**Introduction**

**Go** is a high-level programming language developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. **Go** is statically typed as opposed to dynamically typed languages like Python, means variables are declared to be of particular type before usage. However, Go allows you to use variables in somewhat dynamic fashion by using using short declaration, where a variable could be used locally without declaration by assigning a value to it using the **walrus operator** :=. The compiler then verifies its type based on the value it holds. The syntax of Golang is similar to C, but also has memory safety, garbage collection, and concurrency. Concurrency is one of the standard features of Go. Contemporary languages like Python and C++ do not consider concurrency as a standard feature, but is accessible by importing appropriate modules to access the functionality.

Go is a compiled high-level programming language. However, programs could be executed without compilation (similar to Python) but runs slower.

The hello world program

//Sample Program "Hello World!"

package main

import "fmt"

func main() {

fmt.Print("Hello World!")

}

import is similar to include in c/c++. It is also similar to

Python when accessing resources.

func keyword is used to define a function. Here, the function

that we are defining is the function main(). Again, this is similar

to int main(void) in C/C++ when defining the main function. When we execute

the program it always executes the code of the main function first. It is our

start up function of our program.

It should be noted that the block marker { which means begin is written

on the same line as the definition of the function. If we try to moved it

on the next line, we will encounter errors, unlike in C/C++ in which such case

is valid.

The statement fmt.Print("Hello World!") prints the string which is

enclosed in double quotes. The Print method is accessible through the fmt module.

Statements are separated by a new line, unlike in C/C++, where statements are

terminated using the semi-colon ;.

Finally, the block marker } which means end encloses the codes belonging

to the main function.

**Types of imports**

Direct import - both single and multiple packages can be imported one by one using the import keyword.

import “fmt”

Grouped import - you don’t have to write the import keyword multiple times, instead, you can use the keyword import followed by parentheses, and mention all the packages inside the parentheses that you wish to import

import (

“fmt”

“math”

)

Nested import - importing a sub-package from a larger package file.

import “math/rand”

Aliased import - you do not need to write the full name of the package again and again, use aliases instead.

import m “math”

import f “fmt”

Blank import - we could import packages that we don’t always use in the program. To avoid errors we use blank imports.

import \_ “math”

Relative import - imports from $GOPATH directory (e.g. cloud)

import “github.com/go-sql-driver/mysql”

**Printing output**

The **.Println()** method prints the output then advances to the next line

fmt.Println("Hello World!")

The **.Print()** method prints the output without advancing to the next line.

fmt.Print("a. ")

fmt.Print("Adobo")

The **.Prinf()** method prints in a C-like format.

fmt.Printf("mapA\t%v\n", mapA) //%v prints the default representation //of a value

**var** x int = 5

fmt.Printf("value is %v", x) //prints x as integer.

The **.Sprintf()** method is used to return a formatted string.

City := “Dagupan City”

Population := 549,000

Msg := fmt.Sprintf(“The population of %s is %d”, City, Populatio)

fmt.Println((Msg)

**Common Format Specifiers**

%d Decimal integer

%f Floating-point number

%.nf Floating-point with n digits after the decimal point

%s String

%t Boolean

%v Default format for any value

%q Qouted strings

**Mathematical and Comparison Operators**

**Mathematical Operators**

+ Addition a + b

- Subtraction a - b

\* Multiplication a \* b

/ Division a / b (int / int = int)

% Modulus a % b

**Comparison Operators**

== Equal to

!= Not equal to

> Greater than

< Less than

>= Greater than or equal

<= Less than or equal

**Variables**

To declare variable in go programming, we write

syntax:

var <varlist> <data type>

Go variables are automatically initialized to zero or null upon declaration. Unlike other languages like C or C++, variables assumes a garbage value upon declaration.

example:

var a int

var x,y,z int

**Short Declaration**

Short declaration of variables allows us to declare dynamic variables.

Syntax:

a := 1

The := assignment operator (walrus) is used to initialize the variable (a) to 1, thus, assumes an integer type.

**Data Types**

3 Basic Data Types

bool represents a boolean value and is either true or false

numeric represents integer types, floating point values, and complex

types

string represents a string value

bool true & false

int contains signed 32 or 64 bit

uint contains unsigned 32 or 64 bit

float32 contains fractions of 32 bits

float64 contains fractions of 64 bits

string a byte sequence wrapper

**Inputting Data**

var inpt int

fmt.Print("Enter a number: ")

fmt.Scan(&inpt)

Fmt.Println(“The number is “, inpt)

The .Scan method is similar to scanf() is C programming. The ampersand before the variable x represents the address of x. Unlike C however, there is no need to specify the formatting character of the variable x.

