Golang Training 2 – Intermediate, Functional Mindset

Calvin Lei-Cramer

What are we covering today?

Defer, pointers and references, methods and receivers, interfaces, OOP, packages / modules, labels

Defer

- What is defer?
- executes a function call when function returns for any[1] reason

```go
func sayBye() { fmt.Println("Adios!") }

func main() {
    defer sayBye() // <-- defer!
    switch getFirstCliArgInt() {
    case 0:
        // Normal execution
    case 1:
        panic("Something bad happened!")
    default:
        os.Exit(1)
    }
}
```

```
$ ./defer-test 0
Adios!
$ ./defer-test 1
Adios!
panic: Something bad happened!

goroutine 1 [running]:
main.main()
        /home/ccramer-loc/repos/GOLANG_T
$ ./defer-test 2
$
```

[1] Except for exiting program with os.Exit(). Terms and conditions apply.

Defer

- **Idiomatic defer → first class functions (more on this later)**

```
1 func main() {
2     defer func() {
3         fmt.Println("Adios!")
4     }()
5     ...
6 }
```

Defer

- **Why use? → cleanup code (use with panic in next lesson)**

```
1 var logLock sync.Mutex
2
3 func main() {
4     logLock.Lock()
5     defer logLock.Unlock()
6     // Perform action that needs clea
7 }
```

```
1 func main() {
2     fp, err := os.OpenFile("myFile.txt", os.O_RDWR, 0666)
3     if err != nil {
4         return
5     }
6     defer fp.Close()
7     fp.Read() ...
8     fp.Write() ...
9 }
```

Defer

- **Defer call arguments?**

```go
func main() {
    x := 5
    defer func(num int) {
        fmt.Println(num) // 5
    }(x)
    x = 10
    fmt.Println(x) // 10
}
```

```
$ go run .
10
5
$
```

Defer

- **Multiple defers?**
- **Executes in stack / LIFO order**

```go
func printNum(n int) {
    println(n)
}

func main() {
    for i := range 5 {
        defer printNum(i)
        printNum(i)
    }
}
```

```
$ go run .
0
1
2
3
4
4
3
2
1
0
```

Defer

- **Named return values → able to modify return variables**
- **Will be helpful for later!**

```
1 var seed uint = 478993712093
2
3 func randomIndex(length uint) (idx uint) {
4     defer func() {
5         // Just in case, clamp to safe value
6         idx = max(idx, 0)
7         idx = min(idx, length-1)
8     }()
9     // We don't know about 'rand' package :(
10    seed = (1103515245 * seed) + 12345
11    idx = seed % length
12    return
13 }
```

Pointers and References

- ‘&’ and ‘*’ like C++

```go
func main() {
    var x, y int = 5, 10
    var numPtr *int = nil
    numPtr = &y
    *numPtr += 3
    numPtr = &x
    fmt.Printf(
      "%v\n%v\n%v\n%v\n",
      x, y, numPtr, *numPtr
    )
}
```

```
$ go run .
5
13
0xc000010100
5
$
```

Pointers and References

- **Pass by value, pass by reference?**
- **What types are passed by value, and by reference?**
  - By reference → pointers, functions, slice, map, channel
  - By value → arrays, structs, ints, floats, booleans, everything else

```go
type Foo struct {
    A int
    B *int
}

func modify(foo Foo) {
    foo.A = 100
    *(foo.B) = 100
}

func main() {
    num := 42
    foo := Foo{20, &num}
    fmt.Printf("{%d %v->%d}\n", foo.A, foo.B, *foo.B)
    modify(foo)
    fmt.Printf("{%d %v->%d}\n", foo.A, foo.B, *foo.B)
}
```

```
$ go run .
{20 0xc000010100->42}
{20 0xc000010100->100}
$
```

Pointers and References

- **No pointer arithmetic**
- **‘unsafe’ package if you dare**
- **Automatic dereferencing sugar → `(*x).f === x.f`**
  - Most of the time you don’t need to care if using pointer or raw value.

```
1 func modify(foo *Foo) {
2     foo.A = 100 // no sugar (*foo).A
3     *(foo.B) = 100 // no sugar *((*foo).B)
4 }
```

Methods and Receivers

```
type Movie struct {
    rating int
}
```

- What are methods? Example
  - And Receivers?
- Why it matters? → Binds the function to the type

Goal: compare two movies, return 1, 0, -1

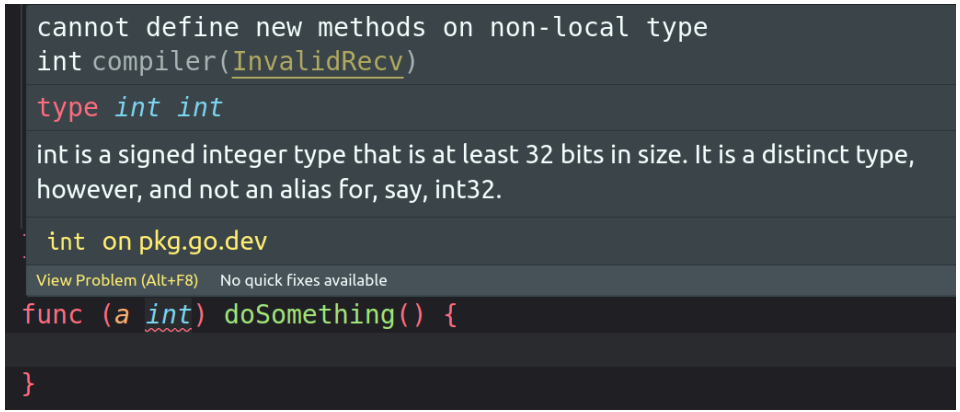
```
func compare1(m1, m2 *Movie) int {
    switch {
    case m1.rating > m2.rating: return 1
    case m1.rating == m2.rating: return 0
    default: return -1
    }
}
```

```
func main() {
    toyStory := &Movie{rating: 8}
    starWars := &Movie{rating: 7}
    fmt.Println(compare1(toyStory, starWars)) // 1
    fmt.Println(toyStory.compare2(starWars))  // 1
}
```

```
func (m1 *Movie) compare2(m2 *Movie) int {
    switch {
    case m1.rating > m2.rating: return 1
    case m1.rating == m2.rating: return 0
    default: return -1
    }
}
```

