Golang

1. Primitive and declarations

```
// Boolean Type
var flag bool // no value assigned, set to false
var isAwesome = true

// Numeric
var num int32
var numF float32
```

Explicit Type Conversion

```
var x int = 10
var y float64 = 30.2
var z float64 = float64(x) + y
var d int = x + int(y)
```

Assign variable

```
var (
     x int
     y = 20
     z int = 30
     d, e = 40, "hello"
     f, g string
)
```

Go also supports a short declaration format. When you are within a function, you can use the := operator to replace a var declaration that uses type inference.

```
var x = 10
x := 10
```

The := operator can do one trick that you cannot do with var: it allows you to **assign values to existing variables**, too.

```
var x, y = 10, "hello"
x, y := 10, "hello
```

There are some situations within functions where you should avoid `:=`:

- When initializing a variable to its zero value, use `var x int`. This makes it clear that the zero value is intended.
- When assigning an untyped constant or a literal to a variable and the default type for the constant or literal isn't the type you want for the variable, use the long `var` form with the type specified. While it is legal to use a type conversion to specify the type of the value and use `:=` to write `x := byte(20)`, it is idiomatic to write `var x byte = 20`.
- Because `:=` allows you to assign to both new and existing variables, it sometimes creates new variables when you think you are reusing existing ones (see "Shadowing Variables" on page 62 for details). In those situations, explicitly declare all of your new variables with `var` to make it clear which variables are new, and then use the assignment operator (=) to assign values to both new and old variables.

Using const

```
const x int64 = 10
const (
    idKey = "id"
    nameKey = "name"
)
const z = 20 * 10
```

const in Go is very limited. Constants in Go are a way to give names to literals. They can only hold values that the compiler can figure out at compile time.

This means that they can be assigned:

- Numeric literals
- true and false
- Strings
- Runes
- The built-in functions complex, real, imag, len, and cap

Go doesn't provide a way to specify that a value calculated at runtime is immutable. Hence there are no immutable **arrays, slices, maps, or structs**, and there's no way to declare that a **field in a struct is immutable.**

Untype vs type constants

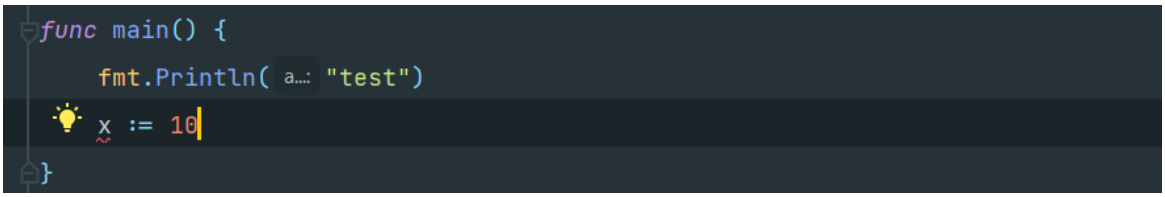
```
// Un typed
const x = 10

var y int = x
var z float64 = x
var d byte = x // no error

// Typed
const typedX int = 10
var typedY int8 = typedX // error int8 != int32
```

Unused variable

`Go requirement is that every declared local variable must be read. It is a compile-time error to declare a local variable and to not read its value.`



2. Composite Types

Arrays

```
var x [3]int
var x = [3]int{10, 20, 30}
var x = [12]int{1, 5: 4, 6, 10: 100, 15}
// [1, 0, 0, 0, 0, 4, 6, 0, 0, 0, 100, 15]

var x = [...]int{10, 20, 30}
var x = [...]int{1, 2, 3}
var y = [3]int{1, 2, 3}
fmt.Println(x == y) // prints true

var x [2][3]int
fmt.Println(len(x))
```

`You cannot use a variable to specify the size of an array, because types must be resolved at compile time, not at runtime.`

Slices

```go
var s = []int{10, 20, 30} // no size declare
// Using […] makes an array. Using [] makes a slice.

s = append(s, 10)

d := []int{10, 20 , 30}
s = append(s, d...) // append two slice
```

A slice is the first type we've seen that isn't comparable. It is a compile-time error to use == to see if two slices are identical or != to see if they are different. The only thing you can compare a slice with is nil

Go is a call by value language. Every time you pass a parameter to a function, Go makes a copy of the value that's passed in. Passing a slice to the append function actually passes a copy of the slice to the function. The function adds the values to the copy of the slice and returns the copy. You then assign the returned slice back to the variable in the calling function

Capacity

As we've seen, a slice is a sequence of values. Each element in a slice is assigned to consecutive memory locations, which makes it quick to read or write these values. Every slice has a *capacity*, which is the number of consecutive memory locations reserved. This can be larger than the length. Each time you append to a slice, one or more values is added to the end of the slice. Each value added increases the length by one. When the length reaches the capacity, there's no more room to put values. If you try to add additional values when the length equals the capacity, the append function uses the Go runtime to allocate a new slice with a larger capacity. The values in the original slice are copied to the new slice, the new values are added to the end, and the new slice is returned.

The Go Runtime

Every high-level language relies on a set of libraries to enable programs written in that language to run, and Go is no exception. The Go runtime provides services like memory allocation and garbage collection, concurrency support, networking, and implementations of built-in types and functions.

The Go runtime is compiled into every Go binary. This is different from languages that use a virtual machine, which must be installed separately to allow programs written in those languages to function. Including the runtime in the binary makes it easier to distribute Go programs and avoids worries about compatibility issues between the runtime and the program.

Capacity increase 25% when slice increase in length

```go
var x []int
fmt.Println(x, len(x), cap(x))
x = append(x, 10)
fmt.Println(x, len(x), cap(x))
x = append(x, 20)
fmt.Println(x, len(x), cap(x))
x = append(x, 30)
fmt.Println(x, len(x), cap(x))
x = append(x, 40)
fmt.Println(x, len(x), cap(x))
x = append(x, 50)
fmt.Println(x, len(x), cap(x))
```

```
[] 0 0
[10] 1 1
[10 20] 2 2
[10 20 30] 3 4
[10 20 30 40] 4 4
[10 20 30 40 50] 5 8
```

make Function

```go
k := make([]int, 5, 10) // length = 5, capacity = 10
k = append(k,10) // [0,0,0,0,0, 10]
```

Slicing slice:

```go
    x := []int{1, 2, 3, 4}
    y := x[:2]
    z := x[1:]
    m := x[1:3]
    e := x[:]
    fmt.Println("x:", x)
    fmt.Println("y:", y)
    fmt.Println("z:", z)
    fmt.Println("d:", m)
    fmt.Println("e:", e)
```

```
x: [1 2 3 4]
y: [1 2]
z: [2 3 4]
d: [2 3]
e: [1 2 3 4]
```

Slices share storage sometimes

When you take a slice from a slice, you are *not* making a copy of the data. Instead, you now have two variables that are sharing memory. This means that changes to an element in a slice affect all slices that share that element. Let's see what happens when we change values. You can run the code in Example 3-5 on The Go Playground.

