The Ultimate Go Book

The best way to learn Go programming language and its internal

by Hoanh An, thanks to Bill Kennedy and the Ardan Labs team

Table of Contents

Table of Contents 1

Introduction 5

Language Mechanics 6

- Syntax 6
  - Variable 6
    - Built-in types 6
    - Zero value concept 6
    - Declare and initialize 7
    - Conversion vs casting 8
  - Struct 8
    - Declare and initialize 8
    - Name type vs anonymous type 10
  - Pointer 11
    - Everything is about pass by value 11
    - Pass by value 11
  - Escape analysis 12
  - Function 15
  - Constant 17
    - Declare and initialize 17
    - iota 19
- Data Structures 20
  - Array 20
    - CPU Cache 20
    - Translation Lookaside Buffer 21
    - Declare and initialize 21
    - What is the cost? 22
    - Iterate over the array of strings 23
    - Different type arrays 24
    - Contiguous memory allocations 24
  - Slice 25
    - Declare and initialize 25
    - Length vs Capacity 25
    - Reference type 26
    - Idea of appending: making slice a dynamic data structure 27
    - Slice of slice 30
    - Copy a slice 31
    - Slice and reference 32
    - UTF-8 33

Map 34
Declare and initialize 35
Map literals 35
Delete key 36
Find key 36
Map key restrictions 36
Decoupling 37
Method 37
Value and Pointer Receiver Call 37
Value and Pointer Semantics 39
Methods are just functions 43
Interface 46
Valueless type 46
Interface via pointer receiver 50
Slice of interface 53
Embedding 55
Declaring fields, NOT Embedding 55
Embedding types 56
Embedded type and Interface 57
Outer and inner type implementing the same interface 58
Exporting 60
Guideline 60
Exported identifier 60
Accessing a value of an unexported identifier 61
Unexported fields from an exported struct 62
Exported types with embedded unexported types 63
Software Design **65**
Composition 65
Grouping types 65
Grouping By State 65
Grouping By Behavior 67
Decoupling 70
Struct Composition 70
Decoupling With Interface 74
Interface Composition 79
Decoupling With Interface Composition 84
Conversion 89
Interface Conversions 89
Type assertion 91
Runtime Type Assertion 92
Interface Pollution 94

Remove Interface Pollution 96
Mocking 98
Package To Mock 98
Client 100
Error Handling 101
Default error values 101
Error variables 103
Type as context 105
Behavior as context 107
Finding the bug 110
Wrapping Errors 111

Concurrency **114**
Mechanics 114
Goroutine 114
Language Mechanics 121
Sequence 123
Goroutine time slicing 124
Goroutines and parallelism 126
Data race 128
Race Detection 128
Atomic Functions 131
Mutexes 133
Read/Write Mutex 135
Channel 139
Language Mechanics 139
Unbuffered channel: Signaling with data 141
Unbuffered channel: Double signal 143
Unbuffered channel: select and receive 144
Unbuffered channel: select and send 145
Buffered channel: Select and drop 146
Unbuffered channel (Tennis match) 148
Unbuffered channel (Replay race) 150
Buffered channel: Fan Out 153
Select 157
Patterns 164
Context 164
Store and retrieve values from a context 164
WithCancel 166
WithDeadline 168
WithTimeout 169
Request/Response 171

Testing and Profiling **173**
- Testing 173
  - Basic Unit Test 173
    - Table Test 176
    - Sub Test 177
    - Web Server 181
    - Mock Server 187
- Benchmarking 191
  - Basic Benchmark 191
  - Sub Benchmark 192
- Profiling 193
  - Stack Trace 193
    - Review Stack Trace 193
    - Packing 195
  - GODEBUG 196
    - Memory Tracing 196

Keep in touch **199**

Introduction

A year ago, in August 2019, I shared one of my projects called Ultimate Go on GitHub, https://github.com/hoanhan101/ultimate-go, and surprisingly, it got a lot of attention from the community. Fast forward to August 2020, it now has over 12K stars, 900 folks with the help of more than 20 contributors.

The project is a collection of my notes while learning Go programming language from Ardan Labs's Ultimate Go course. Honestly, I couldn't thank Bill Kennedy and the Ardan Labs team enough for open-sourcing this wonderful material. They've done such great jobs putting together their knowledge, insights into their courses and making them available to the public.

Different people have different learning styles. For me, I learn best by doing and walking through examples. That said, I take notes carefully, comment directly on the source code to make sure that I understand every single line of code as I am reading and also be mindful of the theories behind the scene.

As Ultimate Go keeps growing, there's one issue that keeps coming up. That's about the format of the project. Specifically, many people have requested an ebook version where the content is more streamlined and they can read it at their convenience.

That's when The Ultimate Go Book is born. For the last 3 months or so, I have spent most of my free time putting together everything from Ultimate Go into a 200-page book. Other than all the good stuff from Ultimate Go, two new and better things in this version are:

- Follow-along code input and output.
- Diagrams.

Hope it makes your journey of learning Go a bit easier. And again, thank you all for your support. I really appreciate it.

Happy reading!

Language Mechanics

Syntax

Variable

Built-in types

Type provides integrity and readability by asking 2 questions:

- What is the amount of memory that we allocate? (e.g. 32-bit, 64-bit)
- What does that memory represent? (e.g. int, uint, bool,..)

Type can be specific such as int32 or int64. For example,

- uint8 contains a base 10 number using one byte of memory
- int32 contains a base 10 number using 4 bytes of memory.

When we declare a type without being very specific, such as uint or int, it gets mapped based on the architecture we are building the code against. On a 64-bit OS, int will map to int64. Similarly, on a 32 bit OS, it becomes int32.

Word size

The word size is the number of bytes in a word, which matches our address size. For example, in 64-bit architecture, the word size is 64 bit (8 bytes), address size is 64 bit then our integer should be 64 bit.

Zero value concept

Every single value we create must be initialized. If we don't specify it, it will be set to the zero value. The entire allocation of memory, we reset that bit to 0.

Type	**Zero value**
Boolean	false
Integer	0
Floating Point	0
Complex	0i

String	“”
Pointer	nil

Declare and initialize

var is the only guarantee to initialize a zero value for a type.

```
var a int
var b string
var c float64
var d bool

fmt.Printf("var a int \t %T [%v]\n", a, a)
fmt.Printf("var b string \t %T [%v]\n", b, b)
fmt.Printf("var c float64 \t %T [%v]\n", c, c)
fmt.Printf("var d bool \t %T [%v]\n\n", d, d)
```

```
var a int         int [0]
var b string      string []
var c float64     float64 [0]
var d bool        bool [false]
```

Strings are a series of uint8 types.

A string is a two-word data structure: the first word represents a pointer to a backing array, the second word represents its length. If it is a zero value then the first word is nil, the second word is 0.

Using the short variable declaration operator, we can define and initialize at the same time.

```
aa := 10
bb := "hello" // 1st word points to a array of characters, 2nd word is 5
bytes
cc := 3.14159
dd := true

fmt.Printf("aa := 10 \t %T [%v]\n", aa, aa)
fmt.Printf("bb := \"hello\" \t %T [%v]\n", bb, bb)
fmt.Printf("cc := 3.14159 \t %T [%v]\n", cc, cc)
```

```go
fmt.Printf("dd := true \t %T [%v]\n\n", dd, dd)
```

```
aa := 10         int [10]
bb := "hello"    string [hello]
cc := 3.14159    float64 [3.14159]
dd := true       bool [true]
```

Conversion vs casting

Go doesn't have casting, but conversion. Instead of telling a compiler to pretend to have some more bytes, we have to allocate more memory like so.

```go
aaa := int32(10)
fmt.Printf("aaa := int32(10) %T [%v]\n", aaa, aaa)
```

```
aaa := int32(10) int32 [10]
```

Struct

example represents a type with different fields.

```go
type example struct {
      flag    bool
      counter int16
      pi      float32
}
```

Declare and initialize

Declare a variable of type example set to its zero value.

```go
var e1 exa

fmt.Printf("%+v\n", e1)
```

```
{flag:false counter:0 pi:0}
```

How much memory do we allocate for *example*?

A bool is 1 byte, int16 is 2 bytes, float32 is 4 bytes. Putting together, we have 7 bytes. However, the actual answer is 8. That leads us to a new concept of padding and alignment. The padding byte is sitting between the bool and the int16. The reason is because of alignment.

The idea of alignment: It is more efficient for this piece of hardware to read memory on its alignment boundary. We will take care of the alignment boundary issues so the hardware people don't.

Rule 1:

Depending on the size of a particular value, Go determines the alignment we need. Every 2 bytes value must follow a 2 bytes boundary. Since the bool value is only 1 byte and starts at address 0, then the next int16 must start on address 2. The byte at the address that gets skipped over becomes 1-byte padding. Similarly, if it is a 4-byte value then we will have a 3-byte padding value.

Rule 2:

The largest field represents the padding for the entire struct. We need to minimize the amount of padding as possible. Always lay out the field from highest to smallest. This will push any padding down to the bottom. In this case, the entire struct size has to follow a 8 bytes value because int64 is 8 bytes.

```go
type example struct {
    counter int64
    pi      float32
    flag    bool
}
```

Declare a variable of type example and init using a struct literal. Every line must end with a comma.

```go
e2 := example{
    flag:    true,
    counter: 10,
    pi:      3.141592,
}

fmt.Println("Flag", e2.flag)
fmt.Println("Counter", e2.counter)
fmt.Println("Pi", e2.pi)
```

```
Counter 10
Pi 3.141592
Flag true
```

Declare a variable of an anonymous type and init using a struct literal. This is a one-time thing.

```go
e3 := struct {
    flag    bool
    counter int16
    pi      float32
}{
    flag:    true,
    counter: 10,
    pi:      3.141592,
}

fmt.Println("Flag", e3.flag)
fmt.Println("Counter", e3.counter)
fmt.Println("Pi", e3.pi)
```

```
Flag true
Counter 10
Pi 3.141592
```

Name type vs anonymous type

If we have two name type identical struct, we can't assign one to another. For example, example1 and example2 are identical struct and we initialize var ex1 example1, var ex2 example2. Letting ex1 = ex2 is not allowed. We have to explicitly say that ex1 = example1(ex2) by performing a conversion. However, if ex is a value of identical anonymous struct type (like e3 above), then it is possible to assign ex1 = ex.

```go
var e4 example
e4 = e3
fmt.Printf("%+v\n", e4)
```

```
{flag:true counter:10 pi:3.141592}
```

Pointer

Everything is about pass by value

Pointers serve only 1 purpose: sharing. Pointers share values across the program boundaries. There are several types of program boundaries. The most common one is between function calls. We can also have a boundary between Goroutines which we will discuss later.

When this program starts up, the runtime creates a Goroutine. Every Goroutine is a separate path of execution that contains instructions that needed to be executed by the machine. We can also think of Goroutines as lightweight threads. This program has only 1 Goroutine: the main Goroutine.

Every Goroutine is given a block of memory, called the stack. The stack memory in Go starts out at 2K. It is very small. It can change over time. Every time a function is called, a piece of stack is used to help that function run. The growing direction of the stack is downward.

Every function is given a stack frame, memory execution of a function. The size of every stack frame is known at compile time. No value can be placed on a stack unless the compiler knows its size ahead of time. If we don't know the size of something at compile time, it has to be on the heap.

Zero value enables us to initialize every stack frame that we take. Stacks are self cleaning. We clean our stack on the way down. Every time we make a function, zero value initialization cleans the stack frame. We leave that memory on the way up because we don't know if we would need that again.

Pass by value

Declare variable of type int with a value of 10. This value is put on a stack with a value of 10.

```go
count := 10

// To get the address of a value, we use &.
println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

// Pass the "value of" count.
increment1(count)

// Printing out the result of count. Nothing has changed.
```

```
println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

// Pass the "address of" count. This is still considered pass by value,
not by reference because the address itself is a value.
increment2(&count)

// Printing out the result of count. count is updated.
println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")
```

```
func increment1(inc int) {
      // Increment the "value of" inc.
      inc++
      println("inc1:\tValue Of[", inc, "]\tAddr Of[", &inc, "]")
}

// increment2 declares count as a pointer variable whose value is always
an address and points to values of type int. The * here is not an
operator. It is part of the type name. Every type that is declared,
whether you declare or it is predeclared, you get for free a pointer.
func increment2(inc *int) {
      // Increment the "value of" count that the "pointer points to".
The * is an operator. It tells us the value of the pointer points to.
      *inc++
      println("inc2:\tValue Of[", inc, "]\tAddr Of[", &inc, "]\tValue
Points To[", *inc, "]")
}
```

```
count:  Value Of[ 10 ]  Addr Of[ 0xc000050738 ]
inc1:   Value Of[ 11 ]  Addr Of[ 0xc000050730 ]
count:  Value Of[ 10 ]  Addr Of[ 0xc000050738 ]
inc2:   Value Of[ 0xc000050738 ]        Addr Of[ 0xc000050748 ] Value
Points To[ 11 ]
```

Escape analysis

stayOnStack shows how the variable does not escape. Since we know the size of the user value at compile time, the compiler will put this on a stack frame.

```
// user represents an user in the system.
type user struct {
```

```go
      name  string
      email string
}

func stayOnStack() user {
      // In the stack frame, create a value and initialize it.
      u := user{
            name:  "Hoanh An",
            email: "hoanhan101@gmail.com",
      }

      // Return the value, pass back up to the main stack frame.
      return u
}
```

escapeToHeap shows how the variable escapes. This looks almost identical to the stayOnStack function. It creates a value of type user and initializes it. It seems like we are doing the same here. However, there is one subtle difference: we do not return the value itself but the address of u. That is the value that is being passed back up the call stack. We are using pointer semantics.

You might think about what we have after this call is: main has a pointer to a value that is on a stack frame below. If this is the case, then we are in trouble. Once we come back up the call stack, this memory is there but it is reusable again. It is no longer valid. Anytime now main makes a function call, we need to allocate the frame and initialize it.

Think about zero value for a second here. It enables us to initialize every stack frame that we take. Stacks are self cleaning. We clean our stack on the way down. Every time we make a function call, zero value, initialization, we are cleaning those stack frames. We leave that memory on the way up because we don't know if we need that again.

Back to the example. It is bad because it looks like we take the address of user value, pass it back up to the call stack giving us a pointer which is about to get erased. However, that is not what will happen.

What is actually going to happen is escape analysis. Because of the line "return &u", this value cannot be put inside the stack frame for this function so we have to put it out on the heap.

Escape analysis decides what stays on the stack and what does not. In the stayOnStack function, because we are passing the copy of the value itself, it is safe to keep these things on the stack. But when we SHARE something above the call stack like this, escape analysis says this memory is no longer valid when we get back to main, we must put it out there on the heap. main will end up having a pointer to the heap. In fact, this allocation happens immediately on the heap. escapeToHeap is gonna have a pointer to the heap. But u is gonna base on value semantics.

```go
func escapeToHeap() *user {
	u := user{
		name:  "Hoanh An",
		email: "hoanhan101@gmail.com",
	}

	return &u
}
```

What if we run out of stack space?

What happens next is during that function call, there is a little preamble that asks "Do we have enough stack space for this frame?". If yes then no problem because at compile time we know the size of every frame. If not, we have to have a bigger frame and these values need to be copied over. The memory on that stack moves. It is a trade off. We have to take the cost of this copy because it doesn't happen a lot. The benefit of using less memory in any Goroutine outweighs the cost.

Because the stack can grow, no Goroutine can have a pointer to some other Goroutine stack. There would be too much overhead for the compiler to keep track of every pointer. The latency will be insane.

The stack for a Goroutine is only for that Goroutine. It cannot be shared between Goroutines.

Garbage collection

Once something is moved to the heap, Garbage Collection has to get in. The most important thing about the Garbage Collector (GC) is the pacing algorithm. It determines the frequency/pace that the GC has to run in order to maintain the smallest possible t.

Imagine a program where you have a 4 MB heap. GC is trying to maintain a live heap of 2 MB. If the live heap grows beyond 4 MB, we have to allocate a larger heap. The pace the GC runs at depends on how fast the heap grows in size. The slower the pace, the less impact it is going to have. The goal is to get the live heap back down.

When the GC is running, we have to take a performance cost so all Goroutines can keep running concurrently. The GC also has a group of Goroutines that perform the garbage collection work. It uses 25% of our available CPU capacity for itself. More details about GC and pacing algorithm can be find at: https://docs.google.com/document/d/1wmjrocXIWTr1JxU-3EQBI6BK6KgtiFArkG47XK73xIQ/edit?usp=sharing

Function

```go
// user is a struct type that declares user information.
type user struct {
	ID   int
	Name string
}

// updateStats provides update stats.
type updateStats struct {
	Modified int
	Duration float64
	Success  bool
	Message  string
}

func main() {
	// Retrieve the user profile.
	u, err := retrieveUser("Hoanh")
	if err != nil {
		fmt.Println(err)
	return
	}

	// Display the user profile. Since the returned u is an address,
use * to get the value.
	fmt.Printf("%+v\n", *u)

	// Update user name. Don't care about the update stats. This _ is
called a blank identifier. Since we don't need anything outside the
scope of if, we can use the compact syntax.
	if _, err := updateUser(u); err != nil {
		fmt.Println(err)
	return
	}

	// Display the update was successful.
	fmt.Println("Updated user record for ID", u.ID)
}
```

retrieveUser retrieves the user document for the specified user. It takes a string type name and returns a pointer to a user type value and bool type error.

```go
func retrieveUser(name string) (*user, error) {
	// Make a call to get the user in a json response.
	r, err := getUser(name)
	if err != nil {
		return nil, err
	}

	// Create a value type user to unmarshal the json document into.
	var u user

	// Share the value down the call stack, which is completely safe
so the Unmarshal function can read the document and initialize it.
	err = json.Unmarshal([]byte(r), &u)

	// Share it back up the call stack. Because of this line, we know
that this creates an allocation. The value is the previous step is not
on the stack but on the heap.
	return &u, err
}
```

getUser simulates a web call that returns a json document for the specified user.

```go
func getUser(name string) (string, error) {
	response := `{"ID":101, "Name":"Hoanh"}`
	return response, nil
}
```

updateUser updates the specified user document

```go
func updateUser(u *user) (*updateStats, error) {
	// response simulates a JSON response.
	response := `{"Modified":1, "Duration":0.005, "Success" : true,
"Message": "updated"}`

	// Unmarshal the JSON document into a value of the userStats
struct type.
	var us updateStats
	if err := json.Unmarshal([]byte(response), &us); err != nil {
```

```
            return nil, err
      }

      // Check the update status to verify the update is successful.
      if us.Success != true {
            return nil, errors.New(us.Message)
      }

      return &us, nil
}
```

```
{ID:101 Name:Hoanh}
Updated user record for ID 101
```

Constant

Constants are not variables. Constants have a parallel type system all to themselves. The minimum precision for constant is 256 bit. They are considered to be mathematically exact. Constants only exist at compile time.

Declare and initialize

Constants can be typed or untyped. When it is untyped, we consider it as a kind. They are implicitly converted by the compiler.

Untyped constants.

```
const ui = 12345    // kind: integer
const uf = 3.141592 // kind: floating-point
```

Typed constants still use the constant type system but their precision is restricted.

```
const ti int = 12345        // type: int
const tf float64 = 3.141592 // type: float64
```

This doesn't work because constant 1000 overflows uint8.

```
const myUint8 uint8 = 1000
```

Constant arithmetic supports different kinds. Kind Promotion is used to determine kind in these scenarios. All of this happens implicitly.

Variable answer will be of type float64.

```
var answer = 3 * 0.333 // KindFloat(3) * KindFloat(0.333)
fmt.Println(answer)
```

```
0.999
```

Constant third will be of kind floating point.

```
const third = 1 / 3.0 // KindFloat(1) / KindFloat(3.0)
fmt.Println(third)
```

```
0.3333333333333333
```

Constant zero will be of kind integer.

```
const zero = 1 / 3 // KindInt(1) / KindInt(3)
fmt.Println(zero)
```

```
0
```

This is an example of constant arithmetic between typed and untyped constants. Must have like types to perform math.

```
const one int8 = 1
const two = 2 * one // int8(2) * int8(1)

fmt.Println(one)
fmt.Println(two)
```

```
1
2
```

Max integer value on 64 bit architecture.

```
const maxInt = 9223372036854775807
fmt.Println(maxInt)
```

```
9223372036854775807
```

bigger is a much larger value than int64 but still compiled because of the untyped system. 256 is a lot of space (depending on the architecture).

```
const bigger = 9223372036854775808543522345
```

However, biggerInt Will NOT compile because it exceeds 64 bit.

```
const biggerInt int64 = 9223372036854775808543522345
```

iota

```
const (
      A1 = iota // 0 : Start at 0
      B1 = iota // 1 : Increment by 1
      C1 = iota // 2 : Increment by 1
)

fmt.Println("1:", A1, B1, C1)

const (
      A2 = iota // 0 : Start at 0
      B2        // 1 : Increment by 1
      C2        // 2 : Increment by 1
)

fmt.Println("2:", A2, B2, C2)

const (
      A3 = iota + 1 // 1 : Start at 0 + 1
      B3            // 2 : Increment by 1
      C3            // 3 : Increment by 1
)

fmt.Println("3:", A3, B3, C3)

const (
      Ldate= 1 << iota // 1 : Shift 1 to the left 0.  0000 0001
      Ltime           //  2 : Shift 1 to the left 1.  0000 0010
      Lmicroseconds   //  4 : Shift 1 to the left 2.  0000 0100
      Llongfile       //  8 : Shift 1 to the left 3.  0000 1000
      Lshortfile      // 16 : Shift 1 to the left 4.  0001 0000
      LUTC            // 32 : Shift 1 to the left 5.  0010 0000
)

fmt.Println("Log:", Ldate, Ltime, Lmicroseconds, Llongfile, Lshortfile,
LUTC)
```

```
1: 0 1 2
2: 0 1 2
3: 1 2 3
Log: 1 2 4 8 16 32
```

Data Structures

Array

CPU Cache

Cores DO NOT access main memory directly but their local caches. What store in caches are data and instruction.

Cache speed from fastest to slowest: L1 -> L2 -> L3 -> main memory. As Scott Meyers has said "If performance matters then the total memory you have is the total amount of caches". Access to main memory is incredibly slow. Practically speaking it might not even be there.

So how do we write code that can be sympathetic with the caching system to make sure that we don't have a cache miss or at least, we minimize cache misses to our fullest potential?

Processor has a Prefetcher. It predicts what data is needed ahead of time. There are different granularities depending on where we are on the machine. Our programming model uses a byte. We can read and write to a byte at a time. However, from the caching system point of view, our granularity is not 1 byte. It is 64 bytes, called a cache line. All memory us junked up in this 64 bytes cache line. Since the caching mechanism is complex, Prefetcher tries to hide all the latency from us. It has to be able to pick up on predictable access patterns to data. That said, we need to write code that creates predictable access patterns to data.

One easy way is to create a contiguous allocation of memory and to iterate over them. The array data structure gives us the ability to do so. From the hardware perspective, array is the most important data structure. From Go perspective, slice is. Array is the backing data structure for slice (like Vector in C++). Once we allocate an array, whatever its size, every element is equally distant from other elements. As we iterate over that array, we begin to walk cache line by cache line. As the Prefetcher sees that access pattern, it can pick it up and hide all the latency from us.

For example, we have a big nxn matrix. We do LinkedList Traverse, Column Traverse, and Row Traverse and benchmark against them. Unsurprisingly, Row Traverse has the best performance. It walks through the matrix cache line by cache line and creates a predictable access pattern. Column Traverse does not walk through the matrix cache line by cache line. It looks like a random access memory pattern. That is why it is the slowest among those. However, that doesn't explain why LinkedList Traverse's performance is in the middle. We just think that it might perform as poorly as the Column Traverse. This leads us to another cache: TLB - Translation Lookaside Buffer. Its job is to maintain the operating system’s page and offset to where physical memory is.

Translation Lookaside Buffer

Back to the different granularity, the caching system moves data in and out the hardware at 64 bytes at a time. However, the operating system manages memory by paging its 4K (traditional page size for an operating system).

For every page that we are managing, let's take our virtual memory addresses because that we use (softwares runs virtual addresses, its sandbox, that is how we use/share physical memory) and map it to the right page and offset for that physical memory.

A miss on the TLB can be worse than just the cache miss alone. The LinkedList is somewhere in between because the chance of multiple nodes being on the same page is probably pretty good. Even though we can get cache misses because cache lines aren't necessary in the distance that is predictable, we probably do not have so many TLB cache misses. In the Column Traverse, not only do we have cache misses, we probably have a TLB cache miss on every access as well.

That said, data-oriented design matters. It is not enough to write the most efficient algorithm, how we access our data can have much more lasting effect on the performance than the algorithm itself.

Declare and initialize

Declare an array of five strings that is initialized to its zero value. To recap, a string is a 2 word data structure: a pointer and a length. Since this array is set to its zero value, every string in this array is also set to its zero value, which means that each string has the first word pointed to nil and the second word is 0.

nil	nil	nil	nil	nil
0	0	0	0	0

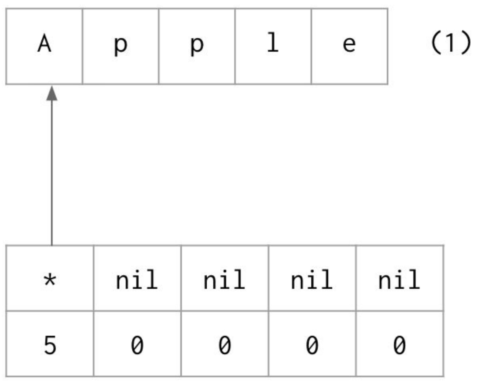
```go
var strings [5]string
```

At index 0, a string now has a pointer to a backing array of bytes (characters in string) and its length is 5.

What is the cost?

The cost of this assignment is the cost of copying 2 bytes. We have two string values that have pointers to the same backing array of bytes. Therefore, the cost of this assignment is just 2 words.

```go
strings[0] = "Apple"
```



Similarly, assign more values to the rest of the slice.

```go
strings[1] = "Orange"
strings[2] = "Banana"
strings[3] = "Grape"
strings[4] = "Plum"
```

Iterate over the array of strings

Using range, not only we can get the index but also a copy of the value in the array. fruit is now a string value; its scope is within the for statement. In the first iteration, we have the word "Apple". It is a string that has the first word also points to (1) and the second word is 5. So we now have 3 different string values all sharing the same backing array.

What are we passing to the Println function?

We are using value semantics here. We are not sharing our string value. Println is getting its own copy, its own string value. It means when we get to the Println call, there are now 4 string values all sharing the same backing array. We don't want to take the address of a string. We know the size of a string ahead of time. That means it has the ability to be on the stack because it is not creating allocation and also not causing pressure on the GC. The string has been designed to leverage value mechanics, to stay on the stack, out of the way of creating garbage. Hence, the only thing that has to be on the heap, if anything is the backing array, which is the one thing that being shared

```
fmt.Printf("\n=> Iterate over array\n")
for i, fruit := range strings {
      fmt.Println(i, fruit)
}
```

```
=> Iterate over array
0 Apple
1 Orange
2 Banana
3 Grape
4 Plum
```

Declare an array of 4 integers that is initialized with some values using literal syntax.

```
numbers := [4]int{10, 20, 30, 40}
```

Iterate over the array of numbers using traditional style.

```
fmt.Printf("\n=> Iterate over array using traditional style\n")
```

```
for i := 0; i < len(numbers); i++ {
    fmt.Println(i, numbers[i])
}
```

```
=> Iterate over array using traditional style
0 10
1 20
2 30
3 40
```

Different type arrays

Declare an array of 5 integers that is initialized to its zero value.

```
var five [5]int
```

Declare an array of 4 integers that is initialized with some values.

```
four := [4]int{10, 20, 30, 40}

fmt.Printf("\n=> Different type arrays\n")
fmt.Println(five)
fmt.Println(four)
```

```
=> Different type arrays
[0 0 0 0 0]
[10 20 30 40]
```

When we try to assign four to five like so five = four, the compiler says that "cannot use four (type [4]int) as type [5]int in assignment". This cannot happen because they have different types (size and representation). The size of an array makes up its type name: [4]int vs [5]int. Just like what we've seen with pointers. The * in *int is not an operator but part of the type name. Unsurprisingly, all arrays have known size at compile time.

Contiguous memory allocations

Declare an array of 6 strings initialized with values.

```
six := [6]string{"Annie", "Betty", "Charley", "Doug", "Edward", "Hoanh"}
```

Iterate over the array displaying the value and address of each element. By looking at the output of this Printf function, we can see that this array is truly a contiguous block of memory. We know a string is 2 words and depending on computer architecture, it will have x byte. The distance between two consecutive IndexAddr is exactly x byte. v is its own variable on the stack and it has the same address every single time.