Methods and Receivers

- Limitation → can't add methods to external types
- Solution → type "alias"
- Type alias → `type Foo = Bar`
- New type → `type Foo Bar`




```go
type MyInt int

func (a MyInt) doSomething() MyInt {
    return a + 1
}

func main() {
    var foo MyInt = 42
    fmt.Println(foo.doSomething())
}

```

Methods and Receivers

- Difference between T and *T receivers
- Same thing as pass by value / reference from before
- Limitation → need to choose between T and *T for a single function. (No function overriding/overloading in Golang)

```
func (a MyInt) doSomething()  {}
func (a *MyInt) doSomething() {}
```

```
./main.go:37:17: method MyInt.doSomething already declared at ./main.go:33:16
```

Interfaces

- **What are interfaces?**
- **“interface type” → a set of function signatures**

```
1 type AudioPlayer interface {
2     LoadTrack(path string) error
3     Play() error
4     Pause() error
5 }
```

Interfaces

```
type AudioPlayer interface {
    LoadTrack(path string) error
    Play() error
    Pause() error
}
```

- How to implement / satisfy and interface (implicit vs explicit)
- “interface value” → any value that implements the functions

Java

```
class MP3Player {
  public error LoadTrack(string path) { ... }
  public error Play() { ... }
  public error Pause() { ... }
}
```

```
class MP3Player implements AudioPlayer {
  public error LoadTrack(string path) { ... }
  public error Play() { ... }
  public error Pause() { ... }
}
```

Golang

```
type MP3Player struct {
    audioFilePath string
}

func (player *MP3Player) LoadTrack(path string) error { ... }
func (player *MP3Player) Play() error                 { ... }
func (player *MP3Player) Pause() error                { ... }
```

Interfaces

Let’s actually use it now

```
 1 type AudioPlayer interface {
 2     LoadTrack(path string) error
 3     Play() error
 4     Pause() error
 5 }
 6
 7 type MP3Player struct {
 8     audioFilePath string
 9 }
10
11 func (player *MP3Player) LoadTrack(path string) error { ... }
12 func (player *MP3Player) Play() error                  { ... }
13 func (player *MP3Player) Pause() error                 { ... }
14
15 func main() {
16     var player AudioPlayer = &MP3Player{}
17     player.LoadTrack("/tmp/beatles.mp3")
18     player.Play()
19     player.Pause()
20 }
21
```

Interfaces

- Interface{} / any
- What type implements the 0 functions of interface{}?
- Everything satisfies it!
- See this a lot when working with ‘encoding/json’ package

```
1 func printIt(thing any) {
2     fmt.Println("%v", thing)
3 }
4
5 func main() {
6     var str any = "asdf"
7     var num any = int(42)
8     var float any = float32(3.14)
9     var obj any = &MyStruct{}
10    printIt(str)
11    printIt(num)
12    printIt(float)
13    printIt(obj)
14 }
15
```

Type Assertions

- Type assertions → from many to one
- `val.(T)`
- `newVal, ok := val.(T)`
- Why use? Need to satisfy type checker at compile time.

```go
func main() {
    var str any = "asdf"
    {
        foo, ok := str.(string)
        fmt.Println(foo, ok) // asdf true
    }
    {
        foo, ok := str.(int) // no panic
        fmt.Println(foo, ok) // 0 false
    }
}
```

Interfaces

- **Interface embedding → combine interfaces together**
- **Composability → reuse interfaces! Keep them as small as possible.**

```
type Swimmer interface {
    Swim()
}

type Runner interface {
    Run()
}

type Athlete interface {
    Swimmer
    Runner
}
```

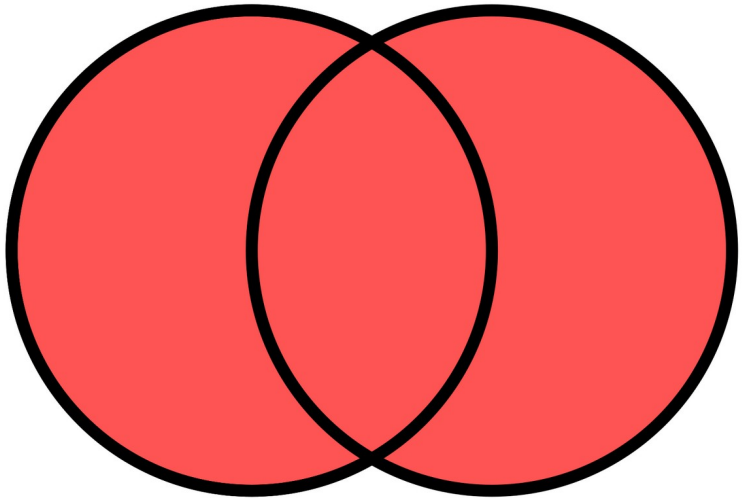
Interfaces

```
1 type NumberMethods interface {
2     ~int
3     IsPrime() bool
4     IsSquare() bool
5 }
```