Example 3-5. Slices with overlapping storage

```
x := []int{1, 2, 3, 4}
y := x[:2]
z := x[1:]
x[1] = 20
y[0] = 10
z[1] = 30
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

You get the following output:

```
x: [10 20 30 4]
y: [10 20]
z: [20 30 4]
```

Changing x modified both y and z, while changes to y and z modified x.

```
// Sub slice

    x = append(x, 1, 2, 3, 4)
    y := x[:2]
    z := x[2:]
    fmt.Println(cap(x), cap(y), cap(z))
    y = append(y, 30, 40, 50)
    x = append(x, 60)
    z = append(z, 70)
    fmt.Println("x:", x)
    fmt.Println("y:", y)
    fmt.Println("z:", z)

5 5 3
x: [1 2 30 40 70]
y: [1 2 30 40 70]
z: [30 40 70]

// Full Slice
// input[low:high:max]
// cap = max - low

    x := make([]int, 0, 5)
    x = append(x, 1, 2, 3, 4)
    y := x[:2:2]
    z := x[2:4:4]
    fmt.Println(cap(x), cap(y), cap(z))
    y = append(y, 30, 40, 50)
    x = append(x, 60)
    z = append(z, 70)
    fmt.Println("x:", x)
    fmt.Println("y:", y)
    fmt.Println("z:", z)

5 2 2
x: [1 2 3 4 60]
y: [1 2 30 40 50]
z: [3 4 70]

// Or you can use copy func
// If you need to create a slice that’s independent of the original,
// use the built-in copy function.
x := []int{1, 2, 3, 4}
y := make([]int, 4)
num := copy(y, x)
fmt.Println(y, num)
```

Strings and Runes and Bytes

```go
var s string = "Hello there"
var s2 string = s[4:7]
var s3 string = s[:5]
var s4 string = s[6:]
```

> While it's handy that Go allows us to use slicing notation to make substrings and use index notation to extract individual entries from a string, you should be very careful when doing so. Since strings are immutable, they don't have the modification problems that slices of slices do. There is a different problem, though. A string is composed of a sequence of bytes, while a code point in UTF-8 can be anywhere from one to four bytes long. Our previous example was entirely composed of code points that are one byte long in UTF-8, so everything worked out as expected. But when dealing with languages other than English or with emojis, you run into code points that are multiple bytes long in UTF-8:

Rune take 1 code point meanwhile byte take 1 byte

```go
runes := []rune(s)
bytes := []byte(s)
fmt.Println(runes) // [72 101 108 108 111 32 32 128123]
fmt.Println(bytes) // [72 101 108 108 111 32 32 240 159 145 187]
```

Maps

```go
var nilMap map[string]int
teams := map[string]int {"ACB": 10, "OCB" : 100}
ages := make(map[int][]string, 10)


m := map[string]int{
    "hello": 5,
    "world": 0,
}
v, ok := m["hello"]
fmt.Println(v, ok) // 5 true
v, ok = m["world"]
fmt.Println(v, ok)
v, ok = m["goodbye"]
fmt.Println(v, ok) // 0 false

m := map[string]int{
    "hello": 5,
    "world": 10,
}
delete(m, "hello")
```

Maps are like slices in several ways:

- Maps automatically grow as you add key-value pairs to them.
- If you know how many key-value pairs you plan to insert into a map, you can use `make` to create a map with a specific initial size.
- Passing a map to the `len` function tells you the number of key-value pairs in a map.
- The zero value for a map is `nil`.
- Maps are not comparable. You can check if they are equal to `nil`, but you cannot check if two maps have identical keys and values using == or differ using `!=`.

Struct

```
type person struct {
    name string
    age int
    pet string
}

var fred person
bob := person{}



julia := person{
    "Julia",
    40,
    "cat",
}

// Anonymous Structs

pet := struct {
    name string
    kind string
}{
    name: "Fido",
    kind: "dog",
}
```

Struct converting:

```
type firstPerson struct {
    name string
    age int
}
f := firstPerson{
    name: "Bob",
    age: 50,
}
var g struct {
    name string
    age int
}
// compiles -- can use = and == between identical named and anonymous structs
g = f
fmt.Println(f == g)
```

3. Blocks, Shadows, and Control Structures

```
n := rand.Intn(10)
if n == 0 {
    fmt.Println("That's too low")
} else if n > 5 {
    fmt.Println("That's too big:", n)
} else {
    fmt.Println("That's a good number:", n)
}



if n := rand.Intn(10); n == 0 {
    fmt.Println("That is too low")
} else if n > 10 {
    fmt.Println("That is too high")
} else {
    fmt.Println("good")
}
```

for, Four Ways

```go
for i := 0; i < 10; i++ {
    fmt.Println(i)
}

i := 1
for i < 100 {
    fmt.Println(i)
    i = i * 2
}

for {
    fmt.Println("Hello")
}

evenVal := []int{2, 4, 6, 8, 10, 12}
for i, v := range evenVal {
    fmt.Println(i, v)   // i , v
}

myMap := map[string]int{"duy" : 10, "dat" : 20}
for k := range myMap {
    fmt.Println(k) // key
    fmt.Println(myMap[k]) // value
}


stringSlice := []string{"hello", "kitty"}
for _, sample := range  stringSlice {
    for i, j := range sample {
        fmt.Println(i, j, string(j)) // index - code point - convert to string
    }
}

values := []int{2, 4, 6, 8, 10, 12}
for _, v := range values { // for-range value is a copy
    v *= 2
}
fmt.Println(values) // [2 4 6 8 10 12]
```

switch

```
for i := 0; i < 10; i++ {
    switch j := i*100; j {
    case 100: fmt.Println("Je")
    case 200, 300, 400, 500: fmt.Println("Jo")
    case 600, 700, 800, 900: fmt.Println("Foo")
    default:
        fmt.Println("No")
    }
}
```

