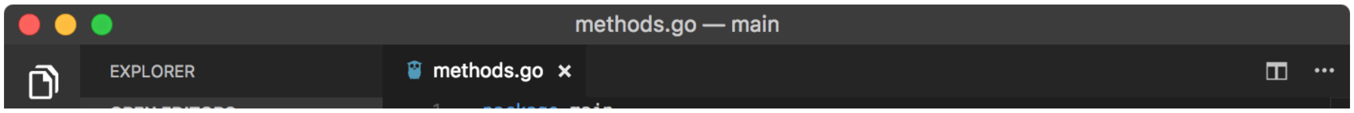
```
main ➤  go run src/methods.go
e before phone change = {Ross Geller 1200 {011 8080 8080 New Delhi, India}}
e after phone change = {Ross Geller 1200 {011 1010 1222 New Delhi, India}}
main ➤ 
```

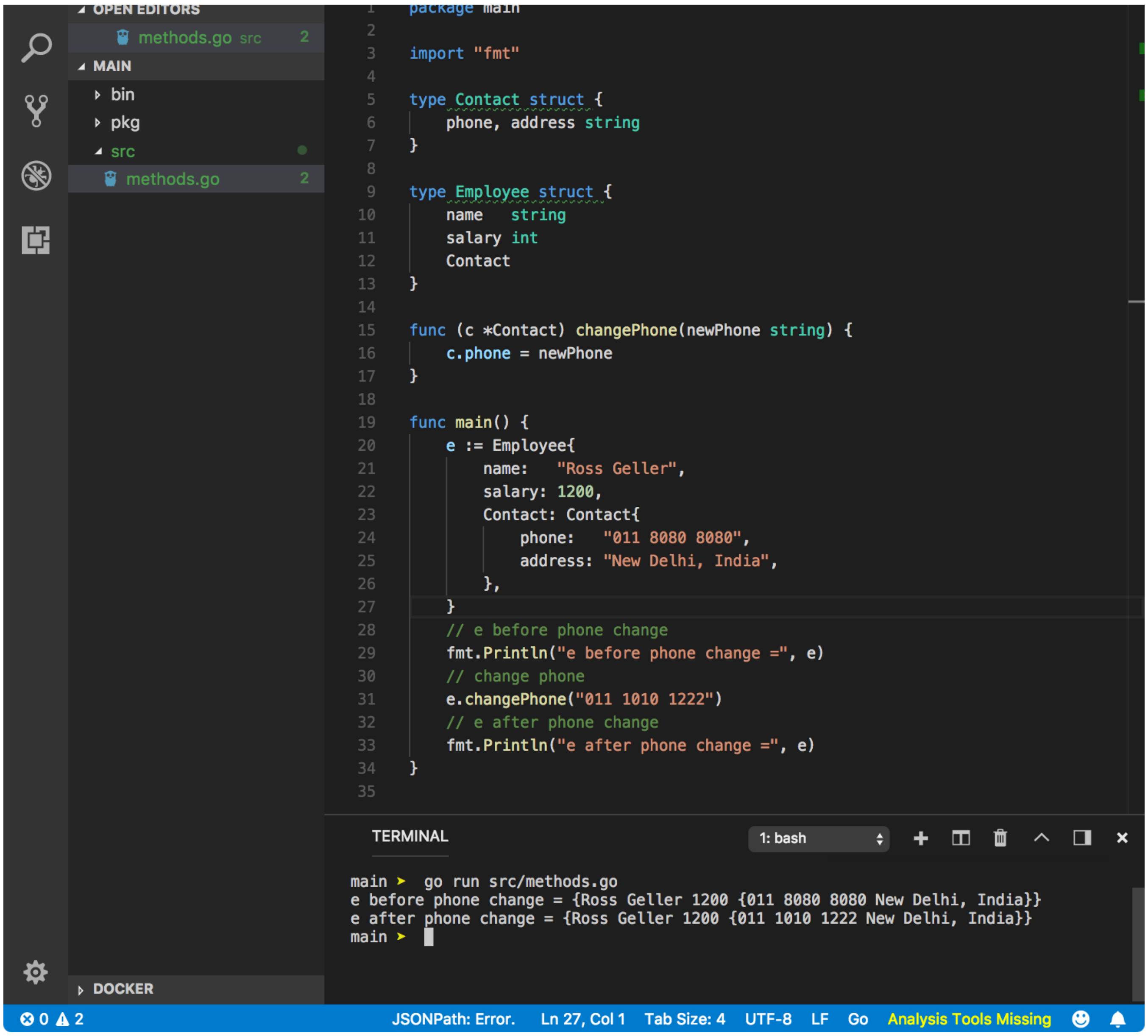
https://play.golang.org/p/d7D8FYYmiof

As we saw from `structs` lesson that if the nested field is anonymous struct then its fields will be accessible from parent struct. Hence any method that accepts struct receiver will also have access to the promoted fields.

