alterra academy

String, Advance Function, Pointer, Struct, Method & Interface

OUR RULES

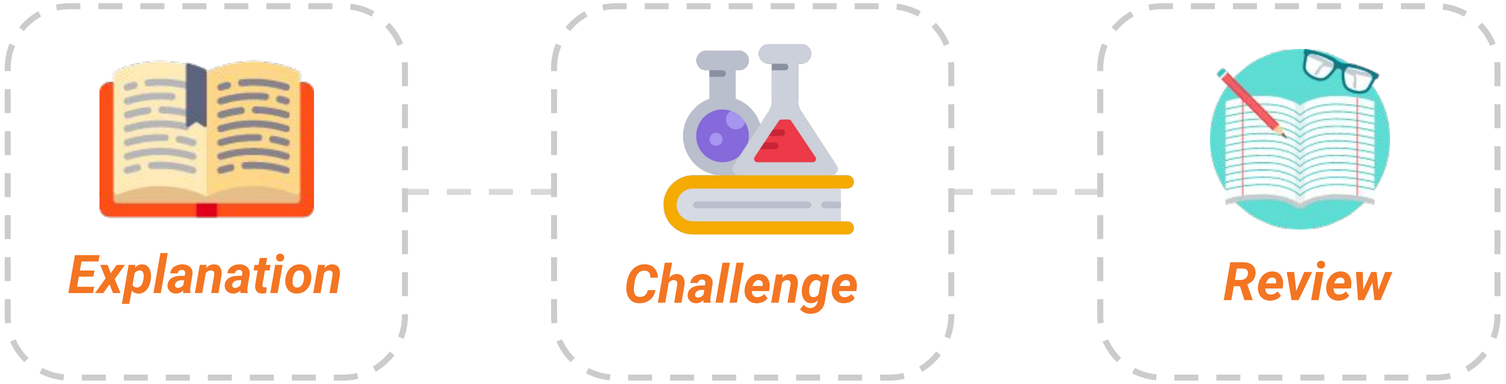
Silent Mode

Ask Question

Go Toilet

alterra academy

TIME ALLOCATION



OUTLINE

- String
- Advance Function
  - Variadic Function
  - Anonymous Function
  - Closure
  - Defer Function
- Pointer
- Package
- Error Handling

alterra academy

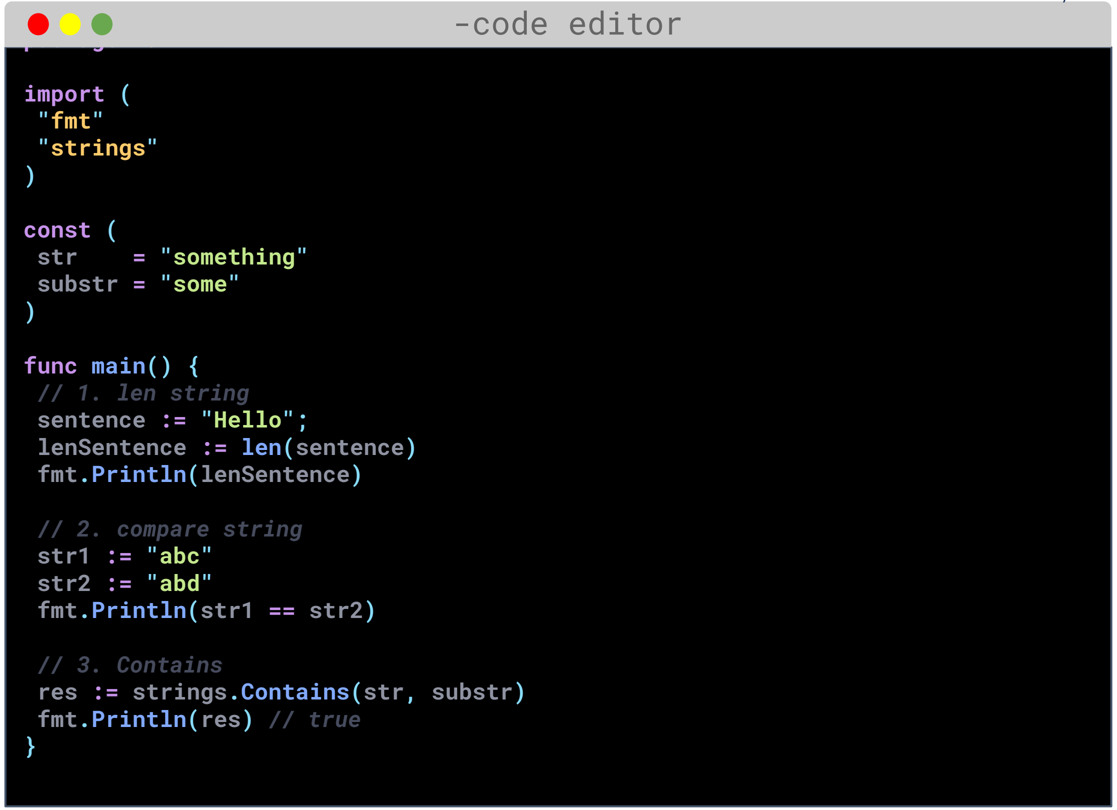
Let's get started!