```
// Blank switch
switch {
case a == 2:
    fmt.Println("a is 2")
case a == 3:
    fmt.Println("a is 3")
case a == 4:
    fmt.Println("a is 4")
default:
    fmt.Println("a is ", a)
}
```

4. Function

```go
func div(num int, de int) int {
    if de == 0 {
        return 0
    }
    return num / de // return is require
}

// FuncOpts Parameters as struct
type FuncOpts struct {
    firstName string
    lastName string
    age int
}
func Func(opts FuncOpts) {
    fmt.Println(opts.firstName)
    fmt.Println(opts.lastName)
    fmt.Println(opts.age)
}

// VarFunc Variadic parameter
func VarFunc(x int, remains ...int) {
    fmt.Println("x ", x)
    for i := 0; i < len(remains); i++ {
        fmt.Println(remains[i])
    }
}

// MultipleFunc Multiple return
func MultipleFunc() (rune, rune) {
    return 1 , 2
}

// Named return values
func divAndRemainder(numerator int, denominator int) (result int, remainder int,
    err error) {
    if denominator == 0 {
        err = errors.New("cannot divide by zero")
        return result, remainder, err
    }
    result, remainder = numerator/denominator, numerator%denominator
    return result, remainder, err
}

// Functions are values
func add(i int, j int) int { return i + j }
func sub(i int, j int) int { return i - j }

func mapOfFunc() {
    mapFunc := map[string] func(int, int) int { "+" : add, "-" : sub}
    _ = mapFunc
```

```
}

// Function type declaration
type operation func(int, int) int

func mapOfOp() {
    mapOps := map[string] operation {"+" : add, "-" : sub}
    _ = mapOps
}

// Anonymous function
func ano() {
    for i := 0; i < 5; i++ {

        func(j int) {
            fmt.Println(j)
        }(i)
    }
}
```

Closures

`Functions declared inside of functions are special; they are closures. This is a computer science word that means that functions declared inside of functions are able to access and modify variables declared in the outer function.`

Defer

> Programs often create temporary resources, like files or network connections, that need to be cleaned up. This cleanup has to happen, no matter how many exit points a function has, or whether a function completed successfully or not. In Go, the cleanup code is attached to the function with the `defer` keyword.

Normally, a function call runs immediately, but defer delays the invocation **until the surrounding function exits**.

> There are a few more things that you should know about `defer`. First, you can `defer` multiple closures in a Go function. They run in last-in-first-out order; the last `defer` registered runs first.
>
> The code within `defer` closures runs *after* the return statement. As I mentioned, you can supply a function with input parameters to a `defer`. Just as `defer` doesn't run immediately, any variables passed into a deferred closure aren't evaluated until the closure runs.

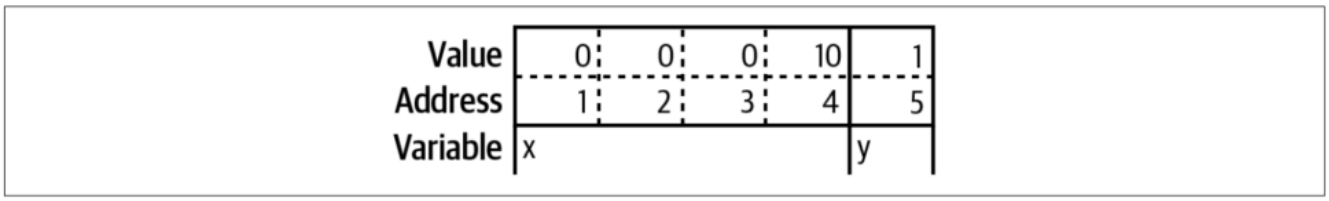
You can supply a function that returns values to a `defer`, but there's no way to read those values.

```
func example() {
    defer func() int {
        return 2 // there's no way to read this value
    }()
}
```

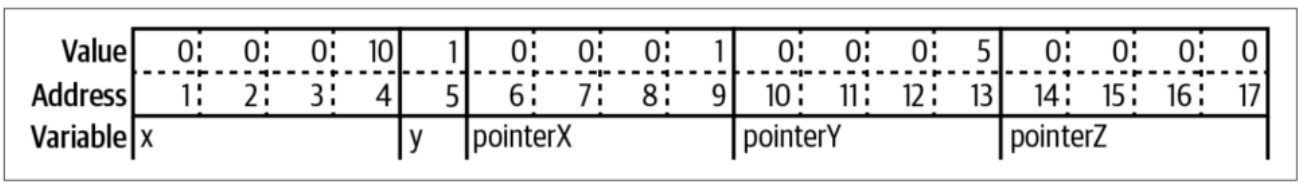
5. Pointers

A pointer is simply a variable that holds the location in memory where a value is stored. If you've taken computer science courses, you might have seen a graphic to represent how variables are stored in memory. The representation of the following two variables would look something like Figure 6-1:

```
var x int32 = 10
var y bool = true
```




```
var x int32 = 10
var y bool = true
pointerX := &x
pointerY := &y
var pointerZ *string
```



While different types of variables can take up different numbers of memory locations, every pointer, no matter what type it is pointing to, is always the same size: a number that holds the location in memory where the data is stored. Our pointer to `x`, `pointerX`, is stored at location 6 and has the value 1, the address of `x`. Similarly, our pointer to `y`, `pointerY`, is stored at location 10 and has the value 5, the address of `y`. The last pointer, `pointerZ`, is stored at location 14 and has the value 0, because it doesn't point to anything.

```
    val := 5
    pVal := &val
    fmt.Println(pVal) // 0xc0000ac058
    fmt.Println(*pVal) // 5

    var x *int
    fmt.Println(x == nil) // prints true
    // fmt.Println(*x)
    // panics null pointer

    var y = new(int)
    fmt.Println(y == nil) // prints false
    fmt.Println(*y) // prints 0
```

Pointer as attribute of struct

```
type person struct {
    FirstName string
    MiddleName *string
    LastName string
}

p := person{
    FirstName: "Pat",
    MiddleName: "Perry", // This line won't compile
    LastName: "Peterson",
}
```

Work around

```
    q := person{
        FirstName: "Pat",
        MiddleName: stringPointer("Perry"), // This works
        LastName: "Peterson",
    }
    _ = q
```

