Savitribai Phule Pune University
(From Academic Year 2021)

T. Y. B. C. A. (Science)
SEMESTER VI, BCA : 367

DSE V: Programming in Go

Name: _____

Roll No:_____Seat No_____

Academic Year: 20__- 20__

From the Chairman's Desk

It gives me a great pleasure to present this workbook prepared by the Board of studies in Computer Applications.

The workbook has been prepared with the objectives of bringing uniformity in implementation of lab assignments across all affiliated colleges, act as a ready reference for both fast and slow learners and facilitate continuous assessment using clearly defined rubrics.

The workbook provides, for each of the assignments, the aims, pre-requisites, related theoretical concepts with suitable examples wherever necessary, guidelines for the faculty/lab administrator, instructions for the students to perform assignments and a set of exercises divided into three sets.

I am thankful to the Chief Editor and the entire team of editors appointed. I am also thankful to the members of BOS. I thank everyone who have contributed directly or indirectly for the preparation of the workbook.

Constructive criticism is welcome and to be communicated to the Chief Editor. Affiliated colleges are requested to collect feedbacks from the students for the further improvements.

I am thankful to Hon. Vice Chancellor of Savitribai Phule Pune University Prof. Dr. Nitin Karmalkar and the Dean of Faculty of Science and Technology Prof. Dr. M G Chaskar for their support and guidance.

Prof. Dr. S. S. Sane

Chairman, BOS in Computer Applications
SPPU, Pune

Chairperson and Editor:

Dr. Kalyani Salla, Modern College of Arts, Science and Commerce, Shivajinagar, Pune: 5

Prepared and Compiled by:

Sr. No.	Assignment Name	Teacher Name	College
1.	Introduction to Go Programming	Dr. Dipali Meher	Modern College of Arts, Science and Commerce, Ganeshkhind, Pune
2.	Functions	Dr. Dipali Meher	Modern College of Arts, Science and Commerce, Ganeshkhind, Pune
3.	Working with data	Dr. Kalyani Salla	Modern College of Arts, Science and Commerce, Shivajinagar, Pune 5
4.	Methods and Interfaces	Dr. Kalyani Salla	Modern College of Arts, Science and Commerce, Shivajinagar, Pune 5
5.	Go routines and channels	Prof. Neeta Nangude	Vidya Pratishthan's Arts, Science & Commerce College, Baramati
6.	Packages and files	Prof. Surekha Thorat	Sangamner College

Reviewed By:
1. Dr. Madhukar Shelar
KTHM College, Nashik: 02

2. Dr. Pallawi Bulakh
PES Modern College, of Arts, Science and Commerce, Ganeshkhind Pune: 16

3. Dr. Sayyad Razak
Ahmednagar College, Ahmednagar

Introduction

1. About the Workbook:
This workbook is intended to be used by TYBCA (Science) students for the Programming in Go Assignments in Semester–VI. This workbook is designed by considering all the practical concepts / topics mentioned in syllabus.

2. The objectives of this Workbook are:
1) Defining the scope of the course
2) To bring uniformity in the practical conduction and implementation in all colleges affiliated to SPPU
3) To have continuous assessment of the course and students
4) Provide ready reference for the students during practical implementation
5) Provide more options to students so that they can have good practice before facing the examination
6) Catering to the demand of slow and fast learners and accordingly provide practice assignments

3. How to use this Workbook:
The Lab on Programming in Go is divided into six assignments. Each assignment has three SETs. It is mandatory for students to complete all the SETs in given slot.

4. Instructions to the students:
Please read the following instructions carefully and follow them.

- Students are expected to carry this workbook every time they come to the lab for practical
- Students should prepare for the assignment by reading the relevant material which is
- mentioned in ready reference
- Instructor will specify which problems to solve in the lab during the allotted slot and student should complete them and get it verified by the Lab Instructor. Viva questions will be asked before you receive a remark for the Assignment in order to check the understanding level of the student for a given topic of the Assignment. Student should spend additional hours in Lab and at home to cover all workbook assignments if needed.
- Students will be assessed for each assignment on a scale from 0 to 5

Not done	0
Incomplete	1
Late Complete	2
Needs Improvement	3
Complete	4
Well done	5

5. Instruction to the Instructors:

- Make sure that students should follow above instructions
- Explain the assignment and related concepts using white board if required or by
- demonstrating the software
- Give specific clues to address student queries which can vary from student to student
- Ask Viva questions before filling the remark for the Assignment in order to check the
- understanding level of the student for a given topic of the Assignment
- Evaluate each assignment carried out by a student on a scale of 5 as specified above by
- ticking appropriate box
- The value should also be entered on assignment completion page of the respective Lab
- course

6. Instructions to the Lab administrator:

You have to ensure appropriate hardware and software is made available to each student.

Table of Contents:

Assignment Completion Sheet

Assignment No	Assignment Name	Marks (5)	Teachers Sign
1	Introduction to Go Programming		
2	Functions		
3	Working with data		
4	Methods and Interfaces		
5	Go routines and channels		
6	Packages and files		
	Total Marks (out of 30)		
	Total Marks (out of 15)		

CERTIFICATE

This is to certify that Mr. / Ms.

_____

has successfully completed the Lab course work for **T.Y.B.C.A., Sem VI, BCA 367, DSE V: Programming in Go** in the academic year _____ and his/her Seat Number is _____.

Instructor **H.O.D / Coordinator**

Internal Examiner **External Examiner**

Installation Procedure

How to download GO on Windows
How to download GO on Linux

1. Go to https://golang.org/dl/ and download the latest version (i.e **1.15.2**) of GoLang in an archive file using wget command as follows:

```
$ wget -c https://golang.org/dl/go1.15.2.linux-amd64.tar.gz   [64-bit]
$ wget -c https://golang.org/dl/go1.15.2.linux-386.tar.gz     [32-bit]
```

2. Next, check the integrity of the tarball by verifying the SHA256 checksum of the archive file using the **shasum** command as below, where the flag -a is used to specify the algorithm to be used:

```
$ shasum -a 256 go1.7.3.linux-amd64.tar.gz

b49fda1ca29a1946d6bb2a5a6982cf07ccd2aba849289508ee0f9918f6bb4552  go1.15.2.linux-amd64.tar.gz
```

ant: To show that the contents of the downloaded archive file are the exact copy provided on the GoLang website, the **256-bit** hash value generated from the command above as seen in the output should be the same as that provided along with the download link.
If that is the case, proceed to the next step, otherwise, download a new tarball and run the check again.

3. Then extract the tar archive files into **/usr/local** directory using the command below.

```
$ sudo tar -C /usr/local -xvzf go1.15.2.linux-amd64.tar.gz
```

Where, -C specifies the destination directory..

Configuring GoLang Environment in Linux

4. First, set up your **Go workspace** by creating a directory ~/go_projects which is the root of your workspace. The workspace is made of three directories namely:

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

Therefore create the above directory tree as follows:

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

5. Now it's time to execute **Go** like the rest of Linux programs without specifying its absolute path, its installation directory must be stored as one of the values of $PATH environment variable.
Now, add /usr/local/go/bin to the **PATH** environment variable by inserting the line below in your **/etc/profile** file for a system-wide installation or **$HOME/.profile** or **$HOME./bash_profile** for user-specific installation:
Using your preferred editor, open the appropriate user profile file as per your distribution and add the line below, save the file, and exit:

```
export  PATH=$PATH:/usr/local/go/bin
```

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

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

Note: If you have installed **GoLang** in a custom directory other than the default (**/usr/local/**), you must specify that directory as the value of the **GOROOT** variable.
For instance, if you have installed **GoLang** in the home directory, add the lines below to your **$HOME/.profile** or **$HOME/.bash_profile** file.

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

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

```
$ source ~/.bash_profile
OR
$ source ~/.profile
```

Verify GoLang Installation
8. Run the commands below to view your **Go** version and environment:

```
$ go version
$ go env
```

```
aaronkilik@tecmint ~ $ go version
go version go1.7.3 linux/amd64
aaronkilik@tecmint ~ $
aaronkilik@tecmint ~ $ go env
GOARCH="amd64"
GOBIN="/home/aaronkilik/go_projects/bin"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/aaronkilik/go_projects"
GORACE=""
GOROOT="/usr/local/go"
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=
/tmp/go-build506710615=/tmp/go-build -gno-record-gcc-switches"
CXX="g++"
CGO_ENABLED="1"
aaronkilik@tecmint ~ $
```

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

```
$ go help
```

9. To test if your **Go** installation is working correctly, write a small **Go hello** world program, save the file in **~/go_projects/src/hello/** directory. All your GoLang source files must end with the .go extension.
Begin by creating the hello project directory under **~/go_projects/src/**:

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

Then use your suitable editor to create the hello.go file:

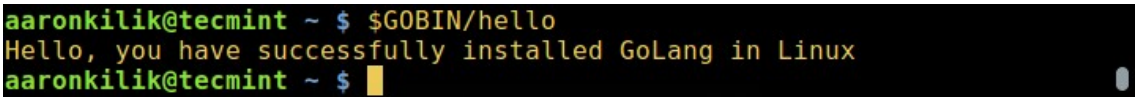
```
$ vi ~/go_projects/src/hello/hello.go
```

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

```
package main
import "fmt"
func main() {
    fmt.Printf("Hello, you have successfully installed GoLang in Linux\n")
}
```

10. Now, compile the program above as using **go** install and run it:

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



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

11. To run your **Go** binary executable like other Linux commands, add **$GOBIN** to your **$PATH** environment variable.