alterra academy

String

Working with string

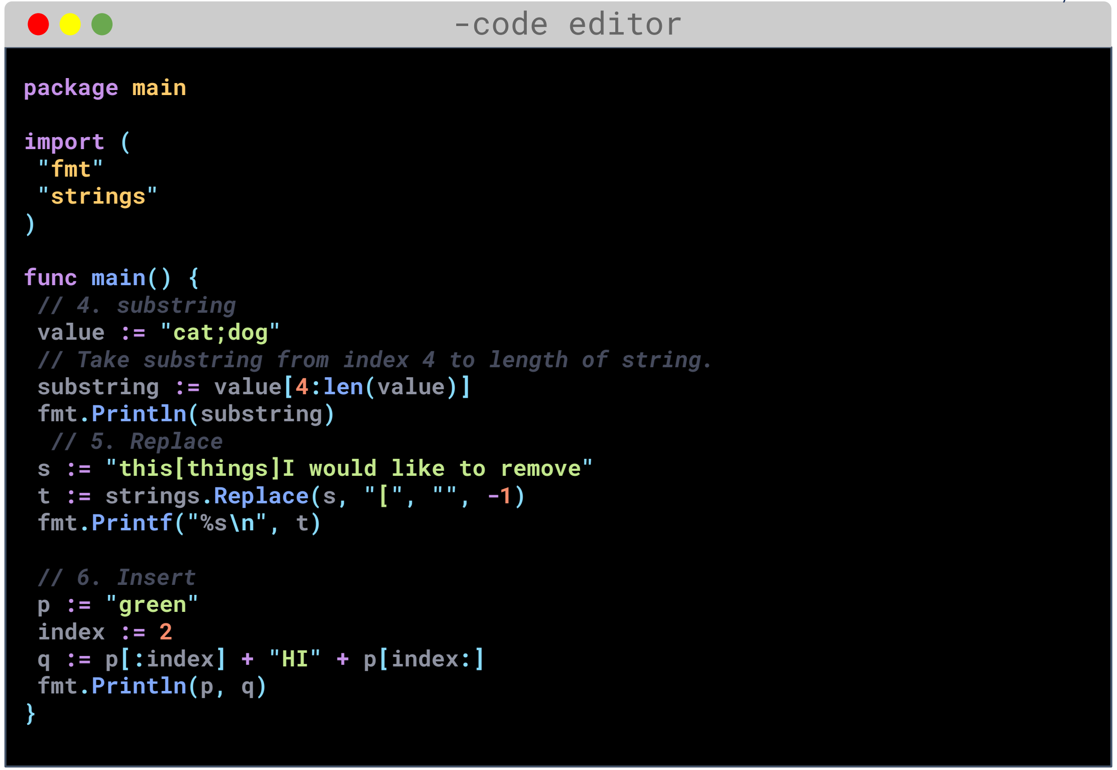
- Len
- Compare
- Contains



```go
import (
 "fmt"
 "strings"
)

const (
 str    = "something"
 substr = "some"
)

func main() {
 // 1. len string
 sentence := "Hello";
 lenSentence := len(sentence)
 fmt.Println(lenSentence)

 // 2. compare string
 str1 := "abc"
 str2 := "abd"
 fmt.Println(str1 == str2)

 // 3. Contains
 res := strings.Contains(str, substr)
 fmt.Println(res) // true
}
```

Working with string

- Substring
- Replace
- Insert


-code editor

```go
package main

import (
 "fmt"
 "strings"
)

func main() {
 // 4. substring
 value := "cat;dog"
 // Take substring from index 4 to length of string.
 substring := value[4:len(value)]
 fmt.Println(substring)
  // 5. Replace
 s := "this[things]I would like to remove"
 t := strings.Replace(s, "[", "", -1)
 fmt.Printf("%s\n", t)

 // 6. Insert
 p := "green"
 index := 2
 q := p[:index] + "HI" + p[index:]
 fmt.Println(p, q)
}
```

alterra academy

f(x) more function

VARIADIC FUNCTION

- To skip creating a temporary slice just to pass to a func
- When the number of input params are unknown
- To express your intent to increase the readability

alterra academy

```go
package main

import (
  "fmt"
)

func sum(numbers ...int) int {

  var total int = 0
  for _, number := range numbers {
    total += number
  }
  return total
}

func main() {
  avg := sum(2, 4, 3, 5)
  fmt.Println(avg)
}
```

variadic

slice

ANONYMOUS FUNCTION == LITERAL FUNCTION

An anonymous function is a function which doesn’t contain any name. It is useful when you want to create an inline function.

```go
package main

import "fmt"

func main() {
  // Anonymous function
  func() {
    fmt.Println("Welcome! to GeeksforGeeks")
  }()

  // Assigning an anonymous function to a variable
  value := func() {
    fmt.Println("Welcome! to GeeksforGeeks")
  }
  value()

  // Passing arguments in anonymous function
  func(sentence string) {
    fmt.Println(sentence)
  }("GeeksforGeeks")
}
```

CLOSURE

A closure is a special type of anonymous function **that references variables declared outside of the function itself**. In this case we will be using variables that weren’t passed into the function as a parameter, but instead were available when the function was declared.