**Inputting a String with spaces**

Inputting of a string data with spaces is not as straightforward as using the

.Scan method. To accept string data with spaces, import the *bufio* and the *os* modules then create an object using the *bufio* class. The functionality is taken from the os class using the *.Stdin* method.

gets := bufio.NewReader(os.Stdin)

Use the *.ReadString* method of the gets object to read string inputs.

s1, \_ := gets.ReadString('\n')

The '\n' is the the end of string delimeter, meaning the string

input is terminated upon pressing the enter key.

Example:

package main

import (

"fmt"

"bufio"

"os"

)

func main() {

gets := bufio.NewReader(os.Stdin)

fmt.Print("Enter your name: ")

s1, \_ := gets.ReadString('\n')

fmt.Println(s1)

}

The .ReadString method returns 2 values, the resulting string inputted by the user, and an error code which could be left as blank by putting a comma followed by an underscore: thus the ,\_.

**Loops**

Golang only uses one structure for iteration which is the **for loop**. The for loop serves both as a while loop and a for loop.

**As While Loop**

syntax:

for <condition> {

//some codes here

}

example:

//display numbers 0 to 9

package main

import "fmt"

func main() {

var i int

for i < 10 {

fmt.Println(i)

i++

}

}

**As For Loop**

//display counting numbers 1 to 10

package main

import "fmt"

func main() {

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

fmt.Println(i)

}

}

This syntax is quite similar to C and other C related languages. It should be noted that the variable i is not explicitly declared. This is called short declaration, which is similar to dynamic data typing. The variable i which is declared inside the for loop is local to that block.

**Loop Control Statements**

**continue statement**

The continue statement transfers loop control back to the conditional statement. Statements below it would be skipped.

Example:

i := 1



for i < 10 {

if i == 5 {



i++

continue



}

fmt.Println(i)

i++

}

Output:

1

2

3

4

6

7

8

9

**break statement**

The break statement exits the loop all together without finishing the loop. It is usually convenient to use when exiting an infinite loop or when we want to exit a program prematurely.

Example:

//inputs a number until a zero is typed

package main

import “fmt”

func main() {

var x int

for true {

fmt.Print(“Enter a number, 0 to exit: “)

fmt.Scan(&x)

if x == 0 {

break



}

}

fmt.Println(“finished”)



}

**goto Statement**

The goto statement allows a program to jump to a specific location (<labelName:>)

Example:

**package** main

**import** "fmt"

**func** main() {

**var** x int

Label1:

fmt.Print("Enter a number: ")

fmt.Scan(&x)

**if** x != 0 {

**goto** Label1

}

}

The program above accepts integer inputs repeatedly until a zero is typed. It does not use for loops, instead, a goto statement transfers operation back to the input statement.

**Making Decisions**

Golang uses if the if statement to implement conditional structures. It also uses the switch-case statement to implement direct one-to-one comparison.

**The if statement**

syntax:

if <condition> {

//some codes here

}

if <condition> {

//some codes here

} else {

//some codes here

}

if <condition1> {

//some codes here

} else if <condition2> {

//some codes here

} else {

//some codes here

}

example:

//odd or even

package main

import "fmt"

func main() {

var x, r int

fmt.Print("Enter a positive integer: ")

fmt.Scan(&x)

r = x % 2

if r == 1 {

fmt.Println("Odd")

} else {

fmt.Println("Even")

}

}

**The switch statement**

The switch-case statement is similar to python.

Example:

day := 4

switch(day) {

case 1:

fmt.Println(“Monday”)

case 2:

fmt.Println(“Tuesday”)

case 3:

fmt.Println(“Wednesday”)

case 4:

fmt.Println(“Thursday”)

case 5:

fmt.Println(“Friday”)

case 6:

fmt.Println(“Saturday”)

case 7:

fmt.Println(“Sunday”)

default:

fmt.Println(“Invalid”)

}

**Functions**

A function is a separate block of code or a sub-routine that performs a task/or tasks, and may or may not return a value. Unlike other languages, a function in Golang could return multiple values depending on the design.

syntax:

def <functionName>(parameter list) (return type list> {

//some codes here

}

example:

//square function

package main

import "fmt"

func mySquare(x int) int {

return x \* x

}

func mySquare2(x int) (result int) {

result = x \* x

return

}

func main() {

var x, result int

fmt.Print("Enter an integer: ")

fmt.Scan(&x)

result = mySquare(x)

fmt.Println("Square = ", result)

}

The return value of a function could be explicitly named (see mySquare2), or simply state the data type of the return value.

