Hello World in Go

```
package main } Package declaration. (*)

import "fmt" (*) (*)

// this is a comment

func main () {
    fmt.Println ("Hello, world")  — there is no semi-colon
}
```

(*) Packages are Go's way of organising and reusing code. There are two types of Go programs: executables and libraries. Executable application are the kind of program that we can run directly from the terminals. ~~Library~~ Libraries are collection of code that we package together so that we can use them in other programs.

(*) (*) The fmt package (short hand for format) implements formatting for input and output.

(*) running Go program

Go to project directory,

```
$ go run .
```

■ Tutorial followed from Official documentation

• Create a module

Declare a greeting package to collect related function

package greetings

import "fmt"

// function Hello returns a greeting for the named person

func Hello (name string) string {

function Name — parameter type — Return Type

// Return a greeting that embeds the name in a message

~~message := fmt.Sprintf("Hi", %v, "Welcom!",~~

message := fmt.Sprintf("Hi, %v. Welcome", name)

return message

Sprintf function used to create a formatted string.

}

■ Call your ~~module~~ code from another module

Directory Structure

<home>/

|-- greetings/

|-- Hello/

We call "greetings" from Hello.go

Hello.go

Declare a main package. In Go, the code executed as an application must be in the 'main' package. ~~This gives your~~

~~package main~~ package main

~~import~~ import (

"fmt"

"example.com/greetings") import two packages. This gives your code access to functions in that package.

)

func main() {

// Get a greeting message and print it.

message := greetings.Hello("Gladys")

fmt.Println(message)

}

- Some commands for production use

Return and handle an error

1. In greetings/greeting.go , add the following code

```
/* # There is no sense in sending a greeting back if you don't
   know who to greet. Return an error to the caller if the
   name is empty. Copy the following code into greetings.go and
   save the file

*/

package greetings

import (
    "errors"  } errors package for error handling.
    "fmt"
)

func Hello (name string) (string, error) {
    if name == "" { // if name is empty
        return "", errors.New("empty name")
    }
    // if the name was recieved, return a value that
    // embeds the name in greeting message

    message := fmt.Sprintf("Hi, %v Welcome!", name)
    return message, nil } second value in a successful return
}
```

no brackets or parenthesis in if statement

return "" → return empty

errors.New("empty name") → show this error

nil error i.e. everything is fine.

2. In your hello/hello.go file, handle the error now returned by the Hello function, along with the non-error value.

Hello.go

```
package main

import (
    "fmt"
    "log"
    "example.com/greetings"
)

func main() {
    // Set properties of a predefined Logger, including
    // the log entry prefix and a flag to disable printing
    // the time, source file and line number.
    log.SetPrefix("greeting: ")
    log.SetFlags(0)

    // Request a greeting message
    message, err := greeting.Hello("")
    // If an error was returned, print it to the console
    // and exit the program
    if err != nil {
        log.Fatal(err)
    }
    // If no error was returned, print the returned message
    // to console
```

fmt.Println(message)

}

Return a random greetings

In this section, we will change the code so that instead of returning a single greeting everytime, it returns one of the several predefined messages.

To do this, we will use a Go slice. A slice is like an array, except that the size changes dynamically as we add and remove items. The slice is one of Go's most useful types.

We will add a small slice to contain three greeting messages then have your code return one of the messages randomly. For more on Slice, see Go slices in the Go blog.

greetings.go

```
package greetings

import (
    "errors"
    "fmt"
    "math/rand"
    "time"
)

func Hello(name string) (string, error) {
    if name == "" {
        return "", errors.New("empty Name")
    }
    message := fmt.Sprintf(randomFormat(), name)
    return message, nil
}
```

A slice can be created with built-in function called 'make' which has the signature.

```
func make([]T, len, cap) []T
```

len → length
cap → capacity (optional, when it is omitted, it defaults to length)

```
var s []byte
s = make([]byte, 5, 5)
// s == []byte{0, 0, 0, 0, 0}
```

Zero value of a slice is 'nil'. The len and cap functions will both return 0 for a nil slice

- Slice Internals

A slice is a descriptor of an array segment. It consists of a pointer to the array, the length of the segment.

