WEEK 2

Introduction to Golang Programming

Go Brief

- **Go** is a modern programming language created by Google in 2007 and released in 2009.

Created by three Google engineers:

- **Robert Griesemer**
- **Rob Pike**
- **Ken Thompson** (co-creator of Unix and C language!)

Go Brief

1. Statically Typed

```
var age int = 25  // Type must be declared or inferred
name := "John"    // Type inferred as string
```

2. Garbage Collected

Automatic memory management
No manual memory allocation/deallocation
Optimized for low-latency applications

3. Fast Compilation

Compiles to native machine code
Single binary with no dependencies
Cross-compilation support

Go Brief

4. Easy Concurrency

```
// Launch a goroutine with just the 'go' keyword
go doSomething()

// Communicate safely between goroutines
ch := make(chan int)
ch <- 42  // Send to channel
value := <-ch  // Receive from channel
```

5. Built-in Testing

Testing framework included in standard library
Benchmarking support
Code coverage tools

6. Standard Library

Rich standard library for common tasks
HTTP server/client
JSON encoding/decoding
Cryptography
File I/O

Go Brief

Feature	Description	Why It Matters for Crypto
Compiled language	Translates directly into machine code.	Faster encryption & network tools.
Memory safety	No buffer overflows or pointer misuse.	Reduces vulnerabilities.
Strong typing	Prevents accidental data-type bugs.	Fewer crypto logic errors.
Concurrency (goroutines)	Run tasks in parallel safely.	Useful for brute-force & crypto analysis.
Rich standard library	Includes crypto, encoding, net, I/O, etc.	You can build secure tools easily.

