

Golang

Introduction

- Go is a statically typed, compiled programming language designed at **Google** by **Robert Griesemer**, **Rob Pike**, and **Ken Thompson** in **2009, November**.
- Go is syntactically similar to C Language, but with memory safety, garbage collection,
- ural typing, and CSP-style concurrency additionally.
- The language is often referred to as **Golang** because of its domain name, **golang.org**, but the proper name is Go

Prerequisites

- Some programming experience. The code here is pretty simple, but it helps to know something about functions.
- A tool to edit your code. **VSCode** (free), **GoLand** (paid), and **Vim** (free).
- A command terminal. Go works well using any terminal on Linux and Mac, and on PowerShell or cmd in Windows.
- C:\Users\Tarkeshwar Barua>go version

go version go1.16.6 windows/amd64

Advantages

- static typing and run-time efficiency (like C),
- readability and usability (like Python or JavaScript)
- high-performance networking and multiprocessing.

Features of Go

- A syntax and environment adopting patterns more common in dynamic languages:
 - Optional concise variable declaration and initialization through type inference (`x := 0` instead of `int x = 0`; or `var x = 0`);.
 - Fast compilation.
 - Remote package management (`go get`) and online package documentation.
- Distinctive approaches to particular problems:
 - **Built-in concurrency primitives**: light-weight processes (`go routines`), channels, and the `select` statement.
 - An interface system in place of **virtual inheritance**, and type embedding instead of **non-virtual inheritance**.
 - A toolchain that, by default, produces statically linked native binaries without external dependencies.
- A desire to keep the language specification simple enough to hold in a programmer's head, in part by omitting features that are common in similar languages.

Applications Written in Go

- **Caddy**, an open source HTTP/2 web server with automatic HTTPS capability
- **CockroachDB**, an open source, survivable, strongly consistent, scale-out SQL database
- **Consul**, a software for DNS-based service discovery and providing distributed Key-value storage, segmentation and configuration.
- **Docker**, a set of tools for deploying Linux containers
- **EdgeX**, a vendor-neutral open-source platform hosted by the Linux Foundation, providing a common framework for industrial IoT edge computing
- **Hugo**, a static site generator
- **InfluxDB**, an open source database specifically to handle time series data with high availability and high performance requirements
- **InterPlanetary File System**, a content-addressable, peer-to-peer hypermedia protocol
- **Juju**, a service orchestration tool by Canonical, packagers of Ubuntu Linux
- **Kubernetes** container management system

Applications Written in Go

- **Ind**, an implementation of the Bitcoin Lightning Network
- **Mattermost**, a teamchat system
- **NATS Messaging**, an open-source messaging system featuring the core design principles of performance, scalability, and ease of use
- **OpenShift**, a cloud computing platform as a service by Red Hat
- **Rclone**, a command line program to manage files on cloud storage and other high latency services
- **Snappy**, a package manager for Ubuntu Touch developed by Canonical
- **Syncthing**, an open-source file synchronization client/server application
- **Terraform**, an open-source, multiple cloud infrastructure provisioning tool from HashiCorp
- **TiDB**, an open-source, distributed HTAP database compatible with the MySQL protocol from PingCAP

Applications Written in Go

- **Cacoo**, for their rendering of the user dashboard page and microservice using Go and gRPC
- **Chango**, a programmatic advertising company uses Go in its real-time bidding systems
- **Cloud Foundry**, a platform as a service
- **Cloudflare**, for their delta-coding proxy Railgun, their distributed DNS service, as well as tools for cryptography, logging, stream processing, and accessing SPDY sites
- **Container Linux (formerly CoreOS)**, a Linux-based operating system that uses Docker containers and rkt containers
- **Couchbase**, Query and Indexing services within the Couchbase Server
- **Dropbox**, who migrated some of their critical components from Python to Go
- **Ethereum**, The go-ethereum implementation of the Ethereum Virtual Machine blockchain for the Ether cryptocurrency

Applications Written in Go

- **Gitlab**, a web-based DevOps lifecycle tool that provides a Git-repository, wiki, issue-tracking, continuous integration, deployment pipeline features
- **Google**, for many projects, notably including download server dl.google.com
- **Heroku**, for Doozer, a lock service
- **Hyperledger Fabric**, an open source, enterprise-focused distributed ledger project
- **MongoDB**, tools for administering MongoDB instances
- **Netflix**, for two portions of their server architecture
- **Nutanix**, for a variety of micro-services in its Enterprise Cloud OS

Applications Written in Go

- **Plug.dj**, an interactive online social music streaming website[138]
- **SendGrid**, a Boulder, Colorado-based transactional email delivery and management service.
- **SoundCloud**, for "dozens of systems"
- **Splice**, for the entire backend (API and parsers) of their online music collaboration platform
- **ThoughtWorks**, some tools and applications for continuous delivery and instant messages (CoyIM)
- **Twitch**, for their IRC-based chat system (migrated from Python)
- **Uber**, for handling high volumes of geofence-based queries

Go Compiler

- The source code written in source file is the human readable source for your program.
- It needs to be compiled and turned into machine language so that your CPU can actually execute the program as per the instructions given.
- The Go programming language compiler compiles the source code into its final executable program.
- Go distribution comes as a binary install-able for FreeBSD (release 8 and above), Linux, Mac OS X (Snow Leopard and above), and Windows operating systems with 32-bit (386) and 64-bit (amd64) x86 processor architectures.

Features Excluded Intentionally

- Support for type inheritance
- Support for method or operator overloading
- Support for circular dependencies among packages
- Support for pointer arithmetic
- Support for assertions
- Support for generic programming

Features of Go Programming

- Support for environment adopting patterns similar to dynamic languages. For example,
- type inference (`x := 0` is valid declaration of a variable `x` of type `int`)
- Compilation time is fast.
- Inbuilt concurrency support: lightweight processes (via go routines), channels, select statement.
- Go programs are simple, concise, and safe.
- Support for Interfaces and Type embedding.
- Production of statically linked native binaries without external dependencies.

Semicolons

- The only place you typically see semicolons is separating the clauses of for loops and the like; they are not necessary after every statement.
- are inserted automatically at the end of every line that looks like the end of a statement. You don't need to type them yourself.
- you can also leave out a semicolon immediately before a closing brace.
- put the opening brace of a construct such as an if statement on the same line as the if; if you don't, there are situations that may not compile or may give the wrong result

Go Installation

- `sudo snap install go --classic`
- Go 1.16.7 from Michael Hudson-Doyle (mwhudson) installed
- `sudo apt install golang-go`
- `sudo apt install gccgo-go`
- `vim ~/.bashrc`
- `# set up Go lang path #`
- `export GOPATH=$HOME/go`
- `export PATH=$PATH:/usr/local/go/bin:$GOPATH/bin`
- `go version`

Text Editor

- Visual Studio Code, Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
- The files you create with the text editor are called source files. They contain program source code.
- The source files for Go programs are typically named with the extension **".go"**.

Hello World

- `package main`
- `// Import OS and fmt packages`
- `import ("fmt" "os")`
- `// Let us start`
- `func main() {`
- `fmt.Println("Hello, world!") // Print simple text on screen`
- `fmt.Println(os.Getenv("USER"), ", Let's be friends!") // Read Linux $USER environment variable`
- `}`

Comments

- Comments are like helping texts in your Go program and they are ignored by the compiler.
- They start with `/*` and terminate with the characters `*/` as shown below:

```
/* my first program in Go */
```

- You cannot have comments within comments and they do not occur within a string or character literals

Identifiers

- A Go identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).

`identifier = letter { letter | unicode_digit } .`

- Go does not allow punctuation characters such as @, \$, and % within identifiers. Go is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in Go. Here are some examples of acceptable identifiers:

`mahesh kumar abc move_name a_123 myname50 _temp j a23b9
retVal`

Project Design

- `mkdir -p -v go/src/my-project`
- `vim go/src/my-project/test.go`
- `go build go/src/my-project/test.go`
- `./test`

```
package main
```

```
import (
```

```
"fmt"
```

```
)
```

```
func main() {
```

```
// we use var that declares one or more variables with or without type
```

```
// type is a keyword
```

```
//var name type
```

```
//var name = value
```

```
//shorthand syntax
```

```
name := 20 // dynamic type of variable according to the value stored
```

```
fmt.Println(name)
```

```
// define i
```

```
var i int // statically mentioning data type
```

```
// set value for i
```

```
i = 10
```

```
// we can also set value as follows
```

```
var y = 5 // style coming from JavaScript
```

Project Design

```
var msg = "Remote host found."
```

```
var foo, bar int = 100, 200
```

```
fmt.Println(bar)
```

```
// shorthand syntax
```

```
vehicle := "Mercedes"
```

```
age := 52
```

```
// Bool true or false
```

```
var is_job_failed = false
```

```
// print it
```

```
fmt.Printf("%d %d %s\n", i, y, msg)
```

```
fmt.Println(foo)
```

```
fmt.Println(age)
```

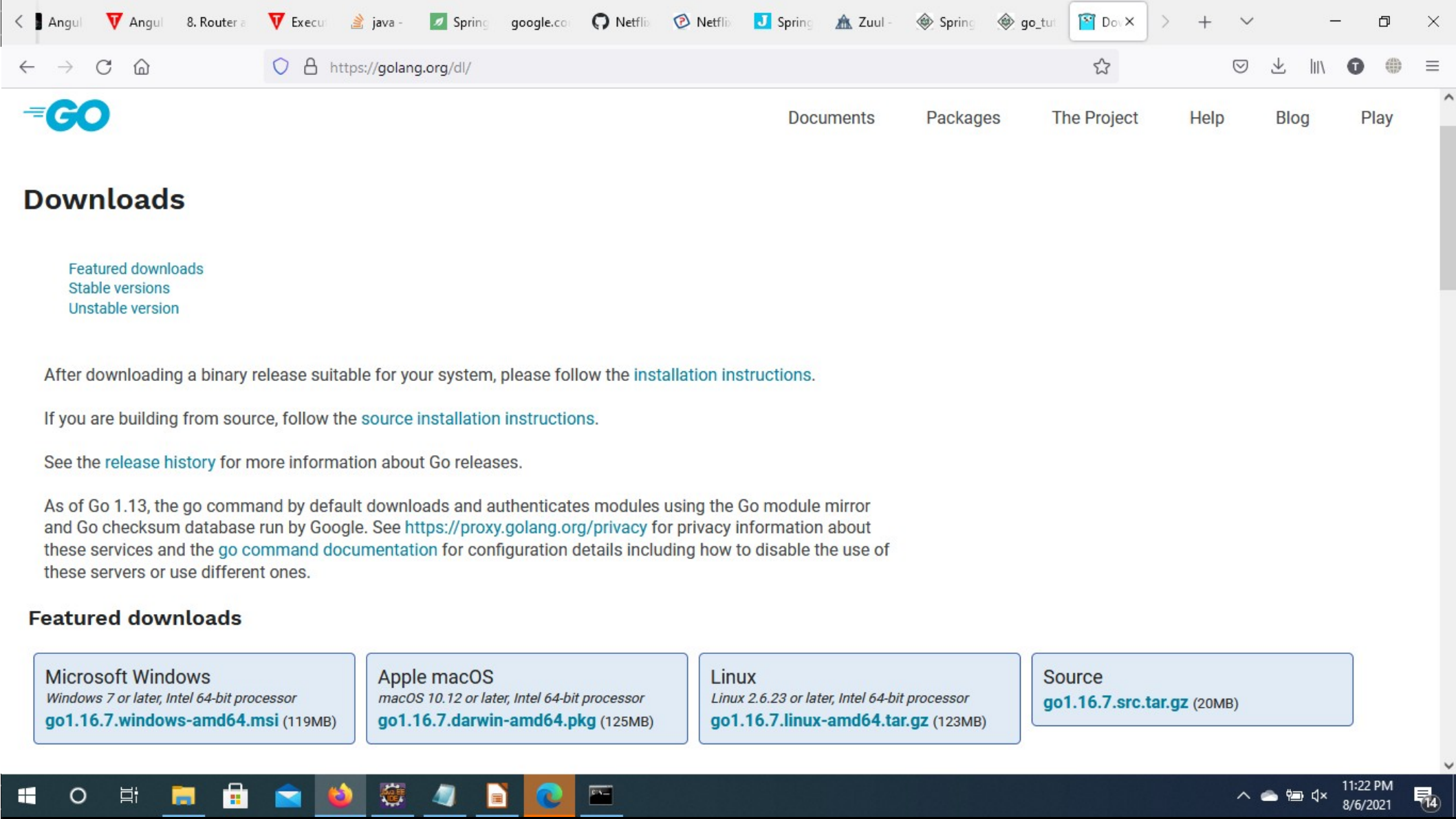
```
fmt.Println(vehicle)
```

```
fmt.Println(is_job_failed)
```

```
}
```

Keywords

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var



Downloads

- Featured downloads
- Stable versions
- Unstable version

After downloading a binary release suitable for your system, please follow the [installation instructions](#).