Assignment 1: Introduction to GO Programming

Objectives

Student will be able to

- Know GO Language Program Structure and Compilations.
- Get familiar with GO Language Programming Methods.
- Get familiar with GO Routines, Keywords, Identifiers, Operators, Assignments, Pointers.
- Know how to do Data Modelling with Go.

Reading

You should read the following topics before starting this exercise

- **Go Language Programme Structure**
- **Operators in GO Language**
- **Strings**
- **Pointers and address**
- **If-else and switch statement**
- **For Loop**
- **Break and continue statement**

Ready Reference

- **Go was designed at Google in 2007.** The main objective of GO language is to improve programming productivity in an era of multicore, networked machines and large codebases. Go is a statically typed, compiled programming language designed at Google by Robert Griesemer (Swiss computer scientist), Rob Pike (Canadian Programr), and Ken Thompson (American pioneer of computer science).

  GO Language Programme Structure

Key Elements	Explanation
Package Declaration	package name in which programme is written
Import Packages	Importing pre-processor directives which requires packages and functions from packages to be used
Functions Variables	main function from which programmes execution starts
Statements and Expressions	Included in { }
Comments.	Represented by /*……..*/

For example
Hello world GO Programme

package main
import "fmt"
// this is a comment
 	
func main() {
	← **Package Declaration**

	← Import Packages
	← Comments
	← main function from which programme execution starts. All functions start with the keyword func followed by the

fmt.Println("Hello World") }	name of the function. a list of zero or more “parameters” surrounded by parentheses, an optional return type and a “body” which is surrounded by curly braces. This function has no parameters, doesn't return anything and has only one statement. The name main is special because it's the function that gets called when you execute the program.
Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.	← Println function used for output(Print Line).

For any function help use following command.
godoc fmt Println

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

identifier = letter { letter | unicode_digit }.

Punctuation characters such as @, $, and % are not allowed within identifiers.

Identifier	Valid/Invalid
mahesh	Valid
Movie_name	Valid
movie@name	Invalid
$abc	Invalid
_abc	Valid

List of Keywords in GO Programming Language: Number of keywords : 25

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

Constants: They are like variables with const keyword.
Constants can only be character, string, Boolean, or numeric values and cannot be declared using the:= syntax. An untyped constant takes the type needed by its context.
For example

```
package main
import "fmt"
const (
PI    = 3.142
```

```
A    = true
Large   = 30 << 34
Small   = Large >> 60
)
func main()  {
const Greeting = "Good Morning"
fmt.Println(Greeting)
fmt.Println(PI)
fmt.Println(A)
fmt.Println(Large)
fmt.Println(Small)
}
```

Output: Good Morning
3.142
true
515396075520
0

Variables

The var statement declares a list of variables with the type declared last. For example, User can declare variables in a group or one by one also.

```
var
{
srno int
sname, saddress string
}

var srnoint
```

For Example 1:

```
var sname string="Kalyani"
```

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

```
func main() {
sname,saddress:= "Kalyani","Camp"
srno:=101
}
```

Print or Println will be used to print constants and variables.

fmt.Println prints the passed in variables' values and appends a newline.

fmt.Printf is used when you want to print one or multiple values using a defined format specifier.

For Example:

```
package main
import "fmt"
func main()
{
    name:= "Number Ten"
    alis:= fmt.Sprintf("Number %d", 10)
    fmt.Printf("%s is also known as %s",name, alis)
}
```

The fmt.Sprintf() function in Go language formats according to a format specifier and returns the resulting string

Output: Number Ten is also known as Number 6

Multiple variables can be defined as follows:

```
var(
```

```
x=15
y=1
z85
)
```

Operators

Operators behaves same as in C Programming

Operator	Description
Arithmetic Operators	
+	Adds two operands
-	Subtracts second operand from the first
*	Multiplies both operands
/	Divides the numerator by the denominator.
%	Modulus operator; gives the remainder after an integer division.
++	Increment operator. It increases the integer value by one.
--	Decrement operator. It decreases the integer value by one.
Relational Operators	
==	**Equals:** It checks whether the values of two operands are equal or not; if yes, then the condition becomes true.
!=	**Not Equal to:** It checks if the values of two operands are equal or not; if the values are not equal, then the condition becomes true.
>	**Greater Than:** It checks if the value of left operand is greater than the value of right operand; if yes, the condition becomes true.
<	**Less Than:** It checks if the value of left operand is less than the value of the right operand; if yes, the condition becomes true.
>=	**Greater than equal to:** It checks if the value of the left operand is greater than or equal to the value of the right operand; if yes, the condition becomes true.
<=	**Less than equal to:** It checks if the value of left operand is less than or equal to the value of right operand; if yes, the condition becomes true.
Logical Operators	
□□	Logical AND operator. If both the operands are non-zero, then condition becomes true.
||	Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
!	Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.
Bitwise Operators	
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.

>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
Assignment Operators	
=	Simple assignment operator, Assigns values from right side operands to left side operand.
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.
<<=	Left shift AND assignment operator.
>>=	Right shift AND assignment operator.
&=	Bitwise AND assignment operator.
^=	bitwise exclusive OR and assignment operator.
|=	bitwise inclusive OR and assignment operator.
Miscellaneous Operators	
&	Returns the address of a variable.

Boolean:

The *boolean* data type can have value either true or false, and is defined as bool when declaring it as a data type. Booleans are used to represent the truth values that are associated with the logic branch of mathematics, which informs algorithms in computer science.

Example, user can store Boolean value in a variable.

K:=51>18

User can print the Boolean value with a call to the fmt.Println() function.

fmt. Println(K)

as 15 is not greater than 18 we receive following output.

Output:

True

Numeric

Type	Explanation	Example
Integers	Integers in programming language are whole numbers i.e positive, negative or zero. In GO Language integers are known as int.	var n int =4
Float	Float is used to represent floating point number i.e. real number which are not expressed as real	var n float =4.2

	numbers. Real numbers include all rational and irrational numbers, and because of this, floating-point numbers can contain a fractional part, such as 8.0 or -4266.25. Float number contains decimal point. With integers and floating-point numbers, it is important to keep in mind that 3 ≠ 3.0, as 3 refers to an integer while 3.0 refers to a float.	
Characters	Golang does not have any data type of '**char**'. Therefore following datatypes were supported by go language	
byte	is used to represent the ASCII character. byte is an alias for uint8, hence is of 8 bits or 1 byte and can represent all ASCII characters from 0 to 255.	A byte representing the character '**a**'
rune	is used to represent all UNICODE characters which include every character that exists. rune is an alias for **int32** and can represent all UNICODE characters. It is 4 bytes in size.	Represented by '**£**'
string	A **string** of one length can also be used to represent a character implicitly. The size of one character string will depend upon the encoding of that character. For utf-8 encoding, it will be between 1-4 bytes	A string having one character micro sign '**µ**'

Pointers and Address:

Pointer is a variable which stores memory address of another variable.

Syntax:

var var_name*var_type

example

var ip *int /* pointer to an integer */

var fp *float32 /* pointer to a float */

Pointer to Pointer:

User can form chain of pointers using pointers to pointer concept. When user declare a pointer variable it contains address of local variable. When pointer to pointer is defined it contained address of pointer variable. Following figure illustrates this concept. Additional asterisk will be used to declare pointer to pointer variable.

Syntax: var ptr **int ;

New function:

Another way of accessing pointers in GO using built-in new function. new takes a type as an argument, allocates enough memory to fit a value of that type, and returns a pointer to it.

For example

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

Output:

1

Strings

A string is a sequence of characters with a definite length used to represent text. Go strings are made up of individual bytes, usually one for each character.

- The GO programming language provide various libraries to handle the string operations:
  1. unicode
  2. regexp
  3. strings

String creation:

Syntax: var string name= "The string"

examplevar s1="Welcome"

A string can hold arbitrary bytes. A string literal holds a valid UTF-8 sequence called runes.

Following table illustrates string functions

Function name	Use
p("Contains:", s.Contains("test", "es"))	This function checks whether second string is substring of first string or not. On success: it returns true. On failure: it returns false.
p("Count:", s.Count("test", "t"))	This function checks number of times the character appears in the string.
p("HasPrefix:", s.HasPrefix("test", "te"))	This function checks whether specified prefix exists in the given string or not. If exists it returns true otherwise false.
p("HasSuffix:", s.HasSuffix("test", "st"))	This function checks whether specified suffix exists in the

	given string or not. If exists it returns true otherwise false.
p("Index:", s.Index("test", "e"))	This function will return the position (of index) given character in the first string.
p("Join:", s.Join([]string{"a", "b"}, "-"))	This function will join two strings with given character.
p("Repeat:", s.Repeat("a", 5))	This function will repeat number of times the given string
p("Replace:", s.Replace("foo", "o", "0", -1))	This function will replace the given character by another character.
p("Split:", s.Split("a-b-c-d-e", "-"))	This function will split the given string into different characters.
p("ToLower:", s.ToLower("TEST"))	This function will convert given string in lower case.
p("ToUpper:", s.ToUpper("test"))	This function will convert given string in upper case.
p("Len:", len("hello"))	This function will return length of the string.
p("Char:", "hello"[1])	This function will print the ASCII value of given character of string given in subscript.

