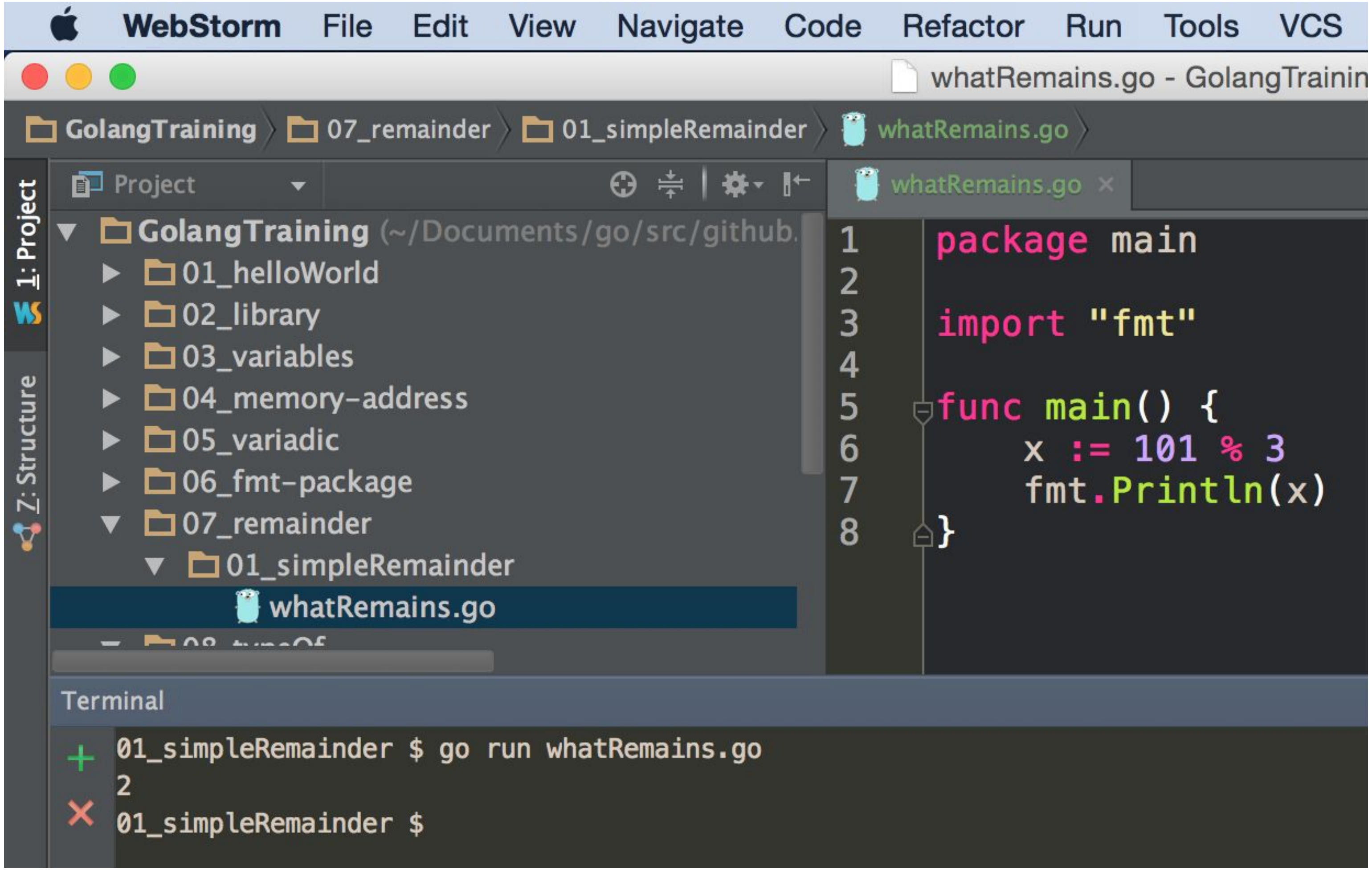
# Loops & Conditionals

**remainders, loops, range, conditionals**

**remainder**

%


WebStorm File Edit View Navigate Code Refactor Run Tools VCS
whatRemains.go - GolangTrainin
GolangTraining 07_remainder 01_simpleRemainder whatRemains.go
1: Project
7: Structure
Project
GolangTraining (~/Documents/go/src/github.
01_helloWorld
02_library
03_variables
04_memory-address
05_variadic
06_fmt-package
07_remainder
01_simpleRemainder
whatRemains.go
Terminal

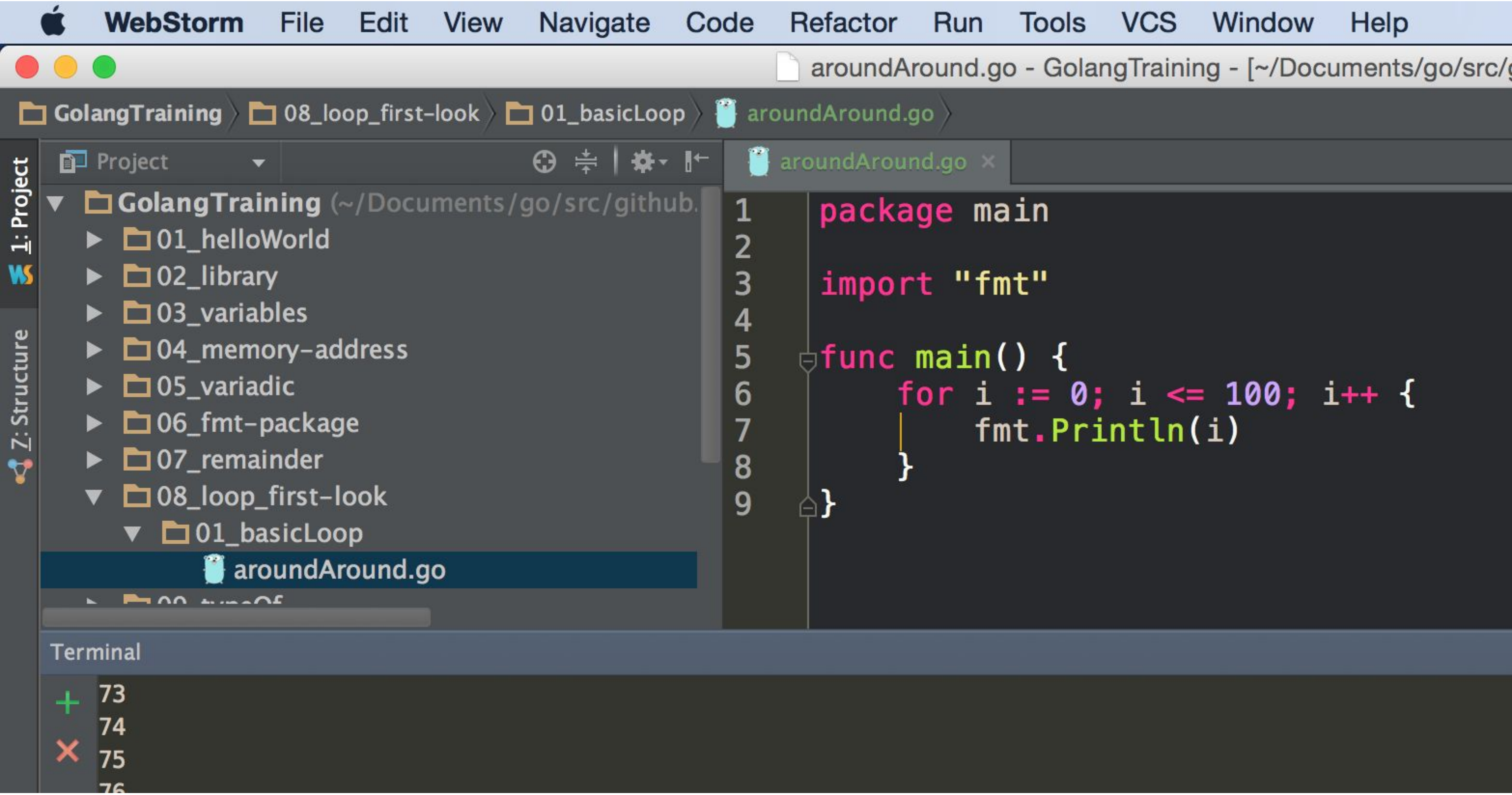
```go
package main

import "fmt"

func main() {
	x := 101 % 3
	fmt.Println(x)
}
```

```
01_simpleRemainder $ go run whatRemains.go
2
01_simpleRemainder $
```

## exercise

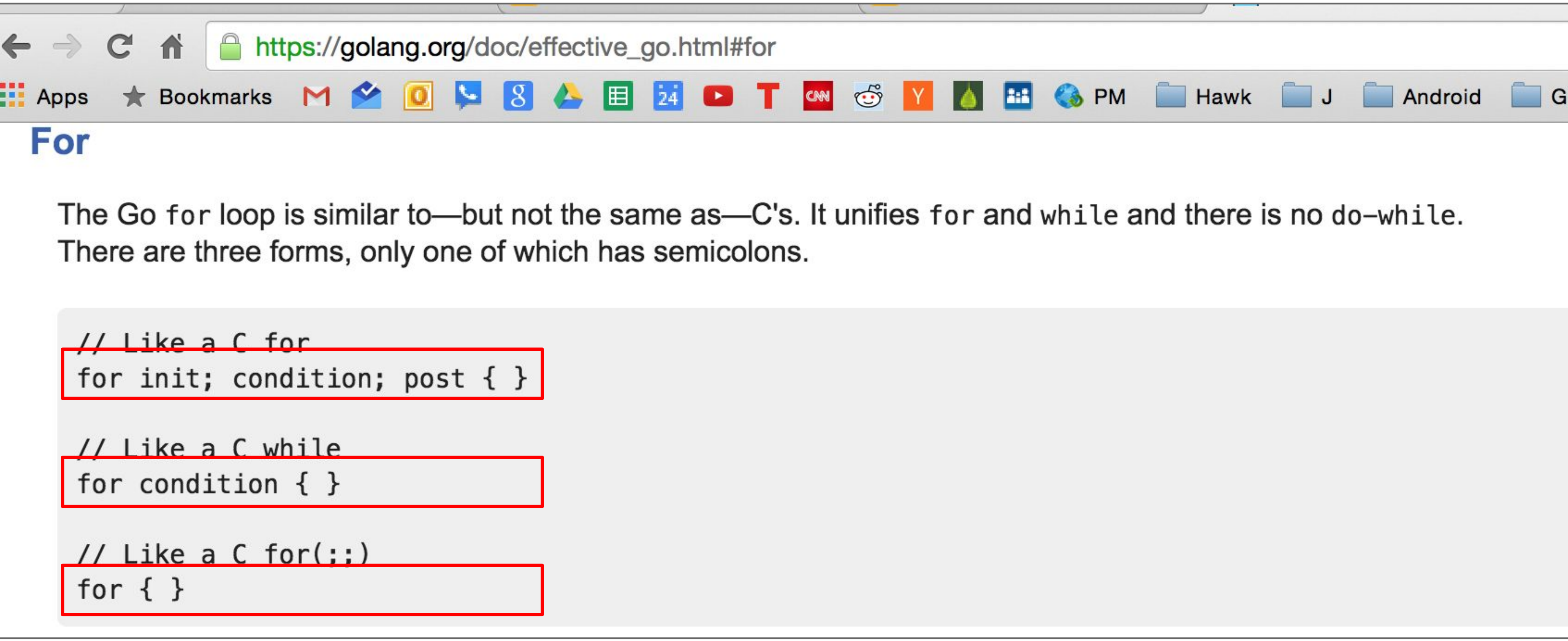
write a program
that allows the user to enter two numbers
then displays the remainder
when one number is divided by the other

## loops


WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
aroundAround.go - GolangTraining - [~/Documents/go/src/
GolangTraining 08_loop_first-look 01_basicLoop aroundAround.go
Project
1: Project
7: Structure
GolangTraining (~/Documents/go/src/github.
01_helloWorld
02_library
03_variables
04_memory-address
05_variadic
06_fmt-package
07_remainder
08_loop_first-look
01_basicLoop
aroundAround.go
aroundAround.go
Terminal

```go
package main

import "fmt"

func main() {
	for i := 0; i <= 100; i++ {
		fmt.Println(i)
	}
}
```

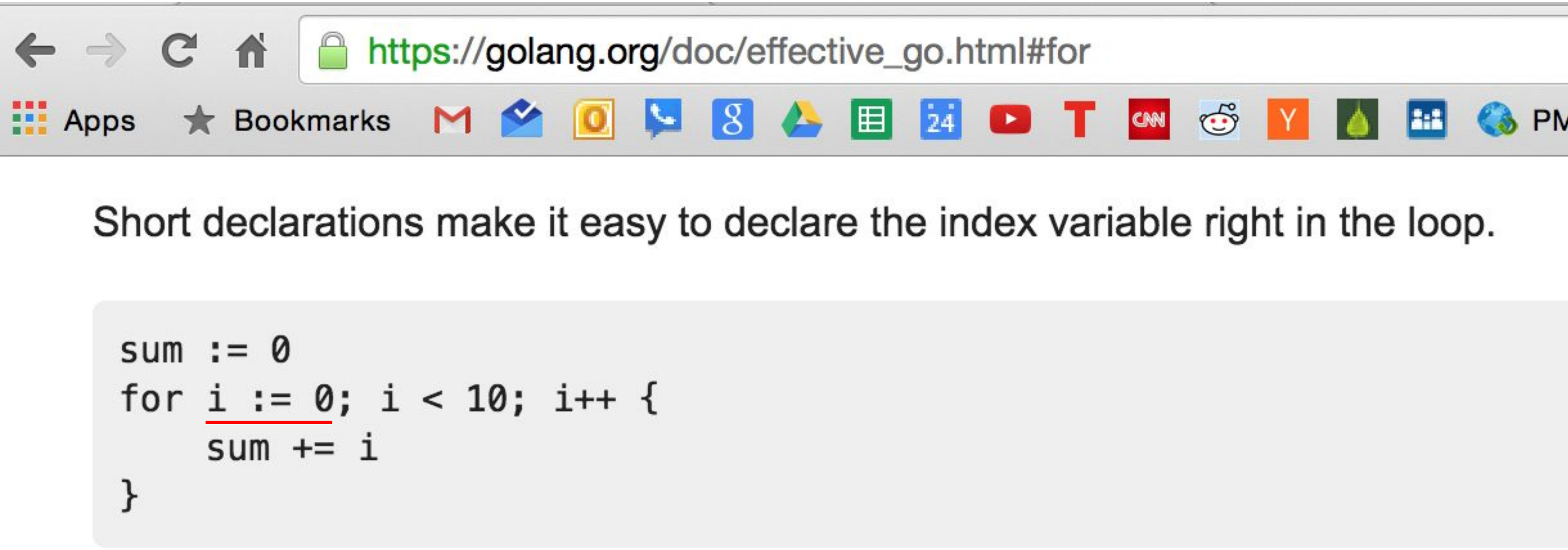
```
73
74
75
```



## For

The Go `for` loop is similar to—but not the same as—C's. It unifies `for` and `while` and there is no `do-while`. There are three forms, only one of which has semicolons.