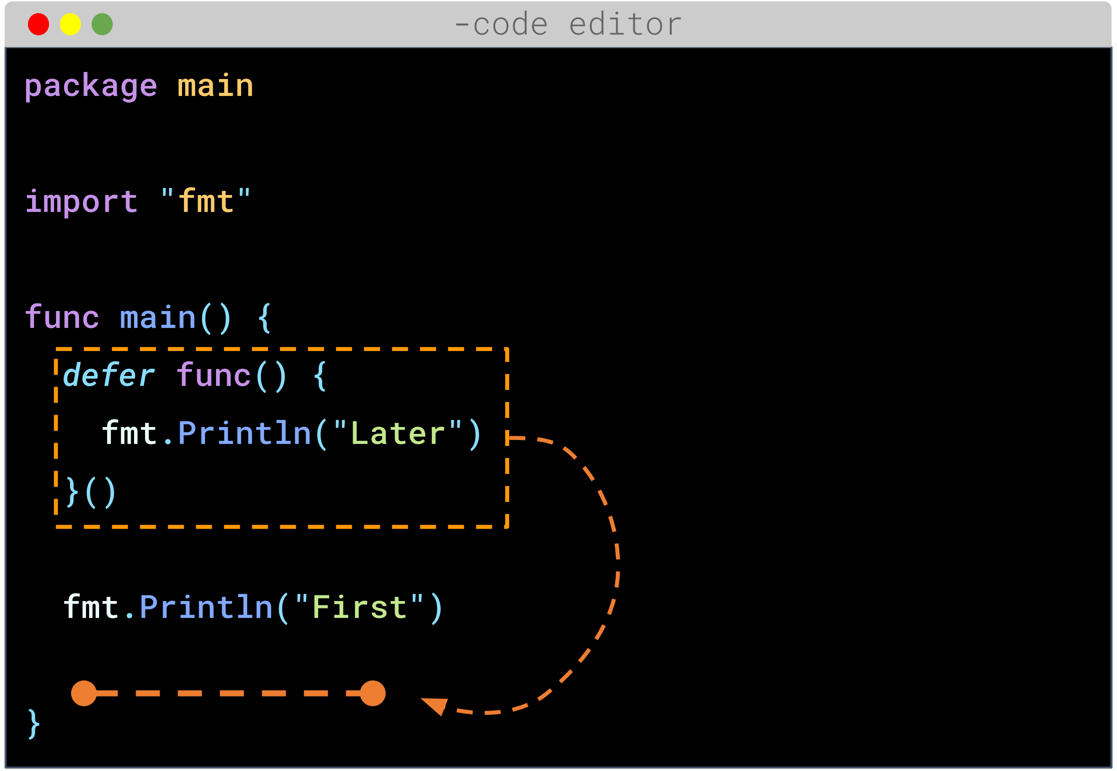
```go
package main

import "fmt"

func newCounter() func() int {
  count := 0
  return func() int {
    count += 1
    return count
  }
}

func main() {
  counter := newCounter()
  fmt.Println(counter())
  fmt.Println(counter())
}
```

Closures provide data isolation

A DEFERRED FUNCTION FUNC IS ONLY EXECUTED AFTER ITS PARENT FUNC RETURN. MULTIPLE RETURN CAN BE USED AS WELL, THEY RUN AS A STACK, ONE BY ONE.

DEFER FUNCTION

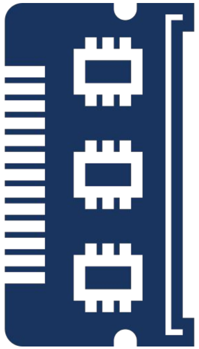
alterra academy



```go
package main

import "fmt"

func main() {
  defer func() {
    fmt.Println("Later")
  }()

  fmt.Println("First")

}
```

* Pointer

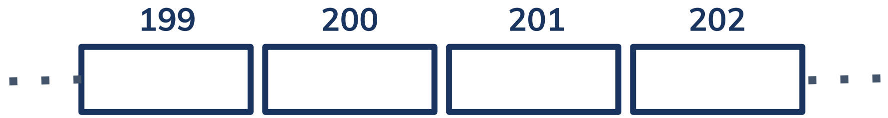
WHAT IS POINTER?



Pointer is a **variable that stores the memory address** of another variable. Pointers have the power to mutate data they are pointing.

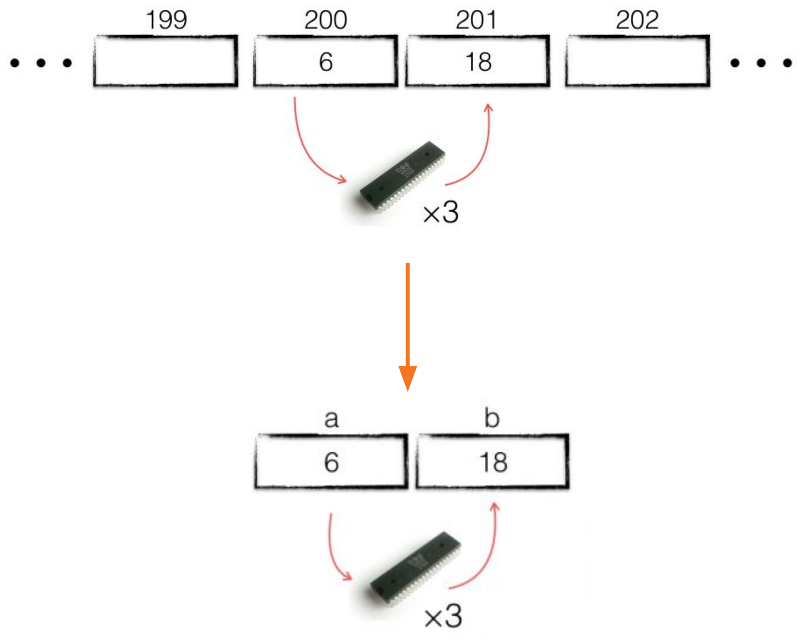
WHAT IS MEMORY?