- Approximation constraint ~T
- ~T = all types whose underlying type is T
- Interfaces can have arbitrary types too, not just functions!
- Useful to open up internal constraints to external use
- WARNING: non “basic” interfaces

Interfaces

- **Union type sets → this or that**
- **Non-basic interface → need to use in “type constraints”**



```
1 type MostlyAnyNumber interface {
2     ~int | ~float32 | ~float64
3 }
4
5 func add[T MostlyAnyNumber](a, b T) T {
6     return a + b
7 }
```

Covering generics later

OOP In Golang

- How do we get polymorphism? Normally with subclasses.

```
type ConfigParser interface {
    ParseConfigFile(path string) error
}
```

```
type ConfigV1 struct {
    colorScheme string
}

func (config *ConfigV1) ParseConfigFile(path string) error {
    ...
}
```

```
type ConfigV2 struct {
    colorScheme string
    uiDensity   float32
}

func (config *ConfigV2) ParseConfigFile(path string) error {
    ...
}
```

...

```
func doConfigParse(parser ConfigParser, path string) {
    if err := parser.ParseConfigFile(path); err != nil {
        panic(err)
    }
}
```

```
$ go run .;
Parsing v1
Parsing v2
$
```

OOP In Golang

- **Subtype polymorphism**
  - different behavior based on type
  - Interface vs class / subclass
- **Parametric polymorphism**
  - same function used for multiple types → generics (next lesson)
- **Ad-hoc polymorphism in Golang → no function overloading, no operating overloading**
- **Use struct and interface embedding to mimic inheritance of data**

OOP In Golang Example

- **Base class**
  - Data
  - 2 functions
- **Child class**
  - Has base data and functions
  - Modifies base data
  - Overrides 2 parent functions
  - Calls parent function

```python
class Player:
    def __init__(self, maxHP, attack, defense):
        self.__maxHP = maxHP
        self.__attack = attack
        self.__defense = defense
        self.__pos = (0, 0)

    def attack(self, other: Player):
        pass

    def move(self, newPos):
        pass


class Wizard(Player):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__attack *= 0.8

    def attack(self, other: Player):
        super().attack(other)
        pass

    def heal(self, target: Player):
        pass
```

OOP In Golang Example

Constructor idiom:
`func NewThing(…) *Thing`

Functions

```
type AttackerI interface {
    Attack(other PlayerI)
}

type MoveableI interface {
    Move(newPos []int)
}

type PlayerI interface {
    AttackerI
    MoveableI
}
```

Data

```
type Player struct {
    MaxHP     int
    AttackPts int
    Defense   int
    Pos       []int
}

type Wizard struct {
    Player
    // Wizard extra data...
}
```

Constructors

```
func NewPlayer(maxHP int, attack int, defense int) *Player {
    return &Player{
      MaxHP: maxHP,
      AttackPts: attack,
      Defense: defense,
      Pos: []int{0, 0},
    }
}

func NewWizard(maxHP int, attack int, defense int) *Wizard {
    temp := &Wizard{
      Player: *NewPlayer(maxHP, attack, defense)
    }
    temp.AttackPts = int(float64(temp.AttackPts) * 0.8)
    return temp
}
```

Implementation (stub)

```
func (p *Player) Attack(other PlayerI) {}
func (p *Player) Move(newPost []int)   {}

func (w *Wizard) Attack(other PlayerI) {}
func (w *Wizard) Move(newPost []int)   {}
```

```
func main() {
    p := NewPlayer(10, 10, 10)
    w := NewWizard(10, 10, 10)

    p.Attack(w)
    p.Move([]int{20, 20})

    w.Attack(p)
    w.Move([]int{20, 20})
}
```

Packages / Modules

- Package → collection of go file in same directory
- Module → collection of packages with go.mod file at root
- Workspace → work with multiple modules at same time with go.work file
- package <packageName> ← must be first line in every file
- Public / private things

```
1 package myMathFuncs
2
3 func Sqrt(n float64) float64
4
5 func secretAlgo(n float64) float64
```