**Functions with Multiple return values**

Example:

//returns the quotient and the remainder

package main

import "fmt"

func divide(numerator int, denominator int) (int, int) {

var quotient, remainder int

quotient = numerator / denominator

remainder = numerator % denominator

return quotient, remainder

}

func divide2(numerator int, denominator int) (q int, r int) {

q = numerator / denominator

r = numerator % denominator

return

}

func main() {

var x, y int

var result1 int

var result2 int

fmt.Print("Enter numerator: ")

fmt.Scan(&x)

fmt.Print("Enter denominator: ")

fmt.Scan(&y)

result1,result2 = divide2(x, y) //or use divide()

fmt.Println("Quotient = ", result1)

fmt.Println("Remainder = ", result2)

}

Having only one receiving variable from a function that returns 2 values results to an error. On the other hand we avoid receiving errors by directly printing the result.

Example:

fmt.Println(divide2(5,2))

Output:

2 1

Another way to avoid errors from happening during such operation is to disregard the other return value.

Example:

result, \_ := divide2(5,2) //use underscore to avoid getting error

fmt.Println(result)

**Functions with Array Parameters**

Functions with **array as parameters** are **call by value by default**. In other languages like C++, arrays when used as parameters are always call by reference. If we want to make an array parameter a call by reference parameter, then we should use pointers by using the unary operator \*. //example here

**Concurrency**

Concurrency is one of the best feature of the Golang language. It is an inherent feature of Golang. Unlike other languages where we need to import or include additional modules or files to perform concurrency, Golang executes functions concurrently by simple adding the go statement before the name of the function.

Syntax:

*go <functionName>*

Hint:

You can use the go statement to run a function concurrently in conjunction with another function that runs normally.

Example:

sortAsc(arrayX, N) //normal function execution

go sortDesc(array, N) //concurrent function being executed

The sample program below sorts two (2) arrays concurrently in ascending and descending respectively.

**package** main

**import** "fmt"

**func** getItems(arx \*[100]int, n int) {

**var** i int

**for** i < n {

fmt.Print(i+1, ". ")

fmt.Scan(&(\*arx)[i])

i++

}

}

**func** sorta(arx \*[100]int, n int) {

**var** temp int

**for** a := 0; a < n-1; a++ {

**for** b := a + 1; b < n; b++ {

**if** (\*arx)[a] > (\*arx)[b] {

temp = (\*arx)[a]

(\*arx)[a] = (\*arx)[b]

(\*arx)[b] = temp

}

}

}

}

**func** sortd(arx \*[100]int, n int) {

**var** temp int

**for** a := 0; a < n-1; a++ {

**for** b := a + 1; b < n; b++ {

**if** (\*arx)[a] < (\*arx)[b] {

temp = (\*arx)[a]

(\*arx)[a] = (\*arx)[b]

(\*arx)[b] = temp

}

}

}

}

**func** display(arx [100]int, n int) {

**for** a := 0; a < n; a++ {

fmt.Println(arx[a])

}

}

**func** main() {

**var** N int

**var** arX [100]int

**var** arY [100]int

fmt.Print("Enter number of items: ")

fmt.Scan(&N)

fmt.Println("Enter data for arX:")

getItems(&arX, N)

fmt.Println("Enter data for arY:")

getItems(&arY, N)

**go** sorta(&arX, N) //concurrency here

sortd(&arY, N)

fmt.Println("arX sorted in asc order:")

display(arX, N)

fmt.Println("arY sorted in desc order:")

display(arY, N)

}

**Channels**

Channels are a fundamental concept for concurrenly. They are used to communicate between two or more concurrent functions.

**Creating Channels**

Channels are created using the **make()** function with the **chan** keyword and type type of data they will carry.

Example:

ch := make(chan int) //unbuffered

chStr := make(chan string, 5) //buffered

Here, ch is a channel of type string with 5 as the limit.

**Sending and Receiving Values**

Values are sent to a channel using the <- operator, placing the value on the right side.

Example:

ch <-25 //send the integer 25 to the channel.