Software and Tools made by Go Lang

Bettercap

Merlin C2

https://go.dev/solutions/case-studies
https://github.com/avelino/awesome-go

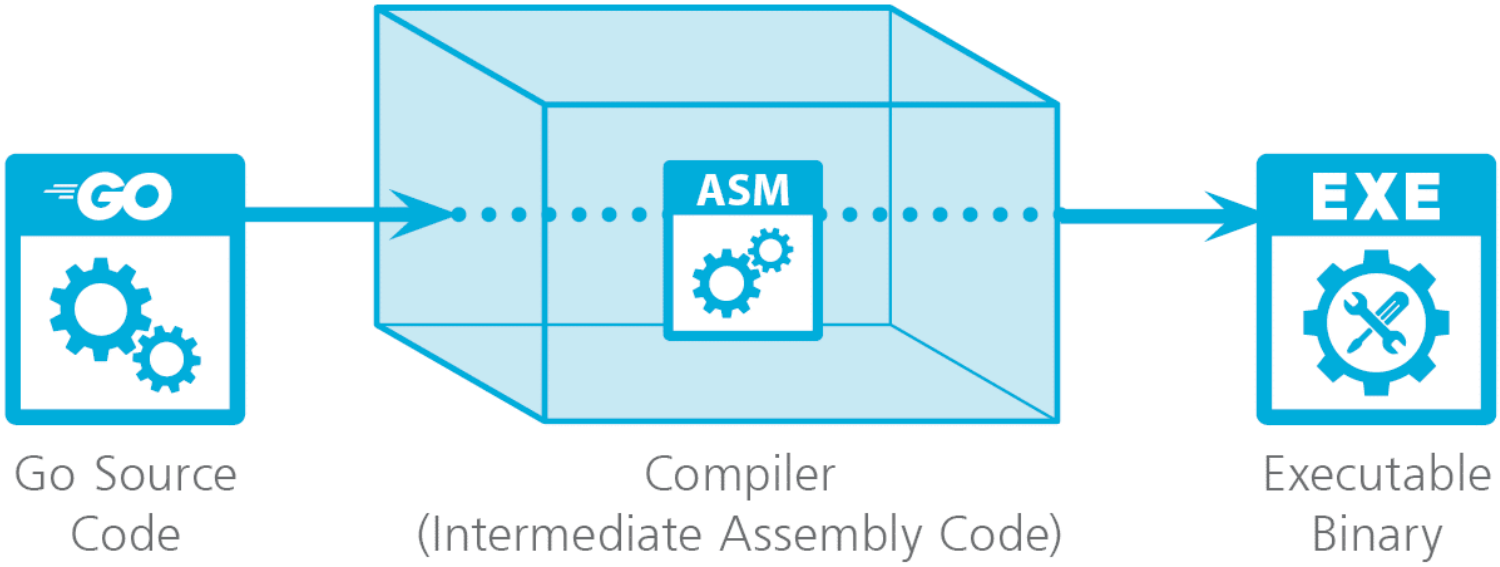
Install GO

- https://go.dev/doc/tutorial/getting-started
- https://go.dev/tour/list (READ/*PRACTICE)
- https://leanpub.com/gocrypto/read

Choosing IDE

- User-friendly GUI IDE
  - JetBrains Goland
  - Microsoft VS Code
- Neovim or Vim (Terminal-based text editors)
  - (NvChad, LazyVim)

Golang Syntax: Compiler



Create a Go module

```
# Make Dir && Change Directory
$ mkdir -p
  lab1/{cmd/app,internal/{service,handler,utils},pkg/mylib,api
  ,configs,deployments,scripts,test,docs}; cd lab1;

# Create the Module (small Project)
$ go mod init example.com/smalllab
or go mod init github.com/u/repo
```

Create a Go workspace

```
# Create the workspace (multiple module)
$ go work use example.com/lab/module1
  example.com/lab/module2
```

Go Command Line

- go version
- go mod init project/package
- go run file.go
- go build file.go ./* (compile the code)
- go get (add/update dependency)
- go doc package – View Document
- go fmt (format code)

lab1 - main_test.go

```go
package main

import (
    "bytes"
    "os/exec"
    "path/filepath"
    "runtime"
    "testing"
)

func TestMainOutput(t *testing.T) {
    // get module root path
    _, file, _, _ := runtime.Caller(0)
    root := filepath.Join(filepath.Dir(file), "../..")

    cmd := exec.Command("go", "run", "./cmd/bin")
    cmd.Dir = root
    out, err := cmd.CombinedOutput()
    if err != nil {
        t.Fatalf("run error: %v", err)
    }

    got := string(out)
    want1 := "HOOL"
    want2 := "Hello, my friend!"

    if !(bytes.Contains(out, []byte(want1)) && bytes.Contains(out, []byte(want2))) {
        t.Fatalf("unexpected output:\n%s", got)
    }
}

```

Go Lib: Cryptograhpy and Hacking

- https://pkg.go.dev/github.com/84kaliplexon3/hacking-with-go#section-readme
- https://pkg.go.dev/github.com/r3dhulk/golang-for-ethical-hackers#section-readme
- https://pkg.go.dev/golang.org/x/crypto
- "crypto/rand"
- "encoding/base64"
- "fmt”
- Crypto/cipher
- Crypto/aes
- Crypto/rsa + Math/big
- Crypto/*

Golang Syntax: Variable & Data Type

- A bit (short for binary digit) is the smallest and most fundamental unit of data in a computer. It can only hold a single value: 0 or 1.
- A **byte** is the standard unit for digital information, universally consisting of 8 bits. It's the smallest chunk of memory that most computers can read from or write to in a single operation. In Go, a **byte** is an alias for an **int8. 00001111 (8 bits = 1byte)**
- An **integer** (int, uint) is a data type that represents whole numbers.
- A **string** is a data type representing a sequence of characters, used for storing human-readable text.
- A **rune** is Go's specific data type for a single character. It's designed to represent any character from any language in the world, from English ('A'), to French ('é'), to Japanese ('あ'), to emojis ('😊'). In Go, a rune is an alias for an **int32**.
  - It’s a char in C/C++
- A **boolean** (bool) is a data type that can only have two possible values: true or false.