```go
type person struct {
    FirstName string
    MiddleName *string
    LastName string
}

func stringPointer(s string) *string {
    return &s
}
```

Pointer Passing Performance

If a struct is large enough, there are performance improvements from using a pointer to the struct as either an input parameter or a return value. The time to pass a pointer into a function is constant for all data sizes, roughly one nanosecond. This makes sense, as the size of a pointer is the same for all data types. Passing a value into a function takes longer as the data gets larger. It takes about a millisecond once the value gets to be around 10 megabytes of data.

Slice characteristic:

> The result is that a slice that's passed to a function can have its contents modified, but the slice can't be resized. As the only usable linear data structure, slices are frequently passed around in Go programs. By default, you should assume that a slice is not modified by a function. Your function's documentation should specify if it modifies the slice's contents.

Using slice to optimize read:

```
file, err := os.Open(fileName)
if err != nil {
    return err
}
defer file.Close()
data := make([]byte, 100)
for {
    count, err := file.Read(data)
    if err != nil {
        return err
    }
    if count == 0 {
        return nil
    }
    process(data[:count])
}
```

6. Types, Methods, and Interfaces

```
type Person struct {
    FirstName string
    LastName string
    Age int
}

type Score int
type Converter func(string)Score
type TeamScores map[string]Score

// Like most modern languages, Go supports methods on user-defined types.
func (person Person) printPerson() {
    _ = fmt.Sprintf("%s %s age %d", person.FirstName, person.LastName, person.Age)
}


person.printPerson()
```

Pointer Receivers and Value Receivers

As we covered in Chapter 6, Go uses parameters of pointer type to indicate that a parameter might be modified by the function. The same rules apply for method receivers, too. They can be *pointer receivers* (the type is a pointer) or *value receivers* (the type is a value type). The following rules help you determine when to use each kind of receiver:

- If your method modifies the receiver, you *must* use a pointer receiver.
- If your method needs to handle `nil` instances (see "Code Your Methods for nil Instances" on page 133), then it *must* use a pointer receiver.
- If your method doesn't modify the receiver, you *can* use a value receiver.

```
// Counter Pointer Receivers and Value Receivers
type Counter struct {
    total int
    lastUpdated time.Time
}
func (c *Counter) Increment() {
    c.total++
    c.lastUpdated = time.Now()
}
func (c Counter) String() string {
    return fmt.Sprintf("total: %d, last updated: %v",
                       c.total, c.lastUpdated)
}

var c Counter
fmt.Println(c.String())
c.Increment()
fmt.Println(c.String())


total: 0, last updated: 0001-01-01 00:00:00 +0000 UTC
total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC m=+0.000000001
```

```
// IntTree Code Your Methods for nil Instances
type IntTree struct {
    data int
    leftTree, rightTree *IntTree
}

func (root *IntTree) insert(val int) *IntTree {
    if root == nil {
        return &IntTree{data: val}
    } else {
        if val < root.data {
            root.leftTree = root.leftTree.insert(val)
        } else {
            root.rightTree = root.rightTree.insert(val)
        }
        return root
    }
}
```

Functions Versus Methods

Since you can use a method as a function, you might wonder when you should declare a function and when you should use a method.

The differentiator is whether or not your function depends on other data. As we've covered several times, package-level state should be effectively immutable. Any time your logic depends on values that are configured at startup or changed while your program is running, those values should be stored in a struct and that logic should be implemented as a method. If your logic only depends on the input parameters, then it should be a function.

Declaring a type based on another type looks a bit like inheritance, but it isn't. The two types have the same underlying type, but that's all. There is no hierarchy between these types. In languages with inheritance, a child instance can be used anywhere the parent instance is used. The child instance also has all the methods and data structures of the parent instance. That's not the case in Go. You can't assign an instance of type `HighScore` to a variable of type `Score` or vice versa without a type conversion, nor can you assign either of them to a variable of type `int` without a type conversion. Furthermore, any methods defined on `Score` aren't defined on `HighScore`:

```
// assigning untyped constants is valid
var i int = 300
var s Score = 100
var hs HighScore = 200
hs = s                  // compilation error!
s = i                   // compilation error!
s = Score(i)            // ok
hs = HighScore(s)       // ok
```

Embedded composistion:

```go
// Employee Use Embedding for Composition
type Employee struct {
    Name string
    ID string
}
func (e Employee) Description() string {
    return fmt.Sprintf("%s (%s)", e.Name, e.ID)
}
type Manager struct {
    employee Employee
    Reports []Employee
}
func (m Manager) printAllEmployee()  {
    // do business logic
    _ = fmt.Sprintf("Manager %s \n", m.employee.Name)
    for _, report := range m.Reports {
        fmt.Println(report.Description())
    }
}

func (m *Manager) newEmployee(e Employee) *Manager {
    m.Reports = append(m.Reports, e)
    return m
}

func testTypeInterface3(){
    e1 := Employee{Name: "Duy", ID:   "100"}
    e2 := Employee{Name: "Dat", ID:   "101"}
    e3 := Employee{Name: "Sang", ID:   "102"}
    m := Manager {
        employee: Employee{
            Name: "A",
            ID:   "F",
        },
        Reports:  []Employee{},
    }
    m.newEmployee(e1)
    m.newEmployee(e2)
    m.newEmployee(e3)
    m.printAllEmployee()
}
```

Interface

Duck typing:

Dynamically typed languages like Python, Ruby, and JavaScript don't have interfaces. Instead, those developers use "duck typing," which is based on the expression "If it walks like a duck and quacks like a duck, it's a duck." The concept is that you can pass an instance of a type as a parameter to a function as long as the function can find a method to invoke that it expects:

```
// Interface

type LogicProvider struct {}
func (lp LogicProvider) Process(data string) string {
    // business logic
    return ""
}

type Logic interface {
    Process(data string) string
}

type Client struct{
    L Logic // ony define interface
}

func(c Client) Program() {
    // get data from somewhere
    c.L.Process("data")
}
func testTypeInterface4() {
    c := Client{L: LogicProvider{}}
    c.Program()
}
```

Accept Interfaces, Return Structs:

Accept Interfaces, Return Structs

You'll often hear experienced Go developers say that your code should "Accept interfaces, return structs." What this means is that the business logic invoked by your functions should be invoked via interfaces, but the output of your functions should be a concrete type. We've already covered why functions should accept interfaces: they make your code more flexible and explicitly declare exactly what functionality is being used.

Type assertion

```go
type MyInt int
func testTypeInterface5() {
    var i interface{}
    var mine MyInt = 20
    i = mine
    i2 := i.(MyInt)
    fmt.Println(i2 + 1)
}
```

Embedding and Interfaces