Values are received from a channel using the <- operator placing the value on the left side.

value := <-ch //receives a value from channel

**Blocking Behaviour**

Unbufferred Channels

Sending to an unbuffered channel blocks, until a receiver is ready to receive. Receiving from an unbuffered channel blocks until sending sends a value. This ensures synchronized communication.

Buffered Channels

Sending to a buffered channel blocks only if the buffer is full. Receiving blocks only if the buffer is empty. This allows asynchronous communication up to the buffer’s capacity.

**Channel Directions**

Channels can be specified as send-only or receive-only for type safety:

Example:

func sendOnly(ch chan <-int) {

ch <- 15

}

func receiveOnly(ch <-chan string) {

msg := <-ch

fmt.Println(msg)

}

**Closing Channels**

Channels can be closed to signal that no more values will be sent.

Example:

close(ch)

Receiving from a closed channel will yield the zero value of the channel’s type and a boolean indicating if the channel is open.

**Select Statement**

The select statement allows a goroutine to wait on multiple channel operations and proceed with the first one that becomes ready.

Example:

select {

case msg1 := ch1:

fmt.Println(“Received from ch1: “, ch1)

case msg2 := ch2:

fmt.Println(Received from ch2: “, ch2)

default:

fmt.Println(“No more message received”)

for range **with Channels**

You can iterate over a channel using a for range loop, which continues until the channel is closed.

Example:

for value := range ch {

fmt.Println(value)

}

**Arrays**

An array is a collection of homogeneous data and is referenced by a single name. The individual elements of the array could be accessed thru an index.

Syntax:

var arrayName[length] type

arrayName := [length] type

arrayName := [length] type {elements}

Sample Declaration:

var arx[10] int

arx := [10] int

arx := [10] int {12,15,73,23,21,43,84,53,20,35}

Sample Program:

//var declaration

//displays the contents of array

**package** main

**import** "fmt"

**func** main() {

**var** arx [10]int

arx[0] = 2

arx[1] = 23

arx[2] = 21

arx[3] = 11

arx[4] = 15

arx[5] = 32

arx[6] = 37

arx[7] = 9

arx[8] = 19

arx[9] = 29

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

fmt.Println(arx[i])

}

}

//shortcut declaration

//displays the contents of array

**package** main

**import** "fmt"

**func** main() {

arx := [10]int{9,23,65,32,65,75,34,26,64,62}

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

fmt.Println(arx[i])

}

}

1. **D Arrays**

Syntax:

var arrayName[row][col] type

arrayName := [row][col] type {array elements}

Sample Declaration:

var arx[3][3] int

arx := [3][3] int {{row 0 elements}, {row 1 elements}, {row 2 elements}}

Sample Program:

**package** main

**import** "fmt"

**func** main() {

arx := [3][3]int{{11, 12, 13}, {21, 22, 23}, {31, 32, 33}}

**for** row := 0; row < 3; row++ {

**for** col := 0; col < 3; col++ {

fmt.Print(arx[row][col], "\t")

}

fmt.Println()

}

}

Sample output:

11 12 13

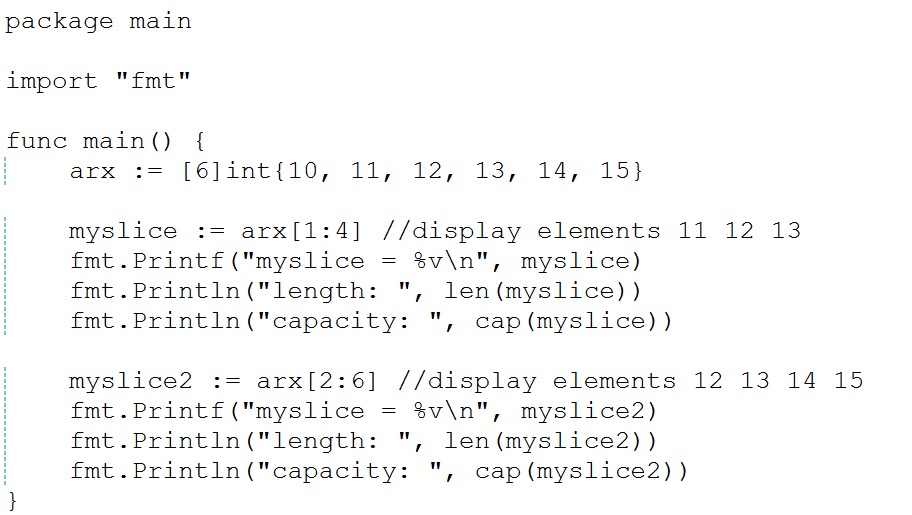
21 22 23

31 32 33

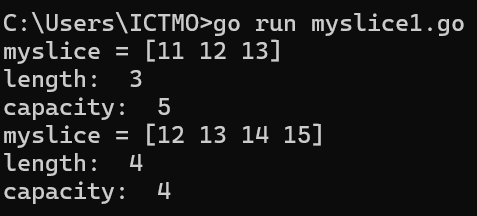
**Slices**

Slices are a powerful and flexible data structures for working with sequences of elements. They are similar to arrays but unlike arrays which have fixed size, slices can dynamically grow or shrink.