If you are building from source, follow the [source installation instructions](#).

See the [release history](#) for more information about Go releases.

As of Go 1.13, the go command by default downloads and authenticates modules using the Go module mirror and Go checksum database run by Google. See <https://proxy.golang.org/privacy> for privacy information about these services and the [go command documentation](#) for configuration details including how to disable the use of these servers or use different ones.

Featured downloads

Microsoft Windows

Windows 7 or later, Intel 64-bit processor

[go1.16.7.windows-amd64.msi](#) (119MB)

Apple macOS

macOS 10.12 or later, Intel 64-bit processor

[go1.16.7.darwin-amd64.pkg](#) (125MB)

Linux

Linux 2.6.23 or later, Intel 64-bit processor

[go1.16.7.linux-amd64.tar.gz](#) (123MB)

Source

[go1.16.7.src.tar.gz](#) (20MB)

Go Installation

- C:\Users\Tarkeshwar Barua>set path="C:\Program Files\Go\bin"
- C:\Users\Tarkeshwar Barua>go version

go version go1.16.6 windows/amd64

Linux export PATH=\$PATH:/usr/local/go/bin

Mac export PATH=\$PATH:/usr/local/go/bin

FreeBSD export PATH=\$PATH:/usr/local/go/bin

Your First Go program - Go Hello World!

- Create a folder called studyGo. In this Go language tutorial, we will create our go programs inside this folder. Go files are created with the extension .go. You can run Go programs using the syntax

```
go run <filename>
```

- Create a file called first.go and add the below code into it and save

```
package main
```

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

```
func main() {
```

```
    fmt.Println("Hello World! This is my first Go program\n")
```

```
}
```

```
C:\Users\Tarkeshwar Barua>go run first.go
```

Data Types

- Types(data types) represent the type of the value stored in a variable, type of the value a function returns, etc.
- There are three basic types in Go Language
 - Numeric
 - String
 - Boolean

Numeric types

- Represent numeric values which includes integer, floating point, and complex values. Various numeric types are:
- int8 - 8 bit signed integers.
- int16 - 16 bit signed integers.
- int32 - 32 bit signed integers.
- int64 - 64 bit signed integers.

Numeric types

- uint8 - 8 bit unsigned integers.
- uint16 - 16 bit unsigned integers.
- uint32 - 32 bit unsigned integers.
- uint64 - 64 bit unsigned integers.
- float32 - 32 bit floating point numbers.
- float64 - 64 bit floating point numbers.
- complex64 – has float32 real and imaginary parts.
- complex128 - has float32 real and imaginary parts

Creating Constant

- `// syntax`
- `const var type = value`
- `// define constant pi with as float32 with 3.14159 value`
- `const pi float32 = 3.14159`
- `// create constant string variable`
- `const error_msg string = "Docker is not installed"`
- `fmt.Println(pi)`
- `fmt.Println(error_msg)`

String types

- Represents a sequence of bytes(characters). You can do various operations on strings like string concatenation, extracting substring, etc
- **Boolean types** - Represents 2 values, either true or false.

Variable

- Variables point to a memory location which stores some kind of value. The type parameter(in the below syntax) represents the type of value that can be stored in the memory location. Variable can be declared using the syntax

```
var <variable_name> <type>
```

```
var x int
```

- Once You declare a variable of a type You can assign the variable to any value of that type. You can also give an initial value to a variable during the declaration itself using

```
var <variable_name> <type> = <value>
```

```
var x int = 10
```

Variable

- A variable is nothing but a name given to a storage area that the programs can manipulate. Each variable in Go has a specific type, which determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable.
- The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Go is case-sensitive.

Variable

<u>Type</u>	<u>Description</u>
Byte	Typically a single octet(one byte). This is an byte type.
int	The most natural size of integer for the machine
float32	A single-precision floating point value.

Compiler

- Gccgo is a Go compiler that uses the GCC back end
- 6g for the 64-bit x86, 8g for the 32-bit x86, and more
- These compilers run significantly faster but generate less efficient code than gccgo

Variable

- Variables can be initialized (assigned an initial value) in their declaration. The type of variable is automatically judged by the compiler based on the value passed to it. The initializer consists of an equal sign followed by a constant expression as follows:
- `variable_name = value;`
- `d = 3, f = 5; // declaration of d and f. Here d and f are int`
- For definition without an initializer: variables with static storage duration are implicitly initialized with nil (all bytes have the value 0); the initial value of all other variables is zero value of their data type.

Sample Program

```
import "math"
```

```
type Shape interface {  
    Area() float64  
}
```

```
type Square struct { // Note: no "implements" declaration
```

```
    side float64
```

```
}
```

```
func (sq Square) Area() float64 { return sq.side * sq.side }
```

```
type Circle struct { // No "implements" declaration here either
```

```
    radius float64
```

```
}
```

```
func (c Circle) Area() float64 { return math.Pi * math.Pow(c.radius, 2) }
```

```
package main
```

Hello World

```
import "fmt" // formatted I/O, similar to C's C  
file input/output
```

```
func main() {
```

```
    fmt.Println("Hello, world!")
```

```
}
```

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var x int
```

```
    x=3 //assigning x the value 3
```

```
    fmt.Println("x:", x) //prints 3
```

```
    var y int=20
```

```
    fmt.Println("y:", y)
```

```
    var z=50
```

```
    fmt.Println("z:", z)
```

```
    var i, j = 100,"hello"
```

```
    fmt.Println("i and j:", i,j)
```

```
}
```

Hello World

```
package main
```

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

```
func main() {
```

```
    a := 20
```

```
    fmt.Println(a)
```

```
    //gives error since a is already declared
```

```
    a := 30
```

```
    fmt.Println(a)
```

```
}
```

Hello World

The lvalues and the rvalues in Go

- **lvalue:** Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue:** The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

The lvalues and the rvalues in Go

- Variables are **lvalues** and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side.

`x = 20.0`

- The following statement is not valid. It would generate compile-time error:

`10 = 20`

Literals

- Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.
- Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.
- Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

- An integer literal can be a decimal, octal, or hexadecimal constant.
- A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.
- An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively.
- The suffix can be uppercase or lowercase and can be in any order.

Integer Literals

- 212 /* Legal */
- 215u /* Legal */
- 0xFeeL /* Legal */
- 078 /* Illegal: 8 is not an octal digit */
- 032UU /* Illegal: cannot repeat a suffix */

Following are other examples of various type of Integer literals:

- 85 /* decimal */
- 0213 /* octal */
- 0x4b /* hexadecimal */
- 30 /* int */
- 30u /* unsigned int */
- 30l /* long */
- 30ul /* unsigned long */

Floating-point Literals

- A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part.
- You can represent floating point literals either in decimal form or exponential form.
- While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both.
- The signed exponent is introduced by **e** or **E**.

Floating-point Literals

3.14159 /* Legal */

314159E-5L /* Legal */

510E /* Illegal: incomplete exponent */

210f /* Illegal: no decimal or exponent */

.e55 /* Illegal: missing integer or fraction */

Example Literals

```
s := "hello"
```

```
if s[1] != 'e' { os.Exit(1) }
```

```
s = "good bye"
```

```
var p *string = &s
```

```
*p = "ciao"
```

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var a, b, c = 3, 4, "foo"
```

```
    fmt.Println(a)
```

```
    fmt.Println(b)
```

```
    fmt.Println(c)
```

```
    fmt.Printf("a is of type %T\n", a)
```

```
    fmt.Printf("b is of type %T\n", b)
```

```
    fmt.Printf("c is of type %T\n", c)
```

```
}
```

Mixed Variable Declaration in Go

Dynamic Type Declaration / Type Inference in Go

- A dynamic type variable declaration requires the compiler to interpret the type of the variable based on the value passed to it. The compiler does not require a variable to have type statically as a necessary requirement.
- Example, where the variables have been declared without any type. Notice, in case of type inference, we initialized the variable y with `:=` operator, whereas x is initialized using `=` operator.

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var x float64 = 20.0
```

```
    y := 42
```

```
    fmt.Println(x)
```

```
    fmt.Println(y)
```

```
    fmt.Printf("x is of type %T\n", x)
```

```
    fmt.Printf("y is of type %T\n", y)
```

```
}
```

Dynamic Type Declaration / Type Inference in Go

Static Type Declaration in Go

- A static type variable declaration provides assurance to the compiler that there is one variable available with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail of the variable.
- A variable declaration has its meaning at the time of compilation only, the compiler needs the actual variable declaration at the time of linking of the program.
- Example, where the variable has been declared with a type and initialized inside the main function:

Static Type Declaration in Go

```
package main
import "fmt"
func main() {
    var x float64
    x = 20.0
    fmt.Println(x)
    fmt.Printf("x is of type %T\n", x)
    fmt.Printf("x is of value %V\n", x)
}
```

Escape Sequence

- When certain characters are preceded by a backslash, they will have a special meaning in Go.
- These are known as Escape Sequence codes which are used to represent newline (`\n`), tab (`\t`), backspace, etc.
- You have a list of some of such escape sequence code

<u>Escape sequence</u>	<u>Meaning</u>
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or Bell
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab
\ooo	Octal Number of one or more digits
\xhh...	Hexadecimal Number of one or more digits

Escape Sequece Character

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Printf("Hello\tWorld!")  
}
```

String Literals in Go

- String literals or constants are enclosed in double quotes `""`.
- A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.
- You can break a long line into multiple lines using string literals and separating them using whitespaces.