Sequence of boxes, placed one after another in a line.

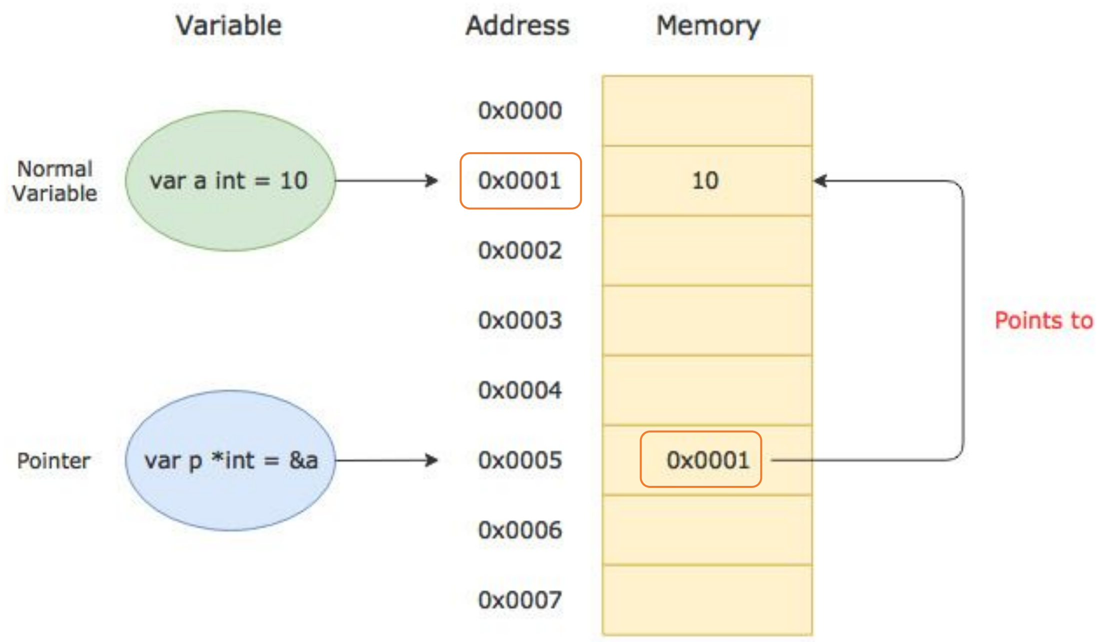


Variable & Memory

```
var a = 6
var b = a * 3
```



VARIABLE AND POINTER



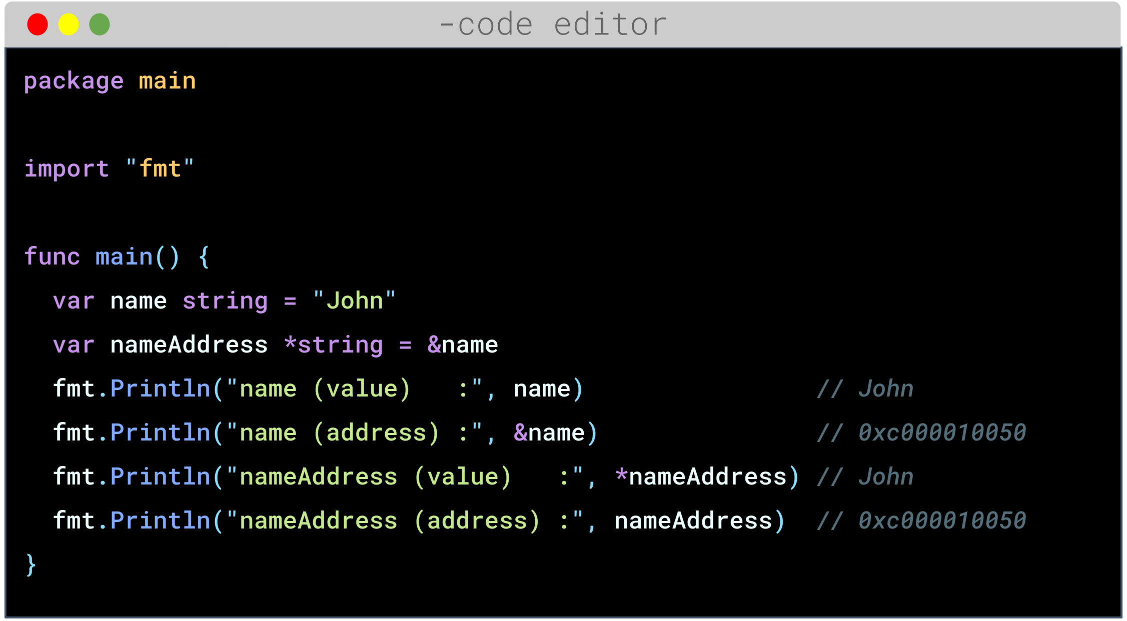
POINTER DECLARATION

Declaration : var <variable_name> *<variable_type>

```
└── var nameAddress *string
```