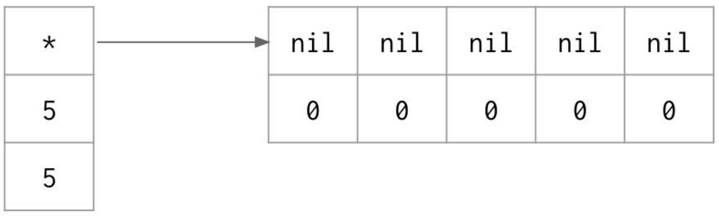
```
for i, v := range six {
      fmt.Printf("Value[%s]\tAddress[%p] IndexAddr[%p]\n", v, &v,
&six[i])
}
```

```
=> Contiguous memory allocations
Value[Annie]      Address[0xc000010250] IndexAddr[0xc000052180]
Value[Betty]      Address[0xc000010250] IndexAddr[0xc000052190]
Value[Charley]    Address[0xc000010250] IndexAddr[0xc0000521a0]
Value[Doug]       Address[0xc000010250] IndexAddr[0xc0000521b0]
Value[Edward]     Address[0xc000010250] IndexAddr[0xc0000521c0]
Value[Hoanh]      Address[0xc000010250] IndexAddr[0xc0000521d0]
```

Slice

Declare and initialize

Create a slice with a length of 5 elements. make is a special built-in function that only works with slice, map and channel. make creates a slice that has an array of 5 strings behind it. We are getting back a 3 word data structure: the first word points to the backing array, second word is length and third one is capacity.



Length vs Capacity

Length is the number of elements from this pointer position we have access to (read and write). Capacity is the total number of elements from this pointer position that exist in the backing array.

Because it uses syntactic sugar, it looks just like an array. It also has the same cost that we've seen in array. However, there’s one thing to be mindful about: there is no value in the bracket []string inside the make function. With that in mind, we can constantly notice that we are dealing with a slice, not array.

```
slice1 := make([]string, 5)
slice1[0] = "Apple"
slice1[1] = "Orange"
slice1[2] = "Banana"
slice1[3] = "Grape"
slice1[4] = "Plum"
```

We can't access an index of a slice beyond its length.

```
Error: panic: runtime error: index out of range slice1[5] = "Runtime
error"
```

We are passing the value of slice, not its address. So the Println function will have its own copy of the slice.

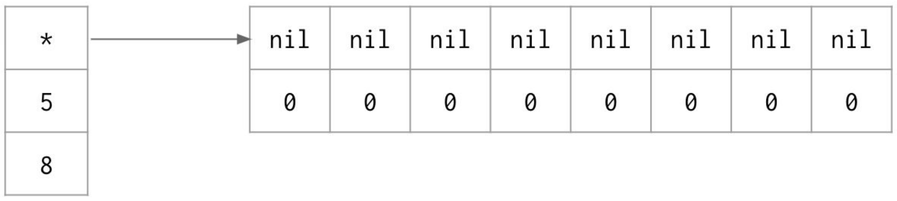
```
fmt.Printf("\n=> Printing a slice\n")
fmt.Println(slice1)
```

```
=> Printing a slice
[Apple Orange Banana Grape Plum]
```

Reference type

To create a slice with a length of 5 elements and a capacity of 8, we can use the keyword *make*. *make* allows us to adjust the capacity directly on construction of this initialization.

What we end up having now is a 3 word data structure where the first word points to an array of 8 elements, length is 5 and capacity is 8. It means that I can read and write to the first 5 elements and I have 3 elements of capacity that I can leverage later.



```go
slice2 := make([]string, 5, 8)
slice2[0] = "Apple"
slice2[1] = "Orange"
slice2[2] = "Banana"
slice2[3] = "Grape"
slice2[4] = "Plum"

fmt.Printf("\n=> Length vs Capacity\n")
inspectSlice(slice2)
```

```go
// inspectSlice exposes the slice header for review.
// Parameter: again, there is no value in side the []string so we want a
slice.
// Range over a slice, just like we did with array.
// While len tells us the length, cap tells us the capacity
// In the output, we can see the addresses are aligning as expected.
func inspectSlice(slice []string) {
    fmt.Printf("Length[%d] Capacity[%d]\n", len(slice), cap(slice))
    for i := range slice {
        fmt.Printf("[%d] %p %s\n", i, &slice[i], slice[i])
    }
}
```

```
=> Length vs Capacity
Length[5] Capacity[8]
[0] 0xc00007e000 Apple
[1] 0xc00007e010 Orange
[2] 0xc00007e020 Banana
[3] 0xc00007e030 Grape
[4] 0xc00007e040 Plum
```

Idea of appending: making slice a dynamic data structure

Declare a nil slice of strings, set to its zero value. 3 word data structure:

first one points to nil, second and last are zero.

```go
var data []string
```

What if I do data := string{}? Is it the same?

No, because data in this case is not set to its zero value. This is why we always use var for zero value because not every type when we create an empty literal we have its zero value in return. What actually happens here is that we have a slice but it has a pointer (as opposed to nil). This is considered an empty slice, not a nil slice. There is a semantic between a nil slice and an empty slice. Any reference type that is set to its zero value can be considered nil. If we pass a nil slice to a marshal function, we get back a string that says null but when we pass an empty slice, we get an empty JSON document. But where does that pointer point to? It is an empty struct, which we will review later.

Capture the capacity of the slice.

```go
lastCap := cap(data)
```

Append ~100k strings to the slice.

```go
for record := 1; record <= 102400; record++ {
```

Use the built-in function append to add to the slice. It allows us to add value to a slice, making the data structure dynamic, yet still allows us to use that contiguous block of memory that gives us the predictable access pattern from mechanical sympathy. The append call is working with value semantic. We are not sharing this slice but appending to it and returning a new copy of it. The slice gets to stay on the stack, not heap.

```go
    data = append(data, fmt.Sprintf("Rec: %d", record))
```

Every time append runs, it checks the length and capacity. If it is the same, it means that we have no room. append creates a new backing array, double its size, copy the old value back in and append the new value. It mutates its copy on its stack frame and returns us a copy. We replace our slice with the new copy. If it is not the same, it means that we have extra elements of capacity we can use. Now we can bring these extra capacities into the length and no copy is being made. This is very efficient. Looking at the last column in the output, when the backing array is 1000 elements or less, it doubles the size of the backing array for growth. Once we pass 1000 elements, the growth rate moves to 25%. When the capacity of the slice changes, display the changes.

```
       if lastCap != cap(data) {
```

Calculate the percent of change.

```
             capChg := float64(cap(data)-lastCap) / float64(lastCap) *
100
```

Save the new values for capacity.

```
             lastCap = cap(data)
```

Display the results.

```
             fmt.Printf("Addr[%p]\tIndex[%d]\t\tCap[%d - %2.f%%]\n",
&data[0], record, cap(data), capChg)
```

```
=> Idea of appending
Addr[0xc0000102a0]      Index[1]         Cap[1 - +Inf%]
Addr[0xc00000c0c0]      Index[2]         Cap[2 - 100%]
Addr[0xc000016080]      Index[3]         Cap[4 - 100%]
Addr[0xc00007e080]      Index[5]         Cap[8 - 100%]
Addr[0xc000100000]      Index[9]         Cap[16 - 100%]
Addr[0xc000102000]      Index[17]        Cap[32 - 100%]
Addr[0xc00007a400]      Index[33]        Cap[64 - 100%]
Addr[0xc000104000]      Index[65]        Cap[128 - 100%]
Addr[0xc000073000]      Index[129]       Cap[256 - 100%]
Addr[0xc000106000]      Index[257]       Cap[512 - 100%]
Addr[0xc00010a000]      Index[513]       Cap[1024 - 100%]
Addr[0xc000110000]      Index[1025]      Cap[1280 - 25%]
Addr[0xc00011a000]      Index[1281]      Cap[1704 - 33%]
Addr[0xc000132000]      Index[1705]      Cap[2560 - 50%]
Addr[0xc000140000]      Index[2561]      Cap[3584 - 40%]
Addr[0xc000154000]      Index[3585]      Cap[4608 - 29%]
Addr[0xc000180000]      Index[4609]      Cap[6144 - 33%]
Addr[0xc000198000]      Index[6145]      Cap[7680 - 25%]
Addr[0xc0001b6000]      Index[7681]      Cap[9728 - 27%]
Addr[0xc000200000]      Index[9729]      Cap[12288 - 26%]
Addr[0xc000230000]      Index[12289]     Cap[15360 - 25%]
Addr[0xc000280000]      Index[15361]     Cap[19456 - 27%]
Addr[0xc000300000]      Index[19457]     Cap[24576 - 26%]
Addr[0xc000360000]      Index[24577]     Cap[30720 - 25%]
```

```
Addr[0xc000400000]      Index[30721]      Cap[38400 - 25%]
Addr[0xc000300000]      Index[38401]      Cap[48128 - 25%]
Addr[0xc000600000]      Index[48129]      Cap[60416 - 26%]
Addr[0xc0006ec000]      Index[60417]      Cap[75776 - 25%]
Addr[0xc000814000]      Index[75777]      Cap[94720 - 25%]
Addr[0xc000600000]      Index[94721]      Cap[118784 - 25%]
```

Slice of slice

Take a slice of slice2. We want just indexes 2 and 3. The length is slice3 is 2 and capacity is 6.

Parameters are [starting_index : (starting_index + length)]

By looking at the output, we can see that they are sharing the same backing array. These slice headers get to stay on the stack when we use these value semantics. Only the backing array that needed to be on the heap.

```
slice3 := slice2[2:4]

fmt.Printf("\n=> Slice of slice (before)\n")
inspectSlice(slice2)
inspectSlice(slice3)
```

When we change the value of the index 0 of slice3, who are going to see this change?

```
=> Slice of slice (before)
Length[5] Capacity[8]
[0] 0xc00007e000 Apple
[1] 0xc00007e010 Orange
[2] 0xc00007e020 Banana
[3] 0xc00007e030 Grape
[4] 0xc00007e040 Plum
Length[2] Capacity[6]
[0] 0xc00007e020 Banana
[1] 0xc00007e030 Grape
```

```
slice3[0] = "CHANGED"
```

The answer is both. We have to always be aware that we are modifying an existing slice. We have to be aware who is using it, who is sharing that backing array.

```
fmt.Printf("\n=> Slice of slice (after)\n")
inspectSlice(slice2)
inspectSlice(slice3)
```

```
=> Slice of slice (after)
Length[5] Capacity[8]
[0] 0xc00007e000 Apple
[1] 0xc00007e010 Orange
[2] 0xc00007e020 CHANGED
[3] 0xc00007e030 Grape
[4] 0xc00007e040 Plum
Length[2] Capacity[6]
[0] 0xc00007e020 CHANGED
[1] 0xc00007e030 Grape
```

How about slice3 := append(slice3, "CHANGED")? Similar problem will occur with append if the length and capacity is not the same. Instead of changing slice3 at index 0, we call append on slice3. Since the length of slice3 is 2, capacity is 6 at the moment, we have extra rooms for modification. We go and change the element at index 3 of slice3, which is index 4 of slice2. That is very dangerous. So, what if the length and capacity is the same? Instead of making slice3 capacity 6, we set it to 2 by adding another parameter to the slicing syntax like this: slice3 := slice2[2:4:4]

When append looks at this slice and sees that the length and capacity is the same, it wouldn't bring in the element at index 4 of slice2. It would detach. slice3 will have a length of 2 and capacity of 2, still share the same backing array. On the call to append, length and capacity will be different. The addresses are also different. This is called a 3 index slice. This new slice will get its own backing array and we don't affect anything at all to our original slice.

Copy a slice

copy only works with string and slice only. Make a new slice big enough to hold elements of the original slice and copy the values over using the built-in copy function.

```
slice4 := make([]string, len(slice2))
copy(slice4, slice2)

fmt.Printf("\n=> Copy a slice\n")
```

```
inspectSlice(slice4)
```

```
=> Copy a slice
Length[5] Capacity[5]
[0] 0xc00005c050 Apple
[1] 0xc00005c060 Orange
[2] 0xc00005c070 CHANGED
[3] 0xc00005c080 Grape
[4] 0xc00005c090 Plum
```

Slice and reference

Declare a slice of integers with 7 values.

```
x := make([]int, 7)
```

Random starting counters.

```
for i := 0; i < 7; i++ {
      x[i] = i * 100
}
```

Set a pointer to the second element of the slice.

```
twohundred := &x[1]
```

Append a new value to the slice. This line of code raises a red flag. We have x is a slice with length 7, capacity 7. Since the length and capacity is the same, append doubles its size then copy values over. x nows points to different memory block and has a length of 8, capacity of 14.

```
x = append(x, 800)
```

When we change the value of the second element of the slice, twohundred is not gonna change because it points to the old slice. Everytime we read it, we will get the wrong value.

```
x[1]++
```

By printing out the output, we can see that we are in trouble.

```
fmt.Printf("\n=> Slice and reference\n")
fmt.Println("twohundred:", *twohundred, "x[1]:", x[1])
```

```
=> Slice and reference
twohundred: 100 x[1]: 101
```

UTF-8

Everything in Go is based on UTF-8 character sets. If we use different encoding scheme, we might have a problem.

Declare a string with both Chinese and English characters. For each Chinese character, we need 3 byte for each one. The UTF-8 is built on 3 layers: bytes, code point and character. From Go perspective, string are just bytes. That is what we are storing.

In our example, the first 3 bytes represents a single code point that represents that single character. We can have anywhere from 1 to 4 bytes representing a code point (a code point is a 32 bit value) and anywhere from 1 to multiple code points can actually represent a character. To keep it simple, we only have 3 bytes representing 1 code point representing 1 character. So we can read s as 3 bytes, 3 bytes, 1 byte, 1 byte,... (since there are only 2 Chinese characters in the first place, the rest are English)

```
s := "世界 means world"
```

UTFMax is 4 -- up to 4 bytes per encoded rune -> maximum number of bytes we need to represent any code point is 4. mRune is its own type. It is an alias for int32 type. Similar to type byte we are using, it is just an alias for uint8.

```
var buf [utf8.UTFMax]byte
```

When we are ranging over a string, are we doing it byte by byte or code point by code point or character by character? The answer is code point by code point. On the first iteration, i is 0. On the next one, i is 3 because we are moving to the next code point. Then i is 6.

```
for i, r := range s {
```

Capture the number of bytes for this rune/code point.

```
      rl := utf8.RuneLen(r)
```

Calculate the slice offset for the bytes associated with this rune.

```
      si := i + rl
```

Copy rune from the string to our buffer. We want to go through every code point and copy them into our array buf, and display them on the screen.
"Every array is just a slice waiting to happen." - Go saying
We are using the slicing syntax, creating our slice header where buf becomes the backing array. All of them are on the stack. There is no allocation here.

```
      copy(buf[:], s[i:si])
```

Display the details.

```
      fmt.Printf("%2d: %q; codepoint: %#6x; encoded bytes: %#v\n", i, r,
r, buf[:rl])
```

```
 0: '世'; codepoint: 0x4e16; encoded bytes: []byte{0xe4, 0xb8, 0x96}
 3: '界'; codepoint: 0x754c; encoded bytes: []byte{0xe7, 0x95, 0x8c}
 6: ' '; codepoint:   0x20; encoded bytes: []byte{0x20}
 7: 'm'; codepoint:   0x6d; encoded bytes: []byte{0x6d}
 8: 'e'; codepoint:   0x65; encoded bytes: []byte{0x65}
 9: 'a'; codepoint:   0x61; encoded bytes: []byte{0x61}
10: 'n'; codepoint:   0x6e; encoded bytes: []byte{0x6e}
11: 's'; codepoint:   0x73; encoded bytes: []byte{0x73}
12: ' '; codepoint:   0x20; encoded bytes: []byte{0x20}
13: 'w'; codepoint:   0x77; encoded bytes: []byte{0x77}
14: 'o'; codepoint:   0x6f; encoded bytes: []byte{0x6f}
15: 'r'; codepoint:   0x72; encoded bytes: []byte{0x72}
16: 'l'; codepoint:   0x6c; encoded bytes: []byte{0x6c}
17: 'd'; codepoint:   0x64; encoded bytes: []byte{0x64}
```

Map

user defines a user in the program.

```go
type user struct {
    name     string
    username string
}
```

Declare and initialize

Declare and make a map that stores values of type user with a key of type string.

```go
func main() {
	users1 := make(map[string]user)

	// Add key/value pairs to the map
	users1["Roy"] = user{"Rob", "Roy"}
	users1["Ford"] = user{"Henry", "Ford"}
	users1["Mouse"] = user{"Mickey", "Mouse"}
	users1["Jackson"] = user{"Michael", "Jackson"}

	// Iterate over map
	fmt.Printf("\n=> Iterate over map\n")
	for key, value := range users1 {
		fmt.Println(key, value)
}
```

```
=> Iterate over map
Roy {Rob Roy}
Ford {Henry Ford}
Mouse {Mickey Mouse}
Jackson {Michael Jackson}
```

Map literals

Declare and initialize the map with values.

```go
users2 := map[string]user{
	"Roy":     {"Rob", "Roy"},
	"Ford":    {"Henry", "Ford"},
	"Mouse":   {"Mickey", "Mouse"},
	"Jackson": {"Michael", "Jackson"},
}
```

```go
// Iterate over the map.
fmt.Printf("\n=> Map literals\n")
for key, value := range users2 {
    fmt.Println(key, value)
}
```

```
=> Map literals
Roy {Rob Roy}
Ford {Henry Ford}
Mouse {Mickey Mouse}
Jackson {Michael Jackson}
```

Delete key

```go
delete(users2, "Roy")
```

Find key

Find the Roy key. If found is True, we will get a copy value of that type. if found is False, u is still a value of type user but is set to its zero value.

```go
u1, found1 := users2["Roy"]
u2, found2 := users2["Ford"]
```

Display the value and found flag.

```go
fmt.Printf("\n=> Find key\n")
fmt.Println("Roy", found1, u1)
fmt.Println("Ford", found2, u2)
```

```
=> Find key
Roy false { }
Ford true {Henry Ford}
```

Map key restrictions

```go
type users []user
```

Using this syntax, we can define a set of users This is a second way we can define users. We can use an existing type and use it as a base for another type. These are two different types. There is no relationship here. However, when we try use it as a key, like: u := make(map[users]int) the compiler says we cannot use that: "invalid map key type users"

The reason is: whatever we use for the key, the value must be comparable. We have to use it in some sort of boolean expression in order for the map to create a hash value for it.

Decoupling

Method

Value and Pointer Receiver Call

```go
type user struct {
	name  string
	email string
}
```

notify implements a method with a value receiver: u of type user In Go, a function is called a method if that function has a receiver declared within itself. It looks and feels like a parameter but it is exactly what it is. Using the value receiver, the method operates on its own copy of the value that is used to make the call.

```go
func (u user) notify() {
	fmt.Printf("Sending User Email To %s<%s>\n", u.name, u.email)
}
```

changeEmail implements a method with a pointer receiver: u of type pointer user, Using the pointer receiver, the method operates on shared access.

```go
func (u *user) changeEmail(email string) {
	u.email = email
	fmt.Printf("Changed User Email To %s\n", email)
}
```

These 2 methods above are just for studying the difference between a value receiver and a pointer receiver. In production, we will have to ask yourself why we choose to use inconsistent receiver's type. We will talk about this later on.

Value and pointer receiver call

Values of type user can be used to call methods declared with both value and pointer receivers.

```
bill := user{"Bill", "bill@email.com"}
bill.notify()
bill.changeEmail("bill@hotmail.com")
```

```
Sending User Email To Bill<bill@email.com>
Changed User Email To bill@hotmail.com
```

Pointers of type user can also be used to call methods declared with both value and pointer receiver.

```
hoanh := &user{"Hoanh", "hoanhan@email.com"}
hoanh.notify()
hoanh.changeEmail("hoanhan101@gmail.com")
```

```
Sending User Email To Hoanh<hoanhan@email.com>
Changed User Email To hoanhan101@gmail.com
```

hoanh in this example is a pointer that has the type *user. We are still able to call notify. This is still correct. As long as we deal with the type user, Go can adjust to make the call. Behind the scene, we have something like (*hoanh).notify(). Go will take the value that hoanh points to and make sure that notify leverages its value semantic and works on its own copy. Similarly, bill has the type user but still be able to call changeEmail. Go will take the address of bill and do the rest for you: (&bill).changeEmail().

Create a slice of user values with two users.

```
users := []user{
      {"bill", "bill@email.com"},
      {"hoanh", "hoanh@email.com"},
}
```

We are ranging over this slice of values, making a copy of each value and call notify to make another copy.

```
for _, u := range users {
      u.notify()
}
```

```
Sending User Email To bill<bill@email.com>
Sending User Email To Hoanh<hoanhan@email.com>
```

Iterate over the slice of users switching semantics. This is not a good practice.

```
for _, u := range users {
      u.changeEmail("it@wontmatter.com")
}
```

```
Changed User Email To it@wontmatter.com
Changed User Email To it@wontmatter.com
```

Value and Pointer Semantics

When it comes to using built-in type (numeric, string, bool), we should always be using value semantics. When a piece of code that takes an address of an integer or a bool, this raises a big flag. It's hard to say because it depends on the context. But in general, why should these values end up on the heap creating garbage? These should be on the stack. There is an exception to everything. However, until we know it is okay to take the exception, we should follow the guideline.

The reference type (slice, map, channel, interface) also focuses on using value semantics. The only time we want to take the address of a slice is when we are sharing it down the call stack to the Unmarshal function since it always requires the address of a value.

Examples below are from the standard library. By studying them, we learn how important it is to use value or pointer semantics in a consistent way.
When we declare a type, we must ask ourselves immediately:

- Does this type require value semantic or pointer semantic?
- If I need to modify this value, should we create a new value or should we modify the value itself so everyone can see it?

It needs to be consistent. It is okay to guess it wrong the first time and refactor it later.

```
package main

import (
	"sync/atomic"
	"syscall"
)
```

Value semantic

These are named types from the net package called IP and IPMask with base types that are slices of bytes. Since we use value semantics for reference types, the implementation is using value semantics for both.

```
type IP []byte
type IPMask []byte
```

Mask is using a value receiver and returning a value of type IP. This method is using value semantics for type IP.

```
func (ip IP) Mask(mask IPMask) IP {
	if len(mask) == IPv6len && len(ip) == IPv4len && allFF(mask[:12])
{
		mask = mask[12:]
	}
	if len(mask) == IPv4len && len(ip) == IPv6len &&
bytesEqual(ip[:12], v4InV6Prefix) {
		ip = ip[12:]
	}
	n := len(ip)
	if n != len(mask) {
		return nil
	}
	out := make(IP, n)
	for i := 0; i < n; i++ {
		out[i] = ip[i] & mask[i]
	}
	return out
}
```

ipEmptyString accepts a value of type IP and returns a value of type string. The function is using value semantics for type IP.

```
func ipEmptyString(ip IP) string {
	if len(ip) == 0 {
		return ""
	}
	return ip.String()
}
```

Pointer semantic

Should Time use value or pointer semantics? If you need to modify a Time value, should you mutate the value or create a new one?

```
type Time struct {
	sec  int64
	nsec int32
	loc  *Location
}
```

The best way to understand what semantic is going to be used is to look at the factory function for type. It dictates the semantics that will be used. In this example, the Now function returns a value of type Time. It is making a copy of its Time value and passing it back up. This means Time value can be on the stack. We should be using value semantic all the way through.

```
func Now() Time {
	sec, nsec := now()
	return Time{sec + unixToInternal, nsec, Local}
}
```

Add is a mutation operation. If we go with the idea that we should be using pointer semantic when we mutate something and value semantic when we don't then Add is implemented wrong. However, it has not been wrong because it is the type that has to drive the semantic, not the implementation of the method. The method must adhere to the semantic that we choose.

Add is using a value receiver and returning a value of type Time. It is mutating its local copy and returning to us something new.

```
func (t Time) Add(d Duration) Time {
	t.sec += int64(d / 1e9)
	nsec := int32(t.nsec) + int32(d%1e9)
	if nsec >= 1e9 {
		t.sec++
            nsec -= 1e9
      } else if nsec < 0 {
            t.sec--
            nsec += 1e9
      }
      t.nsec = nsec
      return t
}
```

div accepts a value of type Time and returns values of built-in types. The function is using value semantics for type Time.

```
func div(t Time, d Duration) (qmod2 int, r Duration) {}
```

The only use of pointer semantics for the Time API are these Unmarshal-related functions:

```
func (t *Time) UnmarshalBinary(data []byte) error {}
func (t *Time) GobDecode(data []byte) error {}
func (t *Time) UnmarshalJSON(data []byte) error {}
func (t *Time) UnmarshalText(data []byte) error {}
```

Most struct types are not going to be able to leverage value semantic. Most struct types are probably gonna be data that should be shared or more efficient to be shared. For example, an User type. Regardless, it is possible to copy an User type but it is not a proper thing to do in the real world.

Other examples:

Factory functions dictate the semantics that will be used. The Open function returns a pointer of type File. This means we should be using pointer semantics and share File values.

```
func Open(name string) (file *File, err error) {
      return OpenFile(name, O_RDONLY, 0)
}
```

Chdir is using a pointer receiver. This method is using pointer semantics for File.

```
func (f *File) Chdir() error {
      if f == nil {
```

```go
			return ErrInvalid
		}
		if e := syscall.Fchdir(f.fd); e != nil {
			return &PathError{"chdir", f.name, e}
		}
		return nil
}
```

epipecheck accepts a pointer of type File. The function is using pointer semantics for type File.

```go
func epipecheck(file *File, e error) {
		if e == syscall.EPIPE {
			if atomic.AddInt32(&file.nepipe, 1) >= 10 {
				sigpipe()
			}
		} else {
			atomic.StoreInt32(&file.nepipe, 0)
		}
}
```

Methods are just functions

Methods are really just made up. They are not real. They are just syntactic sugar. They give us a belief system that some pieces of data expose some capabilities. Object-oriented programming has driven design and capabilities. However, there is no OOP in Go. There is data and behavior.

At some time, data can expose some capabilities, but for specific purposes, not to really design API around. Methods are really just functions.

```go
type data struct {
		name string
		age  int
}
```

displayName provides a pretty print view of the name. It uses data as a value receiver.

```go
func (d data) displayName() {
		fmt.Println("My Name Is", d.name)
}
```

setAge sets the age and displays the value. It uses data as a pointer receiver.

```
func (d *data) setAge(age int) {
      d.age = age
      fmt.Println(d.name, "Is Age", d.age)
}
```

Methods are just functions

Declare a variable of type data.

```
d := data{
      name: "Hoanh",
}
fmt.Println("Proper Calls to Methods:")
```

How we actually call methods in Go.

```
d.displayName()
d.setAge(21)

fmt.Println("\nWhat the Compiler is Doing:")
```

This is what Go is doing underneath. When we call d.displayName(), the compiler will call data.displayName, showing that we are using a value receiver of type data, and pass the data in as the first parameter. Taking a look at the function again: "func (d data) displayName()", that receiver is the parameter because it is truly a parameter. It is the first parameter to a function that calls displayName.

Similar to d.setAge(45). Go is calling a function that is based on the pointer receiver and passing data to its parameters. We are adjusting to make the call by taking the address of d.

```
data.displayName(d)
(*data).setAge(&d, 21)
```

```
Proper Calls to Methods:
My Name Is Hoanh
Hoanh Is Age 21

What the Compiler is Doing:
```

```
My Name Is Hoanh
Hoanh Is Age 21
```

Function variable

```
fmt.Println("\nCall Value Receiver Methods with Variable:")
```

Declare a function variable for the method bound to the d variable. The function variable will get its own copy of d because the method is using a value receiver. f1 is now a reference type: a pointer variable. We don't call the method here. There is no () at the end of displayName.

```
f1 := d.displayName
```

Call the method via the variable.

f1 is pointer and it points to a special 2 word data structure. The first word points to the code for that method we want to execute, which is displayName in this case. We cannot call displayName unless we have a value of type data. So the second word is a pointer to the copy of data. displayName uses a value receiver so it works on its own copy. When we make an assignment to f1, we are having a copy of d.

When we change the value of d to "Hoanh An", f1 is not going to see the change.

```
d.name = "Hoanh An"
```

Call the method via the variable. We don't see the change.

```
f1()
```

```
Call Value Receiver Methods with Variable:
My Name Is Hoanh
My Name Is Hoanh
```

However, if we do this again if f2, then we will see the change.

```
fmt.Println("\nCall Pointer Receiver Method with Variable:")
```

Declare a function variable for the method bound to the d variable. The function variable will get the address of d because the method is using a pointer receiver.

```
f2 := d.setAge
d.name = "Hoanh An Dinh"
```

Call the method via the variable. f2 is also a pointer that has 2 word data structure. The first word points to setAge, but the second word doesn't point to its copy any more, but to its original.

```
f2(21)
```

```
Call Pointer Receiver Method with Variable:
Hoanh An Dinh Is Age 21
```

Interface

Valueless type

reader is an interface that defines the act of reading data. interface is technically a valueless type. This interface doesn't declare state. It defines a contract of behavior. Through that contract of behavior, we have polymorphism. It is a 2 word data structure that has 2 pointers. When we say var r reader, we would have a nil value interface because interface is a reference type.

```
type reader interface {
      read(b []byte) (int, error) // (1)
}
```

We could have written this API a little bit differently. Technically, I could have said: How many bytes do you want me to read and I will return that slice of byte and an error, like so: read(i int) ([]byte, error) (2).

Why do we choose the other one instead?

Every time we call (2), it will cost an allocation because the method would have to allocate a slice of some unknown type and share it back up the call stack. The method would have to allocate a slice of some unknown type and share it back up the call stack. The backing array for that slice has to be an allocation. But if we stick with (1), the caller is allocating a slice. Even the backing array for that is ended up on a heap, it is just 1 allocation. We can call this 10000 times and it is still 1 allocation.

Concrete type vs Interface type

A concrete type is any type that can have a method. Only user defined types can have a method.

Method allows a piece of data to expose capabilities, primarily around interfaces. file defines a system file.

It is a concrete type because it has the method read below. It is identical to the method in the reader interface. Because of this, we can say the concrete type file implements the reader interface using a value receiver.

There is no fancy syntax. The compiler can automatically recognize the implementation here.

Relationship

We store concrete type values inside interfaces.

```
type file struct {
	name string
}
```

read implements the reader interface for a file.

```
func (file) read(b []byte) (int, error) {
	s := "<rss><channel><title>Going Go
Programming</title></channel></rss>"
	copy(b, s)
	return len(s), nil
}
```

pipe defines a named pipe network connection. This is the second concrete type that uses a value receiver. We now have two different pieces of data, both exposing the reader's contract and implementation for this contract.

```
type pipe struct {
	name string
}
```

read implements the reader interface for a network connection.

```
func (pipe) read(b []byte) (int, error) {
	s := `{name: "hoanh", title: "developer"}`
	copy(b, s)
	return len(s), nil
}
```

Create two values one of type file and one of type pipe.

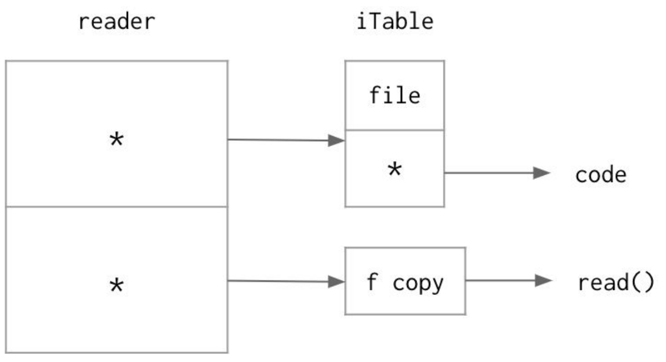
```
f := file{"data.json"}
p := pipe{"cfg_service"}
```

Call the retrieve function for each concrete type. Here we are passing the value itself, which means a copy of f going to pass across the program boundary.

The compiler will ask: Does this file value implement the reader interface? The answer is Yes, because there is a method there using the value receiver that implements its contract. The second word of the interface value will store its own copy of f. The first word points to a special data structure that we call the iTable.

The iTable serves 2 purposes:

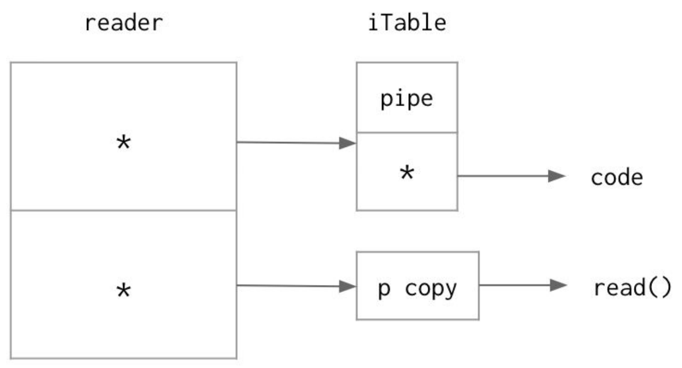
- The first part describes the type of value being stored. In our case, it is the file value.
- The second part gives us a matrix of function pointers so we can actually execute the right method when we call that through the interface.



When we do a read against the interface, we can do an iTable lookup, find that read associated with this type, then call that value against the read method - the concrete implementation of read for this type of value.

```
retrieve(f)
```

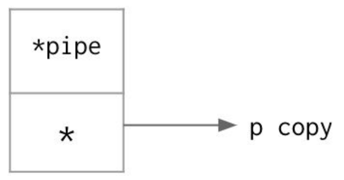
Similar with p. Now the first word of reader interface points to pipe, not file and the second word points to a copy of pipe value.



The behavior changes because the data changes.

```
retrieve(p)
```

Later on, for simplicity, instead of drawing the a pointer pointing to iTable, we only draw *pipe, like so:



Polymorphic function

retrieve can read any device and process the data. This is called a polymorphic function. The parameter is being used here is the reader type. But it is valueless. What does it mean? This function will accept any data that implement the reader contract. This function knows nothing about the concrete and it is completely decoupled. It is the highest level of decoupling we can get. The algorithm is still efficient and compact. All we have is a level of indirection to the concrete type data values in order to be able to execute the algorithm.

```
func retrieve(r reader) error {
      data := make([]byte, 100)
```

```
	len, err := r.read(data)
	if err != nil {
		return err
	}

	fmt.Println(string(data[:len]))
	return nil
}
```

```
<rss><channel><title>Going Go Programming</title></channel></rss>
{name: "hoanh", title: "developer"}
```

Interface via pointer receiver

notifier is an interface that defines notification type behavior.

```
type notifier interface {
	notify()
}
```

printer displays information.

```
type printer interface {
	print()
}
```

user defines a user in the program.

```
type user struct {
	name  string
	email string
}
```

print displays user's name and email.

```
func (u user) print() {
	fmt.Printf("My name is %s and my email is %s\n", u.name, u.email)
}
```

notify implements the notifier interface with a pointer receiver.

```go
func (u *user) notify() {
      fmt.Printf("Sending User Email To %s<%s>\n", u.name, u.email)
}
```

String implements the fmt.Stringer interface. The fmt package that we've been using to display things on the screen, if it receives a piece of data that implements this behavior, it will use this behavior and overwrite its default. Since we are using pointer semantics, only pointer satisfies the interface.

```go
func (u *user) String() string {
      return fmt.Sprintf("My name is %q and my email is %q", u.name,
u.email)
}
```

Create a value of type User

```go
u := user{"Hoanh", "hoanhan@email.com"}
```

Call a polymorphic function but pass u using value semantic: sendNotification(u). However, the compiler doesn't allow it: "cannot use u (type user) as type notifier in argument to sendNotification:
user does not implement notifier (notify method has pointer receiver)" This is setting up for an integrity issue.

Method set

In the specification, there are sets of rules around the concepts of method sets. What we are doing is against these rules.

What are the rules?

- For any value of a given type T, only those methods implemented with a value receiver belong to the method sets of that type.
- For any value of a given type *T (pointer of a given type), both value receiver and pointer receiver methods belong to the method sets of that type.

In other words, if we are working with a pointer of some type, all the methods that have been declared are associated with that pointer. But if we are working with a value of some types, only those methods that operated on value semantic can be applied.

In the previous lesson about method, we are calling them before without any problem. That is true. When we are dealing with method, method call against the concrete values themselves, Go can adjust to make the call.
However, we are not trying to call a method here. We are trying to store a concrete type value inside the interface. For that to happen, that value must satisfy the contract.

The question now becomes: Why can't pointer receivers be associated with the method sets for value? What is the integrity issue here that doesn't allow us to use pointer semantics for value of type T?

It is not 100% guaranteed that any value that can satisfy the interface has an address. We can never call a pointer receiver because if that value doesn't have an address, it is not shareable. For example:

Declare a type named duration that is based on an integer

```
type duration int.
```

Declare a method name notify using a pointer receiver. This type now implements the notifier interface using a pointer receiver.

```
func (d *duration) notify() {
        fmt.Println("Sending Notification in", *d)
}
```

Take a value 42, convert it to type duration and try to call the notify method. Here are what the compiler says:

- "cannot call pointer method on duration(42)"
- "cannot take the address of duration(42)"

```
duration(42).notify()
```

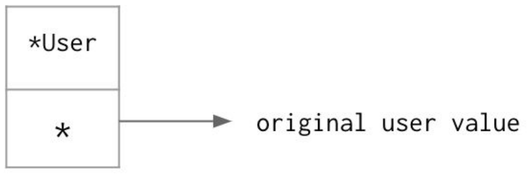
Why can't we get the address? Because 42 is not stored in a variable. It is still literal value that we don't know ahead the type. Yet it still does implement the notifier interface.

Come back to our example, when we get the error, we know that we are mixing semantics. u implements the interface using a pointer receiver and now we are trying to work with a copy of that value, instead of trying to share it. It is not consistent.

The lesson

If we implement an interface using a pointer receiver, we must use pointer semantics. If we implement an interface using value receiver, we then have the ability to use value semantic and pointer semantic. However, for consistency, we want to use value semantics most of the time, unless we are doing something like an Unmarshal function.

To fix the issue, instead of passing value u, we must pass the address of u (&u). We create a user value and pass the address of that, which means the interface now has a pointer of type user and we get to point to the original value.



```
sendNotification(&u)
```

This is our polymorphic function. sendNotification accepts values that implement the notifier interface and sends notifications. This is again saying: I will accept any value or pointer that implements the notifier interface. I will call that behavior against the interface itself.

```
func sendNotification(n notifier) {
      n.notify()
}
```

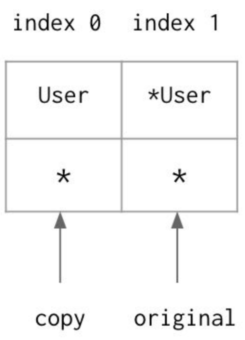
Similarly, when we pass a value of u to Println, in the output we only see the default formatting. When we pass the address through, it now can overwrite it.

```
fmt.Println(u)
fmt.Println(&u)
```

```
Sending User Email To Hoanh<hoanhan@email.com>
{Hoanh hoanhan@email.com}
My name is "Hoanh" and my email is "hoanhan@email.com"
```

Slice of interface

Create a slice of interface values. It means that I can store in this dataset any value or pointer that implements the printer interface.



```
entities := []printer{
```

When we store a value, the interface value has its own copy of the value. Changes to the original value will not be seen.

```
            u,
```

When we store a pointer, the interface value has its own copy of the address. Changes to the original value will be seen.

```
            &u,
}
```

Change the name and email on the user value.

```
u.name = "Hoanh An"
u.email = "hoanhan101@gmail.com"
```

Iterate over the slice of entities and call print against the copied interface value.

```
for _, e := range entities {
      e.print()
}
```

```
My name is Hoanh and my email is hoanhan@email.com
My name is Hoanh An and my email is hoanhan101@gmail.com
```

Embedding

Declaring fields, NOT Embedding

user defines a user in the program.

```go
type user struct {
	name  string
	email string
}
```

notify implements a method notifies users of different events.

```go
func (u *user) notify() {
	fmt.Printf("Sending user email To %s<%s>\n", u.name, u.email)
}
```

admin represents an admin user with privileges. person user is not embedding. All we do here is just create a person field based on that other concrete type named user.

```go
type admin struct {
	person user // NOT Embedding
	level  string
}
```

Create an admin user using struct literal. Since a person also has struct type, we use another literal to initialize it.

```go
func main() {
	ad := admin{
		person: user{
			name:  "Hoanh An",
			email: "hoanhan101@gmail.com",
	},
		level: "superuser",
}
```

We call notify through the person field through the admin type value.

```go
	ad.person.notify()
```

```
Sending user email To Hoanh An<hoanhan101@gmail.com>
```

Embedding types

user defines a user in the program.

```go
type user struct {
	name  string
	email string
}
```

notify implements a method notifies users of different events.

```go
func (u *user) notify() {
	fmt.Printf("Sending user email To %s<%s>\n", u.name, u.email)
}
```

admin represents an admin user with privileges. Notice that we don't use the field person here anymore. We are now embedding a value of type user inside the value of type admin. This is an inner-type-outer-type relationship where user is the inner type and admin is the outer type.

Inner type promotion

What special about embedding in Go is that we have an inner type promotion mechanism. In other words, anything related to the inner type can be promoted up to the outer type. It will mean more in the construction below.

```go
type admin struct {
	user  // Embedded Type
	level string
}
```

We are now constructing an outer type admin and inner type user. This inner type value now looks like a field, but it is not a field. We can access it through the type name like a field. We are initializing the inner value through the struct literal of the user.

```go
func main() {
	ad := admin{
```

```
            user: user{
                  name:  "Hoanh An",
                  email: "hoanhan101@gmail.com",
            },
      level: "superuser",

      // We can access the inner type's method directly.
      ad.user.notify()
      ad.notify()
}
```

Because of inner type promotion, we can access the notify method directly through the outer type. Therefore, the output will be the same.

```
Sending user email To Hoanh An<hoanhan101@gmail.com>
Sending user email To Hoanh An<hoanhan101@gmail.com>
```

Embedded type and Interface

notifier is an interface that defines notification type behavior.

```
type notifier interface {
      notify()
}
```

user defines a user in the program.

```
type user struct {
      name  string
      email string
}
```

notify implements a method notifies users of different events using a pointer receiver.

```
func (u *user) notify() {
      fmt.Printf("Sending user email To %s<%s>\n", u.name, u.email)
}
```

admin represents an admin user with privileges.

```go
type admin struct {
	user
	level string
}

func main() {
	// Create an admin user.
	ad := admin{
		user: user{
			name:  "Hoanh An",
			email: "hoanhan101@gmail.com",
		},
		level: "superuser",
}
```

Send the admin user a notification.

We are passing the address of outer type value. Because of inner type promotion, the outer type now implements all the same contract as the inner type.

```go
	sendNotification(&ad)
```

Embedding does not create a sub typing relationship. user is still user and admin is still admin. The behavior that inner type value uses, the outer type exposes it as well. It means that outer type value can implement the same interface/same contract as the inner type.

We are getting type reuse. We are not mixing or sharing state but extending the behavior up to the outer type.

We have our polymorphic function here. sendNotification accepts values that implement the notifier interface and sends notifications.

```go
func sendNotification(n notifier) {
	n.notify()
}
```

```
Sending user email To Hoanh An<hoanhan101@gmail.com>
```

Outer and inner type implementing the same interface

notifier is an interface that defines notification type behavior.

```go
type notifier interface {
	notify()
}
```

user defines a user in the program.

```go
type user struct {
	name  string
	email string
}
```

notify implements a method notifies users of different events.

```go
func (u *user) notify() {
	fmt.Printf("Sending user email To %s<%s>\n", u.name, u.email)
}
```

admin represents an admin user with privileges.

```go
type admin struct {
	user
	level string
}
```

notify implements a method notifies admins of different events. We now have two different implementations of notifier interface, one for the inner type, one for the outer type. Because the outer type now implements that interface, the inner type promotion doesn't happen. We have overwritten through the outer type anything that the inner type provides to us.

```go
func (a *admin) notify() {
	fmt.Printf("Sending admin email To %s<%s>\n", a.name, a.email)
}
```

Create an admin user.

```go
func main() {
	ad := admin{
		user: user{
			name:  "Hoanh An",
			email: "hoanhan101@gmail.com",
            },
            level: "superuser",
}
```

Send the admin user a notification. The embedded inner type's implementation of the interface is NOT "promoted" to the outer type.

```
      sendNotification(&ad)
```

We can access the inner type's method directly.

```
      ad.user.notify()
```

The inner type's method is NOT promoted.

```
      ad.notify()
```

```
Sending admin email To Hoanh An<hoanhan101@gmail.com>
Sending user email To Hoanh An<hoanhan101@gmail.com>
Sending admin email To Hoanh An<hoanhan101@gmail.com>
```

Exporting

Guideline

Package is a self-contained unit of code. Anything that is named in a given package can be exported or accessible through other packages or unexported or not accessible through other packages.

Exported identifier

Package counters provides alert counter support.

```
package counters
```

alertCounter is an unexported named type that contains an integer counter for alerts. The first character is in upper-case format so it is considered to be exported.

```
type AlertCounter int
```

alertCounter is an unexported named type that contains an integer counter for alerts. The first character is in lower-case format so it is considered to be unexported. It is not accessible for other packages, unless they are part of the package counters themselves.

```
type alertCounter int
```

In the main package, try to import the counters package.

```
package main

import (
	"fmt"

	"github.com/hoanhan101/counters"
)
```

Create a variable of the exported type and initialize the value to 10.

```
counter := counters.AlertCounter(10)
```

However, when we create a variable of the unexported type and initialize the value to 10.

```
counter := counters.alertCounter(10)
```

The compiler will say:
- cannot refer to unexported name counters.alertCounter
- undefined: counters.alertCounter

Accessing a value of an unexported identifier

Package counters provides alert counter support.

```
package counters
```

alertCounter is an unexported named type that contains an integer counter for alerts.

```
type alertCounter int
```

Declare an exported function called New - a factory function that knows how to create and initialize the value of an unexported type. It returns an unexported value of alertCounter.

```
func New(value int) alertCounter {
	return alertCounter(value)
}
```

The compiler is okay with this because exporting and unexporting is not about the value like private and public mechanism, it is about the identifier itself. However, we don't do this since there is no encapsulation here. We can just make the type exported.

Access a value of an unexported identifier.

```
package main

import (
	"fmt"

	"github.com/hoanhan101/counters"
)
```

Create a variable of the unexported type using the exported New function from the package counters.

```
func main() {
	counter := counters.New(10)

	fmt.Printf("Counter: %d\n", counter)
}
```

```
Counter: 10
```

Unexported fields from an exported struct

Package users provides support for user management.

```
package users
```

Exported type User represents information about a user. It has 2 exported fields: Name and ID and 1 unexported field: password.

```go
type User struct {
	Name string
	ID   int

	password string
}
```

```go
package main

import (
	"fmt"

	"github.com/hoanhan101/users"
)
```

Create a value of type User from the users package using struct literal. However, since password is unexported, it cannot be compiled.

```go
func main() {
	u := users.User{
		Name: "Hoanh",
		ID:   101,

		password: "xxxx",
	}

	fmt.Printf("User: %#v\n", u)
}
```

```
unknown field 'password' in struct literal of type users.User
```

Exported types with embedded unexported types

Package users provides support for user management.

```go
package users
```

user represents information about a user. Unexported type with 2 exported fields.

```
type user struct {
	Name string
	ID   int
}
```

Manager represents information about a manager. Exported type embedded the unexported field user.

```
type Manager struct {
	Title string

	user
}
```

```
package main

import (
	"fmt"

	"github.com/hoanhan101/users"
)
```

Create a value of type Manager from the users package. During construction, we are only able to initialize the exported field Title. We cannot access the embedded type directly.

```
func main() {
	u := users.Manager{
		Title: "Dev Manager",
	}
```

However, once we have the manager value, the exported fields from that unexported type are accessible.

```
	u.Name = "Hoanh"
	u.ID = 101

	fmt.Printf("User: %#v\n", u)
}
```

```
User: users.Manager{Title:"Dev Manager", user:users.user{Name:"Hoanh",
ID:101}}
```

Again, we don't do this. A better way is to make user exported.

Software Design

Composition

Grouping types

Grouping By State

This is an example of using type hierarchies with an OOP pattern. This is not something we want to do in Go. Go does not have the concept of sub-typing. All types are their own and the concepts of base and derived types do not exist in Go. This pattern does not provide a good design principle in a Go program.

Animal contains all the base attributes for animals.

```go
type Animal struct {
	Name     string
	IsMammal bool
}
```

Speak provides generic behavior for all animals and how they speak. This is kind of useless because animals themselves cannot speak. This cannot apply to all animals.

```go
func (a *Animal) Speak() {
	fmt.Println("UGH!",
		"My name is", a.Name,
		", it is", a.IsMammal,
		"I am a mammal")
}
```

Dog contains everything from Animal, plus specific attributes that only a Dog has.

```go
type Dog struct {
	Animal
	PackFactor int
}
```

Speak knows how to speak like a dog.

```go
func (d *Dog) Speak() {
	fmt.Println("Woof!",
		"My name is", d.Name,
		", it is", d.IsMammal,
		"I am a mammal with a pack factor of", d.PackFactor)
}
```

Cat contains everything from Animal, plus specific attributes that only a Cat has.

```go
type Cat struct {
	Animal
	ClimbFactor int
}
```

Speak knows how to speak like a cat.

```go
func (c *Cat) Speak() {
	fmt.Println("Meow!",
		"My name is", c.Name,
		", it is", c.IsMammal,
		"I am a mammal with a climb factor of", c.ClimbFactor)
}
```

It's all fine until this one. This code will not compile. Here, we try to group the Cat and Dog based on the fact that they are Animals. We are trying to leverage sub-typing in Go. However, Go doesn't have it. Go doesn't encourage us to group types by common DNA. We need to stop designing APIs around this idea that types have a common DNA because if we only focus on who we are, it is very limiting on who we can group with. Sub-typing doesn't promote diversity. We lock types in a very small subset that can be grouped with. But when we focus on behavior, we open up the entire world to us.

Create a Dog by initializing its Animal parts and then its specific Dog attributes.

```go
animals := []Animal{
	Dog{
		Animal: Animal{
			Name:     "Fido",
			IsMammal: true,
            },
            PackFactor: 5,
},
```

Create a Cat by initializing its Animal parts and then its specific Cat attributes.

```
Cat{
      Animal: Animal{
            Name:     "Milo",
            IsMammal: true,
      },
      ClimbFactor: 4,
      },
}
```

Have the Animals speak.

```
for _, animal := range animals {
           animal.Speak()
      }
}
```

This code smells bad because:

- The Animal type provides an abstraction layer of reusable state.
- The program never needs to create or solely use a value of Animal type.
- The implementation of the Speak method for the Animal type is generalization.
- The Speak method for the Animal type is never going to be called.

Grouping By Behavior

This is an example of using composition and interfaces. This is something we want to do in Go.

This pattern does provide a good design principle in a Go program. We will group common types by their behavior and not by their state. What brilliant about Go is that it doesn't have to be configured ahead of time. The compiler automatically identifies interface and behaviors at compile time. It means that we can write code today that is compliant with any interface that exists today or tomorrow. It doesn't matter where that is declared because the compiler can do this on the fly. Stop thinking about a concrete base type. Let's think about what we do instead.

Speaker provide a common behavior for all concrete types to follow if they want to be a part of this group. This is a contract for these concrete types to follow. We get rid of the Animal type.

```go
type Speaker interface {
	Speak()
}
```

Dog contains everything a Dog needs.

```go
type Dog struct {
	Name       string
	IsMammal   bool
	PackFactor int
}
```

Speak knows how to speak like a dog. This makes a Dog now part of a group of concrete types that know how to speak.

```go
func (d Dog) Speak() {
	fmt.Println("Woof!",
		"My name is", d.Name,
		", it is", d.IsMammal,
		"I am a mammal with a pack factor of", d.PackFactor)
}
```

Cat contains everything a Cat needs. A little copy and paste can go a long way. Decoupling, in many cases, is a much better option than reusing the code.

```go
type Cat struct {
	Name              string
	IsMammal          bool
	ClimbFactor       int
}
```

Speak knows how to speak like a cat. This makes a Cat now part of a group of concrete types that know how to speak.

```go
func (c Cat) Speak() {
	fmt.Println("Meow!",
		"My name is", c.Name,
		", it is", c.IsMammal,
		"I am a mammal with a climb factor of", c.ClimbFactor)
}
```

Create a list of Animals that know how to speak.

```go
func main() {
	speakers := []Speaker{
```

Create a Dog by initializing Dog attributes.

```go
		Dog{
			Name:       "Fido",
			IsMammal:   true,
			PackFactor: 5,
		},
```

Create a Cat by initializing Cat attributes.

```go
		Cat{
			Name:        "Milo",
			IsMammal:    true,
			ClimbFactor: 4,
		},
}
```

Have the Speakers speak.

```go
	for _, spkr := range speakers {
		spkr.Speak()
	}
```

```
Woof! My name is Fido , it is true I am a mammal with a pack factor of 5
Meow! My name is Milo , it is true I am a mammal with a climb factor of
4
```

Guidelines around declaring types:

- Declare types that represent something new or unique. We don't want to create aliases just for readability.
- Validate that a value of any type is created or used on its own.
- Embed types not because we need the state but because we need the behavior. If we are not thinking about behavior, we are locking ourselves into the design that we cannot grow in the future.

- Question types that are aliases or abstraction for an existing type.
- Question types whose sole purpose is to share common state.

Decoupling

Struct Composition

Prototyping is important, as well as writing proof of concept and solving problems in the concrete first. Then we can ask ourselves: What can change? What change is coming? so we can start decoupling and refactor. Refactoring needs to become a part of the development cycle.

Here is the problem that we are trying to solve in this section. We have a system called Xenia that has a database. There is another system called Pillar, which is a web server with some front-end that consumes it. It has a database too. Our goal is to move Xenia's data into Pillar's system.

How long is it gonna take? How do we know when a piece of code is done so we can move on the next piece of code? If you are a technical manager, how do you know whether your debt is "wasting effort" or "not putting enough effort"?

To answer, being done has 2 parts. One is test coverage, 80% in general and 100% on the happy path. Second is about changes. By asking what can change, from a technical perspective and business perspective, we make sure that we refactor the code to be able to handle that change.

One example is, we can give you a concrete version in 2 days but we need 2 weeks to be able to refactor this code to deal with the change that we know it's coming. The plan is to solve one problem at a time. Don't be overwhelmed by everything. Write a little code, write some tests and refactor. Write a layer of APIs that work on top of each other, knowing that each layer is a strong foundation to the next.

Do not pay too much attention to the implementation detail. It's the mechanics here that are important. We are optimizing for correctness, not performance. We can always go back if it doesn't perform well enough to speed things up.

```go
package main


import (
	"errors"
	"fmt"
	"io"
	"math/rand"
	"time"
)
```

The first problem that we have to solve is that we need software that runs on a timer. It needs to connect to Xenia, read that database, identify all the data we haven't moved and pull it in.

```
func init() {
	rand.Seed(time.Now().UnixNano())
}
```

Data is the structure of the data we are copying. For simplicity, just pretend it is a string data.

```
type Data struct {
	Line string
}
```

Xenia is a system we need to pull data from.

```
type Xenia struct {
	Host    string
	Timeout time.Duration
}
```

Pull knows how to pull data out of Xenia. We could do func (*Xenia) Pull() (*Data, error) that return the data and error. However, this would cost an allocation on every call and we don't want that. Using the function below, we know data is a struct type and its size ahead of time. Therefore they could be on the stack.

```
func (*Xenia) Pull(d *Data) error {
	switch rand.Intn(10) {
	case 1, 9:
		return io.EOF

	case 5:
		return errors.New("Error reading data from Xenia")

	default:
		d.Line = "Data"
		fmt.Println("In:", d.Line)
		return nil
```

```
            }
}
```

Pillar is a system we need to store data into.

```
type Pillar struct {
      Host    string
      Timeout time.Duration
}
```

Store knows how to store data into Pillar. We are using pointer semantics for consistency.

```
func (*Pillar) Store(d *Data) error {
      fmt.Println("Out:", d.Line)
      return nil
}
```

System wraps Xenia and Pillar together into a single system. We have the API based on Xenia and Pillar. We want to build another API on top of this and use it as a foundation. One way is to have a type that has the behavior of being able to pull and store. We can do that through composition. System is based on the embedded value of Xenia and Pillar. And because of inner type promotion, System knows how to pull and store.

```
type System struct {
      Xenia
      Pillar
}
```

pull knows how to pull bulks of data from Xenia, leveraging the foundation that we have built.

We don't need to add a method to System to do this. There is no state inside System that we want the System to maintain. Instead, we want the System to understand the behavior.

Functions are a great way of writing API because functions can be more readable than any method can. We always want to start with an idea of writing API from the package level with functions.

