Highlights from the Previous Chapters

We often refer to the full explanation and examples in previous chapters. Here, each section summarizes all the concepts learned in this course and warns what not to do!

WE'LL COVER THE FOLLOWING

Hiding a variable by misusing short declaration

In the following code snippet:

```
var remember bool = false
if something {
  remember := true // Wrong.
}
// use remember
```

The variable `remember` will never become true outside of the if-body. Inside the if-body, a new `remember` variable that hides the outer `remember` is

the if-body, a new remember variable that hides the outer remember is

declared because of `:=`, and there it will be true. However, after the *closing* } of `if`, the variable `remember` regains its outer value *false*. So, write it as:

```
if something {
  remember = true
}
```

This can also occur with a for-loop, and can be particularly subtle in functions with *named return variables*, as the following snippet shows:

```
func shadow() (err error) {
  x, err := check1() // x is created; err is assigned to
  if err != nil {
    return // err correctly returned
  }
  if y, err := check2(x); err != nil { // y and inner err are created
    return // inner err shadows outer err so err is wrongly returned!
  } else {
    fmt.Println(y)
  }
  return
}
```

Misusing strings

When you need to do a lot of manipulations on a string, think about the fact that strings in Go (like in Java and C#) are immutable. String concatenations of the kind `a += b` are inefficient, mainly when performed inside a loop. They cause many reallocations and the copying of memory. Instead, one should use a `bytes.Buffer` to accumulate string content, like in the following snippet:

```
var b bytes.Buffer
...
for condition {
  b.WriteString(str) // appends string str to the buffer
}
return b.String()
```

> **Remark**: Due to compiler-optimizations and the size of the strings, using

> a buffer only starts to become more efficient when the number of concatenations is > **15**.

Using `defer` for closing a file in the wrong scope

Suppose you are processing a range of files in a *for*-loop, and you want to make sure the files are closed after processing by using `defer`, like this:

```
for _, file := range files {
  if f, err = os.Open(file); err != nil {
    return
  }
  // This is wrong. The file is not closed when this loop iteration ends.
  defer f.Close()
  // perform operations on f:
  f.Process(data)
}
```

But, at the end of the for-loop, `defer` is not executed. Therefore, the files are not closed! Garbage collection will probably close them for you, but it can yield errors. Better do it like this:

```
for _, file := range files {
  if f, err = os.Open(file); err != nil {
    return
  }
  // perform operations on f:
  f.Process(data)
  // close f:
  f.Close()
}
```

The keyword `defer` is only executed at the return of a function, not at the end of a loop or some other limited scope.

Confusing `new()` and `make()`

We have talked about this and illustrated it with code already at great length in Chapter 5 and Chapter 8. The point is:

- For slices, maps and channels: use `make`

- For arrays, structs and all value types: use `new`

No need to pass a pointer to a slice to a function

A slice is a pointer to an underlying array. Passing a slice as a parameter to a function is probably what you always want, which means namely passing a pointer to a variable to be able to change it, and not passing a copy of the data. So, you want to do this:

```
func findBiggest( listOfNumbers []int ) int {}
```

not this:

```
func findBiggest( listOfNumbers *[]int ) int {}
```

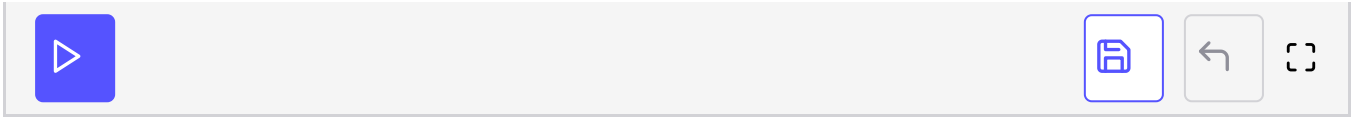
Do not dereference a slice when used as a parameter!

Using pointers to interface types

Look at the following program which *can't* be compiled:

```
package main
import (
"fmt"
)

type nexter interface {
  next() byte
}

func nextFew1(n nexter, num int) []byte {
  var b []byte
  for i:=0; i < num; i++ {
    b[i] = n.next()
  }
  return b
}

func nextFew2(n *nexter, num int) []byte {
  var b []byte

  for i:=0; i < num; i++ {
    b[i] = n.next()
    // compile error:

    // n.next undefined (type *nexter is pointer to interface, not interface)
  }
  return b
```

```
}

func main() {

  fmt.Println("Hello World!")
}
```



Pointers to Interface Type

In the code above, `nexter` is an interface with the method `next()`, which reads the next byte. `nextFew1()` has this interface type as a parameter and reads the next `num` bytes, returning them as a slice. This is ok. However, `nextFew2()` uses a pointer to the interface type as a parameter. When using the `next()` function, we get a clear compiler error: `n.next undefined (type *nexter is pointer to interface, not interface)`.

So never use a pointer to an interface type; this is already a pointer!