Golang Syntax: Variable & Data Type

- A variable is a named storage location in a program that holds a value. You can look up or change the value by using its name. i.e userAge, password, name, amount, lightBulb

- A data type is a classification that tells the program what kind of value a variable can hold and what operations can be performed on it.
  - For example:
    - userAge (age is a number and can't be negative) so var userAge uint = 18
    - password (age is a text/string and can be) so var password string = "SecurePassword!"
    - amount (age is a floating number) so var amount float64 = 100000.0000 //USD Dollar
    - lightbulb (it can be turn on/off) so var lightbulb bool = false or true

Golang Syntax: Data Type

Data Type	Description	Example
String	Immutable sequence of bytes (UTF-8 text by convention)	"hello"
[]byte	Alias of uint8 (raw byte)	[]byte(”A"), 0x41
Int,uint	Signed integer	256
float	64-bit IEEE-754 floating point	2.718281828
bool	Boolean truth value	true,false
rune	Alias of int32 (Unicode code point)	’界’ or ‘A’

https://pkg.go.dev/builtin#pkg-types

Golang Syntax: Data type

- The integer data type has two categories:
  - **Signed integers** - can store both positive and negative values
    - int8
    - int16
    - int32 (int 32bits)
    - Int64 (bit 64 bits)
  - **Unsigned integers** - can only store non-negative values
    - uint8: An 8-bit unsigned integer with a range of 0 to 255. This is also aliased as the byte type and is often used for binary data.
    - uint16: A 16-bit unsigned integer with a range of 0 to 65535.
    - uint32: A 32-bit unsigned integer with a range of 0 to 4,294,967,295.
    - uint64: A 64-bit unsigned integer with a range of 0 to 18,446,744,073,709,551,615.
  - Arbitrary-precision arithmetic (Woking with Big Number)
    - Int (Big Number from Math/big)

Golang Syntax: Naming Rule

- Go variable naming rules:
  - A variable name must start with a letter or an underscore character (_)
  - A variable name cannot start with a digit
  - A variable name can only contain alpha-numeric characters and underscores (a-z, A-Z, 0-9, and _)
  - Variable names are case-sensitive (age, Age and AGE are three different variables)
  - There is no limit on the length of the variable name
  - A variable name cannot contain spaces The variable name cannot be any Go keywords

Golang Syntax: Arithmetic

Operator	Description	Example
+	Adds two operands	A + B gives 30
-	Subtracts second operand from the first	A - B gives -10
*	Multiplies both operands	A * B gives 200
/	Divides the numerator by the denominator.	B / A gives 2
%	Modulus operator; gives the remainder after an integer division.	B % A gives 0
++	Increment operator. It increases the integer value by one.	A++ gives 11
--	Decrement operator. It decreases the integer value by one.	A-- gives 9

Golang Syntax: Operator Type

- **Relational (Comparation operator)**
  - Grater than (>) | Grater than or equal (>=)
  - Less than (<) | Less than or equal (<=)
  - Equal (==)
  - Not Equal (!=)

Golang Syntax: Operator Type

- **Logical**
  - AND (&&)
  - OR (||)
  - XOR (^)
  - Not (!)

Golang Syntax: Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

Golang Syntax: Assignment Operators

Operator	Description	Example
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Golang Syntax: Operator Type

Bitwise operations

- **Shirt x=***0010*
  - **x <<2** *// Binary: 1000*
  - **x >> 2** *// Binary: 0010*

Golang Syntax: Comments

- A comment is a text that is ignored upon execution.
- Comments can be used to explain the code, and to make it more readable.
- Comments can also be used to prevent code execution when testing an alternative code.
- Go supports single-line or multi-line comments.

```
// var name string = “test”

/*

var name string = “test”
var name string = “test”

*/
```

Golang Syntax: Constant

- If a variable should have a fixed value that cannot be changed, you can use the const keyword.
- The const keyword declares the variable as "constant", which means that it is unchangeable and read-only.

```
var name string = “test” //Can change

const var name string = “test” //can’t change
const Pi = 3.14 //can’t change
```

Golang Syntax: Output Printing to Console

- Go has three functions to output text:
  - Print(”Hello”,text) //default format and if you want print new line using \n
  - Println(”Hello”,text) //Similar to print() and (a newline is added)
  - Printf(”i has value: %v and type: %T\n”,i,j)

  function first formats its argument based on the given formatting verb and then prints them.

