Mind Storm

Go Language

Romin Irani | Jan 31 2015 | @iRomin | romin.k.irani@gmail.com

Mind Storm

Objectives

- What is the Go Language all about?
- Whirlwind Tour of the Language
- See lots of code samples
- Hopefully … make you code a little bit too in this session
- Get you interested enough to learn more about Go!

Any Prerequisites?

Mind Storm

Hands-on Exercises

https://goo.gl/eV2L7O

Mind Storm

Why use Go?

- Modern
- Popular
- Simple
- Fun
- Do More With Less

Mind Storm

Go Language History

- Started in 2007 @ Google
- Initial Release in 2009
- Rob Pike, Robert Griesemer, Ken Thompson
- Designed to overcome issues with using Java, Python and other languages across large code base
- Systems Language → General Purpose Language
- Current @ 1.5
- Version 1.6 to be released in Feb 2016

Mind Storm

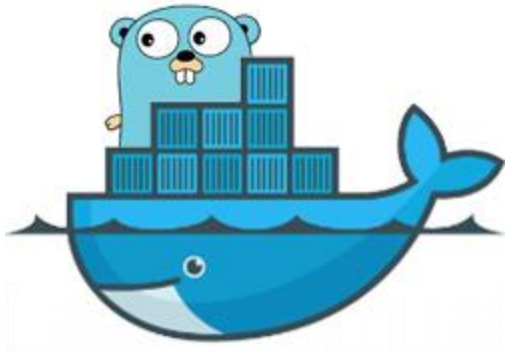
Go Features

- Similar to C Language in Syntax
- Case sensitive
- Only 25 keywords
- Cross Platform Binaries
- Compiled Language
- Statically Typed
- Solid and comprehensive Standard Packages
- Object Oriented Features
  - Composition over Inheritance
- Open Source

Mind Storm

Popular Projects

- Popular with Infrastructure Software Companies
- Popular Projects
  - Docker
  - CoreOS: etcd, rkt
  - Kubernetes
  - InfluxDB, RethinkDB
  - Hugo
  - Consule
  - And more
- Great choice for a modern, general purpose language



Mind Storm

Go Home Page

www.golang.org

The Go Programming Language

Documents | Packages | The Project | Help | Blog | Search

Try Go Pop-out

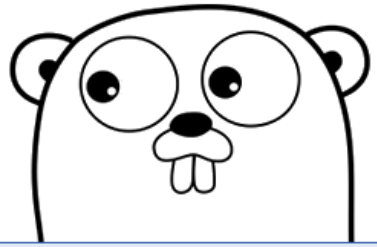
```
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

func main() {
        fmt.Println("Hello, 世界")
}
```

Hello, World! ▾ Run Share Tour

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.



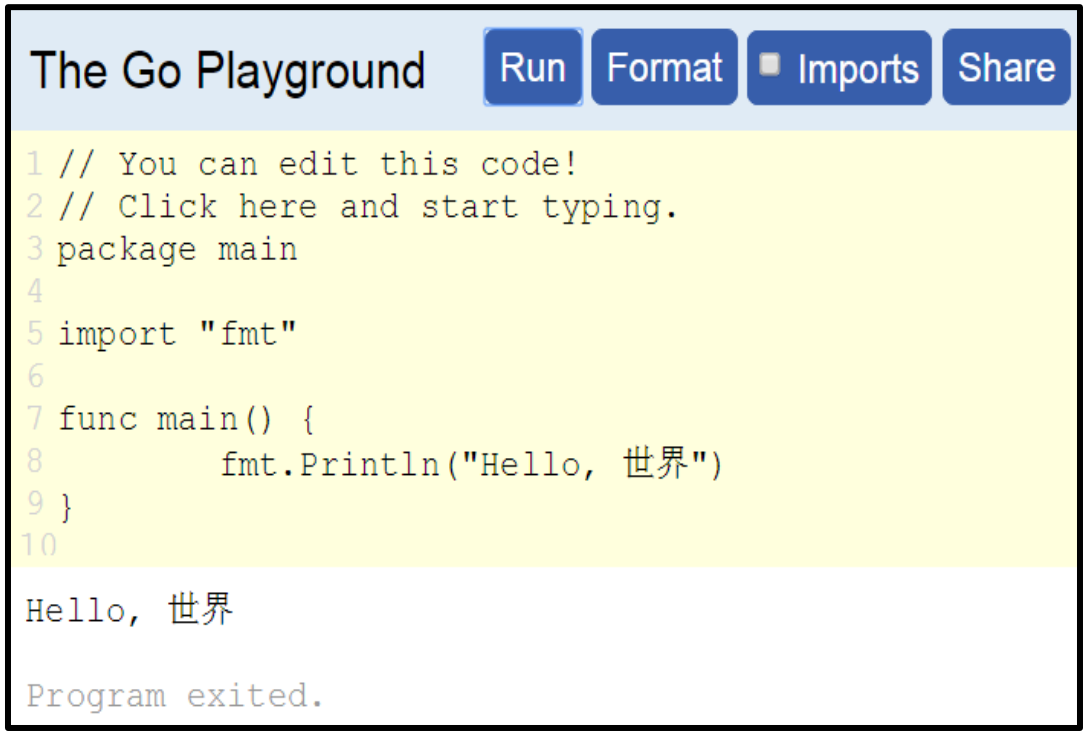
Download Go

Binary distributions available for Linux, Mac OS X, Windows, and more.

