

Go Lang Beginner's Workshop

Sathish VJ

Installation

- Download zip file for your platform
 - Downloads: <https://code.google.com/p/go/downloads/list>
- Extract it to a known location, say `[x]/golang`
 - You should now have `[x]/golang/go` directory
- Set `PATH` to `[x]/golang/go/bin`
- Check version on the command line with
`go version`
- Set `GOROOT` to `[x]/golang/go`
- Set `GOPATH` to `[x]/golang/mygopath`
 - Will be useful later (not today) when installing external packages with `go get` and `go install`
- Ref: <http://golang.org/doc/install>

Documentation

- Documentation available at <http://golang.org/pkg>
- Try `go doc fmt` or `go doc regexp` for documentation on specific packages
- If I'm slowing down for others, entertain yourself at <http://tour.golang.org>
- To run documentation locally:
`godoc -http=:6060`
Then access via browser at: <http://localhost:6060>

1. Hello World

```
package main

import (
    "fmt"
)

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

Use any editor.

Save as `hw.go`

Run it as `go run hw.go`

Hello World

All functionality in a package. The main program has to be in 'main' package.

```
package main
```

```
import (  
    "fmt"  
)
```

Packages of functionality are 'imported'

Programs (generally) start at this function signature.

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

Notice capital "P". Try what happens with "fmt.println"

Use dot notation to access package functions. Note capitalization of 'P' in Print. Note that all package names are in small.

Hello World

No semicolons

```
package main
```

```
import (  
    "fmt"  
)
```

Note naming convention. Small readable, variables that sound right

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

Functions/methods defined with "func" keyword

Immediately observable benefits

- Very fast building
- Compiled into native executable; real fast execution
- Works equally well on multiple platforms
- No virtual machines and repeated VM update reminders
- Clean, clear, concise code
- Familiar programming concepts

Go addresses ...

- Computers fast but software construction slow.
- Dependency analysis necessary for speed, safety.
- Types get in the way too much.
- Garbage collection, concurrency poorly supported.
- Multi-core seen as crisis not opportunity.

2. Variables

```
package main

import "fmt"

var gi int

func main() {
    fmt.Println(gi) //0

    var i int
    fmt.Println(i) //0
    i = 25
    fmt.Println(i) //25

    j := 5
    s := "Hello!"
    fmt.Println("The two values are:", j, s)
    //The two values are: 5, Hello

    fmt.Printf("The integer is %d, and the string is %s.\n", j, s)
    //The integer is 5, and the string is Hello.

    var arr1 []int
    arr1 = []int{1, 2, 3, 4}
    arr2 := []int{1, 2, 3, 4}
    fmt.Println(arr1, arr2) //[1,2,3,4] [1,2,3,4]
}
```

Variables

```
package main
```

```
import "fmt"
```

Inverse of other ways of declaration. C/Java - `int i = 10`

```
var gi int
```

```
func main() {
```

```
    fmt.Println(gi) //0
```

Default/Zero values. String="", Int=0, Float=0.0, etc.

```
    var i int
```

```
    fmt.Println(i) //0
```

```
    i = 25
```

```
    fmt.Println(i) //25
```

`:=` is definition and initialization

```
    j := 5
```

```
    s := "Hello!"
```

`Println` automatically formats data type for printing

```
    fmt.Println("The two values are:", j, s)
```

```
    //The two values are: 5, Hello
```

Formatted prints with `Printf`

```
    fmt.Printf("The integer is %d, and the string is %s.\n", j, s)
```

```
    //The integer is 5, and the string is Hello.
```

```
    var arr1 []int
```

```
    arr1 = []int{1, 2, 3, 4}
```

```
    arr2 := []int{1, 2, 3, 4}
```

```
    fmt.Println(arr1, arr2) //[1,2,3,4] [1,2,3,4]
```

Arrays and how to initialize them

```
}
```

3. Functions

```
package main

import (
    "fmt"
)

func Add(i, j int) int {
    return i + j
}

func main() {
    s := Add(5, 10)
    fmt.Println("Sum is: ", s)
}
```

3. Functions

```
package main
```

```
import (  
    "fmt"  
)
```

Function takes parameters in a similar format - *name type*
Combine many: *(s1 string, s2 string, i int, j int) = (s1, s2 string, i, j int)*

```
func Add(i, j int) int {  
    return i + j  
}
```

Awesomeness: Functions that are visible outside the package begin with a capital letter.

```
func main() {  
    s := Add(5, 10)  
    fmt.Println("Sum is: ", s)  
}
```

You can also ignore return values with an underscore.
`_ := Add(5, 10)`

3.5 Hello Web

```
package main

import "fmt"
import "net/http"

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, world")
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

Save as [hweb.go](#)

Run it as [go run hweb.go](#)

In your browser, go to <http://localhost:8080>

Hello Web

```
package main
```

```
import "fmt"
```

```
import "net/http"
```

can also be written as `import ("fmt" "net/http")` //on separate lines

write your response into this

request data

```
func handler(w http.ResponseWriter, r *http.Request) {  
    fmt.Fprint(w, "Hello, world")  
}
```

handler function follows the defined function signature: `func HandleFunc(pattern string, handler func(ResponseWriter, *Request))`

```
func main() {  
    http.HandleFunc("/", handler)  
    http.ListenAndServe(":8080", nil)  
}
```

nil as empty value (not null)

Start built in server. No external server required!

4. for loop

```
package main

import (
    "fmt"
)

func main() {
    arr := []int{1, 2, 3, 4}

    fmt.Println("\nWithin for loop ...")
    for i := 0; i < len(arr); i++ {
        fmt.Println(i)
    }

    j := 0
    fmt.Println("\nWithin infinite for loop ...")
    for {
        if j > len(arr) {
            break
        }

        fmt.Println(j)
        j = j + 1
    }
}
```

4. for loop

```
package main
```

```
import (  
    "fmt"  
)
```

```
func main() {  
    arr := []int{1, 2, 3, 4}
```

```
    fmt.Println("\nWithin for loop ...")
```

```
    for i := 0; i < len(arr); i++ {  
        fmt.Println(i)  
    }
```

For loop similar to Java, C
(No parentheses though)

```
    j := 0
```

```
    fmt.Println("\nWithin infinite for loop ...")
```

```
    for {  
        if j > len(arr) {  
            break  
        }
```

Infinite loop.

(Can also give some parts). *for i:=0;; { ... }*

Use *break* to get out of current loop.

Use *continue* to go to next loop index.

```
        fmt.Println(j)  
        j = j + 1  
    }
```

```
}
```


5. struct and object representation

```
package main
import "fmt"

type MyCar struct {
    color    string
    maxSpeed int
}

func main() {
    m := MyCar{}
    fmt.Println(m) //{ 0}

    m = MyCar{"red", 100}
    fmt.Println(m) //{red, 100}

    m.color = "blue"
    m.maxSpeed = 150
    fmt.Println(m) //{blue, 150}
    fmt.Println("color is:", m.color)
    //color is: blue

    m = MyCar{maxSpeed: 150, color: "green"}
    fmt.Println(m) //{green, 150}
}
```

5. struct and object representation

```
package main
import "fmt"
```

Use the *type ... struct {}* keywords

```
type MyCar struct {
    color    string
    maxSpeed int
}
```

Member variable definitions

```
func main() {
    m := MyCar{}
    fmt.Println(m) //{ 0}
```

Initializing an empty struct instance. Default/zero values assigned.

```
m = MyCar{"red", 100}
fmt.Println(m) //{red, 100}
```

Assigning values to member variables in order of struct declaration.

```
m.color = "blue"
m.maxSpeed = 150
fmt.Println(m) //{blue, 150}
fmt.Println("color is:", m.color)
//color is: blue
```

Use dot operator to access individual member variables.

```
m = MyCar{maxSpeed: 150, color: "green"}
fmt.Println(m) //{green, 150}
```

Mix order or drop some variables with named variable definitions

6. struct methods

```
package main
import "fmt"

type MyCar struct {
    speed int
}

func (m *MyCar) acc() {
    m.speed = m.speed + 10
}

func main() {
    m := MyCar{}
    fmt.Println(m)

    m.acc()
    fmt.Println(m)
}
```

6. struct methods

```
package main
import "fmt"

type MyCar struct {
    speed int
}

func (m *MyCar) acc() {
    m.speed = m.speed + 10
}

func main() {
    m := MyCar{}
    fmt.Println(m)

    m.acc()
    fmt.Println(m)
}
```

Functions 'associated' with a type ... not physically in lexical scope of type.

Same dot notation to reach member methods.

Pointers automatically resolved ... note that `acc()` requires a pointer to a `MyCar` instance.

7. multiple assignment,error handling

```
package main

import (
    "fmt"
    "strconv"
)

func SumProd(i, j int) (int, int) {
    return i + j, i * j
}

func main() {
    s, p := SumProd(5, 6)
    fmt.Println(s, p)

    arr := []string{"Hello", "how", "are", "you?"}
    for i, v := range arr {
        fmt.Println(i, v)
    }

    a := "20a"
    if _, err := strconv.Atoi(a); err != nil {
        fmt.Println("Error! ", err)
    }
}
```

multiple assignment, error handling

```
package main
```

```
import (  
    "fmt"  
    "strconv"  
)
```

Package to use in converting to and from string

Multiple return values should be in parentheses

```
func SumProd(i, j int) (int, int) {  
    return i + j, i * j  
}
```

Named return values possible
Eg. *func f() (sum, prod int)*

```
func main() {  
    s, p := SumProd(5, 6)  
    fmt.Println(s, p)
```

Accepting variables are in same order

```
    arr := []string{"Hello", "how", "are", "you?"}  
    for i, v := range arr  
    {  
        fmt.Println(i, v)
```

"range" keyword gives a *key, value* pair.
For arrays, it is *index* and *value at index*
For maps (hashtables), it is *key* and *value of key*

```
    a := "20a"  
    if _, err := strconv.Atoi(a); err != nil {  
        fmt.Println("Error! ", err)
```

The same multiple return value is used in an error checking paradigm

```
}
```

8. Testing

```
package main
```

```
import (  
    "fmt"  
    "strconv"  
)
```

Package to use in converting to and from string

Multiple return values should be in parentheses

```
func SumProd(i, j int) (int, int) {  
    return i + j, i * j  
}
```

Named return values possible
Eg. *func f() (sum, prod int)*

```
func main() {  
    s, p := SumProd(5, 6)  
    fmt.Println(s, p)
```

Accepting variables are in same order

```
    arr := []string{"Hello", "how", "are", "you?"}  
    for i, v := range arr  
    {  
        fmt.Println(i, v)
```

"range" keyword gives a *key, value* pair.
For arrays, it is *index* and *value at index*
For maps (hashtables), it is *key* and *value of key*

```
    a := "20a"  
    if _, err := strconv.Atoi(a); err != nil {  
        fmt.Println("Error! ", err)
```

The same multiple return value is used in an error checking paradigm

```
}
```

8. Testing

```
package mymath

func Add2(i, j int) int {
    return i + j
}
```

- create dir
my`math`
- save file as
add2.go

```
package mymath

import (
    "testing"
)

func Test_Add2(t *testing.T) {
    s := Add2(5, 10)
    if s != 15 {
        t.Errorf("FAIL: Expected 15. Received %d.", s)
    } else {
        t.Logf("PASS: Expected 15, also received %d.", s)
    }

    s = Add2(15, 10)
    if s != 25 {
        t.Errorf("FAIL: Expected 25. Received %d.", s)
    }
}
```

- save file as
add2_test.go
- go test

8. Testing

add2_test.go

File has to end in `_test.go`

```
package mymath
```

```
import (  
    "testing"  
)
```

Import package *testing* to run tests.

Test functions have to begin with *Test_*

```
func Test_Add2(t *testing.T) {
```

Use a reference to **testing.T* in all tests. These functions will be called by the testing framework.

```
    s := Add2(5, 10)
```

```
    if s != 15 {
```

```
        t.Errorf("FAIL: Expected 15. Received %d.", s)
```

```
    } else {
```

```
        t.Logf("PASS: Expected 15, also received %d.", s)
```

```
    }
```

Use functions *Fatalf*, *Logf*, *Errorf* to report test statuses.

```
    s = Add2(15, 10)
```

```
    if s != 25 {
```

```
        t.Errorf("FAIL: Expected 25. Received %d.", s)
```

```
    } else {
```

```
        t.Logf("PASS: Expected 25, also received %d.", s)
```

```
    }
```

```
}
```

Questions

Thank You

Sathish VJ