Specifier	Type	Example
`%d`	Integer	`fmt.Printf("%d", 42)` → 42
`%f`	Float	`fmt.Printf("%f", 3.14)` → 3.140000
`%.2f`	Float (2 decimals)	`fmt.Printf("%.2f", 3.14159)` → 3.14
`%s`	String	`fmt.Printf("%s", "Hi")` → Hi
`%t`	Boolean	`fmt.Printf("%t", true)` → true
`%v`	Any type	`fmt.Printf("%v", anything)`

Golang Syntax: Input into the Console

- Go has three functions to input text into the console:
  - Scan() //default format and if you want print new line using \n
  - ScanLn(”Hello”,text) //Similar to print() and (a newline is added)
  - Scanf("%d-%d-%d", &y, &m, &d)

Golang Syntax: Control Structure – if/else

The **if/else** structure is used to execute a block of code only if a certain condition is true. You can optionally add an else block to run if the condition is false.

- **Syntax**

```
if condition:
        # Block of code if condition is True

elif another_condition:
        # Block of code if another_condition is True

else:
        # Block of code if none of the conditions are True
```

Golang Syntax: Control Structure – if/else

- **Example If/else in Go**

```
name :="Hello”;

if name=="Hello" {
        fmt.Println("Hello")
}

else if name=="Goodbye" {
        fmt.Println("Goodbye")
}

else {
        fmt.Println("Unknown greeting")
}
```

Golang Syntax: Control Structure - switch

The **switch** statement is a cleaner way to handle a sequence of if/else if checks. It compares an expression against a series of **case** values and executes the block that matches.

- **Syntax**

```
switch value_to_check {
 case condition :
      # Block of code if another_condition is True

 case: another_condition :
      # Block of code if another_condition is True

 default:
      # Block of code if none of the conditions are True
}
```

Golang Syntax: Control Structure - switch

- **Example Switch/case in Go**

```
day := "Wednesday"

switch day {
        case "Monday” :
                fmt.Println("Time to start the week.")
        case "Wednesday" :
                fmt.Println("Halfway there!")
        case "Friday" :
                fmt.Println("Weekend is coming!")
        default :
                fmt.Println("It's just another day.")
}
```

Golang Syntax: Control Structure – for/loop

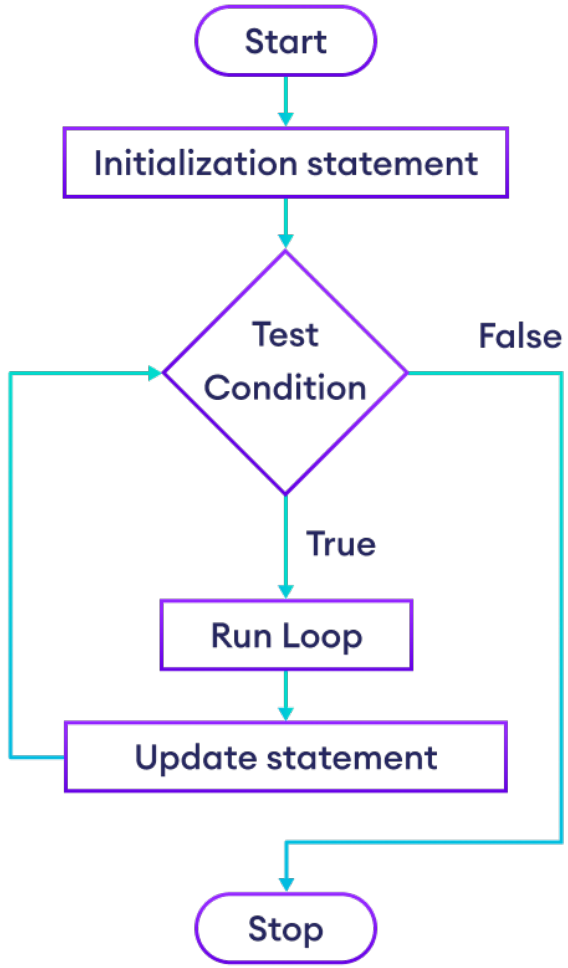
The for loop is Go's only looping structure, and it's used to repeat a block of code multiple times. It's very versatile and can be used in several ways.