If else and switch

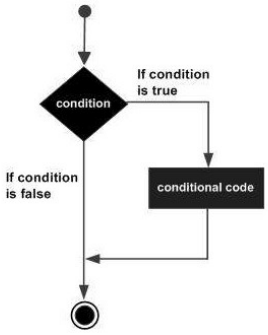
Following types of decision-making statements are provided by GO programming language.

Sr. No.	Statement and Description
1.	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2.	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
3.	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4.	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5.	select statement A **select** statement is similar to **switch** statement with difference that case statements refers to channel communications.

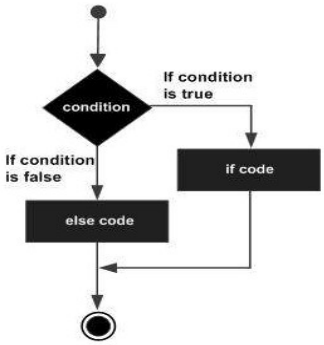
if statement:

Syntax:

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```



if else statement:



If..elseif..if..else statement

Syntax:

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
} else if( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
} else if( boolean_expression 3)
{
   /* Executes when the boolean expression 3 is true */
} else
{
   /* executes when the none of the above condition is true */
}
```

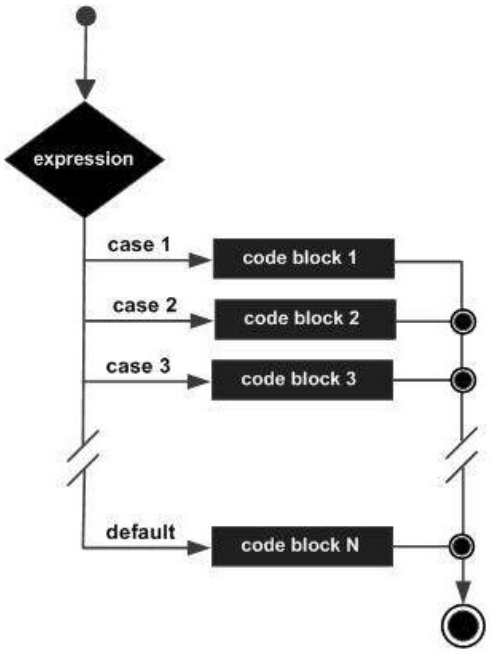
Switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.In Go programming, switch statements are of two types:

- **Expression Switch:** In expression switch, a case contains expressions, which is compared against the value of the switch expression.
- **Type Switch:** In type switch, a case contains type which is compared against the type of a specially annotated switch expression.

Syntax:

```
switch(boolean-expression or integral type)
{
  case boolean-expression or integral type:
    statement(s);
  case boolean-expression or integral type:
    statement(s);
    /* Number of case statements */
    default: /* Optional */
    statement(s);
}
```



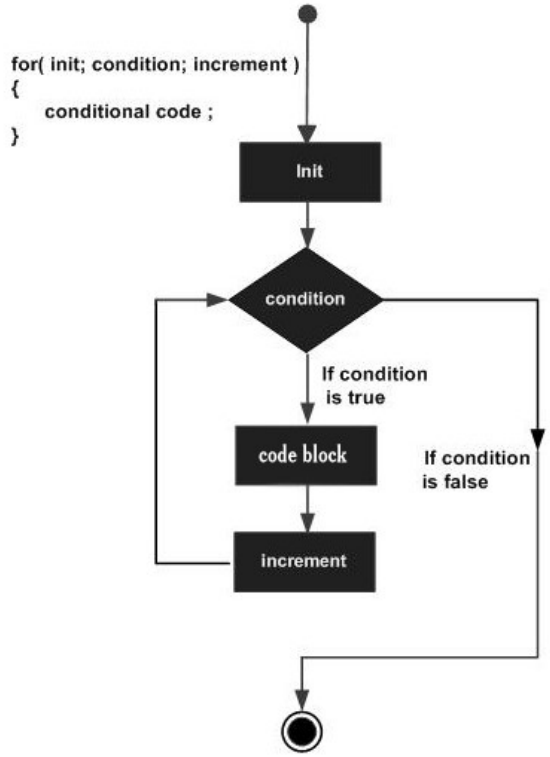
- For loop For loop is used to run Go language script n number of times.

  Syntax:

```
for [condition |  (init; condition; increment ) | Range]
{
  statement(s);
}
```

- If a **condition** is available, then for loop executes as long as condition is true.

- If **range** is available, then the for loop executes for each item in the range.



Example to print a raise to b

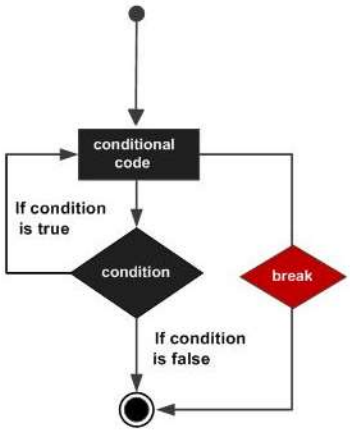
```
package main
import "fmt"
func main()
{
    var   a, b, z   int
    z=1
    fmt.Printf("\nEnter base and power:")
    fmt.Scanf("%d %d ",&a,&b)
    for i:= 1; i<= b; i++
    {
        z= z*a
    }
    fmt.Printf("%d Raise to %d = %d \n", a,b,z)
}
```

Output:

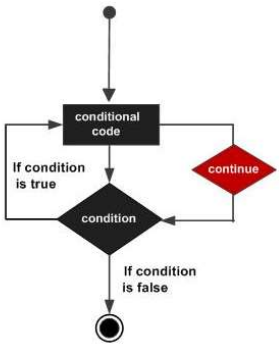
```
Enter base and power:3 4
3 Raise to  4 = 81
```

Using break and continue

Break statement: This statement terminates a for loop or switch statement and transfers execution to the statement immediately following the for loop or switch.



Continue: This statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. i.e. Instead of forcing termination, a continue statement forces the next iteration of the loop to take place, skipping any code in between. In case of the for loop, continue statement causes the conditional test and increment portions of the loop to execute.



Goto statement:

- This statement is used for unconditional jumping from goto statement to the label provided. Use of this statement is highly discouraged as it becomes difficult to manage iterations while using goto statement and hence program flow will loose.

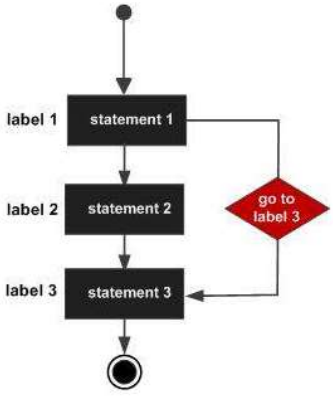
  Syntax:

  goto label;

  …

  ….

  label: statement;



Example to represent byte, rune, and string

```
package main
import
{
   "fmt"
   "reflect"
   "unsafe"
}
func main()
{
    var b byte = 'a'
    fmt.Println("Printing Byte:")
    //Print Size, Type and Character
    fmt.Printf("Size: %d\nType: %s\nCharacter: %c\n", unsafe.Sizeof(b), reflect.TypeOf(b), b)
    r:= '£'
    fmt.Println("\nPrinting Rune:")
    //Print Size, Type, CodePoint and Character
    fmt.Printf("Size: %d\nType: %s\nUnicodeCodePoint: %U\nCharacter: %c\n", unsafe.Sizeof(r),
    reflect.TypeOf(r), r, r)
    s:= "µ" //Micro sign
    fmt.Println("\nPrinting String:")
    fmt.Printf("Size: %d\nType: %s\nCharacter: %s\n", unsafe.Sizeof(s), reflect.TypeOf(s), s)
}
```

Output:

```
Printing Byte:
Size: 1
Type: uint8
Character: a
Printing Rune:
Size: 4
Type: int32
Unicode CodePoint: U+00A3
Character: £
Printing String:
Size: 16
Type: string
Character: µ
```

Lab Assignments:

SET A

1. WAP in go language to print Student name, rollno, division and college name
2. WAP in go language to print whether number is even or odd.
3. WAP in go language to swap the number without temporary variable.
4. WAP in go Language to print address of a variable.

SET B

1. WAP in go to print table of given number.
2. WAP in go language to print PASCALS triangle.
3. WAP in go language to print Fibonacci series of n terms.
4. WAP in go language to illustrate pointer to pointer concept.
5. WAP in go language to explain new function

SET C

1. WAP in go language to concatenate two strings using pointers.
2. WAP in go language to accept two strings and compare them.
3. WAP in go language to accept user choice and print answer of using arithmetical operators.
4. WAP in go language to check whether accepted number is single digit or not.
5. WAP in go language to check whether first string is substring of another string or not.

Signature of the Instructor: **Date:**

Assignment Evaluation :

0: Not Done		2: Late Complete		4: Complete	
1: Complete		3: Needs Improvement		5: Well Done	

Assignment 2: Functions

Objectives

Student will be able

- To learn the syntax and semantics of the functions in GO programming language
- To understand how to access and use library functions in Go Language
- To understand proper use of user defined functions in Go Language
- To understand how to write functions and pass arguments in GO Language

Reading

You should read the following topics before starting this exercise

- Functions and its types
- Call by value and call by reference
- Named return variables
- Multiple return values from function

Ready Reference

Functions are the building blocks of a Go program. The general form of function definition is as follows:

Functions are the building blocks of a Go program. The general form of function definition is as follows:

```
func function_name( [parameter list] ) [return_types]
{
   body of the function
}
```

- **func:** It is a keyword in Go language, which is used to create a function. It starts function declaration.
- **function_name:** It is actual name of function. The function name and the parameter list together constitute the function signature.
- **Parameter's list:** This list contains name and type of the function parameters with order. Parameters are also optional, a function may or may not contain parameters.
- **Return_type:** A function may or may not return a value. So, return type is optional. It contains the types of the values that function returns.
- **Function_body:** It contains a collection of statements that define what the function does.
- **Example:** A function prints addition of two numbers:

```
func abc(n1, n2 int) int
{
    var r int// local variables in function
    r = n1 + n2
    return r //returning value
}
```

Formal Parameters:

These parameters are representing actual parameters and actual parameters are accepted in these parameters when writing function definition.

Actual Parameters:

These parameters are passed to the function from where it has been called as an argument.The numbers of parameters, types of parameters and sequence of parameters should be matched. If there is any mismatch then function shows an error.

Function returning multiple values:

A Go function can return multiple values. Go supports this feature internally. The syntax of the multi-return function is shown below:

```
func funcName(p1 paramType1, p2 paramType2, ...) (returnType1, returnType2, ..., returnTypeN) {}
```

Declaration of return types:

- There are multiple ways to declare return types and values. Following example shows how to return single value.

```
package main
import    "fmt"
func add(q int) (int)
{
   return q + 2
}
func main()
{
   fmt.Println(mult(2))  // prints 4
}
```

Declaration of the same return types:

- We can use same-typed return values. Consider the following example.

```
package main
import "fmt"
func add(a int) (int, int)
{
    return a/2, a*2  //mult function returns two values
}
func main()
{
    x, y := mult(16)
    fmt.Println(x, y)  // output will be 8 32
}
```

Declaration of the different return types

- We can use different-typed return values.

For Example:

```
package main
import "fmt"
func mult(a int) (int, float64)
{
    return a/2, float64(a)*2.05
    // mult function returns two different parameters of different data types.
}
func main()
{
    x, y := mult(12)
    fmt.Println(x, y)  // output 6 24.599999999999998
```

- Go has excellent error handling support for multiple return values. We simply can return an error when doing any operation. Then check the errors sequentially as shown below.

```
package main
import ("fmt" "errors")
func f(a int) (int, error)
{
    if a == 0
    {
        return 0, errors.New("Zero not allowed")
    }
    else
    {
        return a * 2, nil
    }
}
func main()
{
    v, e := f(0)
    if e != nil
    {   // check error here
        fmt.Println("Error!!!")
    }
    else
    {
        fmt.Println(v)
    }
    // output:
    // Error!!!
}
```

Errors package in Golang is used to implement the functions to manipulate the errors. errors. New()function returns an error that formats like given text

Call by Value	Call by Reference
The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. By default, Go programming language uses call by value method to pass arguments	The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument. To pass the value by reference, argument pointers are passed to the functions just like any other value.
Example /* function definition to swap the values */ func swap(int x, int y) int { var temp int temp = x /* save the value of x */	Example: /* function definition to swap the values */ func swap(x *int, y *int) { var temp int

```
    x = y    /* put y into x */
    y = temp /* put temp into y */
    return temp;
}
```

```
    temp = *x      /* save the value at address x */
    *x = *y      /* put y into x */
    *y = temp    /* put temp into y */
}
```

Named return variables

Named return types are a feature in Go where user can define the names of the return values. The benefit of using the named return types is that it allows us to not explicitly return each one of the values. It will be helpful for complicated calculations where multiple values are calculated in many different steps. Consider the following example.

```
package main
import "fmt"
func try(b int) (x, y int)
{
    x = b+4
    y = b-4
    return          // notice that we are not returning any value
}
func main()
{
    x, y := try(12)
    fmt.Println(x, y)    // output will 16 8
}
```

Ignore Compiler Errors

The blank identifier can be used as a placeholder where the variables are going to be ignored for some purpose. Later those variables can be added by replacing the operator. In many cases, it helps to debug code.

For Example:

```
package main
import   "fmt"
func f() int
{
    return 42
}
func main()
{
    r := f()
    _ = r + 10     // ignore it now can add the variable later
    fmt.Println(r) // 42
```

multiple return values from function

Go has excellent error handling support for multiple return values. We simply can return an error when doing any operation. Then check the errors sequentially as shown below.

```
package main
import ("fmt" "errors")
func f(a int) (int, error)
```

```
{
    if a == 0
    {
        return 0, errors.New("Zero not allowed")
    }
    else
    {
        return a * 2, nil
    }
}
func main()
{
    v, e := f(0)
    if e != nil
    {   // check error here
        fmt.Println("Error!!!")
    }
    else
    {
        fmt.Println(v)
    }
    // output:
    // Error!!!
}
```

Recursion

- Recursion is the process in which function calls itself directly or indirectly. For example

Direct recursion	Indirect recursion

func abc()
{
abc()/* function calls itself*/
}
func main()
{
abc()
}
	func abc()
	{
	pqr()/* function abc calls calls function pqr*/
	}
	func pqr()
	{
	abc()/* function pqr calls function abc*/
	}
	func main()
	{
	abc()
	}

Function calls itself.	One function calls another function and another function again calls the same function.

Example to print factorial

```
func fact(i int)int
{
    if(i<= 1)
    {
```

```
            return 1
        }
        return i * fact(i - 1)
    }
```

Example of factorial calculations: for number 5

1! = 1		1! = 1
2! = 2 × 1 = 2		2! = 2 × 1! = 2
3! = 3 × 2 × 1 = 6	OR	3! = 3 × 2! = 6
4! = 4 × 3 × 2 × 1 = 24		4! = 4 × 3! = 24
5! = 5 × 4 × 3 × 2 × 1 = 120		5! = 5 × 4! = 120

Lab Assignments:

SET A

1. WAP in go language to print addition of two number using function.
2. WAP in go language to print recursive sum of digits of given number.
3. WAP in go language using function to check whether accepts number is palindrome or not.

SET B

1. WAP in go language to swap two numbers using call by reference concept.
2. WAP in go language to demonstrate use of names returns variables.
3. WAP in go language to show the compiler throws an error if a variable is declared but not used.

SET C

1. WAP in go language to illustrate the concept of call by value.
2. WAP in go language to create a file and write hello world in it and close the file by using defer statement.
3. WAP in go language to illustrate the concept of returning multiple values from a function

Signature of the Instructor: Date:

Assignment Evaluation :

0: Not Done		2: Late Complete		4: Complete	
1: Complete		3: Needs Improvement		5: Well Done	

Assignment 3: Working with Data

Objectives

- To understand Array Literals in Go Language
- To work with Slices and Slice parameters
- To understand the use of Structure Parameters

Reading

- Array Literals
- Multidimensional Array
- Slices and Slice parameters
- Multidimensional slices
- Structures and Structure Parameters

Read from: http://www.golangbootcamp.com/book

Ready Reference

Arrays

Arrays are one of the most popular data structures for two main reasons: arrays are simple to use and easy to understand and they are very flexible and can store many different kinds of data.
You can declare an array that stores four integers as follows:

```
anArray := [4]int{1, 2, 4, -4}
```

The size of the array is stated before its type, which is defined before hand. The length of an array can be found with the help of the len() function: len(anArray).

The index of the first element of any dimension of an array is 0, the index of the second element of any array dimension is 1, and so on. For an array with one dimension named a, the valid indexes are from 0 to len(a)-1.

To visit all of the elements of an array in Go that involve the use of the range keyword that allow you to bypass the use of the len() function in the for loop.

Multi-dimensional arrays

Arrays can have more than one or two dimensions. However, using more than three dimensions without a valid reason can make your program difficult to read and might create errors. Arrays can store all the types of elements. We are just using integers here because they are easier to understand and type.

The following Go code shows how you can create an array with two dimensions (twoDArray) and another one with three dimensions (threeDArray):

```
twoDArray := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}

threeDArray := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Accessing, assigning or printing a single element from one of the previous two arrays can be done easily. As an example, the first element of the twoDArray array is twoDArray[0][0] and its value is 1.