```go
type Reader interface {
    Read(p []byte) (n int, err error)
}
type Closer interface {
    Close() error
}
type ReadCloser interface {
    Reader
    Closer
}
```

Accept Interfaces, Return Structs

You'll often hear experienced Go developers say that your code should "Accept interfaces, return structs." What this means is that the business logic invoked by your functions should be invoked via interfaces, but the output of your functions should be a concrete type. We've already covered why functions should accept interfaces: they make your code more flexible and explicitly declare exactly what functionality is being used.

If you create an API that returns interfaces, you are losing one of the main advantages of implicit interfaces: decoupling. You want to limit the third-party interfaces that your client code depends on because your code is now permanently dependent on the module that contains those interfaces, as well as any dependencies of that module, and so on. (We talk about modules and dependencies in Chapter 9.) This limits future flexibility. To avoid the coupling, you'd have to write another interface and do a type conversion from one to the other. While depending on concrete instances can lead to dependencies, using a dependency injection layer in your application limits the effect. We'll talk more about dependency injection in "Implicit Interfaces Make Dependency Injection Easier" on page 155.

`Another reason to avoid returning interfaces is versioning. If a concrete type is returned, new methods and fields can be added without breaking existing code. The same is not true for an interface. Adding a new method to an interface means that you need to update all existing implementations of the interface, or your code breaks`

Nil and interface

```go
var s *string
fmt.Println(s == nil) // prints true
var i interface{}
fmt.Println(i == nil) // prints true
i = s
fmt.Println(i == nil) // prints false
```

Empty interface say nothing:

Sometimes in a statically typed language, you need a way to say that a variable could store a value of any type. Go uses `interface{}` to represent this:

```go
var i interface{}
i = 20
i = "hello"
i = struct {
    FirstName string
    LastName string
} {"Fred", "Fredson"}
```

You should note that `interface{}` isn't special case syntax. An empty interface type simply states that the variable can store any value whose type implements zero or more methods. This just happens to match every type in Go. Because an empty interface doesn't tell you anything about the value it represents, there isn't a lot you can do with it. One common use of the empty interface is as a placeholder for data of uncertain schema that's read from an external source, like a JSON file:

Empty interface can be use a temporary for all data type or can be used as generic:

```
    data := map[string]interface{}{}
    contents, err := ioutil.ReadFile("data.json")
    if err != nil {
        return err
    }

    _ = json.Unmarshal(contents, &data)
    // the contents are now in the data map
    fmt.Println(data)
    return nil
```

```
// Use as generic

type LinkedList struct {
    Value interface{}
    Next *LinkedList
}
func (ll *LinkedList) Insert(pos int, val interface{}) *LinkedList {
    if ll == nil || pos == 0 {
        return &LinkedList{
            Value: val,
            Next: ll,
        }
    }
    ll.Next = ll.Next.Insert(pos-1, val)
    return ll
}
```

Type Assertions and Type Switches

A type assertion names the concrete type that implemented the interface, or names another interface that is also implemented by the concrete type underlying the interface.

```
    var i interface{}
    var mine MyInt = 20
    i = mine
    i2 := i.(MyInt)
    fmt.Println(i2 + 1)
```

Dependency Injection

```go
// Dependency Injection

func LogOutput(message string) {
    fmt.Println(message)
}

type LoggerAdapter func(message string)

func (lg LoggerAdapter) Log(message string) {
    lg(message)
}

type SimpleDataStore struct {
    userData map[string]string
}

func (sds SimpleDataStore) UserNameForID(userID string) (string, bool) {
    name, ok := sds.userData[userID]
    return name, ok
}

// NewSimpleDataStore Factory function
func NewSimpleDataStore() SimpleDataStore {
    return SimpleDataStore{
        userData: map[string]string{
            "1": "Fred",
            "2": "Mary",
            "3": "Pat",
        },
    }
}

/* Define Interface */

type DataStore interface {
    UserNameForID(userID string) (string, bool)
}
type Logger interface {
    Log(message string)
}

/* Impl */

type SimpleLogic struct {
    d DataStore
    l Logger
}

func (sl SimpleLogic) sayHello(userId string) (string, error) {
    sl.l.Log("Hello")
```

```
    name, ok := sl.d.UserNameForID(userId)
    if !ok {
        return "", errors.New("unknown user")
    }
    return name, nil
}

func (sl SimpleLogic) SayGoodbye(userID string) (string, error) {
    sl.l.Log("in SayGoodbye for " + userID)
    name, ok := sl.d.UserNameForID(userID)
    if !ok {
        return "", errors.New("unknown user")
    }
    return "Goodbye, " + name, nil
}


func NewSimpleLogic(l Logger, ds DataStore) SimpleLogic {
    return SimpleLogic{
        l: l,
        d: ds,
    }
}
```

7. Handling Errors

`error` is a built-in interface that defines a single method:

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

Sentinel errors are one of the few variables that are declared at the package level. By convention, their names start with `Err` (with the notable exception of `io.EOF`). They should be treated as read-only; there's no way for the Go compiler to enforce this, but it is a programming error to change their value.

```go
func calcRemainderAndMod(numerator, denominator int) (int, int, error) {
    if denominator == 0 {
        return 0, 0, errors.New("denominator is 0")
    }
    return numerator / denominator, numerator % denominator, nil
}


func handleErr() {
    numerator := 20
    denominator := 3
    remainder, mod, err := calcRemainderAndMod(numerator, denominator)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(remainder, mod)
}

func doubleEven(i int) (int, error) {
    if i % 2 != 0 {
        return 0, fmt.Errorf("%d isn't an even number", i)
        // Using format
    }
    return i * 2, nil
}

// Errors Are Values

type Status int
const (
    InvalidLogin Status = iota + 1
    NotFound
)

type StatusErr struct {
    Status Status
    Message string
}

func (se StatusErr) Error() string {
    return se.Message
}

func login(uid string, pwd string) error {return nil}
func getData(file string) (string, error) {return "", nil}

func LoginAndGetData(uid, pwd, file string) ([]byte, error) {
    err := login(uid, pwd)
    if err != nil {
        return nil, StatusErr{
```

```
            Status: InvalidLogin,
            Message: fmt.Sprintf("invalid credentials for user %s", uid),
        }
    }
    _, err = getData(file)
    if err != nil {
        return nil, StatusErr{
            Status: NotFound,
            Message: fmt.Sprintf("file %s not found", file),
        }
    }
    return []byte{}, nil
}