When we write a function, all the input must be passed in. When we use a method, its signature doesn't indicate any level, what field or state that we are using on that value that we use to make the call.

```
func pull(x *Xenia, data []Data) (int, error) {
```

Range over the slice of data and share each element with the Xenial's Pull method.

```
	for i := range data {
		if err := x.Pull(&data[i]); err != nil {
			return i, err
		}
	}

	return len(data), nil
}
```

store knows how to store bulks of data into Pillar. Similar to the function above. We might wonder if it is efficient. However, we are optimizing for correctness, not performance. When it is done, we will test it. If it is not fast enough, we will add more complexities to make it run faster.

```
func store(p *Pillar, data []Data) (int, error) {
	for i := range data {
		if err := p.Store(&data[i]); err != nil {
			return i, err
		}
	}

	return len(data), nil
}
```

Copy knows how to pull and store data from the System. Now we can call the pull and store functions, passing Xenia and Pillar through.

```
func Copy(sys *System, batch int) error {
	data := make([]Data, batch)


	for {
		i, err := pull(&sys.Xenia, data)
		if i > 0 {
			if _, err := store(&sys.Pillar, data[:i]); err != nil
{
				return err
				}
			}


			if err != nil {
				return err
			}
		}
}
```

```
func main() {
	sys := System{
		Xenia: Xenia{
			Host:    "localhost:8000",
			Timeout: time.Second,
		},
		Pillar: Pillar{
			Host:    "localhost:9000",
			Timeout: time.Second,
		},
	}


	if err := Copy(&sys, 3); err != io.EOF {
		fmt.Println(err)
	}
}
```

```
In: Data
In: Data
In: Data
Out: Data
Out: Data
Out: Data
In: Data
In: Data
Out: Data
Out: Data
```

Decoupling With Interface

By looking at the API (functions), we need to decouple the API from the concrete implementation. The decoupling that we do must get all the way down into initialization. To do this right, the only piece of code that we need to change is initialization. Everything else should be able to act on the behavior that these types are gonna provide.

pull is based on the concrete. It only knows how to work on Xenia. However, if we are able to decouple pull to use any system that knows how to pull data, we can get the highest level of decoupling. Since the algorithm we have is already efficient, we don't need to add another level of generalization and destroy the work we did in the concrete. Same thing with store.

It is nice to work from the concrete up. When we do this, not only are we solving problems efficiently and reducing technical debt but the contracts, they come to us. We already know what the contract is for pulling/storing data. We already validate that and this is what we need.

Let's just decouple these 2 functions and add 2 interfaces. The Puller interface knows how to pull and the Storer knows how to store.
Xenia already implemented the Puller interface and Pillar already implemented the Storer interface. Now we can come into pull/store, decouple this function from the concrete.

Instead of passing Xenial and Pillar, we pass in the Puller and Storer. The algorithm doesn't change. All we are doing now is calling pull/store indirectly through the interface value.

```go
package main

import (
	"errors"
	"fmt"
	"io"
	"math/rand"
	"time"
)

func init() {
	rand.Seed(time.Now().UnixNano())
}
```

Data is the structure of the data we are copying.

```go
type Data struct {
	Line string
}
```

Puller declares behavior for pulling data.

```
type Puller interface {
	Pull(d *Data) error
}
```

Storer declares behavior for storing data.

```
type Storer interface {
	Store(d *Data) error
}
```

Xenia is a system we need to pull data from.

```
type Xenia struct {
	Host    string
	Timeout time.Duration
}
```

Pull knows how to pull data out of Xenia.

```
func (*Xenia) Pull(d *Data) error {
	switch rand.Intn(10) {
	case 1, 9:
		return io.EOF

	case 5:
		return errors.New("Error reading data from Xenia")

	default:
		d.Line = "Data"
		fmt.Println("In:", d.Line)
		return nil
	}
}
```

Pillar is a system we need to store data into.

```
type Pillar struct {
	Host    string
```

```go
	Timeout time.Duration
}
```

Store knows how to store data into Pillar.

```go
func (*Pillar) Store(d *Data) error {
	fmt.Println("Out:", d.Line)
	return nil
}
```

System wraps Xenia and Pillar together into a single system.

```go
type System struct {
	Xenia
	Pillar
}
```

pull knows how to pull bulks of data from any Puller.

```go
func pull(p Puller, data []Data) (int, error) {
	for i := range data {
		if err := p.Pull(&data[i]); err != nil {
			return i, err
		}
	}

	return len(data), nil
}
```

store knows how to store bulks of data from any Storer.

```go
func store(s Storer, data []Data) (int, error) {
	for i := range data {
		if err := s.Store(&data[i]); err != nil {
			return i, err
		}
	}
	return len(data), nil
}
```

Copy knows how to pull and store data from the System.

```go
func Copy(sys *System, batch int) error {
	data := make([]Data, batch)

	for {
		i, err := pull(&sys.Xenia, data)
		if i > 0 {
			if _, err := store(&sys.Pillar, data[:i]); err != nil
{
				return err
			}
		}

		if err != nil {
			return err
		}
	}
}
```

```go
func main() {
	sys := System{
		Xenia: Xenia{
			Host:    "localhost:8000",
			Timeout: time.Second,
		},
		Pillar: Pillar{
			Host:    "localhost:9000",
			Timeout: time.Second,
		},
	}

	if err := Copy(&sys, 3); err != io.EOF {
		fmt.Println(err)
	}
}
```

```
In: Data
In: Data
In: Data
Out: Data
Out: Data
Out: Data
In: Data
In: Data
Out: Data
```

```
Out: Data
```

Interface Composition

Let's just add another interface. Let's use interface composition to do this. PullStorer has both behaviors: Puller and Storer. Any concrete type that implements both pull and store is a PullStorer. System is a PullStorer because it is embedded in these 2 types, Xenia and Pillar. Now we just need to go into Copy, replace the system pointer with PullStorer and no other code needs to change.

Looking closely at Copy, there is something that could potentially confuse us. We are passing the PullStorer interface value directly into pull and store respectively.

If we look into pull and store, they don't want a PullStorer. One wants a Puller and one wants a Storer. Why does the compiler allow us to pass a value of different type value while it didn't allow us to do that before?

This is because Go has what is called: implicit interface conversion. This is possible because:

- All interface values have the exact same model (implementation details).
- If the type information is clear, the concrete type that exists in one interface has enough behaviors for another interface. It is true that any concrete type that is stored inside of a PullStorer must also implement the Storer and Puller.

Let's further look into the code.

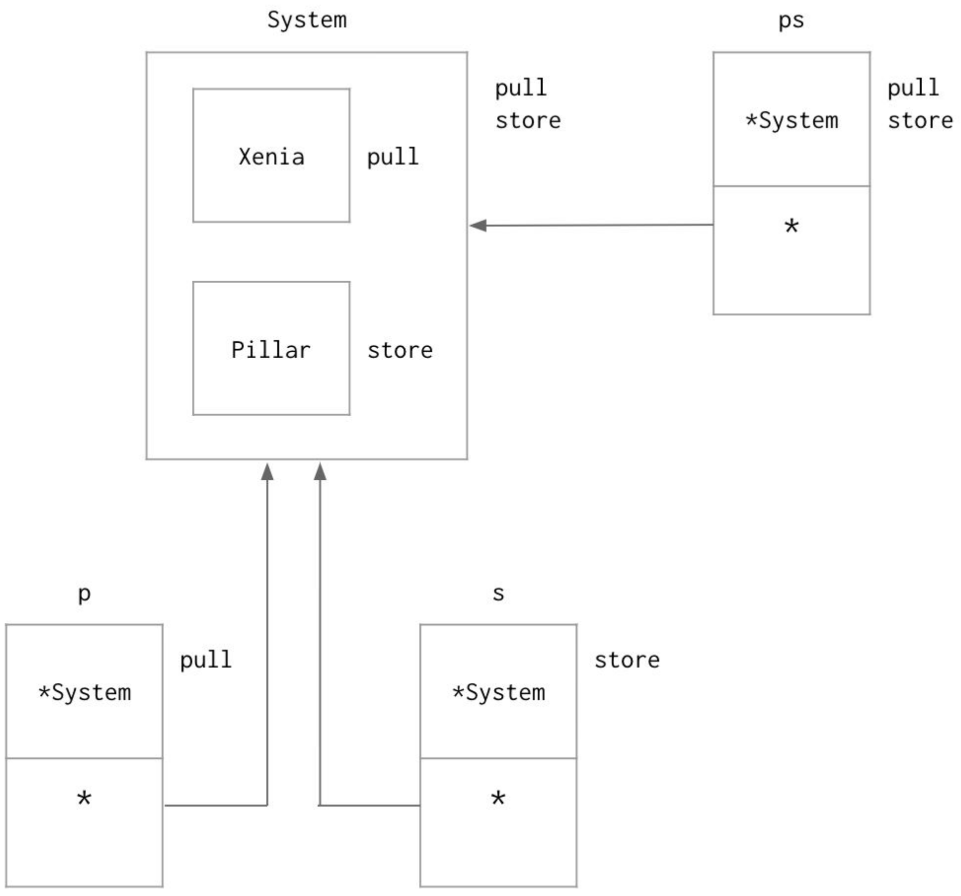
In the main function, we are creating a value of our System type. As we know, our System type value is based on the embedding of two concrete types: Xenia and Pillar, where Xenia knows how to pull and Pillar knows how to store. Because of inner type promotion, System knows how to pull and store both inherently. We are passing the address of our System to Copy. Copy then creates the PullStorer interface. The first word is a System pointer and the second word points to the original value. This interface now knows how to pull and store. When we call pull off of ps, we call pull off of System, which eventually call pull off of Xenia.

Here is the kicker: the implicit interface conversion.

We can pass the interface value ps to pull because the compiler knows that any concrete type stored inside the PullStorer must also implement Puller. We end up with another interface called Puller. Because the memory models are the same for all interfaces, we just copy those 2 words so they are all sharing the same interface type. Now when we call pull off of Puller, we call pull off of System.

Similar to Storer.

All using value semantic for the interface value and pointer semantic to share.



```go
package main

import (
	"errors"
	"fmt"
	"io"
	"math/rand"
	"time"
)

func init() {
	rand.Seed(time.Now().UnixNano())
}
```

Data is the structure of the data we are copying.

```go
type Data struct {
	Line string
}
```

Puller declares behavior for pulling data.

```go
type Puller interface {
	Pull(d *Data) error
}
```

Storer declares behavior for storing data.

```go
type Storer interface {
	Store(d *Data) error
}
```

PullStorer declares behaviors for both pulling and storing.

```go
type PullStorer interface {
	Puller
	Storer
}
```

Xenia is a system we need to pull data from.

```go
type Xenia struct {
	Host    string
	Timeout time.Duration
}
```

Pull knows how to pull data out of Xenia.

```go
func (*Xenia) Pull(d *Data) error {
	switch rand.Intn(10) {
		case 1, 9:
			return io.EOF

		case 5:
			return errors.New("Error reading data from Xenia")

		default:
			d.Line = "Data"
			fmt.Println("In:", d.Line)
			return nil
		}
}
```

Pillar is a system we need to store data into.

```
type Pillar struct {
	Host    string
	Timeout time.Duration
}
```

Store knows how to store data into Pillar.

```
func (*Pillar) Store(d *Data) error {
	fmt.Println("Out:", d.Line)
	return nil
}
```

System wraps Xenia and Pillar together into a single system.

```
type System struct {
	Xenia
	Pillar
}
```

pull knows how to pull bulks of data from any Puller.

```
func pull(p Puller, data []Data) (int, error) {
	for i := range data {
		if err := p.Pull(&data[i]); err != nil {
			return i, err
		}
	}

	return len(data), nil
}
```

store knows how to store bulks of data from any Storer.

```
func store(s Storer, data []Data) (int, error) {
```

```go
	for i := range data {
		if err := s.Store(&data[i]); err != nil {
			return i, err
		}
	}

	return len(data), nil
}
```

Copy knows how to pull and store data from any System.

```go
func Copy(ps PullStorer, batch int) error {
	data := make([]Data, batch)

	for {
		i, err := pull(ps, data)
		if i > 0 {
			if _, err := store(ps, data[:i]); err != nil {
				return err
			}
		}

		if err != nil {
			return err
		}
	}
}
```

```go
func main() {
	sys := System{
		Xenia: Xenia{
			Host:    "localhost:8000",
			Timeout: time.Second,
		},
		Pillar: Pillar{
			Host:    "localhost:9000",
			Timeout: time.Second,
		},
	}

	if err := Copy(&sys, 3); err != io.EOF {
		fmt.Println(err)
	}
}
```

```
In: Data
In: Data
In: Data
Out: Data
Out: Data
Out: Data
In: Data
In: Data
Out: Data
Out: Data
```

Decoupling With Interface Composition

We change our concrete type System. Instead of using two concrete types Xenia and Pillar, we use 2 interface types Puller and Storer. Our concrete type System where we can have concrete behaviors is now based on the embedding of 2 interface types. It means that we can inject any data, not based on the common DNA but on the data that provides the capability, the behavior that we need.

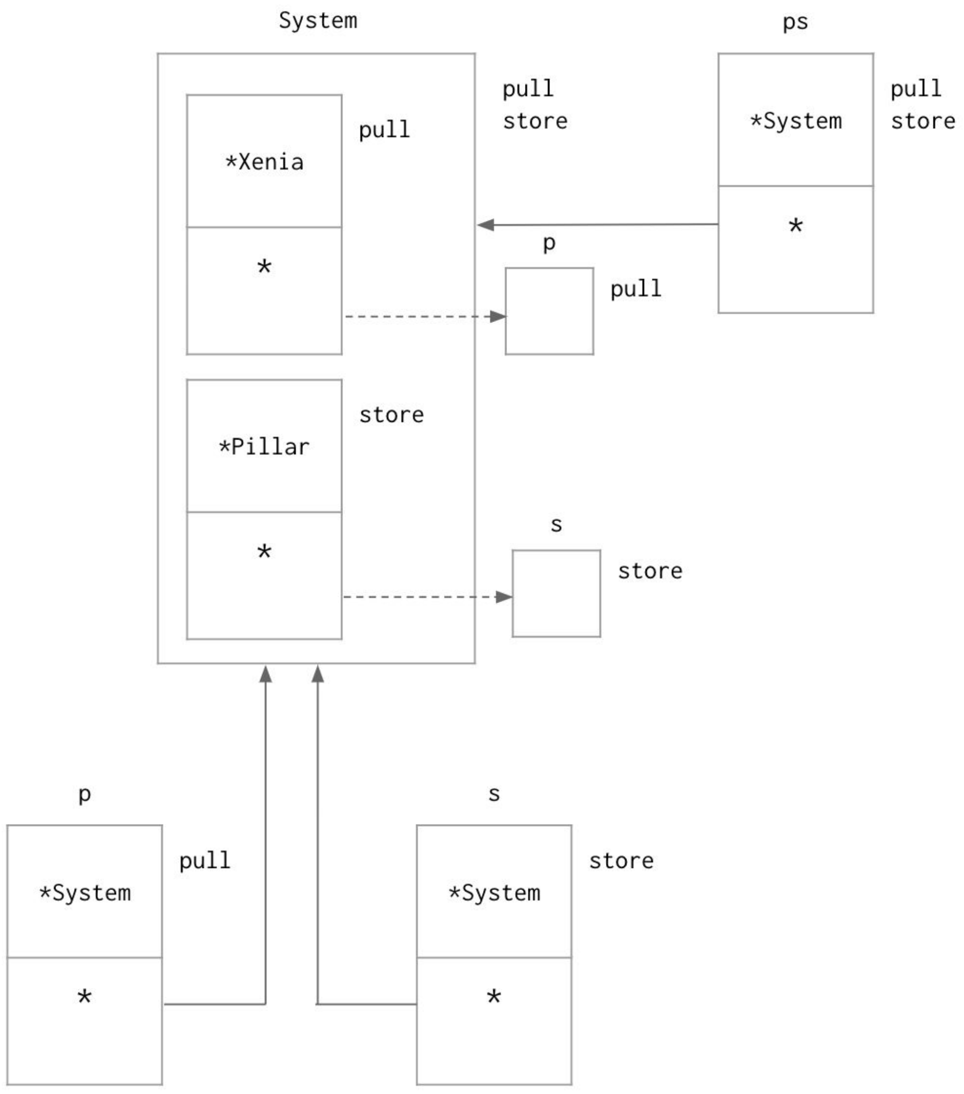
Now our code can be fully decoupled because any value that implements the Puller interface can be stored inside the System (same with Storer interface). We can create multiple Systems and that data can be passed in Copy.

We don't need a method here. We just need one function that accepts data and its behavior will change based on the data we put in.

Now System is not based on Xenia and Pillar anymore. It is based on 2 interfaces, one that stores Xenia and one that stores Pillar. We get the extra layer of decoupling.

If the system changes, no big deal. We replace the system as we need to during the program startup.

We solve this problem. We put this in production. Every single refactoring that we did went into production before we did the next one. We keep minimizing technical debt.



```go
package main

import (
	"errors"
	"fmt"
	"io"
	"math/rand"
	"time"
)

func init() {
	rand.Seed(time.Now().UnixNano())
}
```

Data is the structure of the data we are copying.

```go
type Data struct {
	Line string
}
```

Puller declares behavior for pulling data.

```go
type Puller interface {
	Pull(d *Data) error
}
```

Storer declares behavior for storing data.

```go
type Storer interface {
	Store(d *Data) error
}
```

PullStorer declares behaviors for both pulling and storing.

```go
type PullStorer interface {
	Puller
	Storer
}
```

Xenia is a system we need to pull data from.

```go
type Xenia struct {
	Host    string
	Timeout time.Duration
}
```

Pull knows how to pull data out of Xenia.

```go
func (*Xenia) Pull(d *Data) error {
	switch rand.Intn(10) {
		case 1, 9:
			return io.EOF

		case 5:
			return errors.New("Error reading data from Xenia")


		default:
			d.Line = "Data"
			fmt.Println("In:", d.Line)
			return nil
		}
}
```

Pillar is a system we need to store data into.

```go
type Pillar struct {
	Host    string
	Timeout time.Duration
}
```

Store knows how to store data into Pillar.

```go
func (*Pillar) Store(d *Data) error {
	fmt.Println("Out:", d.Line)
	return nil
}
```

System wraps Pullers and Stores together into a single system.

```go
type System struct {
	Puller
	Storer
}
```

pull knows how to pull bulks of data from any Puller.

```go
func pull(p Puller, data []Data) (int, error) {
	for i := range data {
		if err := p.Pull(&data[i]); err != nil {
			return i, err
		}
	}

	return len(data), nil
}
```

store knows how to store bulks of data from any Storer.

```go
func store(s Storer, data []Data) (int, error) {
	for i := range data {
		if err := s.Store(&data[i]); err != nil {
			return i, err
		}
	}

	return len(data), nil
}
```

Copy knows how to pull and store data from any System.

```
func Copy(ps PullStorer, batch int) error {
	data := make([]Data, batch)

	for {
		i, err := pull(ps, data)
		if i > 0 {
			if _, err := store(ps, data[:i]); err != nil {
				return err
			}
		}


		if err != nil {
			return err
		}
	}
}


func main() {
	sys := System{
		Puller: &Xenia{
			Host:    "localhost:8000",
			Timeout: time.Second,
		},
		Storer: &Pillar{
		Host:    "localhost:9000",
		Timeout: time.Second,
		},
	}


	if err := Copy(&sys, 3); err != io.EOF {
		fmt.Println(err)
```

```
      }
}
```

```
In: Data
In: Data
In: Data
Out: Data
Out: Data
Out: Data
In: Data
In: Data
Out: Data
Out: Data
```

Conversion

Interface Conversions

Mover provides support for moving things.

```go
type Mover interface {
      Move()
}
```

Locker provides support for locking and unlocking things.

```go
type Locker interface {
      Lock()
      Unlock()
}
```

MoveLocker provides support for moving and locking things.

```go
type MoveLocker interface {
      Mover
      Locker
}
```

bike represents a concrete type for the example.

```go
type bike struct{}
```

Move can change the position of a bike.

```
func (bike) Move() {
	fmt.Println("Moving the bike")
}
```

Lock prevents a bike from moving.

```
func (bike) Lock() {
	fmt.Println("Locking the bike")
}
```

Unlock allows a bike to be moved.

```
func (bike) Unlock() {
	fmt.Println("Unlocking the bike")
}
```

```
func main() {
```

Declare variables of the MoveLocker and Mover interfaces set to their zero value.

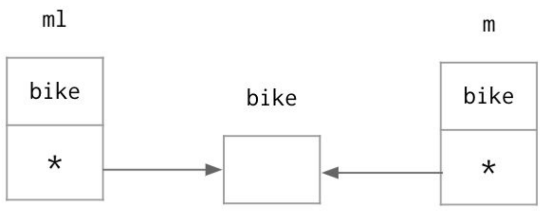
```
	var ml MoveLocker
	var m Mover
```

Create a value of type bike and assign the value to the MoveLocker interface value.

```
	ml = bike{}
```

An interface value of type MoveLocker can be implicitly converted into a value of type Mover. They both declare a method named move.

```
	m = ml
```



However, we cannot go in the other direction, like so:

```
    ml = m
```

The compiler will say:

```
cannot use m (type Mover) as type MoveLocker in assignment: Mover does
not implement MoveLocker (missing Lock method).
```

Type assertion

Interface type Mover does not declare methods named lock and unlock. Therefore, the compiler can't perform an implicit conversion to assign a value of interface type Mover to an interface value of type MoveLocker. It is irrelevant that the concrete type value of type bike that is stored inside of the Mover interface value implements the MoveLocker interface. We can perform a type assertion at runtime to support the assignment.

Perform a type assertion against the Mover interface value to access a COPY of the concrete type value of type bike that was stored inside of it. Then assign the COPY of the concrete type to the MoveLocker interface.

This is the syntax for type assertion. We are taking the interface value itself, dot (bike). We are using bike as a parameter. If m is not nil and there is a bike inside of m, we will get a copy of it since we are using value semantics. Or else, a panic occurs.

b is having a copy of bike value.

```
    b := m.(bike)
```

We can prevent panic when type assertion breaks by destructuring the boolean value that represents type assertion result.

```
	b, ok := m.(bike)
	fmt.Println("Does m has value of bike?:", ok)

	ml = b
```

```
Does m has value of bike?: true
```

It's important to note that the type assertion syntax provides a way to state what type of value is stored inside the interface. This is more powerful from a language and readability standpoint, than using a casting syntax, like in other languages.

Runtime Type Assertion

```
package main

import (
	"fmt"
	"math/rand"
	"time"
)
```

car represents something you drive.

```
type car struct{}
```

String implements the fmt.Stringer interface.

```
func (car) String() string {
	return "Vroom!"
}
```

cloud represents somewhere you store information.

```
type cloud struct{}
```

String implements the fmt.Stringer interface.

```
func (cloud) String() string {
	return "Big Data!"
}
```

Seed the number random generator.

```go
func main() {
	rand.Seed(time.Now().UnixNano())
```

Create a slice of the Stringer interface values.

```go
	mvs := []fmt.Stringer{
		car{},
		cloud{},
	}
```

Let's run this experiment ten times.

```go
	for i := 0; i < 10; i++ {
		rn := rand.Intn(2)
```

Perform a type assertion that we have a concrete type of cloud in the interface value we randomly chose. This shows us that this checking is at runtime, not compile time.

```go
		if v, ok := mvs[rn].(cloud); ok {
			fmt.Println("Got Lucky:", v)
			continue
		}
```

We have to guarantee that the variable in question (x in `x.(T)`) can always be asserted correctly as T type Or else, We wouldn't want to use that ok variable because we want it to panic if there is an integrity issue. We must shut it down immediately if that happens if we cannot recover from a panic and guarantee that we are back at 100% integrity, the software has to be restarted.

Shutting down means you have to call log.Fatal, os.exit, or panic for stack trace. When we use type assertion, we need to understand when it is okay that whatever we are asking for is not there.

If the type assertion is causing us to call the concrete value out, that should raise a big flag. We are using interface to maintain a level of decoupling and now we are using type assertion to go back to the concrete.

When we are in the concrete, we are putting our codes in the situation where cascading changes can cause widespread refactoring. What we want with interface is the opposite, internal changes minimize cascading changes.

```
            fmt.Println("Got Unlucky")
        }
}
```

```
Got Unlucky
Got Unlucky
Got Lucky: Big Data!
Got Unlucky
Got Lucky: Big Data!
Got Lucky: Big Data!
Got Unlucky
Got Lucky: Big Data!
Got Unlucky
Got Unlucky
```

Interface Pollution

It comes from the fact that people are designing software from the interface first down instead of concrete type up.

So, why are we using an interface here?

Myth #1: We are using interfaces because we have to use interfaces.

Answer: No. We don't have to use interfaces. We use it when it is practical and reasonable to do so.
Even though they are wonderful, there is a cost of using interfaces: a level of indirection and potential allocation when we store concrete type inside of them. Unless the cost of that is worth whatever decoupling we are getting, we shouldn't be using interfaces.

Myth #2: We need to be able to test our code so we need to use interfaces.

Answer: No. We must design our API that is usable for user application developer first, not our test.

Below is an example that creates interface pollution by improperly using an interface when one is not needed.

Server defines a contract for TCP servers. This is a little bit of a smell because this is some sort of APIs that is going to be exposed to users and already that is a lot of behaviors brought in a generic interface.

```
type Server interface {
	Start() error
	Stop() error
	Wait() error
}
```

server is our Server implementation. They match the name. However, that is not necessarily bad.

```
type server struct {
	host string
}
```

NewServer returns an interface value of type Server with a server implementation. Here is the factory function. It immediately starts to smell even worse. It is returning the interface value.

It is not that functions and interfaces cannot return interface values. They can. But normally, that should raise a flag. The concrete type is the data that has the behavior and the interface normally should be used as accepting the input to the data, not necessarily going out.

SMELL - Storing an unexported type pointer in the interface.

```
func NewServer(host string) Server {
	return &server{host}
}
```

Start allows the server to begin to accept requests. From now, let’
s pretend there is a specific implementation for each of these methods.

```
func (s *server) Start() error {
	return nil
}
```

Stop shuts the server down.

```
func (s *server) Stop() error {
	return nil
}
```

Wait prevents the server from accepting new connections.

```
func (s *server) Wait() error {
	return nil
}
```

```
func main() {
```

Create a new Server.

```
	srv := NewServer("localhost")
```

Use the API.

```
	srv.Start()
	srv.Stop()
	srv.Wait()
}
```

This code here couldn't care less nor would it change if srv was the concrete type, not the interface. The interface is not providing any level of support whatsoever. There is no decoupling here that is happening. It is not giving us anything special here. All is doing is causing us another level of indirection.

It smells because:
- The package declares an interface that matches the entire API of its own concrete type.
- The interface is exported but the concrete type is unexported.
- The factory function returns the interface value with the unexported concrete type value inside.
- The interface can be removed and nothing changes for the user of the API.
- The interface is not decoupling the API from change.

Remove Interface Pollution

We're going to remove the improper interface usage from the previous program.

Server implementation.

```
type Server struct {
```

```go
	host string
}
```

NewServer returns just a concrete pointer of type Server

```go
func NewServer(host string) *Server {
	return &Server{host}
}
```

Start allows the server to begin to accept requests.

```go
func (s *Server) Start() error {
	return nil
}
```

Stop shuts the server down.

```go
func (s *Server) Stop() error {
	return nil
}
```

Wait prevents the server from accepting new connections.

```go
func (s *Server) Wait() error {
	return nil
}
```

Create a new Server.

```go
func main() {
	srv := NewServer("localhost")
```

Use the APIs.

```go
	srv.Start()
	srv.Stop()
	srv.Wait()
}
```

Guidelines around interface pollution:

Use an interface:

- When users of the API need to provide an implementation detail.
- When APIs have multiple implementations that need to be maintained.
- When parts of the APIs that can change have been identified and require decoupling.

Question an interface:

- When its only purpose is for writing testable API’s (write usable APIs first).
- When it’s not providing support for the API to decouple from change.
- When it's not clear how the interface makes the code better.

Mocking

Package To Mock

It is important to mock things. Most things over the network can be mocked in our test. However, mocking our database is a different story because it is too complex. This is where Docker can come in and simplify our code by allowing us to launch our database while running our tests and have that clean database for everything we do.

Every API only needs to focus on its test. We no longer have to worry about the application user or user over API test. We used to worry about: if we don't have that interface, the user who uses our API can't write tests. That is gone. The example below will demonstrate the reason.

Imagine we are working at a company that decides to incorporate Go as a part of its stack. They have their internal pubsub system that all applications are supposed to use. Maybe they are doing event sourcing and there is a single pubsub platform they are using that is not going to be replaced. They need the pubsub API for Go that they can start building services that connect into this event source. So what can change? Can the event source change?

If the answer is no, then it immediately tells us that we don't need to use interfaces. We can build the entire API in the concrete, which we would do first anyway. We then write tests to make sure everything works.

A couple days later, they come to us with a problem. They have to write tests and they cannot hit the pubsub system directly when my test runs so they need to mock that out. They want us to give them an interface. However, we don't need an interface because our API doesn't need an interface. They need an interface, not us. They need to decouple from the pubsub system, not us.