Packages / Modules

<root>/go.mod

```
1 module myModName
2
3 go 1.24.3
```

<root>/mypkg/stuff.go

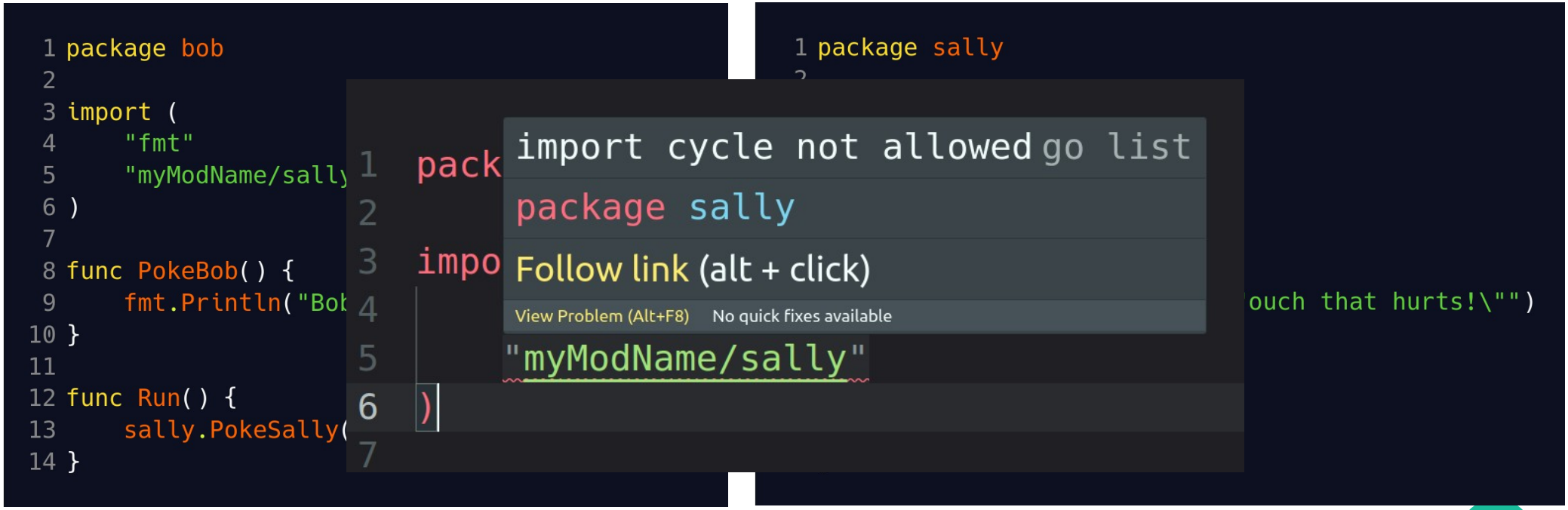
```
 1 package mypkg
 2
 3 func ExportedFunc() {}
 4 func privateFunc()  {}
 5
 6 type ExportedStruct struct {
 7     ExportedMember int
 8     privateMember  int
 9 }
10 type privateStruct struct{}
11
12 type ExportedInterface interface {
13     Action1()
14     action2()
15     // ^ makes interface un-implementable
16     // by outside world
17 }
18
19 type privateInterface interface {
20     Action3()
21     action4()
22 }
```

<root>/usepkg/use.go

```
 1 package use
 2
 3 import (
 4     "fmt"
 5     "myModName/mypkg"
 6 )
 7
 8 func DoSomething(foo mypkg.ExportedInterface) {}
 9
10 type localStruct struct{}
11
12 func (ls *localStruct) Action1() {}
13 func (ls *localStruct) action2() {}
14 // ^ does not satisfy ExportedInterface
15
16 func Run() {
17     mypkg.ExportedFunc()
18     foo := mypkg.ExportedStruct{ExportedMember: 500}
19     fmt.Println(foo)
20     ls := &localStruct{}
21     DoSomething(ls) // error
22 }
```

Packages / Modules

- Circular import? Better than python?



Packages / Modules etc

- **Create module →** `go mod init <module name>`
- **Single file no module →** `go run <file>`
- `func init() {…}` **→ called once when package in initialized**
- **Install third party module → `go get github.com/google/uuid`**
  - Automatically updates go.mod
- **Not covered → publishing modules, versioning, mod tidy**

Jumping to Labels

- **goto label, break label, continue label**
- **Break to label works with ‘for’, ‘switch’, ‘select’**
- **Goto label (and Golang rules for goto)**
  - Goto is considered less harmful in Golang
  - CANNOT jump to a place where new variables are in scope
  - CANNOT jump into a scope the ‘goto’ is not apart of
  - Therefore CANNOT jump outside of function

Jumping to Labels

```
1 var table [][]int = ...
2
3 func main() {
4 outerLoop:
5     for y := 0; y < len(table); y++ {
6     innerLoop:
7         for x := 0; x < len(table[y]); x++ {
8             if table[y][x] > 500 {
9                 // skip the rest of the row
10                break innerLoop
11            } else if table[y][x] < 0 {
12                // skip the rest of the table
13                break outerLoop
14            }
15        }
16    }
17 }
```

Practice!

- https://github.com/calvincramer/golang-training-2

Golang Training 2 – Intermediate, Functional Mindset

Calvin Lei-Cramer

What are we covering today?

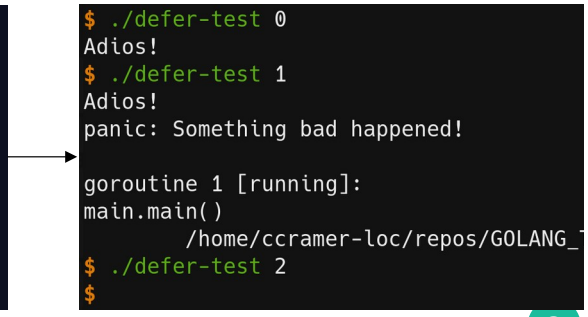
Defer, pointers and references, methods and receivers, interfaces, OOP, packages / modules, labels

Defer

- What is defer?
- executes a function call when function returns for any[1] reason

```go
func sayBye() { fmt.Println("Adios!") }

func main() {
    defer sayBye() // <-- defer!
    switch getFirstCliArgInt() {
    case 0:
        // Normal execution
    case 1:
        panic("Something bad happened!")
    default:
        os.Exit(1)
    }
}
```



```
$ ./defer-test 0
Adios!
$ ./defer-test 1
Adios!
panic: Something bad happened!

goroutine 1 [running]:
main.main()
        /home/ccramer-loc/repos/GOLANG_
$ ./defer-test 2
$
```

[1] Except for exiting program with os.Exit(). Terms and conditions apply.