- **Syntax**

```
for initialization; condition ; post statement {
      # Block of code if none of the conditions are True
}
```

Example for/loop in Go

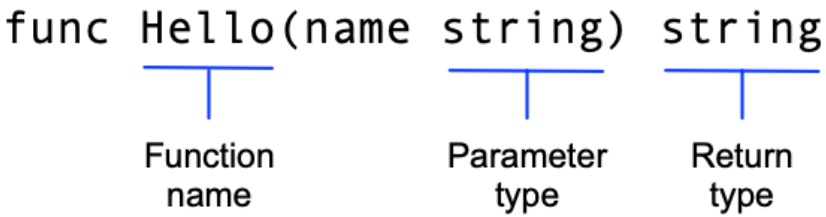
```
for i := 0; i < 3; i++ {
      fmt.Println(i) // Prints 0, 1, 2
}
```



Golang Syntax: Function

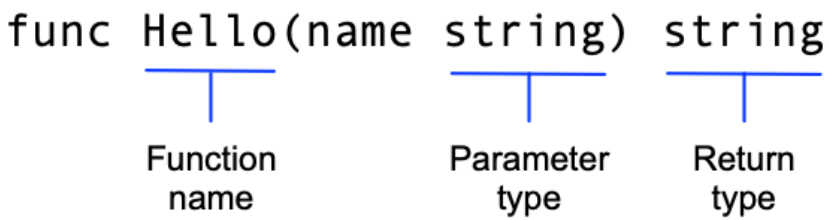
There are four common function forms:

1. No parameters, no return values
2. Parameters, no return values
3. No parameters, return values
4. Parameters, return values.



Golang Syntax: Function

```go
// 1) No params, no returns
func Hello() { fmt.Println("Hello") }


// 2) Params, no returns
func PrintN(n int) { fmt.Println(n) }


// 3) No params, returns
func Pi() float64 { return 3.14159 }


// 4) Params, returns (single and multiple)
func Add(a, b int) int { return a + b }
func DivMod(a, b int) (int, int) { return a / b, a % b } // multiple
```



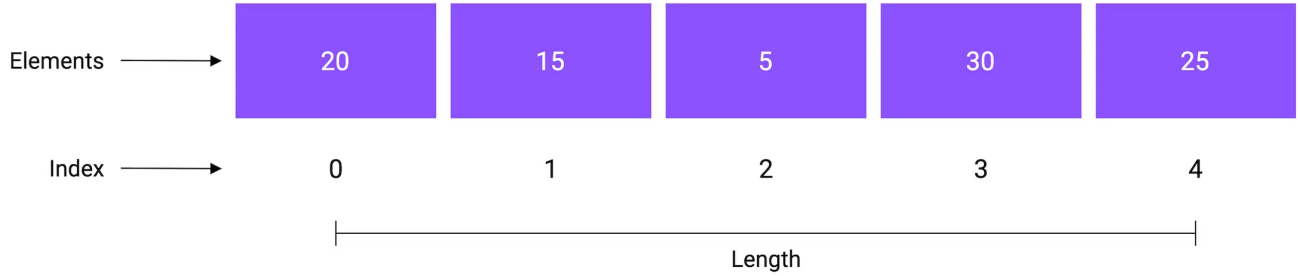
Golang Syntax: Array and Slide

- Array
  - Fixed-size: Arrays in Go have a fixed size determined at their declaration. This size cannot change during runtime.
- Slices
  - Dynamic and flexible: Slices are built on top of arrays and provide a dynamic, flexible view into an underlying array. Their size can grow or shrink.

Note: Python == list and Java ArrayLists, C++ Vector

Golang Syntax: Array and slice

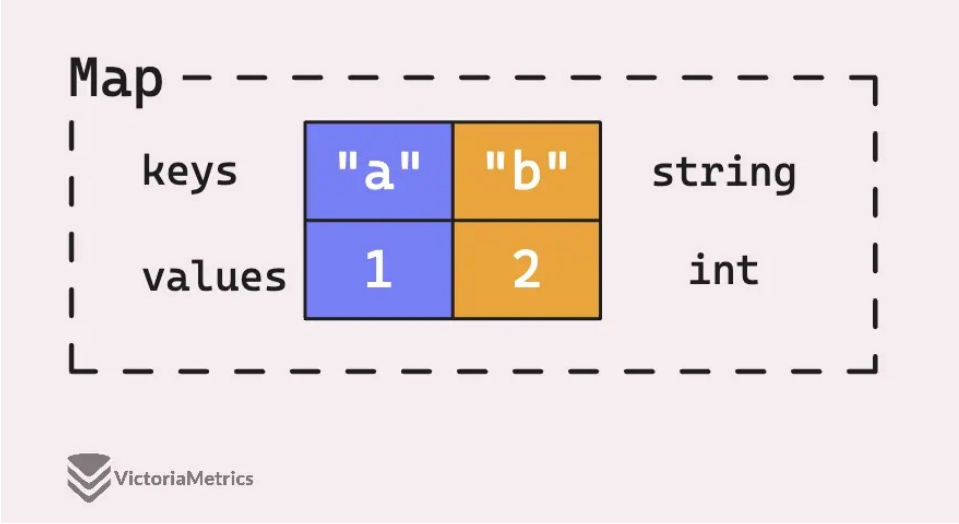
- **Array Declaration**
  - var array [5]int = {20,15,5,30,25}
- **Slices Declaration**
  - var slice []int = []int {20,15,5,30,25}
  - slice := []int {20,15,5,30,25}



Golang Syntax: Maps

- A map is a built-in data structure that stores collections of unordered key-value pairs. Each key within a map is unique and maps to a single value. Maps are similar to dictionaries in Python or hash tables in other programming languages.
- **Syntax:** Map types are declared using

  map[KeyType]ValueType



Golang Syntax: Naming Conventions aka Import / Export

Naming conventions are tied directly to visibility and control whether an identifier such as a variable, function, type, or struct field—is "exported" and accessible to other packages or "unexported" and limited to its own package

- **Exporting identifiers:** An identifier is exported and accessible from outside its package if its name begins with a capital letter.
  - Using HelloFunction() (Public)
- **Unexported identifiers** An identifier is unexported (private) and accessible only within its own package if its name begins with a lowercase letter.
  - Using helloFunction() (Private)

Golang Syntax: Structs and Object

- A struct (short for structure) is used to create a collection of members of different data types, into a single variable. While arrays are used to store multiple values of the same data type into a single variable, structs are used to store multiple values of different data types into a single variable. A struct can be useful for grouping data together to create record

- **Syntax:** struct types are declared using

```
type struct_name struct {
    member1 datatype;
    member2 datatype;
    Name string;   ...
 }
```

Golang Syntax: Structs and Object

Syntax: struct types are declared using

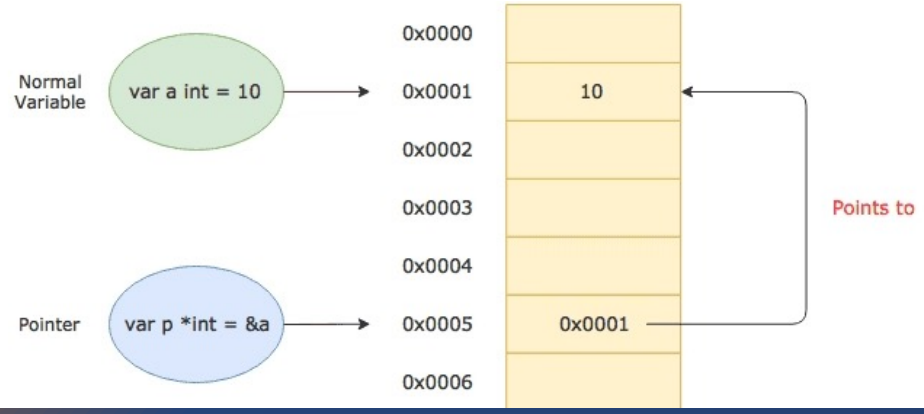
```
type CryptoAccount struct {
    username string;
    password string;
    iv []byte;   ...
 }
func main() {
    cAccount := CryptoAccount{ “user1”, “c4ca4238a0..09a6f75849b” }
    fmt.Println(“username”,cAccount.username,”Md5:”,cAccount.password)
}
```

Golang Syntax: Pointer

- A pointer is a variable that stores the memory address of another variable. Instead of holding a data value directly, it holds the location in memory where that value is stored. This allows for indirect interaction with and alteration of the values of targeted variables.