Mind Storm

Go Playground

- Go Playground : http://play.golang.org
  - Excellent way to get familiar with syntax
  - Run Go code in the cloud
  - Save and Share Go Snippets
  - Some restrictions



Mind Storm

First Go Project

- We will be creating a new Project
- Let us call it **helloworld**
- Create this project folder in the **GOPATH** root i.e. **$GOPATH**\helloworld
- Open Atom Editor. Add the **$GOPATH**\helloworld Project Folder
- Create a new file named **hello-world.go**

Mind Storm

Hello World in Go

hello-world.go

```go
package main

import "fmt"

func main() {
  fmt.Println("Hello World")
}

```

Mind Storm

Running our Application

- go run <filename>
- go run hello-world.go

```
$>go run hello-world.go
Hello World
```

Mind Storm

Built-in Types

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64 uintptr
- byte // alias for uint8
- rune // alias for int32
  // represents a Unicode code point
- float32 float64 complex64 complex128

Mind Storm

Variable Declaration

- Use the var keyword
- Format

  var <variablename> <type>

  var <variablename> <type> = <initialvalue>

- Examples:

var phoneModel string = “Samsung S5”
var osName string = “Android”
var price float32 = 40000.0

Mind Storm

Variable Declaration

- Use the block declaration to combine multiple variables

```
var (
    phoneModel string = “Samsung S5”
    osName string = “Android”
    price float32 = 40000.0
)
```

Mind Storm

Variable Declaration

- Declare multiple variables of the same type as follows:

var firstName, lastName string = “A”, “B”

var i,j int, k string =10,20,”Hello”

- If initializer is present, the type can be omitted:

var i,j = 10,20

Mind Storm

Short Variable Declaration

- Use the := short assignment statement instead of var
- The type is implicit i.e. determined by Go
- Available only inside a function and not at package level

```
function main() {
        price := 20.50
        firstName := “Gopher”
}
```

Constants

- Declared with the **const** keyword
- They can be character, string, boolean or numeric
- The scope depends on where they are defined

```
const tax_rate = 10.5
const city = “Mumbai”
```

Mind Storm

Go : Zero values

- Every type has a Zero value

Zero Value	Type
0	Numeric
false	Boolean
“”	String
nil	Pointer, channel, struct, func, interface, map, Slice

Mind Storm

Go : Arithmetic Operators

Operator	For?
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Mind Storm

Go : Logical Operators

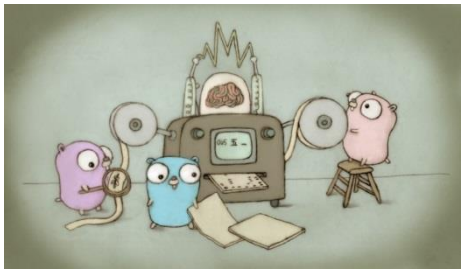
Operator	For?
&&	Boolean AND
||	Boolean OR
!	NOT
==	Test equality

Mind Storm

Reference Types

- Go also has the following Reference Types
  - Arrays
  - Maps
  - Slices
  - Structs
  - Channels
- We will look at them later

Hands On Exercise #1 & #2



Hello World: http://play.golang.org/p/achXqgsH1v
Variables, Constants: http://play.golang.org/p/dQuYzSgR0P

Mind Storm

What are functions?

- Functions are the building blocks of Go programs
- An independent section of code
- Also known as procedure or subroutine
- Can take input parameters
- Can return single or multiple return values
- Variable number of arguments
- Functions are first class citizens in Go. They can be passed around as any other value.
- Anonymous Functions

Mind Storm

Writing a function

- You have already seen the main() function in action

hello-world.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    fmt.Println("Hello World")
7  }
8
```

Mind Storm

Writing a function

- Define your function by specifying the following:

```
func funcname() {
}
```

- This is the simplest way to define a function.
- It does not take any input parameters
- It does not return any values
- E.g.

```
func sayHello() {
    fmt.Println("Hello")
}
```

Mind Storm

Function with input parameters

- Define your function by specifying the following:

  func *funcname(param1 type, param2 type)* {
  }

- This function takes two input parameters
- It does not return any values
- E.g.

```
func sayHello(firstName string, lastName string, age int) {
    fmt.Println("Hello",firstName,lastName,"You
are",age,"years old")
}
```

Mind Storm

Function with input parameters

- If multiple parameters are of same type, you can use a short-hand way of declaring:

  func *funcname(param1,param2 type1, param3 type2)* {
  }

- f1(num1,num2 int, name string)
- f2(num1 int, firstname, lastname string)

Mind Storm

Function with return values

- A function can return a value : single or multiple
  func *funcname(param1,param2 type1, param3 type2) (return value types)* {
  }

- f1(num1 int, num2 int) (int) { return 0}
- f2(num1 int, firstname, lastname string) (int, string) {
  return 1, “Hello”}
- f3(num1 int, num2 int) (int, bool) { return false}
- f4(num1 int, num2 int) (int, error) {
  return 10, nil
  //return -1, errors.New(“Some error”)
  }

Mind Storm

Variadic functions

- The last input parameter can contain multiple or variable number of arguments
- They have to be of the same type

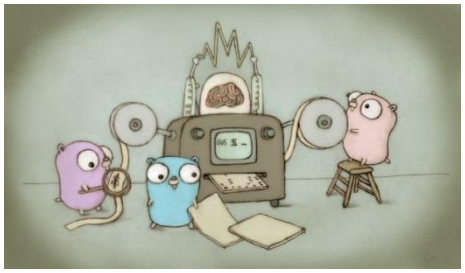
func *funcname(param1 type1, ...string)* {
}

E.g.

```
func addNumbers(numbers ...int) (int) {
}
```

Invoking it:
addNumbers(1,2,3,4,5)
addNumbers(1,2)
addNumbers(1,2,3,4,5,6,7,8,9,10)

Hands On Exercise #3



http://play.golang.org/p/Z85-wKYITI

Mind Storm

for construct

- The only looping construct available in Go
- 3 variants
  - Single condition
  - Classical For Loop : Initial Value, Final Value, Increment
  - Forever Loop

Mind Storm

for : Single Condition

- for <condition> {
  }
- Repeatedly executes the block while the condition is true

Example:

```
for i < 3 {
        fmt.Println(i)
        i++
}
```

Mind Storm

for : Classical Loop


```
for i := 1; i<=10; i++ {
        fmt.Println(i)
}
```

- i:=1 , both declares, initializes the variable i

Mind Storm

for : forever loop


```
for  {
        //do something
        //if some condition met , then use break
}
```

- Similar to while true { … } in other languages
- Break out of the loop via the **break** or **return** statement

Mind Storm

Sample Code

- Go Playground:
  http://play.golang.org/p/Z7bKxJ-IjK

if statement

- if statement is a condition followed by a block
- The block is executed only if the condition evaluates to true