Defer

- **Idiomatic defer → first class functions (more on this later)**

```
1 func main() {
2     defer func() {
3         fmt.Println("Adios!")
4     }()
5     ...
6 }
```

Defer

- **Why use? → cleanup code (use with panic in next lesson)**

```go
var logLock sync.Mutex

func main() {
    logLock.Lock()
    defer logLock.Unlock()
    // Perform action that needs clea
}
```

```go
func main() {
    fp, err := os.OpenFile("myFile.txt", os.O_RDWR, 0666)
    if err != nil {
        return
    }
    defer fp.Close()
    fp.Read() ...
    fp.Write() ...
}
```

Defer

- **Defer call arguments?**

```go
func main() {
    x := 5
    defer func(num int) {
        fmt.Println(num) // 5
    }(x)
    x = 10
    fmt.Println(x) // 10
}
```

```
$ go run .
10
5
$
```

Defer

- **Multiple defers?**
- **Executes in stack / LIFO order**

```go
func printNum(n int) {
    println(n)
}

func main() {
    for i := range 5 {
        defer printNum(i)
        printNum(i)
    }
}
```

```
$ go run .
0
1
2
3
4
4
3
2
1
0
```

Defer

- Named return values → able to modify return variables
- Will be helpful for later!

```go
var seed uint = 478993712093

func randomIndex(length uint) (idx uint) {
    defer func() {
        // Just in case, clamp to safe value
        idx = max(idx, 0)
        idx = min(idx, length-1)
    }()
    // We don't know about 'rand' package :(
    seed = (1103515245 * seed) + 12345
    idx = seed % length
    return
}
```

Pointers and References

- ‘&’ and ‘*’ like C++

```go
func main() {
    var x, y int = 5, 10
    var numPtr *int = nil
    numPtr = &y
    *numPtr += 3
    numPtr = &x
    fmt.Printf(
      "%v\n%v\n%v\n%v\n",
      x, y, numPtr, *numPtr
    )
}
```

```
$ go run .
5
13
0xc000010100
5
$
```

Pointers and References

- **Pass by value, pass by reference?**
- **What types are passed by value, and by reference?**
  - By reference → pointers, functions, slice, map, channel
  - By value → arrays, structs, ints, floats, booleans, everything else

```
1 type Foo struct {
2     A int
3     B *int
4 }
5
6 func modify(foo Foo) {
7     foo.A = 100
8     *(foo.B) = 100
9 }
10
11 func main() {
12     num := 42
13     foo := Foo{20, &num}
14     fmt.Printf("{%d %v->%d}\n", foo.A, foo.B, *foo.B)
15     modify(foo)
16     fmt.Printf("{%d %v->%d}\n", foo.A, foo.B, *foo.B)
17 }
18
```

```
$ go run .
{20 0xc000010100->42}
{20 0xc000010100->100}
$
```

10

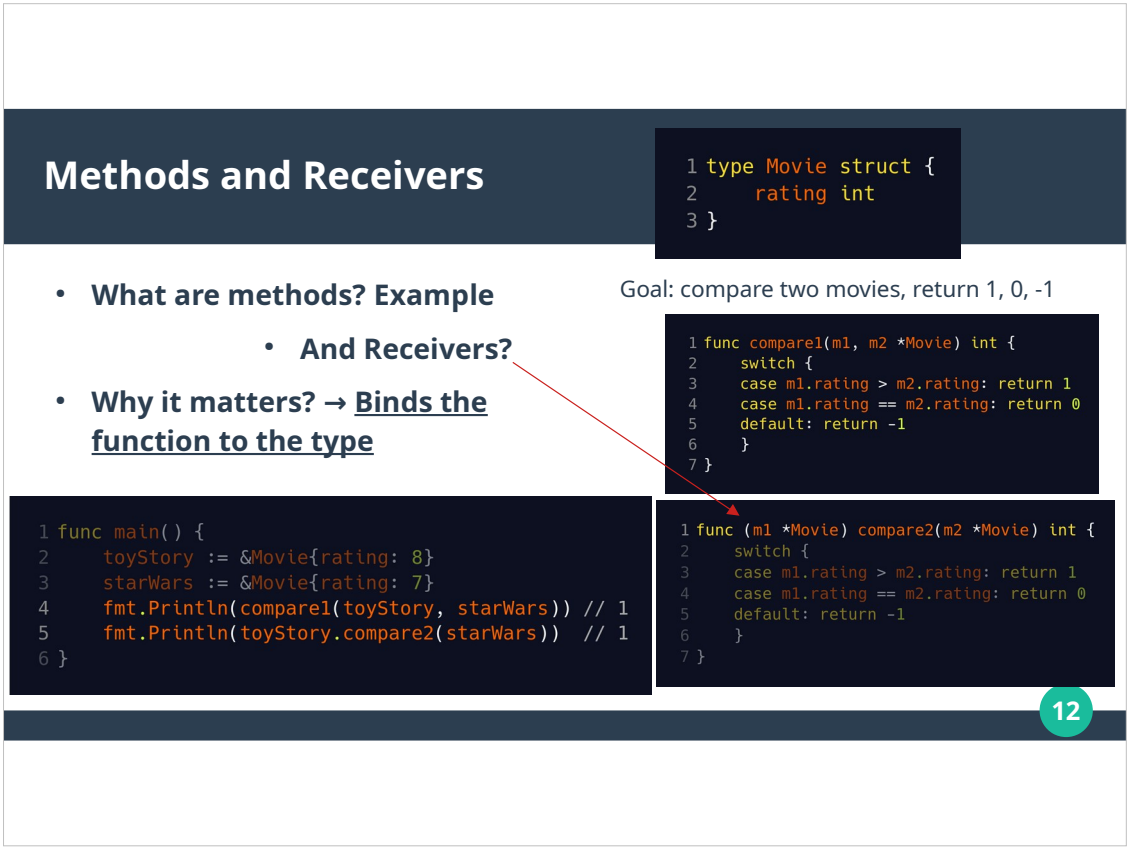
Technically everything is pass by value, since pointer addresses are passed by value.