Therefore, accessing all of the elements of the twoDArray, array with the help of multiple for loops can be done as follows:

```
for i := 0; i < len(threeDArray); i++ {
for j := 0; j < len(v); j++ {
for k := 0; k < len(m); k++ {
}
}
}
```

As we know that we need as many for loops as the dimensions of the array in order to access all of its elements. The same rules apply to slices.

Following example shows how to use a simple array

```
package main

import "fmt"

func main() {
        var theArray [3]string
        theArray[0] = "India"  // Assign a value to the first element
        theArray[1] = "Canada" // Assign a value to the second element
        theArray[2] = "USA"  // Assign a value to the third element

        fmt.Println(theArray[0]) // Access the first element value
        fmt.Println(theArray[1]) // Access the second element valu
        fmt.Println(theArray[2]) // Access the third element valu
}
```

Output

```
India
Canada
USA
```

Initializing an Array with an Array Literal

You can initialize an array with pre-defined values using an array literal. An array literal has the number of elements it will hold in square brackets, followed by the type of its elements. This is followed by a list of initial values separated by commas of each element inside the curly braces.

Example

```
package main
```

```
import "fmt"

func main() {
	x := [5]int{10, 20, 30, 40, 50}   // Intialized with values
	var y [5]int = [5]int{10, 20, 30} // Partial assignment

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

Output

```
[10 20 30 40 50]
[10 20 30 0 0]
```

Initializing an Array with ellipses

When we use ... instead of specifying the length. The compiler can calculate the length of an array, based on the elements specified in the array declaration.

Example

```
package main

import (
	"fmt"
	"reflect"
)

func main() {
	x := [...]int{10, 20, 30}

	fmt.Println(reflect.ValueOf(x).Kind())
	fmt.Println(len(x))
}
```

Output

```
array
3
```

Copy an Array

You can create copy of an array, by assigning an array to a new variable either by value or reference.

Example

```
package main

import "fmt"

func main() {
```

```
        strArray1 := [3]string{"India", "Australia", "Germany"}
        strArray2 := strArray1  // data is passed by value
        strArray3 := &strArray1 // data is passed by reference

    fmt.Printf("strArray1: %v\n", strArray1)
    fmt.Printf("strArray2: %v\n", strArray2)

    strArray1[0] = "Canada"

        fmt.Printf("strArray1: %v\n", strArray1)
        fmt.Printf("strArray2: %v\n", strArray2)
        fmt.Printf("*strArray3: %v\n", *strArray3)
}
```

Output

```
strArray1: [India Australia Germany]
strArray2: [India Australia Germany]
strArray1: [Canada Australia Germany]
strArray2: [India Australia Germany]
*strArray3: [Canada Australia Germany]
```

Check if Element Exists

To determine if a specific element exist in an array, we need to iterate each array element using for loop and check using if condition.

Example

```
package main

import (
        "fmt"
        "reflect"
)

func main() {
        strArray := [5]string{"India", "Canada", "Japan", "Germany", "Italy"}
        fmt.Println(itemExists(strArray, "India"))
        fmt.Println(itemExists(strArray, "USA"))
}

func itemExists(arrayType interface{}, item interface{}) bool {
        arr := reflect.ValueOf(arrayType)

        if arr.Kind() != reflect.Array {
                panic("Invalid data-type")
        }

        for i := 0; i < arr.Len(); i++ {
                if arr.Index(i).Interface() == item {
                        return true
                }
        }
```

```
        return false
}
```

Output

```
true
false
```

The disadvantages of Go arrays

Go arrays have certain disadvantages. First of all, once you define an array, you cannot change its size, which means that Go arrays are not dynamic. If you need to add an element to an existing array that has no space left, you will need to create a bigger array and copy all of the elements of the old array to the new one.

Secondly, when you pass an array to a function as a parameter, you actually pass a copy of the array, which means that any changes you make to an array inside a function will be lost after the function exits. Lastly, passing a large array to a function can be very slow, mostly because Go has to create a copy of the array. The solution to all of these problems is the use of **Go slices**.

Go slices

A slice is a flexible and extensible data structure to implement and manage collections of data. Slices are made up of multiple elements which are all of the same type. A slice is a segment of dynamic arrays that can grow and shrink as required. Like arrays, slices are index-able and have a length. Slices have a capacity and use length property. Go **slices** are very powerful. There are only a few times where you will need to use an array instead of a slice. The most obvious one is when we are very sure that you will need to store a fixed number of elements.

Slices are implemented using arrays internally, which means that Go uses an underlying array for each slice. Slices are passed by reference to functions, which means that what is actually passed is the memory address of the slice variable, any modifications that you make to a slice inside a function will not get lost after the function exits. Additionally, passing a big slice to a function is significantly faster than passing an array with the same number of elements because Go will not have to make a copy of the slice, it will just pass the memory address of the slice variable.

Performing basic operations on slices
Creation of a new slice literal is as follows:

```
aSliceLiteral := []int{1, 2, 3, 4, 5}
```

Slice literals are defined just like arrays but without the element count. If we put an element count in a definition, we will get an array instead.

The make() function allows us to create empty slices with the desired length and capacity as the parameters passed to make(). The capacity parameter is optional, in this case, the capacity of the slice will be the same as its length.

You can define a new empty slice with 20 places that can be automatically expanded when needed as follows:

integer := make([]int, 20)

Go automatically initializes the elements of an empty slice to the zero value of its type, which means that the value of the initialization depends on the type of the object stored in the slice.

After this, you can access all of the elements of a slice in the following way:

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

If we want to empty an existing slice, the zero value for a slice variable is nil.
aSliceLiteral = nil

We can add an element to the slice, which will automatically increase its size, using the append() function:

```
integer = append(integer, -5000)
```

We can access the first element of the integer slice as integer[0], whereas the last element of the integer slice as integer[len(integer)-1].

Also, you can access multiple continuous slice elements using the [:] notation. The next statement selects the second and the third elements of a slice:

```
integer[1:3]
```

Furthermore, we can use [:] notation for creating a new slice from an existing slice or array:

```
s2 := integer[1:3]
```

Please note that this process is called **re-slicing**, and it may cause some errors

Create Empty Slice

To declare the type for a variable that holds a slice, use an empty pair of square brackets, followed by the type of elements the slice will hold.

Example

```
package main

import (
        "fmt"
        "reflect"
)

func main() {
        var intSlice []int
        var strSlice []string
```

```
	fmt.Println(reflect.ValueOf(intSlice).Kind())
	fmt.Println(reflect.ValueOf(strSlice).Kind())
}
```

Output

```
slice
slice
```

Declare Slice using Make

Slice can be created using the built-in function make. When you use make, one option you have is to specify the length of the slice. When you just specify the length, the capacity of the slice is the same.

Example

```
package main

import (
	"fmt"
	"reflect"
)

func main() {
	var intSlice = make([]int, 10)		// when length and capacity is same
	var strSlice = make([]string, 10, 20) // when length and capacity is different

	fmt.Printf("intSlice \tLen: %v \tCap: %v\n", len(intSlice), cap(intSlice))
	fmt.Println(reflect.ValueOf(intSlice).Kind())

	fmt.Printf("strSlice \tLen: %v \tCap: %v\n", len(strSlice), cap(strSlice))
	fmt.Println(reflect.ValueOf(strSlice).Kind())
}
```

Output

```
intSlice	Len: 10	Cap: 10
slice
strSlice	Len: 10	Cap: 20
slice
```

Initialize Slice with values using a Slice Literal

A slice literal contain empty brackets followed by the type of elements the slice will hold, and a list of the initial values each element will have in curly braces.

Example

```
package main

import "fmt"
```

```
func main() {
	var intSlice = []int{10, 20, 30, 40}
	var strSlice = []string{"India", "Canada", "Japan"}

	fmt.Printf("intSlice \tLen: %v \tCap: %v\n", len(intSlice), cap(intSlice))
	fmt.Printf("strSlice \tLen: %v \tCap: %v\n", len(strSlice), cap(strSlice))
}
```

Declare Slice using new Keyword

A slice can be declared using new keyword followed by capacity in square brackets then type of elements the slice will hold.

Example

```
package main

import (
	"fmt"
	"reflect"
)

func main() {
	var intSlice = new([50]int)[0:10]

	fmt.Println(reflect.ValueOf(intSlice).Kind())
	fmt.Printf("intSlice \tLen: %v \tCap: %v\n", len(intSlice), cap(intSlice))
	fmt.Println(intSlice)
}
```

Output

```
slice
intSlice	Len: 10	Cap: 50
[0 0 0 0 0 0 0 0 0 0]
```

Add Items

To add an item to the end of the slice, use the append() method.

Example

```
package main

import "fmt"

func main() {
	a := make([]int, 2, 5)
	a[0] = 10
	a[1] = 20
	fmt.Println("Slice A:", a)
	fmt.Printf("Length is %d Capacity is %d\n", len(a), cap(a))
```