Constants

- Constant variables are those variables whose value cannot be changed once assigned.
- A constant in Go programming language is declared by using the keyword "**const**"
- You can use **const** prefix to declare constants with a specific type as follows:

```
const variable type = value;
```

Constants

```
package main
```

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

```
func main() {
```

```
    const b =10
```

```
    fmt.Println(b)
```

```
    b = 30
```

```
    fmt.Println(b)
```

```
}
```

Constants

```
package main  
import "fmt"  
func main() {  
    const LENGTH int = 10  
    const WIDTH int = 5  
    var area int  
    area = LENGTH * WIDTH  
    fmt.Printf("value of area : %d", area)  
}
```

Operators Precedence in Go

- Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example,
- the multiplication operator has higher precedence than the addition operator.
- For example $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.
- Here, operators with the highest precedence appear at the top of the table, and those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

```
package main

import (
    "os"
    "flag" // command line option
    parser
)







var omitNewline = flag.Bool("n",
false, "don't print final newline")

const (
    Space = " "
    Newline = "\n"
)
```

Coding

```
func main() {
    flag.Parse() // Scans the arg list and sets up flags
    var s string = ""
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 {
            s += Space
        }
        s += flag.Arg(i)
    }
    if !*omitNewline {
        s += Newline
    }
    os.Stdout.WriteString(s)
}
```

Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical
- manipulations. Go language is rich in built-in operators and provides the following types of operators:
 -  Arithmetic Operators
 -  Relational Operators
 -  Logical Operators
 -  Bitwise Operators
 -  Assignment Operators
 -  Miscellaneous Operators

Bitwise Operators

<u>X</u>	<u>Y</u>	<u>X&Y</u>	<u>X Y</u>	<u>X^Y</u>
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Miscellaneous Operators

<u>Description</u>	<u>Operator</u>	<u>Example</u>
Returns the address of a variable.	&	&a ; provides actual address of the variable.
Pointer to a variable.	*	*a ; provides pointer to a variable.


```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var a int = 4
```

```
    var b int32
```

```
    var c float32
```

```
    var ptr *int
```

```
    /* example of type operator */
```

```
    fmt.Printf("Line 1 - Type of variable a = %T\n", a );
```

```
    fmt.Printf("Line 2 - Type of variable b = %T\n", b );
```

```
    fmt.Printf("Line 3 - Type of variable c= %T\n", c );
```

```
    /* example of & and * operators */
```

```
    ptr = &a /* 'ptr' now contains the address of 'a'*/
```

```
    fmt.Printf("value of a is  %d\n", a);
```

```
    fmt.Printf("*ptr is %d.\n", *ptr);
```

Miscellaneous Operators

Operators Precedence in Go

<u>Category</u>	<u>Operator</u>	<u>Associativity</u>
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right

Operators Precedence in Go

Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= =	Right to left
Comma	,	Left to right

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var a int = 20
```

```
    var b int = 10
```

```
    var c int = 15
```

```
    var d int = 5
```

```
    var e int;
```

```
    e = (a + b) * c / d;    // ( 30 * 15 ) / 5
```

```
    fmt.Printf("Value of (a + b) * c / d is : %d\n", e );
```

```
    e = ((a + b) * c) / d;    // (30 * 15) / 5
```

```
    fmt.Printf("Value of ((a + b) * c) / d is : %d\n", e );
```

```
    e = (a + b) * (c / d);    // (30) * (15/5)
```

```
    fmt.Printf("Value of (a + b) * (c / d) is : %d\n", e );
```

```
    e = a + (b * c) / d;    // 20 + (150/5)
```

```
    fmt.Printf("Value of a + (b * c) / d is : %d\n", e );
```

Operators Precedence in Go

If else

```
package main  
import "fmt"  
func main() {  
    var x = 50  
    if x < 10 {  
        //Executes if x < 10  
        fmt.Println("x is less than 10")  
    }  
}
```

- Here since the value of x is greater than 10, the statement inside if block condition will not executed.

If else

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    for i := 1; i <= 10; i++ {
```

```
        fmt.Printf("Welcome %d times.\n",i)
```

```
        // break out of for loop when i is 5
```

```
        if i == 5 {
```

```
            break
```

```
        }
```

```
    }
```

```
}
```

If else

```
package main
import "fmt"
func main() {
    var n int
    fmt.Print("Enter your number: ")
    fmt.Scanf("%d", &n)
    fmt.Println(n)
    if n >= 0 {
        fmt.Println("Number is positive")
    } else {
        fmt.Println("Number is negative")
    }
}
```

If else

```
package main
import "fmt"
func main() {
    var x = 50
    if x < 10 {
        //Executes if x is less than 10
        fmt.Println("x is less than 10")
    } else {
        //Executes if x >= 10
        fmt.Println("x is greater than or equals 10")
    }
}
```


If else

```
package main
import "fmt"
func main() {
    var x = 100
    if x < 10 {
        //Executes if x is less than 10
        fmt.Println("x is less than 10")
    } else if x >= 10 && x <= 90 {
        //Executes if x >= 10 and x<=90
        fmt.Println("x is between 10 and 90")
    } else {
        //Executes if both above cases fail i.e x>90
        fmt.Println("x is greater than 90")
    }
}
```

If else

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var x string = "hello"
```

```
    var y string = "hello"
```

```
    fmt.Println(x == y)
```

```
}
```

Addition in Print Statement

```
$ go help
```

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    fmt.Println("1 + 1 =", 1.0 + 1.0)
```

```
}
```

Concatenation

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    fmt.Println(len("Hello World"))
```

```
    fmt.Println("Hello World"[1])
```

```
    fmt.Println("Hello " + "World")
```

```
}
```

Boolean Demostration

```
package main  
import "fmt"  
func main() {  
    fmt.Println(true && true)  
    fmt.Println(true && false)  
    fmt.Println(true || true)  
    fmt.Println(true || false)  
    fmt.Println(!true)  
}
```

\$ go env

```
set GOM11MODULE=  
set GOARCH=amd64  
set GOBIN=  
set GOCACHE=C:\Users\Tarkeshwar Barua\AppData\Local\go-build  
set GOENV=C:\Users\Tarkeshwar Barua\AppData\Roaming\go\env  
set GOEXE=.exe  
set GOFLAGS=  
set GOHOSTARCH=amd64  
set GOHOSTOS=windows  
set GOINSECURE=  
set GOMODCACHE=C:\Users\Tarkeshwar Barua\go\pkg\mod  
set GONOPROXY=  
set GONOSUMDB=  
set GOOS=windows  
set GOPATH=C:\Users\Tarkeshwar Barua\go  
set GOPRIVATE=
```

\$ go env

```
set GOPROXY=https://proxy.golang.org,direct
set GOROOT=C:\Program Files\Go
set GOSUMDB=sum.golang.org
set GOTMPDIR=
set GOTOOLDIR=C:\Program Files\Go\pkg\tool\windows_amd64
set GOVCS=
set GOVERSION=go1.16.6
set GCCGO=gccgo
set AR=ar
set CC=gcc
set CXX=g++
set CGO_ENABLED=1
set GOMOD=NUL
set CGO_CFLAGS=-g -O2
set CGO_CPPFLAGS=
set CGO_CXXFLAGS=-g -O2
set CGO_FFLAGS=-g -O2
set CGO_LDFLAGS=-g -O2
set PKG_CONFIG=pkg-config
set GOGCCFLAGS=-m64 -mthreads -fmessage-length=0 -fdebug-prefix-map=C:\Users\TARKES~1\AppData\Local\Temp\go-build3435550131=/tmp/go-build
set gno-record-gcc-switches
```

Create a Go module

- you'll create two modules. The first is a library which is intended to be imported by other libraries or applications. The second is a caller application which will use the first.
- Start your module using the **go mod init** command.
- `go mod init example.com/greetings`

Create a Go module

- The **go mod init** command creates a **go.mod** file to track your code's dependencies.
- The file includes only the **name of your module** and the **Go version** your code supports.
- The **go.mod** file will list the versions your code depends on. This keeps builds reproducible and gives you direct control over which module versions to use.
- In your text editor, create a file in which to write your code and call it **greetings.go**.
- Paste the following code into your **greetings.go** file and save the file.

greetings.go

```
package greetings
```

```
import "fmt"
```

```
// Hello returns a greeting for the named person.
```

```
func Hello(name string) string {
```

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

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

```
    return message
```

```
}
```

Hello.go

```
package main
```

```
import (
```

```
    "fmt"
```

```
    "example.com/greetings"
```

```
)
```

```
func main() {
```

```
    // Get a greeting message and print it.
```

```
    message := greetings.Hello("Tarkeshwar Barua")
```

```
    fmt.Println(message)
```

```
}
```

Scope

```
package main  
import "fmt"  
var x string = "Hello World"  
func main() {  
    fmt.Println(x)  
    f()  
}  
func f() {  
    fmt.Println(x)  
}
```

Notice that we moved the variable outside of the main function. This means that other functions can access this variable:

Defining Multiple Variables

```
var (
```

```
    a = 5
```

```
    b = 10
```

```
    c = 15
```

```
)
```

Use the keyword **var** (or **const**) followed by parentheses with each variable on its own line.

Switch

- Switch is another conditional statement. Switch statements evaluate an expression and the result is compared against a set of available values(cases).
- Once a match is found the statements associated with that match(case) is executed. If no match is found nothing will be executed.
- You can also add a default case to switch which will be executed if no other matches are found.

Switch

```
switch expression {  
    case value_1:  
        statements_1  
    case value_2:  
        statements_2  
    case value_n:  
        statements_n  
    default:  
        statements_default  
}
```

- Here the value of the expression is compared against the values in each case. Once a match is found the statements associated with that case is executed. If no match is found the statements under the default section is executed.

Control Structures

First let's write a program that counts to 10, starting from 1, with each number on its own line. Using what we've learned so far we could write this:

```
package main
import "fmt"
func main() {
    fmt.Println(1)
    fmt.Println(2)
    fmt.Println(3)
    fmt.Println(4)
    fmt.Println(5)
    fmt.Println(6)
    fmt.Println(7)
    fmt.Println(8)
    fmt.Println(9)
    fmt.Println(10)
}
```


Switch

A switch statement starts with the keyword **switch** followed by an expression and then a series of cases.

The value of the expression is compared to the expression following each **case** keyword.

If they are equivalent then the statement(s) following the **:** is executed.

If statement each case is checked top down and the first one to succeed is chosen.

A switch also supports a default case which will happen if none of the cases matches the value.

Switch

Suppose we wanted to write a program that printed the English names for numbers. Using what we've learned so far we might start by doing this:

```
if i == 0 {  
    fmt.Println("Zero")  
} else if i == 1 {  
    fmt.Println("One")  
} else if i == 2 {  
    fmt.Println("Two")  
} else if i == 3 {  
    fmt.Println("Three")  
} else if i == 4 {  
    fmt.Println("Four")  
} else if i == 5 {  
    fmt.Println("Five")  
}
```

Switch

```
package main
import "fmt"
func main() {
    var i int
    fmt.Println("Please Enter a Number : ")
    fmt.Scanf("%d", &i)
    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")
    }
}
```

Switch

```
package main
import "fmt"
func main() {
    a,b := 2,1
    switch a+b {
    case 1:
        fmt.Println("Sum is 1")
    case 2:
        fmt.Println("Sum is 2")
    case 3:
        fmt.Println("Sum is 3")
    default:
        fmt.Println("Printing default")
    }
}
```

Arrays

An array is a numbered sequence of elements of a single type with a fixed length.