They can do any decoupling they want because this is Go. The next file will be an example of their application. Package pubsub simulates a package that provides publication/subscription type services.

```
package main

import (
	"fmt"
)
```

PubSub provides access to a queue system.

```
type PubSub struct {
	host string
}
```

New creates a pubsub value for use.

```
func New(host string) *PubSub {
	ps := PubSub{
		host: host,
	}

	return &ps
}
```

Publish sends the data to the specified key.

```
func (ps *PubSub) Publish(key string, v interface{}) error {

	fmt.Println("Actual PubSub: Publish")
	return nil
}
```

Subscribe sets up a request to receive messages from the specified key.

```
func (ps *PubSub) Subscribe(key string) error {
	fmt.Println("Actual PubSub: Subscribe")
	return nil
}
```

Client

Sample program to show how we can personally mock concrete types when we need to for our own packages or tests.

```
package main

import (
	"fmt"
)
```

publisher is an interface to allow this package to mock the pubsub package. When we are writing our applications, declare our own interface that maps out all the APIs calls we need for the APIs. The concrete types APIs in the previous files satisfy it out of the box. We can write the entire application with mocking decoupling from concrete implementations.

```
type publisher interface {
	Publish(key string, v interface{}) error
	Subscribe(key string) error
}
```

mock is a concrete type to help support the mocking of the pubsub package.

```
type mock struct{}
```

Publish implements the publisher interface for the mock.

```
func (m *mock) Publish(key string, v interface{}) error {
	// ADD YOUR MOCK FOR THE PUBLISH CALL.
	fmt.Println("Mock PubSub: Publish")
	return nil
}
```

Subscribe implements the publisher interface for the mock.

```
func (m *mock) Subscribe(key string) error {
	// ADD YOUR MOCK FOR THE SUBSCRIBE CALL.
	fmt.Println("Mock PubSub: Subscribe")
	return nil
```

```go
}
```

Create a slice of publisher interface values. Assign the address of a pubsub. PubSub value and the address of a mock value.

```go
func main() {
	pubs := []publisher{
		New("localhost"),
		&mock{},
	}
```

Range over the interface value to see how the publisher interface provides the level of decoupling the user needs. The pubsub package did not need to provide the interface type.

```go
	for _, p := range pubs {
		p.Publish("key", "value")
		p.Subscribe("key")
		}
}
```

Error Handling

Default error values

Integrity matters. Nothing trumps integrity. Therefore, part of integrity is error handling. It is a big part of what we do everyday. It has to be a part of the main code. First, let's look at the language mechanic first on how the default error type is implemented.

```go
package main

import "fmt"
```

http://golang.org/pkg/builtin/#error

This is pre-included in the language so it looks like an unexported type. It has one active behavior, which is Error returned a string. Error handling is decoupled because we are always working with error interface when we are testing our code.

Errors in Go are really just values. We are going to valuate these through the decoupling of the interface. Decoupling error handling means that cascading changes will bubble up through the user application, causes cascading wide effect through the code base. It's important that we leverage the interface here as much as we can.

```
type error interface {
      Error() string
}
```

http://golang.org/src/pkg/errors/errors.go

This is the default concrete type that comes from the error package. It is an unexported type that has an unexported field. This gives us enough context to make us form a decision.

We have responsibility around error handling to give the caller enough context to make them form a decision so they know how to handle this situation.

```
type errorString struct {
      s string
}
```

http://golang.org/src/pkg/errors/errors.go

This is using a pointer receiver and returning a string. If the caller must call this method and parse a string to see what is going on then we fail.

This method is only for logging information about the error.

```
func (e *errorString) Error() string {
      return e.s
}
```

http://golang.org/src/pkg/errors/errors.go

New returns an error interface that formats as the given text. When we call New, what we are doing is creating errorString value, putting some sort of string in there.. Since we are returning the address of a concrete type, the user will get an error interface value where the first word is a *errorString and the second word points to the original value. We are going to stay decoupled during the error handling.

```
func New(text string) error {
      return &errorString{text}
}
```

This is a very traditional way of error handling in Go. We are calling webCall and returning the error interface and storing that in a variable.

nil is a special value in Go. What "error != nil" actually means is that we are asking if there is a concrete type value that is stored in error type interface. Because if error is not nil, there is a concrete value stored inside. If it is the case, we've got an error.

Now do we handle the error, do we return the error up the call stack for someone else to handle? We will talk about this later.

```go
func main() {
	if err := webCall(); err != nil {
		fmt.Println(err)
		return
	}

	fmt.Println("Life is good")
}
```

webCall performs a web operation.

```go
func webCall() error {
	return New("Bad Request")
}
```

Error variables

Let's use error variables to determine the exact error being returned.

```go
5package main

import (
	"errors"
	"fmt"
)
```

We want these to be on the top of the source code file. Naming convention: starting with Err. They have to be exported because our users need to access them.

These are all error interfaces that we have discussed in the last file, with variables tied to them. The contexts for these errors are the variables themselves. This allows us to continue using the default error type, that unexported type with unexported field to maintain a level of decoupling through error handling.

ErrBadRequest is returned when there are problems with the request. ErrPageMoved is returned when a 301/302 is returned.

```go
var (
	ErrBadRequest = errors.New("Bad Request")
	ErrPageMoved = errors.New("Page Moved")
)
```

```go
func main() {
	if err := webCall(true); err != nil {
		switch err {
		case ErrBadRequest:
			fmt.Println("Bad Request Occurred")
			return

		case ErrPageMoved:
			fmt.Println("The Page moved")
			return

		default:
			fmt.Println(err)
			return
		}
	}

	fmt.Println("Life is good")
}
```

webCall performs a web operation.

```go
func webCall(b bool) error {
	if b {
		return ErrBadRequest
	}

	return ErrPageMoved
}
```

```
Bad Request Occurred
```

Type as context

It is not always possible to be able to say the interface value itself will be enough context. Sometimes, it requires more context. For example, a networking problem can be really complicated. Error variables wouldn't work there. Only when the error variables wouldn't work, we should go ahead and start working with a custom concrete type for the error.

Below are two custom error types from the JSON package in the standard library and how we can use those. This is type as context.

http://golang.org/src/pkg/encoding/json/decode.go

package main

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

An UnmarshalTypeError describes a JSON value that was not appropriate for
a value of a specific Go type. Naming convention: The word "Error" ends at the name of the type.

```go
type UnmarshalTypeError struct {
	Value string       // description of JSON value
	Type  reflect.Type // type of Go value it could not be assigned to
}
```

UnmarshalTypeError implements the error interface. We are using pointer semantics. In the implementation, we are validating all the fields being used in the error message. If not, we have a problem. Because why would you add a field to the custom error type and not displaying it on your log when this method would call. We only do this when we really need it.

```go
func (e *UnmarshalTypeError) Error() string {
	return "json: cannot unmarshal " + e.Value + " into Go value of
type " + e.Type.String()
}
```

An InvalidUnmarshalError describes an invalid argument passed to Unmarshal. The argument to Unmarshal must be a non-nil pointer. This concrete type is used when we don't pass the address of a value into the Unmarshal function.

```
type InvalidUnmarshalError struct {
	Type reflect.Type
}
```

InvalidUnmarshalError implements the error interface.

```
func (e *InvalidUnmarshalError) Error() string {
	if e.Type == nil {
		return "json: Unmarshal(nil)"
	}

	if e.Type.Kind() != reflect.Ptr {
		return "json: Unmarshal(non-pointer " + e.Type.String() +
")"
	}
	return "json: Unmarshal(nil " + e.Type.String() + ")"
}
```

user is a type for use in the Unmarshal call.

```
type user struct {
	Name int
}
```

```
func main() {
	var u user
	err := Unmarshal([]byte(`{"name":"bill"}`), u) // Run with a value
and pointer.
	if err != nil {
This is a special type assertion that only works on the switch.
		switch e := err.(type) {
			case *UnmarshalTypeError:
				fmt.Printf("UnmarshalTypeError: Value[%s]
Type[%v]\n",e.Value, e.Type)
			case *InvalidUnmarshalError:
				fmt.Printf("InvalidUnmarshalError: Type[%v]\n",
e.Type)
			default:
				fmt.Println(err)
```

```go
            }
            return
      }

      fmt.Println("Name:", u.Name)
}
```

Unmarshal simulates an unmarshal call that always fails. Notice the parameters here: The first one is a slice of byte and the second one is an empty interface. The empty interface basically says nothing, which means any value can be passed into this function. We are going to reflect on the concrete type that is stored inside this interface and we are going to validate that it is a pointer or not nil. We then return different error types depending on these.

```go
func Unmarshal(data []byte, v interface{}) error {
      rv := reflect.ValueOf(v)
      if rv.Kind() != reflect.Ptr || rv.IsNil() {
            return &InvalidUnmarshalError{reflect.TypeOf(v)}
      }

      return &UnmarshalTypeError{"string", reflect.TypeOf(v)}
}
```

There is one flaw when using type as context here. In this case, we are now going back to the concrete. We walk away from the decoupling because our code is now bound to these concrete types. If the developer who wrote the json package makes any changes to these concrete types, that's gonna create a cascading effect all the way through our code. We are no longer protected by the decoupling of the error interface.

This sometimes has to happen. Can we do something different not to lose the decoupling. This is where the idea of behavior as context comes in.

Behavior as context

Behavior as context allows us to use a custom error type as our context but avoid that type assertion back to the concrete. We get to maintain a level of decoupling in our code.

```go
package main

import (
      "bufio"
      "fmt"
```

```
	"io"
	"log"
	"net"
)
```

client represents a single connection in the room.

```
type client struct {
	name   string
	reader *bufio.Reader
}
```

TypeAsContext shows how to check multiple types of possible custom error types that can be returned from the net package.

```
func (c *client) TypeAsContext() {
	for {
```

We are using reader interface value to decouple ourselves from the network read.

```
		line, err := c.reader.ReadString('\n')
		if err != nil {
```

This is using type as context like the previous example. What special here is the method named Temporary. If it is, we can keep going but if not, we have to break things down and build things back up. Every one of these cases care only about 1 thing: the behavior of Temporary. This is what important. We can switch here, from type as context to type as behavior if we do this type assertion and only ask about the potential behavior of that concrete type itself. We can go ahead and declare our own interface called temporary like below.

```
			switch e := err.(type) {
			case *net.OpError:
				if !e.Temporary() {
					log.Println("Temporary: Client leaving
chat")
					return
				}

			case *net.AddrError:
				if !e.Temporary() {
					log.Println("Temporary: Client leaving
chat")
```

```
                        return
                    }

                case *net.DNSConfigError:
                    if !e.Temporary() {
                        log.Println("Temporary: Client leaving
chat")
                        return
                    }

                default:
                    if err == io.EOF {
                        log.Println("EOF: Client leaving chat")
                        return
                    }

                    log.Println("read-routine", err)
                }
            }

            fmt.Println(line)
        }
}
```

temporary is declared to test for the existence of the method coming from the net package. Because Temporary is the only behavior we care about. If the concrete type has the method named temporary then this is what we want. We get to stay decoupled and continue to work at the interface level.

```
type temporary interface {
      Temporary() bool
}
```

BehaviorAsContext shows how to check for the behavior of an interface that can be returned from the net package.

```
func (c *client) BehaviorAsContext() {
      for {
            line, err := c.reader.ReadString('\n')
            if err != nil {
                  switch e := err.(type) {
```

We can reduce 3 cases into 1 by asking in the case here during type assertion: Does the concrete type stored inside the error interface also implement this interface. We can declare and leverage that interface ourselves.

```
                    case temporary:
                          if !e.Temporary() {
                                log.Println("Temporary: Client leaving
chat")
                                return
                          }

                    default:
                          if err == io.EOF {
                                log.Println("EOF: Client leaving chat")
                                return
                          }

                          log.Println("read-routine", err)
                    }
              }

              fmt.Println(line)
        }
}
```

Lesson:

Thank to Go Implicit Conversion. We can maintain a level of decoupling by creating an interface with methods or behaviors that we only want, and use it instead of concrete type for type assertion switch.

Finding the bug

```
package main

import "log"
```

customError is just an empty struct.

```
type customError struct{}
```

Error implements the error interface.

```
func (c *customError) Error() string {
```

```go
	return "Find the bug."
}
```

fail returns nil values for both return types.

```go
func fail() ([]byte, *customError) {
	return nil, nil
}
```

```go
func main() {

	var err error
```

When we call fail, it returns the value of nil. However, we have the nil value of type *customError. We always want to use the error interface as the return value. The customError type is just an artifact, a value that we store inside. We cannot use the custom type directly. We must use the error interface, like so func fail() ([]byte, error)

```go
	if _, err = fail(); err != nil {
		log.Fatal("Why did this fail?")
	}

	log.Println("No Error")
}
```

Wrapping Errors

Error handling has to be part of our code and usually it is bound to logging. The main goal of logging is to debug. We only log things that are actionable. Only log the contexts that are allowed us to identify what is going on. Anything else ideally is noise and would be better suited up on the dashboard through metrics. For example, socket connection and disconnection, we can log these but these are not actionable because we don't necessarily lookup the log for that.

There is a package that is written by Dave Cheney called errors that let us simplify error handling and logging at the same time. Below is a demonstration on how to leverage the package to simplify our code. By reducing logging, we also reduce a large amount of pressure on the heap (garbage collection).

```go
import (
	"fmt"

	"github.com/pkg/errors"
)
```

AppError represents a custom error type.

```
type AppError struct {
	State int
}
```

AppError implements the error interface.

```
func (c *AppError) Error() string {
	return fmt.Sprintf("App Error, State: %d", c.State)
}
```

```
func main() {
```

Make the function call and validate the error. firstCall calls secondCall calls thirdCall then results in AppError. Start down the call stack, in thirdCall, where the error occurs. This is the root of the error. We return it up the call stack in our traditional error interface value.

Back to secondCall, we get the interface value and there is a concrete type stored inside the value. secondCall has to make a decision whether to handle the error and push up the call stack if it cannot handle. If secondCall decides to handle the error, it has the responsibility of logging it. If not, its responsibility is to move it up. However, if we are going to push it up the call stack, we cannot lose context. This is where the error package comes in. We create a new interface value that wraps this error, add a context around it and push it up. This maintains the call stack of where we are in the code.

Similarly, firstCall doesn't handle the error but wraps and pushes it up. In main, we are handling the call, which means the error stops here and we have to log it.

In order to properly handle this error, we need to know that the root cause of this error was. It is the original error that is not wrapped. Cause method will bubble up this error out of these wrapping and allow us to be able to use all the language mechanics we have.

We are not only able to access the State even though we've done this assertion back to concrete, we can log out the entire stack trace by using %+v for this call.

Use type as context to determine cause. We got our custom error type.

```
	if err := firstCall(10); err != nil {
		switch v := errors.Cause(err).(type) {
		case *AppError:
			fmt.Println("Custom App Error:", v.State)
```

Display the stack trace for the error.

```
		fmt.Println("\nStack
Trace\n********************************")
		fmt.Printf("%+v\n", err)
		fmt.Println("\nNo Trace\n********************************")
		fmt.Printf("%v\n", err)
	}
}
```

firstCall makes a call to a secondCall function and wraps any error.

```
func firstCall(i int) error {
	if err := secondCall(i); err != nil {
		return errors.Wrapf(err, "firstCall->secondCall(%d)", i)
	}
	return nil
}
```

secondCall makes a call to a thirdCall function and wraps any error.

```
func secondCall(i int) error {
	if err := thirdCall(); err != nil {
		return errors.Wrap(err, "secondCall->thirdCall()")
	}
	return nil
}
```

thirdCall function creates an error value we will validate.

```
func thirdCall() error {
	return &AppError{99}
}
```

```
Custom App Error: 99

Stack Trace
********************************
App Error, State: 99
secondCall->thirdCall()
main.secondCall
      /tmp/sandbox880380539/prog.go:74
main.firstCall
      /tmp/sandbox880380539/prog.go:65
main.main
      /tmp/sandbox880380539/prog.go:43
runtime.main
      /usr/local/go-faketime/src/runtime/proc.go:203
runtime.goexit
      /usr/local/go-faketime/src/runtime/asm_amd64.s:1373
firstCall->secondCall(10)
main.firstCall
      /tmp/sandbox880380539/prog.go:66
main.main
      /tmp/sandbox880380539/prog.go:43
runtime.main
      /usr/local/go-faketime/src/runtime/proc.go:203
runtime.goexit
      /usr/local/go-faketime/src/runtime/asm_amd64.s:1373

No Trace
********************************
firstCall->secondCall(10): secondCall->thirdCall(): App Error, State: 99
```

Concurrency

Mechanics

Goroutine

Go Scheduler Internals

Every time our Go's program starts up, it looks to see how many cores are available. Then it creates a logical processor.

The operating system scheduler is considered a preemptive scheduler. It runs down there in the kernel. Its job is to look at all the threads that are in runnable states and gives them the opportunity to run on some cores. These algorithms are fairly complex: waiting, bouncing threads, keeping memory of threads, caching,... The operating system is doing all of that for us. The algorithm is really smart when it comes to multicore processors. Go doesn't want to reinvent the wheel. It wants to sit on top of the operating system and leverage it.

The operating system is still responsible for operating system threads, scheduling operating system threads efficiently. If we have a 2 core machine and a thousands threads that the operating system has to schedule, that's a lot of work. A context switch on some operating system thread is expensive when the operating system has no clues of what that thread is doing.

It has to save all the possible states in order to be able to restore that to exactly the way it was. If there are fewer threads, each thread can get more time to be rescheduled. If there are more threads, each thread has less time over a long period of time.

"Less is more" is a really big concept here when we start to write concurrent software. We want to leverage the preemptive scheduler. So the Go's scheduler, the logical processor actually runs in user mode, the mode our application is running at. Because of that, we have to call the Go's scheduler a cooperating scheduler. What is brilliant here is the runtime that coordinates the operation. It still looks and feels as a preemptive scheduler up in user land. We will see how "less is more" concept gets to present itself and we get to do a lot more work with less. Our goal needs to be how much work we get done with the less number of threads.

Think about this in a simple way because processors are complex: hyperthreading, multiple threads per core, clock cycle. We can only execute one operating system thread at a time on any given core. If we only have 1 core, only 1 thread can be executed at a time. Anytime we have more threads in runnable states than we have cores, we are creating load, latency and we are getting more work done as we want. There needs to be this balance because not every thread is necessarily gonna be active at the same time. It all comes down to determining, understanding the workload for the software that we are writing.
Back to the first idea, when our Go program comes up, it has to see how many cores that are available. Let's say it found 1. It is going to create a logical processor P for that core.

Again, the operating system is scheduling things around operating system threads. What this processor P will get is an m, where m stands for machine. It represents an operating system thread that the operating system is going to schedule and allows our code to run.

The Linux scheduler has a run queue. Threads are placed in a run queue in certain cores or family of cores and those are constantly bound as threads are running. Go is gonna do the same thing. Go has its run queue as well. It has Global Run Queue (GRQ) and every P has a Local Run Queue (LRQ).

Goroutine

What is a Goroutine? It is a path of execution. Threads are paths of execution. That path of execution needs to be scheduled. In Go, every function or method can be created to be a Goroutine, and can become an independent path of execution that can be scheduled to run on some operating system threads against some cores.
When we start our Go program, the first thing runtime gonna do is create a Goroutine and put that in some main LRQ for some P. In our case, we only have 1 P here so we can imagine that Goroutine is attached to P.

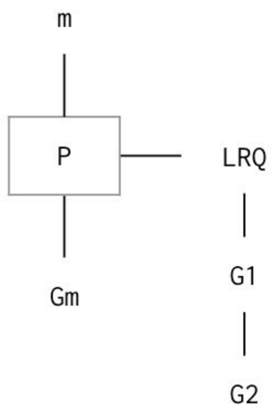
A Goroutine, just like thread, can be in one of three major states: sleeping, executing or in a runnable state asking to wait for some time to execute on the hardware. When the runtime creates a Goroutine, it is placed in P and multiplex on this thread. Remember that it's the operating system that takes the thread, scheduling it, placing it on some core and doing execution. So Go's scheduler is gonna take all the code related to that Goroutines path of execution, place it on a thread, tell the operating system that this thread is in runnable state and can we execute it. If the answer is yes, the operating system starts to execute on some cores there in the hardware.

As the main Goroutine runs, it might want to create more paths of execution, more Goroutines.

When that happens, those Goroutines might find themselves initially in the GRQ. These would be Goroutines that are in runnable state but haven't been assigned to some Ps yet. Eventually, they would end up in the LRQ where they're saying they would like some time to execute.

This queue does not necessarily follow First-In-First-Out protocol. We have to understand that everything here is non-deterministic, just like the operating system scheduler. We cannot predict what the scheduler is gonna do when all things are equal. It is gonna make sure there is a balance. Until we get into orchestration, till we learn how to coordinate these executions of these Goroutines, there is no predictability.

Here is the mental model of our example.

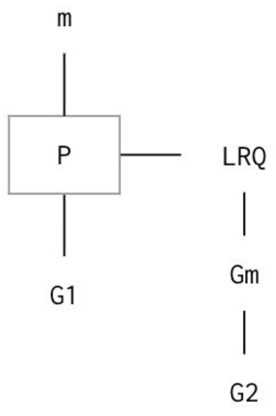


We have Gm executing on for this thread for this P, and we are creating 2 more Goroutines G1 and G2. Because this is a cooperating scheduler, that means that these Goroutines have to cooperate to be scheduled and to have a context switch on this operating system thread m.

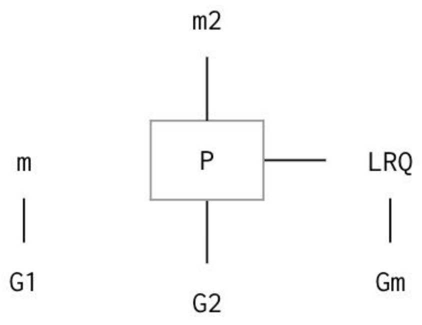
There are 4 major places in our code where the scheduler has the opportunity to make a scheduling decision.

- The keyword go that we are going to create Goroutines. That is also an opportunity for the scheduler to rebalance when it has multiple P.
- A system call. These system calls tend to happen all the time already.
- A channel operation because there is mutex (blocking call) that we will learn later.
- Garbage collection.

Back to the example, says the scheduler might decide Gm has enough time to run, it will put Gm back to the run queue and allow G1 to run on that m. We are now having a context switch.

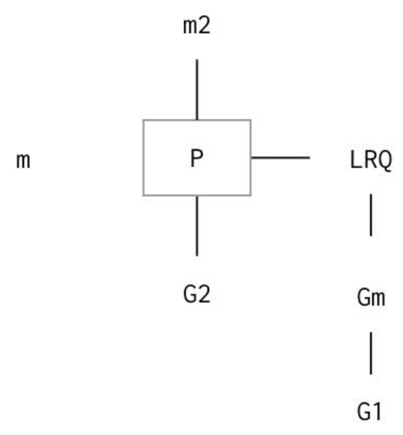


Let's say G1 decides to open up a file. Opening up a file can take microseconds or 10 milliseconds. We don't really know. If we allow this Goroutine to block this operating system thread while we open up that file, we are not getting more work done. In this scenario here, having a single P, we have a single threaded software application. All Goroutines only execute on the m attached to this P. What happens is this Goroutine is gonna block this m for a long time. We are basically stalled while we still have work that needs to get done. So the scheduler is not gonna allow that to happen, What actually happens is that the scheduler is gonna detach that m and G1. It is gonna bring a new m, say m2, then decide what G from the run queue should run next, say G2.



We now have 2 threads in a single threaded program. From our perspective, we are still single threading because the code that we are writing, the code associated with any G can only run against this P and this m. However, we don't know at any given time what m we are running on. M can get swapped out but we are still single threaded.

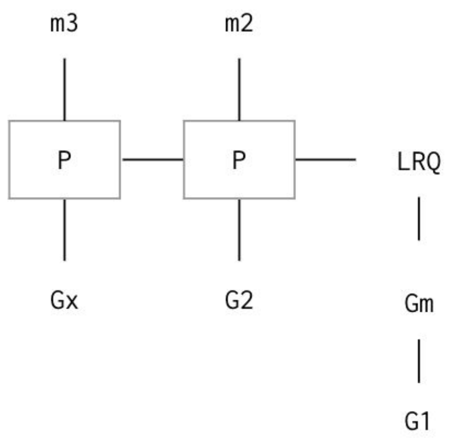
Eventually, G1 will come back, the file will be opened. The scheduler is gonna take this G1 and put it back to the run queue so we can be executed again on this P for some m (m2 in this case). m is placed on the side for later use. We are still maintaining these 2 threads. The whole process can happen again.



It is a really brilliant system of trying to leverage this thread to its fullest capability by doing more on 1 thread. Let's do so much on this thread we don't need another.

There is something called a Network poller. It is gonna do all the low level networking asynchronous networking stuff. Our G, if it is gonna do anything like that, it might be moved out to the Network poller and then brought back in. From our perspective, here is what we have to remember: The code that we are writing always runs on some P against some m. Depending on how many P we have, that's how many threads variables for us to run.

Concurrency is about managing a lot of things at once. This is what the scheduler is doing. It manages the execution of these 3 Goroutines against this one m for this P. Only 1 Goroutine can be executed at a single time.
If we want to do something in parallel, which means doing a lot of things at once, then we would need to create another P that has another m, say m3.



Both are scheduled by the operating system. So now we can have 2 Goroutines running at the same time in parallel.

Let's try another example.

We have multiple threaded software. The program launched 2 threads. Even if both threads end up on the same core, each wants to pass a message to each other. What has to happen from the operating system point of view?

We have to wait for thread 1 to get scheduled and placed on some cores - a context switch (CTX) has to happen here. While that's happening, thread is asleep so it's not running at all. From thread 1, we send a message over and want to wait to get a message back. In order to do that, there is another context switch needs to happen because we can put a different thread on that core. We are waiting for the operating system to schedule thread 2 so we are going to get another context switch, waking up and running, processing the message and sending the message back. On every single message that we are passing back and forth, thread is going from executable state to runnable state to asleep state. This is gonna cost a lot of context switches to occur.

Let's see what happens when we are using Goroutines, even on a single core.
G1 wants to send a message to G2 and we perform a context switch. However, the context here is the user's space switch. G1 can be taken out of the thread and G2 can be put on the thread. From the operating system point of view, this thread never goes to sleep. This thread is always executing and never needed to be context switched out. It is the Go's scheduler that keeps the Goroutines context switched.

If a P for some m here has no work to do, there is no G, the runtime scheduler will try to spin that m for a little bit to keep it hot on the core. Because if that thread goes cold, the operating system will pull it off the core and put something else on. So it just spins a little bit to see if there will be another G coming in to get some work done.

This is how the scheduler works underneath. We have a P, attached to thread m. The operating system will do the scheduling. We don't want any more than cores we have. We don't need any more operating system threads than cores we have. If we have more threads than cores we have, all we do is put load on the operating system. We allow the Go's scheduler to make decisions on our Goroutines, keeping the least number of threads we need and hot all the time if we have work. The Go's scheduler is gonna look and feel preemptive even though we are calling a cooperating scheduler.

However, let's not think about how the scheduler works. Think the following way makes it easier for future development. Every single G, every Goroutine that is in runnable state, is running at the same time.

One of the most important things that we must do from day one is to write software that can startup and shutdown cleanly. This is very very important.

```
package main

import (
	"fmt"
	"runtime"
	"sync"
)
```

init calls a function from the runtime package called GOMAXPROCS. This is also an environment variable, which is why it is all capitalized.

Prior to 1.5, when our Go program came up for the first time, it came up with just a single P, regardless of how many cores. The improvement that we made to the garbage collector and scheduler changed all that.

Allocate one logical processor for the scheduler to use.

```
func init() {
	runtime.GOMAXPROCS(1)
}

func main() {
```

wg is used to manage concurrency. wg is set to its zero value. This is one of the very special types in Go that are usable in its zero value state. It is also called Asynchronous Counting Semaphore. It has three methods: Add, Done and Wait. n number of Goroutines can call this method at the same time and it's all serialized.

- Add keeps a count of how many Goroutines out there.
- Done decrements that count because some Goroutines are about to be terminated.
- Wait holds the program until that count goes back down to zero.

```
	var wg sync.WaitGroup
```

We are creating 2 Goroutines. We rather call Add(1) and call it over and over again to increment by 1. If we don't know how many Goroutines that we are going to create, that is a smell.

```
	wg.Add(2)

	fmt.Println("Start Goroutines")
```

Create a Goroutine from the uppercase function using anonymous function. We have a function decoration here with no name and being called by the () in the end. We are declaring and calling this function right here, inside of main. The big thing here is the keyword go in front of func().
We don't execute this function right now in series here. Go schedules that function to be a G, say G1, and load in some LRQ for our P. This is our first G. Remember, we want to think that every G that is in runnable state is running at the same time. Even though we have a single P, even though we have a single thread, we don't care. We are having 2 Goroutines running at the same time: main and G1.

```
	go func() {
		lowercase()
		wg.Done()
	}()
```

Create a Goroutine from the lowercase function. We are doing it again. We are now having 3 Goroutines running at the same time.

```
	go func() {
		uppercase()
		wg.Done()
	}()
```

Wait for the Goroutines to finish. This is holding main from terminating because when the main terminates, our program terminates, regardless of what any other Goroutine is doing.