SOME THINGS ARE AUTOMATICALLY PASSED AS POINTERS. SOME THINGS ARE AUTOMATICALLY PASSED AS VALUES (COPIED)

Pointers and References

- **No pointer arithmetic**
- **‘unsafe’ package if you dare**
- **Automatic dereferencing sugar → `(*x).f === x.f`**
  - Most of the time you don’t need to care if using pointer or raw value.

```
1 func modify(foo *Foo) {
2     foo.A = 100 // no sugar (*foo).A
3     *(foo.B) = 100 // no sugar *((*foo).B)
4 }
```

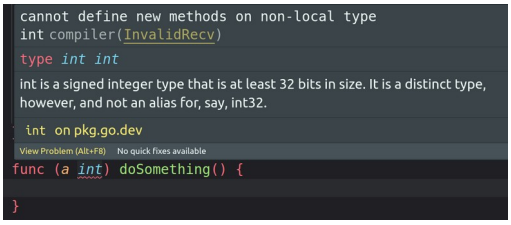


A method is just a function with a receiver

A receiver is simply the first argument.

Methods and Receivers

- Limitation → can't add methods to external types
- Solution → type "alias"
- Type alias → `type Foo = Bar`
- New type → `type Foo Bar`




```go
type MyInt int

func (a MyInt) doSomething() MyInt {
    return a + 1
}

func main() {
    var foo MyInt = 42
    fmt.Println(foo.doSomething())
}

```

Methods and Receivers

- **Difference between T and *T receivers**
- **Same thing as pass by value / reference from before**
- **Limitation → need to choose between T and *T for a single function. (No function overriding/overloading in Golang)**

```
func (a MyInt) doSomething()  {}
func (a *MyInt) doSomething() {}
```

```
./main.go:37:17: method MyInt.doSomething already declared at ./main.go:33:16
```


Recommend to always use pointer receivers, unless you've got a really good reason otherwise.

Interfaces

- What are interfaces?
- “interface type” → a set of function signatures

```
1 type AudioPlayer interface {
2     LoadTrack(path string) error
3     Play() error
4     Pause() error
5 }
```

Interfaces

```
type AudioPlayer interface {
    LoadTrack(path string) error
    Play() error
    Pause() error
}
```

- How to implement / satisfy and interface (implicit vs explicit)
- "interface value" → any value that implements the functions

Java

```
class MP3Player {
  public error LoadTrack(string path) { ... }
  public error Play() { ... }
  public error Pause() { ... }
}
```

```
class MP3Player implements AudioPlayer {
  public error LoadTrack(string path) { ... }
  public error Play() { ... }
  public error Pause() { ... }
}
```

Golang

```
type MP3Player struct {
    audioFilePath string
}

func (player *MP3Player) LoadTrack(path string) error { ... }
func (player *MP3Player) Play() error                 { ... }
func (player *MP3Player) Pause() error                { ... }
```

Interfaces

Let’s actually use it now

```
1 type AudioPlayer interface {
2     LoadTrack(path string) error
3     Play() error
4     Pause() error
5 }
6
7 type MP3Player struct {
8     audioFilePath string
9 }
10
11 func (player *MP3Player) LoadTrack(path string) error { ... }
12 func (player *MP3Player) Play() error                  { ... }
13 func (player *MP3Player) Pause() error                 { ... }
14
15 func main() {
16     var player AudioPlayer = &MP3Player{}
17     player.LoadTrack("/tmp/beatles.mp3")
18     player.Play()
19     player.Pause()
20 }
21
```

Interfaces

- Interface{} / any
- What type implements the 0 functions of interface{}?
- Everything satisfies it!
- See this a lot when working with ‘encoding/json’ package

```go
func printIt(thing any) {
    fmt.Println("%v", thing)
}

func main() {
    var str any = "asdf"
    var num any = int(42)
    var float any = float32(3.14)
    var obj any = &MyStruct{}
    printIt(str)
    printIt(num)
    printIt(float)
    printIt(obj)
}

```

Type Assertions

- **Type assertions → from many to one**
- **`val.(T)`**
- **`newVal, ok := val.(T)`**
- **Why use? Need to satisfy type checker at compile time.**

```go
func main() {
    var str any = "asdf"
    {
        foo, ok := str.(string)
        fmt.Println(foo, ok) // asdf true
    }
    {
        foo, ok := str.(int) // no panic
        fmt.Println(foo, ok) // 0 false
    }
}
```

19

Type assertions check that the value is of type T.
Fails if value is nil!
How does Go know if val is of type T? Usually it means that val implements T interface.
Panic if fails! Or not if accept OK arg.
NewVal will be zero value of type on failure.

