Introduction to Golang
Part 2

DOMINEUM

BLOCKCHAIN

Calistus Igwilo
https://linkedin.com/in/calistus-igwilo
https://twitter.com/CalistusIgwilo

Go Example

```go
package main

import "fmt"

func main() {
    fmt.Printf("Hello world!\n")
}
```

Go Example

```go
package main

import "fmt"

func Hello(name string) string {
    message := fmt.Sprintf("Hi, %v. Welcome!", name)

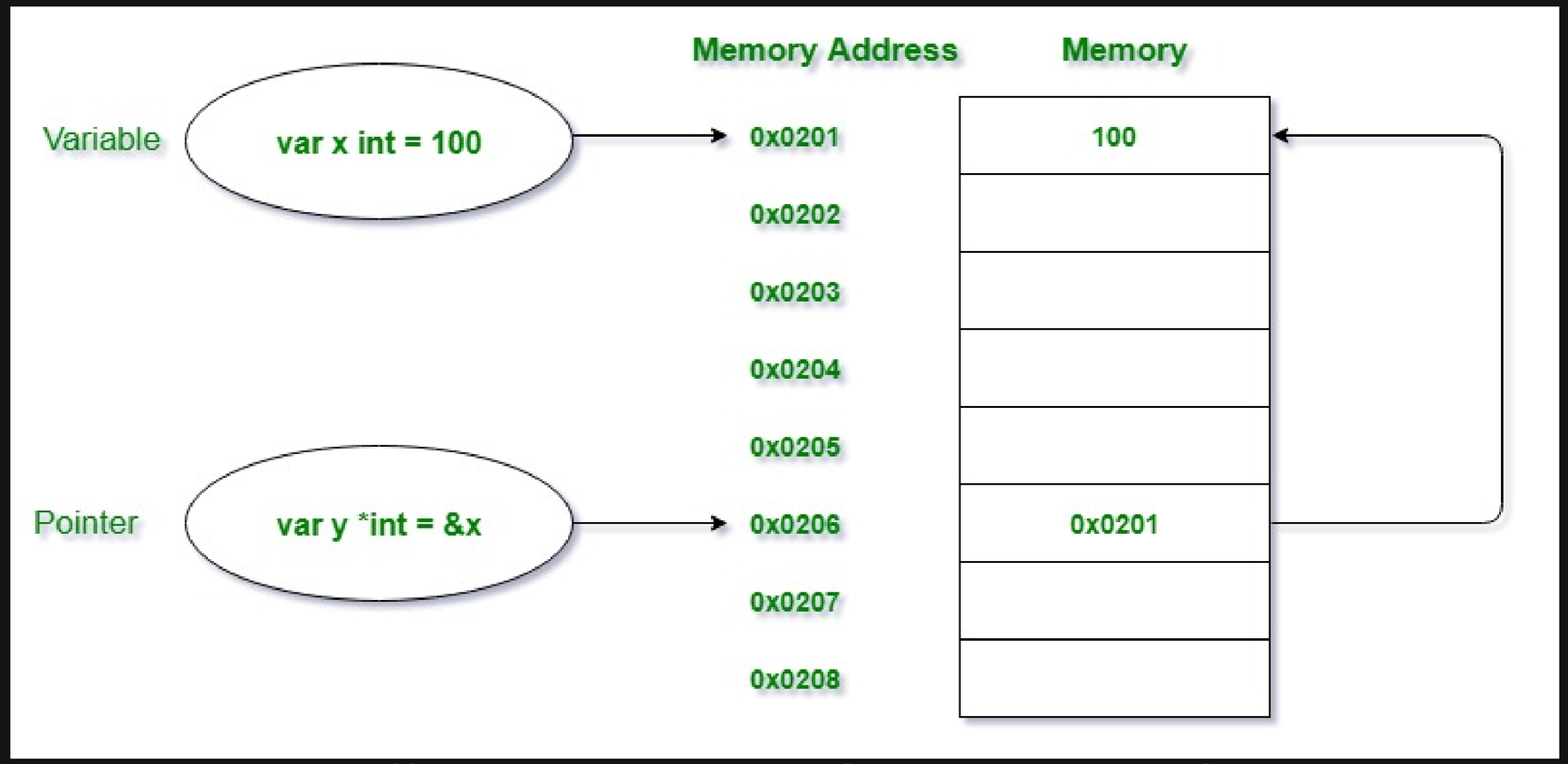
    return message
}

func main(){
    fmt.Printf(Hello("Calistus"))
}
```