There is a golden rule here: We are not allowed to create a Goroutine unless we can tell when and how it terminates. Wait allows us to hold the program until the two other Goroutines report that they are done. It is gonna wait, count from 2 to 0. When it reaches 0, the scheduler will wake up the main Goroutine again and allow it to be terminated.

```
	fmt.Println("Waiting To Finish")
	wg.Wait()
```

```go
      fmt.Println("\nTerminating Program")
}
```

lowercase displays the set of lowercase letters three times. Display the alphabet three times.

```go
func lowercase() {
      for count := 0; count < 3; count++ {
            for r := 'a'; r <= 'z'; r++ {
                  fmt.Printf("%c ", r)
            }
      }
}
```

uppercase displays the set of uppercase letters three times. Display the alphabet three times.

```go
func uppercase() {
      for count := 0; count < 3; count++ {
            for r := 'A'; r <= 'Z'; r++ {
                  fmt.Printf("%c ", r)
            }
      }
}
```

```
Start Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z a b c d
e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n
o p q r s t u v w x y z
Terminating Program
```

Sequence

We call the uppercase after lowercase but Go's scheduler chooses to call the lowercase first. Remember we are running on a single thread so there is only one Goroutine is executed at a given time here. We can't see that we are running concurrently that the uppercase runs before the lowercase. Everything starts and completes cleanly.

What if we forget to hold Wait?

We would see no output of uppercase and lowercase. This is pretty much a data race. It's a race to see the program terminate before the scheduler stops it and schedules another Goroutine to run. By not waiting, these Goroutines never get a chance to execute at all.

What if we forget to call Done?

Deadlock!

This is a very special thing in Go. When the runtime determines that all the Goroutines are there and can no longer move forward, it's gonna panic.

Goroutine time slicing

How the Go's scheduler, even though it is a cooperating scheduler (not preemptive), it looks and feels preemptive because the runtime scheduler is making all the decisions for us. It is not coming for us.

The program below will show us a context switch and how we can predict when the context switch is going to happen. It is using the same pattern that we've seen in the last file. The only difference is the printPrime function.

```go
package main

import (
	"fmt"
	"runtime"
	"sync"
)
```

Allocate one logical processor for the scheduler to use.

```go
func init() {
	runtime.GOMAXPROCS(1)
}
```

wg is used to manage concurrency.

```go
func main() {
	var wg sync.WaitGroup
```

```
        wg.Add(2)

        fmt.Println("Create Goroutines")
```

Create the first goroutine and manage its lifecycle here.

```
        go func() {
                printPrime("A")
                wg.Done()
        }()
```

Create the second goroutine and manage its lifecycle here.

```
        go func() {
                printPrime("B")
                wg.Done()
        }()
```

Wait for the goroutines to finish.

```
        fmt.Println("Waiting To Finish")
        wg.Wait()

        fmt.Println("Terminating Program")
}
```

printPrime displays prime numbers for the first 5000 numbers. printPrime is not special. It just requires a little bit more time to complete. When we run the program, what we will see are context switches at some point for some particular prime number. We cannot predict when the context switch will happen. That's why we say the Go's scheduler looks and feels very preemptive even though it is a cooperating scheduler.

```
func printPrime(prefix string) {
next:
        for outer := 2; outer < 5000; outer++ {
                for inner := 2; inner < outer; inner++ {
                        if outer%inner == 0 {
                                continue next
                        }
                }

                fmt.Printf("%s:%d\n", prefix, outer)
```

```
    }

    fmt.Println("Completed", prefix)
}
```

```
Create Goroutines
Waiting To Finish
B:2
B:3
B:5
B:7
B:11
B:13
B:17
B:19
...
B:4999
Completed B
A:2
A:3
A:5
A:7
A:11
A:13
A:17
A:19
...
A:4999
Completed A
Terminating Program
```

Goroutines and parallelism

This programs show how Goroutines run in parallel. We are going to have 2 P with 2 m, and 2 Goroutines running in parallel on each m. This is still the same program that we are starting with. The only difference is that we are getting rid of the lowercase and uppercase function and putting their code directly inside Go's anonymous functions.

```go
package main

import (
    "fmt"
```

```
	"runtime"
	"sync"
)

func init() {
```

Allocate 2 logical processors for the scheduler to use.

```
	runtime.GOMAXPROCS(2)
}

func main() {
```

wg is used to wait for the program to finish. Add a count of two, one for each goroutine.

```
	var wg sync.WaitGroup
	wg.Add(2)

	fmt.Println("Start Goroutines")
```

Declare an anonymous function and create a goroutine. Display the alphabet three times.

```
	go func() {
		for count := 0; count < 3; count++ {
			for r := 'a'; r <= 'z'; r++ {
				fmt.Printf("%c ", r)
			}
		}
```

Tell main we are done.

```
		wg.Done()
	}()
```

Wait for the goroutines to finish.

```
	fmt.Println("Waiting To Finish")
	wg.Wait()

	fmt.Println("\nTerminating Program")
```

```
}
```

Looking at the output, we can see a mix of uppercase or lowercase characters.

```
Start Goroutines
Waiting To Finish
a b c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j
k l m n o p A B C D E F G H I J K L M N O P Q R S q r s t u v w x y z a
b c d e f g h i j k l m n o p q r s t u v w x y z T U V W X Y Z A B C D
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
O P Q R S T U V W X Y Z
Terminating Program
```

Data race

Race Detection

As soon as we add another Goroutine to our program, we add a huge amount of complexity. We can't always let the Goroutine run stateless. There has to be coordination. There are, in fact, 2 things that we can do with multi threaded software.

- We either have to synchronize access to share state like that WaitGroup is done with Add, Done and Wait.
- Or we have to coordinate these Goroutines to behave in a predictable or responsible manner.

Up until the use of channel, we have to use atomic function, mutex, to do both. The channel gives us a simple way to do orchestration. However, in many cases, using atomic function, mutex, and synchronizing access to shared state is the best way to go.

Atomic instructions are the fastest way to go because deep down in memory, Go is synchronizing 4-8 bytes at a time.

Mutexes are the next fastest. Channels are very slow because not only they are mutexes, there are all data structures and logic that go with them. Data races happen when we have multiple Goroutines trying to access the same memory location. For example, in the simplest case, we have an integer that is a counter. We have 2 Goroutines that want to read and write to that variable at the same time. If they are actually doing it at the same time, they are going to trash each other read and write. Therefore, this type of synchronizing access to the shared state has to be coordinated.

The problem with data races is that they always appear random. Sample program to show how to create race conditions in our programs. We don't want to do this.

```
package main

import (
	"fmt"
	"runtime"
	"sync"
)
```

counter is a variable incremented by all Goroutines.

```
var counter int

func main() {
```

Number of Goroutines to use.

```
	const grs = 2
```

wg is used to manage concurrency.

```
	var wg sync.WaitGroup
	wg.Add(grs)
```

Create two Goroutines.

They loop twice: perform a read to a local counter, increase by 1, write it back to the shared state. Every time we run the program, the output should be 4.
The data races that we have here is that: at any given time, both Goroutines could be reading and writing at the same time. However, we are very lucky in this case. What we are seeing is that each Goroutine is executing the 3 statements atomically completely by accident every time this code run.

If we put the line runtime.Gosched(), it will tell the scheduler to be part of the cooperation here and yield my time on that m. This will force the data race to happen. Once we read the value out of that shared state, we are gonna force the context switch. Then we come back, we are not getting 4 as frequent.

```
	for i := 0; i < grs; i++ {
		go func() {
			for count := 0; count < 2; count++ {
```

Capture the value of Counter.

```
                        value := counter
```

Yield the thread and be placed back in queue.

FOR TESTING ONLY! DO NOT USE IN PRODUCTION CODE!

```
                        runtime.Gosched()
```

Increment our local value of Counter.

```
                        value++
```

Store the value back into Counter.

```
                        counter = value
                  }
                  wg.Done()
            }()
      }
```

Wait for the goroutines to finish.

```
      wg.Wait()
      fmt.Println("Final Counter:", counter)
}
```

To identify race condition : go run -race <file_name>.

```
==================
WARNING: DATA RACE
Read at 0x000001228340 by goroutine 8:
  main.main.func1()

/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/data_race_1.go
:65 +0x47

Previous write at 0x000001228340 by goroutine 7:
  main.main.func1()

/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/data_race_1.go
```

```
:75 +0x68

Goroutine 8 (running) created at:
  main.main()

/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/data_race_1.go
:62 +0xab

Goroutine 7 (finished) created at:
  main.main()

/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/data_race_1.go
:62 +0xab
==================
Final Counter: 4
Found 1 data race(s)
exit status 66
```

Atomic Functions

```go
package main

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

counter is a variable incremented by all Goroutines. Notice that it's not just an int but int64. We are being very specific about the precision because the atomic function requires us to do so.

```go
var counter int64

func main() {
```

Number of Goroutines to use.

```go
	const grs = 2
```

wg is used to manage concurrency.

```
	var wg sync.WaitGroup
	wg.Add(grs)
```

Create two goroutines.

```
	for i := 0; i < grs; i++ {
		go func() {
			for count := 0; count < 2; count++ {
```

Safely add one to the counter. Add the atomic functions that we have taken an address as the first parameter and that is being synchronized, no matter how many Goroutines they are. If we call one of these functions on the same location, they will get serialized. This is the fastest way to serialization.

We can run this program all day long and still get 4 every time.

```
				atomic.AddInt64(&counter, 1)
```

This call is now irrelevant because by the time AddInt64 function completes, counter is incremented.

```
				runtime.Gosched()
			}

			wg.Done()
		}()
	}
```

Wait for the Goroutines to finish.

```
	wg.Wait()
```

Display the final value.

```
	fmt.Println("Final Counter:", counter)
}
```

```
Final Counter: 4
```

We don't always have the luxury of using 4-8 bytes of memory as shared data. This is where the mutex comes in. Mutex allows us to have an API like the WaitGroup (Add, Done and Wait) where any Goroutine can execute one at a time.

```
package main

import (
	"fmt"
	"sync"
)

var (
```

counter is a variable incremented by all Goroutines.

```
	counter int
```

mutex is used to define a critical section of code. Picture mutex as a room where all Goroutines have to go through. However, only one Goroutine can go at a time. The scheduler will decide who can get in and which one is next. We cannot determine what the scheduler is gonna do. Hopefully, it is gonna be fair. Just because one Goroutine got to the door before another, it doesn't mean that Goroutine will get to the end first. Nothing here is predictable.

The key here is, once a Goroutine is allowed in, it must report that it's out. All the Goroutines will ask for a lock and unlock when they leave for another one to get in. Two different functions can use the same mutex which means only one Goroutine can execute any of given functions at a time.

```
	mutex sync.Mutex
)
```

```
func main() {
```

Number of Goroutines to use.

```
	const grs = 2
```

wg is used to manage concurrency.

```
    var wg sync.WaitGroup
    wg.Add(grs)
```

Create two Goroutines.

```
    for i := 0; i < grs; i++ {
        go func() {
            for count := 0; count < 2; count++ {
```

Only allow one Goroutine through this critical section at a time. Creating these artificial curly brackets gives readability. We don't have to do this but it is highly recommended. The Lock and Unlock function must always be together in line of sight.

```
                mutex.Lock()
                {
```

Capture the value of counter.

```
                    value := counter
```

Increment our local value of counter.

```
                    value++
```

Store the value back into counter.

```
                    counter = value
                }
                mutex.Unlock()
```

Release the lock and allow any waiting Goroutine through.

```
            }

            wg.Done()
        }()
    }
```

Wait for the Goroutines to finish.

```
	wg.Wait()
	fmt.Printf("Final Counter: %d\n", counter)
}
```

```
Final Counter: 4
```

Read/Write Mutex

There are times when we have a shared resource where we want many Goroutines reading it.

Occasionally, one Goroutine can come in and make change to the resource. When that happens, everybody has to stop reading. It doesn't make sense to synchronize reads in this type of scenario because we are just adding latency to our software for no reason.

```
package main

import (
	"fmt"
	"math/rand"
	"sync"
	"sync/atomic"
	"time"
)
```

data is a slice that will be shared.

```
var (
	data []string
```

rwMutex is used to define a critical section of code. It is a little bit slower than Mutex but we are optimizing the correctness first so we don't care about that for now.

```
	rwMutex sync.RWMutex
```

Number of reads occurring at any given time. As soon as we see int64 here, we should start thinking about using atomic instruction.

```
	readCount int64
)
```

init is called prior to main.

```
func init() {
	rand.Seed(time.Now().UnixNano())
}

func main() {
```

wg is used to manage concurrency.

```
	var wg sync.WaitGroup
	wg.Add(1)
```

Create a writer Goroutine that performs 10 different writes.

```
	go func() {
		for i := 0; i < 10; i++ {
			time.Sleep(time.Duration(rand.Intn(100)) *
time.Millisecond)
			writer(i)
		}
		wg.Done()
	}()
```

Create eight reader Goroutines that runs forever.

```
	for i := 0; i < 8; i++ {
		go func(i int) {
			for {
				reader(i)
			}
		}(i)
	}
```

Wait for the write Goroutine to finish.

```
	wg.Wait()
	fmt.Println("Program Complete")
}
```

writer adds a new string to the slice in random intervals.

```
func writer(i int) {
```

Only allow one Goroutine to read/write to the slice at a time.

```
	rwMutex.Lock()
	{
```

Capture the current read count. Keep this safe though we can do it without this call. We want to make sure that no other Goroutines are reading. The value of rc should always be 0 when this code runs.

```
		rc := atomic.LoadInt64(&readCount)
```

Perform some work since we have a full lock.

```
		fmt.Printf("****> : Performing Write : RCount[%d]\n", rc)
		data = append(data, fmt.Sprintf("String: %d", i))
	}
	rwMutex.Unlock()
}
```

reader wakes up and iterates over the data slice.

```
func reader(id int) {
```

Any Goroutine can read when no write operation is taking place. RLock has the corresponding RUnlock.

```
	rwMutex.RLock()
	{
```

Increment the read count value by 1.

```
		rc := atomic.AddInt64(&readCount, 1)
```

Perform some read work and display values.

```
            time.Sleep(time.Duration(rand.Intn(10)) * time.Millisecond)
            fmt.Printf("%d : Performing Read : Length[%d] RCount[%d]\n",
id, len(data), rc)
```

Decrement the read count value by 1.

```
            atomic.AddInt64(&readCount, -1)
      }
      rwMutex.RUnlock()
}
```

The output will look similar to this.

```
0 : Performing Read : Length[0] RCount[1]
4 : Performing Read : Length[0] RCount[5]
5 : Performing Read : Length[0] RCount[6]
7 : Performing Read : Length[0] RCount[7]
3 : Performing Read : Length[0] RCount[4]
6 : Performing Read : Length[0] RCount[8]
4 : Performing Read : Length[0] RCount[8]
1 : Performing Read : Length[0] RCount[2]
2 : Performing Read : Length[0] RCount[3]
5 : Performing Read : Length[0] RCount[8]
0 : Performing Read : Length[0] RCount[8]
7 : Performing Read : Length[0] RCount[8]
7 : Performing Read : Length[0] RCount[8]
2 : Performing Read : Length[0] RCount[8]
...
1 : Performing Read : Length[10] RCount[8]
5 : Performing Read : Length[10] RCount[8]
3 : Performing Read : Length[10] RCount[8]
4 : Performing Read : Length[10] RCount[8]
6 : Performing Read : Length[10] RCount[8]
7 : Performing Read : Length[10] RCount[8]
2 : Performing Read : Length[10] RCount[8]
2 : Performing Read : Length[10] RCount[8]
```

Lesson:

The atomic functions and mutexes create latency in our software. Latency can be good when we have to coordinate orchestrating. However, if we can reduce latency using Read/Write Mutex, life is better.

If we are using mutex, make sure that we get in and out of mutex as fast as possible. Do not do anything extra. Sometimes just reading the shared state into a local variable is all we need to do. The less operation we can perform on the mutex, the better. We then reduce the latency to the bare minimum.

Channel

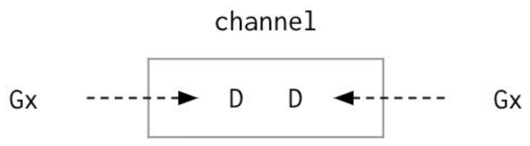
Language Mechanics

Channels are for orchestration. They allow us to have 2 Goroutines participate in some sort of workflow and give us the ability to orchestrate in a predictable way. The one thing that we really need to think about is not that a channel is a queue, even though it seems to be implemented like a queue, first in first out. We will have a difficult time if we think that way. What we want to think about instead is a channel as a way of signaling events to another Goroutine. What is nice here is that we can signal an event with data or without data.

If everything we do has signaling in mind, we are going to use channel in a proper way. Go has 2 types of channels: unbuffered and buffered. They both allow us to signal with data. The big difference is that, when we use an unbuffered channel, we are signaling and getting a guarantee the signal was received. We are not gonna be sure if that Goroutine is done whatever work we assign it to do but we do have the guarantee. The trade off for the guarantee that the signal was received is higher latency because we have to wait to make sure that the Goroutine on the other side of that unbuffered channel receives the data.

This is how the unbuffered channel is going to work. There is gonna be a Goroutine coming to the channel. The channel wants to signal with some piece of data. It is gonna put the data right there in the channel. However, the data is locked in and cannot move because the channel has to know if there is another Goroutine on the other side to receive it. Eventually a Goroutine comes and says that it wants to receive the data. Both of Goroutines are not putting their hands in the channel. The data now can be transferred.

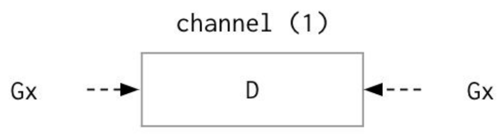
Here is the key to why that unbuffered channel gives us that guarantee: the receive happens first. When the receive happens, we know that the data transfer has occurred and we can walk away.



The unbuffered channel is a very powerful channel. We want to leverage that guarantee as much as possible. But again, the cost of the guarantee is higher latency because we have to wait for this.

The buffered channel is a bit different: we do not get the guarantee but we get to reduce the amount of latencies on any given send or receive.

Back to the previous example, we replace the unbuffered channel with a buffered channel. We are gonna have a buffered channel of just 1. It means there is a space in this channel for 1 piece of data that we are using the signal and we don't have to wait for the other side to get it. So now a Goroutine comes in, puts the data in and then moves away immediately. In other words, the send is happening before the receive. All the sending Goroutine knows is that it issues the signal, puts that data but has no clue when the signal is going to be received. Now hopefully a Goroutine comes in. It sees that there is data there, receive it and move on.



We use a buffered channel of 1 when dealing with these types of latency. We may need buffers that are larger but there are some design rules that we are gonna learn later on we use buffers that are greater than 1. But if we are in a situation where we can have these sends coming in and they could potentially be locked then we have to think again: if the channel of 1 is fast enough to reduce the latency that we are dealing with. Because what's gonna happen is the following: What we are hoping is, the buffered channel is always empty every time we perform a send.

Buffered channels are not for performance. What the buffered channel needs to be used for is continuity, to keep the wheel moving. One thing we have to understand is that everybody can write a piece of software that works when everything is going well. When things are going bad, it's where the architecture and engineer really come in. Our software doesn't enclose and it doesn't cost stress. We need to be responsible.

Back to the example, it's not important that we know exactly the signaling data was received but we do have to make sure that it was. The buffered channel of 1 gives us almost guarantee because what happens is: it performs a send, puts the data in there, turns around and when it comes back, it sees that the buffered is empty. Now we know that it was received. We don't know immediately at the time that we sent but by using a buffer of 1, we do know that it is empty when we come back.

Then it is okay to put another piece of data in there and hopefully when we come back again, it is gone. If it's not gone, we have a problem. There is a problem upstream. We cannot move forward until the channel is empty. This is something that we want to report immediately because we want to know why the data is still there. That's how we can build systems that are reliable.

We don't take more work at any given time. We identify upstream when there is a problem so we don't put more stress on our systems. We don't take more responsibilities for things that we shouldn't be.

Unbuffered channel: Signaling with data

```go
package main


import (
	"fmt"
	"time"
)


func main() {
	fmt.Printf("\n=> Basics of a send and receive\n")
	basicSendRecv()


	fmt.Printf("\n=> Close a channel to signal an event\n")
	signalClose()
}
```

basicSendRecv shows the basics of a send and receive. We are using make function to create a channel. We have no other way of creating a channel that is usable until we use make. Channel is also based on type, a type of data that we are gonna do the signaling. In this case, we use string. That channel is a reference type. ch is just a pointer variable to larger data structure underneath.

```go
func basicSendRecv() {
```

This is an unbuffered channel.

```go
	ch := make(chan string)

	go func() {
```

This is a send: a binary operation with the arrow pointing into the channel. We are signaling with a string "hello".

```
		ch <- "hello"
	}()
```

This is a receive: also an arrow but it is a unary operation where it is attached to the left hand side of the channel to show that it is coming out. We now have an unbuffered channel where the send and receive have to come together. We also know that the signal has been received because the receive happens first. Both are gonna block until both come together so the exchange can happen.

```
	fmt.Println(<-ch)
}
```

signalClose shows how to close a channel to signal an event.

```
func signalClose() {
```

We are making a channel using an empty struct. This is a signal without data.

```
	ch := make(chan struct{})
```

We are gonna launch a Goroutine to do some work. Suppose that it's gonna take 100 millisecond. Then, it wants to signal another Goroutine that it's done. It's gonna close the channel to report that it's done without the need of data. When we create a channel, buffered or unbuffered, that channel can be in 2 different states. All channels start out in open state so we can send and receive data. When we change the state to be closed, it cannot be opened. We also cannot close the channel twice because that is an integrity issue. We cannot signal twice without data twice.

```
	go func() {
		time.Sleep(100 * time.Millisecond)
		fmt.Println("signal event")
		close(ch)
	}()
```

When the channel is closed, the receive will immediately return. When we receive on a channel that is open, we cannot return until we receive the data signal. But if we receive on a channel that is closed, we are able to receive the signal without data. We know that event has occurred. Every receive on that channel will immediately return.

```
	<-ch

	fmt.Println("event received")
}
```

```
=> Basics of a send and receive
hello

=> Close a channel to signal an event
signal event
event received
```

Unbuffered channel: Double signal

```
package main


import (
	"fmt"
	"math/rand"
	"time"
)


func main() {
	fmt.Printf("\n=> Double signal\n")
	signalAck()


	fmt.Printf("\n=> Select and receive\n")
	selectRecv()


	fmt.Printf("\n=> Select and send\n")
	selectSend()


	fmt.Printf("\n=> Select and drop\n")
	selectDrop()
}
```

signalAck shows how to signal an event and wait for an acknowledgement it is done, It does not only want to guarantee that a signal is received but also want to know when that work is done. This is gonna like a double signal.

```
func signalAck() {
	ch := make(chan string)


	go func() {
		fmt.Println(<-ch)
		ch <- "ok done"
	}()
```

It blocks on the receive. This Goroutine can no longer move on until we receive a signal.

```
	ch <- "do this"
	fmt.Println(<-ch)
}
```

```
=> Double signal
do this
ok done
```

Unbuffered channel: select and receive

Select allows a Goroutine to work with multiple channels at a time, including send and receive. This can be great when creating an event loop but not good for serializing shared state. selectRecv shows how to use the select statement to wait for a specific amount of time to receive a value.

```
func selectRecv() {
	ch := make(chan string)
```

Wait for some amount of time and perform a send.

```
	go func() {
		time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
		ch <- "work"
	}()
```

Perform 2 different receives on 2 different channels: one above and one for time. time.After returns a channel that will send the current time after that duration. We want to receive the signal from the work sent but we are not willing to wait forever. We only wait 100 milliseconds then we will move on.

```
	select {
	case v := <-ch:
		fmt.Println(v)
	case <-time.After(100 * time.Millisecond):
		fmt.Println("timed out")
	}
```

However, there is a very common bug in this code. One of the biggest bugs we are going to have and potential memory is when we write code like this and we don't give the Goroutine an opportunity to terminate. We are using an unbuffered channel and this Goroutine at some point, its duration will finish and it will want to perform a send. But this is an unbuffered channel. This send cannot be completed unless there is a corresponding receive. What if this Goroutine times out and moves on? There is no more corresponding receive. Therefore, we will have a Goroutine leak, which means it will never be terminated.

The cleanest way to fix this bug is to use the buffered channel of 1. If this send happens, we don't necessarily have the guarantee. We don't need it. We just need to perform the signal then we can walk away. Therefore, either we get the signal on the other side or we walk away. Even if we walk away, this send can still be completed because there is room in the buffer for that send to happen.

```
=> Select and receive
work
```

Unbuffered channel: select and send

selectSend shows how to use the select statement to attempt a send on a channel for a specific amount of time.

```
func selectSend() {
	ch := make(chan string)

	go func() {
		time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
		fmt.Println(<-ch)
	}()


	select {
	case ch <- "work":
		fmt.Println("send work")
	case <-time.After(100 * time.Millisecond):
		fmt.Println("timed out")
	}
```

Similar to the above function, Goroutine leak will occur. Once again, a buffered channel of 1 will save us here.

```
=> Select and send
work
send work
```

Buffered channel: Select and drop

selectDrop shows how to use the select to walk away from a channel operation if it will immediately block.

This is a really important pattern. Imagine a situation where our service is flushed with work to do or work is gonna come. Something upstream is not functioning properly. We can't just back up the work. We have to throw it away so we can keep moving on.

A Denial-of-service attack is a great example. We get a bunch of requests coming to our server. If we try to handle every single request, we are gonna implode. We have to handle what we can and drop other requests.
Using this type of pattern (fanout), we are willing to drop some data. We can use buffer that are larger than 1. We have to measure what the buffer should be. It cannot be random.

```
func selectDrop() {
	ch := make(chan int, 5)


	go func() {
```

We are in the receive loop waiting for data to work on.

```
            for v := range ch {
                  fmt.Println("recv", v)
            }
      }()
```

This will send the work to the channel. If the buffer fills up, which means it blocks, the default case comes in and drops things.

```
      for i := 0; i < 20; i++ {
            select {
            case ch <- i:
                  fmt.Println("send work", i)
            default:
                  fmt.Println("drop", i)
            }
      }


      close(ch)
}
```

```
=> Select and drop
send work 0
send work 1
send work 2
send work 3
send work 4
send work 5
drop 6
drop 7
drop 8
drop 9
drop 10
drop 11
drop 12
drop 13
drop 14
drop 15
recv 0
recv 1
recv 2
recv 3
recv 4
recv 5
drop 16
send work 17
send work 18
send work 19
```

Unbuffered channel (Tennis match)

This program will put 2 Goroutines in a tennis match. We use an unbuffered channel because we need to guarantee that the ball is hit on both sides or missed.

```go
package main


import (
	"fmt"
	"math/rand"
	"sync"
	"time"
)


func init() {
	rand.Seed(time.Now().UnixNano())
}


func main() {
```

Create an unbuffered channel.

```go
	court := make(chan int)
```

wg is used to manage concurrency.

```go
	var wg sync.WaitGroup
	wg.Add(2)
```

Launch two players. Both are gonna start out in a receive mode. We are not really sure who is gonna get the ball first. Imagine the main Goroutine is the judge. It depends on the judge to choose.

```
	go func() {
		player("Hoanh", court)
		wg.Done()
	}()


	go func() {
		player("Andrew", court)
		wg.Done()
	}()
```

Start the set. The main Goroutine here is performing a send. Since both players are in receive mode, we cannot predict which one will go first.

```
	court <- 1
```

Wait for the game to finish.

```
	wg.Wait()
}
```

player simulates a person playing the game of tennis. We are asking for a channel value using value semantic.

```
func player(name string, court chan int) {
	for {
```

Wait for the ball to be hit back to us. Notice that this is another form of receive. Instead of getting just the value, we can get a flag indicating how the receive is returned. If the signal happens because of the data, ok will be true. If the signal happens without data, in other words, the channel is closed, ok will be false. In this case, we are gonna use that to determine who won.

```
		ball, ok := <-court
		if !ok {
```

If the channel was closed we won.

```
			fmt.Printf("Player %s Won\n", name)
			return
		}
```

Pick a random number and see if we miss the ball (or we lose). If we lose the game, we are gonna close the channel. It then causes the other player to know that he is receiving the signal but without data. The channel is closed so he won. They both return.

```
            n := rand.Intn(100)
            if n%13 == 0 {
                  fmt.Printf("Player %s Missed\n", name)
```

Close the channel to signal we lost.

```
                  close(court)
                  return
            }
```

Display and then increment the hit count by one.If the 2 cases above doesn't happen, we still have the ball. Increase the value of the ball by one and perform a send. We know that the other player is still in receive mode, therefore, the send and receive will eventually come together. Again, in an unbuffered channel, the receive happens first because it gives us the guarantee.