☛ Promoted methods

Like promoted fields on a struct, methods implemented by inner struct is available on parent struct. As we saw in the previous example, `Contact` field is anonymous. Hence we could access `e.phone`, the phone number from inner struct as `phone` field is promoted to `Employee` struct. In the same scenario, any method implemented by `Contact` struct will be available on `Employee` struct. Let's rewrite the previous example.




OPEN EDITORS
methods.go src 2
MAIN
bin
pkg
src
methods.go 2
DOCKER
TERMINAL
1: bash
⊗ 0 ⚠ 2 JSONPath: Error. Ln 27, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

```go
package main

import "fmt"

type Contact struct {
	phone, address string
}

type Employee struct {
	name   string
	salary int
	Contact
}

func (c *Contact) changePhone(newPhone string) {
	c.phone = newPhone
}

func main() {
	e := Employee{
		name:   "Ross Geller",
		salary: 1200,
		Contact: Contact{
			phone:   "011 8080 8080",
			address: "New Delhi, India",
		},
	}
	// e before phone change
	fmt.Println("e before phone change =", e)
	// change phone
	e.changePhone("011 1010 1222")
	// e after phone change
	fmt.Println("e after phone change =", e)
}
```

```
main ➤  go run src/methods.go
e before phone change = {Ross Geller 1200 {011 8080 8080 New Delhi, India}}
e after phone change = {Ross Geller 1200 {011 1010 1222 New Delhi, India}}
main ➤ 
```