Pointers

Stores the memory address of other variables

- Shows where the memory is located
- Can show the value stored at that memory location



https://www.geeksforgeeks.org/pointers-in-golang/

Pointer Operators

& Operator: (address operator), returns the address of a variable or function

* Operator: (dereferencing operator), returns the data at an address

Pointers

```
var x int = 1
var y int
var z *int    // z is a pointer to an int (integer)

z = &x        // z now points to x
y = *z        // y is now 1 (dereferencing)
```

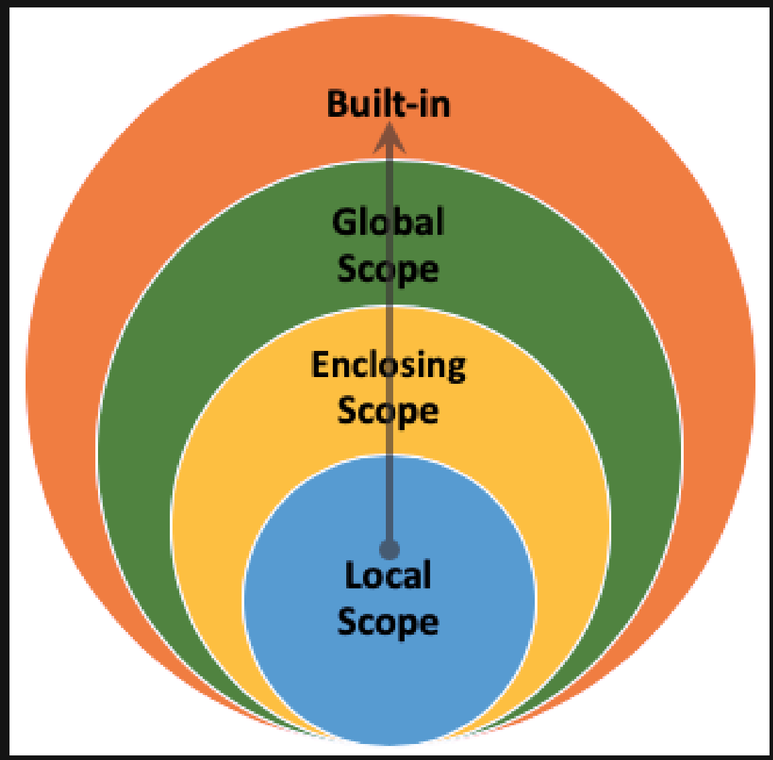
new()

- Alternate way to create a variable
- new() function creates a variable and returns a pointer to the variable
- Variable is initialized to zero

```
ptr = new(int)
*ptr = 3
```

Variable Scope

A place in code where a variable can be accessed



https://www.datacamp.com/tutorial/scope-of-variables-python

Variable Scope

```
var x = 7

func f() {
    fmt.Printf("%d", x)
}


func g() {
 fmt.Printf("%d", x)
}
```

```
func f() {
    var x = 7
    fmt.Printf("%d", x)
}


func g() {
 fmt.Printf("%d", x)
}
```

Blocks

A series of declarations and statements withing matching brackets {}

- Including functions definitions

Hierachy of implicit blocks

- Universal block: All Go source
- Package block: All source in a package

File block: All source in a file

"if", "for", "switch": All code inside the statement

Clause in "switch" or "select". Each gets a block

Lexical Scoping

A series of declarations and statements withing matching brackets {}

- Including functions definitions

Hierachy of implicit blocks

- Universal block: All Go source
- Package block: All source in a package

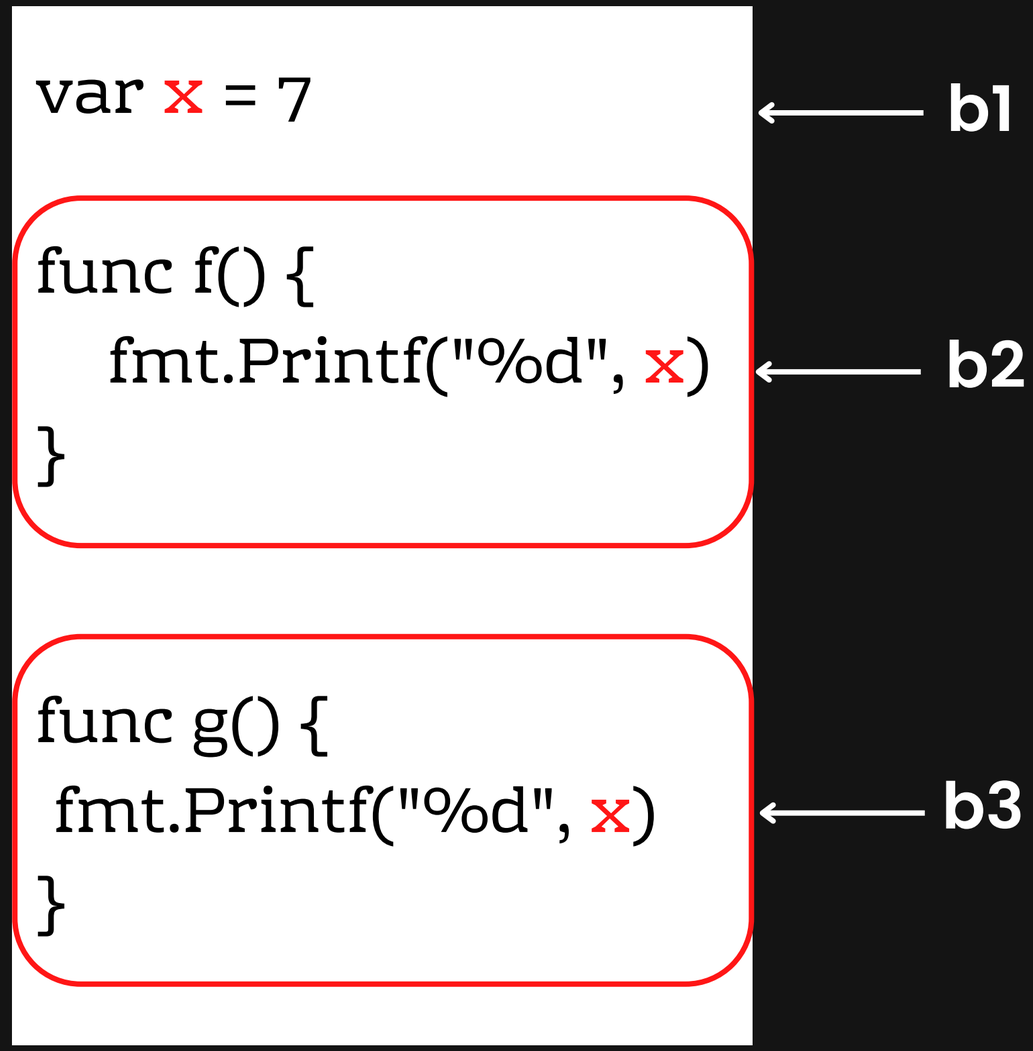
File block: All source in a file

"if", "for", "switch": All code inside the statement

Clause in "switch" or "select". Each gets a block

Lexical Scoping

bi >= bj if bj is defined inside bi



Scope of Variable

- Variable is accessible from block bj if
  - variable is declared in block bi and
  - block bi >= bj

```
var x = 7

func f() {
    fmt.Printf("%d", x)
}


func g() {
 fmt.Printf("%d", x)
}
```

```
func f() {
    var x = 7
    fmt.Printf("%d", x)
}


func g() {
 fmt.Printf("%d", x)
}
```

Deallocating Space

- When a variable is no longer needed it should be deallocated
  - Memory spce is made available
- Otherwise, we will run out of memory eventually

```
var x = 7

func f() {
    fmt.Printf("%d", x)
}
```

- Each func() call, creates an integer

Stack vs Heap

- Stack is dedicated to function calls Memory
  - Local variables are stored here
  - Deallllocated after function call

- Heap is a persistent memory region
  - You have to explicitly deallocate it
  -

Manual Deallocation

- Data on Heap must be deallocated when it is done being used
- In most compiled languages (like c) it is done manually.
  - x = malloc(32);
  - free(x);
- Error prone but fast

Pointers & Deallocation

- Hard to determine when a variable is no longer in use

- foo() returns pointer to x

```
func foo() *int {
    x := 1
    return &x
}

func main () {
    var y *int
    y = foo()
    fmt.Printf("%d", *y)
}
```

Garbage Collection

- In interpreted languages, this is done by the Interpreter
  - Java Virtual Machine
  - Python Interpreter
- It keeps track of pointers and determine when a variable is no longer in use or has references to it, then deallocates it

- Easy for the programmer (but requires an interpreter)

Garbage Collection in Go

- Go is a compiled language which enables garbage collection
- Implementation is fast
- Compiler determins stack vs heap
- Garbage collection in the background
  - slows down things a bit, but it is a good trade-off

Comments

- Comments are text for code understandability
- Ignored by compilers
- Single line comments
  - // This is a comment
  - var x int // Another comment

- Block comments
  - /* comment 1
  - comment 2
  - */
  - var x int

Printing

- Import the fmt package
- fmt.Printf() (fmt.Println)
  - Prints a string
  - fmt.Printf("Hi")
  - x := Calis
  - fmt.Printf("Hi" + x)
  -

Printing

- Format strings are good for formatting
- Conversion characters for each string
  - fmt.printf("Hi %s", x)
  -

Integers

- Generic int declaration
  - var x int
- Different lenghts and signs
  - int8, int16, int32, int64, uint8, uint16, uint32, uint64

Integers

- **Binary operators**
  - Arithmetic: + - * / % << >>
  - Comparison: == != < > <= >=
  - Boolean: && ||

Type Conversions

- **Most binary operations need operands of the same type**
  - **var** x int32 = 4
  - var y int16 = 2
  - x = y // this will fail
- Convert type with T() operation
  - x = int32(y)

Floating Point

- **float 16 - ~ 6 digits precision**
- **float 32 - ~ 15 digits precision**
- Expressed using decimal or scientific notation
  - var x float64 = 123.45
  - var y float64 = 1.2345e2

ASCII & Unicode

- American Standard Code for Information Interchange
- Character coding - each character is associated with a 7 (8) bit number
  - "A" = 0x41
- This is sufficient for the English alphabets

ASCII & Unicode

- Unicode is a 32bit Character code
- UTF-8 is variable length（can go from 8 to 32 bits）
  - 8bit UTF code are same as ASCII
- In Go, the default is UTF-8