"slicing" a slice

```
s = []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// s[1:4] = []byte{'o', 'l', 'a'}
```

from 1 → upto 4 (excluding 4)

- Growing slices

To increase the capacity of a slice one must a new, a larger slice and copy the contents of the original slice into it. This technique is similar to dynamic array implementation in other languages.

doubling the capacity of slice.

```
t := make([]byte, len(s), (cap(s)+1)*2)
copy(t, s)
s = t
```

A common operation is to append data to the end of a slice. This function appends byte elements to a slice of bytes, growing the slice if necessary, and returns the updated slice value :-

```
func AppendByte ( slice []byte, data ...byte) []byte {
        m := len(slice)
        n := m + len(data)
        if n > cap(slice) {  // if necessary, reallocate
            newSlice := make( []byte, (n+1) * 2)
            copy (newSlice, slice)
            slice = newSlice
        }
        slice = slice [0:n]
        copy ( slice [m:n], data)
        return slice
}
```

Variable number of arguments
name 'data' of type 'byte'

One could use AppendByte like this :-

```
p := []byte { 2, 3, 5}
p = AppendByte (p, 7, 11, 13)
// p == []byte { 2, 3, 5, 7, 11, 13}
```

To append one slice into another, use ... to expand the second argument to a list of arguments

```
a := []string {"John", "Paul"}
b := []string {"George", "Ringo", "Pete"}
a = append(a, b...) // equivalent to append(a, b[0], b[1], b[2])
```

A possible "gotcha"

Re-slicing a slice doesn't make a copy of the underlying ar
The full array will be kept in the memory until it is
no longer referenced. Occasionally this can cause the progra
to hold all the data in memory when only a small pie
of it is needed.

Ex. FindDigits function loads a file into memory and
searches it for the first group of consecutive numeric digits,
returning them as ~~arrays~~ new slice.

```
var digitRegexp = regexp.MustCompile("[0-9]+")

function FindDigits (~~filename~~ filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return digitRegexp.Find(b)
}
```

This fix this problem one can copy the interesting data
to a new slice before returning it:

```
func CopyDigits (filename String) []byte {
    b, _ := ioutil.ReadFile(filename)
    b = digitRegexp.Find(b)
    c := make([]byte, len(b))
    copy(c, b)
    return c
}
```

Return greetings for multiple people

Here, we will add support for getting greeting for multiple people in one request. In other words, we will handle a multiple-value input, then pair values in that input with a multiple value output. To do this, we need to pass a set of names to a function that can return a greeting for each of them.

- greetings.go

```
package greetings
import (
    "errors"
    "fmt"
    "math/rand"
    "time"
)

// Hellos returns a map that associates each of the named
// people with a greeting message
func Hellos(names []string) (map[string]string, error) {
    // A map to associate names with messages
    messages := make(map[string]string)
    // loop through the recieved slice of name, calling the
    // Hello function to get a message for each time.
    for _, name := range names {
        message, err := Hello(name) // Hello function which
                                    // we defined previously
                                    // Didn't write it again here
                                    // but it's there.
        if err != nil {
            return nil, err
        }
        // In the map, associate the retrieved message to the name
        messages[name] = message
    } return messages, nil
```

- hello.go

```
package main

import (
    "fmt"
    "log"

    "example.com/greetings"
)

func main() {

    log.SetPrefix("greetings: ")
    log.SetFlags(0)

    // a slice of names
    names := []string{"Gladys", "Samantha", "Darrin"}

    messages, err := greetings.Hellos(names)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(messages)
}
```

A Tour Of Go

- Exported Names

In Go, a name is exported if it begins with a capital letter. For example, Pizza is an exported name, as is Pi which is exported from the math package.

pizza and pi do not start with a capital letter, so they are not exported.

When importing a package, you can refer only to its exported names. Any "unexported" names are not accessible from outside the package.

↪.

```
import math

func main() {
    fmt.Println(math.Pi)  // math.pi will show error
}
```

- Functions

notice that the type comes after the variable name

```
func add(x int, y int) int {
    return x+y
}
```

(x int, y int) — parameters; int — return type

When two or more consecutive named function parameters share a type, you can omit the type from all but the last

```
func add(x, y int) int {
    return x+y
}
```

A function can return multiple results

```
func swap(x, y string) (string, string) {
    return y, x
}
```

```
a, b := swap('a', 'b')
```

Named return value

Go's return values may be named. If so, they are treated as variables defined at the top of the function.

These names should be used to document the meaning of the return values.

A return statement without arguments returns the named return values. This is known as a "naked" return.

Naked return statements should only be used only in short functions. They can harm readability in the longer function.

```
func split (sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

Variables

The var statement declares a list of variables as in function argument lists, the type is last.

A var statement can be at package or function level.

A variable declaration can include initializers, one per variable

If an initializer is present, the type can be omitted, the variable will take the type of the initializer.

```
var i, j int = 1, 2
var c, python, java = true, false, "no!"
```

Short variable declaration

Inside a function, the := short assignment statement can be used in place of a var declaration with implicit type.

Outside a function, every statement begins with a keyword (var, func and so on) and so the := construct is not available.

```
func main() {
    k := 3
}
```

Basic Types

Go's basic types are

bool int int8 int16 int32 int64
string uint uint8 uint16 uint32 uint64 uintptr

```
byte // alias for uint8
rune // alias for int32
     // represents a Unicode code point.
```

~~float32~~ float32, float64

complex64, complex128

The int, uint, uintptr types are usually 32 bits wide on 32 bit systems and 64 bits wide on 64-bit systems. When you need an integer value, you should use int unless you have a specific reason to use a sized or unsigned integer type.

Zero values

variable declared without an explicit initial value are given their zero value

The zero value is :-

0 for numeric types

false for boolean

"" (empty string) for string.

Type conversion

```
i := 42
f := float64(i)
u := uint(f)
```

Constants

Constants are declared with const keyword

Constant can be character, string, boolean or numeric val

Constants cannot be declared using the := syntax.

```
package main
import "fmt"
const Pi = 3.14
func main() {
    const Truth = "true"
    const Word = "Word"
}
```

for loop

init statement. Executed before the first iteration

post statement: executed at the end of every iteration

```
for i := 0 ; i < 10 ; i++ {
    sum += i
}
```

Condition expression. Evaluated before every iteration

- Unlike other languages like C, Java or JavaScript, there are no parenthesis surrounding the three components of the for loop.

Sometimes, the init and post statements are optional

```
for ; sum < 1000 ; {
    sum += sum
}
```

* for is Go's "while"

At that point, you can drop the semicolons. It will make for loop behave like while in Go.

```
for sum < 1000 {
    sum += sum
}
```

* forever

If you omit the loop condition it loops forever, so an infinite loop is compactly expressed.

```
for {       // infinite loop.
}
```

If, else

```
if x<0 {
    ____
}
// Normal if statement
```

Short statement execute before the condition

```
func (n, n, int) {
    if v := math.Pow(n,n); v < 100 {
        return v
    }
    return v
}
// if with a short statement.
```

Condition

this only stays in the scope of if statement

```
func pow (x, n int) int {
    if v := math.Pow(x, n); v < 100 {
        return v
    } else {
        return v/10
    }
    return 0 // can't use v
}
```

v can only be used in the scope of if/else statement

Switch

A switch statement is a multiway branch statement. It provides an efficient way to transfer the execution to different parts of a code based on the value (also called case) of the expression. Go language supports two types of switch statements:

1. Expression switch
2. Type switch

Expression switch

→ Similar to the switch statements in C++ or Java.

Syntax: (both are optional statements)

```
switch optstatement ; optexpression {
    case expression 1: statement....
    case expression 2: statement....
    ......
    default : statement.....
}
```

3

→ Contains simple statements like variable declaration, increment or assignment statements

- If a variable is present in the optional statement then the scope of that variable is limited to that switch statement

- If there is no expression, then the compiler assume that the expression is true.

Go Switch examples

#1

```
func main() {
    fmt.Print("Enter Number: ")
    var input int
    fmt.Scan(&input)
    switch(input) {
    case 10:
        fmt.Print("the value is 10)
        break
    case 20:
        fmt.Print("the value is 20)
    default:
        fmt.Print("it is something else")
}
```

#2

```
func main() {
    // switch statement with both optional statement
    // i.e. day := 4 and expression, i.e. day
    switch day := 4; day {
    case 1:
        fmt.Println("Monday")
        break
    case 2:
        fmt.Println("Tuesday")
        . . . .
    default:
        fmt.Println("Invalid")
    }
}
```

#3

```
func main() {
    var value int = 2
    // switch statement without an optional statement
    // and expression

    switch {
        case value == 1:
            fmt.Println("Hello")
            break
        case value == 2:
            fmt.Println("Bonjour")
            break
        default:
            fmt.Println("Invalid")
    }
}
```

#4

```
func main() {
    var value string = "five"
    // switch statement without default statement
    // Multiple value in case statement

    switch value {
        case "one":
            fmt.Println("C#")
            break
        case "two":
            fmt.Println("Java")
            break
        default:
            fmt.Println("Python")
        case "three", "four", "five":
            fmt.Println("Python")
    }
}
```

using multiple values.