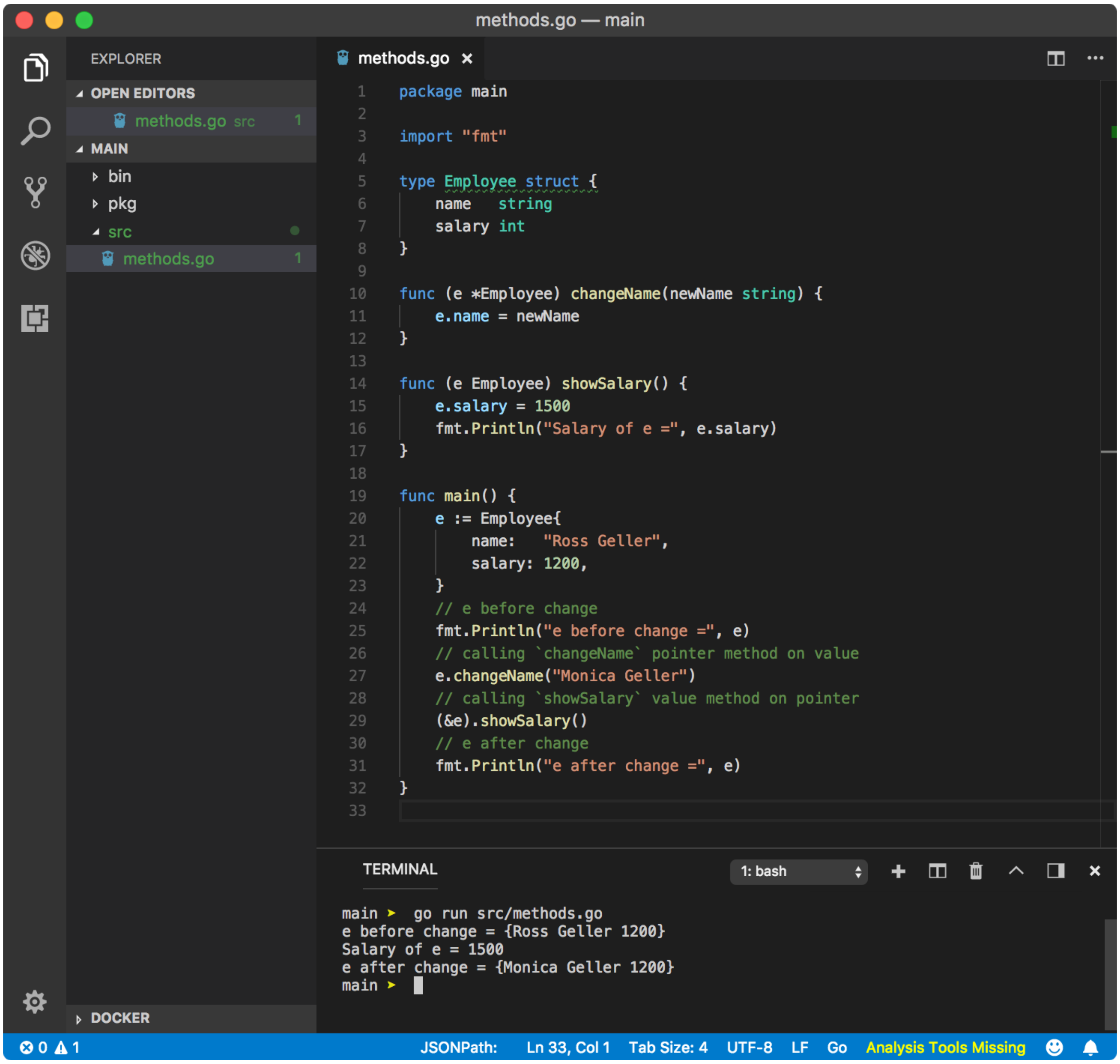
https://play.golang.org/p/x2L-rR-TKNE

We made only one change to `changePhone` function. Instead of receiving `Employee` type, method now expects `Contact` pointer receiver. Since, `Contact` field is promoted, any method on it will be promoted too. Hence we could use `e.changePhone` as if the type `Employee` of struct `e` implemented `changePhone` method.

☛ A method can accept both pointer and value

When a function has a value argument, it will only accept the value of the parameter. If you passed a pointer to the function which expects a value, it will not work. This is also true when function accepts pointer, you simply can not pass a value to it.

When it comes to a method, that's not the case. We can define a method with value or pointer receive and call it on pointer or value.



```go
package main

import "fmt"

type Employee struct {
	name   string
	salary int
}

func (e *Employee) changeName(newName string) {
	e.name = newName
}

func (e Employee) showSalary() {
	e.salary = 1500
	fmt.Println("Salary of e =", e.salary)
}

func main() {
	e := Employee{
		name:   "Ross Geller",
		salary: 1200,
	}
	// e before change
	fmt.Println("e before change =", e)
	// calling `changeName` pointer method on value
	e.changeName("Monica Geller")
	// calling `showSalary` value method on pointer
	(&e).showSalary()
	// e after change
	fmt.Println("e after change =", e)
}
```

```
main ➤  go run src/methods.go
e before change = {Ross Geller 1200}
Salary of e = 1500
e after change = {Monica Geller 1200}
main ➤
```