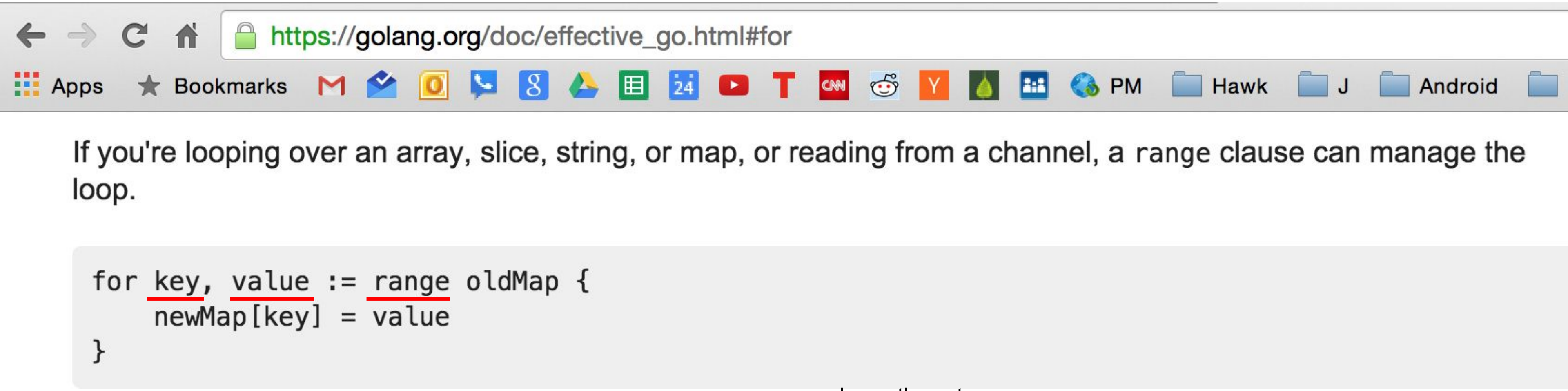
```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```



Short declarations make it easy to declare the index variable right in the loop.

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

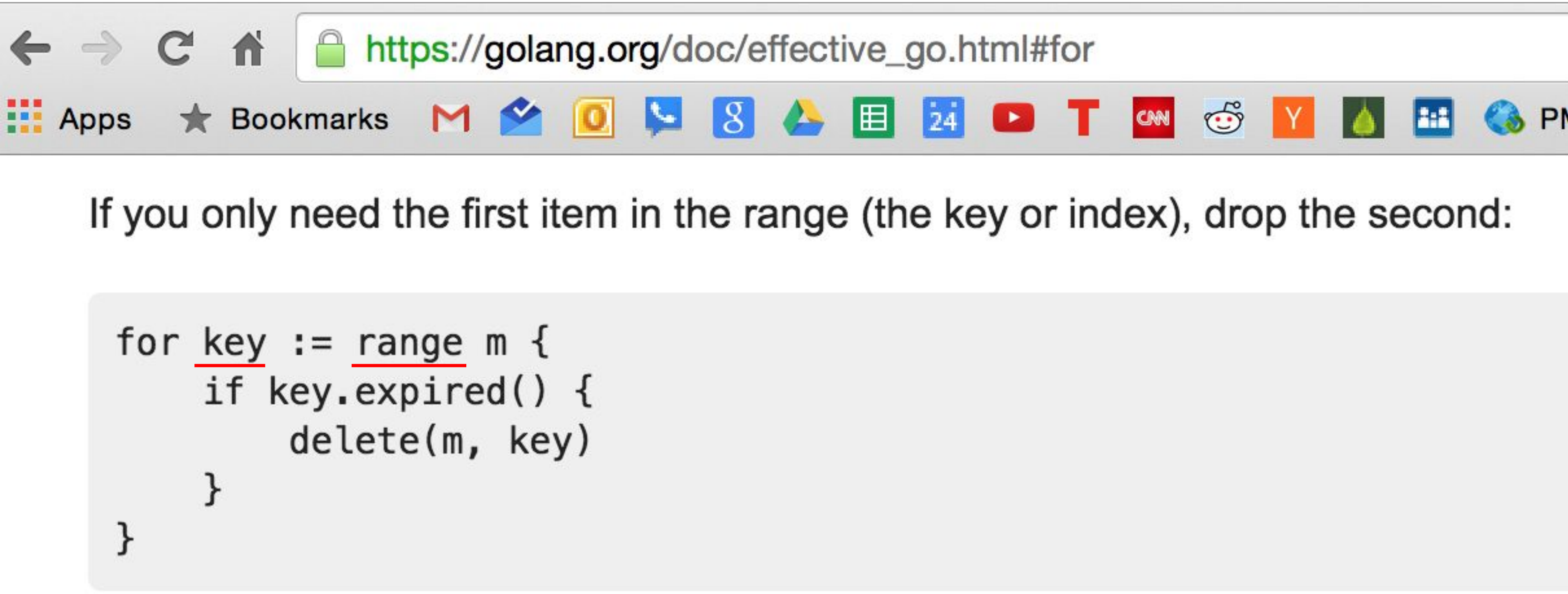
the scope of i will only be this loop



If you're looping over an array, slice, string, or map, or reading from a channel, a range clause can manage the loop.

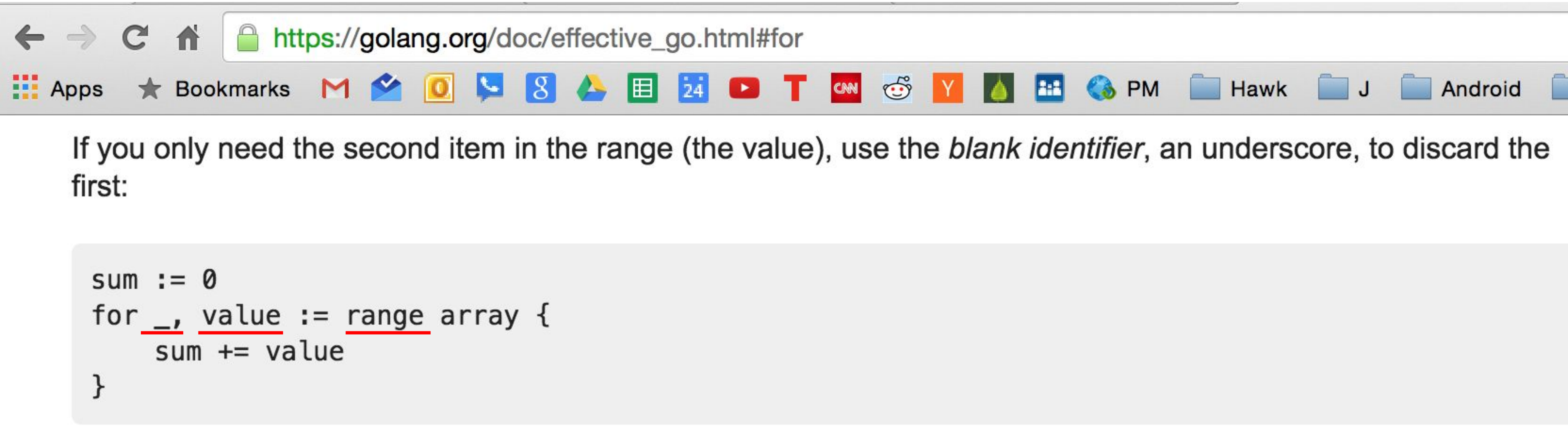
```
for key, value := range oldMap {
    newMap[key] = value
}
```

- range works on these types:
  - slice or array
  - string
    - gives us a rune (code point to UTF-8 character)
  - map
    - key:value
  - channel
    - a channel is a way to communicate between threads (different go routines)
    - you can use the "for range" to read off of a channel continuously



If you only need the first item in the range (the key or index), drop the second:

```
for key := range m {
    if key.expired() {
        delete(m, key)
    }
}
```

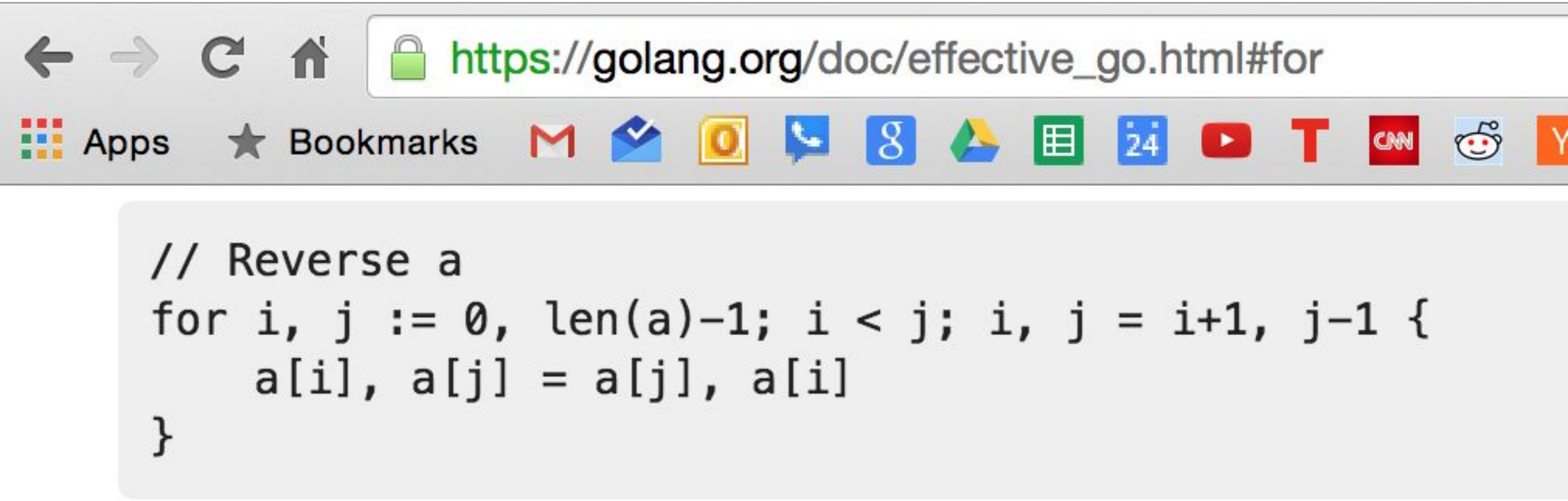

https://golang.org/doc/effective_go.html#for

If you only need the second item in the range (the value), use the *blank identifier*, an underscore, to discard the first:

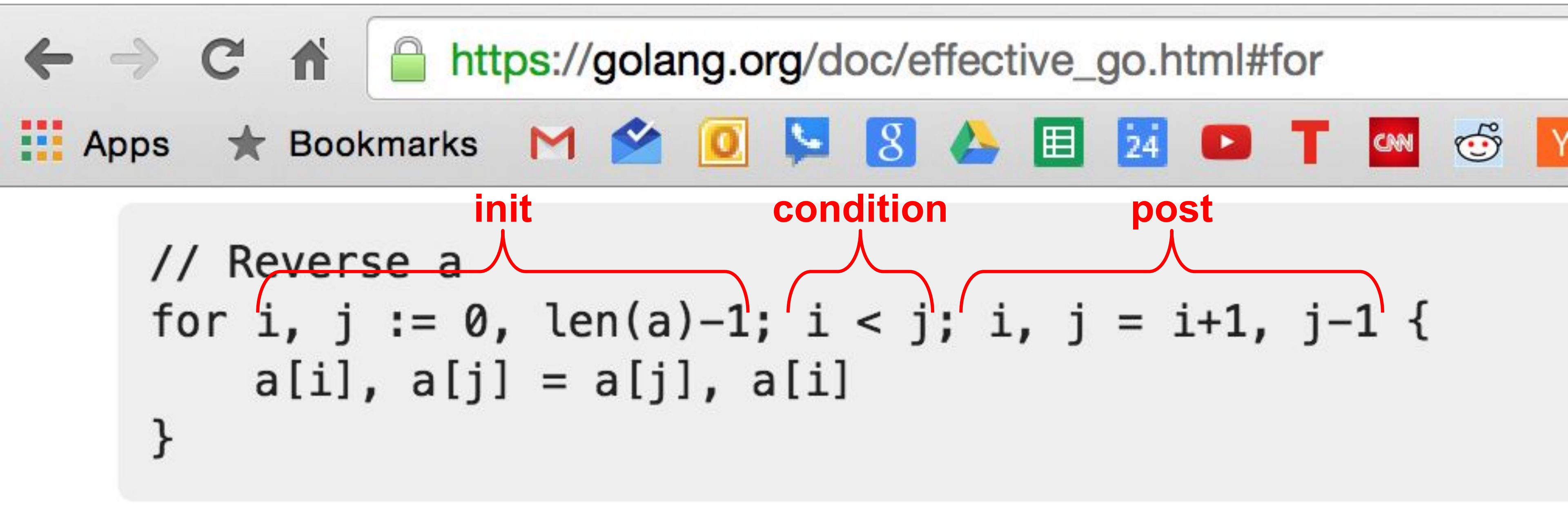
```
sum := 0
for _, value := range array {
    sum += value
}
```

## What’s going on here?

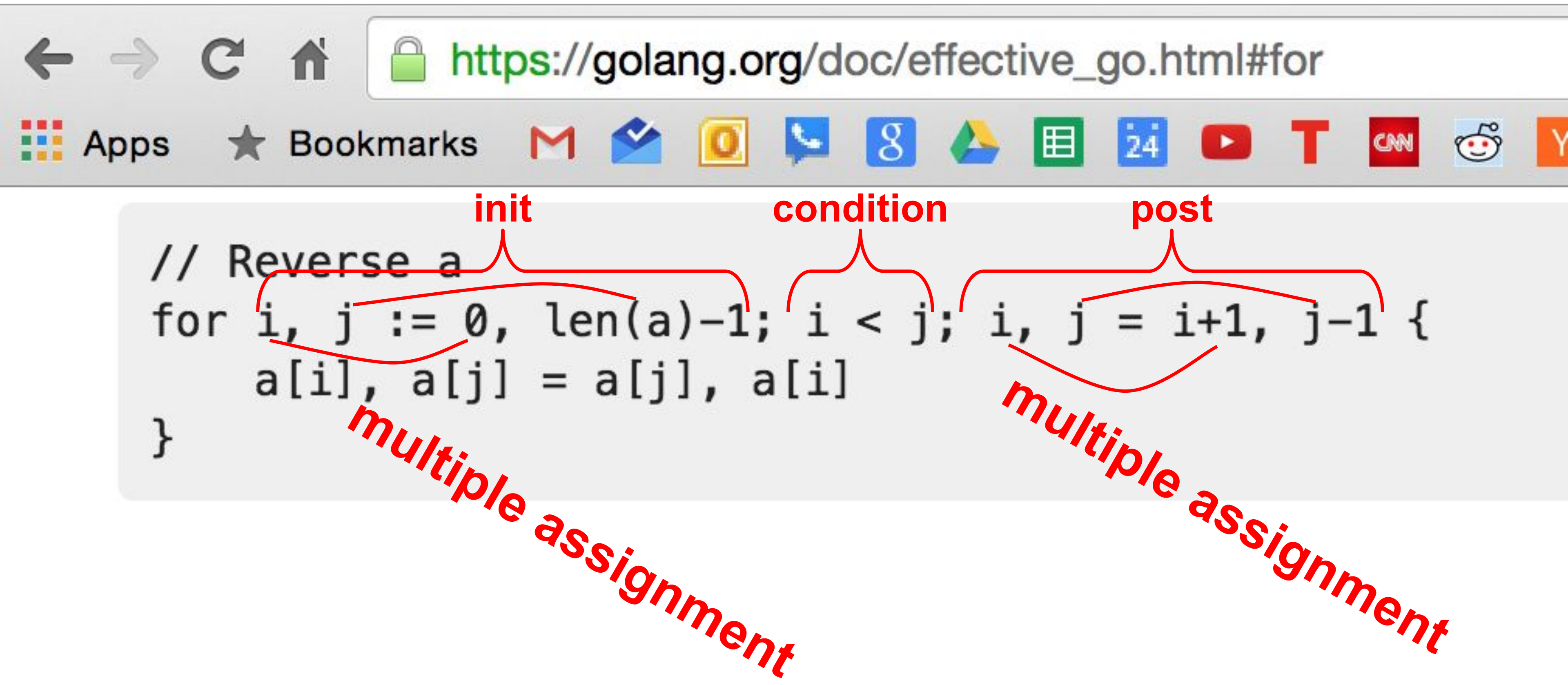
(see the next slide for help)



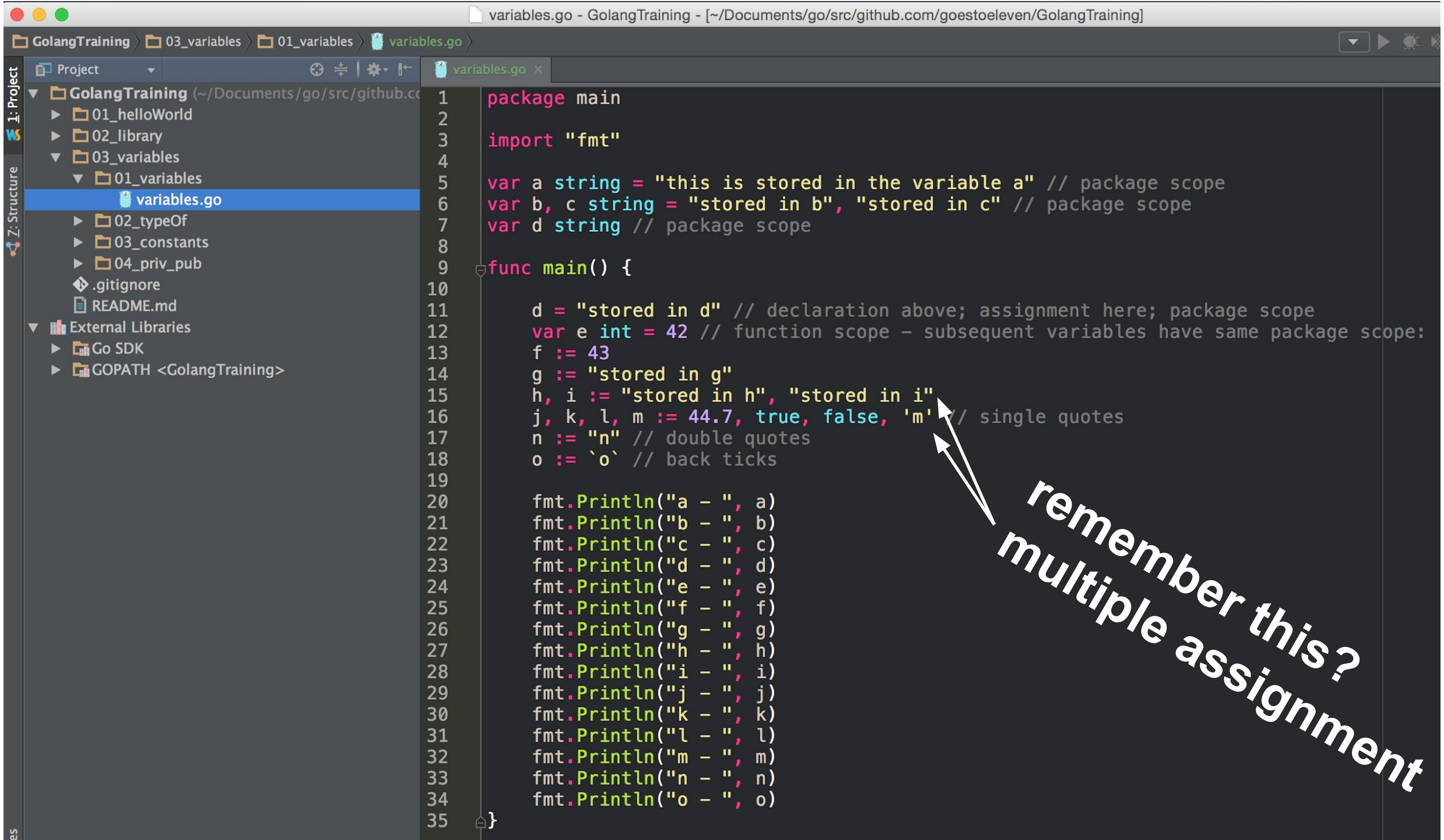
```go
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```



```go
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```



```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```



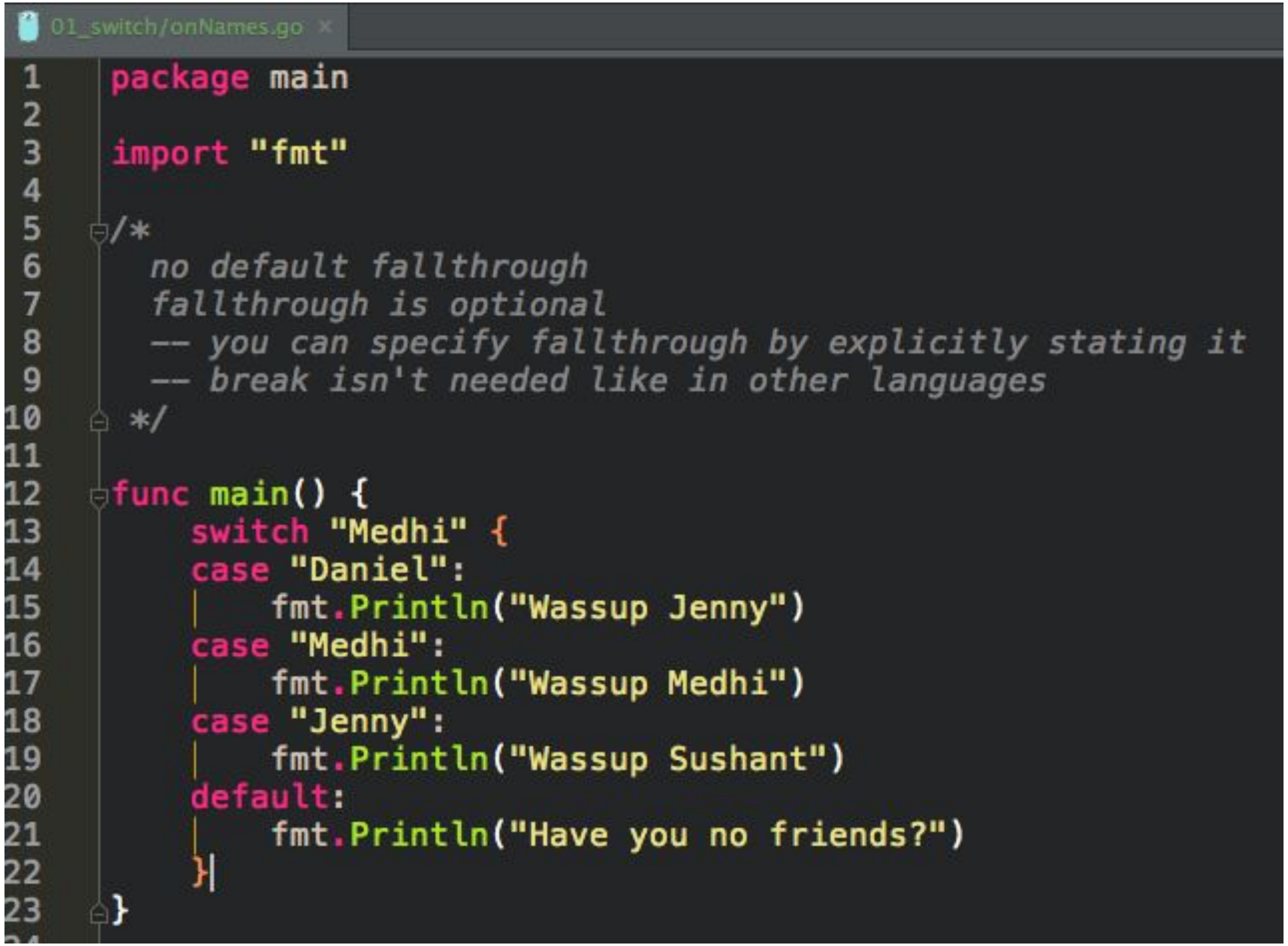
```go
package main

import "fmt"

var a string = "this is stored in the variable a" // package scope
var b, c string = "stored in b", "stored in c" // package scope
var d string // package scope

func main() {

	d = "stored in d" // declaration above; assignment here; package scope
	var e int = 42 // function scope - subsequent variables have same package scope:
	f := 43
	g := "stored in g"
	h, i := "stored in h", "stored in i"
	j, k, l, m := 44.7, true, false, 'm' // single quotes
	n := "n" // double quotes
	o := `o` // back ticks

	fmt.Println("a - ", a)
	fmt.Println("b - ", b)
	fmt.Println("c - ", c)
	fmt.Println("d - ", d)
	fmt.Println("e - ", e)
	fmt.Println("f - ", f)
	fmt.Println("g - ", g)
	fmt.Println("h - ", h)
	fmt.Println("i - ", i)
	fmt.Println("j - ", j)
	fmt.Println("k - ", k)
	fmt.Println("l - ", l)
	fmt.Println("m - ", m)
	fmt.Println("n - ", n)
	fmt.Println("o - ", o)
}
```