Misusing pointers with value types

Passing a value as a parameter in a function or as a receiver to a method may seem a misuse of memory because a value is always copied. But, on the other hand, values are allocated on the stack, which is quick and relatively cheap. If you pass a pointer to the value instead of the Go compiler, in most cases, we will see this as the making of an object, and we will move this object to the heap, causing an additional memory allocation. Therefore, nothing was gained in using a pointer instead of the value.

Misusing goroutines and channels

For didactic reasons and for gaining insight into their working, a lot of the examples in Chapter 12 applied goroutines and channels in very simple algorithms, like as a generator or iterator. In practice, often, you don't need the concurrency, or you don't need the overhead of the goroutines with channels; passing parameters using the stack is, in many cases, far more efficient. Moreover, it is likely to leak memory if you break, return or panic your way out of the loop because the goroutine blocks in the middle of doing something. In real code, it is often better to just write a simple procedural loop. Use goroutines and channels only where concurrency is important!

Using closures with goroutines

Look at the following code:

```
package main
import (
"fmt"
"time"
)

var values = [5]int{10, 11, 12, 13, 14}

func main() {
  // version A:
  for ix := range values { // ix is the index
    func() {
      fmt.Print(ix, " ")
    }() // call closure, prints each index
  }
  fmt.Println()
  // version B: same as A, but call closure as a goroutine
  for ix := range values {
    go func() {
      fmt.Print(ix, " ")
    }()
  }

  fmt.Println()
  time.Sleep(5e9)
  // version C: the right way
  for ix := range values {
    go func(ix int) {
     fmt.Print(ix, " ")
    }(ix)
  }
  fmt.Println()
  time.Sleep(5e9)
  // version D: print out the values:
  for ix := range values {
    val := values[ix]
    go func() {
      fmt.Print(val, " ")
    }()
  }
  time.Sleep(1e9)
}
```

Closures with Goroutines

Version A calls a closure five times, which prints the value of the index. **Version B** does the same but invokes each closure as a goroutine, with an argument that this would be faster because the closures execute in parallel. If

we leave enough time for all goroutines to run, the output of version B is **4 4 4 4 4**. Why is this? The `ix` variable in the above loop B is a single variable that takes on the index of each array element. Because the closures are all only bound to that one variable, there is a very good chance that, when you run this code, you will see the last index (4) printed for every iteration, instead of each index in the sequence. This is because the goroutines will probably not begin executing until after the loop, when `ix` has the value 4.

The right way to code that loop is **version C**: invoke each closure with `ix` as a parameter. Then, `ix` is evaluated at each iteration and placed on the stack for the goroutine, so each index is available to the goroutine when it is eventually executed. Note that the output depends on when each of the goroutines starts.

In **version D**, we print out the values of the array. Why does this work and version B do not? Because variables declared within the body of a loop (as `val` here) are not shared between iterations, and thus can be used separately in a closure.

Bad error handling

Stick to the patterns described in Chapter 11 and summarized in the chapter after it.

Don't use booleans

Making a *boolean* variable whose value is a test on the error-condition, like in the following, is superfluous:

```
var good bool
// test for an error, good becomes true or false
if !good {
  return errors.New("things aren't good")
}
```

Instead, test on the error immediately:

```
_, err1 := api.Func1()
if err1 != nil { ... }
```

Don't clutter code with error-checking

Avoid writing code like this:

```
... err1 := api.Func1()
if err1 != nil {
  fmt.Println("err: " + err.Error())
  return
}
err2 := api.Func2()
if err2 != nil {
  ...
  return
}
```

First, include the call to the functions in an initialization statement of the *if's*. Even then, the errors are reported (by printing them) with if-statements scattered throughout the code. With this pattern, it is hard to tell what is normal program logic and what is error checking/reporting. Also, notice that most of the code is dedicated to error conditions at any point in the code. A good solution is to wrap your error conditions in a closure wherever possible, like in the following example:

```
func httpRequestHandler(w http.ResponseWriter, req *http.Request) {
  err := func () error {
    if req.Method != "GET" {
      return errors.New("expected GET")
    }
    if input := parseInput(req); input != "command" {
      return errors.New("malformed command")
    }
    // other error conditions can be tested here
   } ()
   if err != nil {
     w.WriteHeader(400)
     io.WriteString(w, err)
     return
   }
  doSomething() ...
  ...
```

This approach clearly separates the error checking, error reporting, and normal program logic.

for `range` in arrays

for `range` in arrays

If you make a for `range` over an array with 1 value after the for, you get the indices back, not the values of the array. In:

```
for n := range array { ... }
```

The variable `n` gets the values **0, 1, ... , len(array)-1**. Moreover, the compiler won't warn you if the operations in the loop are compatible with *ints*.

These are some of the important concepts that are mostly overlooked when programming. In the next lesson, we'll be studying the comma, ok pattern used several times before in this course.