https://play.golang.org/p/tiQQbimhQ8_O

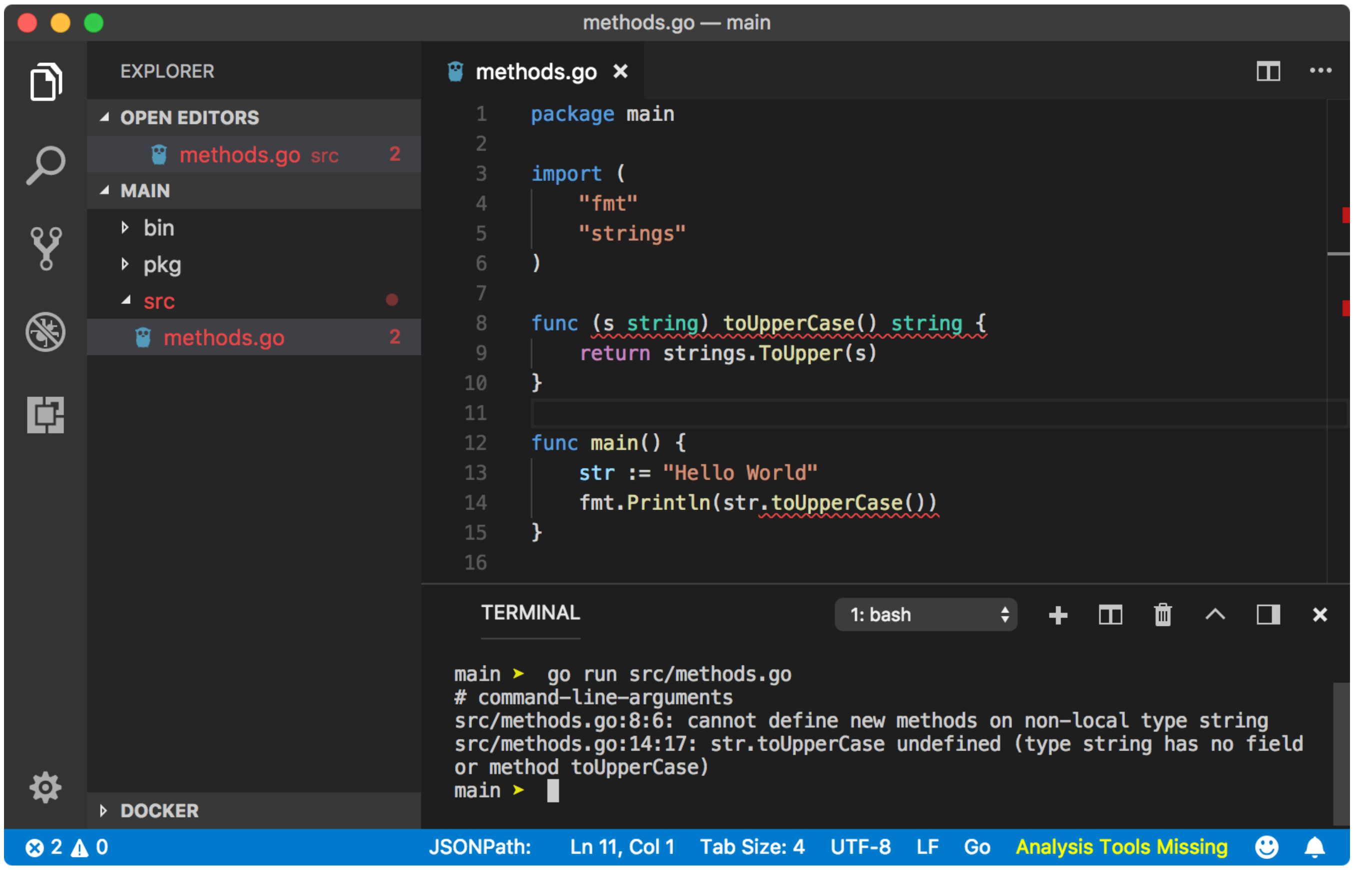
In the above program, we defined `changeName` method which accepts a pointer but we called on the value `e` which is legal because Go under the hood will pass a pointer of `e` to it. Also, we defined `showSalary` method which accepts value but we called on the pointer to `e` which is legal because Go under the hood will pass the value of the pointer to it.

> *We tried to change salary of `e` inside `showSalary` method but did not work as we can see from the result. This is because even we call this method on a pointer, Go sent only copy of the value to that method.*

☛ Methods on non-struct type

So far we have seen methods on struct type but as from the definition of the method, it can accept any type as a receiver as long as type definition and method definition is in the same package. So far, we defined struct and method in the same `main` package, hence it worked.

To show this, we will try to add a method `toUpperCase` on the built-in type `string`.



```go
package main

import (
	"fmt"
	"strings"
)

func (s string) toUpperCase() string {
	return strings.ToUpper(s)
}

func main() {
	str := "Hello World"
	fmt.Println(str.toUpperCase())
}
```

```
main ➤  go run src/methods.go
# command-line-arguments
src/methods.go:8:6: cannot define new methods on non-local type string
src/methods.go:14:17: str.toUpperCase undefined (type string has no field
or method toUpperCase)
main ➤
```