// Is and As

func fileCheck(name string) error {
    f, err := os.Open(name)
    if err != nil {
        return fmt.Errorf("file check fail: %w", err)
    }
    _ = f.Close()
    return nil
}

func errMain1(){
    err := fileCheck("hello")
    if err != nil {
        if errors.Is(err, os.ErrNotExist) {
            // Checking Sentinel Errors
            fmt.Println("File not found")
        }
    }
}
```

panic and recover

In previous chapters, we've mentioned panics in passing without going into any details on what they are. Go generates a panic whenever there is a situation where the Go runtime is unable to figure out what should happen next. This could be due to a programming error (like an attempt to read past the end of a slice) or environmental problem (like running out of memory). As soon as a panic happens, the current function exits immediately and any defers attached to the current function start running. When those defers complete, the defers attached to the calling function run, and so on, until `main` is reached. The program then exits with a message and a stack trace.

If there are situations in your programs that are unrecoverable, you can create your own panics. The built-in function `panic` takes one parameter, which can be of any type. Usually, it is a string. Let's make a trivial program that panics and run it on The Go Playground:

```
func doPanic(msg string) {
     panic(msg)
}
```

```
func div60(i int) {
    defer func() {
        if v := recover(); v != nil {
            fmt.Println(v)
        }
    }()
    fmt.Println(60 / i)
}

func main() {
    for _, val := range []int{1, 2, 0, 6} {
        div60(val)
    }
}
```

8. Import package module

This leads to the question: how do you export an identifier in Go? Rather than use a special keyword, Go uses *capitalization* to determine if a package-level identifier is visible outside of the package where it is declared. An identifier whose name starts with an uppercase letter is *exported*. Conversely, an identifier whose name starts with a lowercase letter or underscore can only be accessed from within the package where it is declared.

```
package main

import (
    "fmt"

    "github.com/learning-go-book/package_example/formatter"
    "github.com/learning-go-book/package_example/math"
)

func main() {
    num := math.Double(2)
    output := print.Format(num)
    fmt.Println(output)
}
```

It is a compile-time error to import a package but not use any of the identifiers exported by the package. This ensures that the binary produced by the Go compiler only includes code that's actually used in the program.

Overriding a Package's Name

```
import (
    crand "crypto/rand"
    "encoding/binary"
    "fmt"
    "math/rand"
)
```

```
package main

import (
    "fmt"
    "module/github.com/config"
)

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

9. Concunrrency in Go

```go
func process(val int) int {
    val += 100
    return val
}

func GoProcess(in <- chan int, out chan <- int) {
    go func() {
        for vaL := range in{
            result := process(vaL)
            out <- result
        }
    }()
}
```

Channels

Goroutines communicate using *channels*. Like slices and maps, channels are a built-in type created using the `make` function:

```go
ch := make(chan int)
```

Like maps, channels are reference types. When you pass a channel to a function, you are really passing a pointer to the channel. Also like maps and slices, the zero value for a channel is `nil`.

```go
ch := make(chan int, 10)
```

for-range and Channels

You can also read from a channel using a `for-range` loop:

```go
for v := range ch {
    fmt.Println(v)
}
```

Unlike other `for-range` loops, there is only a single variable declared for the channel, which is the value. The loop continues until the channel is closed, or until a `break` or `return` statement is reached.

Closing a Channel

When you are done writing to a channel, you close it using the built-in `close` function:

```
close(ch)
```

Once a channel is closed, any attempts to write to the channel or close the channel again will panic. Interestingly, attempting to read from a closed channel always succeeds. If the channel is buffered and there are values that haven't been read yet, they will be returned in order. If the channel is unbuffered or the buffered channel has no more values, the zero value for the channel's type is returned.

```
close(out) // close channel
value, ok := <-in // always success
if ok {
    fmt.Println(value)
}
```

How Channels Behave

Channels have many different states, each with a different behavior when reading, writing, or closing. Use Table 10-1 to keep them straight.

Table 10-1. How channels behave

	Unbuffered, open	Unbuffered, closed	Buffered, open	Buffered, closed	Nil
Read	Pause until something is written	Return zero value (use comma ok to see if closed)	Pause if buffer is empty	Return a remaining value in the buffer. If the buffer is empty, return zero value (use comma ok to see if closed)	Hang forever
Write	Pause until something is read	**PANIC**	Pause if buffer is full	**PANIC**	Hang forever
Close	Works	**PANIC**	Works, remaining values still there	**PANIC**	**PANIC**

Select

```
select {
case v := <-ch:
    fmt.Println(v)
case v := <-ch2:
    fmt.Println(v)
case ch3 <- x:
    fmt.Println("wrote", x)
case <-ch4:
    fmt.Println("got value on ch4, but ignored it")
}
```

Deadlock code

```
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go func() {
        v := 1
        ch1 <- v // Add v to ch1
        v2 := <-ch2 // wait for ch2
        fmt.Println(v, v2)
    }()
    v := 2
    ch2 <- v // Add v to ch2
    v2 := <-ch1 // wait for 1
    fmt.Println(v, v2)
}

fatal error: all goroutines are asleep - deadlock!
```

Since `select` is responsible for communicating over a number of channels, it is often embedded within a `for` loop:

```
for {
    select {
    case <-done:
        return
    case v := <-ch:
        fmt.Println(v)
    }
}
```

Loops routine:

```
// Wrong code
func VaryInLoops(){
    a := []int {2, 4, 6, 8, 10}
    ch := make(chan int, len(a))
    for _, v := range a {
        go func() {
            ch <- v * 2
        }()
    }
    for i := 0; i < len(a); i++ {
        fmt.Println(<-ch)
    }
}

12
16
16
20
20
```

```
// Code fix

func VaryInLoopsFix(){
    a := []int {2, 4, 6, 8, 10}
    ch := make(chan int, len(a))
    for _, v := range a {
        go func(val int) {
            ch <- val * 2
        }(v)
    }
    for i := 0; i < len(a); i++ {
        fmt.Println(<-ch)
    }
}
4
8
12
16
20
```

Always Clean Up Your Goroutines

Whenever you launch a goroutine function, you must make sure that it will eventually exit. Unlike variables, the Go runtime can't detect that a goroutine will never be used again. If a goroutine doesn't exit, the scheduler will still periodically give it time to do nothing, which slows down your program. This is called a *goroutine leak*.