```
if <condition> {
        //Some statements
}

if (sum > 100) {
        …
}
```

Mind Storm

if – else statement


```
if <condition> {
        //block 1
} else {
        //block 2
}


a:=1
b:=2
if a > b {
        fmt.Println("a is greater than b")
} else {
        fmt.Println("a is less than or equal to b")
}
```

Mind Storm

If-else-if statement


```
if <condition 1> {
        //block 1
} else if <condition 2> {
        //block 2
} else if <condition 3> {
}
else {
}
```

switch statement

- switch statement can help in evaluating multiple conditions
- The **switch** keyword is followed by an expression and then multiple **case** statements, out of which one will be executed
- The case data types could be int, float, string, bool, etc. or even a complex expression / function call
- Unlike other languages, there is no need for the **break** statement after each **case** statement.
- A **default** case statement can be put too in case none of the case conditions are met.

Mind Storm

switch statement


```
switch i {
    case 0: fmt.Println("Zero")
    case 1: fmt.Println("One")
    case 2: fmt.Println("Two")
    case 3: fmt.Println("Three")
    case 4: fmt.Println("Four")
    case 5: fmt.Println("Five")
    default: fmt.Println("Unknown Number")
}
```

Mind Storm

switch statement

- Notice that there was no break statement between the case statements
- If you want the code to fallthrough like it is in under languages, use **fallthrough**

Mind Storm

switch statement

- Use commas to separate multiple expressions in same case statement.

```
switch i {
    case 1,3,5,7,9: fmt.Println(“Odd")
    case 2,4,6,8,10: fmt.Println(“Even")
    default: fmt.Println("Unknown Number")
}
```

Mind Storm

switch – without expression

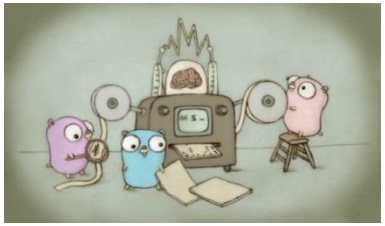
- Eliminate the expression in switch statement. This way, it can evaluate the case expression

```
switch {
    case i < 10: fmt.Println(“Less than 10")
    case i>=10 && i <=100 : fmt.Println(“Between 10 and 100")
    case somefunc(i) > 200 : fmt.Println(“some statement”)
    default: fmt.Println("Unknown Number")
}
```

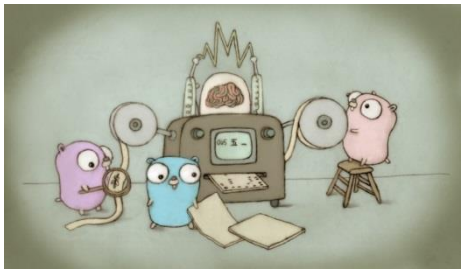
Mind Storm

Sample Code

- Go Playground:
  - if / if - else: http://play.golang.org/p/bA0qVEx_eP
  - switch :
    - http://play.golang.org/p/wjwTpPLujg
    - http://play.golang.org/p/lQQIiLzC1G
    - http://play.golang.org/p/5rj5Y1hVPR



Hands On Exercise #4



Arrays in Go

- Fixed length data type for holding elements of same type
- Array can be of any types like int, string, struct, pointers, etc
- Individual element of Array can be accessed using integer index, using []. For e.g. num[0]
- Array index is 0 bound that if Array size is 5, index ranges from 0 to 4

Mind Storm

Declaring / Initializing Arrays

- var iArr = [5]int{}
  - Creates an Array of length 5
  - Each element of Array initialized to zero value i.e. 0

- iArr := [5]int {0, 1, 2, 3, 4}

  Declare and initialize Array iArr of length 5, and with values 0, 1, 2, 3, 4

- iArr := [5]int {2: 20, 4: 50}:

  Declare and initialize Array iArr of length 5, and will set value at index 2 and 4 to 20 and 50 respectively

Mind Storm

Accessing Array elements

- iArr[2], will access element at index 2 of Array iArr
- iArr[3] = 10, will set 10 as value at index 3
- If index is more than length of Array, then an error is thrown. For e.g. iArr[5] for an integer Array of 5 elements will thrown the follow error:

  "invalid Array index 5 (out of bounds for 5-element Array)"

Mind Storm

Iterating through Array

- Use the **for** or **range** statement

```
var numbers = [5]int{1,2,3,4,5}
for i:=0;i<5;i++ {
    //numbers[i]
}
```

Use built-in **len** function for length of Array. For e.g.

```
for i:=0;i<len(numbers);i++ {
    //numbers[i]
}
```

Mind Storm

Iterating through Array

- **for + range statement** allows you to iterate over a collection.
- It provides both index and value of each item

```
var numbers = [5]int{1,2,3,4,5}
for index,value := range numbers {
    fmt.Println(index,value)
}
```

Output:

0,1

1,2

2,3

3,4

4,5

Mind Storm

Iterating through Array

- **You can ignore any of the index or value , if not interested**

```
var numbers = [5]int{1,2,3,4,5}
for _,value := range numbers {
    fmt.Println(value)
}
```