```
            fmt.Printf("Player %s Hit %d\n", name, ball)
            ball++
```

Hit the ball back to the opposing player.

```
            court <- ball
      }
}
```

```
Player Andrew Missed
Player Hoanh Won
```

Unbuffered channel (Replay race)

The program shows how to use an unbuffered channel to simulate a relay race between four Goroutines. Imagine we have 4 runners that are on the track. Only 1 can run at a time. We have the second runner on the track until the last one. The second one waits to be exchanged.

```go
package main


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

wg is used to wait for the program to finish.
var wg sync.WaitGroup

```go
func main() {
```

Create an unbuffered channel.

```go
	track := make(chan int)
```

Add a count of one for the last runner. We only add one because all we care about is the last runner in the race telling us that he is done.

```go
	wg.Add(1)
```

Create a first runner to his mark.

```go
	go Runner(track)
```

The main Goroutine starts the race (shoots the gun). At this moment, we know that on the other side, a Goroutine is performing a receive.

```go
	track <- 1
```

Wait for the race to finish.

```go
	wg.Wait()
}
```

Runner simulates a person running in the relay race. This Runner doesn't have a loop because it's gonna do everything from the beginning to end and then terminate. We are gonna keep adding Goroutines (Runners) in order to make this pattern work.

```go
func Runner(track chan int) {
```

The number of exchanges of the baton.

```go
	const maxExchanges = 4

	var exchange int
```

Wait to receive the baton with data.

```go
	baton := <-track
```

Start running around the track.

```go
	fmt.Printf("Runner %d Running With Baton\n", baton)
```

New runner to the line. Are we the last runner on the race? If not, we increment the data by 1 to keep track which runner we are on. We will create another Goroutine. It will go immediately into a receive. We are now having a second Groutine on the track, in the receive waiting for the baton. (1)

```go
	if baton < maxExchanges {
		exchange = baton + 1
		fmt.Printf("Runner %d To The Line\n", exchange)
		go Runner(track)
	}
```

Running around the track.

```go
	time.Sleep(100 * time.Millisecond)
```

Is the race over.

```go
	if baton == maxExchanges {
		fmt.Printf("Runner %d Finished, Race Over\n", baton)
		wg.Done()
		return
	}
```

Exchange the baton for the next runner.

```
      fmt.Printf("Runner %d Exchange With Runner %d\n", baton, exchange)
```

Since we are not the last runner, perform a send so (1) can receive it.

```
      track <- exchange
}
```

```
Runner 1 Running With Baton
Runner 2 To The Line
Runner 1 Exchange With Runner 2
Runner 2 Running With Baton
Runner 3 To The Line
Runner 2 Exchange With Runner 3
Runner 3 Running With Baton
Runner 4 To The Line
Runner 3 Exchange With Runner 4
Runner 4 Running With Baton
Runner 4 Finished, Race Over
```

Buffered channel: Fan Out

This is a classic use of a buffered channel that is greater than 1. It is called a Fan Out Pattern.

Idea: A Goroutine is doing its thing and decides to run a bunch of database operations. It is gonna create a bunch of Goroutines, say 10, to do that. Each Goroutine will perform 2 database operations. We end up having 20 database operations across 10 Goroutines. In other words, the original Goroutine will fan 10 Goroutines out, wait for them all to report back.

The buffered channel is fantastic here because we know ahead of time that there are 10 Goroutines performing 20 operations, so the size of the buffer is 20. There is no reason for any of these operation signals to block because we know that we have to receive this at the end of the day.

```
package main


import (
      "fmt"
      "log"
```

```
	"math/rand"
	"time"
)
```

result is what is sent back from each operation.

```
type result struct {
	id  int
	op  string
	err error
}

func init() {
	rand.Seed(time.Now().UnixNano())
}

func main() {
```

Set the number of Goroutines and insert operations.

```
	const routines = 10
	const inserts = routines * 2
```

Buffered channel to receive information about any possible insert.

```
	ch := make(chan result, inserts)
```

Number of responses we need to handle. Instead of using a WaitGroup, since this Goroutine can maintain its stack space, we are gonna use a local variable as our WaitGroup. We will decrement that as we go. Therefore, we set it to 20 inserts right out the box.

```
	waitInserts := inserts
```

Perform all the inserts. This is the fan out. We are gonna have 10 Goroutines. Each Goroutine performs 2 inserts. The result of the insert is used in a ch channel. Because this is a buffered channel, none of these send blocks.

```
	for i := 0; i < routines; i++ {
		go func(id int) {
			ch <- insertUser(id)
```

We don't need to wait to start the second insert thanks to the buffered channel. The first send will happen immediately.

```
			ch <- insertTrans(id)
		}(i)
	}
```

Process the insert results as they complete.

```
	for waitInserts > 0 {
```

Wait for a response from a Goroutine. This is a receive. We are receiving one result at a time and decrement the waitInserts until it gets down to 0.

```
		r := <-ch
```

Display the result.

```
		log.Printf("N: %d ID: %d OP: %s ERR: %v", waitInserts, r.id,
r.op, r.err)
```

Decrement the wait count and determine if we are done.

```
		waitInserts--
	}
	log.Println("Inserts Complete")
}
```

insertUser simulates a database operation.

```
func insertUser(id int) result {
	r := result{
		id: id,
		op: fmt.Sprintf("insert USERS value (%d)", id),
	}
```

Randomize if the insert fails or not.

```
	if rand.Intn(10) == 0 {
		r.err = fmt.Errorf("Unable to insert %d into USER table",
id)
```

```
	}
	return r
}
```

insertTrans simulates a database operation.

```
func insertTrans(id int) result {
	r := result{
		id: id,
		op: fmt.Sprintf("insert TRANS value (%d)", id),
	}
```

Randomize if the insert fails or not.

```
	if rand.Intn(10) == 0 {
		r.err = fmt.Errorf("Unable to insert %d into USER table",
id)
	}
	return r
}
```

```
2020/08/24 18:18:19 N: 20 ID: 0 OP: insert USERS value (0) ERR: <nil>
2020/08/24 18:18:19 N: 19 ID: 0 OP: insert TRANS value (0) ERR: <nil>
2020/08/24 18:18:19 N: 18 ID: 1 OP: insert USERS value (1) ERR: <nil>
2020/08/24 18:18:19 N: 17 ID: 1 OP: insert TRANS value (1) ERR: <nil>
2020/08/24 18:18:19 N: 16 ID: 2 OP: insert USERS value (2) ERR: <nil>
2020/08/24 18:18:19 N: 15 ID: 2 OP: insert TRANS value (2) ERR: Unable
to insert 2 into USER table
2020/08/24 18:18:19 N: 14 ID: 3 OP: insert USERS value (3) ERR: Unable
to insert 3 into USER table
2020/08/24 18:18:19 N: 13 ID: 3 OP: insert TRANS value (3) ERR: <nil>
2020/08/24 18:18:19 N: 12 ID: 4 OP: insert USERS value (4) ERR: <nil>
2020/08/24 18:18:19 N: 11 ID: 4 OP: insert TRANS value (4) ERR: <nil>
2020/08/24 18:18:19 N: 10 ID: 5 OP: insert USERS value (5) ERR: <nil>
2020/08/24 18:18:19 N: 9 ID: 5 OP: insert TRANS value (5) ERR: <nil>
2020/08/24 18:18:19 N: 8 ID: 6 OP: insert USERS value (6) ERR: <nil>
2020/08/24 18:18:19 N: 7 ID: 6 OP: insert TRANS value (6) ERR: <nil>
2020/08/24 18:18:19 N: 6 ID: 7 OP: insert USERS value (7) ERR: <nil>
2020/08/24 18:18:19 N: 5 ID: 7 OP: insert TRANS value (7) ERR: Unable to
insert 7 into USER table
2020/08/24 18:18:19 N: 4 ID: 8 OP: insert USERS value (8) ERR: <nil>
2020/08/24 18:18:19 N: 3 ID: 8 OP: insert TRANS value (8) ERR: <nil>
2020/08/24 18:18:19 N: 2 ID: 9 OP: insert USERS value (9) ERR: <nil>
```

```
2020/08/24 18:18:19 N: 1 ID: 9 OP: insert TRANS value (9) ERR: <nil>
2020/08/24 18:18:19 Inserts Complete
```

Select

This sample program demonstrates how to use a channel to monitor the amount of time the program is running and terminate the program if it runs too long.

```go
package main

import (
	"errors"
	"log"
	"os"
	"os/signal"
	"time"
)
```

Give the program 3 seconds to complete the work.

```go
const timeoutSeconds = 3 * time.Second
```

There are 4 channels that we are gonna use: 3 unbuffered and 1 buffered of 1.

```go
var (
```

sigChan receives operating signals. This will allow us to send a Ctrl-C to shut down our program cleanly.

```go
	sigChan = make(chan os.Signal, 1)
```

timeout limits the amount of time the program has. We really don't want to receive on this channel because if we do, that means something bad happens, we are timing out and we need to kill the program.

```go
	timeout = time.After(timeoutSeconds)
```

complete is used to report processing is done. This is the channel we want to receive on. When the Goroutine finishes the job, it will signal to us on this complete channel and tell us any error that occurred.

```
	complete = make(chan error)
```

shutdown provides system wide notification.

```
	shutdown = make(chan struct{})
)

func main() {
	log.Println("Starting Process")
```

We want to receive all interrupt based signals. We are using a Notify function from the signal package, passing sigChan telling the channel to look for anything that is os.Interrupt related and sending us a data signal on this channel. One important thing about this API is that it won't wait for us to be ready to receive the signal. If we are not there, it will drop it on the floor. That's why we are using a buffered channel of 1. This way we guarantee to get at least 1 signal. When we are ready to act on that signal, we can come over there and do it.

```
	signal.Notify(sigChan, os.Interrupt)
```

Launch the process.

```
	log.Println("Launching Processors")
```

This Goroutine will do the processing job, for example image processing.

```
	go processor(complete)
```

The main Goroutine here is in this event loop and it's gonna loop forever until the program is terminated. There are 3 cases in select, meaning that there are 3 channels we are trying to receive on at the same time: sigChan, timeout, and complete.

```
ControlLoop:
	for {
		select {
		case <-sigChan:
```

Interrupt event signaled by the operating system.

```
			log.Println("OS INTERRUPT")
```

Close the channel to signal to the processor it needs to shutdown.

```
                close(shutdown)
```

Set the channel to nil so we no longer process any more of these events.
If we try to send on a closed channel, we are gonna panic. If we receive on a closed channel, that's gonna immediately return a signal without data. If we receive on a nil channel, we are blocked forever. Similar with send. Why do we want to do that?

We don't want users to hold down Ctrl C or hit Ctrl C multiple times. If they do that and we process the signal, we have to call close multiple times. When we call close on a channel that is already closed, the code will panic. Therefore, we cannot have that.

```
                sigChan = nil
            case <-timeout:
```

We have taken too much time. Kill the app hard.

```
                log.Println("Timeout - Killing Program")
```

os.Exit will terminate the program immediately.

```
                os.Exit(1)
            case err := <-complete:
```

Everything completed within the time given.

```
                log.Printf("Task Completed: Error[%s]", err)
```

We are using a label break here. We put one at the top of the for loop so the case has a break and the for has a break.

```
                break ControlLoop
            }
        }

        log.Println("Process Ended")
}
```

processor provides the main program logic for the program. There is something interesting in the parameter. We put the arrow on the right hand side of the chan keyword. It means this channel is a send-only channel. If we try to receive on this channel, the compiler will give us an error.

```
func processor(complete chan<- error) {
	log.Println("Processor - Starting")
```

Variable to store any error that occurs. Passed into the defer function via closures.

```
	var err error
```

Defer the send on the channel so it happens regardless of how this function terminates. This is an anonymous function call like we saw with Goroutine. However, we are using the keyword defer here.
We want to execute this function but after the processor function returns. This gives us a guarantee that we can have certain things happen before control go back to the caller.

Also, defer is the only way to stop a panic. If something bad happens, say the image library is blowing up, that can cause a panic situation throughout the code. In this case, we want to recover from that panic, stop it and then control the shutdown.

```
	defer func() {
```

Capture any potential panic.

```
		if r := recover(); r != nil {
			log.Println("Processor - Panic", r)
		}
```

Signal the Goroutine we have shut down.

```
		complete <- err
	}()
```

Perform the work.

```
	err = doWork()
```

```go
	log.Println("Processor - Completed")
}
```

doWork simulates task work. Between every single call, we call checkShutdown. After completing every task, we are asked: Have we been asked to shutdown? The only way we know is that the shutdown channel is closed. The only way to know if the shutdown channel is closed is to try to receive. If we try to receive on a channel that is not closed, it's gonna block. However, the default case is gonna save us here.

```go
func doWork() error {
	log.Println("Processor - Task 1")
	time.Sleep(2 * time.Second)

	if checkShutdown() {
		return errors.New("Early Shutdown")
	}

	log.Println("Processor - Task 2")
	time.Sleep(1 * time.Second)

	if checkShutdown() {
		return errors.New("Early Shutdown")
	}

	log.Println("Processor - Task 3")
	time.Sleep(1 * time.Second)

	return nil
}
```

checkShutdown checks the shutdown flag to determine if we have been asked to interrupt processing.

```go
func checkShutdown() bool {
	select {
	case <-shutdown:
```

We have been asked to shut down cleanly.

```go
		log.Println("checkShutdown - Shutdown Early")
		return true

```

```
    default:
```

If the shutdown channel was not closed, presume with normal processing.

```
            return false
    }
}
```

When we let the program run, since we configure the timeout to be 3 seconds, it will then timeout and be terminated.

```
2020/08/24 18:31:27 Starting Process
2020/08/24 18:31:27 Launching Processors
2020/08/24 18:31:27 Processor - Starting
2020/08/24 18:31:27 Processor - Task 1
2020/08/24 18:31:29 Processor - Task 2
2020/08/24 18:31:30 Timeout - Killing Program
exit status 1
```

When we hit Ctrl C while the program is running, we will see the OS INTERRUPT and the program is being shutdown early.

```
2020/08/24 18:21:02 Starting Process
2020/08/24 18:21:02 Launching Processors
2020/08/24 18:21:02 Processor - Starting
2020/08/24 18:21:02 Processor - Task 1
^C2020/08/24 18:21:03 OS INTERRUPT
2020/08/24 18:21:04 checkShutdown - Shutdown Early
2020/08/24 18:21:04 Processor - Completed
2020/08/24 18:21:04 Task Completed: Error[Early Shutdown]
2020/08/24 18:21:04 Process Ended
```

When we send a signal quit by hitting Ctrt \, we will get a full stack trace of all the Goroutines.

```
2020/08/24 18:31:44 Starting Process
2020/08/24 18:31:44 Launching Processors
2020/08/24 18:31:44 Processor - Starting
2020/08/24 18:31:44 Processor - Task 1
2020/08/24 18:31:46 Processor - Task 2
^\SIGQUIT: quit
PC=0x7fff70c3e882 m=0 sigcode=0
```

```
goroutine 0 [idle]:
runtime.pthread_cond_wait(0x12201e8, 0x12201a8, 0x7ffe00000000)
        /usr/local/go/src/runtime/sys_darwin.go:378 +0x39
runtime.semasleep(0xffffffffffffffff, 0x7ffeefbff678)
        /usr/local/go/src/runtime/os_darwin.go:63 +0x85
runtime.notesleep(0x121ffa8)
        /usr/local/go/src/runtime/lock_sema.go:173 +0xe0
runtime.stoplockedm()
        /usr/local/go/src/runtime/proc.go:2068 +0x88
runtime.schedule()
        /usr/local/go/src/runtime/proc.go:2469 +0x485
runtime.park_m(0xc00007cd80)
        /usr/local/go/src/runtime/proc.go:2610 +0x9d
runtime.mcall(0x108ca06)
        /usr/local/go/src/runtime/asm_amd64.s:318 +0x5b

goroutine 1 [select]:
main.main()

/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/channel_6.go:6
7 +0x278

goroutine 19 [syscall]:
os/signal.signal_recv(0x108ebb1)
        /usr/local/go/src/runtime/sigqueue.go:144 +0x96
os/signal.loop()
        /usr/local/go/src/os/signal/signal_unix.go:23 +0x30
created by os/signal.init.0
        /usr/local/go/src/os/signal/signal_unix.go:29 +0x4f

goroutine 5 [sleep]:
runtime.goparkunlock(...)
        /usr/local/go/src/runtime/proc.go:310
time.Sleep(0x3b9aca00)
        /usr/local/go/src/runtime/time.go:105 +0x157
main.doWork(0xc000054768, 0x1)

/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/channel_6.go:1
57 +0x14a
main.processor(0xc000096060)

/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/channel_6.go:1
38 +0xbc
created by main.main
```

```
/Users/hoanhan/work/hoanhan101/ultimate-go/go/concurrency/channel_6.go:5
8 +0x160

rax    0x104
rbx    0x2
rcx    0x7ffeefbff498
rdx    0x200
rdi    0x12201e8
rsi    0x20100000300
rbp    0x7ffeefbff530
rsp    0x7ffeefbff498
r8     0x0
r9     0xa0
r10    0x0
r11    0x202
r12    0x12201e8
r13    0x16
r14    0x20100000300
r15    0x10863dc0
rip    0x7fff70c3e882
rflags 0x203
cs     0x7
fs     0x0
gs     0x0
exit status 2
```

Patterns

Context

Store and retrieve values from a context

Context package is the answer to cancellation and deadline in Go.

```go
package main

import (
	"context"
	"fmt"
)
```

user is the type of value to store in the context.

```
type user struct {
	name string
}
```

userKey is the type of value to use for the key. The key is type specific and only values of the same type will match. When we store a value inside a context, what getting stored is not just a value but also a type associated with the storage. We can only pull a value out of that context if we know the type of value that we are looking for. The idea of this userKey type becomes really important when we want to store a value inside the context.

```
type userKey int

func main() {
```

Create a value of type user.

```
	u := user{
		name: "Hoanh",
	}
```

Declare a key with the value of zero of type userKey.

```
	const uk userKey = 0
```

Store the pointer to the user value inside the context with a value of zero of type userKey. We are using context.WithValue because a new context value and we want to initialize that with data to begin with. Anytime we work a context, the context has to have a parent context. This is where the Background function comes in. We are gonna store the key uk to its value (which is 0 in this case), and address of user.

```
	ctx := context.WithValue(context.Background(), uk, &u)
```

Retrieve that user pointer back by user the same key type value. Value allows us to pass the key of the corrected type (in our case is uk of userKey type) and returns an empty interface. Because we are working with an interface, we have to perform a type assertion to pull the value that we store in there out the interface so we can work with the concrete again.

```
	if u, ok := ctx.Value(uk).(*user); ok {
		fmt.Println("User", u.nam
	}
```

Attempt to retrieve the value again using the same value but of a different type. Even though the key value is 0, if we just pass 0 into this function call, we are not gonna get back that address to the user because 0 is based on integer type, not our userKey type.

It's important that when we store the value inside the context to not use the built-in type.

Declare our own key type. That way, only us and who understand that type can pull that out. Because what if multiple partial programs want to use that value of 0, we are all being tripped up on each other. That type extends an extra level of protection on being able to store and retrieve value out of context.
If we are using this, we want to raise a flag because we have to ask twice why we want to do that instead of passing down the call stack. Because if we can pass it down the call stack, it would be much better for readability and maintainability for our legacy code in the future.

```
	if _, ok := ctx.Value(0).(*user); !ok {
		fmt.Println("User Not Found")
	}
}
```

```
User Hoanh
User Not Found
```

WithCancel

Different ways we can do cancellation, timeout in Go.

```
package main

import (
	"context"
	"fmt"
	"time"
)

func main() {
```

Create a context that is cancellable only manually. The cancel function must be called regardless of the outcome. WithCancel allows us to create a context and provides us a cancel function that can be called in order to report a signal, a signal without data, that we want whatever that Goroutine is doing to stop right away. Again, we are using Background as our parents context.

```
    ctx, cancel := context.WithCancel(context.Background())
```

The cancel function must be called regardless of the outcome. The Goroutine that creates the context must always call cancel. These are things that have to be cleaned up. It's the responsibility that the Goroutine creates the context the first time to make sure to call cancel after everything is done. The use of the defer keyword is perfect here for this use case.

```
    defer cancel()
```

We launch a Goroutine to do some work for us. It is gonna sleep for 50 milliseconds and then call cancel. It is reporting that it wants to signal a cancel without data.

```
    go func() {
```

Simulate work. If we run the program using 50 ms, we expect the work to be complete. But if it is 150ms, then we move on.

```
        time.Sleep(50 * time.Millisecond)
```

Report the work is done.

```
        cancel()
    }()
```

The original Goroutine that creates that channel is in its select case. It is gonna receive after time.After. We are gonna wait 100 milliseconds for something to happen. We are also waiting on context.Done. We are just gonna sit here, and if we are told to Done, we know that work up there is complete.

```
    select {
    case <-time.After(100 * time.Millisecond):
        fmt.Println("moving on")
    case <-ctx.Done():
        fmt.Println("work complete")
    }
```

```
}
```

```
work complete
```

WithDeadline

```
package main

import (
	"context"
	"fmt"
	"time"
)

type data struct {
	UserID string
}

func main() {
```

Set a deadline.

```
	deadline := time.Now().Add(150 * time.Millisecond)
```

Create a context that is both manually cancellable and will signal a cancel at the specified date/time. We use Background as our parents context and set out deadline time.

```
	ctx, cancel := context.WithDeadline(context.Background(),
deadline)
	defer cancel()
```

Create a channel to received a signal that work is done.

```
	ch := make(chan data, 1)
```

Ask a Goroutine to do some work for us.

```
	go func() {
```

Simulate work.

```
            time.Sleep(200 * time.Millisecond)
```

Report the work is done.

```
            ch <- data{"123"}
      }()
```

Wait for the work to finish. If it takes too long move on.

```
      select {
      case d := <-ch:
            fmt.Println("work complete", d)

      case <-ctx.Done():
            fmt.Println("work cancelled")
      }
}
```

```
work cancelled
```

WithTimeout

```
package main


import (
      "context"
      "fmt"
      "time"
)

type data struct {
      UserID string
}

func main() {
```

Set a duration.

```
    duration := 150 * time.Millisecond
```

Create a context that is both manually cancellable and will signal a cancel at the specified duration.

```
    ctx, cancel := context.WithTimeout(context.Background(), duration)
    defer cancel()
```

Create a channel to received a signal that work is done.

```
    ch := make(chan data, 1)
```

Ask the goroutine to do some work for us.

```
    go func() {
```

Simulate work.

```
        time.Sleep(50 * time.Millisecond)
```

Report the work is done.

```
        ch <- data{"123"}
    }()
```

Wait for the work to finish. If it takes too long, move on.

```
    select {
    case d := <-ch:
        fmt.Println("work complete", d)
    case <-ctx.Done():
        fmt.Println("work cancelled")
    }
}
```

```
work complete {123}
```

Sample program that implements a web request with a context that is used to timeout the request if it takes too long.

```go
package main

import (
	"context"
	"io"
	"log"
	"net"
	"net/http"
	"os"
	"time"
)

func main() {
```

Create a new request.

```go
	req, err := http.NewRequest("GET",
"https://www.ardanlabs.com/blog/post/index.xml", nil)
	if err != nil {
		log.Println(err)
		return
	}
```

Create a context with a timeout of 50 milliseconds.

```go
	ctx, cancel := context.WithTimeout(req.Context(),
50*time.Millisecond)
	defer cancel()
```

Declare a new transport and client for the call.

```go
	tr := http.Transport{
		Proxy: http.ProxyFromEnvironment,
		DialContext: (&net.Dialer{
			Timeout:   30 * time.Second,
			KeepAlive: 30 * time.Second,
			DualStack: true,
        }).DialContext,
        MaxIdleConns:          100,
        IdleConnTimeout:       90 * time.Second,
        TLSHandshakeTimeout:   10 * time.Second,
        ExpectContinueTimeout: 1 * time.Second,
    }
    client := http.Client{
        Transport: &tr,
    }
```

Make the web call in a separate Goroutine so it can be cancelled.

```
    ch := make(chan error, 1)
    go func() {
        log.Println("Starting Request")
```

Make the web call and return any error.client.Do is going out and trying to hit the request URL. It's probably blocked right now because it will need to wait for the entire document to come back.

```
        resp, err := client.Do(req)
```

If the error occurs, we perform a send on the channel to report that we are done. We are going to use this channel at some point to report back what is happening.

```
        if err != nil {
            ch <- err
            return
        }
```

If it doesn't fail, we close the response body on the return.

```
        defer resp.Body.Close()
```

Write the response to stdout.

```
        io.Copy(os.Stdout, resp.Body)
```

Then send back the nil instead of error.

```
        ch <- nil
```

```
	}()
```

Wait the request or timeout. We perform a receive on ctx.Done saying that we want to wait 50 ms for that whole process above to happen. If it doesn't, we signal back to that Goroutine to cancel the sending request. We don't have to just walk away and let that eat up resources and finish because we are not gonna need it. We are able to call CancelRequest and underneath, we are able to kill that connection.

```
	select {
	case <-ctx.Done():
		log.Println("timeout, cancel work...")
```

Cancel the request and wait for it to complete.

```
		tr.CancelRequest(req)
		log.Println(<-ch)
	case err := <-ch:
		if err != nil {
			log.Println(err)
		}
	}
}
```

```
2020/08/24 18:37:18 Starting Request
2020/08/24 18:37:18 timeout, cancel work...
2020/08/24 18:37:18 Get https://www.ardanlabs.com/blog/post/index.xml:
net/http: request canceled while waiting for connection
```

Testing and Profiling

Testing

Basic Unit Test

All of our tests must have the format <filename>_test.go. Otherwise, the testing tool is not gonna find the tests. Test files are not compiled into our final binary. Test files should be in the same package as your code. We might also want to have a folder called test for more than unit test, say integration test. The package name can be the name only or name_test.

If we go with name_test, it allows us to make sure these tests work with the package. The only reason that we don't want to do this is when we have a function or method that is unexported.

However, if we don't use name_test, it will raise a red flag because if we cannot test the exported API to get the coverage for the unexported API then we know we are missing something. Therefore, 9/10 this is what we want.

```go
package main

import (
	"net/http"
	"testing" // This is Go testing package.
)
```

These constants give us checkboxes for visualization.

```go
const (
	succeed = "\u2713"
	failed  = "\u2717"
)
```

TestBasic validates the http.Get function can download content. Every test will be associated with the test function. It starts with the word Test and the first word after Test must be capitalized. It uses a testing.T pointer as its parameter. When writing test, we want to focus on usability first. We must write it the same way as we would write it in production. We also want the verbosity of tests so we are 3 different methods of t: Log or Logf, Fatal or Fatalf, Error or Error f. That is the core APIs for testing.

Log: Write documentation out into the log output.

Error: Write documentation and also say that this test is failed but we are continuing moving forward to execute code in this test.

Fatal: Similarly, document that this test is failed but we are done. We move on to the next test function.

Given, When, Should format.

Given: Why are we writing this test?

When: What data are we using for this test?

Should: When are we expected to see it happen or not happen?