Use : var <variable_name> *<variable_type>

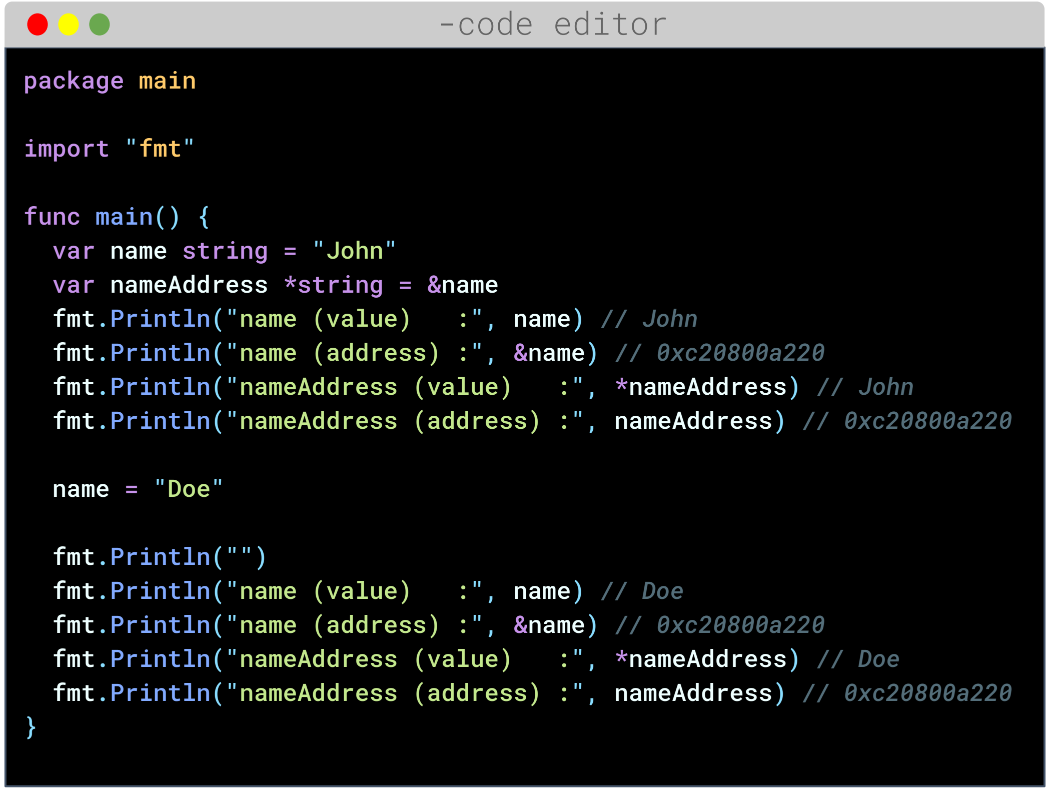
```
└──     var name = “John”
        var nameAddress *string
        nameAddress = &name
```



-code editor

```go
package main

import "fmt"

func main() {
  var name string = "John"
  var nameAddress *string = &name
  fmt.Println("name (value)    :", name)                  // John
  fmt.Println("name (address) :", &name)                  // 0xc000010050
  fmt.Println("nameAddress (value)    :", *nameAddress) // John
  fmt.Println("nameAddress (address) :", nameAddress)   // 0xc000010050
}
```

Changes in variables with same memory reference will affects one to another



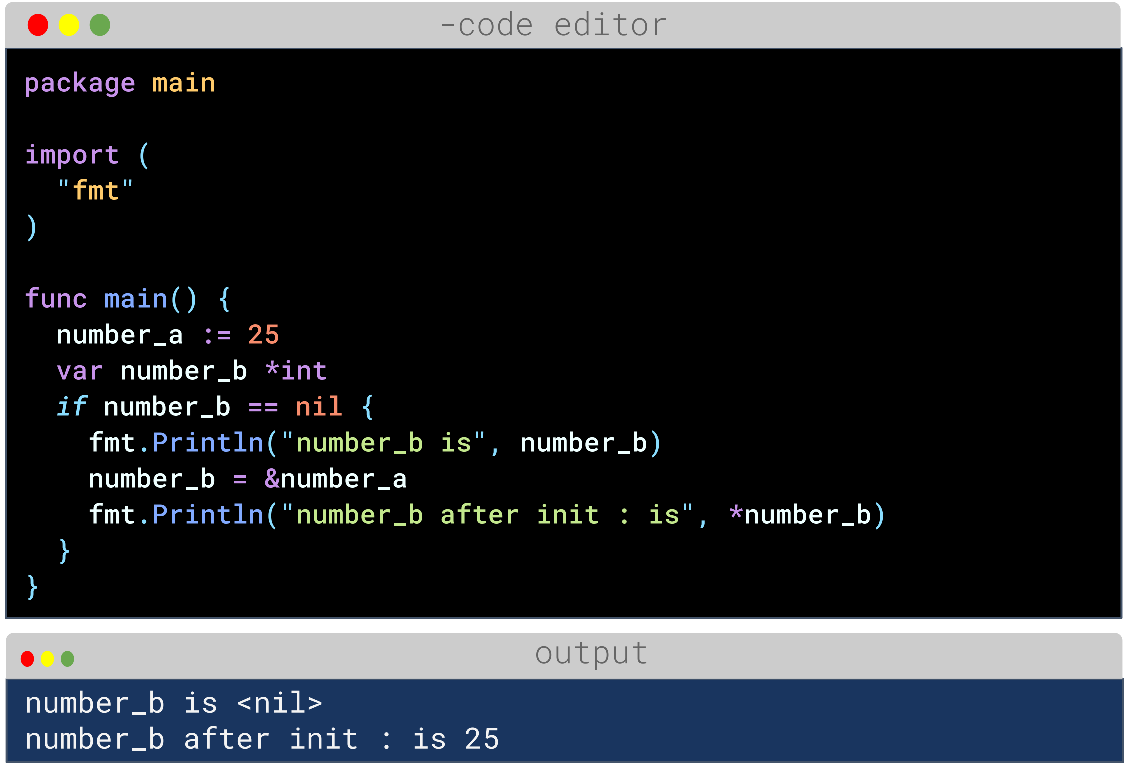
-code editor

```go
package main

import "fmt"

func main() {
  var name string = "John"
  var nameAddress *string = &name
  fmt.Println("name (value)   :", name) // John
  fmt.Println("name (address) :", &name) // 0xc20800a220
  fmt.Println("nameAddress (value)   :", *nameAddress) // John
  fmt.Println("nameAddress (address) :", nameAddress) // 0xc20800a220

  name = "Doe"

  fmt.Println("")
  fmt.Println("name (value)   :", name) // Doe
  fmt.Println("name (address) :", &name) // 0xc20800a220
  fmt.Println("nameAddress (value)   :", *nameAddress) // Doe
  fmt.Println("nameAddress (address) :", nameAddress) // 0xc20800a220
}
```

2 Important Operator in Pointer

- *** Operator** — **Dereferencing**
  - Declare pointer variable
  - Access the value stored in the address
- **& Operator** — **Referencing**
  - Returns the address of a variable
  - Access the address of a variable to a pointer

ZERO VALUE POINTER <nil>



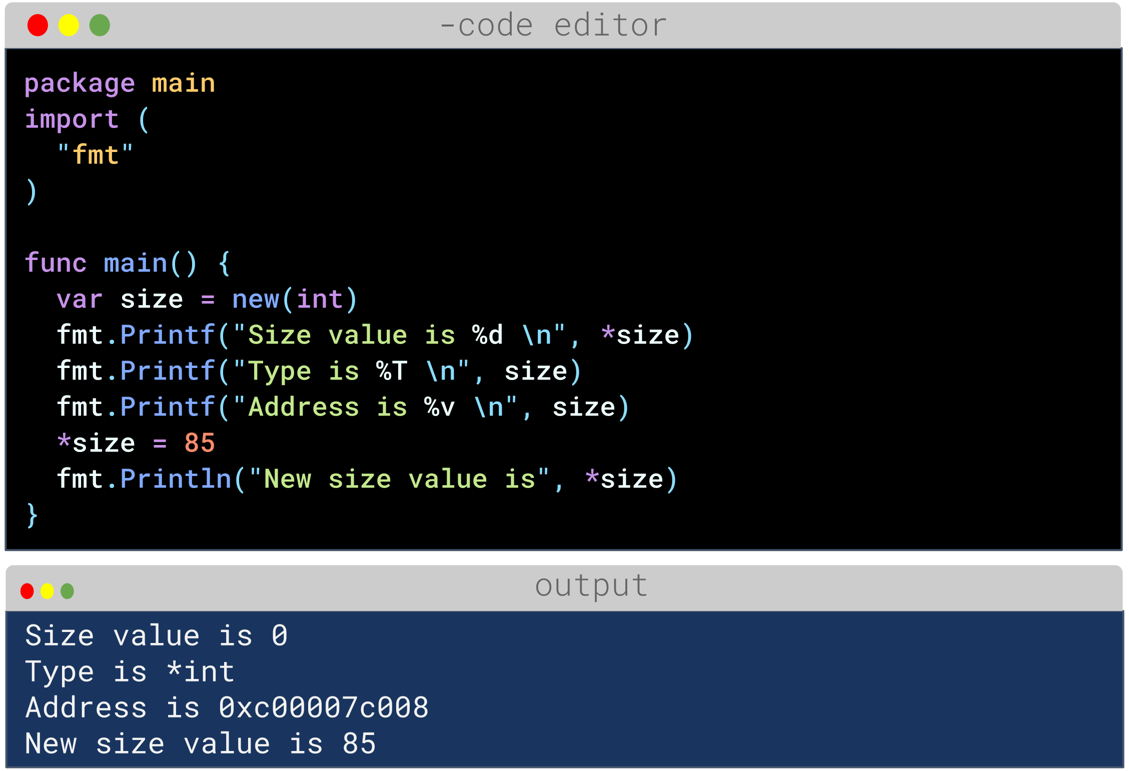
-code editor

```go
package main

import (
  "fmt"
)

func main() {
  number_a := 25
  var number_b *int
  if number_b == nil {
    fmt.Println("number_b is", number_b)
    number_b = &number_a
    fmt.Println("number_b after init : is", *number_b)
  }
}
```

output

```
number_b is <nil>
number_b after init : is 25
```

POINTER DECLARATION WITH BUILT-IN NEW()


-code editor

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

func main() {
  var size = new(int)
  fmt.Printf("Size value is %d \n", *size)
  fmt.Printf("Type is %T \n", size)
  fmt.Printf("Address is %v \n", size)
  *size = 85
  fmt.Println("New size value is", *size)
}
```

output

```
Size value is 0
Type is *int
Address is 0xc00007c008
New size value is 85
```

STRUCT

Call Object in Golang

A struct is a user-defined type that contains a collection of named fields/properties or functions (methods).

DECLARATION STRUCT

```
type struct_variable_name struct
{
    field <data_type>
    field <data_type>
    ...
    field <data_type>
}
```

INITIALIZATION & ACCESS FIELDS

```go
package main

import "fmt"

type Person struct {
  FirstName string
  LastName  string
  Age       int
}

func main() {
  //
}
```

```go
// long declaration
var Person0 Person
Person0.FirstName = "Muchson"
Person0.LastName = "Ibi"
Person0.Age = 27
fmt.Println(Person0.FirstName, Person0.LastName, Person0.Age)

// long declaration with assigned value
var Person1 = Person{"Rizky", "Kurniawan", 26}
fmt.Println(Person1)

// long declaration with assigned value each name fields
var Person2 = Person{
  FirstName: "Iswanul",
  LastName:  "Umam",
  Age:       25,
}
fmt.Println(Person2)

// sort declaration
Person3 := Person{"Pranadya", "Bagus", 23}
fmt.Println(Person3)

// short declaration with new keyword
Person4 := new(Person)
Person4.FirstName = "Muhammad"
Person4.LastName = "Ismail"
Person4.Age = 30
fmt.Println(*Person4)
```

alterra
academy

Method

What Is Method?

Method is a function that attaches to a type (can be a struct or other data type).

METHOD DECLARATION

Same as the function, only the declaration of the object variable needs to be added between the func keyword and the function name.

```
func (receiver StructType) MethodName(parameterList) (returnTypes) {
    // block statement
}
```

METHOD VS FUNCTION.

```go
func (receiver StructType) functionName(input type) returnType {
  // block statement method
}


func functionName(input type) returnType {
  // block statement function
}
```

WHAT IS THE PROBLEM?

```
package main

import "fmt"

type Employee struct {
    FirstName, LastName string
}

func fullName(firstName string, lastName string)
(fullName string) {
    fullName = firstName + " " + lastName
    return
}

func main() {
    e := Employee{
        FirstName: "Ross",
        LastName:  "Geller",
    }

    fmt.Println(fullName(e.FirstName, e.LastName))
}
```

HOW TO MAKE SIMPLE WITH METHOD

```go
package main

import "fmt"

type Employee struct {
    FirstName, LastName string
}

func (e Employee) fullName() string {
    return e.FirstName + " " +
e.LastName
}

func main() {
    e := Employee{
        FirstName: "Ross",
        LastName:  "Geller",
    }
    fmt.Println(e.fullName())
}
```

WHY METHOD INSTEAD OF FUNCTION?

- Help you write object-oriented style code in Go.
- Methods help you avoid naming conflicts.
- Method calls are much easier to read and understand than function calls.

Example Using struct For Object Oriented Style (Encapsulation Behavior)

```go
package main

import "fmt"

type Person struct {
  name string // Both non exported fields.
  age  int
}

func (P Person) GetName() string {
  return P.name + " amazing!"
}

func (P *Person) IncreaseAge() {
  P.age = P.age + 1
}

func main() {
  PersonA := Person{"John", 50}
  fmt.Printf("%v\n", PersonA)
  fmt.Println(PersonA.GetName())

  PersonA.IncreaseAge()
  fmt.Println(PersonA.age)
}
```

-terminal

```
{John 50}
John amazing!
51
```

Methods help you avoid naming conflicts.

```go
package main

import (
  "fmt"
  "math"
)

type Rect struct {
  width  float64
  height float64
}

type Circle struct {
  radius float64
}

func (r Rect) Area() float64 {
  return r.width * r.height
}

func (c Circle) Area() float64 {
  return math.Pi * c.radius * c.radius
}

func main() {
  rect := Rect{5.0, 4.0}
  cir := Circle{5.0}
  fmt.Printf("Area of rectangle rect = %0.2f\n", rect.Area())
  fmt.Printf("Area of circle cir = %0.2f\n", cir.Area())
}
```

-terminal

```
Area of rectangle rect = 20.00
Area of circle cir = 78.54
```

Struct using Pointer Receiver.

```go
package main

import "fmt"

type Employee struct {
  name   string
  salary int
}

func (e *Employee) changeName(newName string) {
  (*e).name = newName
}

func main() {
  e := Employee{
    name:    "Ross Geller",
    salary: 1200,
  }

  // e before name change
  fmt.Println("e before name change =", e)
  // create pointer to `e`
  ep := &e
  // change name
  ep.changeName("Monica Geller")
  // e after name change
  fmt.Println("e after name change =", e)
}
```

-terminal

```
e before name change = {Ross Geller 1200}
e after name change = {Monica Geller 1200}
```

Interface

- What is Interface?
- Declaring Interface
- Implementing Interface
- Empty Interface
- Type Assertion
- Type Switch

WHAT IS INTERFACE?

An interface is a collection of **method signatures** that an **object** can implement. Hence interface defines the behavior of the object.

DECLARATION INTERFACE

```
type interface_name interface {
    method_name1 <return_type>
    method_name2 <return_type>
    method_name3 <return_type>
    ...
    method_namen <return_type>
}
```

ZERE VALUE INTERFACE

Nil

```go
package main

import "fmt"

type calculate interface {
  large() int
}

type square struct {
  side int
}

func (s square) large() int {
  return s.side * s.side
}

func main() {
  var dimResult calculate
  dimResult = square{10}
  fmt.Println("large square :", dimResult.large())
}
```

alterra academy

IMPLEMENTING INTERFACE

OUTPUT

command line

```
large square : 100
```

```go
package main

import "fmt"

func describe(i interface{}) {
  fmt.Printf("(%v, %T)\n", i, i)
}

func main() {
  var i interface{}
  describe(i)

  i = 42
  describe(i)

  i = "hello"
  describe(i)
}
```

Empty Interface for Dynamic Value

Output

command line

```
(<nil>, <nil>)
(42, int)
(hello, string)
```

```go
package main

import "fmt"
import "strings"

func main() {
  var secret interface{}

  secret = 2
  var number = secret.(int) * 10
  fmt.Println(secret, "multiplied by 10 is :", number)

  secret = []string{"apple", "manggo", "banana"}
  var gruits = strings.Join(secret.([]string), ", ")
  fmt.Println(gruits, "is my favorite fruits")
}
```

TYPE ASSERTION

i.(Type)

WHERE I IS AN INTERFACE AND TYPE IS A TYPE THAT IMPLEMENTS THE INTERFACE I

command line

```
2 multiplied by 10 is : 20
apple, manggo, banana is my favorite fruits
```

```go
package main

import (
  "fmt"
  "strings"
)

func explain(i interface{}) {
  switch i.(type) {
  case string:
    fmt.Println("i stored string ",
strings.ToUpper(i.(string)))
  case int:
    fmt.Println("i stored int", i)
  default:
    fmt.Println("i stored something else", i)
  }
}

func main() {
  explain("Hello World")
  explain(52)
  explain(true)
}
```

TYPE SWITCH

SIMILAR TO TYPE ASSERTION AND IT IS **`i.(type)`** BUT ONLY WORK IN SWITCH STATEMENT

command line

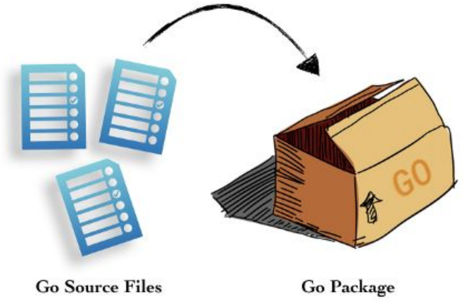
```
i stored string  HELLO WORLD
i stored int 52
i stored something else true
```

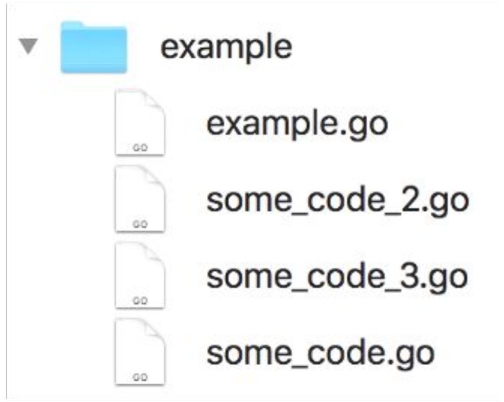
alterra academy

Package

WHAT IS PACKAGE?

A package is a collection of functions and data.





alterra academy

Example Package

-code editor

```go
// aritmatika/package.go
package aritmatika

func Tambah(a, b int) int {
  return a + b
}

func Kurang(a, b int) int {
  return a - b
}
```

-code editor

```go
// main.go
package main

import (
  "aritmatika"
  "fmt"
)

func main() {
  fmt.Println(aritmatika.Tambah(2, 3))
}
```

MAKING VARIABLE ACCESSIBLE TO OTHER PACKAGE

```
var ageOfUniverse int
```

Other package can't see this because the first letter is lowercase

```
var AgeOfUniverse int
```

Other package can see this because the first letter is UPPERCASE

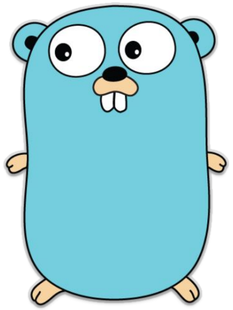
alterra
academy

Error Handling

ERROR, PANIC & RECOVER

THERE IS NO TRY CATCH IN GOLANG

❌



PANIC & RECOVER

✓

IF YOU ARE WRITING A METHOD YOURSELF THAT REQUIRES TO RETURN ERROR IF SOMETHING WRONG HAPPENS IN BETWEEN, USE THE 'ERRORS' PACKAGE FOR SUCH PURPOSE. LETS SEE A SMALL EXAMPLE:

ERROR HANDLING OBJECT

```go
package main

import "fmt"

import (
  "errors"
)

func myFunc(i int) (int, error) {
  if i <= 0 {
    return -1, errors.New("should be greater than zero")
  }
  return i, nil
}

func main() {
  result, err := myFunc(-1)
  fmt.Println(result, err)
}
```

Panic

When the Go runtime detects these mistakes, it panics.

Recover

To add the ability to recover from a panic error, either add an anonymous function or define a custom function and call it with 'defer' keyword from inside the method, where panic might be occurring from other internal calls.

```go
package main

import "fmt"

func myMethod() {
  defer func() {
    if err := recover(); err != nil {
      fmt.Println("Error Message:", err)
    }
  }()

  anOddCondition := true
  if anOddCondition {
    panic("I am panicking")
  }
}

func main() {
  myMethod()
}
```

alterra academy

“It is not enough for code to work.”

- Robert C. Martin -

alterra academy

Any Question

?