Interfaces

- Interface embedding → combine interfaces together
- Composability → reuse interfaces! Keep them as small as possible.

```
type Swimmer interface {
    Swim()
}

type Runner interface {
    Run()
}

type Athlete interface {
    Swimmer
    Runner
}
```

Interfaces

```
type NumberMethods interface {
    ~int
    IsPrime() bool
    IsSquare() bool
}
```

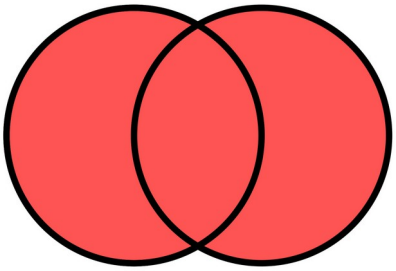
- Approximation constraint ~T
- ~T = all types whose underlying type is T
- Interfaces can have arbitrary types too, not just functions!
- Useful to open up internal constraints to external use
- WARNING: non “basic” interfaces

Interfaces are considered “not basic” if they contain something else other than function definitions.

Non-basic interface have RESTRICTIONS. Non-basic interfaces need to be used in type constraints (will see more of this in generics).

Interfaces

- **Union type sets → this or that**
- **Non-basic interface → need to use in “type constraints”**

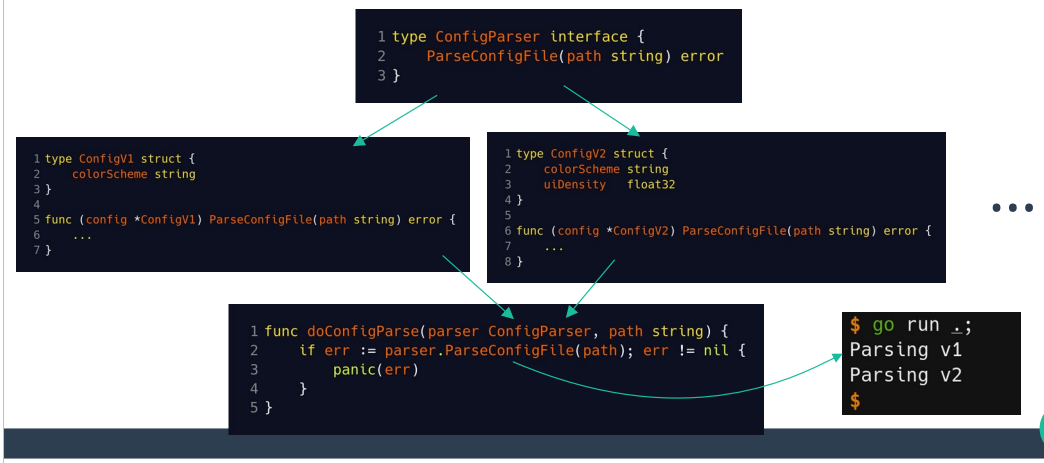


```
1 type MostlyAnyNumber interface {
2     ~int | ~float32 | ~float64
3 }
4
5 func add[T MostlyAnyNumber](a, b T) T {
6     return a + b
7 }
```

Covering generics later

OOP In Golang

- **How do we get polymorphism? Normally with subclasses.**



Achieve polymorphism by using interfaces where underlying type can be anything that satisfies the interface.

Golang achieves polymorphism is different ways.

Golang's polymorphism looks different than class-based polymorphism.

Subtype polymorphism is achieved with interfaces and multiple implementations.

Parametric polymorphism is achieved with generics which we'll cover later.

Use struct and interface embedding to "inherit" data.

OOP In Golang Example

- **Base class**
  - Data
  - 2 functions
- **Child class**
  - Has base data and functions
  - Modifies base data
  - Overrides 2 parent functions
  - Calls parent function

```python
class Player:
    def __init__(self, maxHP, attack, defense):
        self.__maxHP = maxHP
        self.__attack = attack
        self.__defense = defense
        self.__pos = (0, 0)

    def attack(self, other: Player):
        pass

    def move(self, newPos):
        pass


class Wizard(Player):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__attack *= 0.8

    def attack(self, other: Player):
        super().attack(other)
        pass

    def heal(self, target: Player):
        pass
```

OOP In Golang Example

Constructor idiom:
`func NewThing(…) *Thing`

Functions

```go
type AttackerI interface {
    Attack(other PlayerI)
}

type MoveableI interface {
    Move(newPos []int)
}

type PlayerI interface {
    AttackerI
    MoveableI
}
```

Data

```go
type Player struct {
    MaxHP     int
    AttackPts int
    Defense   int
    Pos       []int
}

type Wizard struct {
    Player
    // Wizard extra data...
}
```

Constructors

```go
func NewPlayer(maxHP int, attack int, defense int) *Player {
    return &Player{
      MaxHP: maxHP,
      AttackPts: attack,
      Defense: defense,
      Pos: []int{0, 0},
    }
}

func NewWizard(maxHP int, attack int, defense int) *Wizard {
    temp := &Wizard{
      Player: *NewPlayer(maxHP, attack, defense)
    }
    temp.AttackPts = int(float64(temp.AttackPts) * 0.8)
    return temp
}
```