Sample Code:



Sample Run:



**Structures in Golang**

A structure (struct) is a collection of variables of heterogeneous types. It is similar to C arrays but contains aggregate types instead of a single data type only.

**Creating a Structure**

Syntax:

type structName struct {

varName1 type

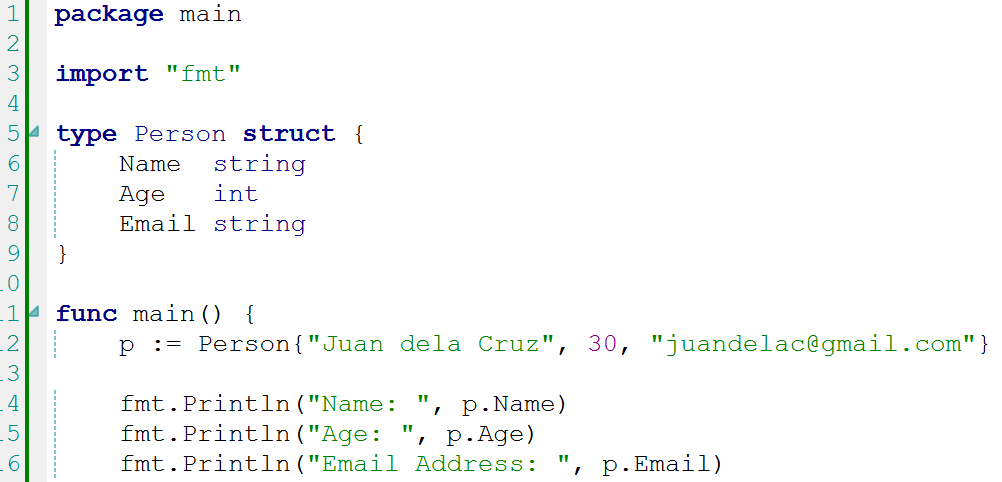
varName2 type

varName3 type

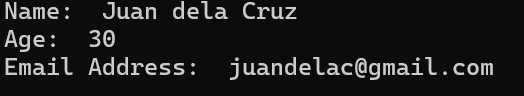
:

}

Example:



Sample Run:



**Array of Structures**

To facilitate inputting of multiple structures or records, we could create an array of structures.

Syntax:

structVar := [size]StructName {}

Example:

**package** main

**import** (

"bufio"

"fmt"

"os"

)

**type** Person **struct** {

Name string

Age int

Email string

}

**func** main() {

**var** N int

p := [50]Person{}

gets := bufio.NewReader(os.Stdin)

fmt.Print("Enter number of inputs: ")

fmt.Scan(&N)

**for** i := 0; i < N; i++ {

fmt.Print("Name: ")

n, \_ := gets.ReadString('\r') //enter key to end string

p[i].Name = n

fmt.Print("Age: ")

fmt.Scan(&p[i].Age)

fmt.Print("Email: ")

fmt.Scan(&p[i].Email)

}

fmt.Println()

fmt.Println("DATA Listing:")

**for** j := 0; j < N; j++ {

fmt.Println("Name: ", p[j].Name)