```
var x [5]int
```

x is an example of an array which is composed of 5 ints.

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var x [5]int
```

```
    x[4] = 100
```

```
    fmt.Println(x)
```

```
}
```

Arrays

`x[4] = 100` should be read "set the 5th element of the array `x` to 100". It might seem strange that `x[4]` represents the 5th element instead of the 4th but like strings, arrays are indexed starting from 0. Arrays are accessed in a similar way. We could change `fmt.Println(x)` to `fmt.Println(x[4])` and we would get 100.

```
func main() {  
    var x [5]float64  
    x[0] = 98  
    x[1] = 93  
    x[2] = 77  
    x[3] = 82  
    x[4] = 83  
    var total float64 = 0  
    for i := 0; i < 5; i++ {  
        total += x[i]  
    }  
    fmt.Println(total / 5)  
}
```

Arrays

```
package main
import "fmt"
func main() {
    // define array named domains as string type
    var domains [2]string
    fmt.Println("current values for array:", domains)
    // add value and print it
    domains[0] = "cyberciti.biz"
    fmt.Println("Set value : ", domains)
    fmt.Println("Get value for 0 element : ", domains[0])
    // get array length
    fmt.Println("Array length : ", len(domains))
    // add one more value
    domains[1] = "nixcraft.com"
    // use for loop to print our array
    for i := 0; i < len(domains); i++ {
        fmt.Println("Get value for element ", i, " is ", domains[i])
    }
}
```

Maps

- A map is an unordered collection of key-value pairs. Also known as an associative array, a hash table or a dictionary, maps are used to look up a value by its associated key.

```
var x map[string]int
```

- The map type is represented by the keyword map, followed by the key type in brackets and finally the value type. If you were to read this out loud you would say “x is a map of strings to ints.” Like arrays and slices maps can be accessed using brackets.

```
var x map[string]int
```

```
x["key"] = 10
```

```
fmt.Println(x)
```


Maps

```
package main
import "fmt"
func main() {
    elements := make(map[string]string)
    elements["H"] = "Hydrogen"
    elements["He"] = "Helium"
    elements["Li"] = "Lithium"
    elements["Be"] = "Beryllium"
    elements["B"] = "Boron"
    elements["C"] = "Carbon"
    elements["N"] = "Nitrogen"
    elements["O"] = "Oxygen"
    elements["F"] = "Fluorine"
    elements["Ne"] = "Neon"
    fmt.Println(elements["Li"])
}
```

Maps

The statement `x["key"] = 10` is similar to what we saw with arrays but the key, instead of being an integer, is a string because the map's key type is string. We can also create maps with a key type of int:

```
x := make(map[int]int)
```

```
x[1] = 10
```

```
fmt.Println(x[1])
```

First the length of a map (found by doing `len(x)`) can change as we add new items to it. When first created it has a length of 0, after `x[1] = 10` it has a length of 1. Second maps are not sequential. We have `x[1]`, and with an array that would imply there must be an `x[0]`, but maps don't have this requirement.

We can also delete items from a map using the built-in delete function:

```
delete(x, 1)
```

Maps

```
fmt.Println(elements["Un"])
```

If you run this you should see nothing returned. Technically a map returns the zero value for the value type (which for strings is the empty string). Although we could check for the zero value in a condition (**elements["Un"] == ""**)

```
name, ok := elements["Un"]
```

```
fmt.Println(name, ok)
```

Accessing an element of a map can return two values instead of just one. The first value is the result of the lookup, the second tells us whether or not the lookup was successful.

```
if name, ok := elements["Un"]; ok {
```

```
    fmt.Println(name, ok)
```

```
}
```

For Loop

- Loops are used to execute a block of statements repeatedly based on a condition. Most of the programming languages provide 3 types of loops - for, while, do while. But Go programming language supports only for loop. The syntax of a Golang for loop is

```
for initialisation_expression; evaluation_expression; iteration_expression{  
    // one or more statement  
}
```

- The initialisation_expression is executed first (and only once) in Golang for loop. Then the evaluation_expression is evaluated and if it's true the code inside the block is executed. The iteration_expression is executed, and the evaluation_expression is evaluated again. If it's true the statement block gets executed again. This will continue until the evaluation_expression becomes false.

For Loop

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var i int
```

```
    for i = 1; i <= 5; i++ {
```

```
        fmt.Println(i)
```

```
    }
```

```
}
```

For Loop

```
package main  
import "fmt"  
func main() {  
    for i := 1; i <= 10; i++ {  
        if i % 2 == 0 {  
            fmt.Println(i, "even")  
        } else {  
            fmt.Println(i, "odd")  
        }  
    }  
}
```

For Loop

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    i := 1
```

```
    for i <= 10 {
```

```
        fmt.Println(i)
```

```
        i = i + 1
```

```
    }
```

```
}
```

For Loop

First we create a variable called `i` that we use to store the number we want to print. Then we create a for loop by using the keyword `for`, providing a conditional expression which is either true or false and finally supplying a block to execute.

We evaluate (run) the expression `i <= 10` (“`i` less than or equal to 10”). If this evaluates to true then we run the statements inside of the block. Otherwise we jump to the next line of our program after the block.

After we run the statements inside of the block we loop back to the beginning of the for statement and repeat step 1.

For Loop

```
package main
import "fmt"
func main() {
    // single condition for loop
    m := 1
    for m <= 5 {
        fmt.Printf("Welcome %d times.\n",m)
        m = m + 1
    }
    // classic for loop example
    for i := 6; i <= 10; i++ {
        fmt.Printf("Welcome %d times.\n",i)
    }
}
```

Arrays

- Array represents a fixed size, named sequence of elements of the same type. You cannot have an array which contains both integer and characters in it. You cannot change the size of an array once You define the size. The syntax for declaring an array is

`var arrayname [size] type`

- Each array element can be assigned value using the syntax

`arrayname [index] = value`

- Array index starts from 0 to size-1.

Arrays

- You can assign values to array elements during declaration using the syntax

```
arrayname := [size] type {value_0,value_1,...,value_size-1}
```

- You can also ignore the size parameter while declaring the array with values by replacing size with ... and the compiler will find the length from the number of values. Syntax is

```
arrayname := [...] type {value_0,value_1,...,value_size-1}
```

- You can find the length of the array by using the syntax

```
len(arrayname)
```

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var numbers [3] string //Declaring a string array of size 3 and adding elements
```

```
    numbers[0] = "One"
```

```
    numbers[1] = "Two"
```

```
    numbers[2] = "Three"
```

```
    fmt.Println(numbers[1]) //prints Two
```

```
    fmt.Println(len(numbers)) //prints 3
```

```
    fmt.Println(numbers) // prints [One Two Three]
```

```
    directions := [...] int {1,2,3,4,5} // creating an integer array and the size of the array is  
defined by the number of elements
```

```
    fmt.Println(directions) //prints [1 2 3 4 5]
```

```
    fmt.Println(len(directions)) //prints 5
```

```
    //Executing the below commented statement prints invalid array index 5 (out of bounds for  
5-element array)
```

```
    //fmt.Println(directions[5])
```

```
}
```

Arrays

Functions

- A function represents a block of statements which performs a specific task. A function declaration tells us function name, return type and input parameters. Function definition represents the code contained in the function. The syntax for declaring the function is

```
func function_name(parameter_1 type, parameter_n type)
return_type {
//statements
}
```

- The parameters and return types are optional. Also, you can return multiple values from a function.

Problem Statement

```
func main() {  
    xs := []float64{98,93,77,82,83}  
    total := 0.0  
    for _, v := range xs {  
        total += v  
    }  
    fmt.Println(total / float64(len(xs)))  
}
```

- This program computes the average of a series of numbers. Finding the average like this is a very general problem, so it's an ideal candidate for definition as a function. The average function will need to take in a slice of float64s and return one float64.

```
func average(xs []float64) float64 {  
    panic("Not Implemented")  
}
```

Problem Statement

```
func average(xs []float64) float64 {  
    total := 0.0  
    for _, v := range xs {  
        total += v  
    }  
    return total / float64(len(xs))  
}
```

- Notice that we changed the `fmt.Println` to be a `return` instead. The `return` statement causes the function to immediately stop and return the value after it to the function that called this one. Modify `main` to look like this:

```
func main() {  
    xs := []float64{98,93,77,82,83}  
    fmt.Println(average(xs))  
}
```

Stack of Functions

Functions are built up in a “stack”. Suppose we had this program:

```
func main() {  
    fmt.Println(f1())  
}  
func f1() int {  
    return f2()  
}  
func f2() int {  
    return 1  
}
```


Returning Multiple Values

Go is also capable of returning multiple values from a function:

```
func f() (int, int) {  
    return 5, 6  
}  
  
func main() {  
    x, y := f()  
}
```

Three changes are necessary: change the return type to contain multiple types separated by `,`, change the expression after the return so that it contains multiple expressions separated by `,`, and finally change the assignment statement so that multiple values are on the left side of the `:=` or `=`.

Multiple values are often used to return an error value along with the result (`x, err := f()`), or a boolean to indicate success (`x, ok := f()`).

Variadic Functions

There is a special form available for the last parameter in a Go function:

```
func add(args ...int) int {  
    total := 0  
    for _, v := range args {  
        total += v  
    }  
    return total  
}  
  
func main() {  
    fmt.Println(add(1,2,3))  
}
```

By using ... before the type name of the last parameter you can indicate that it takes zero or more of those parameters. In this case we take zero or more ints. We invoke the function like any other function except we can pass as many ints as we want. This is precisely how the **fmt.Println** function is implemented:

```
func Println(a ...interface{}) (n int, err error)
```

Variadic Functions

The `Println` function takes any number of values of any type. We can also pass a slice of ints by following the slice with `...`:

```
func main() {  
    xs := []int{1,2,3}  
    fmt.Println(add(xs...))  
}
```

```
package main
```

```
import "fmt"
```

```
// define function named total that accept int values and returns int
```

```
func total(x int, y int) int {
```

```
    return x + y
```

```
}
```

```
func main() {
```

```
// call our function and store result in the answer variable
```

```
    answer := total(10, 20)
```

```
    fmt.Println("10 + 20 = ", answer)
```

```
    fmt.Println("100 + 500 = ", total(100,500))
```

```
}
```

Functions

```
package main
```

```
import "fmt"
```

```
//calc is the function name which accepts two integers num1 and num2
```

```
//(int, int) says that the function returns two values, both of integer type.
```

```
func calc(num1 int, num2 int)(int, int) {
```

```
    sum := num1 + num2
```

```
    diff := num1 - num2
```

```
    return sum, diff
```

```
}
```

```
func main() {
```

```
    x,y := 15,10
```

```
//calls the function calc with x and y and gets sum, diff as output
```

```
    sum, diff := calc(x,y)
```

```
    fmt.Println("Sum",sum)
```

```
    fmt.Println("Diff",diff)
```

```
}
```

Functions

Closure

It is possible to create functions inside of functions:

```
func main() {  
    add := func(x, y int) int {  
        return x + y  
    }  
    fmt.Println(add(1,1))  
}
```

add is a local variable that has the type **func(int, int) int**. When you create a local function like this it also has access to other local variables.

Closure

```
func main() {  
    x := 0  
    increment := func() int {  
        x++  
        return x  
    }  
    fmt.Println(increment())  
    fmt.Println(increment())  
}
```

increment adds 1 to the variable x which is defined in the main function's scope. This x variable can be accessed and modified by the increment function. This is why the first time we call increment we see 1 displayed, but the second time we call it we see 2 displayed.

A function like this together with the non-local variables it references is known as a closure. In this case increment and the variable x form the closure.

Closure

One way to use closure is by writing a function which returns another function which – when called – can generate a sequence of numbers. For example here's how we might generate all the even numbers:

```
func makeEvenGenerator() func() uint {  
    i := uint(0)  
    return func() (ret uint) {  
        ret = i  
        i += 2  
        return  
    }  
}  
  
func main() {  
    nextEven := makeEvenGenerator()  
    fmt.Println(nextEven()) // 0  
    fmt.Println(nextEven()) // 2  
    fmt.Println(nextEven()) // 4  
}
```


Recursion

Finally a function is able to call itself. Here is one way to compute the factorial of a number:

```
func factorial(x uint) uint {  
    if x == 0 {  
        return 1  
    }  
    return x * factorial(x-1)  
}
```

factorial calls itself, which is what makes this function recursive. In order to better understand how this function works, lets walk through factorial(2):

Slices

- A slice is a segment of an array. Like arrays slices are **indexable** and have a length. Unlike arrays this length is allowed to change.