remember this?
multiple assignment

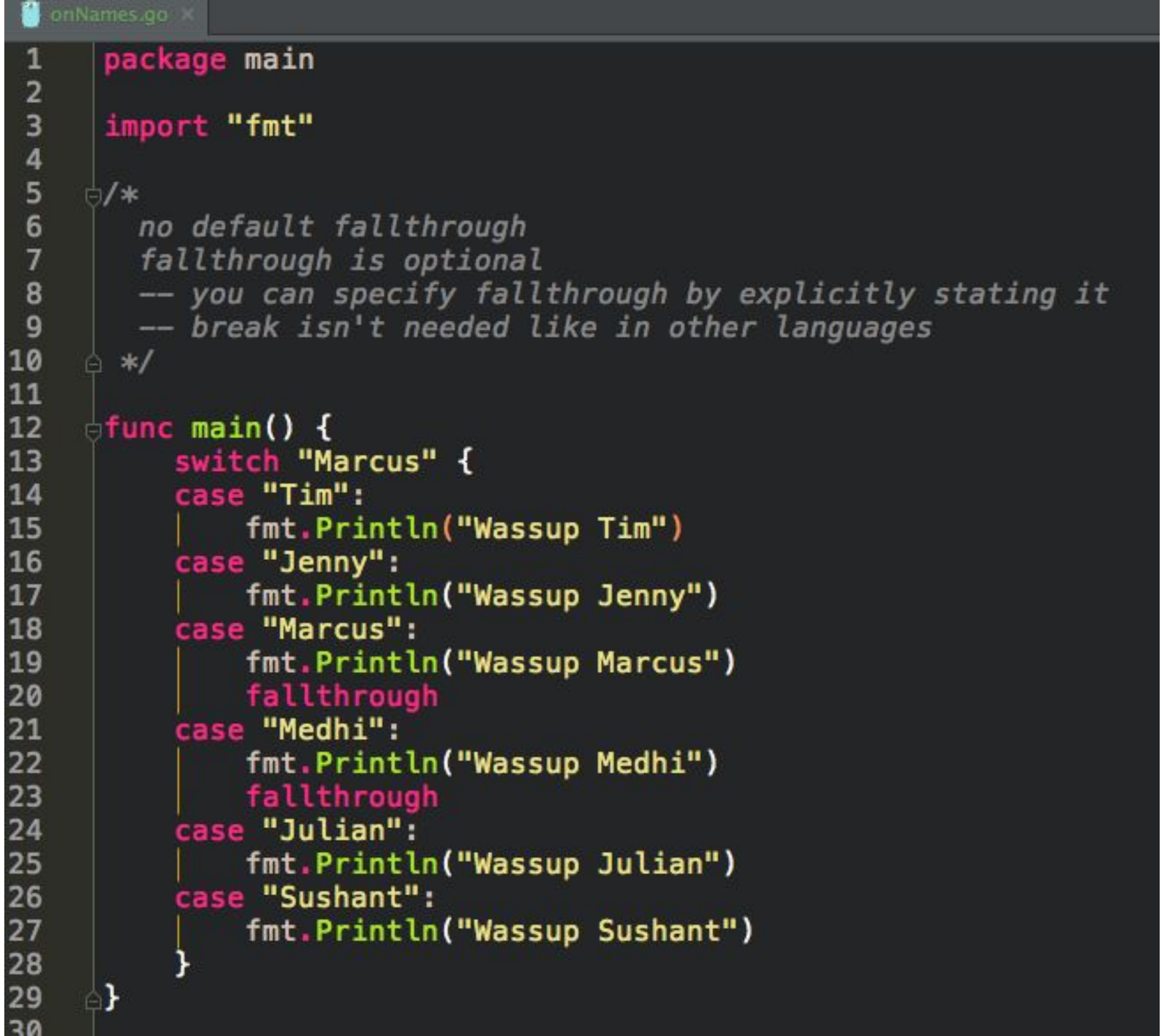
## exercise

write a program
that loops from 1 - 1,000
printing even numbers

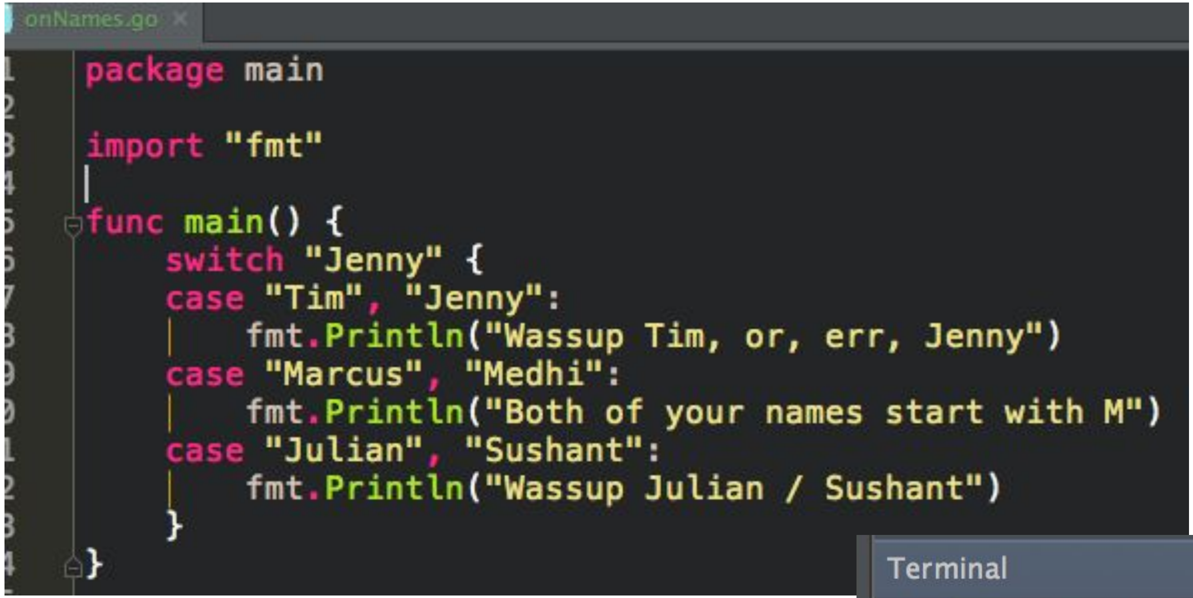
## switch statements


01_switch/onNames.go

```go
package main

import "fmt"

/*
  no default fallthrough
  fallthrough is optional
  -- you can specify fallthrough by explicitly stating it
  -- break isn't needed like in other languages
 */

func main() {
	switch "Medhi" {
	case "Daniel":
		fmt.Println("Wassup Jenny")
	case "Medhi":
		fmt.Println("Wassup Medhi")
	case "Jenny":
		fmt.Println("Wassup Sushant")
	default:
		fmt.Println("Have you no friends?")
	}
}
```

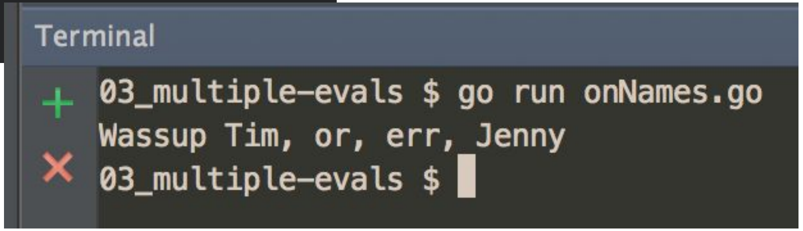


```go
package main

import "fmt"

/*
  no default fallthrough
  fallthrough is optional
  -- you can specify fallthrough by explicitly stating it
  -- break isn't needed like in other languages
 */

func main() {
    switch "Marcus" {
    case "Tim":
        fmt.Println("Wassup Tim")
    case "Jenny":
        fmt.Println("Wassup Jenny")
    case "Marcus":
        fmt.Println("Wassup Marcus")
        fallthrough
    case "Medhi":
        fmt.Println("Wassup Medhi")
        fallthrough
    case "Julian":
        fmt.Println("Wassup Julian")
    case "Sushant":
        fmt.Println("Wassup Sushant")
    }
}
```



onNames.go

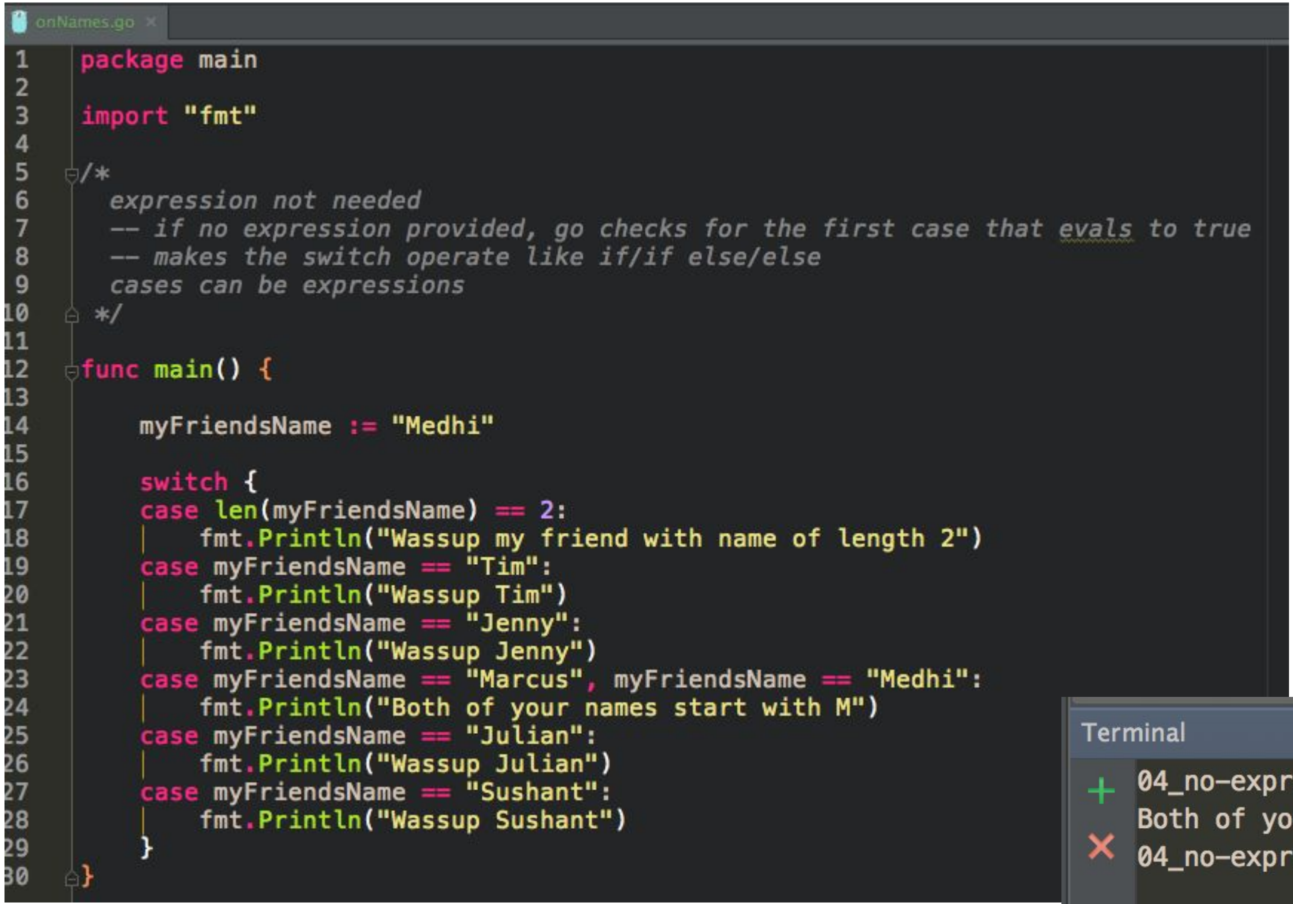
```go
package main

import "fmt"

func main() {
	switch "Jenny" {
	case "Tim", "Jenny":
		fmt.Println("Wassup Tim, or, err, Jenny")
	case "Marcus", "Medhi":
		fmt.Println("Both of your names start with M")
	case "Julian", "Sushant":
		fmt.Println("Wassup Julian / Sushant")
	}
}
```



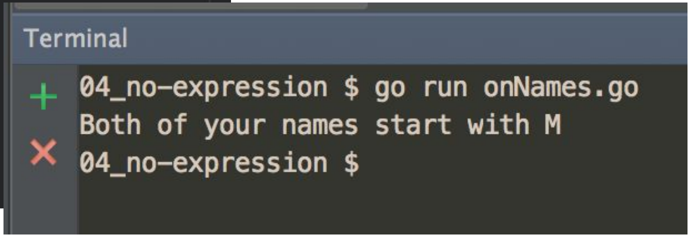
Terminal

```
03_multiple-evals $ go run onNames.go
Wassup Tim, or, err, Jenny
03_multiple-evals $
```

multiple evals

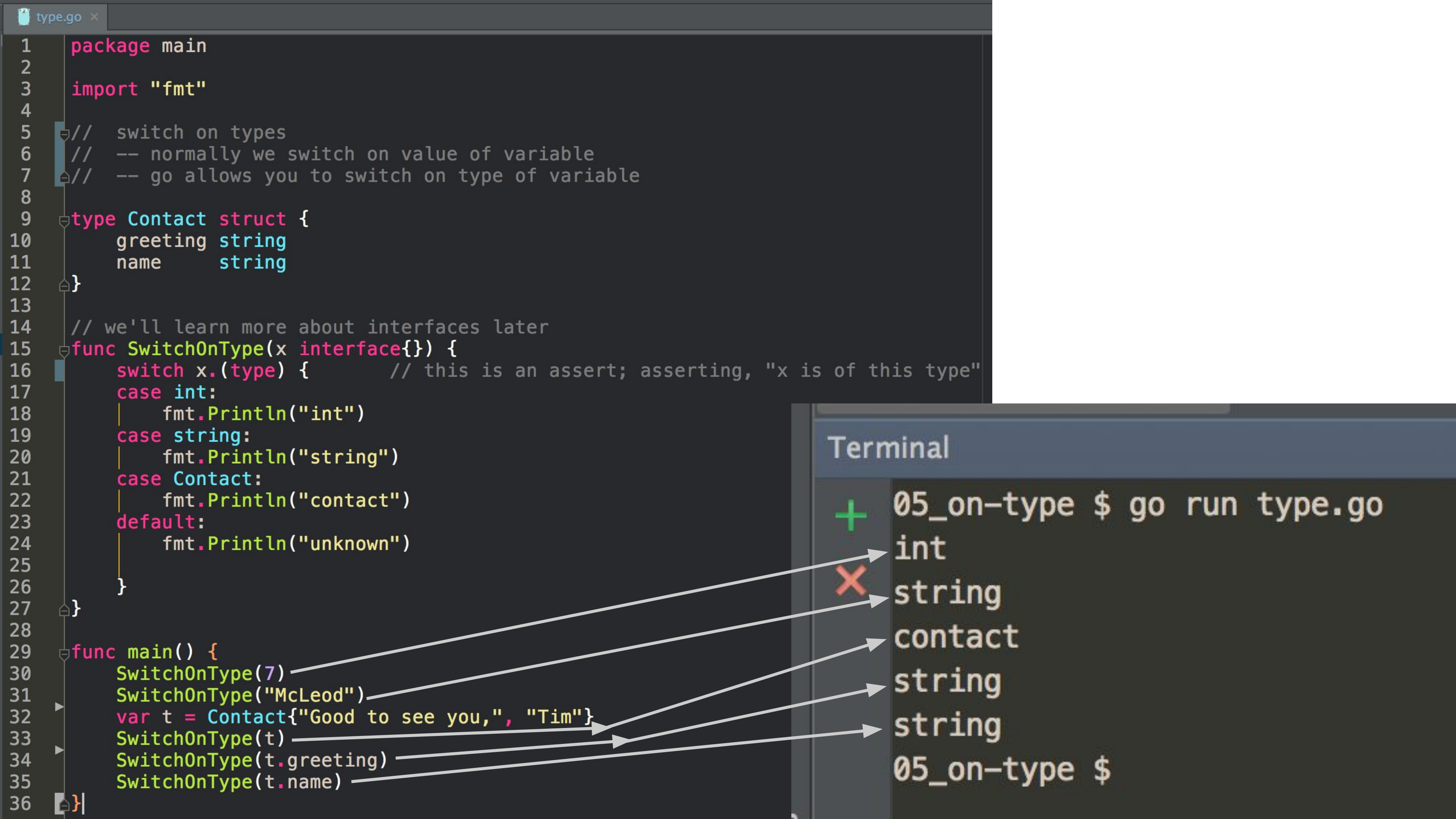

onNames.go

```go
package main

import "fmt"

/*
  expression not needed
  -- if no expression provided, go checks for the first case that evals to true
  -- makes the switch operate like if/if else/else
  cases can be expressions
 */

func main() {

	myFriendsName := "Medhi"

	switch {
	case len(myFriendsName) == 2:
		fmt.Println("Wassup my friend with name of length 2")
	case myFriendsName == "Tim":
		fmt.Println("Wassup Tim")
	case myFriendsName == "Jenny":
		fmt.Println("Wassup Jenny")
	case myFriendsName == "Marcus", myFriendsName == "Medhi":
		fmt.Println("Both of your names start with M")
	case myFriendsName == "Julian":
		fmt.Println("Wassup Julian")
	case myFriendsName == "Sushant":
		fmt.Println("Wassup Sushant")
	}
}
```


