Chapter 1

Objective

- History Of Go Language
- Setting Up Go Compiler
- Configuring GoLang Environment in Linux
- Basic Data Types
- Go program structure
- Variable Declaration
- Variable Declaration within Function Body
- Constants Declaration
- Go Enumerated Values
- The iota Represent Enumerated Values
- Control Flow
- Switch Statement

This page is left blank intentionally

Unit 1

History Of Go Language

- Go originated as an experiment by Google engineers Robert Griesemer, Rob Pike, and Ken Thompson to design a new programming language.

1 person clipped this slide
Who made this thing?
Robert Griesemer, Rob Pike and Ken Thompson
7

- Objective was to resolve common criticisms of other languages while maintaining their positive characteristics.

- The Go programming language was released in late 2007 by Google.
- The language is based on C/C++, Java, and Python.
- Go is a compiled, concurrent, garbage-collected, statically typed language and it is an open source project.

Setting Up Go Compiler

Download Go compiler from the following site:

- https://golang.org/dl/

Or

- https://golang.org/doc/install

Install GoLang in **ubuntu** Linux

- (https://www.tecmint.com/install-go-in-linux/)

- Extract the tar archive files into **/usr/local** directory using the command below.

```
$ sudo tar -C /usr/local -xvzf go1.11.linux-amd64.tar.gz
```

Configuring GoLang Environment in Linux

- Setup your Go **workspace** by creating a directory **~/go_projects** which will hold your workspace.

- The workspace is made of **three directories** namely:

  bin which will contain Go executable binaries.
  src which will store your source files and
  pkg which will store package objects.

Follow command will create the above directory tree as follows:

```
$ mkdir -p ~/go_projects/{bin,src,pkg}
$ cd ~/go_projects
$ ls
```

To compile Golang programs from terminal window using command line, you must add **`/usr/local/go/bin`** path to **$HOME/.profile** file located under user home directory.

- **`export PATH=${PATH}:/usr/local/go/bin`**

Also, set the of **GOPATH** and **GOBIN** environment variables in your user **.profile** file **(~/.profile or ~/bash_profile**) to point to your workspace directory.

```
export GOPATH="$HOME/go_projects"
export GOBIN="$GOPATH/bin"
```

Note: If you installed GoLang in a custom directory other than the default (/usr/local/), you must specify that directory as the value of **GOROOT** variable.

For instance, if you have installed GoLang in home directory, add the lines below to your $HOME/.profile or $HOME/.bash_profile file.

```
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
```

The final step under this section is to effect the changes made to the user profile in the current bash session like so:

- **$ source ~/.bash_profile**

OR

- **$ source ~/.profile**

Verify GoLang Installation

Check GoLang Version and Environment by using following commands:

- **$ go version**
- **$ go env**

Type the following command to display usage information for Go tool, which manages Go source code:

- **go help**

To test if your Golang installation is working correctly type in following golang program using your favorite text editor and save it as hello.go file under ~/go_projects/src/hello/ directory. All your GoLang source files must end with the .go extension.

Create the hello project directory under ~/go_projects/src/:

- **$ mkdir -p ~/go_projects/src/hello**

Use favorite editor to create the hello.go file under hello directory:

- **$ vi ~/go_projects/src/hello/hello.go**

Add the lines below in the file, save it and exit:

```
package main

import "fmt"

func main() {
   fmt.Printf("GoLang installed successfully under Linux\n")
}
```

Compile above program using **`go install hello.go`** which will create **`hello`** executable file under **`~/go_projects/bin`** that you can execute:

- **`$ go install $GOPATH/src/hello/hello.go`**
- **`$ $GOBIN/hello`**

If you see the output showing you the message in the program file, then your golang compiler installation is working correctly.

To run your executable binary from any directory (like other Linux commands) you must add ${GOBIN} to your $PATH environment variable located in .profile file.

Basic Data Types

bool
string

Numeric types:

uint	either 32 or 64 bits
int	same size as uint
uintptr	an unsigned integer large enough to store the uninterpreted bits of a pointer value
uint8	the set of all unsigned 8-bit integers (0 to 255)
uint16	the set of all unsigned 16-bit integers (0 to 65535)
uint32	the set of all unsigned 32-bit integers (0 to 4294967295)
uint64	the set of all unsigned 64-bit integers (0 to 18446744073709551615)
int8	the set of all signed 8-bit integers (-128 to 127)
int16	the set of all signed 16-bit integers (-32768 to 32767)
int32	the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64	the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
float32	the set of all IEEE-754 32-bit floating-point numbers
float64	the set of all IEEE-754 64-bit floating-point numbers
complex64	the set of all complex numbers with float32 real and imaginary parts
complex128	the set of all complex numbers with float64 real and imaginary parts
byte	alias for uint8
rune	alias for int32 (represents a Unicode code point)

```
```

Unit 2

Go program structure

- Go is a simple programming language and it was designed to feel familiar and to stay simple.

- First Go program (hello.go) will print the classic "hello world" message.

```
package main
import "fmt"

func main() {
    fmt.Println("hello world")
}
```

- The package "**main**" tells the Go compiler that the **package** should compile as an executable program instead of a shared library.
- The main function in the package "main" will be the entry point of our executable program.
- When you build shared libraries, you will not have any main package and main function in the package.

- The keyword "import" is used for importing a package into other packages.

- fmt is the name of a package that includes a variety of functions related to formatting and output to the screen.

- The package fmt, does not recompile every time it is used in Go source code that gets changed frequently.

- Bundling code in this way serves 3 purposes:

  1) It reduces the chance of having overlapping names. This keeps our function names short and succinct

  2) It organizes code so that its easier to find code you want to reuse.

  3) It speeds up the compiler by only requiring recompilation of smaller chunks of a program.

- To run the program, save the code in hello-world.go file and use go run command.

```
➢ $ go run hello-world.go
➢ hello world
```

- To compile the program into a executable it must be build into binaries.

```
➢ $ go build hello-world.go
➢ $ ls
➢ hello-world  hello-world.go
```

- We can then execute the built binary directly.

```
➢ $ ./hello-world
➢ hello world
```

Now that we can run and build basic Go programs, let’s learn more about the language.

Variable Declaration

- Two ways to declare variables, outside of the function body or within the function body.

- Use var keyword when declaring the variable outside of the function body one by one.

```
var name string
var age int
var address string
```

- Use var keyword followed by the variable name within the parenthesis as a list of variables with the data type declared last.

```
var (
    name, location  string
    age              int
)
```

- A var declaration can include initializes, one per variable.

```
var (
    name     string = "Ken Thompson"
    age      int    =  75
    location string = "Berkeley"
)
```

- If an initializer is present, the type can be omitted, the variable will take the type of the initializer (inferred typing).

```
var (
    name     = "Ken Thompson"
    age      =  75
    location = "Berkeley"
)
```

- You can also initialize variables on the same line:

```
var (
    name, location, age = "Ken Thompson", "Berkeley", 75
)
```

Variable Declaration within Function Body

Use **:=** (colon with assignment operator) to declare variables inside the function body

```
func main() {
    name, location := "Ken Thompson", "Berkeley"
    age := 75
}
```

A variable can contain any type, including functions:

```
func main() {
    action := func() {
        // do something
    }
    action()
}
```

- Outside a function every construct begins with a keyword (‘var’, ‘func’, and so on)
- Outside a function **:=** construct is **not available**

Constants Declaration

- Constants are declared like variables, but with the `const` keyword.
- A const statement can appear anywhere a var statement can.

- Constants can only be character, string, boolean, or numeric values

- Constants cannot be declared using the “:= “ syntax.

- An untyped constant takes the type needed by its context.

```
const Pi = 3.14
const str string = "constant"


const (
        StatusOK                         = 200
        StatusCreated                    = 201
        StatusAccepted                   = 202
        StatusNonAuthoritativeInfo = 203
        StatusNoContent                  = 204
        StatusResetContent               = 205
        StatusPartialContent             = 206
)

const (
    Pi    = 3.14
    Truth = false
    Big   = 1 << 62
    Small = Big >> 61
)
```

Go Enumerated Values

- In other programming languages enums are provided for following reasons:
  - Grouping and expecting only some related values
  - Sharing common behavior
  - Avoids using invalid values
  - To increase the code readability and the maintainability

- In Go language there is no enums, but it provides a way to imitate them.

Create a type for grouping

```
type Weekday int
```

- Declare related constants for Weekday and assign values

```
const (
   Sunday    Weekday = 0
   Monday    Weekday = 1
   Tuesday   Weekday = 2
   Wednesday Weekday = 3
   Thursday  Weekday = 4
   Friday    Weekday = 5
   Saturday  Weekday = 6
)

fmt.Println(Sunday)
// will display 0

fmt.Println(Saturday)
// will display 6
```

The iota Represent Enumerated Values

- The iota is a numeric universal counter starting at 0
- The iota expression is repeated by the other constants until another assignment or type declaration shows up.

```
const (
   Sunday    Weekday = iota + 1
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
   Saturday
)
```

A basic expression

```
type Timezone int

const (
    // iota: 0, EST: -5
    EST Timezone = -(5 + iota)

    // iota: 1, CST: -6
    CST

    // iota: 2, MST: -7
    MST

    // iota: 3, PST: -8
    PST
)
```

Resetting iota

```
    // iota reset: it will be 0.
    const (
        Zero = iota  // Zero = 0
        One          // One = 1
    )
    // iota reset: will be 0 again
    const (
        Two = iota   // Two = 0
    )

    // iota: reset
    const Three = iota // Three = 0
```

Skipping some values

- iota increases by 1 after each line except empty and comment lines.

```
    type Timezone int

    const (
        // iota: 0, EST: -5
        EST Timezone = -(5 + iota)
        // _ is the blank identifier
        // iota: 1
        _
        // iota: 2, MST: -7
        MST
        // iota: 3, PST: -8
        PST
    )

    type Timezone int

    const (
        // iota: 0, EST: -5
        EST Timezone = -(5 + iota)
        // On a comment or an empty line, iota will not increase
        // iota: 1, CST: -6
        CST
        // iota: 2, MST: -7
        MST
        // iota: 3, PST: -8
        PST
    )
```

Using iota in the middle

```
const (
    One   = 1
    Two   = 2
    // Three = 2 + 1 => 3; iota in the middle
    Three = iota + 1
    // Four  = 3 + 1 => 4
    Four
)
```

Multiple iotas in a single line

- In the same line, all constants will get the same iota values.

```
const (
    // Active = 0, Moving = 0, Running = 0
    Active, Moving, Running = iota, iota, iota
    // Passive = 1, Stopped = 1, Stale = 1
    Passive, Stopped, Stale
)
```

Producing using iota Alphabets

```
const (
    // string will convert the expression into string.
    // or, it'll assign character codes.
    a = string(iota + 'a') // a
    b                      // b
    c                      // c
    d                      // d
    e                      // e
)
```

Unit 3

Control Flow

The if Statement

- The `if` statement looks as it does in C or Java, except that the **`( )`** are gone and the **`{ }`** are required.
- The `if` statement can start with a short statement to execute before the condition.
- Variables declared by the statement are only in scope until the end of the `if`.

```
if answer != 42 {
    return "Wrong answer"
}

if err := foo(); err != nil {
    panic(err)
}
```

- Branching with if and else in Go is straight-forward.

```
if 7%2 == 0 {
        fmt.Println("7 is not even")
    } else {
        fmt.Println("7 is odd")
    }
```

- Variables declared inside an if short statement are also available inside any of the else blocks.

```
if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

- Note that you don't need parentheses around conditions in Go, but that the braces are required.
- There is no **conditional operator (? :)** in Go language and therefore use if statement for conditional logic.

The switch statement patterns

- A switch statement runs the first case equal to the condition expression.
- The cases are evaluated from top to bottom, stopping when a case succeeds.
- If no case matches and there is a default case, its statements are executed.

Basic switch with default

```
package main
import "fmt"

func main() {

// Basic switch.
    i := 2
    fmt.Print("Write ", i, " as ")
    switch i {
        case 1:
            fmt.Println("one")
        case 2:
            fmt.Println("two")
        case 3:
            fmt.Println("three")
    default:
        fmt.Println("No match found.")
    }
```

No condition

- A switch without a condition is the same as switch true.

```
switch hour := time.Now().Hour(); { // missing expression means "true"
        case hour < 12:
           fmt.Println("Good morning!")
        case hour < 17:
           fmt.Println("Good afternoon!")
        default:
           fmt.Println("Good evening!")
}
```

- The switch without an expression is an alternate way to express if/else logic.
- Show how the case expressions can be non-constants.

```
t := time.Now()
switch {
  case t.Hour() < 12:
      fmt.Println("It's before noon")
  default:
      fmt.Println("It's after noon")
}
```

Case list

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

```
switch c {
    case ' ', '\t', '\n', '\f', '\r':
        return true
}
```

- The default statement is optional default.

```
switch time.Now().Weekday() {
  case time.Saturday, time.Sunday:
      fmt.Println("It's the weekend")
  default:
      fmt.Println("It's a weekday")
}
```