```
var x []float64
```

- The only difference between this and an array is the missing length between the brackets. In this case x has been created with a length of 0.
- If you want to create a slice you should use the built-in make function:

```
x := make([]float64, 5)
```

Slices

- This creates a slice that is associated with an underlying float64 array of length 5. Slices are always associated with some array, and although they can never be longer than the array, they can be smaller. The make function also allows a 3rd parameter:

```
x := make([]float64, 5, 10)
```

- 10 represents the capacity of the underlying array which the slice points to:
- Another way to create slices is to use the [low : high] expression:

```
arr := [5]float64{1,2,3,4,5}
```

```
x := arr[0:5]
```

Slice Functions

- Go includes two built-in functions to assist with slices: `append` and `copy`.

```
func main() {  
    slice1 := []int{1,2,3}  
    slice2 := append(slice1, 4, 5)  
    fmt.Println(slice1, slice2)  
}
```

After running this program `slice1` has `[1,2,3]` and `slice2` has `[1,2,3,4,5]`. `append` creates a new slice by taking an existing slice (the first argument) and appending all the following arguments to it.

```
func main() {  
    slice1 := []int{1,2,3}  
    slice2 := make([]int, 2)  
    copy(slice2, slice1)  
    fmt.Println(slice1, slice2)  
}
```

- After running this program `slice1` has `[1,2,3]` and `slice2` has `[1,2]`. The contents of `slice1` are copied into `slice2`, but since `slice2` has room for only two elements only the first two elements of `slice1` are copied.

Golang Slice and Append Function

- A slice is a portion or segment of an array. Or it is a view or partial view of an underlying array to which it points. You can access the elements of a slice using the slice name and index number just as you do in an array. You cannot change the length of an array, but you can change the size of a slice.
- Contents of a slice are actually the pointers to the elements of an array. It means if you change any element in a slice, the underlying array contents also will be affected. The syntax for creating a slice is

```
var slice_name [] type = array_name[start:end]
```

- This will create a slice named slice_name from an array named array_name with the elements at the index start to end-1.

Golang Slice and Append Function

- There are certain functions like Golang len, Golang append which you can apply on slices

`len(slice_name)` - returns the length of the slice

`append(slice_name, value_1, value_2)` - Golang append is used to append value_1 and value_2 to an existing slice.

`append(slice_name1, slice_name2...)` – appends slice_name2 to slice_name1

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    // declaring array
```

```
    a := [5] string {"one", "two", "three", "four", "five"}
```

```
    fmt.Println("Array after creation:",a)
```

```
    var b [] string = a[1:4] //created a slice named b
```

```
    fmt.Println("Slice after creation:",b)
```

```
    b[0]="changed" // changed the slice data
```

```
    fmt.Println("Slice after modifying:",b)
```

```
    fmt.Println("Array after slice modification:",a)
```

```
}
```

Golang Slice and Append Function

Golang Slice and Append Function

```
package main
import "fmt"
func main() {
a := [5] string {"1","2","3","4","5"}
slice_a := a[1:3]
b := [5] string {"one","two","three","four","five"}
slice_b := b[1:3]
    fmt.Println("Slice_a:", slice_a)
    fmt.Println("Slice_b:", slice_b)
    fmt.Println("Length of slice_a:", len(slice_a))
    fmt.Println("Length of slice_b:", len(slice_b))
    slice_a = append(slice_a,slice_b...) // appending slice
    fmt.Println("New Slice_a after appending slice_b :", slice_a)
    slice_a = append(slice_a,"text1") // appending value
    fmt.Println("New Slice_a after appending text1 :", slice_a)
}
```


Methods(not functions)

A method is a function with a receiver argument. Architecturally, it's between the func keyword and method name. The syntax of a method is

```
func (variable variabletype)  
methodName(parameter1 paramether1type) {  
}
```

```
package main
```

```
import "fmt"
```

```
//declared the structure named emp
```

```
type emp struct {
```

```
    name string
```

```
    address string
```

```
    age int
```

```
}
```

```
//Declaring a function with receiver of the type emp
```

```
func(e emp) display() {
```

```
    fmt.Println(e.name)
```

```
}
```

Methods(not functions)

```
func main() {
```

```
    //declaring a variable of type emp
```

```
    var empdata1 emp
```

```
    //Assign values to members
```

```
    empdata1.name = "John"
```

```
    empdata1.address = "Street-1, Lodon"
```

```
    empdata1.age = 30
```

```
    //declaring a variable of type emp and assign values to members
```

```
    empdata2 := emp {
```

```
        "Raj", "Building-1, Paris", 25}
```

```
    //Invoking the method using the receiver of the type emp
```

```
    // syntax is variable.methodname()
```

```
    empdata1.display()
```

```
    empdata2.display()
```

```
}
```

Methods(not functions)

Packages

- Packages are used to organize the code. In a big project, it is not feasible to write code in a single file. Go programming language allow us to organize the code under different packages
- <https://www.golang-book.com/books/intro/11>.
- This increases code readability and reusability. An executable Go program should contain a package named main and the program execution starts from the function named main. You can import other packages in our program using the syntax

```
import package_name
```

Advantages of Packages

- reduces the chance of having overlapping names. This keeps our function names short and succinct
- organizes code so that its easier to find code you want to reuse.
- speeds up the compiler by only requiring recompilation of smaller chunks of a program. Although we use the package fmt, we don't have to recompile it every time we change our program.

Packages

- Step 1) Create a file called package_example.go and add the below code

```
package main

import "fmt"

//the package to be created
import "calculation"

func main() {
    x,y := 15,10
    //the package will have function Do_add()
    sum := calculation.Do_add(x,y)
    fmt.Println("Sum",sum)
}
```

Packages

- Step 2) First, you should create the package calculation inside a folder with the same name under src folder of the go. The installed path of go can be found from the PATH variable.
- For mac, find the path by executing `echo $PATH`
- So the path is `/usr/local/go`
- For windows, find the path by executing `echo %GOROOT%`
- Here the path is `C:\Go\`

Packages

- Step 3) Navigate to to the src folder(/usr/local/go/src for mac and C:\Go\src for windows). Now from the code, the package name is calculation. Go requires the package should be placed in a directory of the same name under src directory. Create a directory named calculation in src folder.
- Step 4) Create a file called calc.go (You can give any name, but the package name in the code matters. Here it should be calculation) inside calculation directory and add the below code

```
package calculation
```

```
func Do_add(num1 int, num2 int)(int) {
```

```
    sum := num1 + num2
```

```
    return sum
```

```
}
```


Packages

- Step 5) Run the command `go install` from the calculation directory which will compile the `calc.go`.
- Step 6) Now go back to `package_example.go` and run `go run package_example.go`. The output will be `Sum 25`.

Defer, Panic & Recover

- defer which schedules a function call to be run after the function completes. Consider the following example:

```
package main
import "fmt"
func first() {
    fmt.Println("1st")
}
func second() {
    fmt.Println("2nd")
}
func main() {
    defer second()
    first()
}
```

- This program prints 1st followed by 2nd. Basically defer moves the call to second to the end of the function:

```
func main() {
    first()
    second()
}
```

Defer, Panic & Recover

- defer is often used when resources need to be freed in some way. For example when we open a file we need to make sure to close it later. With defer:
- `f, _ := os.Open(filename)`
- `defer f.Close()`
- This has 3 advantages:
- (1) it keeps our Close call near our Open call so it's easier to understand,
- (2) if our function had multiple return statements (perhaps one in an if and one in an else) Close will happen before both of them and
- (3) deferred functions are run even if a run-time panic occurs.

Panic & Recover

- Earlier we created a function that called the panic function to cause a run time error. We can handle a run-time panic with the built-in recover function. recover stops the panic and returns the value that was passed to the call to panic. We might be tempted to use it like this:

```
package main  
import "fmt"  
func main() {  
    panic("PANIC")  
    str := recover()  
    fmt.Println(str)  
}
```

Panic & Recover

- But the call to recover will never happen in this case because the call to panic immediately stops execution of the function. Instead we have to pair it with defer:

```
package main
import "fmt"
func main() {
    defer func() {
        str := recover()
        fmt.Println(str)
    }()
    panic("PANIC")
}
```

- A panic generally indicates a programmer error (for example attempting to access an index of an array that's out of bounds, forgetting to initialize a map, etc.) or an exceptional condition that there's no easy way to recover from. (Hence the name “panic”)

Defer and stacking defers

- Defer statements are used to defer the execution of a function call until the function that contains the defer statement completes execution. Lets learn this with an example:

```
package main

import "fmt"

func sample() {
    fmt.Println("Inside the sample()")
}

func main() {
    //sample() will be invoked only after executing the statements of main()
    defer sample()
    fmt.Println("Inside the main()")
}
```

Defer and stacking defers

- Stacking defer is using multiple defer statements. Suppose you have multiple defer statements inside a function. Go places all the deferred function calls in a stack, and once the enclosing function returns, the stacked functions are executed in the Last In First Out(LIFO) order.

```
package main

import "fmt"

func display(a int) {
    fmt.Println(a)
}

func main() {
    defer display(1)
    defer display(2)
    defer display(3)
    fmt.Println(4)
}
```

I/O Package

```
package file
import (
    "os"
    "syscall"
)
type File struct {
    fd int // file descriptor number
    name string // file name at Open time
}
```


I/O Package

- The first few lines declare the name of the package—file—and then import two packages. The os package hides the differences between various operating systems to give a consistent view of files and so on; here we're going to use its error handling utilities and reproduce the rudiments of its file I/O.
- The other item is the low-level, external syscall package, which provides a primitive interface to the underlying operating system's calls.
- Next is a type definition: the type keyword introduces a type declaration, in this case a data structure called File. To make things a little more interesting, our File includes the name of the file that the file descriptor refers to.

I/O Package

```
func newFile(fd int, name string) *File {  
    if fd < 0 {  
        return nil  
    }  
    return &File{fd, name}  
}  
  
n := new(File)  
n.fd = fd  
n.name = name  
return n
```

I/O Package Example

The io package consists of a few functions, but mostly interfaces used in other packages. The two main interfaces are Reader and Writer. Readers support reading via the Read method. Writers support writing via the Write method. Many functions in Go take Readers or Writers as arguments. For example the io package has a Copy function which copies data from a Reader to a Writer:

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

To read or write to a []byte or a string you can use the Buffer struct found in the bytes package:

```
var buf bytes.Buffer
```

```
buf.Write([]byte("test"))
```

A Buffer doesn't have to be initialized and supports both the Reader and Writer interfaces. You can convert it into a []byte by calling buf.Bytes(). If you only need to read from a string you can also use the strings.NewReader function which is more efficient than using a buffer.

Pointers

- Before explaining pointers let's will first discuss '&' operator. The '&' operator is used to get the address of a variable. It means '&a' will print the memory address of variable a.

```
package main  
import "fmt"  
func main() {  
    a := 20  
    fmt.Println("Address:",&a)  
    fmt.Println("Value:",a)  
}
```

A pointer variable stores the memory address of another variable. You can define a pointer using the syntax. The asterisk(*) represents the variable is a pointer.

```
var variable_name *type
```

Pointers

- When we call a function that takes an argument, that argument is copied to the function:

```
func zero(x int) {  
    x = 0  
}  
  
func main() {  
    x := 5  
    zero(x)  
    fmt.Println(x) // x is still 5  
}
```

- In this program the zero function will not modify the original x variable in the main function. But what if we wanted to? One way to do this is to use a special data type known as a pointer:

Pointers

```
func zero(xPtr *int) {  
    *xPtr = 0  
}  
  
func main() {  
    x := 5  
    zero(&x)  
    fmt.Println(x) // x is 0  
}
```

- Pointers reference a location in memory where a value is stored rather than the value itself. (They point to something else) By using a pointer (*int) the zero function is able to modify the original variable.