Terminal

```
04_no-expression $ go run onNames.go
Both of your names start with M
04_no-expression $
```

cases can be expressions

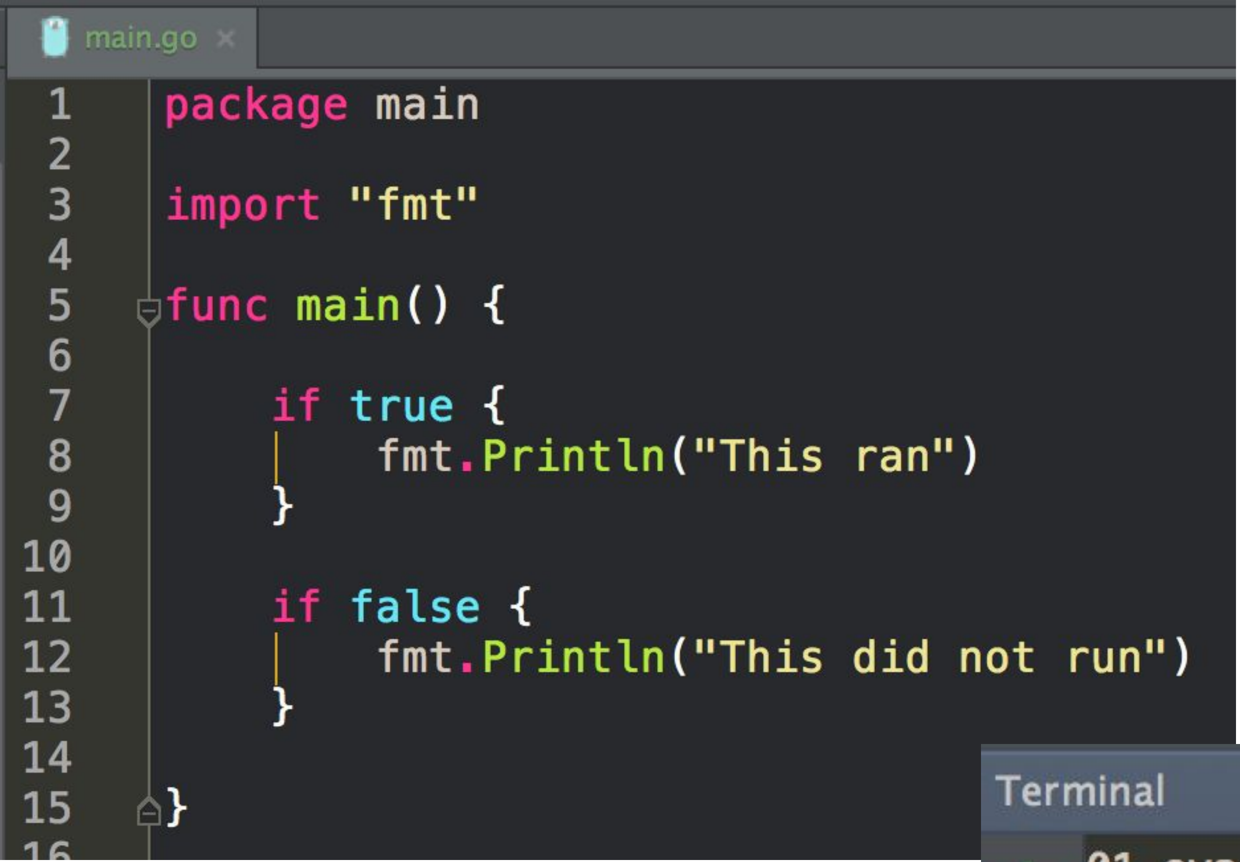


```go
package main

import "fmt"

//  switch on types
//  -- normally we switch on value of variable
//  -- go allows you to switch on type of variable

type Contact struct {
    greeting string
    name     string
}

// we'll learn more about interfaces later
func SwitchOnType(x interface{}) {
    switch x.(type) {       // this is an assert; asserting, "x is of this type"
    case int:
        fmt.Println("int")
    case string:
        fmt.Println("string")
    case Contact:
        fmt.Println("contact")
    default:
        fmt.Println("unknown")

    }
}

func main() {
    SwitchOnType(7)
    SwitchOnType("McLeod")
    var t = Contact{"Good to see you,", "Tim"}
    SwitchOnType(t)
    SwitchOnType(t.greeting)
    SwitchOnType(t.name)
}
```

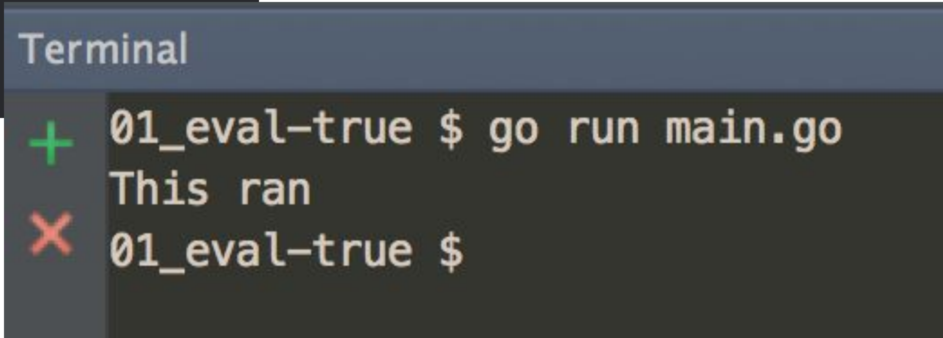
```
05_on-type $ go run type.go
int
string
contact
string
string
05_on-type $
```

## conditional

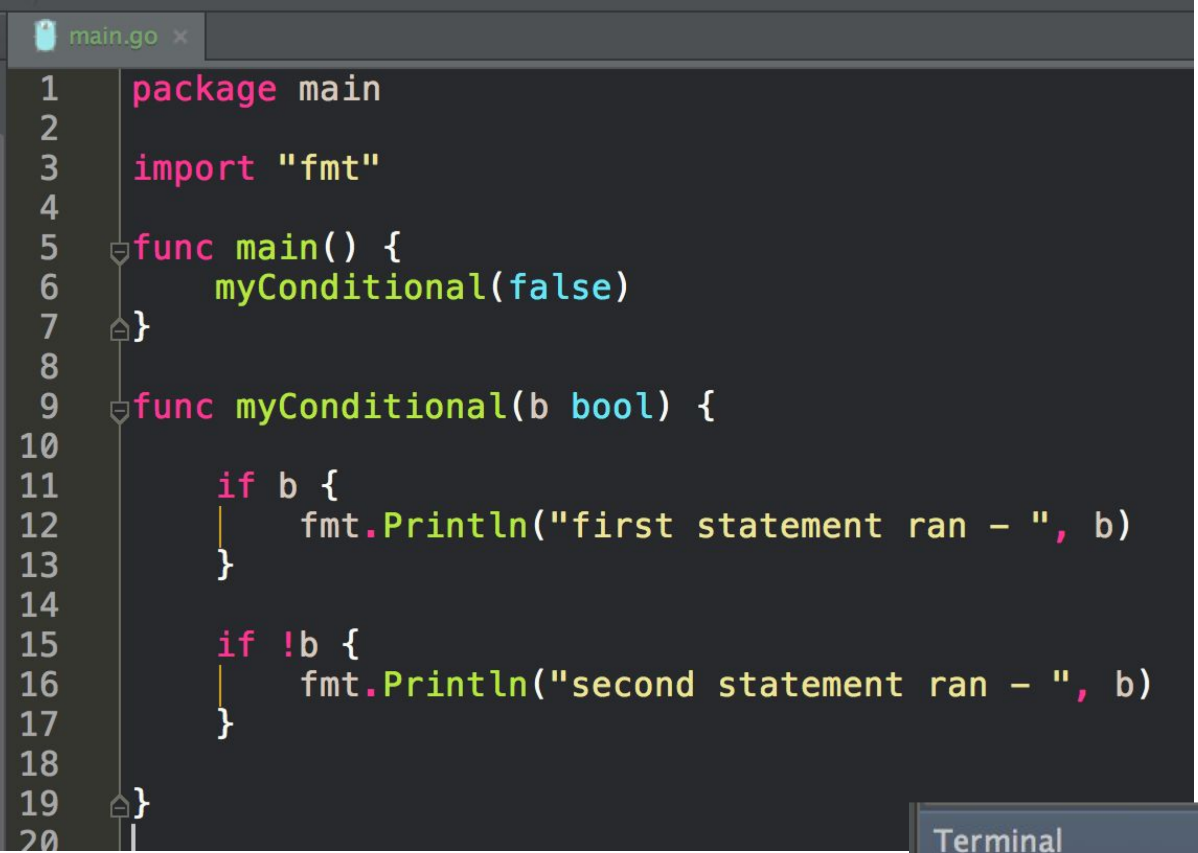
if statement


main.go

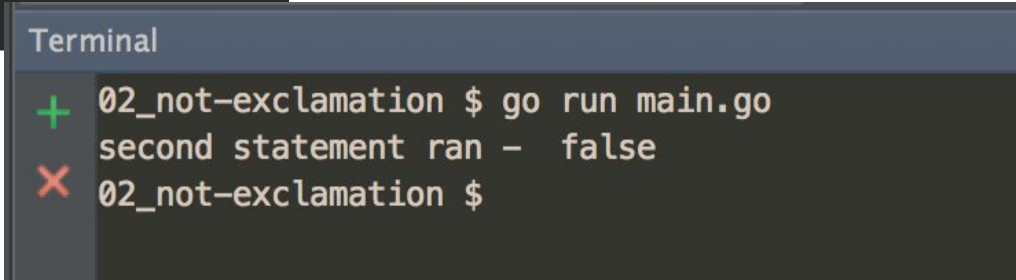
```go
package main

import "fmt"

func main() {

	if true {
		fmt.Println("This ran")
	}

	if false {
		fmt.Println("This did not run")
	}

}
```


Terminal

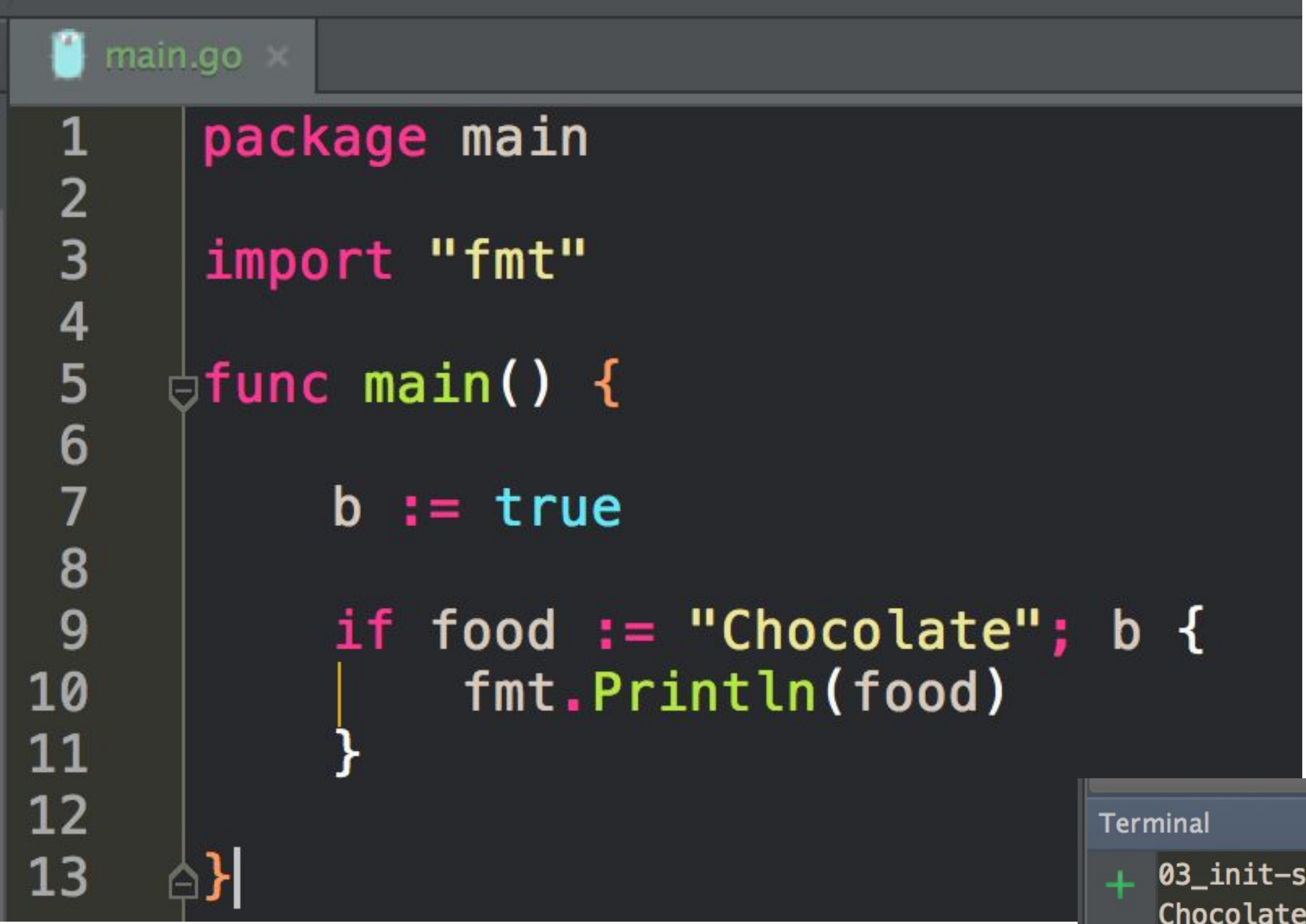
```
01_eval-true $ go run main.go
This ran
01_eval-true $
```


main.go

```go
package main

import "fmt"

func main() {
	myConditional(false)
}

func myConditional(b bool) {

	if b {
		fmt.Println("first statement ran - ", b)
	}

	if !b {
		fmt.Println("second statement ran - ", b)
	}

}
```

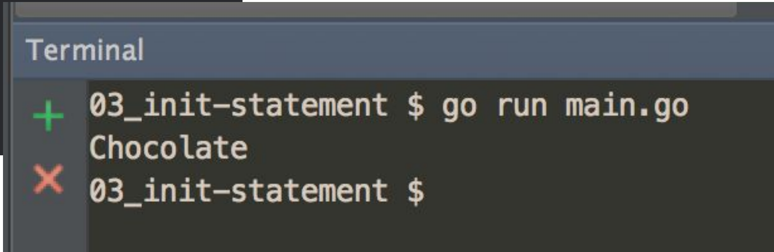

Terminal

```
02_not-exclamation $ go run main.go
second statement ran -  false
02_not-exclamation $
```