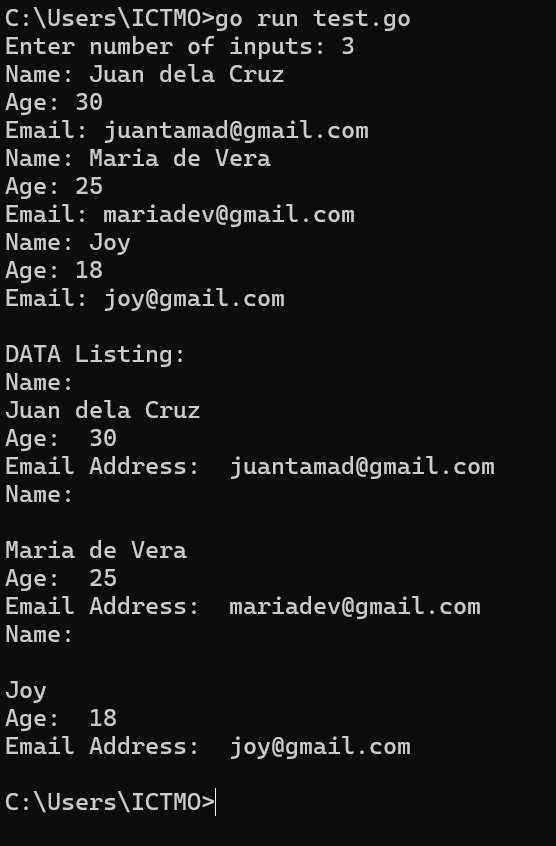
fmt.Println("Age: ", p[j].Age)

fmt.Println("Email Address: ", p[j].Email)

}

}

Sample Run:



**Go Maps**

Maps provides a built-in type that implements a hash table (dictionary in other languages). This data structure stores data in key-value pairs, allowing for very fast data retrieval.

Syntax:

map[KeyType]ValueType{

key-value pairs

}

Example 1:

**package** main

**import** (

"fmt"

)

**func** main() {

**var** mapCars = **map**[int]string{

1: "toyota",

2: "honda",

3: "nissan",

}

**for** i := 0; i <= len(mapCars); i++ {

fmt.Println(mapCars[i])

}

fmt.Println()

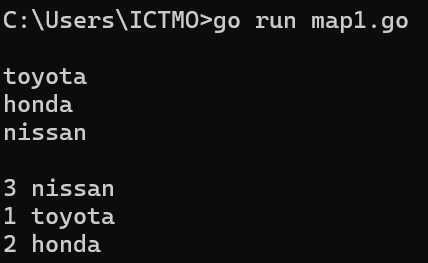
**for** key, value := **range** mapCars {

fmt.Println(key, value)

}

}

Sample run:



Example 2:

**package** main

**import** (

"fmt"

)

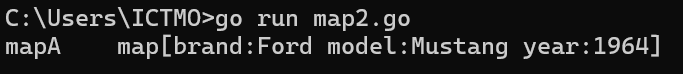
**func** main() {

**var** mapA = **map**[string]string{"brand": "Ford", "model": "Mustang", "year": "1964"}

fmt.Printf("mapA\t%v\n", mapA)

}

Sample run:



Creating an Empty Map

To create an empty map, use the make() function. If you make an empty map in a different way, it will cause a runtime panic.

Example 3:

// map3

**package** main

**import** (

"fmt"

)

**func** main() {

**var** mapA = make(**map**[string]string) //empty

mapA["brand"] = "Toyota"

mapA["model"] = "Vios"

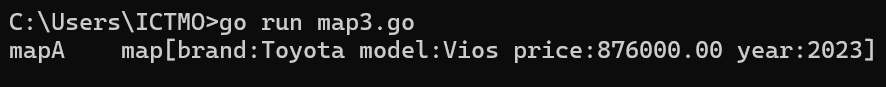
mapA["year"] = "2023"

mapA["price"] = "876000.00"

fmt.Printf("mapA\t%v\n", mapA)

}

Sample run:



**Golang with MySQL**

To connect MySQL to a Go application, you can use the ***go-sql-driver/mysql*** package.

1. Install the MySQL Driver

In the command prompt type the command ***go get -u github.com/go-sql-driver/mysql***press enter key.

1. In your application, import the necessary packages then set up the connection string.

Where: root - user name

123456 - mysql password

@tcp - protocol used

localhost - server name

3306 - mysql port number

dbstudent - database name to open

//sample code

**package** main

**import** (

"database/sql"

"fmt"

\_ "github.com/go-sql-driver/mysql"

)

**func** main() {

dsn := "root:123456@tcp(localhost:3306)/dbstudent"

db, err := sql.Open("mysql", dsn)

**if** err != nil {

fmt.Println("error connecting ", err)

**return**

}

fmt.Println("Connected...")

rows, err := db.Query("SELECT \* FROM tbstudent")

**for** rows.Next() {

**var** sno, last, first, middle, course string

rows.Scan(&sno, &last, &first, &middle, &course)

fmt.Printf("%s, %s, %s, %s, %s \n", sno, last, first, middle, course)

}

}