It may not be obvious that a goroutine isn't guaranteed to exit. For example, say you used a goroutine as a generator:

```
func countTo(max int) <-chan int {
    ch := make(chan int)
    go func() {
        for i := 0; i < max; i++ {
            ch <- i
        }
        close(ch)
    }()
    return ch
}

func main() {
    for i := range countTo(10) {
        fmt.Println(i)
    }
}
```

The done patterns:

```
func searchData(s string, searchers []func(string) []string) [] string {
    done := make(chan struct{})
    result := make(chan []string)

    for _, searcher := range searchers {
        go func(funS func(string) []string) {
            select {
                case result <- funS(s):
                case <- done:
            }
        }(searcher)
    }
    r := <- result
    close(done)
    return r
}
```

In our function, we declare a channel named `done` that contains data of type `struct{}`. We use an empty struct for the type because the value is unimportant; we never write to this channel, only close it. We launch a goroutine for each searcher passed in. The `select` statements in the worker goroutines wait for either a write on the `result` channel (when the `searcher` function returns) or a read on the `done` channel. Remember that a read on an open channel pauses until there is data available and that a read on a closed channel always returns the zero value for the channel. This means that the case that reads from `done` will stay paused until `done` is closed. In `searchData`, we read the first value written to `result`, and then we close `done`. This signals to the goroutines that they should exit, preventing them from leaking.

Using buffered channel:

```
func ProcessChannel(ch chan int) []int {
    const con = 5
    result := make(chan int, con)
    for i := 0 ; i < con ; i++ {
        go func() {
            v := <- ch
            result <- v*10
        }()
    }
    var out [] int
    for i := 0 ; i < con ; i++ {
        out = append(out, <-result)
    }
    return out
}

    input := make(chan int, 5)
    input <- 1
    input <- 2
    input <- 3
    input <- 4
    input <- 5

    fmt.Println(routine.ProcessChannel(input))
```

Test timout

```go
func TestTimeout() {
    c1 := make(chan int, 1)

    go func() {
        time.Sleep(6 * time.Second)
        c1 <- 10
    }()

    select {
        case res := <-c1:
            fmt.Println(res)
        case <-time.After(1 * time.Second):
            fmt.Println("timeout 1")
    }
}
```

WaitGroups:

```go
func WaitGroupTest() {
    wg := sync.WaitGroup{}
    wg.Add(2)

    go func() {
        time.Sleep(1 * time.Second)
        fmt.Println("Func 1")
        defer wg.Done()
    }()
    go func() {
        time.Sleep(2 * time.Second)
        fmt.Println("Func 2")
        defer wg.Done()
    }()

    wg.Wait()
}


func ProcessAndGather(num int, processor func(int) int) []int {
    var temp chan int
    temp = make(chan int, num)
    temp <- 10
    temp <- 20
    temp <- 30
    temp <- 40

    out := make(chan int, num)
    var wg sync.WaitGroup
    wg.Add(num)
    for i := 0; i < num; i++ {
        go func() {
            defer wg.Done()
            out <- processor(<- temp)
        }()
    }

    go func() {
        wg.Wait()
        close(out)
    }()

    var res []int
    for v := range out {
        res = append(res, v)
    }
    fmt.Println(res)
    return res
}
```

10. The standard lib

10.1 io

```
func CountLetter(r io.Reader) (map[string]int, error) {
    buf := make([]byte, 2048)
    out := map[string]int{}
    for {
        n, err := r.Read(buf)
        for _, b := range buf[:n] {
            if (b >= 'A' && b <= 'Z') || (b >= 'a' && b <= 'z') {
                out[string(b)] += 1
            }
        }

        if err == io.EOF {
            return out, nil
        }

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

}

func TestStringReader() {
    s := "Hello World I am X"
    count, err := CountLetter(strings.NewReader(s))
    if err == nil {
        fmt.Println("No err")
        fmt.Println(count)
    } else {
        fmt.Println("Err")
        fmt.Println(err)
    }
}
```

10.2 http

Http Client:

```go
func Client() {
    client := &http.Client{
        Timeout: 30*time.Second,
    }

    req, err := http.NewRequestWithContext(context.Background(),
                                           http.MethodGet,

"https://jsonplaceholder.typicode.com/todos/1",
                                           nil)
    req.Header.Add("X-My-Client", "Learning Go")
    res, err := client.Do(req)

    if err != nil {
        panic(err)
    }
    defer res.Body.Close()

    if res.StatusCode != http.StatusOK {
        panic(fmt.Sprintf("unexpected status: got %v", res.Status))
    }

    fmt.Println(res.Header.Get("Content-Type"))

    var data struct {UserID int `json:"userId"`
        ID int `json:"id"`
        Title string `json:"title"`
        Completed bool `json:"completed"`
    }
    err = json.NewDecoder(res.Body).Decode(&data)

    if err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", data)
}
```

11. Testing

```go
func AddNumbers(x, y int) int {
    return x + y
}

func TestAdding(t *testing.T) {
    result := addNumbers(2, 3)
    if result != 5 {
        t.Error("incorrect result", result)
    }
}


var testTime time.Time

func TestMain(m *testing.M) {
    fmt.Println("Set up stuff for tests here")
    testTime = time.Now()

    exitVal := m.Run()
    fmt.Println("Clean up stuff after tests here")
    os.Exit(exitVal)
}
```

12. Recipe

12.1 Panic and recovery

```go
func PanicRecovery (){
    defer fmt.Println("Defer call 1")
    defer func() {
        fmt.Println("Defer call 2")
        if e := recover(); e != nil {
            fmt.Println("Recovery Here")
        }
    }()

    panic("I just shat myself")
    fmt.Println("This code will never come here")
}

Defer call 2
Recovery Here
Defer call 1
```

12.2 Struct and Interface

```go
type Customer struct {
    FirstName string
    LastName string
    Email string
    Phone string
}

func (c Customer) ToString() string {
    return fmt.Sprintf("Customer: %s %s, Email:%s", c.FirstName, c.LastName, c.Email)
}

func (c *Customer) updatePhone(p string) {
    c.Phone = p // Change state of struct
}


var data Customer

func TestMain(m *testing.M) {
    data = Customer{
        FirstName: "D",
        LastName:  "Ng",
        Email:     "duy@gmail.com",
        Phone:     "0903858818",
    }

    exitVal := m.Run()
    os.Exit(exitVal)
}

func TestCustomer_greeting(t *testing.T) {
    fmt.Println(data.ToString())
}

func TestCustomer_updatePhone(t *testing.T) {
    data.updatePhone("1111")
    fmt.Println(data.ToString())
}
```