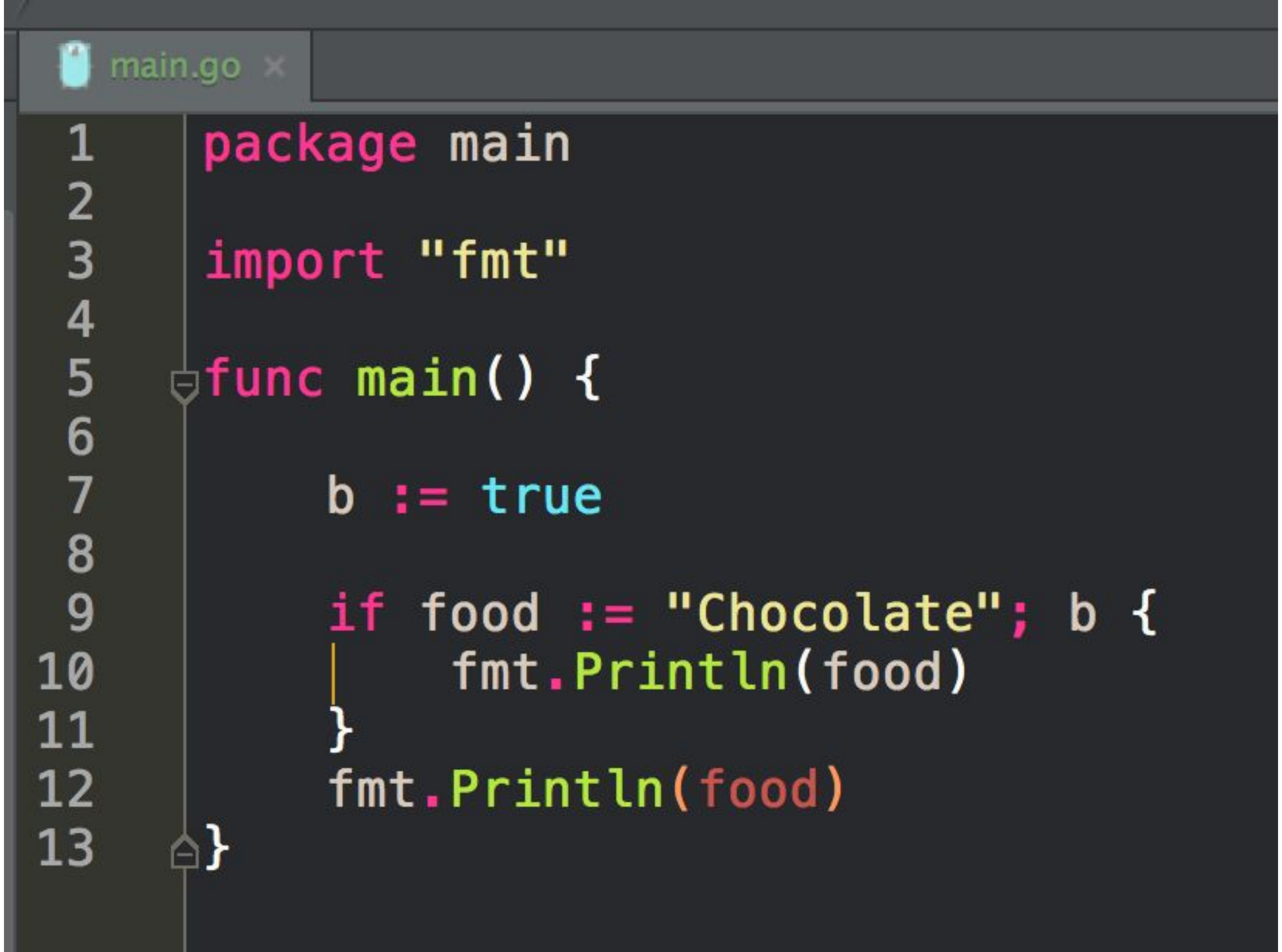
main.go

```go
package main

import "fmt"

func main() {

	b := true

	if food := "Chocolate"; b {
		fmt.Println(food)
	}

}
```



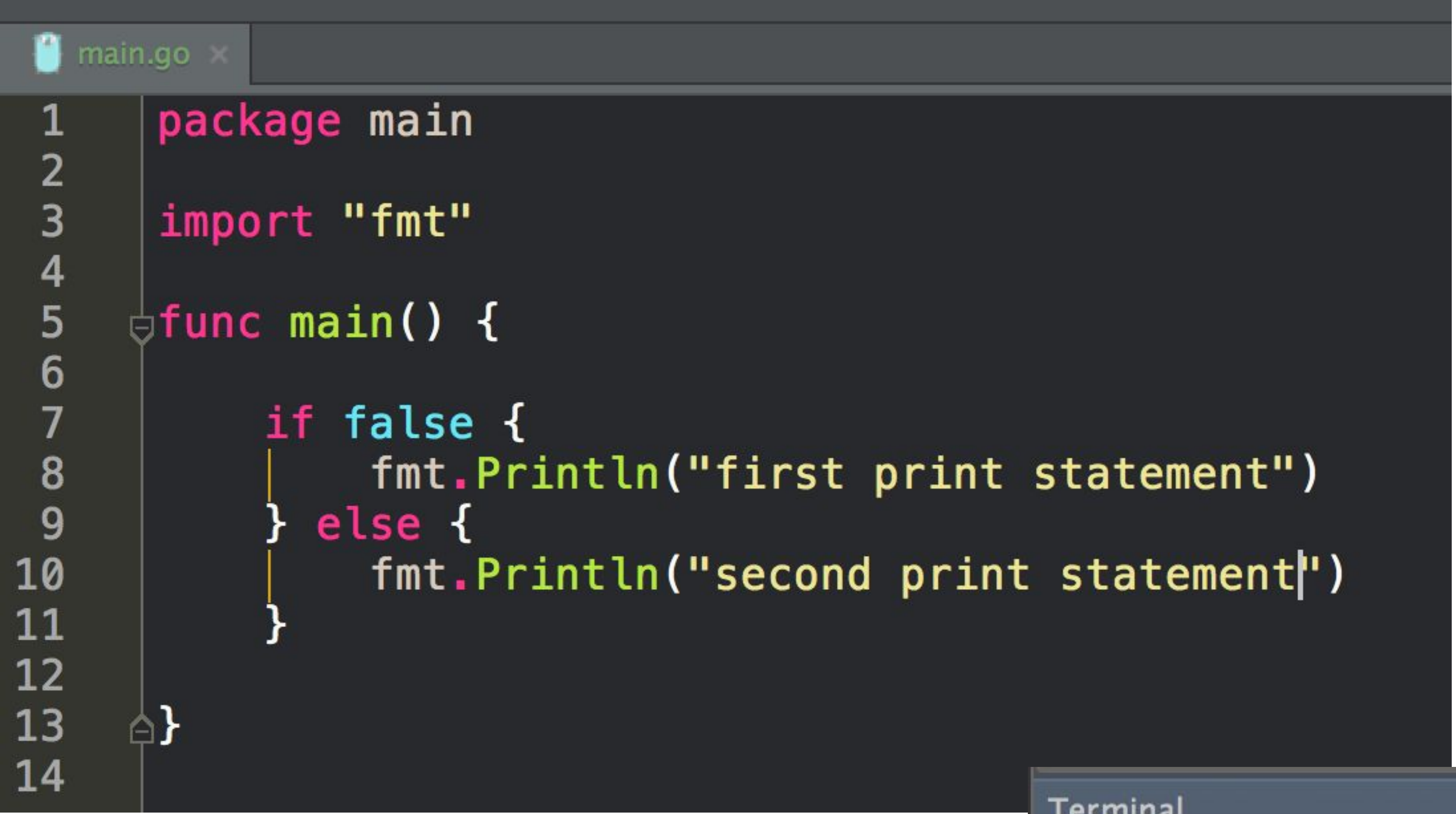
Terminal

```
03_init-statement $ go run main.go
Chocolate
03_init-statement $
```

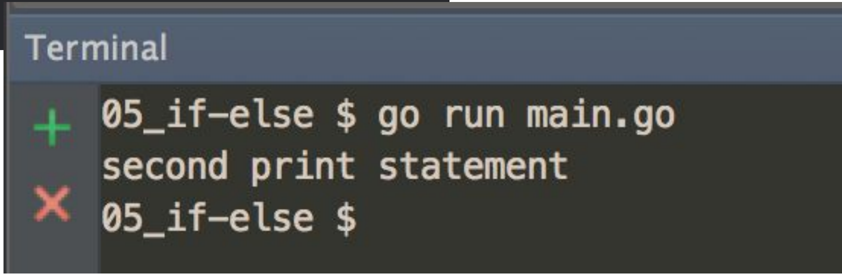


main.go

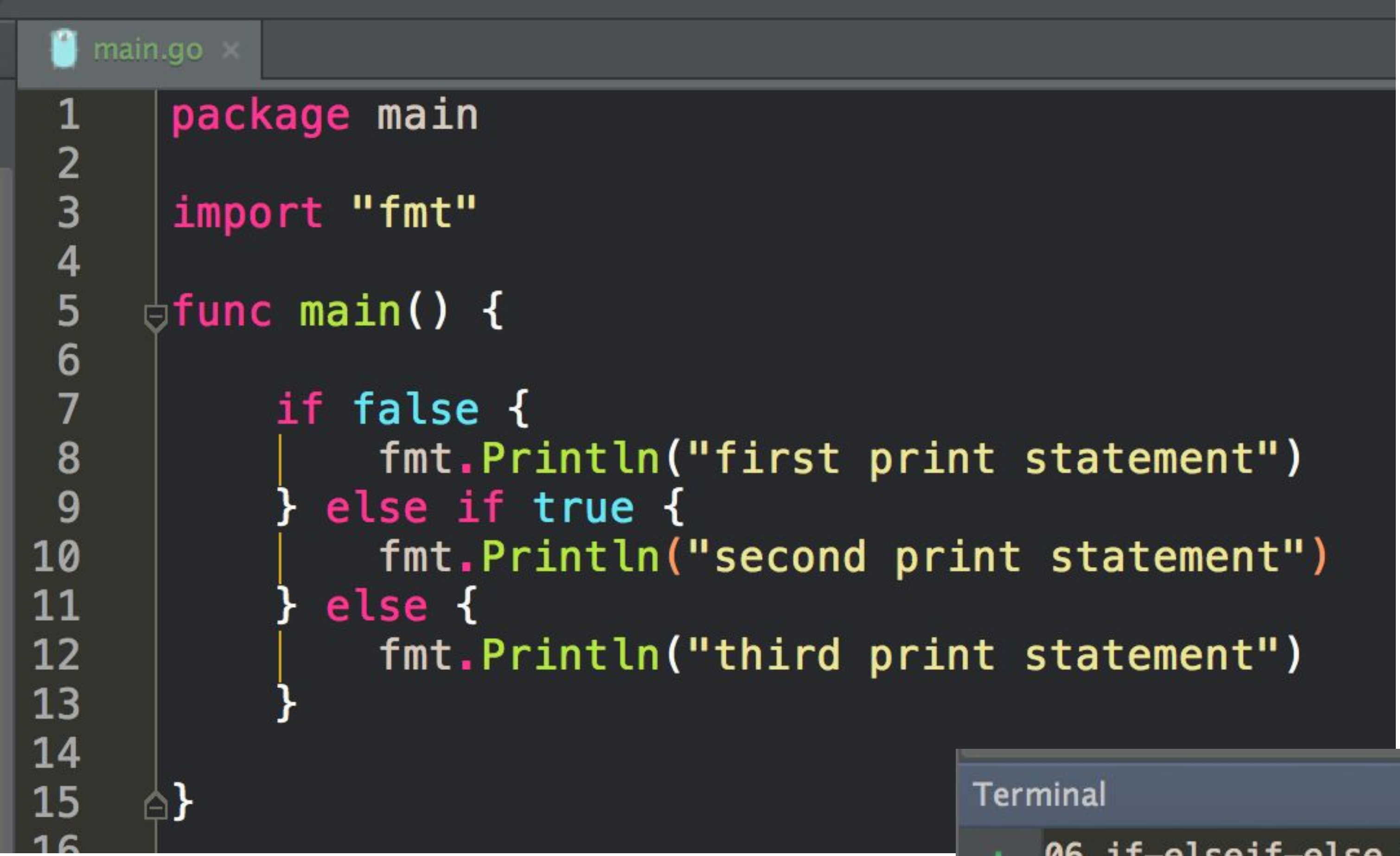
```go
package main

import "fmt"

func main() {

	b := true

	if food := "Chocolate"; b {
		fmt.Println(food)
	}
	fmt.Println(food)
}
```


main.go

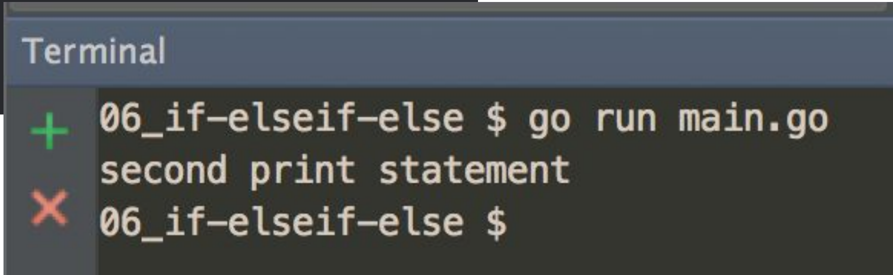
```go
package main

import "fmt"

func main() {

	if false {
		fmt.Println("first print statement")
	} else {
		fmt.Println("second print statement")
	}

}
```