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    //Create an integer variable a with value 20
```

```
    a := 20
```

```
    //Create a pointer variable b and assigned the address of a
```

```
    var b *int = &a
```

```
    //print address of a(&a) and value of a
```

```
    fmt.Println("Address of a:",&a)
```

```
    fmt.Println("Value of a:",a)
```

```
    //print b which contains the memory address of a i.e. &a
```

```
    fmt.Println("Address of pointer b:",b)
```

```
    /*b prints the value in memory address which b contains i.e. the value of a
```

```
    fmt.Println("Value of pointer b",*b)
```

```
    //increment the value of variable a using the variable b
```

```
    *b = *b+1
```

```
    //prints the new value using a and *b
```

```
    fmt.Println("Value of pointer b",*b)
```

```
    fmt.Println("Value of a:",a)}
```

Pointers

The * and & operators

- a pointer is represented using the * (asterisk) character followed by the type of the stored value. In the zero function **xPtr** is a pointer to an int.
- * is also used to “**dereference**” pointer variables. **Dereferencing** a pointer gives us access to the value the pointer points to. When we write ***xPtr = 0** we are saying “**store the int 0 in the memory location xPtr refers to**”. If we try **xPtr = 0** instead we will get a compiler error because **xPtr** is not an int it's a ***int**, which can only be given another ***int**.
- Finally we use the & operator to find the address of a variable. **&x** returns a ***int** (pointer to an int) because **x** is an int. This is what allows us to modify the original variable. **&x** in main and **xPtr** in zero refer to the same memory location.

new

- Another way to get a pointer is to use the built-in new function:

```
func one(xPtr *int) {  
    *xPtr = 1  
}  
  
func main() {  
    xPtr := new(int)  
    one(xPtr)  
    fmt.Println(*xPtr) // x is 1  
}
```

- new takes a type as an argument, allocates enough memory to fit a value of that type and returns a pointer to it.

Call code in an external package

```
package main
```

```
import "fmt"
```

```
import "rsc.io/quote"
```

```
func main() {
```

```
    fmt.Println(quote.Go())
```

```
}
```

Structures

- A Structure is a user defined datatype which itself contains one more element of the same or different type.
- Using a structure is a 2 step process.
- First, create(declare) a structure type
- Second, create variables of that type to store values.
- Structures are mainly used when you want to store related data together.
- Consider a piece of employee information which has name, age, and address. You can handle this in 2 ways
- Create 3 arrays - one array stores the names of employees, one stores age and the third one stores address

- Declare a structure type with 3 fields- name, address, and age. Create an array of that structure type where each element is a structure object having name, address, and age.
- The first approach is not efficient. In these kinds of scenarios, structures are more convenient. The syntax for declaring a structure is

Structures

```
type structname struct {  
    variable_1 variable_1_type  
    variable_2 variable_2_type  
    variable_n variable_n_type  
}
```

- An example of a structure declaration is

```
type emp struct {  
    name string  
    address string  
    age int  
}
```

- Here a new user defined type named emp is created. Now, you can create variables of the type emp using the syntax

```
var variable_name struct_name
```

- An example is

```
var empdata1 emp
```

- You can set values for the empdata1 as

```
empdata1.name = "John"
```

```
empdata1.address = "Street-1, Bangalore"
```

```
empdata1.age = 30
```

- You can also create a structure variable and assign values by

```
empdata2 := emp{"Raj", "Building-1, Delhi", 25}
```

Structures

```
package main
```

```
import "fmt"
```

```
//declared the structure named emp
```

```
type emp struct {
```

```
    name string
```

```
    address string
```

```
    age int
```

```
}
```

```
//function which accepts variable of emp type and prints name  
property
```

```
func display(e emp) {
```

```
    fmt.Println(e.name)
```

```
}
```

Structures

Structures

```
func main() {  
    // declares a variable, empdata1, of the type emp  
    var empdata1 emp  
    //assign values to members of empdata1  
    empdata1.name = "John"  
    empdata1.address = "Street-1, London"  
    empdata1.age = 30  
    //declares and assign values to variable empdata2 of type emp  
    empdata2 := emp{"Raj", "Building-1, Paris", 25}  
    //prints the member name of empdata1 and empdata2 using display  
    function  
    display(empdata1)  
    display(empdata2)  
}
```

Struct

- An easy way to make this program better is to use a struct. A struct is a type which contains named fields. For example we could represent a Circle like this:

```
type Circle struct {  
    x float64  
    y float64  
    r float64  
}
```

- The type keyword introduces a new type. It's followed by the name of the type (Circle), the keyword struct to indicate that we are defining a struct type and a list of fields inside of curly braces. Each field has a name and a type. Like with functions we can collapse fields that have the same type:

```
type Circle struct {  
    x, y, r float64  
}
```


Struct Initialization

- We can create an instance of our new Circle type in a variety of ways:

`var c Circle`

- Like with other data types, this will create a local Circle variable that is by default set to zero. For a struct zero means each of the fields is set to their corresponding zero value (0 for ints, 0.0 for floats, "" for strings, nil for pointers, ...) We can also use the new function:

`c := new(Circle)`

- This allocates memory for all the fields, sets each of them to their zero value and returns a pointer. (*Circle) More often we want to give each of the fields a value.

`c := Circle{x: 0, y: 0, r: 5}`

- Or we can leave off the field names if we know the order they were defined:
- `c := Circle{0, 0, 5}`

Struct Fields

- We can access fields using the . operator:

```
fmt.Println(c.x, c.y, c.r)
```

```
c.x = 10
```

```
c.y = 5
```

Let's modify the circleArea function so that it uses a Circle:

```
func circleArea(c Circle) float64 {  
    return math.Pi * c.r*c.r  
}
```

- In main we have:

```
c := Circle{0, 0, 5}
```

```
fmt.Println(circleArea(c))
```

Struct Fields

- One thing to remember is that arguments are always copied in Go. If we attempted to modify one of the fields inside of the `circleArea` function, it would not modify the original variable.

```
func circleArea(c *Circle) float64 {  
    return math.Pi * c.r*c.r  
}
```

- And change main:

```
c := Circle{0, 0, 5}  
fmt.Println(circleArea(&c))
```

Struct Methods

```
func (c *Circle) area() float64 {  
    return math.Pi * c.r*c.r  
}
```

- In between the keyword `func` and the name of the function we've added a "receiver". The receiver is like a parameter – it has a name and a type – but by creating the function in this way it allows us to call the function using the `.` operator:

```
fmt.Println(c.area())
```

- This is much easier to read, we no longer need the `&` operator (Go automatically knows to pass a pointer to the circle for this method) and because this function can only be used with Circles we can rename the function to just `area`.

Struct Methods

```
type Rectangle struct {  
    x1, y1, x2, y2 float64  
}  
  
func (r *Rectangle) area() float64 {  
    l := distance(r.x1, r.y1, r.x1, r.y2)  
    w := distance(r.x1, r.y1, r.x2, r.y1)  
    return l * w  
}  
  
• main has:  
r := Rectangle{0, 0, 10, 10}  
fmt.Println(r.area())
```

Embedded Types

- A struct's fields usually represent the has-a relationship. For example a Circle has a radius. Suppose we had a person struct:

```
type Person struct {  
    Name string  
}  
  
func (p *Person) Talk() {  
    fmt.Println("Hi, my name is", p.Name)  
}
```

- And we wanted to create a new Android struct. We could do this:

```
type Android struct {  
    Person Person  
    Model string  
}
```

Embedded Types

- This would work, but we would rather say an Android is a Person, rather than an Android has a Person. Go supports relationships like this by using an embedded type. Also known as anonymous fields, embedded types look like this:

```
type Android struct {  
    Person  
    Model string  
}
```

- We use the type (Person) and don't give it a name. When defined this way the Person struct can be accessed using the type name:

```
a := new(Android)  
a.Person.Talk()
```

- But we can also call any Person methods directly on the Android:

```
a := new(Android)  
a.Talk()
```

Golang Interface

- Golang **Interface** is a collection of method signatures used by a Type to implement the behavior of objects.
- The main goal of Golang **interface** is to provide method signatures with names, arguments, and return types.
- It is up to a Type to declare and implement the method. An interface in Golang can be declared using the keyword “**interface.**”

Golang Interface

```
type Shape interface {
```

```
    area() float64
```

```
}
```

- Like a struct an interface is created using the type keyword, followed by a name and the keyword interface. But instead of defining fields, we define a “method set”. A method set is a list of methods that a type must have in order to “implement” the interface.
- In our case both Rectangle and Circle have area methods which return float64s so both types implement the Shape interface. By itself this wouldn't be particularly useful, but we can use interface types as arguments to functions:

```
func totalArea(shapes ...Shape) float64 {
```

```
    var area float64
```

```
    for _, s := range shapes {
```

```
        area += s.area()
```

```
    }
```

```
    return area
```

```
}
```

- Mutex is the short form for **mutual exclusion**. Mutex is used when you don't want to allow a resource to be accessed by multiple subroutines at the same time.
- Mutex has 2 methods - **Lock** and **Unlock**. Mutex is contained in **sync** package. So, you have to import the **sync** package.
- The statements which have to be mutually exclusively executed can be placed inside **mutex.Lock()** and **mutex.Unlock()**.
- Mutex is counting the number of times a loop is executed. In this program we expect routine to run loop 10 times and the count is stored in sum.
- You call this routine 3 times so the total count should be 30. The count is stored in a global variable count.

Mutex

Mutex

```
package main
```

```
import "fmt"
```

```
import "time"
```

```
import "strconv"
```

```
import "math/rand"
```

```
//declare count variable, which is accessed by all the routine instances
```

```
var count = 0
```

```
//copies count to temp, do some processing(increment) and store back to count //random delay is added between reading and writing of count variable
```

```
func process(n int) {
```

```
//loop incrementing the count by 10
```

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

```
        time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
```

```
        temp := count
```

```
        temp++
```

```
        time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
```

```
        count = temp
```

```
    }
```

```
    fmt.Println("Count after i="+strconv.Itoa(n)+" Count:", strconv.Itoa(count))
```

```
}
```

```
func main(){
```

```
    process(2)
```

```
}
```

Mutex

```
func main() {  
    //loop calling the process() 3 times  
    for i := 1; i < 4; i++ {  
        go process(i)  
    }  
    //delay to wait for the routines to complete  
    time.Sleep(25 * time.Second)  
    fmt.Println("Final Count:", count)  
}
```

- Suppose soon after performing step 3 by **Mutex** goroutine1; another goroutine might have an old value say 3 does the above steps and store 4 back, which is wrong.
- This can be prevented by using mutex which causes other routines to wait when one routine is already using the variable.
- Now You will run the program with mutex. Here the above mentioned 3 steps are executed in a mutex.

Mutex

```
package main
import "fmt"
import "time"
import "sync"
import "strconv"
import "math/rand"
//declare a mutex instance
var mu sync.Mutex
//declare count variable, which is accessed by all the routine instances
var count = 0
//copies count to temp, do some processing(increment) and store back to count
//random delay is added between reading and writing of count variable
func process(n int) {
//loop incrementing the count by 10
for i := 0; i < 10; i++ {
time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
//lock starts here
mu.Lock()
temp := count
temp++
time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
count = temp
//lock ends here
mu.Unlock()
}
fmt.Println("Count after i="+strconv.Itoa(n)+" Count:", strconv.Itoa(count))
}
```

Mutex

```
func main() {  
    //loop calling the process() 3 times  
    for i := 1; i < 4; i++ {  
        go process(i)  
    }  
    //delay to wait for the routines to complete  
    time.Sleep(25 * time.Second)  
    fmt.Println("Final Count:", count)  
}
```

Concurrency

- Go supports concurrent execution of tasks. It means Go can execute multiple tasks simultaneously.
- It is different from the concept of parallelism. In parallelism, a task is split into small subtasks and are executed in parallel.
- But in concurrency, multiple tasks are being executed simultaneously. Concurrency is achieved in Go using Goroutines and Channels.