```
        a = append(a, 30, 40, 50, 60, 70, 80, 90)
        fmt.Println("Slice A after appending data:", a)
        fmt.Printf("Length is %d Capacity is %d\n", len(a), cap(a))
}
```

Output

```
Slice A: [10 20]
Length is 2 Capacity is 5
Slice A after appending data: [10 20 30 40 50 60 70 80 90]
Length is 9 Capacity is 12
```

If there is sufficient capacity in the underlying slice, the element is placed after the last element and the length gets incremented. However, if there isn't a sufficient capacity, a new slice is created, all of the existing elements are copied over, the new element is added onto the end, and the new slice is returned.

Please note that the re-slice process does not make a copy of the original slice. Another problem of re-slicing is that, even if you re-slice a slice in order to use a small part of the original slice, the underlying array from the original slice will be kept in memory for as long as the smaller re-slice exists because the original slice is being referenced by the smaller re-slice. Although this is not truly important for small slices, it can cause problems when you are reading big files into slices and you only want to use a small part of them.

Slices are being expanded automatically

Slices have two main properties: **capacity** and **length**. The complex part is that usually these two properties have different values. The length of a slice is the same as the length of an array with the same number of elements and can be found using the len() function. The capacity of a slice is the current room that has been allocated for this particular slice, and it can be found with the cap() function. As slices are dynamic in size, if a slice runs out of room,

Go automatically doubles its current length to make room for more elements. If the length and the capacity of a slice have the same values and we try to add another element to the slice, the capacity of the slice will be doubled, whereas its length will be increased by one. Although this might work well for small slices, adding a single element to a really huge slice might take more memory than expected.

Multidimensional slices

Slices can have many dimensions as is also the case with arrays. The next statement creates a slice with two dimensions:

```
s1 := make([][]int, 4)
```

If you find yourselves using slices with many dimensions all of the time, you might need to reconsider your approach and choose a simpler design that does not require multidimensional slices.

Structures and Structure Parameters

To define a structure, you must use **type** and **struct** statements. The struct statement defines a new data type, with multiple members for your program. The type statement binds a name with the type which is struct in our case. The format of the struct statement is as follows −

```
type struct_variable_type struct {
  member definition;
  member definition;
```

```
   ...
   member definition;
}
```

Once a structure type is defined, it can be used to declare variables of that type using the following syntax.

variable_name := structure_variable_type {value1, value2...valuen}

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)** The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. The following example explains how to use a structure −

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books    /* Declare Book1 of type Book */
   var Book2 Books    /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "ABC"
   Book1.subject = "Go Programming Tutorial"
   Book1.book_id = 6495407

   /* book 2 specification */
   Book2.title = "Java Programming"
   Book2.author = "Herbert Schildt"
   Book2.subject = "Java Programming Tutorial"
   Book2.book_id = 6495700

   /* print Book1 info */
   fmt.Printf( "Book 1 title : %s\n", Book1.title)
   fmt.Printf( "Book 1 author : %s\n", Book1.author)
   fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
   fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)

   /* print Book2 info */
   fmt.Printf( "Book 2 title : %s\n", Book2.title)
   fmt.Printf( "Book 2 author : %s\n", Book2.author)
   fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
   fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
```

```
}
```

When the above code is compiled and executed, it produces the following result :

```
Book 1 title      : Go Programming
Book 1 author     : ABC
Book 1 subject    : Go Programming Tutorial
Book 1 book_id    : 6495407
Book 2 title      : Java Programming
Book 2 author     : Herbert Schildt
Book 2 subject    : Java Programming Tutorial
Book 2 book_id    : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the same way as you did in the above example −

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books    /* Declare Book1 of type Book */
   var Book2 Books    /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "ABC"
   Book1.subject = "Go Programming Tutorial"
   Book1.book_id = 6495407

   /* book 2 specification */
   Book2.title = "Java Programming"
   Book2.author = "Herbert Schildt"
   Book2.subject = "Java Programming Tutorial"
   Book2.book_id = 6495700

   /* print Book1 info */
   printBook(Book1)

   /* print Book2 info */
   printBook(Book2)
}
func printBook( book Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
```

```
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title     : Go Programming
Book author    : ABC
Book subject   : Go Programming Tutorial
Book book_id   : 6495407
Book title     : Java Programming
Book author    : Herbert Schildt
Book subject   : Java Programming Tutorial
Book book_id   : 6495700
```

Lab Assignments

SET A

1. WAP in go language to find the largest and smallest number in an array.
2. WAP in go language to accept the book details such as BookID, Title, Author, Price. Read and display the details of n number of books.
3. WAP in go language to Initialize a Slice using Multi-Line Syntax and display

SET B

1. WAP in go language to create and print multidimensional Slice.
2. WAP in go language to sort array elements in ascending order.
3. WAP in go language to accept n student details like roll_no, stud_name, mark1,mark2, mark3. Calculate the total and average of marks using structure.

SET C

1. WAP in go language to accept two matrices and display it's multiplication.
2. WAP in go language to accept n records of employee information (eno,ename,salary) and display record of employees having maximum salary.
3. WAP in go language to demonstrate working of slices (like append, remove, copy etc.)

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Assignment 4: Methods and Interfaces

Objectives

- To understand the concept of Methods and Interfaces
- To work with Methods and Interfaces

Reading

You should read the following topics before starting this exercise:

- Methods in Go Language
- Interfaces in Go Language

Read: https://go.dev/tour/methods/9

Ready Reference

In Golang, we declare a function **using the func keyword**. A function has a name, a list of comma-separated input parameters along with their types, the result type(s), and a body. The input parameters and return type(s) are optional for a function. A function can be declared without any input and output.

Functions are commonly the block of codes or statements in a program that gives the user the ability to reuse the same code which eventually saves the excessive use of memory, acts as a time saver and more importantly, provides us with better readability of the code. A function is a collection of statements that perform some specific task and return the result to the caller. A function can also perform some specific task without returning anything.

Function Declaration

Function declaration is a way to construct a function.

Syntax:

```
func function_name(Parameter-list)(Return_type){
   // function body.....
}
```

The declaration of the function contains the following:

- **func:** It is a keyword in Go language, which is used to create a function
- **function_name:** It is the name of the function
- **Parameter-list:** It contains the name and the type of the function parameters
- **Return_type:** It is optional and it contains the types of the values that function returns. If you are using return_type in your function, then it is necessary to use a return statement in your function.

Function Invocation or Function Calling is done when the user wants to execute the function. The function needs to be called for the purpose of execution. As shown in the below example, we have a function named area() which accepts two parameters. Now we call this function in the main function by using its name, i.e., area(12, 10) with two parameters.

```
// Go program to illustrate the use of function

package main

import "fmt"



// area() is used to find the

// area of the rectangle

// area() function two parameters,

// i.e, length and width

func area(length, width int)int{

        Ar := length* width

        return Ar

}

// Main function

func main() {

// Display the area of the rectangle

// with method calling

fmt.Printf("Area of rectangle is : %d", area(15, 10))

}
```

Output:

```
Area of rectangle is : 150
```

Method Declarations

Go language supports methods. Go methods are similar to Go function with one difference i.e. the method contains a receiver argument in it. With the help of the receiver argument, the method can access the properties of the receiver. Here, the receiver can be of struct type or non-struct type.

When you create a method in your code the receiver and receiver type must be present in the same package. And you are not allowed to create a method in which the receiver type is already defined in another package including inbuilt type like int, string, etc. If you try to do so, then the compiler will give an error.

Syntax:

```
func(receiver_name Type) method_name(parameter_list)(return_type){
// Code
}
```

Here, the receiver can be accessed within the method.

Method with struct type receiver

In Go language, you are allowed to define a method whose receiver is of a struct type. This receiver is accessible inside the method as shown in the below example:

Example:

```
// Go program to illustrate the method with struct type receiver
package main
import "fmt"
// Author structure
type author struct {
        name    string
        branch string
        particles int
        salary int
}

// Method with a receiver of author type
```

```
func (a author) show() {
        fmt.Println("Author's Name: ", a.name)
        fmt.Println("Branch Name: ", a.branch)
        fmt.Println("Published articles: ", a.particles)
        fmt.Println("Salary: ", a.salary)
}
// Main function
func main() {
        // Initializing the values of the author structure
        res := author{
                name:   "ABC",
                branch: "CSE",
                particles: 203,
                salary: 34000,
        }

        // Calling the method
        res.show()
}
```

Output:

```
Author's Name:  ABC
Branch Name:  CSE
Published articles:  203
Salary:  34000
```

Method with Non-Struct Type Receiver

In Go language, you are allowed to create a method with non-struct type receiver as long as the type and the method definitions are present in the same package. If they are present in different packages like int, string, etc, then the compiler will give an error because they are defined in different packages.