Terminal

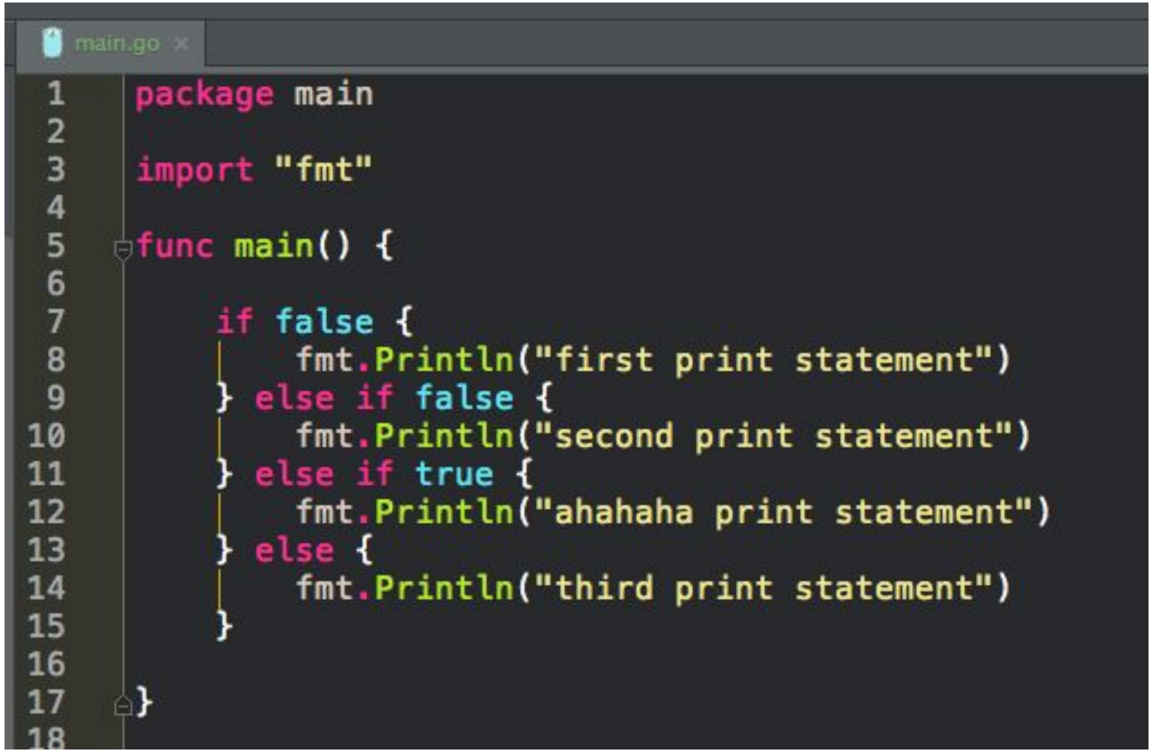
```
05_if-else $ go run main.go
second print statement
05_if-else $
```


main.go

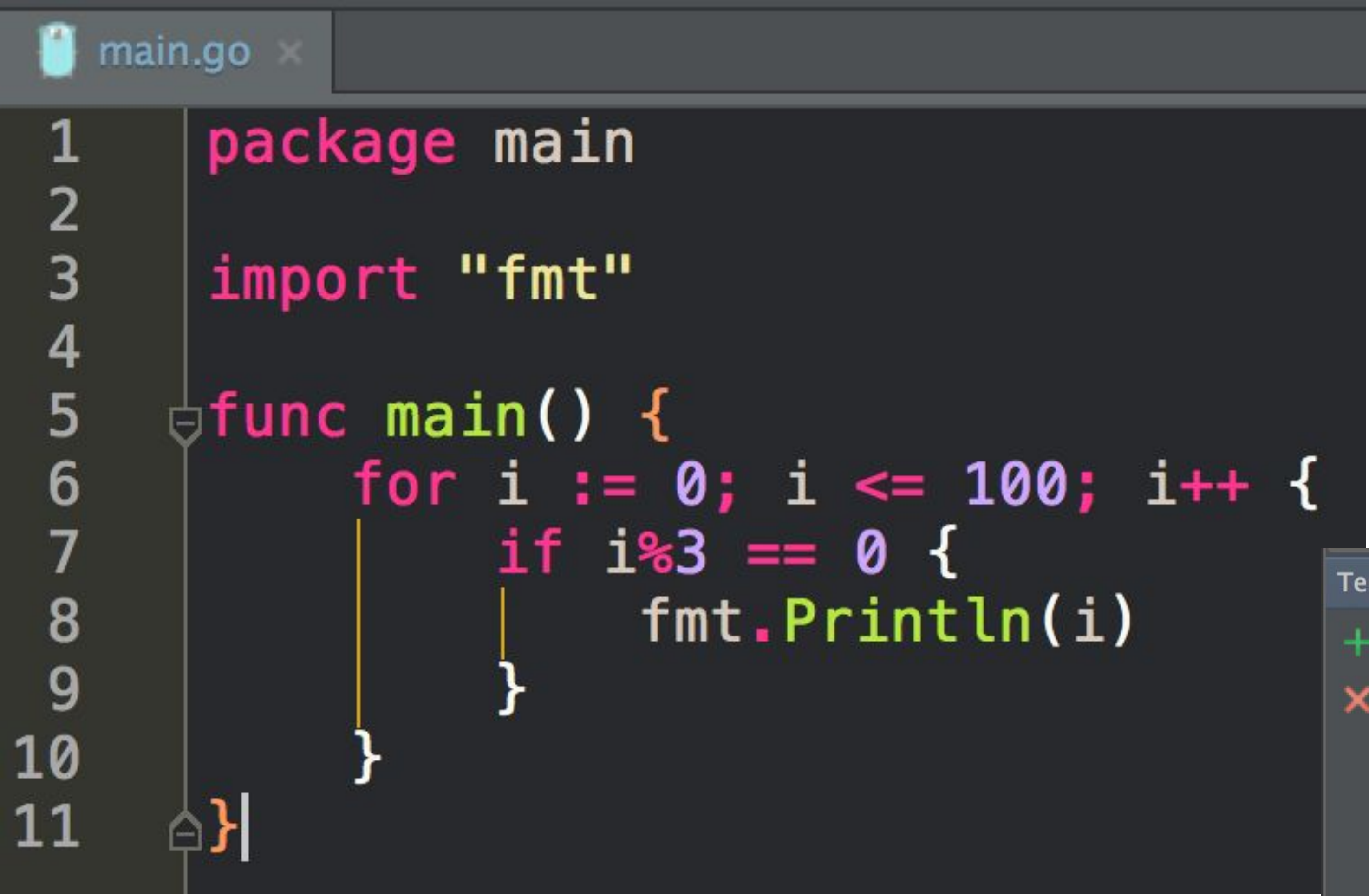
```go
package main

import "fmt"

func main() {

	if false {
		fmt.Println("first print statement")
	} else if true {
		fmt.Println("second print statement")
	} else {
		fmt.Println("third print statement")
	}

}
```


Terminal

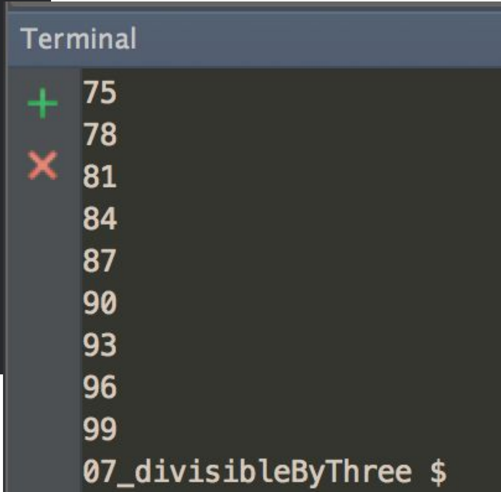
```
06_if-elseif-else $ go run main.go
second print statement
06_if-elseif-else $
```



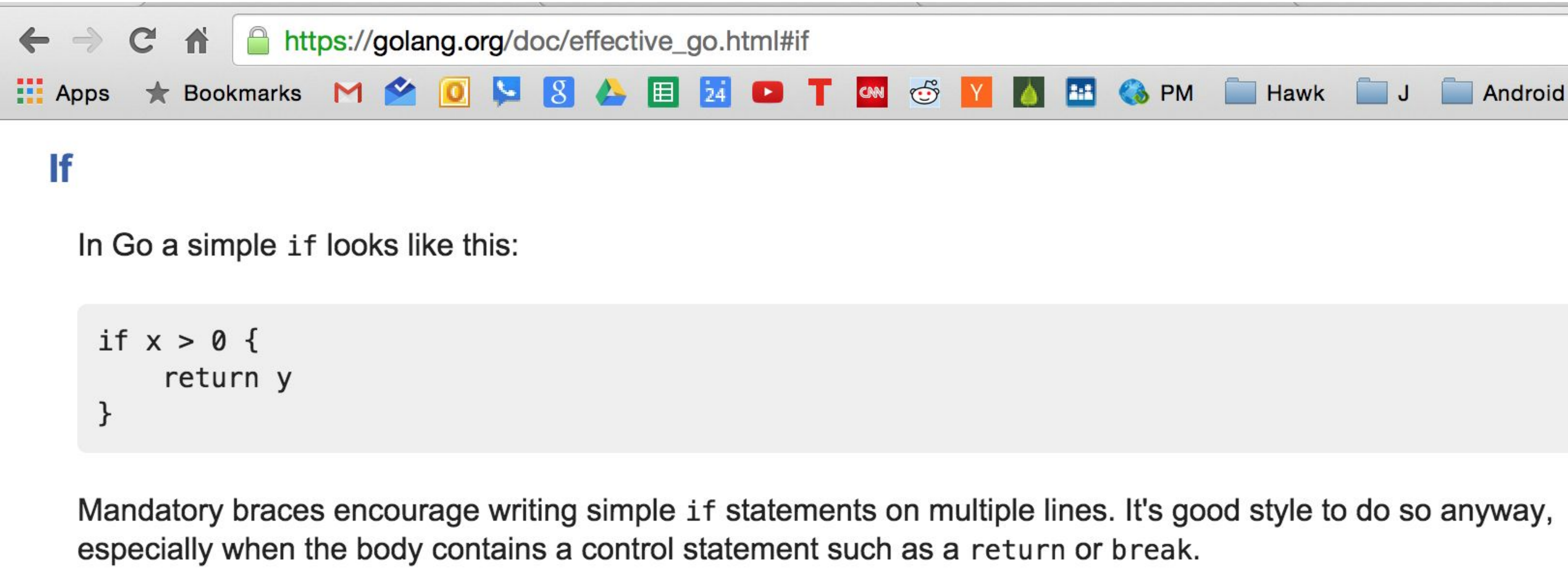
```go
package main

import "fmt"

func main() {

	if false {
		fmt.Println("first print statement")
	} else if false {
		fmt.Println("second print statement")
	} else if true {
		fmt.Println("ahahaha print statement")
	} else {
		fmt.Println("third print statement")
	}

}
```



```go
package main

import "fmt"

func main() {
	for i := 0; i <= 100; i++ {
		if i%3 == 0 {
			fmt.Println(i)
		}
	}
}
```



```
75
78
81
84
87
90
93
96
99
07_divisibleByThree $
```

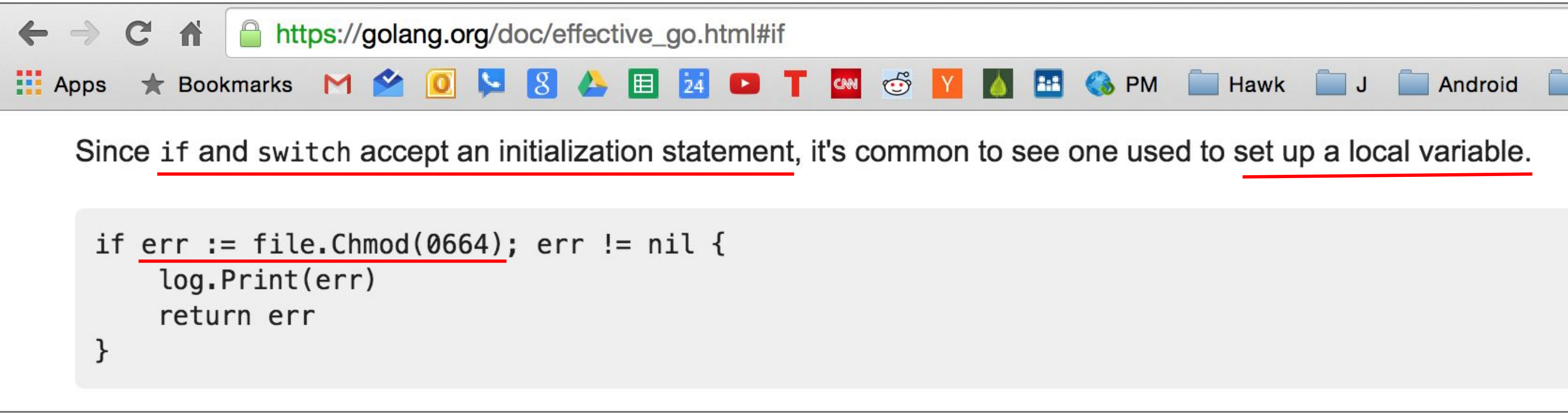


## If

In Go a simple `if` looks like this:

```
if x > 0 {
    return y
}
```

Mandatory braces encourage writing simple `if` statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a `return` or `break`.

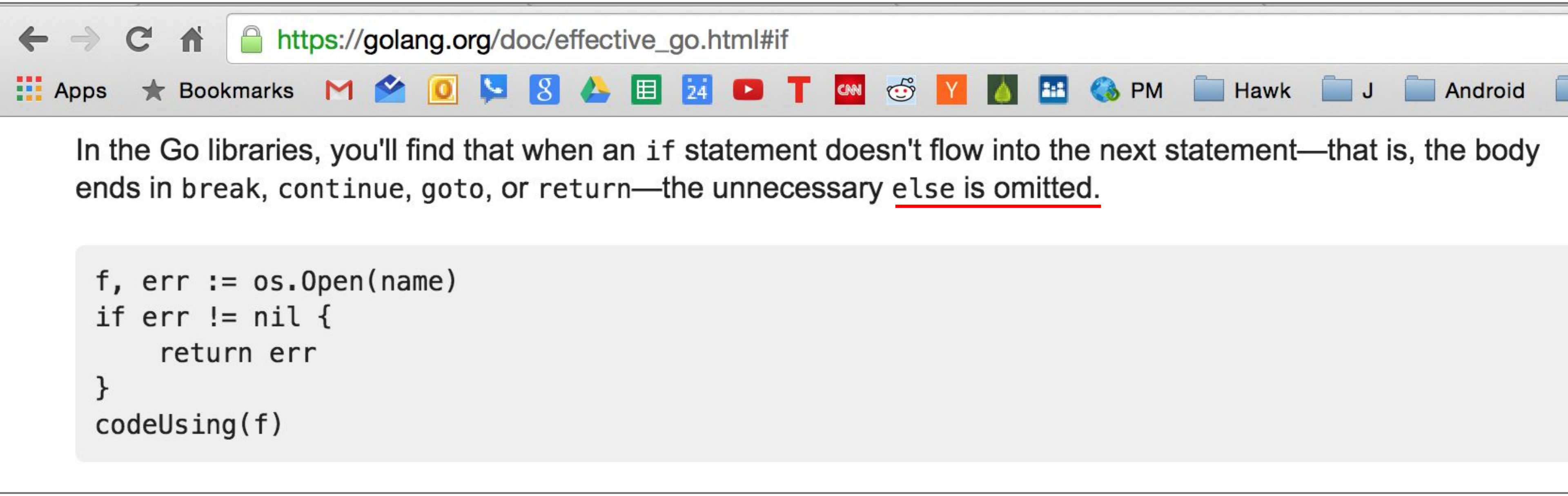


https://golang.org/doc/effective_go.html#if

Apps Bookmarks PM Hawk J Android

Since if and switch accept an initialization statement, it's common to see one used to set up a local variable.