We are also using the artificial block between a long Log function. They help with readability.

```go
func TestBasic(t *testing.T) {
	url := "https://www.google.com/"
	statusCode := 200

	t.Log("Given the need to test downloading content.")
	{
		t.Logf("\tTest 0:\tWhen checking %q for status code %d",
url, statusCode)
		{
			resp, err := http.Get(url)
			if err != nil {
				t.Fatalf("\t%s\tShould be able to make the Get
call : %v", failed, err)
			}
			t.Logf("\t%s\tShould be able to make the Get call.",
succeed)

			defer resp.Body.Close()

			if resp.StatusCode == statusCode {
				t.Logf("\t%s\tShould receive a %d status code.",
succeed, statusCode)
			} else {
				t.Errorf("\t%s\tShould receive a %d status code :
%d", failed, statusCode, resp.StatusCode)
			}
		}
	}
}
```

We can just say "go test" and the testing tool will find that function. We can also say "go test -v" for verbosity, we will get a full output of the logging. Suppose that we have a lot of test functions, "go test -run TestBasic" will only run the TestBasic function.

```
=== RUN   TestBasic
--- PASS: TestBasic (0.24s)
    basic_test.go:58: Given the need to test downloading content.
    basic_test.go:60:    Test 0: When checking "https://www.google.com/"
for status code 200
    basic_test.go:66:    ✓        Should be able to make the Get call.
    basic_test.go:71:   ✓       Should receive a 200 status code.
PASS
ok      command-line-arguments  0.316s
```

Table Test

Set up a data structure of input to expected output. This way we don't need a separate function for each one of these. We just have 1 test function. As we go along, we just add more to the table.

```go
package main

import (
	"net/http"
	"testing"
)
```

TestTable validates the http Get function can download content and handles different status conditions properly.

```go
func TestTable(t *testing.T) {
```

This table is a slice of anonymous struct type. It is the URL we are gonna call and statusCode is what we expect.

```go
	tests := []struct {
		url        string
		statusCode int
	}{
		{"https://www.google.com/", http.StatusOK},
		{"http://rss.cnn.com/rss/cnn_topstorie.rss",
http.StatusNotFound},
	}

	t.Log("Given the need to test downloading different content.")
	{
		for i, tt := range tests {
			t.Logf("\tTest: %d\tWhen checking %q for status code
%d", i, tt.url, tt.statusCode)
			{
				resp, err := http.Get(tt.url)
				if err != nil {
```

```
                              t.Fatalf("\t%s\tShould be able to make the
Get call : %v", failed, err)
                        }
                        t.Logf("\t%s\tShould be able to make the Get
call.", succeed)

                        defer resp.Body.Close()

                        if resp.StatusCode == tt.statusCode {
                              t.Logf("\t%s\tShould receive a %d status
code.", succeed, tt.statusCode)
                        } else {
                              t.Errorf("\t%s\tShould receive a %d status
code : %v", failed, tt.statusCode, resp.StatusCode)
                        }
                  }
            }
      }
}
```

```
=== RUN   TestTable
--- PASS: TestTable (0.31s)
    table_test.go:35: Given the need to test downloading different
content.
    table_test.go:38:   Test: 0 When checking "https://www.google.com/"
for status code 200
    table_test.go:44:   ✓       Should be able to make the Get call.
    table_test.go:49:   ✓       Should receive a 200 status code.
    table_test.go:38:   Test: 1 When checking
"http://rss.cnn.com/rss/cnn_topstorie.rss" for status code 404
    table_test.go:44:   ✓       Should be able to make the Get call.
    table_test.go:49:   ✓       Should receive a 404 status code.
PASS
ok      command-line-arguments  0.472s
```

Sub Test

Sub test helps us streamline our test functions, filters out command-line level big tests into smaller sub tests.

```
package main

import (
	"net/http"
	"testing"
)
```

TestSub validates the http Get function can download content and handles different status conditions properly.

```
func TestSub(t *testing.T) {
	tests := []struct {
		name       string
		url        string
		statusCode int
	}{
		{"statusok", "https://www.google.com/", http.StatusOK},
		{"statusnotfound",
"http://rss.cnn.com/rss/cnn_topstorie.rss", http.StatusNotFound},
	}

	t.Log("Given the need to test downloading different content.")
	{
```

Range over our table but this time, create an anonymous function that takes a testing T parameter. This is a test function inside a test function. What's nice about it is that we are gonna have a new function for each set of data that we have in our table. Therefore, we will end up with 2 different functions here.

```
		for i, tt := range tests {
			tf := func(t *testing.T) {
				t.Logf("\tTest: %d\tWhen checking %q for status
code %d", i, tt.url, tt.statusCode)
				{
					resp, err := http.Get(tt.url)
					if err != nil {
						t.Fatalf("\t%s\tShould be able to
make the Get call : %v", failed, err)
					}
					t.Logf("\t%s\tShould be able to make the
Get call.", succeed)

					defer resp.Body.Close()

					if resp.StatusCode == tt.statusCode {
						t.Logf("\t%s\tShould receive a %d
status code.", succeed, tt.statusCode)
```

```
                        } else {
                              t.Errorf("\t%s\tShould receive a %d
status code : %v", failed, tt.statusCode, resp.StatusCode)
                        }
                  }
            }
```

Once we declare this function, we tell the testing tool to register it as a sub test under the test name.

```
                  t.Run(tt.name, tf)
            }
      }
}
```

TestParallelize validates the http Get function can download content and handles different status conditions properly but runs the tests in parallel.

```
func TestParallelize(t *testing.T) {
      tests := []struct {
            name       string
            url        string
            statusCode int
      }{
            {"statusok", "https://www.goinggo.net/post/index.xml",
http.StatusOK},
            {"statusnotfound",
"http://rss.cnn.com/rss/cnn_topstorie.rss", http.StatusNotFound},
      }

      t.Log("Given the need to test downloading different content.")
      {
            for i, tt := range tests {
                  tf := func(t *testing.T) {
```

The only difference here is that we call Parallel function inside each of these individual sub test functions.

```
                        t.Parallel()

                        t.Logf("\tTest: %d\tWhen checking %q for status
code %d", i, tt.url, tt.statusCode)
                        {
                              resp, err := http.Get(tt.url)
                              if err != nil {
                                    t.Fatalf("\t%s\tShould be able to
make the Get call : %v", failed, err)
                              }
                              t.Logf("\t%s\tShould be able to make the
Get call.", succeed)

                              defer resp.Body.Close()

                              if resp.StatusCode == tt.statusCode {
                                    t.Logf("\t%s\tShould receive a %d
status code.", succeed, tt.statusCode)
                              } else {
                                    t.Errorf("\t%s\tShould receive a %d
status code : %v", failed, tt.statusCode, resp.StatusCode)
                              }
                        }
                  }

                  t.Run(tt.name, tf)
            }
      }
}
```

Because we have sub tests, we can run the following to separate them:
"go test -run TestSub -v"
"go test -run TestSub/statusok -v"
"go test -run TestSub/statusnotfound -v"
"go test -run TestParallelize -v"

```
=== RUN   TestSub
=== RUN   TestSub/statusok
=== RUN   TestSub/statusnotfound
--- PASS: TestSub (0.32s)
    sub_test.go:32: Given the need to test downloading different
content.
    --- PASS: TestSub/statusok (0.24s)
        sub_test.go:40:         Test: 0 When checking
"https://www.google.com/" for status code 200
        sub_test.go:46:         ✓       Should be able to make the Get
call.
        sub_test.go:51:         ✓       Should receive a 200 status
code.
    --- PASS: TestSub/statusnotfound (0.08s)
```

```
        sub_test.go:40:         Test: 1 When checking
"http://rss.cnn.com/rss/cnn_topstorie.rss" for status code 404
        sub_test.go:46:         ✓       Should be able to make the Get
call.
        sub_test.go:51:         ✓       Should receive a 404 status
code.
=== RUN   TestParallelize
=== RUN   TestParallelize/statusok
=== PAUSE TestParallelize/statusok
=== RUN   TestParallelize/statusnotfound
=== PAUSE TestParallelize/statusnotfound
=== CONT  TestParallelize/statusok
=== CONT  TestParallelize/statusnotfound
--- PASS: TestParallelize (0.00s)
    sub_test.go:77: Given the need to test downloading different
content.
    --- PASS: TestParallelize/statusok (0.09s)
        sub_test.go:85:         Test: 1 When checking
"http://rss.cnn.com/rss/cnn_topstorie.rss" for status code 404
        sub_test.go:91:         ✓       Should be able to make the Get
call.
        sub_test.go:96:         ✓       Should receive a 404 status
code.
    --- PASS: TestParallelize/statusnotfound (0.09s)
        sub_test.go:85:         Test: 1 When checking
"http://rss.cnn.com/rss/cnn_topstorie.rss" for status code 404
        sub_test.go:91:         ✓       Should be able to make the Get
call.
        sub_test.go:96:         ✓       Should receive a 404 status
code.
PASS
ok      command-line-arguments  0.618s
```

Web Server

Web Server

If we write our own web server, we would like to test it as well without manually having to stand up a server. The Go standard library also supports this.

Below is our simple web server.

```
package main
```

```
import (
	"log"
	"net/http"
```

Import handler package that has a set of routes that we are gonna work with.

```
	"github.com/hoanhan101/ultimate-go/go/testing/web_server/handlers"
)

func main() {
	handlers.Routes()

	log.Println("listener : Started : Listening on:
http://localhost:4000")
	http.ListenAndServe(":4000", nil)
}
```

Handlers

Package handlers provides the endpoints for the web service.

```
package handlers

import (
	"encoding/json"
	"net/http"
)
```

Routes sets the routes for the web service. It has 1 route call /sendjson. When that route is executed, it will call the SendJSON function.

```
func Routes() {
	http.HandleFunc("/sendjson", SendJSON)
}
```

SendJSON returns a simple JSON document. This has the same signature that we had before using ResponseWriter and Request. We create an anonymous struct, initialize it and unmarshall it into JSON and pass it down the line.

```
func SendJSON(rw http.ResponseWriter, r *http.Request) {
	u := struct {
		Name  string
		Email string
	}{
		Name:  "Hoanh An",
		Email: "hoanhan101@gmail.com",
	}

	rw.Header().Set("Content-Type", "application/json")
	rw.WriteHeader(200)
	json.NewEncoder(rw).Encode(&u)
}
```

Example Test

This is another type of test in Go. Examples are both tests and documentations. If we execute "godoc -http :3000", Go will generate for us a server that presents the documentation of our code. The interface will look like the official golang interface, but then inside the Packages section are our local packages.

Example functions are a little bit more concrete in terms of showing people how to use our API. More interestingly, Examples are not only for documentation but they can also be tests.

For them to be tested, we need to add a comment at the end of the functions: one is Output and one is expected output. If we change the expected output to be something wrong then, the compiler will tell us when we run the test. Below is an example.

Example tests are really powerful. They give users examples how to use the API and validate that the APIs and examples are working.

```
package handlers_test

import (
	"encoding/json"
	"fmt"
	"log"
	"net/http"
	"net/http/httptest"
)
```

ExampleSendJSON provides a basic example. Notice that we are binding that Example to our SendJSON function.

```
func ExampleSendJSON() {
	r := httptest.NewRequest("GET", "/sendjson", nil)
	w := httptest.NewRecorder()
	http.DefaultServeMux.ServeHTTP(w, r)

	var u struct {
		Name  string
		Email string
	}

	if err := json.NewDecoder(w.Body).Decode(&u); err != nil {
		log.Println("ERROR:", err)
	}

	fmt.Println(u)
}
```

```
=== RUN   ExampleSendJSON
--- PASS: ExampleSendJSON (0.00s)
PASS
ok      github.com/hoanhan101/ultimate-go/go/testing/web_server/handlers
0.096s
```

Internal Test

Below is how to test the execution of an internal endpoint without having to stand up the server. Run test using "go test -v -run TestSendJSON"

We are using handlers_test for package name because we want to make sure we only touch the exported API.

```
package handlers_test

import (
	"encoding/json"
	"net/http"
	"net/http/httptest"
	"testing"

	"github.com/hoanhan101/ultimate-go/go/testing/web_server/handlers"
)

const (
	succeed = "\u2713"
	failed  = "\u2717"
)
```

This is very critical. If we forget to do this then nothing will work.

```go
func init() {
	handlers.Routes()
}
```

TestSendJSON testing the sendjson internal endpoint.
In order to mock this call, we don't need the network. What we need to do is create a request and run it through the Mux so we are gonna bypass the network call together, run the request directly through the Mux to test the route and the handler.

```go
func TestSendJSON(t *testing.T) {
	url := "/sendjson"
	statusCode := 200

	t.Log("Given the need to test the SendJSON endpoint.")
	{
```

Create a nil request GET for the URL.

```go
		r := httptest.NewRequest("GET", url, nil)
```

NewRecorder gives us a pointer to its concrete type called ResponseRecorder that already implements the ResponseWriter interface.

```go
		w := httptest.NewRecorder()
```

ServerHTTP asks for a ResonseWriter and a Request. This call will perform the Mux and call that handler to test it without network. When his call comes back, the recorder value w has the result of the entire execution. Now we can use that to validate.

```go
		http.DefaultServeMux.ServeHTTP(w, r)

		t.Logf("\tTest 0:\tWhen checking %q for status code %d",
url, statusCode)
		{
			if w.Code != 200 {
				t.Fatalf("\t%s\tShould receive a
status code of %d for the response. Received[%d].", failed, statusCode,
w.Code)
                        }
                        t.Logf("\t%s\tShould receive a status code of %d
for the response.", succeed, statusCode)
```

If we got the 200, we try to unmarshal and validate it.

```
                  var u struct {
                        Name  string
                        Email string
                  }

                  if err := json.NewDecoder(w.Body).Decode(&u); err !=
nil {
                        t.Fatalf("\t%s\tShould be able to decode the
response.", failed)
                  }
                  t.Logf("\t%s\tShould be able to decode the response.",
succeed)

                  if u.Name == "Hoanh An" {
                  t.Logf("\t%s\tShould have \"Hoanh An\" for Name in the
response.", succeed)
                  } else {
t.Errorf("\t%s\tShould have \"Hoanh An\" for Name in the response : %q",
failed, u.Name)
                  }

                  if u.Email == "hoanhan101@gmail.com" {
                        t.Logf("\t%s\tShould have
\"hoanhan101@gmail.com\" for Email in the response.", succeed)
                  } else {
                        t.Errorf("\t%s\tShould have
\"hoanhan101@gmail.com\" for Email in the response : %q", failed,
u.Email)
                  }
            }
      }
}
```

```
=== RUN   TestSendJSON
--- PASS: TestSendJSON (0.00s)
    handlers_test.go:41: Given the need to test the SendJSON endpoint.
    handlers_test.go:55:        Test 0: When checking "/sendjson" for
status code 200
    handlers_test.go:60:        ✓       Should receive a status code of
200 for the response.
    handlers_test.go:71:        ✓       Should be able to decode the
response.
    handlers_test.go:74:        ✓       Should have "Hoanh An" for Name
in the response.
    handlers_test.go:80:        ✓       Should have
"hoanhan@bennington.edu" for Email in the response.
PASS
ok      github.com/hoanhan101/ultimate-go/go/testing/web_server/handlers
0.151s
```

Mock Server

Those basic tests that we just went through were cool but had a flaw: they require the use of the Internet. We cannot assume that we always have access to the resources we need. Therefore, mocking becomes an important part of testing in many cases. (Mocking databases if not the case here because it is hard to do so but other networking related things, we surely can do that).

The standard library already has the http test package that let us mock different http stuff right out of the box. Below is how to mock an http GET call internally.

```go
package main

import (
	"encoding/xml"
	"fmt"
	"net/http"
	"net/http/httptest"
	"testing"
)
```

feed is mocking the XML document we expect to receive. Notice that we are using backtick ` instead of double quotes " so we can reserve special characters.

```go
var feed = `<?xml version="1.0" encoding="UTF-8"?>
<rss>
<channel>
    <title>Going Go Programming</title>
    <description>Golang : https://github.com/goinggo</description>
    <link>http://www.goinggo.net/</link>
    <item>
        <pubDate>Sun, 15 Mar 2015 15:04:00 +0000</pubDate>
        <title>Object Oriented Programming Mechanics</title>
        <description>Go is an object oriented language.</description>
        <link>http://www.goinggo.net/2015/03/object-oriented</link>
    </item>
</channel>
</rss>`
```

Item defines the fields associated with the item tag in the mock RSS document.

```go
type Item struct {
	XMLName     xml.Name `xml:"item"`
	Title       string   `xml:"title"`
	Description string   `xml:"description"`
	Link        string   `xml:"link"`
}
```

Channel defines the fields associated with the channel tag in the mock RSS document.

```go
type Channel struct {
	XMLName     xml.Name `xml:"channel"`
	Title       string   `xml:"title"`
	Description string   `xml:"description"`
	Link        string   `xml:"link"`
	PubDate     string   `xml:"pubDate"`
	Items       []Item   `xml:"item"`
}
```

Document defines the fields associated with the mock RSS document.

```go
type Document struct {
	XMLName xml.Name `xml:"rss"`
	Channel Channel  `xml:"channel"`
	URI     string
}
```

mockServer returns a pointer of type httptest.Server to handle the mock get call. This mock function calls NewServer function that is gonna stand up a web server for us automatically. All we have to give NewServer is a function of the Handler type, which is f. f creates an anonymous function with the signature of

ResponseWriter and Request. This is the core signature of everything related to http in Go. ResponseWriter is an interface that allows us to write the response out. Normally when we get this interface value, there is already a concrete type value stored inside of it that supports what we are doing.

Request is a concrete type that we are gonna get with the request. This is how it's gonna work. We are gonna get a mock server started by making a NewServer call. When the request comes into it, execute f. Therefore, f is doing the entire mock. We are gonna send 200 down the line, set the header to XML and use Fprintln to take the Response Writer interface value and feed it with the raw string we defined above.

```
func mockServer() *httptest.Server {
	f := func(w http.ResponseWriter, r *http.Request) {
		w.WriteHeader(200)
		w.Header().Set("Content-Type", "application/xml")
		fmt.Fprintln(w, feed)
	}
	return httptest.NewServer(http.HandlerFunc(f))
}
```

TestWeb validates the http Get function can download content and the content can be unmarshalled and clean.

```
func TestWeb(t *testing.T) {
	statusCode := http.StatusOK
```

Call the mock server and defer close to shut it down cleanly.

```
	server := mockServer()
	defer server.Close()
```

Now, it's just the matter of using server value to know what URL we need to use to run this mock. From the http.Get point of view, it is making an URL call. It has no idea that it's hitting the mock server. We have mocked out a perfect response.

```
	t.Log("Given the need to test downloading content.")
	{
		t.Logf("\tTest 0:\tWhen checking %q for status code %d",
server.URL, statusCode)
		{
			resp, err := http.Get(server.URL)
			if err != nil {
```

```
                        t.Fatalf("\t%s\tShould be able to make the Get
call : %v", failed, err)
                    }
                    t.Logf("\t%s\tShould be able to make the Get call.",
succeed)

                    defer resp.Body.Close()

                    if resp.StatusCode != statusCode {
                        t.Fatalf("\t%s\tShould receive a %d status code :
%v", failed, statusCode, resp.StatusCode)
                    }
                    t.Logf("\t%s\tShould receive a %d status code.",
succeed, statusCode)
```

When we get the response back, we are unmarshaling it from XML to our struct type and do some extra validation with that as we go.

```
                    var d Document
                    if err := xml.NewDecoder(resp.Body).Decode(&d); err !=
nil {
                        t.Fatalf("\t%s\tShould be able to unmarshal the
response : %v", failed, err)
                    }
                    t.Logf("\t%s\tShould be able to unmarshal the
response.", succeed)

                    if len(d.Channel.Items) == 1 {
                        t.Logf("\t%s\tShould have 1 item in the feed.",
succeed)
                    } else {
                        t.Errorf("\t%s\tShould have 1 item in the feed :
%d", failed, len(d.Channel.Items))
                    }
                }
        }
}
```

```
=== RUN   TestWeb
--- PASS: TestWeb (0.00s)
    web_test.go:109: Given the need to test downloading content.
    web_test.go:111:    Test 0: When checking "http://127.0.0.1:58548"
for status code 200
    web_test.go:117:    ✓       Should be able to make the Get call.
    web_test.go:124:    ✓       Should receive a 200 status code.
    web_test.go:132:    ✓       Should be able to unmarshal the
response.
    web_test.go:135:    ✓       Should have 1 item in the feed.
PASS
ok      command-line-arguments  0.191s
```

Benchmarking

Basic Benchmark

Benchmark file's have to have <file_name>_test.go and use the Benchmark functions like below. The goal is to know what performs better and what allocate more or less between Sprint and Sprintf. Our guess is that Sprint is gonna be better because it doesn't have any overhead doing the formatting. However, this is not true. Remember we have to optimize for correctness so we don't want to guess.

```go
package main

import (
	"fmt"
	"testing"
)
var gs string
```

BenchmarkSprint tests the performance of using Sprint. All the code we want to benchmark need to be inside the b.N for loop. The first time the tool call it, b.N is equal to 1. It will keep increasing the value of N and run long enough based on our bench time. fmt.Sprint returns a value and we want to capture this value so it doesn't look like dead code. We assign it to the global variable gs.

```go
func BenchmarkSprintBasic(b *testing.B) {
	var s string

	for i := 0; i < b.N; i++ {
		s = fmt.Sprint("hello")
	}
	gs = s
}
```

BenchmarkSprint tests the performance of using Sprintf.

```go
func BenchmarkSprintfBasic(b *testing.B) {
	var s string

	for i := 0; i < b.N; i++ {
		s = fmt.Sprintf("hello")
	}
	gs = s
}
```

go test -run none -bench . -benchtime 3s -benchmem

```
goos: darwin
goarch: amd64
BenchmarkSprintBasic-16         52997451                        56.9 ns/op
5 B/op          1 allocs/op
BenchmarkSprintfBasic-16        72737234                        45.7 ns/op
5 B/op          1 allocs/op
PASS
ok      command-line-arguments  6.637s
```

Sub Benchmark

Like sub tests, we can also do sub benchmark.

```go
package main

import (
	"fmt"
	"testing"
)
```

BenchmarkSprint tests all the Sprint related benchmarks as sub benchmarks.

```go
func BenchmarkSprintSub(b *testing.B) {
	b.Run("none", benchSprint)
	b.Run("format", benchSprintf)
}
```

benchSprint tests the performance of using Sprint.

```go
func benchSprint(b *testing.B) {
	var s string


	for i := 0; i < b.N; i++ {
		s = fmt.Sprint("hello")
	}
	gs = s
}
```

benchSprintf tests the performance of using Sprintf.

```
func benchSprintf(b *testing.B) {
	var s string

	for i := 0; i < b.N; i++ {
		s = fmt.Sprintf("hello")
	}
	gs = s
}
```

go test -run none -bench . -benchtime 3s -benchmem

```
goos: darwin
goarch: amd64
BenchmarkSprintSub/none-16              54088082                60.6
ns/op                  5 B/op          1 allocs/op
BenchmarkSprintSub/format-16            67906119                52.3
ns/op                  5 B/op          1 allocs/op
PASS
ok      command-line-arguments  7.131s
```

Other running subtests like so:

- go test -run none -bench BenchmarkSprintSub/none -benchtime 3s -benchmem
- go test -run none -bench BenchmarkSprintSub/format -benchtime 3s -benchmem

Profiling

Stack Trace

Review Stack Trace

How to read stack traces?

```
package main

func main() {
```

We are making a slice of length 2, capacity 4 and then passing that slice value into a function call example. example takes a slice, a string, and an integer.

```
	example(make([]string, 2, 4), "hello", 10)
}
```

examples call the built-in function panic to demonstrate the stack traces.

```
func example(slice []string, str string, i int) {
	panic("Want stack trace")
}
```

This is the output that we get:

```
panic: Want stack trace

goroutine 1 [running]:
main.example(0xc420053f38, 0x2, 0x4, 0x1066c02, 0x5, 0xa)

/Users/hoanhan/go/src/github.com/hoanhan101/ultimate-go/go/profiling/sta
ck_trace.go:18 +0x39
main.main()

/Users/hoanhan/go/src/github.com/hoanhan101/ultimate-go/go/profiling/sta
ck_trace.go:13 +0x72
exit status 2
```

We already know that the compiler tells us the lines of problems. That's good. What is even better is that we know exactly what values are passed to the function at the time stack traces occur. Stack traces show words of data at a time. We know that a slice is a 3-word data structure. In our case, the 1st word is a pointer, 2nd is 2 (length) and 3rd is 4 (capacity). String is a 2-word data structure: a pointer and length of 5 because there are 5 bytes in string "hello". Then we have a 1 word integer of value 10.

In the stack traces, main.example(0xc420053f38, 0x2, 0x4, 0x1066c02, 0x5, 0xa), the corresponding values in the function are address, 2, 4, address, 5, a (which is 10 in base 2).

If we ask for the data we need, this is a benefit that we can get just by looking at the stack traces and see the values that are going in. If we work with the error package from Dave, wrap it and add more context, and log package, we have more than enough information to debug a problem.

Packing

Sample stack races that pack values.

```go
package main

func main() {
```

Passing values that are 1-byte values.

```go
	example(true, false, true, 25)
}

func example(b1, b2, b3 bool, i uint8) {
	panic("Want stack trace")
}
```

This is the output that we get:

```
panic: Want stack trace
goroutine 1 [running]:
main.example(0xc419010001)

/Users/hoanhan/go/src/github.com/hoanhan101/ultimate-go/go/profiling/sta
ck_trace_2.go:12 +0x39
main.main()

/Users/hoanhan/go/src/github.com/hoanhan101/ultimate-go/go/profiling/sta
ck_trace_2.go:8 +0x29
exit status 2
```

Since stack traces show 1 word at a time, all of these 4 bytes fit in a half-word on a 32-bit platform and a full word on 64-bit. Also, the system we are looking at is using little endian so we need to read from right to left. In our case, the word value 0xc419010001 can be represented as:

Bits	Binary	Hex	Value
00-07	0000 0001	01	true
08-15	0000 0000	00	false
16-23	0000 0001	01	true
24-31	0001 1001	19	25

GODEBUG

Memory Tracing

Memory Tracing gives us a general idea if our software is healthy as related to the GC and memory in the heap that we are working with. Below is a sample program that causes memory leaks.

```go
package main

import (
	"os"
	"os/signal"
)

func main() {
```

Create a Goroutine that leaks memory. Dumping key-value pairs to put tons of allocation.

```go
	go func() {
		m := make(map[int]int)
		for i := 0; ; i++ {
			m[i] = i
		}
	}()
```

Shutdown the program with Ctrl-C.

```go
	sig := make(chan os.Signal, 1)
	signal.Notify(sig)
	<-sig
}
```

We are using a special environmental variable called GODEBUG. It gives us the ability to do a memory trace and a scheduler trace. Here are the steps to build and run:

Build the program by: go build memory_tracing.go

Run the binary: GODEBUG=gctrace=1 ./memory_tracing

Setting the GODEBUG=gctrace=1 causes the garbage collector to emit a single line to standard error at each collection, summarizing the amount of memory collected and the length of the pause.

What we are gonna see are bad traces followed by this pattern:

gc {0} @{1}s {2}%: {3}+...+{4} ms clock, {5}+...+{6} ms cpu, {7}->{8}->{9} MB, {10} MB goal, {11} P

where:
    {0} : The number of times gc run
    {1} : The amount of time the program has been running.
    {2} : The percentage of CPU the gc is taking away from us.
    {3} : Stop of wall clock time - a measure of the real time including time that passes due to programmed delays or waiting for resources to become available.
    {4} : Stop of wall clock. This is normally a more important number to look at.
    {5} : CPU clock
    {6} : CPU clock
    {7} : The size of the heap prior to the gc starting.
    {8} : The size of the heap after the gc run.
    {9} : The size of the live heap.
    {10}: The goal of the gc, pacing algorithm.
    {11}: The number of processes.

Here’s the actual output for the run:

```
gc 1 @0.007s 0%: 0.010+0.13+0.030 ms clock, 0.080+0/0.058/0.15+0.24 ms
cpu, 5->5->3 MB, 6 MB goal, 8 P
gc 2 @0.013s 0%: 0.003+0.21+0.034 ms clock, 0.031+0/0.030/0.22+0.27 ms
cpu, 9->9->7 MB, 10 MB goal, 8 P
gc 3 @0.029s 0%: 0.003+0.23+0.030 ms clock, 0.029+0.050/0.016/0.25+0.24
ms cpu, 18->18->15 MB, 19 MB goal, 8 P
gc 4 @0.062s 0%: 0.003+0.40+0.040 ms clock, 0.030+0/0.28/0.11+0.32 ms
cpu, 36->36->30 MB, 37 MB goal, 8 P
gc 5 @0.135s 0%: 0.003+0.63+0.045 ms clock, 0.027+0/0.026/0.64+0.36 ms
cpu, 72->72->60 MB, 73 MB goal, 8 P
```

```
gc 6 @0.302s 0%: 0.003+0.98+0.043 ms clock, 0.031+0.078/0.016/0.88+0.34
ms cpu, 65->66->42 MB, 120 MB goal, 8 P
gc 7 @0.317s 0%: 0.003+1.2+0.080 ms clock, 0.026+0/1.1/0.13+0.64 ms cpu,
120->121->120 MB, 121 MB goal, 8 P
gc 8 @0.685s 0%: 0.004+1.6+0.041 ms clock, 0.032+0/1.5/0.72+0.33 ms cpu,
288->288->241 MB, 289 MB goal, 8 P
gc 9 @1.424s 0%: 0.004+4.0+0.081 ms clock, 0.033+0.027/3.8/0.53+0.65 ms
cpu, 577->577->482 MB, 578 MB goal, 8 P
gc 10 @2.592s 0%: 0.003+11+0.045 ms clock, 0.031+0/5.9/5.2+0.36 ms cpu,
499->499->317 MB, 964 MB goal, 8 P
```

It goes really fast in the beginning and starts to slow down. This is bad. The size of the heap is increasing every time the GC runs. Hence, we know that there is a memory leak.

Keep in touch

Let me know how The Ultimate Go Book works out for you. Send an email to hoanhan101@gmail.com.

If you're interested in getting updates on my latest insights, projects and becoming a better software engineer as well as a better person overall, feel free to join my visit my website at https://hoanhan101.github.io/.

Thanks for reading and good luck!