```
var numbers = [5]int{1,2,3,4,5}
for index, _ := range numbers {
    fmt.Println(index)
}
```

Question : Why do you need to ignore any of the return values that you are not interested in?

Mind Storm

Array Sample code

1. Multiple Array examples
   http://play.golang.org/p/p-eV7eZXYp
2. Pass by Value to function
3. Using for and range to iterate over Array
   http://play.golang.org/p/nN1Q3R0Z1X
4. A first look at Slices:
   http://play.golang.org/p/WarnTGxDaE

Mind Storm

Arrays : Practical Considerations

- Fixed in Length
- Arrays are passed by value to function
  - If the Array data is large in size (or multi-dimensional), impact memory and performance
- Go has a solution to addressing this that we shall see in the next section : **Slices**

Slices - Overview

- Slice data type is similar to an Array
- Just like an Array, a Slice too holds elements of same type
- It does not have a fixed length
- A Slice allows for dynamic expansion and shrinking of the Array
- It is an abstraction built on top of an Array type and shares same memory as that of underlying Array

Mind Storm

Declaring / Initializing

- Use **make** function for creating a Slice

  make([]T, length, capacity)

  - **T** is the type of element to hold
  - **length,** of Slice to be created. It is mandatory to specify length
  - **capacity**, of Slice to be created. It is optional to specify capacity. In case if not specified , capacity is assumed to be same as length

Mind Storm

Declaring / Initializing

- Slice := make([]int, 5, 10)

  This will create Slice of type **int**, of length 5 and capacity 10.

- Slice := make([]int, 5)

  This will create Slice of type int, of length 5 and capacity 5.

Declaring / Initializing

- Slice is used in same way as Array.

  Slice[0] = 1

  Slice[1] = 2

Setting values at index 0 and 1 into Slice

- Use function **len** to find length of Slice
- Use function **cap** to find capacity of Slice

Mind Storm

append function

- It is always recommended to use **append** method while adding value(s) to Slice
- Function **append**, internally checks if value to be added in more than the length, it will create new Array using capacity and add new value.
- Call to **append** always returns new Slice

Mind Storm

Slicing Arrays

- So far we created Slice using **make** function, which also creates underlying Array for Slice
- But we can also create Slice on top of Array

  Let us assume Array i := [5] int {1, 2, 3, 4, 5}

  - Slice := i[1:3]

    This will create Slice over Array i, using values starting from index 1 to 3 i.e. 2, 3, 4.

  - Slice := i[:]

    This will create Slice covering entire Array i.e. from index 0 to 4.

Mind Storm

Declaring / Initializing

- Slice := [] int {1, 2, 3, 4, 5}

  Creating Slice in same way as Array. But here we don't specify length in the preceding square brackets

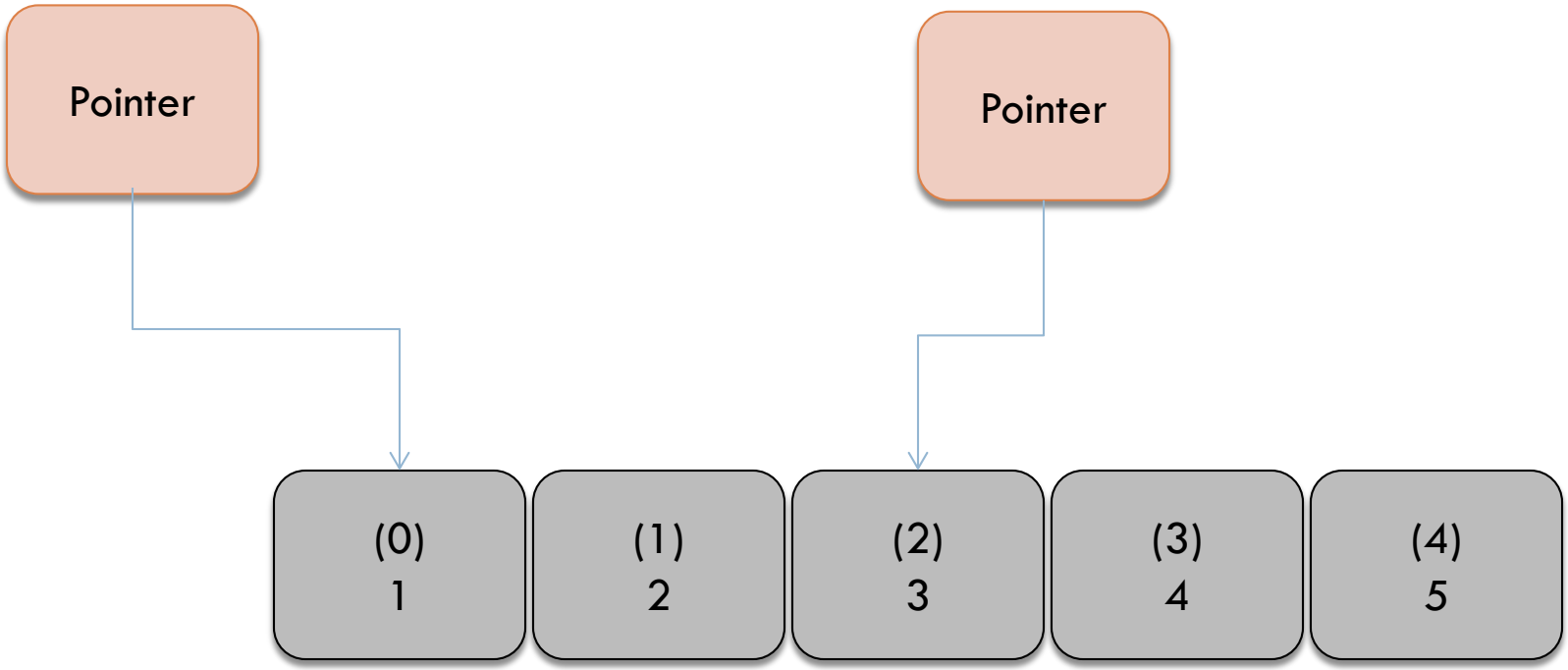
- Empty Slices:
  - Slice := []int {}
  - Slice := make ([]int, 0)
- Creating new Slice from Slice
  - Slice2 := Slice[1: 3]

Mind Storm

Slice Representation

Slice := [] int {1, 2, 3, 4, 5}

new_Slice := Slice[2 : 4]



Mind Storm

Key points while using Slices

- Slice is a reference to an Array
- All the Slices on an Array shares same memory as that of Array
- Can create more than one Slice on same underlying Array
- Any changes made to one Slice will be reflected in other Slices and in Array too
- Useful to use a slice instead of entire Array in case of large amount of data

Mind Storm

Iterating over Slices

- Use the **range** keyword