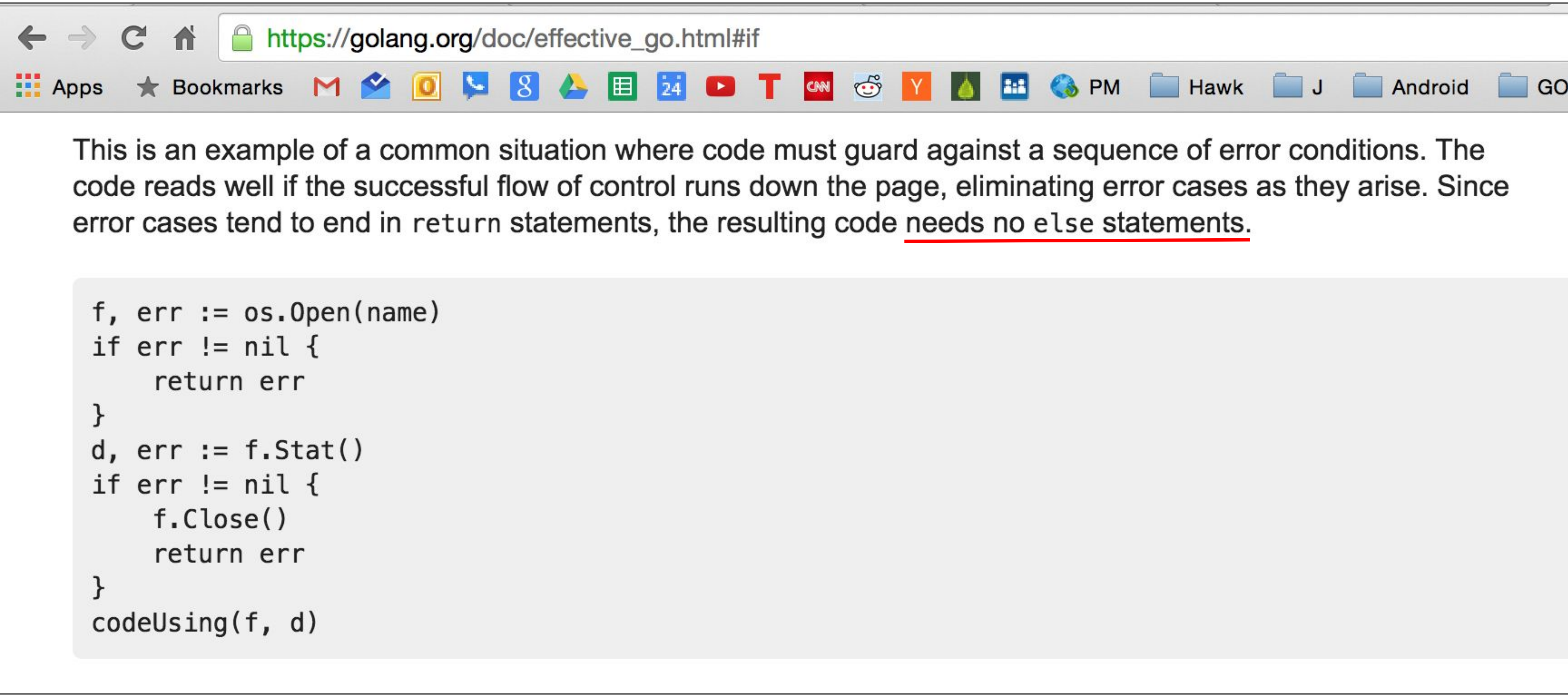
```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```


https://golang.org/doc/effective_go.html#if

Apps Bookmarks PM Hawk J Android

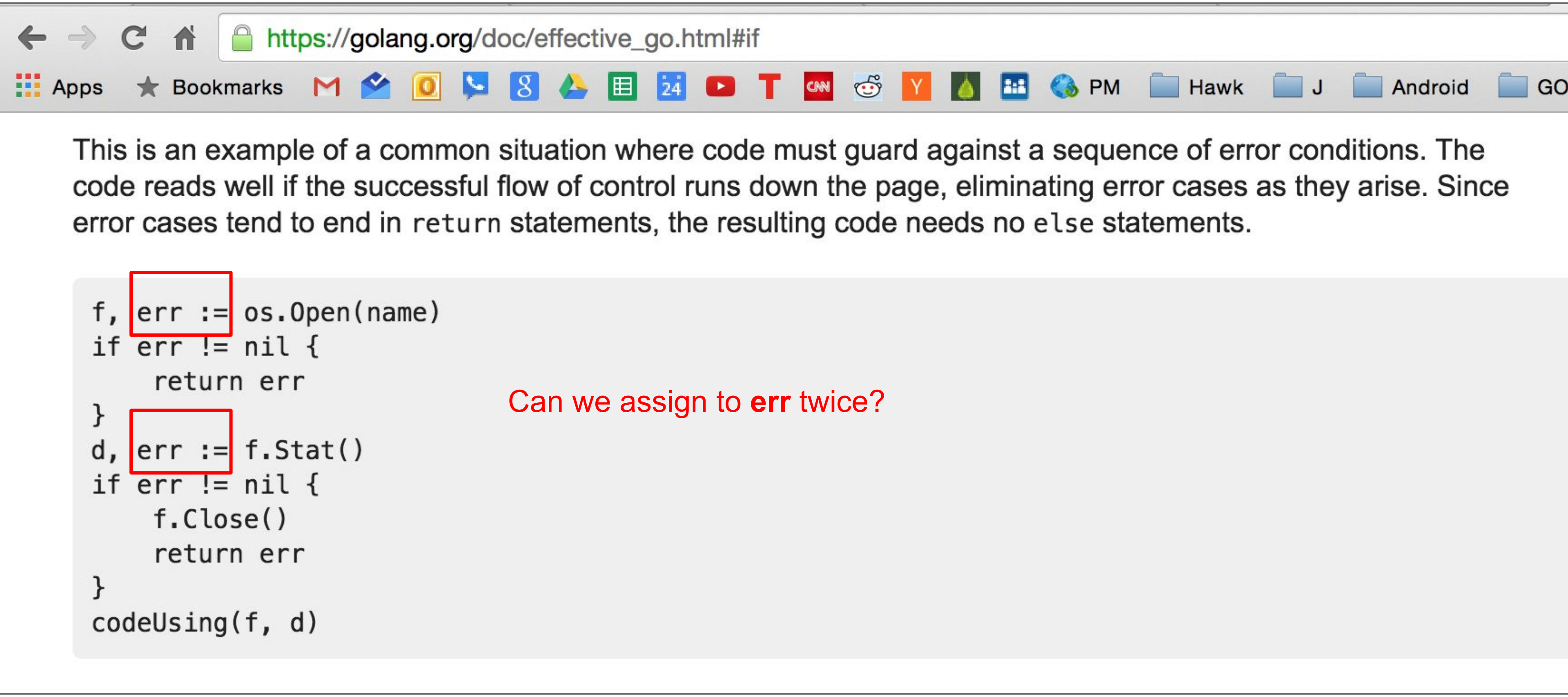
In the Go libraries, you'll find that when an `if` statement doesn't flow into the next statement—that is, the body ends in `break`, `continue`, `goto`, or `return`—the unnecessary `else` is omitted.

```
f, err := os.Open(name)
if err != nil {
    return err
}
codeUsing(f)
```



This is an example of a common situation where code must guard against a sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in `return` statements, the resulting code needs no `else` statements.

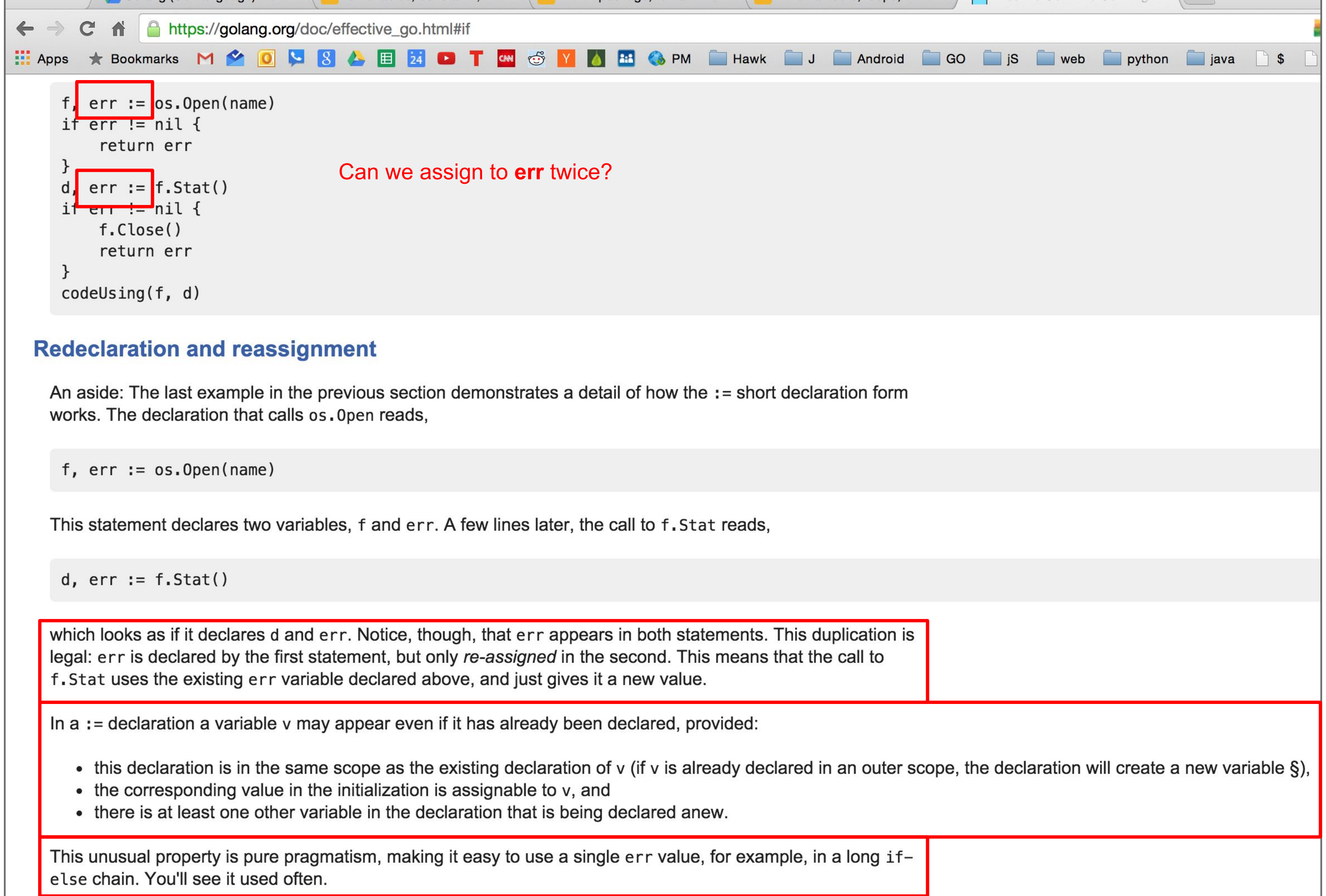
```go
f, err := os.Open(name)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```



This is an example of a common situation where code must guard against a sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in `return` statements, the resulting code needs no `else` statements.

```
f, err := os.Open(name)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

Can we assign to **err** twice?



```
f, err := os.Open(name)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

Can we assign to **err** twice?

## Redeclaration and reassignment

An aside: The last example in the previous section demonstrates a detail of how the `:=` short declaration form works. The declaration that calls `os.Open` reads,

```
f, err := os.Open(name)
```

This statement declares two variables, `f` and `err`. A few lines later, the call to `f.Stat` reads,

```
d, err := f.Stat()
```

which looks as if it declares `d` and `err`. Notice, though, that `err` appears in both statements. This duplication is legal: `err` is declared by the first statement, but only *re-assigned* in the second. This means that the call to `f.Stat` uses the existing `err` variable declared above, and just gives it a new value.

In a `:=` declaration a variable `v` may appear even if it has already been declared, provided:

- this declaration is in the same scope as the existing declaration of `v` (if `v` is already declared in an outer scope, the declaration will create a new variable §),
- the corresponding value in the initialization is assignable to `v`, and
- there is at least one other variable in the declaration that is being declared anew.

This unusual property is pure pragmatism, making it easy to use a single `err` value, for example, in a long `if-else` chain. You'll see it used often.

## exercise

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

## exercise

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

# Review

- remainder
  - %
- loops
  - for init; condition; post { }
  - for condition { }
  - for { }

  -
    ```
    for key, value := range oldMap {
        newMap[key] = value
    }
    ```

  -
    ```
    for key := range m {
        if key.expired() {
            delete(m, key)
        }
    }
    ```

  -
    ```
    sum := 0
    for _, value := range array {
        sum += value
    }
    ```

  -
    ```
    for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
        a[i], a[j] = a[j], a[i]
    }
    ```
- switch
  - no default fallthrough
    - no “break” needed
  - “fallthrough” can be added
- if

  -
    ```
    if x > 0 {
        return y
    }
    ```

  -
    ```
    if err := file.Chmod(0664); err != nil {
        log.Print(err)
        return err
    }
    ```

## Review Questions

## remainder

We use **%** to find a remainder in go. Is **%** an **operator** or an **operand**?

## loop

Write a program that uses all three of these loops:

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

For the last “for” make sure it includes a “break”.

Take a screenshot of your code and the results to submit for credit.

www.gobyexample.com is helpful

## loop - range

Provide the syntax for looping over a map using **range**.