Example:

```
// Go program to illustrate the method with non-struct type receiver
package main
 import "fmt"
 // Type definition
type data int
 // Defining a method with non-struct type receiver
func (d1 data) multiply(d2 data) data {
    return d1 * d2
}

/*
// if you try to run this code,
// then compiler will throw an error
func(d1 int)multiply(d2 int)int{
return d1 * d2
}
*/

// Main function
func main() {
```

```
    value1 := data(25)

    value2 := data(20)

    res := value1.multiply(value2)

    fmt.Println("Final result: ", res)

}
```

Output:
Final result: 500

Methods with Pointer Receiver

In Go language, you are allowed to create a method with a pointer receiver. With the help of a pointer receiver, if a change is made in the method, it will reflect in the caller which is not possible with the value receiver methods.

Syntax:

```
func (p *Type) method_name(...Type) Type {
// Code
}
```

Example:

```
// Go program to illustrate pointer receiver
package main
 import "fmt"
// Author structure
  type author struct {
    name      string
    branch    string
    particles int
}

// Method with a receiver of author type
func (a *author) show(abranch string) {
    (*a).branch = abranch
}

// Main function
func main() {

    // Initializing the values
    // of the author structure
```

```
    res := author{
        name:   "ABC",
        branch: "CSE",
    }

    fmt.Println("Author's name: ", res.name)
    fmt.Println("Branch Name(Before): ", res.branch)

    // Creating a pointer
    p := &res

    // Calling the show method
    p.show("ECE")
    fmt.Println("Author's name: ", res.name)
    fmt.Println("Branch Name(After): ", res.branch)
}
```

Output:

```
Author's name:  ABC
Branch Name(Before):  CSE
Author's name:  ABC
Branch Name(After):  ECE
```

Method Can Accept both Pointer and Value

As we know that in Go, when a function has a value argument, then it will only accept the values of the parameter, and if you try to pass a pointer to a value function, then it will not accept and the reverse. But a Go method can accept both value and pointer, whether it is defined with pointer or value receiver. As shown in the below example:

Example:

```
// Go program to illustrate how the method can accept pointer and value

package main

import "fmt"

// Author structure

type author struct {

    name   string

    branch string

}

// Method with a pointer  receiver of author type

func (a *author) show_1(abranch string) {

    (*a).branch = abranch

}
```

```
// Method with a value receiver of author type
func (a author) show_2() {
    a.name = "ABC"
   fmt.Println("Author's name(Before) : ", a.name)
}

// Main function
func main() {
    // Initializing the values of the author structure
   res := author{
       name:   "XYZ",
       branch: "CSE",
   }

   fmt.Println("Branch Name(Before): ", res.branch)

   // Calling the show_1 method
   // (pointer method) with value
   res.show_1("ECE")
   fmt.Println("Branch Name(After): ", res.branch)
```

```
    // Calling the show_2 method
    // (value method) with a pointer
    (&res).show_2()
    fmt.Println("Author's name(After): ", res.name)
}
```

Output:

```
Branch Name(Before):  CSE
Branch Name(After):  ECE
Author's name(Before) :  ABC
Author's name(After):  XYZ
```

Functions that return multiple values

In Go language, we are allowed to return multiple values from a function, using the return statement. In a function, a single return statement can return multiple values. The type of the return values is similar to the type of the parameter defined in the parameter list.

Syntax:

```
func function_name(parameter_list)(return_type_list){
    // code...
}
```

Here,

function_name: It is the name of the function
parameter-list: It contains the name and the type of the function parameters
return_type_list: It is optional and it contains the types of the values that function returns. If you are using return_type in your function, then it is necessary to use a return statement in your function.

Example:

```
// Go program to illustrate how a function return multiple values
package main
 import "fmt"
 // myfunc return 3 values of int type
func myfunc(p, q int)(int, int, int ){
    return p - q, p * q, p + q
}
 // Main Method
func main() {
   // The return values are assigned into three different variables
   var myvar1, myvar2, myvar3 = myfunc(4, 2)
   // Display the values
   fmt.Printf("Result is: %d", myvar1)
   fmt.Printf("\nResult is: %d", myvar2)
   fmt.Printf("\nResult is: %d", myvar3)
}
```

Output:

```
Result is: 2
Result is: 8
Result is: 6
```

Giving Name to the Return Values

In Go language, you are allowed to provide names to the return values. And you can also use those variable names in your code. It is not necessary to write these names with a return statement because the Go compiler will automatically understand that these variables have to dispatch back.

Syntax:

```
func function_name(para1, para2 int)(name1 int, name2 int){
    // code...
}
```

or

```
func function_name(para1, para2 int)(name1, name2 int){
  // code...
}
```

Here, name1 *and* name2 is the name of the return value and para1 and para2 are the parameters of the function.

Example:

```
// Go program to illustrate how to give names to the return values

package main

import "fmt"

// myfunc return 2 values of int type here, the return value name is rectangle and square

func myfunc(p, q int)( rectangle int, square int ){

    rectangle = p*q

    square = p*p

    return

}

func main() {
```

```
    // The return values are assigned into two different variables

    var area1, area2 = myfunc(2, 4)

    // Display the values

    fmt.Printf("Area of the rectangle is: %d", area1 )

    fmt.Printf("\nArea of the square is: %d", area2)

    }
```

Output:

Area of the rectangle is: 8

Area of the square is: 4

Difference Between Method and Function

Method	Function
It contains a receiver	It does not contain a receiver
Methods of the same name but different types can be defined in the program.	Functions of the same name but different type are not allowed to be defined in the program
It cannot be used as a first-order object	It can be used as first-order objects and can be passed

Pointer and Value Receivers

Function can take pointer parameters provided that their signature allows it. **A function** is declared by specifying the types of the arguments, the return values, and the function body.

Example

```
type Person struct {
   Name string
   Age  int
}func NewPerson(name string, age int) *Person {
 return &Person{
```

```
    Name: name,
    Age:  age,
  }
}
```

A method is just a function with a receiver argument. It is declared with the same syntax with the addition of the **receiver**.

```
func (p *Person) isAdult bool {
  return p.Age > 18
}
```

In the above method declarations, we declared the isAdult method on the *Person type.

Difference between the value Receiver and Pointer receiver.

Value receiver makes a copy of the type and pass it to the function. The function stack now holds an equal object but at a different location on memory. That means any changes done on the passed object will remain local to the method. The original object will remain unchanged.

Pointer receiver passes the address of a type to the function. The function stack has a reference to the original object. So any modifications on the passed object will modify the original object.

Example

```
package main
import (
  "fmt"
)
type Person struct {
    Name string
    Age  int
}
func ValueReceiver(p Person) {
    p.Name = "ABC"
    fmt.Println("Inside ValueReceiver : ", p.Name)
}
func PointerReceiver(p *Person) {
    p.Age = 24
    fmt.Println("Inside PointerReceiver model: ", p.Age)
}
func main() {
```

```
    p := Person{"XYZ", 28}
    p1:= &Person{"LMN", 68}
    ValueReceiver(p)
    fmt.Println("Inside Main after value receiver : ", p.Name)
    PointerReceiver(p1)
    fmt.Println("Inside Main after value receiver : ", p1.Age)
}
```

OUTPUT:

```
Inside ValueReceiver :  ABC
Inside Main after value receiver :  XYZ
Inside PointerReceiver :  24
Inside Main after pointer receiver :  24
```

This shows that the method with value receivers modifies a copy of an object, and the original object remains unchanged. Like, Name of a person changed from ABC to XYZ by ValueReceiver method, but this change was not reflected in the main method. On the other hand, the method with pointer receivers modifies the actual object. Like Age of a person changed from 68 to 24 by the PointerReceiver method, and the same changes reflected in the main method. You can check the fact by printing out the address of the object before and after manipulation by pointer or value receiver.

If you want to change the state of the receiver in a method, manipulating the value of it, **use a pointer receiver**. It's not possible with a value receiver, which copies by value. Any modification to a value receiver is local to that copy. If you don't need to manipulate the receiver value, **use a value receiver**.

The Pointer receiver avoids copying the value on each method call. This can be more efficient if the receiver is a large struct, Value receivers are concurrency safe, while pointer receivers are not concurrency safe. Hence a programmer needs to take care of it.

Functions that accept other functions as parameters:

Go functions can accept other Go functions as parameters, which is a feature that adds flexibility to what we can do with a Go function. The single most common use of this functionality is for sorting elements.

Golang function are first-order variables with the following features:

- They can be assigned to a variable
- Passed around as function argument
- Returned from a function

In GO function is also a type. Two functions are of the same type if they have the same arguments and the same return values. While passing a function as an argument to another function, the exact signature of the function has to be specified in the argument list. As in below example print function accept first argument which is a function of type **func(int, int) int**

```
func print(f func(int, int) int, a, b int)
```

Example:

```
package main

import "fmt"

func main() {
   print(area, 2, 4)
   print(sum, 2, 4)
}

func print(f func(int, int) int, a, b int) {
   fmt.Println(f(a, b))
}

func area(a, b int) int {
   return a * b
}

func sum(a, b int) int {
   return a + b
}
```

Output

```
8
6
```

Important points to note about the above program:

function **area** is a function of **func(int, int) int**

function **sum** is a function of type **func(int, int) int**

area and **sum** are of same type as they have same arguments type and same return values type

print function accepts a function as its first argument of type **func(int, int) int**

Thus both **area** and **sum** function can be passed as an argument to the **print** function.