Implementation (stub)

```go
func (p *Player) Attack(other PlayerI) {}
func (p *Player) Move(newPost []int)   {}

func (w *Wizard) Attack(other PlayerI) {}
func (w *Wizard) Move(newPost []int)   {}
```

```go
func main() {
    p := NewPlayer(10, 10, 10)
    w := NewWizard(10, 10, 10)

    p.Attack(w)
    p.Move([]int{20, 20})

    w.Attack(p)
    w.Move([]int{20, 20})
}
```

Packages / Modules

- Package → collection of go file in same directory
- Module → collection of packages with go.mod file at root
- Workspace → work with multiple modules at same time with go.work file
- package <packageName> ← must be first line in every file
- Public / private things

```
1 package myMathFuncs
2
3 func Sqrt(n float64) float64
4
5 func secretAlgo(n float64) float64
```

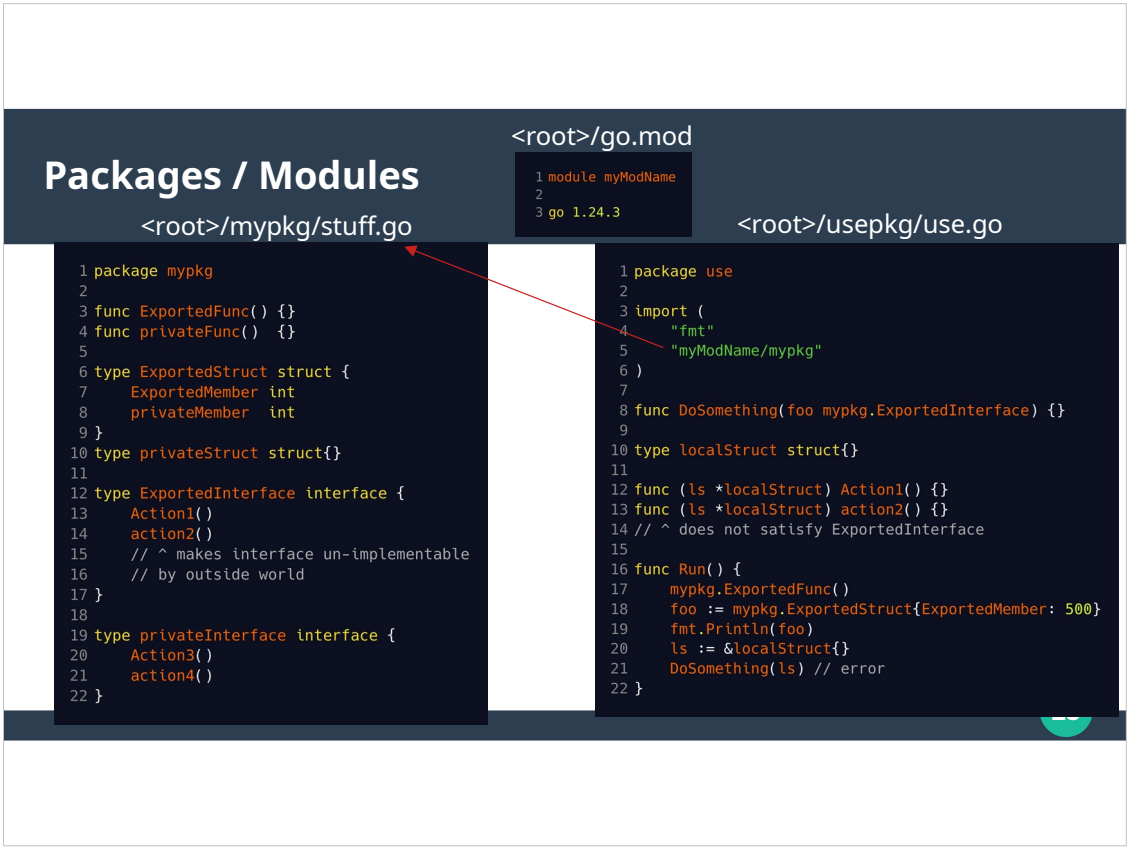
27

Public and private are at the PACKAGE level

Within a package, everything in that package is available to itself.

Public functions, structs, struct fields, interfaces, interface methods START WITH CAPITAL LETTER

Everything else is “private” to the package



Exported interfaces should always have each function be exported too.

Packages / Modules

- Circular import? Better than python?

Compile time error! A little better than python

Packages / Modules etc

- **Create module →** `go mod init <module name>`
- **Single file no module →** `go run <file>`
- `func init() {…}` **→ called once when package in initialized**
- **Install third party module → `go get github.com/google/uuid`**
  - Automatically updates go.mod
- **Not covered → publishing modules, versioning, mod tidy**

Jumping to Labels

- **goto label, break label, continue label**
- **Break to label works with ‘for’, ‘switch’, ‘select’**
- **Goto label (and Golang rules for goto)**
  - Goto is considered less harmful in Golang
  - CANNOT jump to a place where new variables are in scope
  - CANNOT jump into a scope the ‘goto’ is not apart of
  - Therefore CANNOT jump outside of function

Jumping to Labels

```
1 var table [][]int = ...
2
3 func main() {
4 outerLoop:
5     for y := 0; y < len(table); y++ {
6     innerLoop:
7         for x := 0; x < len(table[y]); x++ {
8             if table[y][x] > 500 {
9                 // skip the rest of the row
10                break innerLoop
11            } else if table[y][x] < 0 {
12                // skip the rest of the table
13                break outerLoop
14            }
15        }
16    }
17 }
```


Example for break

Practice!

- https://github.com/calvincramer/golang-training-2