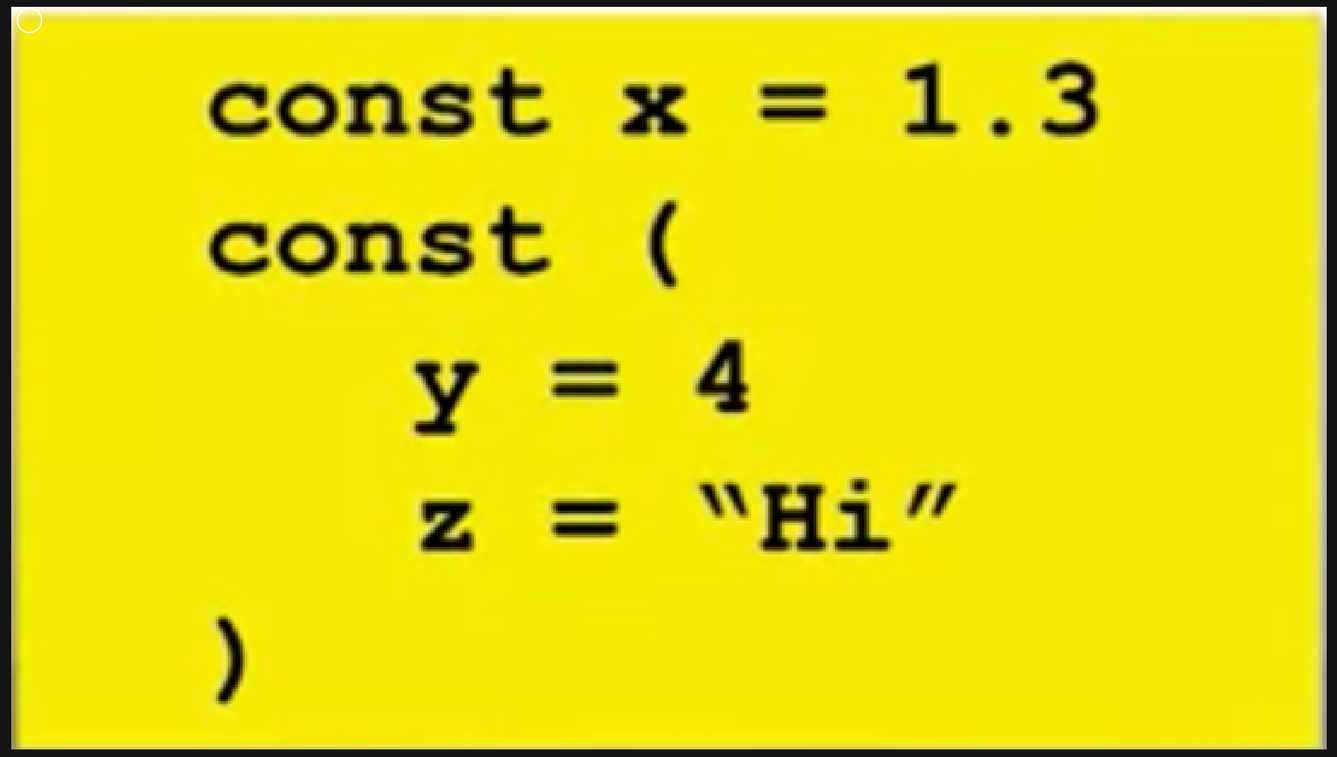
Strings

- **Arbitrary sequence of bytes represented in UTF-8**
  - **Read-only**
  - Often meant to be printed

- **String literals: denoted by double quotes**
  - **x := "Hi there"**
- Each byte is a rune (UTF-8 code point)

Constants

- Expression whose value is known at runtime
- Type is inferred from right hand side



Control Structures

Statements which alter control flows

- ```
  if <condition> {
      <statement>
  }
  ```
  ```

Control Structures

- Expression <condition> is evaluated
  - <statements> are evaluated if condition is true
  - if (y > 0) {
  - fmt.Printf("Positive")
  - }

For Loop

- Iterates while condition is true
- May have initialization and update operation
    - for <init>; <condition>; <update> {
    - statements
    - }

For Loop Forms

```
for i:=0; i<10; i++ {
    fmt.Printf("hi ")
}
```

```
i = 0
for i < 10 {
    fmt.Printf("hi ")
    i++
}
```

```
for {
    fmt.Printf("hi ")
}
```