```
package main
```

```
import (
```

```
    "fmt"
```

```
    "time"
```

```
)
```

```
func readword(ch chan string) {
```

```
    fmt.Println("Type a word, then hit Enter.")
```

```
    var word string
```

```
    fmt.Scanf("%s", &word)
```

```
    ch <- word
```

```
}
```

```
func timeout(t chan bool) {
```

```
    time.Sleep(5 * time.Second)
```

```
    t <- false
```

```
}
```

Concurrency

```
func main() {
```

```
    t := make(chan bool)
```

```
    go timeout(t)
```

```
    ch := make(chan string)
```

```
    go readword(ch)
```

```
    select {
```

```
    case word := <-ch:
```

```
        fmt.Println("Received", word)
```

```
    case <-t:
```

```
        fmt.Println("Timeout.")
```

```
    }
```

```
}
```

Goroutines

- A goroutine is a function which can run concurrently with other functions. Usually when a function is invoked the control gets transferred into the called function, and once its completed execution control returns to the calling function. The calling function then continues its execution. The calling function waits for the invoked function to complete the execution before it proceeds with the rest of the statements.
- The calling function will not wait for the execution of the invoked function to complete. It will continue to execute with the next statements. You can have multiple goroutines in a program.
- Also, the main program will exit once it completes executing its statements and it will not wait for completion of the goroutines invoked.
- Goroutine is invoked using keyword go followed by a function call.

Example

```
go add(x,y)
```

Goroutines

```
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```

Goroutines

```
package main
import "fmt"
func display() {
    for i:=0; i<5; i++ {
        fmt.Println("In display")
    }
}
func main() {
    //invoking the goroutine display()
    go display()
    //The main() continues without waiting for display()
    for i:=0; i<5; i++ {
        fmt.Println("In main")
    }
}
```

Goroutines

- Here the main program completed execution even before the goroutine started. The `display()` is a goroutine which is invoked using the syntax
- `go function_name(parameter list)`
- The `main()` doesn't wait for the `display()` to complete, and the `main()` completed its execution before the `display()` executed its code. So the print statement inside `display()` didn't get printed.
- Now we modify the program to print the statements from `display()` as well. We add a time delay of 2 sec in the for loop of `main()` and a 1 sec delay in the for loop of the `display()`.

Goroutines

```
package main
import (
    "fmt"
    "time"
    "math/rand"
)
func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}
func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

Goroutines

```
package main
import "fmt"
import "time"
func display() {
    for i:=0; i<5; i++ {
        time.Sleep(1 * time.Second)
        fmt.Println("In display")
    }
}
func main() {
    //invoking the goroutine display()
    go display()
    for i:=0; i<5; i++ {
        time.Sleep(2 * time.Second)
        fmt.Println("In main")
    }
}
```

Goroutines

```
package main
import (
    "fmt"
    "time"
    "math/rand"
)
func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}
func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
```


Channels

- Channels are a way for functions to communicate with each other.
- It can be thought as a medium to where one routine places data and is accessed by another routine in Golang server.
- A channel can be declared with the syntax

```
channel_variable := make(chan datatype)
```

Example:

```
ch := make(chan int)
```

- You can send data to a channel using the syntax

```
channel_variable <- variable_name
```

- Example

```
ch <- x
```

- You can receive data from a channel using the syntax

```
variable_name := <- channel_variable
```

- Example

```
y := <- ch
```

- In the above Go language examples of goroutine, you have seen the main program doesn't wait for the goroutine. But that is not the case when channels are involved. Suppose if a goroutine pushes data to channel, the main() will wait on the statement receiving channel data until it gets the data.

Channels

```
package main
import "fmt"
import "time"
func display() {
    time.Sleep(5 * time.Second)
    fmt.Println("Inside display()")
}
func main() {
    go display()
    fmt.Println("Inside main()")
}
```

Channels

The main() finished the execution and did exit before the goroutine executes. So the print inside the display() didn't get executed. Now modify the above program to use channels and see the behaviour.

Channels

```
package main

import "fmt"
import "time"

func display(ch chan int) {
    time.Sleep(5 * time.Second)
    fmt.Println("Inside display()")
    ch <- 1234
}

func main() {
    ch := make(chan int)
    go display(ch)
    x := <-ch
    fmt.Println("Inside main()")
    fmt.Println("Printing x in main() after taking from channel:",x)
}
```

Channels

- Here what happens is the main() on reaching `x := <-ch` will wait for data on channel `ch`. The `display()` has a wait of 5 seconds and then push data to the channel `ch`.
- The main() on receiving the data from the channel gets unblocked and continues its execution.
- The sender who pushes data to channel can inform the receivers that no more data will be added to the channel by closing the channel. This is mainly used when you use a loop to push data to a channel. A channel can be closed using

`close(channel_name)`

- And at the receiver end, it is possible to check whether the channel is closed using an additional variable while fetching data from channel using

`variable_name, status := <- channel_variable`

- If the status is True it means you received data from the channel. If false, it means you are trying to read from a closed channel

- You can also use channels for communication between goroutines. Need to use 2 goroutines – one pushes data to the channel and other receives the data from the channel. See the below program

Channels

```
package main
import "fmt"
import "time"
//This subroutine pushes numbers 0 to 9 to the channel and closes the channel
func add_to_channel(ch chan int) {
    fmt.Println("Send data")
    for i:=0; i<10; i++ {
        ch <- i //pushing data to channel
    }
    close(ch) //closing the channel
}
//This subroutine fetches data from the channel and prints it.
func fetch_from_channel(ch chan int) {
    fmt.Println("Read data")
    for {
        //fetch data from channel
        x, flag := <- ch
```

```
//flag is true if data is received from the channel
```

```
//flag is false when the channel is closed
```

```
if flag == true {
```

```
    fmt.Println(x)
```

```
}else{
```

```
    fmt.Println("Empty channel")
```

```
    break
```

```
}
```

```
}
```

```
}
```

```
func main() {
```

```
    //creating a channel variable to transport integer values
```

```
    ch := make(chan int)
```

```
    //invoking the subroutines to add and fetch from the channel
```

```
    //These routines execute simultaneously
```

```
    go add_to_channel(ch)
```

```
    go fetch_from_channel(ch)
```

```
    //delay is to prevent the exiting of main() before goroutines finish
```

```
    time.Sleep(5 * time.Second)
```

```
    fmt.Println("Inside main()")
```

```
}
```

Channels

Channels

- Here there are 2 subroutines one pushes data to the channel and other prints data to the channel. The function `add_to_channel` adds the numbers from 0 to 9 and closes the channel. Simultaneously the function `fetch_from_channel` waits at
- `x, flag := <- ch` and once the data become available, it prints the data. It exits once the flag is false which means the channel is closed.
- The `wait` in the `main()` is given to prevent the exiting of `main()` until the goroutines finish the execution.


```
package main
import (
    "fmt"
    "time"
)
func pinger(c chan string) {
    for i := 0; ; i++ {
        c <- "ping"
    }
}
func printer(c chan string) {
    for {
        msg := <- c
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}
func main() {
    var c chan string = make(chan string)
    go pinger(c)
    go printer(c)
    var input string
    fmt.Scanln(&input)
}
```

Channels

Channels

- Above program will print “ping” forever (hit enter to stop it). A channel type is represented with the keyword `chan` followed by the type of the things that are passed on the channel (in this case we are passing strings).
- The `<-` (left arrow) operator is used to send and receive messages on the channel. `c <- "ping"` means send "ping". `msg := <- c` means receive a message and store it in `msg`. The `fmt` line could also have been written like this: `fmt.Println(<-c)` in which case we could remove the previous line.
- Using a channel like this synchronizes the two goroutines. When `pinger` attempts to send a message on the channel it will wait until `printer` is ready to receive the message. this is known as blocking

Select

- Select can be viewed as a switch statement which works on channels. Here the case statements will be a channel operation.
- Usually, each case statements will be read attempt from the channel.
- When any of the cases is ready(the channel is read), then the statement associated with that case is executed.
- If multiple cases are ready, it will choose a random one. You can have a default case which is executed if none of the cases is ready.

```
package main
import "fmt"
import "time"
//push data to channel with a 4 second delay
func data1(ch chan string) {
    time.Sleep(4 * time.Second)
    ch <- "from data1()"
}
//push data to channel with a 2 second delay
func data2(ch chan string) {
    time.Sleep(2 * time.Second)
    ch <- "from data2()"
}
```

Select

```
func main() {
```

```
//creating channel variables for transporting string values
```

```
chan1 := make(chan string)
```

```
chan2 := make(chan string)
```

```
//invoking the subroutines with channel variables
```

```
go data1(chan1)
```

```
go data2(chan2)
```

```
//Both case statements wait for data in the chan1 or chan2. //chan2 gets data first since the  
delay is only 2 sec in data2(). //So the second case will execute and exits the select block
```

```
select {
```

```
case x := <-chan1:
```

```
    fmt.Println(x)
```

```
case y := <-chan2:
```

```
    fmt.Println(y)
```

```
}
```

```
}
```

Select

Select

- Here the select statement waits for data to be available in any of the channels. The `data2()` adds data to the channel after a sleep of 2 seconds which will cause the second case to execute.
- Add a default case to the select in the same program and see the output. Here, on reaching select block, if no case is having data ready on the channel, it will execute the default block without waiting for data to be available on any channel.

Select

```
package main
import "fmt"
import "time"
//push data to channel with a 4 second delay
func data1(ch chan string) {
    time.Sleep(4 * time.Second)
    ch <- "from data1()"
}
//push data to channel with a 2 second delay
func data2(ch chan string) {
    time.Sleep(2 * time.Second)
    ch <- "from data2()"
}
```

Documentation

- Go has the ability to automatically generate documentation for packages we write in a similar way to the standard package documentation. In a terminal run this command:

```
go doc golang-book/chapter11/math Average
```

```
go doc fmt
```

```
go doc math
```

```
go doc strings
```

- You should see information displayed for the function we just wrote. We can improve this documentation by adding a comment before the function:
- ```
func Average(xs []float64) float64 {}
```



# Godoc installation

- `go get -v golang.org/x/tools/cmd/godoc`
- `go doc fmt`
- <https://godoc.org/>

# Error handling

- Errors are abnormal conditions like closing a file which is not opened, open a file which doesn't exist, etc.
- Functions usually return errors as the last return value.

# File handling

- To open a file in Go use the Open function from the os package.
- How to read the contents of a file and display them on the terminal:

```
package main
import (
 "fmt"
 "os"
)
func main() {
 file, err := os.Open("test.txt")
 if err != nil {
 // handle the error here
 return
 }
 defer file.Close()
 // get the file size
 stat, err := file.Stat()
 if err != nil {
 return
 }
 // read the file
 bs := make([]byte, stat.Size())
 _, err = file.Read(bs)
 if err != nil {
 return
 }
 str := string(bs)
 fmt.Println(str)
}
```

# File handling

We use `defer file.Close()` right after opening the file to make sure the file is closed as soon as the function completes. Reading files is very common, so there's a shorter way to do this:

# File Closing

```
package main
import (
 "fmt"
 "io/ioutil"
)
func main() {
 bs, err := ioutil.ReadFile("test.txt")
 if err != nil {
 return
 }
 str := string(bs)
 fmt.Println(str)
}
```