```
numbers := []int{100, 200, 300, 400, 500}
for index, value := range numbers {
        fmt.Println(index, value)
}
```

- Go Playground
  http://play.golang.org/p/P_nSXzxAC1

Mind Storm

Examples

- Slice from **make** function
  http://play.golang.org/p/sGQ2WOT1S3
- Slice from Literal
  http://play.golang.org/p/R60h2f2xt9
- append, len and capacity functions
  http://play.golang.org/p/gUPJNCeIX0
- Slices are passed by value to functions
  http://play.golang.org/p/nluakS7P-s
- Multiple Slices work over the same array
  http://play.golang.org/p/BsHTCIiECU
- Iterating over Slices
  http://play.golang.org/p/6Ax7L6Ud7E

Map

- Only key/value paired collection in Go
- Map is an unordered collection i.e. order in which key/value returned is not always the same
- Map, uses a hash table for storing key/value

Mind Storm

Declaring/Initializing

- Use function **make** for creating Map

  make(**map**[key type]value type)

  In make function, use keyword **map**, specify the type to be used as key in square brackets and type to be used as value outside square bracket

  m := make(map[int]string)

  This will create a Map, with key as int and value as string

Mind Storm

Declaring/Initializing

- We can also create without using make function

  m := map[int]string{}

- It is also possible to declare and initialize map in one statement

  m := map[int]string {

  1: “One”,

  2: “Two”,

  3: “Three”,

}

Mind Storm

Using Map

- Write to map

  m[4] = “Four”

  This will add, key 4, and assign “Four” as it is value
- Read from map

  s := m[2]

  Will return value for key 2 i.e. “Two”.
- In case if key is not present in map, it will return zero value for type of value.

  In our case if we use m[100] and since key 100 is not present in map, empty string will be returned as for string, empty string is zero value

Using Map

- Function **len**, can be used to determine length/size of the map
- Function delete, to be used to deleting key/value from map

  delete(m, 1)

  Will delete key/value pair whose key is 1 from map m.

  If m is nil or key is not present in map, delete will be no operation

Mind Storm

Using Map

- i, ok := m[1]

  Here 2 values will be returned, 1st the value for key and 2nd boolean to indicate if key/value exists

- i := m[100]

  Here only value for the key will be returned

- _, ok := m[100]

  This can be used to check if key/value exists. We have used **blank Identifier** (Underscore) as 1st value to skip it

Mind Storm

Iterating a Map


```
m := map[int]string{
    1: “One”,
    2 : “Two”,
    3:  “Three”,
}
```

We can iterate over map using **range** as shown below

```
for k, v : range m {
    fmt.Println(“Key :”, k, “ Value :”, v)
}
```

Two values are returned, 1st is key and 2nd is value.

Examples

- Creating a Map

http://play.golang.org/p/8P-IR6XXe0

Accessing Map (Example 1)

http://play.golang.org/p/nSh2EpFsGa

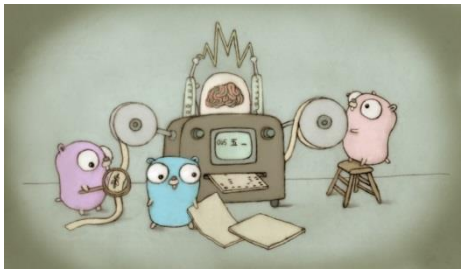
- Accessing Map (Example 2)

http://play.golang.org/p/VQ0EZJoCWt

- Function

http://play.golang.org/p/YNtjqHYCbd

Hands On Exercise #5



Structs

- Struct is user defined data type
- Struct, as the name says is a structure used for logical grouping of fields/property
- If you are familiar with OO language Java, you can co-relate it with **class**
- A Struct, like most other data types when passed as parameter to function is **passed by value**

Declaring/Initializing

- Following is the template used for declaring a struct

  type <name> **struct** {

  }

  keyword **type** is used to define a new type followed with name of the struct, followed by keyword **struct** to indicate that type is a **struct**

- **Example**

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

Declaring/Initializing

- Just like any other data type, A Struct too can be declared and initialized in two steps

  var u User

  This will declare variable of **u** of type struct **User** and initialize it to **zerovalue**.

- We can do declaration and initialization in one statement as shown below:

  u := new(User)

  Using **new** will allocate memory and return pointer.

Mind Storm

Declaring/Initializing

- We can pass values for each of the property to be initialized at the time of creating an instance of struct.

  u := User{“Fn”, “Ln”, 50}

  Here we have passed value as **Fn, Ln** and **50** which will be set to **FirstName, LastName** and **Age** respectively.

- We can also specify property name explicitly in case if we don’t wish to initialize all the properties or if we don’t want to follow sequential approach

  u := User{Age: 50, FirstName: “Fn”}

Accessing

- We can access individual fields as follows:

  u := User{Age: 50, FirstName: “Fn”}

  Here we have not initialized **LastName** while creating instance of **User**. Lets do it now:

  u.LastName = “Ln”

- Use Println, for printing a Struct including all it fields

  fmt.Println(u)

  This will print: {Fn Ln 50}

Nested Struct


```
type Address struct {
        Building, Area, City, State, Country string
}
type User struct{
        …
        Address  Address
}

u := User{Age: 50, FirstName: "Fn", LastName:"Ln"}
u.Address = Address{"building", "area", "city", "state", "INDIA"}
fmt.Println(u)
```

Output:

```
{Fn Ln 50 {building area city state INDIA}}
```

Struct : Receiver Methods

- Method is a function which is called upon instance. For example:

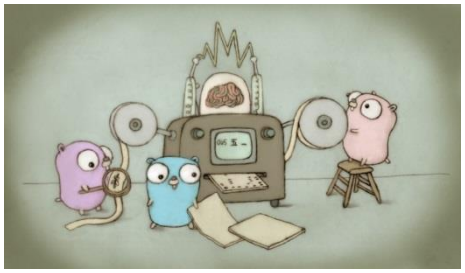
```
func (u User) isSeniorCitizen() bool{
        isSrCitizen := false
        if (u.Age > 60){
                isSrCitizen = true
        }
        return isSrCitizen
}
```

Note the signature, we are preceding the function name with instance of user. This is called as receiver and function is called as method on User.

We can call this method as

```
u := User{Age: 50, FirstName: "Fn", LastName:"Ln"}
isSrCitizen := u.isSeniorCitizen()
```

Hands On Exercise #6



http://play.golang.org/p/TRt2wOuBkT

Interfaces

- Interface is a way to add behaviour
- It is GoLang's way to Polymorphism
- Interface is a contract that implementer needs to fulfill
- Interface only defines methods with signature without definition. Definition of methods is left up-to implementer
- Interface leads to IS-A relationship
- You can add behaviour anytime by implementing the Interface method. This is one of the nice features of the language

Declaring


```
type <name> interface{
        ....
}
```

- Interface is declared using keyword **type** followed by name followed by keyword **interface**
- Example

```
type Shape interface {
        printArea()
}
```

Here we have declared interface of type Shape with method printArea, which needs to be implemented by implementer

Implementation


```
type Square struct {
    side int
}

func (s Square) printArea (){
    fmt.Println("Area of
    Square", s.side * s.side)
}
```

```
type Rectangle struct {
        length, breath int
}

func (r Rectangle ) printArea(){
    fmt.Println("Area of
    Rectangle", r.length *
    r.breath)
}
```

Implementation


```
func main(){
        var s, r Shape
        s = Square{5}
        r = Rectangle{5, 4}
        s.printArea()
        r.printArea()
}
```

Output:

Area of Square 25

Area of Rectangle 20

We have created instance of Square and Rectangle and called printArea() method on each of them.

Mind Storm

Composition over Inheritance

- Go does not support inheritance
- It supports Composition via Type Embedding
- The design philosophy is to compose large things from smaller components
- Composition over Inheritance helps to keep the components loosely coupled even as changes happen in the system + helps with practical benefits of inheritance

Mind Storm

Composition Example

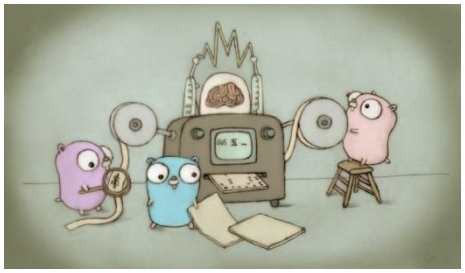
- Composition is Go is done via Embedding the Type
- The embedding struct gets the behaviour of the embedded Type
- It can also access the struct elements directly
- It can override the behaviour of Embedded Type if needed
- Though Composition is HAS-A , it is easier to think of via IS-A

Mind Storm

Interfaces + Type Composition

- You can define an Interface with its methods
- A Type (Struct) can implement it.
- You can then embed that type into any struct
- The Embedding Struct can override behaviour if needed
- You can treat all the instances like IS-A type of that particular Interface

Hands On
Exercise #7 & #8



http://play.golang.org/p/toycLtAVxz

http://play.golang.org/p/i1Hpf2YA5-

Mind Storm

Concurrency in Go

- What is concurrency?
- Concurrency v/s Parallelism
- Go routines
- Channels
- See sample code to see it work

Mind Storm

Concurrency v/s Parallelism

"… concurrency is the *composition* of independently executing processes, while parallelism is the simultaneous *execution* of (possibly related) computations. Concurrency is about *dealing with* lots of things at once. Parallelism is about *doing* lots of things at once."

— Rob Pike

Mind Storm

Threads Anyone?

- Usually the execution of things concurrently is done via an OS Thread
- On a single core CPU, only one thread can be executing at a single time
- Threads are resource hungry and not that lightweight
- Go does not use Threads
- Instead it uses go routines. Let's check that.

Mind Storm

go routines

- It is a light weight construct
- Go manages go routines
- Go routines are laid on top of threads
- Switching Threads is expensive
- Since multiple go routines could be in a thread, switching is less expensive , therefore less scheduling of Threads
- Faster startup times
- Communicating between go routines is via Channels

Mind Storm

Step 1 : Running Sequentially

- Go Playground
  http://play.golang.org/p/OCHJV_xbI0
- This code invokes two functions , one after the other
- There is no concurrency here

Mind Storm

Step 2 : Introduce go routines

- Go Playground
  http://play.golang.org/p/VZqsEyZbKG
- Simply put the keyword go before the function call
- This will invoke the two functions concurrently
- Investigate : What happened?
- The program just exited
- This is because the main routine ended immediately
- Suggested Fix : Maybe a timer ?

Mind Storm

Step 3 : Introduce a Timer

- Go Playground
  http://play.golang.org/p/dsno9tkdb0
- Introduced the time.Sleep method in both the go routines so that each one of them allows the other to run
- Additionally, we have put a longer timer in the main routine so that there is sufficient time for the go routines to complete
- Issue : We cannot keep playing with the timer

Mind Storm

Step 4 : Introducing sync.WaitGroup

- Go Playground
  http://play.golang.org/p/vavJveNVer
- Use a WaitGroup from sync package
- In the main routine, you can specify how many go routines need to finish and one waits for that
- Each go routine will indicate to the Wait Group that is done

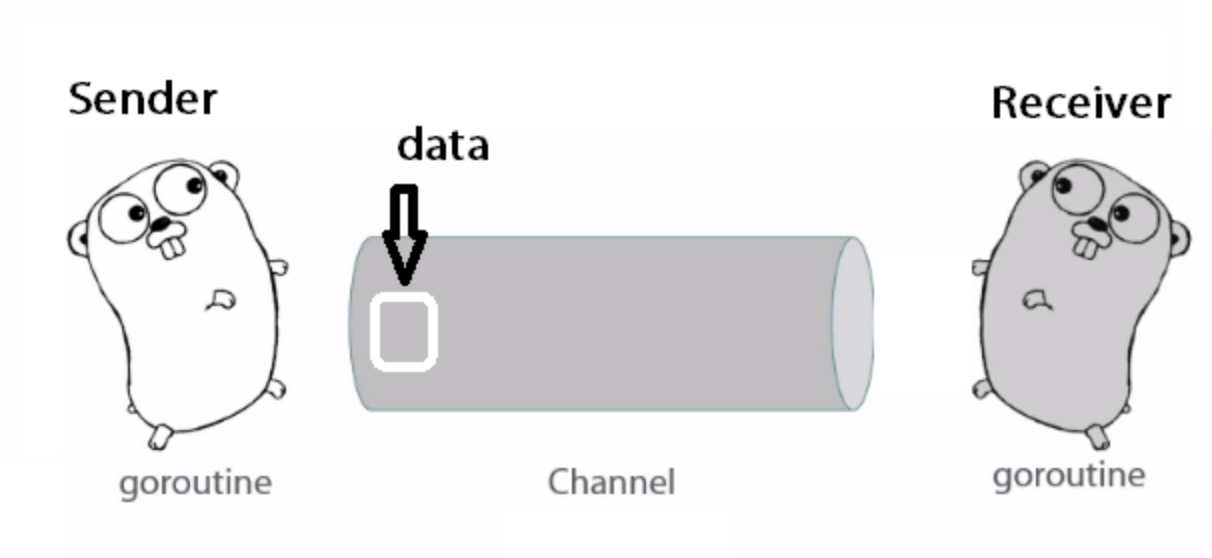
Go Channels

- Synchronization between go routines is done via Channels
- A Channel can be thought of as a pipe that connects the go routines
- Communication means “exchange of data”
- Channels take care of safely transmitting data between the go routines and takes care of sync operations
- Channels
  - Buffered
  - Un-buffered (**Default**)

Mind Storm

Unbuffered Channel

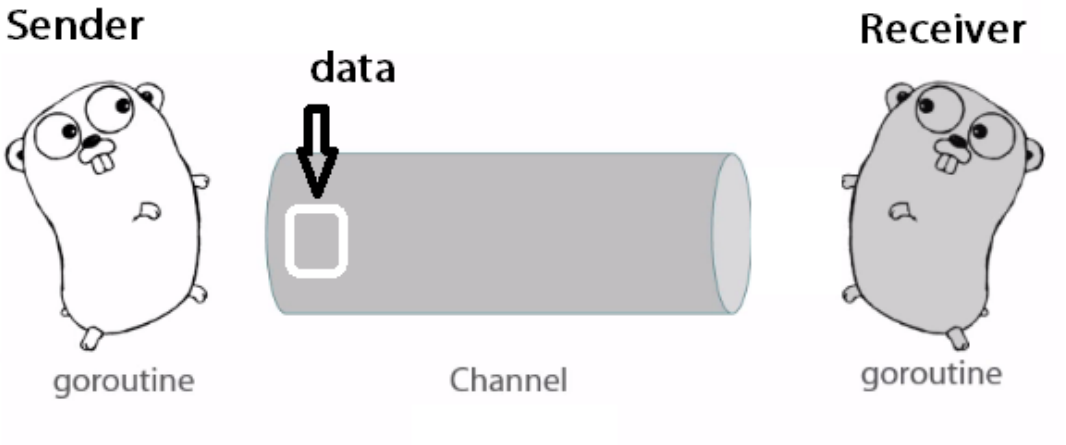
- Creation
  - ch1 := make(chan <datatype>)
  - Writing to it : ch1 <- somedata
  - Reading from it : somedata <- ch1



Mind Storm

Unbuffered Channel

- Sender go routine will write data to the Channel
- Receiver go routine will wait on the Channel
- The Sender blocks till the Receiver gets the data
- The Receiver blocks till it receives the data
- Unbuffered Channel = Combines Communication with Sync



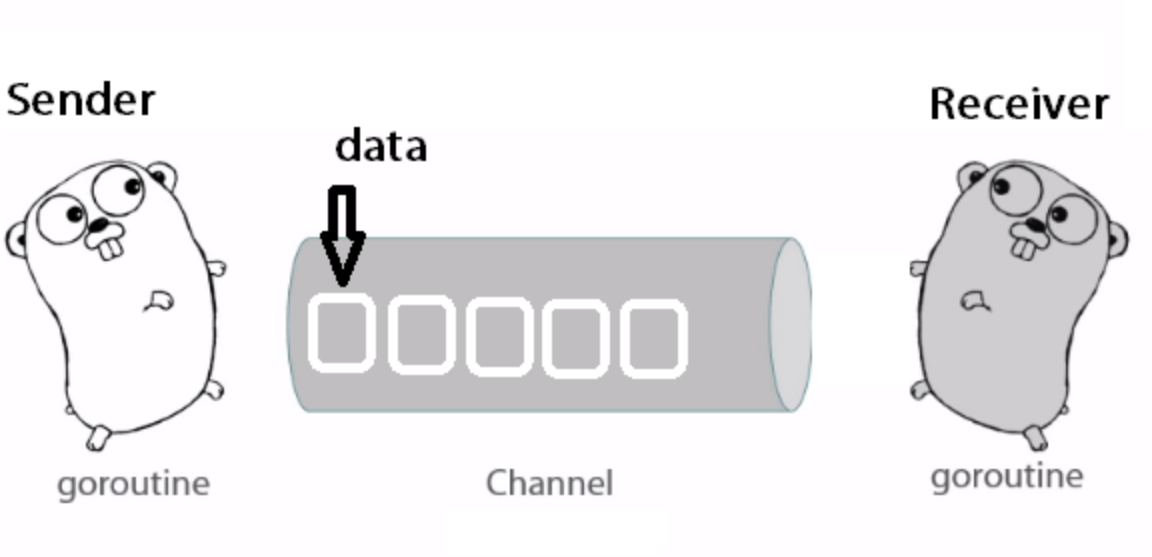
Mind Storm

Unbuffered Channel : Example

- Go Playground
  http://play.golang.org/p/T8_4wz0fjw
- Main routine (RECEIVER) waits for a message on the channel
- go routine (SENDER) – completes it's task and sends the message on the channel
- Try: Comment out the go f1() line and see what happens when you try to run!

Buffered Channel

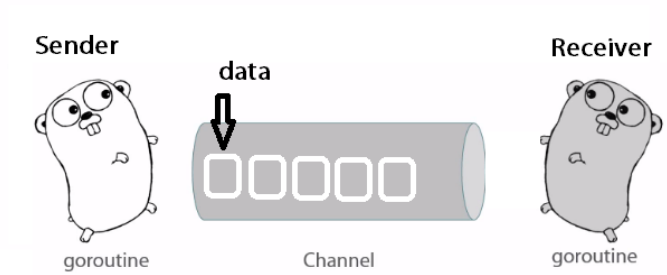
- Creation
  - ch1 := make(chan <datatype>, **size**)
  - Writing to it : ch1 <- somedata
  - Reading from it : somedata <- ch1



Mind Storm

Buffered Channel

- Sender go routine will write data to the Channel
- Receiver go routine will wait on the Channel
- The number of data blocks that can be written depend on size
- The Sender go routine will not block after writing if the buffer is still available
- The Receiver blocks till it receives the data
- Buffered Channel = Semaphore to limit throughput



Buffered Channel : Example

- Go Playground: http://play.golang.org/p/ft03nGBshb
- Create a channel that accepts a string with a buffer size of 2
- Main routine can wait for both go routines to write to that channel
- Try : Uncomment the extra fmt.Println(<-messageChannel) line

Buffered Channel : Example

- Go Playground: http://play.golang.org/p/E3YikZ8UEv
- Channel that accepts string data type and buffer size of 5
- Go routine sends the 5 messages and since it is buffered, it is not blocked till the RECEIVER gets it.
- Try : Introduce a timer if you want to sleep in main routine and get the message later.

Mind Storm

Understand Buffered Channels

- Go Playground: http://play.golang.org/p/4Cbc7BfE7s
- The channel size is 5
- Notice how the go routine (SENDER) after publishing 5 messages to the Buffered Channel BLOCKS on the 6th message till the RECEIVER starts reading from the channel

Mind Storm

Go & Web Development

- Well suited to power web applications
- Sweet spot lies in creating Web APIs that can be consumed by clients
- Core packages : net/http and html/template
- Web Frameworks:
  - Martini
  - Revel
  - Others

Mind Storm

net/http Package

- Powerful standard library to handle Request / Response pattern
- The key component here is http.HTTPHandler , which encapsulates the Request / Response

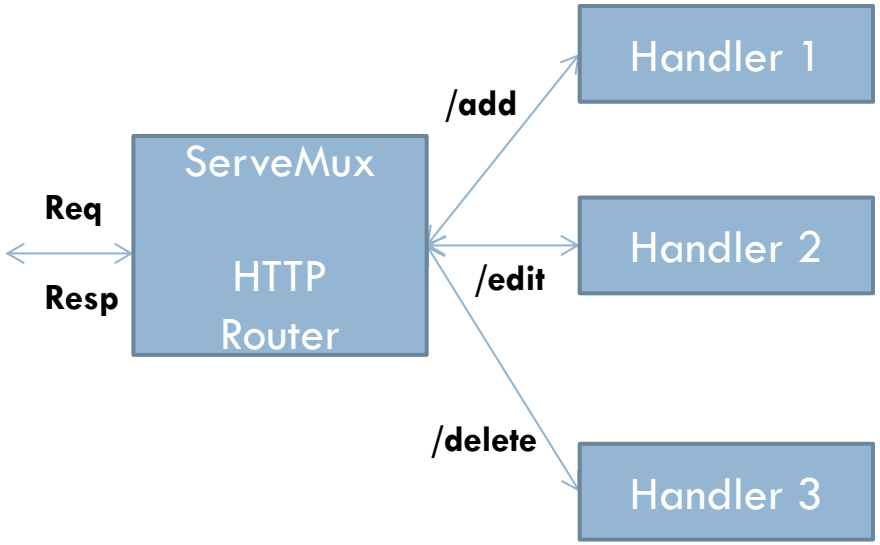
Various HTTP Handlers

- net/http provides several HTTP Handlers like FileServer, NotFoundHandler, etc.
- Example : FileServer can be used for Static File Serving
- http.FileServer(root FileSystem)
- Using http.ListenAndServe(port,handler) – we can serve a static site
- Create the following file http://play.golang.org/p/KyRDyXSpmZ in any server and run the application there. Visit localhost:8080 in your browser.

Mind Storm

net/http Package

- We need to typically map multiple URL Paths to their HTTP Handlers
- For e.g.
  - /add -> Handler 1
  - /edit -> Handler 2
  - /delete -> Handler 3
- ServeMux in net/htttp



Sample REST API

- Write a REST API for managing Employees
- To keep it simple, we will implement single GET method call /employees to get all employees
- Go Playground
  http://play.golang.org/p/ged5yNOekx

Mind Storm

HTTP Networking - Invoking REST API

- Use the net/http package
- Package http provides HTTP client and server implementations
- Get, Head, Post, and PostForm methods to make HTTP (or HTTPS) requests
- resp, err := http.Get("http://example.com/")
- resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)
- resp, err := http.PostForm("http://example.com/form", url.Values{"key": {"Value"}, "id": {"123"}})

Sample Code

- USD to INR Currency Rate
- Invokes a Currency REST API that returns JSON Data
- http://apilayer.net/api/live?access_key=API_KEY&currencies=**INR**&format=1
- Go Playground
  - Step 1 : Retrieve JSON Data : http://play.golang.org/p/H5-JX-1VEP
  - Step 2 : Parse the JSON Data : http://play.golang.org/p/Fp8S78V5UC

Mind Storm

Several Other things …

- Unit Testing
- Go Tools
- Go Doc
- Take a look at : https://github.com/avelino/awesome-go

Naming conventions

- Create the test file with name ending in _test.go
- Inside the test file, each test is written in the following form:

  func TestXyz(t *testing.T)
- **go build** ignores the files ending in _test.go
- **go test** will take all the files ending in _test.go
- **go help test**

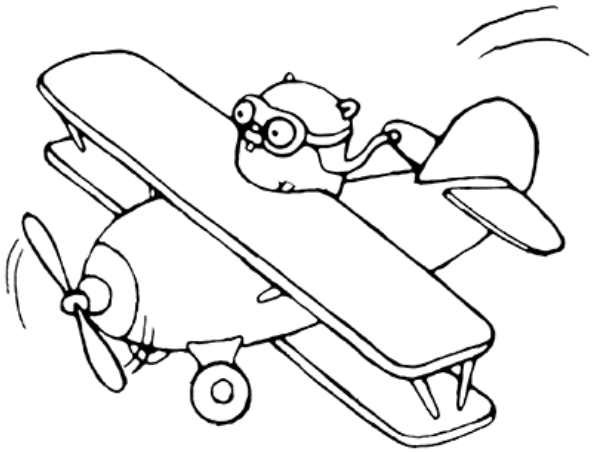
Mind Storm

Go Documentation

- Go provides godoc tool for generating documentation from source files
- You can use it to get information on packages too
- See all documentation
  godoc -http=:4000
- See specific documentation on a package
  godoc math
- See specific documentation on a package + function
  godoc strconv Atoi

Mind Storm

Thank You



- Q & A
- Website : http://www.rominirani.com
- Email : romin.k.irani@gmail.com
- Twitter : @iRomin