12.3 Channel

```go
func testChannel(){
    ch := make(chan int, 3)
    ch <- 10 // push to channel
    x := <- ch // pull from channel
    fmt.Println(x)
    close(ch)

    // Pull Close Channel will default = 0
    y := <- ch
    fmt.Println(y)

    // Push Close Channel will panic
    ch <- 5

    defer func() {
        fmt.Println("Push to close channel is invalid")
        if e := recover(); e != nil {
            log.Println("panic occurred:", e)
            fmt.Println("Push to close channel is invalid")
        }
    }()
}
```

Unbuffered channel:

When a send operation performs on an unbuffered channel from one goroutine, a corresponding receive operation must be executed on the same channel from **another goroutine** to complete the send operation

```go
func blockingCode(){
    // Declare an unbuffered channel
    counter := make(chan int)
    // This will create a deadlock
    counter <- 10 // Send operation to a channel from main goroutine
    fmt.Println(<-counter) // Receive operation from the channel
}


func nonBlockingCode(){
    counter := make (chan int)

    go func() {
        counter <- 10
    }()

    fmt.Println(<- counter)
}
```

Using the Output of One Goroutine as the Input of Another

```go
/* Pipeline */

type Pair struct{
    input int
    output float64
}

func generateCounter(out chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    var random int
    for i := 0; i < 10; i++ {
        random = rand.Intn(50)
        out <- random
    }
    close(out)
}

func powerCounter(out chan<- Pair, in <-chan int, wg *sync.WaitGroup) {
    defer wg.Done()

    for v := range in {
        out <- Pair {
            v,
            float64(v) * float64(v),
        }
    }
    close(out)
}

func displayCounter(in <- chan Pair, wg *sync.WaitGroup) {
    defer wg.Done()

    for v := range in {
        fmt.Printf("Receive [%.d] -> [%.2f] \n", v.input, v.output)
    }
}

func mainCounter(){
    var wg sync.WaitGroup
    wg.Add(3)

    input := make(chan int)
    output := make(chan Pair)
    go generateCounter(input, &wg)
    go powerCounter(output, input, &wg)
    go displayCounter(output, &wg)
    wg.Wait()

    fmt.Println("Done")
}
```

```
Receive [31] -> [961.00]
Receive [37] -> [1369.00]
Receive [47] -> [2209.00]
Receive [9] -> [81.00]
Receive [31] -> [961.00]
Receive [18] -> [324.00]
Receive [25] -> [625.00]
Receive [40] -> [1600.00]
Receive [6] -> [36.00]
Receive [] -> [0.00]
Done
```

Async with buffer channel:

A buffered channel can hold elements up to its capacity. If one goroutine makes a send operation on a buffered channel that exceeds its capacity, which means that the channel is full and trying to perform another send operation on the same channel, it blocks the sending goroutine until a space is available to insert a new element on the channel by a receive operation of another goroutine.

```go
type Task struct {
    Id int
    JobId int
    Status string
    CreatedOn time.Time
}

func (t *Task) run() {
    sleep := rand.Int63n(1000)
    // Delaying the execution for the sake of example
    time.Sleep(time.Duration(sleep) * time.Millisecond)
    t.Status = "Completed"
}


var wga sync.WaitGroup

func Async() {
    const noOfWorkers = 3
    wga.Add(noOfWorkers)
    taskQueue := make(chan *Task, 10)

    for i := 0; i < noOfWorkers; i++ {
        go Worker(taskQueue, i)
    }

    for i := 0; i < 10; i++ {
        taskQueue <- &Task{
            Id:        i,
            JobId:     100 + i,
            CreatedOn: time.Now(),
        }
    }
    close(taskQueue)
    wga.Wait()
}

func Worker(taskQueue <- chan *Task, workerId int) {
    defer wga.Done()

    for v := range taskQueue {
        fmt.Printf("Worker[%d] work on task [%d] - job [%d] \n", workerId, v.Id,
v.JobId)
        v.run()
        fmt.Printf("Worker%d: Status:%s for Task:%d - Job:%d \n", workerId, v.Status,
v.Id, v.JobId)
    }
}
```

A **select block** is written with multiple case statements that lets a goroutine wait until one of the cases can run; it then executes the code block of that case. If multiple case blocks are ready for execution, it randomly picks one of them and executes the code block of that case.

```go
type Stock struct {
    name string
    value float64
}


func generateHoseStock(out chan <- Stock, wg *sync.WaitGroup) {
    defer wg.Done()
    var charSet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    for i := 0; i < 10; i++ {
        b := make([]byte, 3)
        for j := 0; j < 3; j++ {
            b[j] = charSet[rand.Intn(len(charSet))]
        }
        out <- Stock{
            name:  string(b),
            value: rand.Float64(),
        }
    }
    close(out)
}

func generateHnxStock(out chan <- Stock, wg *sync.WaitGroup) {
    defer wg.Done()
    var charSet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    for i := 0; i < 10; i++ {
        b := make([]byte, 3)
        for j := 0; j < 3; j++ {
            b[j] = charSet[rand.Intn(len(charSet))]
        }
        out <- Stock{
            name:  string(b),
            value: rand.Float64(),
        }
    }
    close(out)
}

func displayStock(hoseIn <- chan Stock, hnxIn <- chan Stock, wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 1; i <= 20; i++ {
        select {
        case hose := <-hoseIn:
            fmt.Printf("[%d] Hose stock %s value %.2f \n", i, hose.name, hose.value)
        case hnx := <-hnxIn:
            fmt.Printf("[%d] Hnx stock %s value %.2f \n", i, hnx.name, hnx.value)
        }
    }
```

```
}

func selectStock(){
    var wg sync.WaitGroup
    wg.Add(3)

    hoseChan := make(chan Stock, 10)
    hnxChan := make(chan Stock, 10)

    go generateHnxStock(hnxChan, &wg)
    go generateHoseStock(hoseChan, &wg)
    go displayStock(hoseChan, hnxChan, &wg)

    wg.Wait()
}
```

12.4 Stand library package

```
func StructTag() {
    cus1 := User{
        ID:       "101",
        Password: "pass",
        Username: "duyntc",
    }

    cus2 := User{
        ID:       "",
        Password: "pass",
        Username: "duyntc",
    }

    str1, _ := json.Marshal(cus1)
    str2, _ := json.Marshal(cus2)
    fmt.Println(string(str1))
    fmt.Println(string(str2))

    // {"id":"101","usr":"duyntc"}
    // {"usr":"duyntc"}

    p := []byte(`{"id":"101","usr":"duyntc", "password": "12345"}`)
    var pObj User
    _ = json.Unmarshal(p, &pObj)
    fmt.Printf("%+v\n", pObj)
    // {ID:101 Password: Username:duyntc}
}
```