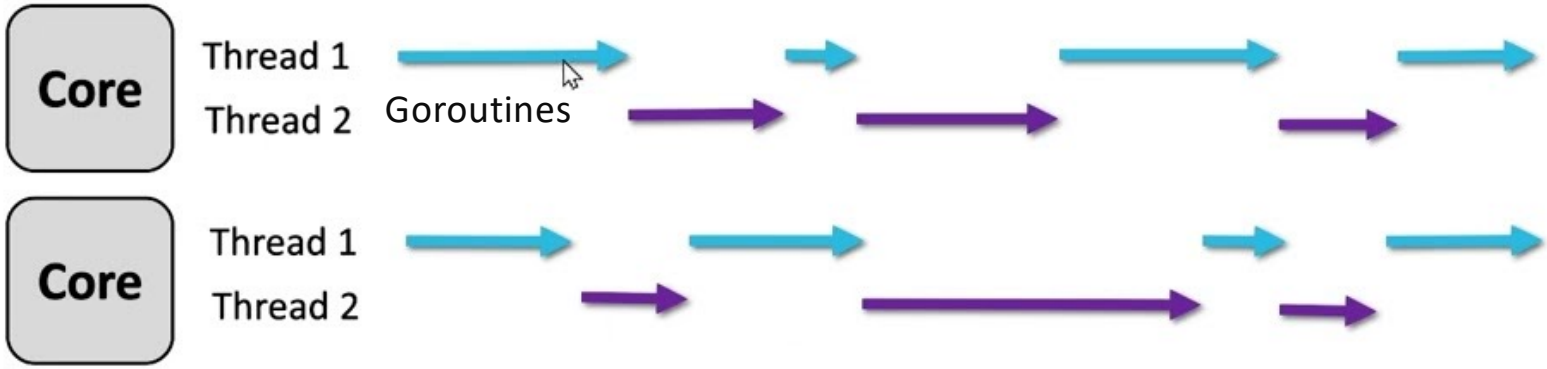
  & (Address-of Operator): Used to get the memory address of a variable.

  * (Dereference Operator): Used to access the value stored at a memory address pointed to by a pointer.



Golang Syntax: Goroutines

Concurrent and Parallel



- "Doing lots of things at once" with multiple cores
- "Concurrency is about dealing with lots of things at once" - efficiently handling multiple things with threads

Golang Syntax: Goroutines

Syntax: using concurrently executing functions

```
func employee(ch chan string) {
        ch <- “task done!”
        ...
 }
func main() {
    ch := make(chan string) // create a channel

    go employee(ch)          // run worker concurrently

    msg := <-ch             // receive message (blocks until available)
    fmt.Println(msg)
}
```

Golang Syntax: Error Handling

- Core Ideas

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

- Sample error handling

```
//Using error handling
_, err := os.ReadFile(path)
if err != nil {
        return fmt.Errorf("read %q: %w", path, err) // wrap
}

//or
_, err := os.ReadFile(path)
if err != nil {
        panic(“err”)
}
```

Golang Syntax: Race condition/Critical section

- **Race condition**: two or more goroutines access the same **mutable** memory at the same time, and at least one access is a write. Outcome depends on timing.
- **Critical section**: the chunk of code that touches that shared state and therefore must not run concurrently.

Golang Syntax: Race condition/Critical section

```
func main() {
    var (
    wg sync.WaitGroup
    mu sync.Mutex
    )
    count := 0

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count++
        }()
    }

    wg.Wait()
    fmt.Println("count =", count) // always 1000
}
```

Golang Syntax: Race condition/Critical section

```
func main() {
    var (
    wg sync.WaitGroup
    mu sync.Mutex
    )
    count := 0

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            mu.Lock() // enter critical section
            count++
            mu.Unlock() // leave critical section
        }()
    }

    wg.Wait()
    fmt.Println("count =", count) // always 1000
}
```