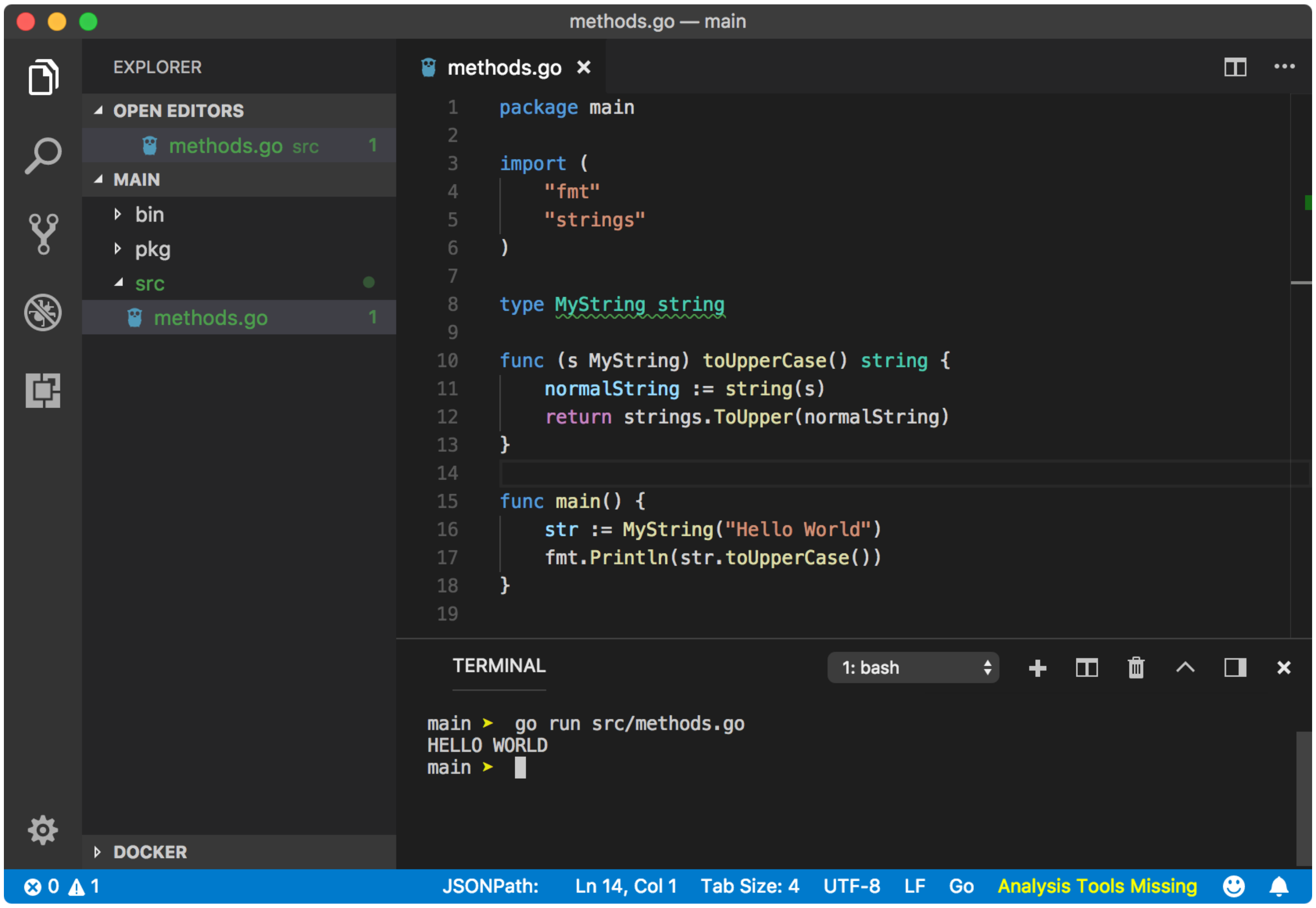
https://play.golang.org/p/FVvlh-YQBLH

From the above program, we created the method `toUpperCase` which accepts `string` as receiver type. Hence we expect `string.toUpperCase()` to work and return uppercase version of the receiver `s`. We used `strings` package to convert a string to uppercase.

But the above program will run into a compilation error

```
program.go:8: cannot define new methods on non-local type string
program.go:14: str.toUpperCase undefined (type string has no field or
method toUpperCase)
```

This is because, type `string` and method `toUpperCase` is not defined in the same package. Let's create a type alias `MyString` of `string` and make it work.



```go
package main

import (
	"fmt"
	"strings"
)

type MyString string

func (s MyString) toUpperCase() string {
	normalString := string(s)
	return strings.ToUpper(normalString)
}

func main() {
	str := MyString("Hello World")
	fmt.Println(str.toUpperCase())
}
```

```
main ➤ go run src/methods.go
HELLO WORLD
main ➤
```

https://play.golang.org/p/N-lLB3xPSDM

From the above program, we created the method `toUpperCase` which now accepts `MyString` type. We needed to modify the internals of this method to pass string type to `strings.ToUpper` function, but we get it.

Now we can call `str.toUpperCase()` because `str` is of type `MyString` as we used **type conversion** on the line no. 16 to convert from `string` type to `MyString` type.