```
package main
```

```
import (
```

```
 "os"
```

```
)
```

```
func main() {
```

```
 file, err := os.Create("test.txt")
```

```
 if err != nil {
```

```
 // handle the error here
```

```
 return
```

```
 }
```

```
 defer file.Close()
```

```
 file.WriteString("test")
```

```
}
```

# File Creation

To get the contents of a directory we use the same `os.Open` function but give it a directory path instead of a file name. Then we call the `Readdir` method:

```
package main
import (
 "fmt"
 "os"
)func main() {
 dir, err := os.Open(".")
 if err != nil {
 return
 }
 defer dir.Close()
 fileInfos, err := dir.Readdir(-1)
 if err != nil {
 return
 }
 for _, fi := range fileInfos {
 fmt.Println(fi.Name())
 }
}
```

# Listing Directory

Often we want to recursively walk a folder (read the folder's contents, all the sub-folders, all the sub-sub-folders, ...). To make this easier there's a Walk function provided in the path/filepath package:

```
package main

import (
 "fmt"
 "os"
 "path/filepath"
)

func main() {
 filepath.Walk(".", func(path string, info os.FileInfo, err error) error {
 fmt.Println(path)
 return nil
 })
}
```

# Listing Directory Recursively



```
package main
```

```
import "fmt"
```

```
import "os"
```

```
//function accepts a filename and tries to open it.
```

```
func fileopen(name string) {
```

```
 f, er := os.Open(name)
```

```
//er will be nil if the file exists else it returns an error object
```

```
 if er != nil {
```

```
 fmt.Println(er)
```

```
 return
```

```
 }else{
```

```
 fmt.Println("file opened", f.Name())
```

```
 }
```

```
}
```

```
func main() {
```

```
 fileopen("invalid.txt")
```

```
}
```

# Error handling

Go has a built-in type for errors that we have already seen (the error type). We can create our own errors by using the New function in the errors package:

```
package main

import "errors"

func main() {
 err := errors.New("error message")
}
```

# Custom errors

- Using this feature, you can create custom errors.
- This is done by using `New()` of error package. We will rewrite the above program to make use of custom errors.

# Custom errors

```
package main
import "fmt"
import "os"
import "errors"
//function accepts a filename and tries to open it.
func fileopen(name string) (string, error) {
 f, er := os.Open(name)
 //er will be nil if the file exists else it returns an error object
 if er != nil {
 //created a new error object and returns it
 return "", errors.New("Custom error message: File name is wrong")
 }else{
 return f.Name(),nil
 }
}
```

# Custom errors

```
func main() {
 //receives custom error or nil after trying to open the file
 filename, error := fileopen("invalid.txt")
 if error != nil {
 fmt.Println(error)
 }else{
 fmt.Println("file opened", filename)
 }
}
```

# List

- The container/list package implements a doubly-linked list. A linked list is a type of data structure that looks like this:
- Each node of the list contains a value (1, 2, or 3 in this case) and a pointer to the next node. Since this is a doubly-linked list each node will also have pointers to the previous node. This list could be created by this program:

# List

```
package main
import ("fmt" ; "container/list")
func main() {
 var x list.List
 x.PushBack(1)
 x.PushBack(2)
 x.PushBack(3)
 for e := x.Front(); e != nil; e=e.Next() {
 fmt.Println(e.Value.(int))
 }
}
```

The zero value for a List is an empty list (a \*List can also be created using list.New). Values are appended to the list using PushBack. We loop over each item in the list by getting the first element, and following all the links until we reach nil.

# Testing

Go includes a special program that makes writing tests easier, so let's create some tests for the package. In the math folder from chapter11 create a new file called math\_test.go that contains this:

```
package math
import "testing"
func TestAverage(t *testing.T) {
 var v float64
 v = Average([]float64{1,2})
 if v != 1.5 {
 t.Error("Expected 1.5, got ", v)
 }
}
```

Now run this command:

```
go test
```



```

package math

import "testing"

type testpair struct {
 values []float64
 average float64
}

var tests = []testpair{
 { []float64{1,2}, 1.5 },
 { []float64{1,1,1,1,1,1}, 1 },
 { []float64{-1,1}, 0 },
}

func TestAverage(t *testing.T) {
 for _, pair := range tests {
 v := Average(pair.values)
 if v != pair.average {
 t.Error(
 "For", pair.values,
 "expected", pair.average,
 "got", v,
)
 }
 }
}

```

# Testing

# Reading files

- Files are used to store data. Go allows us to read data from the files. First create a file, **data.txt**, in your present directory with the below content.

Line one

Line two

Line three

- Now run the below program to see it prints the contents of the entire file as output

# Reading files

```
package main
import "fmt"
import "io/ioutil"
func main() {
 data, err := ioutil.ReadFile("data.txt")
 if err != nil {
 fmt.Println("File reading error", err)
 return
 }
 fmt.Println("Contents of file:", string(data))
}
```

- Here the **data, err := ioutil.ReadFile("data.txt")** reads the data and returns a byte sequence. While printing it is converted to string format.

```
package main
```

```
import "fmt"
```

```
import "os"
```

```
func main() {
```

```
 f, err := os.Create("file1.txt")
```

```
 if err != nil {
```

```
 fmt.Println(err)
```

```
 return
```

```
 }
```

```
 l, err := f.WriteString("Write Line one")
```

```
 if err != nil {
```

```
 fmt.Println(err)
```

```
 f.Close()
```

```
 return
```

```
 }
```

```
 fmt.Println(l, "bytes written")
```

```
 err = f.Close()
```

```
 if err != nil {
```

```
 fmt.Println(err)
```

```
 return
```

```
 }
```

```
}
```

# Writing files

# Testing

```
func ExtractUsername(email string) string {
 at := strings.Index(email, "@")
 return email[:at]
}
```

```
package main
```

```
import (
```

```
 "testing"
```

```
)
```

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

```
 t.Run("withoutDot", func(t *testing.T) {
```

```
 username := ExtractUsername("tbarua1@google.com")
```

```
 if username != "r" {
```

```
 t.Fatalf("Got: %v\n", username)
```

```
 }
```

```
 })
```

```
 t.Run("withDot", func(t *testing.T) {
```

```
 username := ExtractUsername("jonh.smith@example.com")
```

```
 if username != "jonh.smith" {
```

```
 t.Fatalf("Got: %v\n", username)
```

```
 }
```

```
 })
```

```
}
```

# Testing

```

package main
import (
 "fmt"
 "time"
)

func main() {
 p := fmt.Println
 now := time.Now()
 p(now)
 then := time.Date(
 2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
 p(then)
 p(then.Year())
 p(then.Month())
 p(then.Day())
 p(then.Hour())
 p(then.Minute())
 p(then.Second())
 p(then.Nanosecond())
 p(then.Location())
 p(then.Weekday())
 p(then.Before(now))
 p(then.After(now))
 p(then.Equal(now))
 diff := now.Sub(then)
 p(diff)
 p(diff.Hours())
 p(diff.Minutes())
 p(diff.Seconds())
 p(diff.Nanoseconds())
 p(then.Add(diff))
 p(then.Add(-diff))
}

```

# Time Api

```
package main
```

```
import (
```

```
 "encoding/xml"
```

```
 "fmt"
```

```
)
```

```
type Plant struct {
```

```
 XMLName xml.Name `xml:"plant"`
```

```
 Id int `xml:"id,attr"`
```

```
 Name string `xml:"name"`
```

```
 Origin []string `xml:"origin"`
```

```
}
```

```
func (p Plant) String() string {
```

```
 return fmt.Sprintf("Plant id=%v, name=%v, origin=%v",
```

```
 p.Id, p.Name, p.Origin)
```

```
}
```

# XML



```

func main() {
 coffee := &Plant{Id: 27, Name: "Coffee"}
 coffee.Origin = []string{"Ethiopia", "Brazil"}
 out, _ := xml.MarshalIndent(coffee, " ", " ")
 fmt.Println(string(out))
 fmt.Println(xml.Header + string(out))
 var p Plant
 if err := xml.Unmarshal(out, &p); err != nil {
 panic(err)
 }
 fmt.Println(p)
 tomato := &Plant{Id: 81, Name: "Tomato"}
 tomato.Origin = []string{"Mexico", "California"}
 type Nesting struct {
 XMLName xml.Name `xml:"nesting"`
 Plants []*Plant `xml:"parent>child>plant"`
 }
 nesting := &Nesting{}
 nesting.Plants = []*Plant{coffee, tomato}
 out, _ = xml.MarshalIndent(nesting, " ", " ")
 fmt.Println(string(out))
}

```

# XML

```
package main
```

```
import (
```

```
"encoding/json"
```

```
"fmt"
```

```
)
```

```
type Bird struct {
```

```
 Species string
```

```
 Description string
```

```
}
```

```
func main() {
```

```
 birdJson := `{"species": "pigeon", "description": "likes to perch on rocks"}`
```

```
 var bird Bird
```

```
 json.Unmarshal([]byte(birdJson), &bird)
```

```
 fmt.Printf("Species: %s, Description: %s", bird.Species, bird.Description)
```

```
 //Species: pigeon, Description: likes to perch on rocks
```

```
}
```

# JSON

```
package main
import (
 "encoding/json"
 "fmt"
 "os"
)
type response1 struct {
 Page int
 Fruits []string
}
type response2 struct {
 Page int `json:"page"`
 Fruits []string `json:"fruits"`
}
func main() {
 bolB, _ := json.Marshal(true)
 fmt.Println(string(bolB))
 intB, _ := json.Marshal(1)
 fmt.Println(string(intB))
 fltB, _ := json.Marshal(2.34)
 fmt.Println(string(fltB))
 strB, _ := json.Marshal("gopher")
 fmt.Println(string(strB))
 slcD := []string{"apple", "peach", "pear"}
 slcB, _ := json.Marshal(slcD)
 fmt.Println(string(slcB))
 mapD := map[string]int{"apple": 5, "lettuce": 7}
 mapB, _ := json.Marshal(mapD)
 fmt.Println(string(mapB))
}
```

# JSON

# JSON

```
res1D := &response1{
 Page: 1,
 Fruits: []string{"apple", "peach", "pear"}}
res1B, _ := json.Marshal(res1D)
fmt.Println(string(res1B))
res2D := &response2{
 Page: 1,
 Fruits: []string{"apple", "peach", "pear"}}
res2B, _ := json.Marshal(res2D)
fmt.Println(string(res2B))
byt := []byte(`{"num":6.13,"strs":["a","b"]}`)
var dat map[string]interface{}
if err := json.Unmarshal(byt, &dat); err != nil {
 panic(err)
}
fmt.Println(dat)
num := dat["num"].(float64)
fmt.Println(num)
strs := dat["strs"].([]interface{})
str1 := strs[0].(string)
fmt.Println(str1)
str := `{"page": 1, "fruits": ["apple", "peach"]}`
res := response2{}
json.Unmarshal([]byte(str), &res)
fmt.Println(res)
fmt.Println(res.Fruits[0])
enc := json.NewEncoder(os.Stdout)
d := map[string]int{"apple": 5, "lettuce": 7}
enc.Encode(d)
```

```
package main
```

```
import (
```

```
 "fmt"
```

```
 "log"
```

```
 "net/http"
```

```
)
```

```
func helloFunc(w http.ResponseWriter, r *http.Request) {
```

```
 fmt.Fprintf(w, "Hello world!")
```

```
}
```

```
func main() {
```

```
 http.HandleFunc("/", helloFunc)
```

```
 log.Fatal(http.ListenAndServe(":8080", nil))
```

```
}
```

# Web App

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |