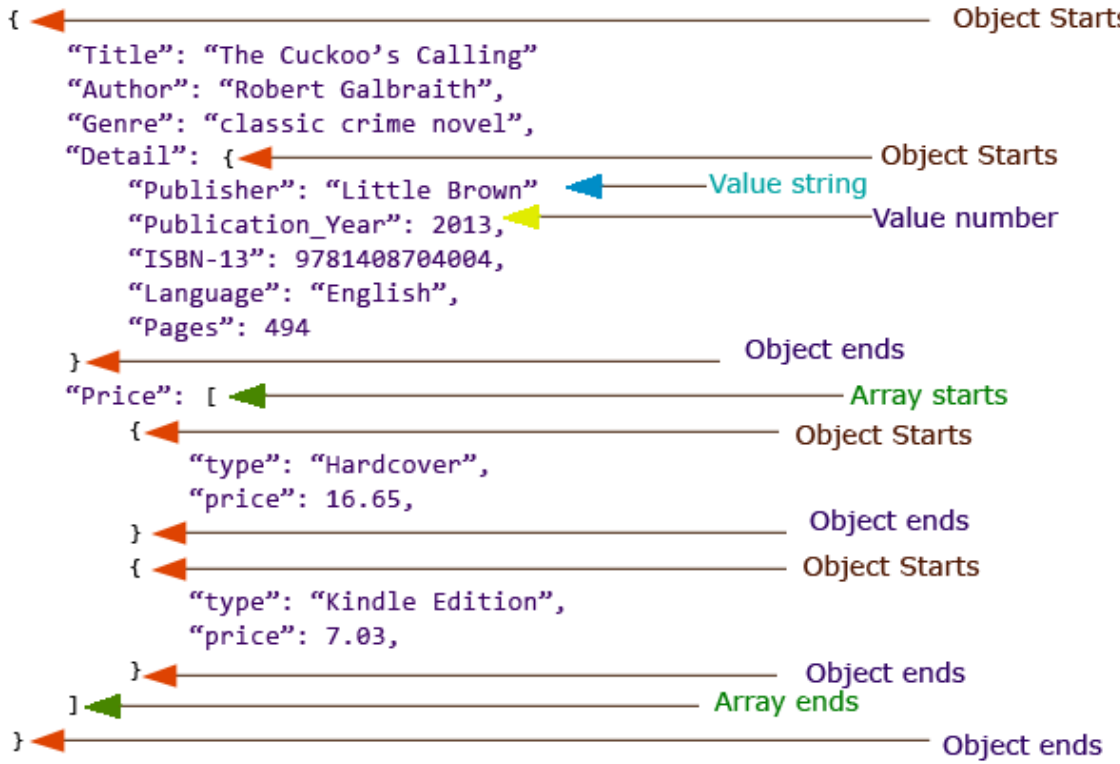
Golang Syntax: Conversions

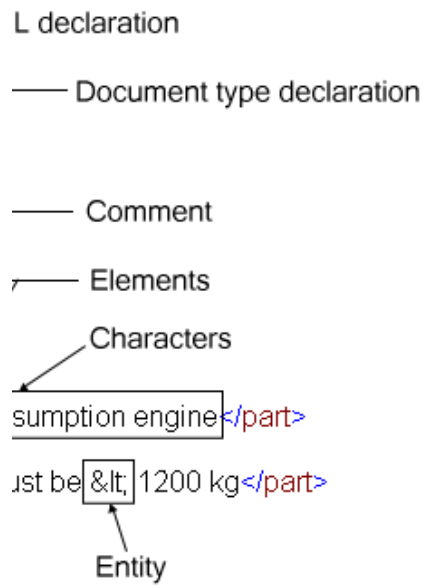
```go
import "strconv"

// String to Int
num, _ := strconv.Atoi("42") // 42

// Int to String
str := strconv.Itoa(42) // "42"

// String to Float
price, _ := strconv.ParseFloat("19.99", 64) // 19.99

// Float to String
str = strconv.FormatFloat(3.14, 'f', 2, 64) // "3.14"

// String to Bool
flag, _ := strconv.ParseBool("true") // true
```

Golang Syntax: XML

- **Library**

Encoding/xml





Golang Syntax: JSON

- **Library**

encoding/json