Method Values and Expressions

Regular expressions and pattern matching

Pattern matching, which plays a key role in Go, is a technique used for searching a string for some set of characters based on a specific search pattern that is based on **regular expressions** and **grammars**. If pattern matching is successful, it allows you to extract the desired data from the string, replace it or delete it.

The Go package which is responsible for defining regular expressions and performing pattern matching is called regexp. When using a regular expression in your code, you should consider the definition of the regular expression as the most important part of your program.

Every regular expression is compiled into a recognizer by building a generalized transition diagram called a **finite automaton**. A finite automaton can be either deterministic or non-deterministic. Non-deterministic means that more than one transition out of a state can be possible for the same input. A **recognizer** is a program that takes a string x as input and is able to tell if x is a sentence of a given language.

A **grammar** is a set of production rules for strings in a formal language. The production rules describe how to create strings from the alphabet of the language that are valid according to the syntax of the language. A grammar does not describe the meaning of a string or what can be done with it in whatever context-it only describes its form. What is important here is to realize that grammars are the important part of regular expressions because, without grammar, you cannot define or use a regular expression.

Interface Types and Values

Go interface type defines the behaviour of other types by specifying a set of methods that need to be implemented. For a type to satisfy an interface, it needs to implement all of the methods required by that interface. Interfaces are abstract types that define a set of functions that need to be implemented so that a type can be considered an instance of the interface. When this happens, we say that the type satisfies this interface. So, an interface has two things: a set of methods and a type, and it is used for defining the behaviour of other types.

The biggest advantage that you receive from having and using an interface is that you can pass a variable of a type that implements that particular interface to any function that expects a parameter of that specific interface.

Two very common Go interfaces are io.Reader and io.Writer, which are used in file input and output operations. More specifically, io.Reader is used for reading a file, whereas io.Writer is used for writing to a file.

io.Reader is defined as follows:

```
type Reader interface {
Read(p []byte) (n int, err error)
}
```

In order for a type to fulfil the io.Reader interface, you will need to implement the Read() method as described in the interface.

io.Writer, is defined as follows:

```
type Writer interface {
Write(p []byte) (n int, err error)
}
```

To fulfil the io.Writer interface, we just need to implement a single method named Write().

Each one of the io.Reader and io.Writer interfaces requires the implementation of just one method, yet both interfaces are good and simple.

Interfaces should be utilized when there is a need for making sure that certain conditions will be met and certain behaviours will be anticipated from a Go element.

Type Assertions and Type Switches

A **type assertion** is the x(T) notation where x is of interface type and T is a type. Additionally, the actual value stored in x is of type T, and T must satisfy the interface type of x.

Type assertions help you do two things. The first thing is to check whether an interface value keeps a particular type. When used in this manner, a type assertion returns two values - the underlying value and a bool value. Although the underlying value is what you might want to use, the Boolean value communicates whether the type assertion was successful or not. The second thing a type assertion is to allow the use of concrete value stored in an interface or assign it to a new variable. This means that if there is an int variable in an interface, you can get that value using type assertion.

Developing your own interfaces:

The technique will be illustrated using the Go code of myInterface.go, which will be presented below. The interface that will be created will help you work with geometric shapes of the plane.

The Go code of myInterface.go follows next:

```
package myInterface
type Shape interface {
Area() float64
Perimeter() float64
}
```

The definition of the shape interface is truly simple, as it requires that we implement just two functions named Area() and Perimeter(), which both return a float64 value. The first function will be used for calculating the area of a shape in the plane and the second one will be used for calculating the perimeter of a shape in the plane. After that, we will need to install the myInterface.go package and make it available to the current user. As we already know, the installation process involves the execution of the following Unix commands:

$ mkdir ~/go/src/myInterface
$ cpmyInterface.go ~/go/src/myInterface
$ go install myInterface

Method Sets with Interfaces

A type determines a set of values together with operations and methods specific to those values. A type may be denoted by a type name, if it has one, or specified using a type literal, which composes a type from existing types.

A type has a (possibly empty) method set associated with it. The method set of an interface type is its interface. The method set of any other type T consists of all methods declared with receiver type T. The method set of the corresponding pointer type *T is the set of all methods declared with receiver *T or T which also contains the method set of T.

Further rules apply to structs containing embedded fields, as described in the section on struct types. Any other type has an empty method set. In a method set, each method must have a unique non-blank method name. The method set of a type determines the interfaces that the type implements and the methods that can be called using a receiver of that type.

An interface type specifies a method set called its interface. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to implement the interface. The value of an uninitialized variable of interface type is nil.

```
InterfaceType      = "interface" "{" { (MethodSpec | InterfaceTypeName ) ";" } "}" .
MethodSpec         = MethodNameSignature .
MethodName           = identifier .InterfaceTypeName  =TypeName .
```

An interface type may specify methods explicitly through method specifications, or it may embed methods of other interfaces through interface type names.

```
// A simple File interface.
interface {
        Read([]byte) (int, error)
        Write([]byte) (int, error)
        Close() error
}
```

The name of each explicitly specified method must be unique and not blank.

Embedded Interfaces

An interface A may use an interface type name B in place of a method specification. This is called embedding interface B in A. The method set of A is the union of the method sets of A's explicitly declared methods and of A's embedded interfaces.

In Go language, the interface is a collection of method signatures and it is also a type means you can create a variable of an interface type. As we know that the Go language does not support inheritance, but the Go interface fully supports embedding.

In embedding, an interface can embed other interfaces or an interface can embed other interface's method signatures in it, the result of both is the same as shown in Example 1 and 2. You are allowed to embed any

number of interfaces in a single interface. And when an interface, embeds other interfaces in it, if we make any changes in the methods of the interfaces, then it will reflect in the embedded interface also.

Syntax:

```
type interface_name1 interface {
Method1()
}
type interface_name2 interface {
Method2()
}
type finalinterface_name interface {
   interface_name1
   interface_name2
}
```

OR

```
type interface_name1 interface {
Method1()
}
type interface_name2 interface {
Method2()
}
type finalinterface_name interface {
Method1()
Method2()
}
```

Example 1:

Go program to illustrate the concept of the embedding interfaces

```
package main
  import "fmt"
  // Interface 1
type AuthorDetails interface {
     details()
}
  // Interface 2
type AuthorArticles interface {
     articles()
}
  // Interface 3
```

```
// Interface 3 embedded with
// interface 1 and 2
type FinalDetails interface {
    AuthorDetails
    AuthorArticles
}
// Structure
type author struct {
    a_name    string
    branch    string
    college   string
    year      int
    salary    int
    particles int
    tarticles int
}
// Implementing method of
// the interface 1
func (a author) details() {
    fmt.Printf("Author Name: %s", a.a_name)
    fmt.Printf("\nBranch: %s and passing year: %d",
                            a.branch, a.year)
    fmt.Printf("\nCollege Name: %s", a.college)
    fmt.Printf("\nSalary: %d", a.salary)
    fmt.Printf("\nPublished articles: %d", a.particles)
}
// Implementing method of the interface 2
func (a author) articles() {
    pendingarticles := a.tarticles - a.particles
    fmt.Printf("\nPending articles: %d", pendingarticles)
}
  // Main value
func main() {
    // Assigning values
    // to the structure
    values := author{
        a_name:    "Shirwaikar",
        branch:    "Computer science",
        college:   "XYZ",
        year:      1990,
        salary:    80000,
        particles: 209,
        tarticles: 309,
    }
    // Accessing the methods of
    // the interface 1 and 2
    // Using FinalDetails interface
    var f FinalDetails = values
    f.details()
    f.articles()
}
```

Output:

Author Name: Shirwaikar

Branch: Computer science and passing year: 1990

College Name: XYZ

Salary: 80000

Published articles: 209

Pending articles: 100

Empty Interfaces

The interface type that specifies zero methods is known as the empty interface:

```
interface{}
```

An empty interface may hold values of any type. (Every type implements at least zero methods). Empty interfaces are used by code that handles values of unknown type.

```
package main

import "fmt"

func myfun(a interface{}) {

   // Extracting the value of a
   val := a.(string)
   fmt.Println("Value: ", val)
}
func main() {

   var val interface {
   } = "SPPU"

   myfun(val)
}
```

OUTPUT:

Value: SPPU

Lab Assignments:

SET A

1. Write a program in go language to create an interface shape that includes area and perimeter. Implements these methods in circle and rectangle type.

2. Write a program in go language to print multiplication of two numbers using method.

3. Write a program in go language to create structure author. Write a method show() whose receiver is struct author.

SET B

1. Write a program in go language to create structure student. Write a method show() whose receiver is a pointer of struct student.

2. Write a program in go language to demonstrate working type switch in interface.

3. Write a program in go language to copy all elements of one array into another using method.

SET C

1. Write a program in go language to create an interface and display it's values with the help of type assertion.

2. Write a program in go language to store n student information(rollno, name, percentage) and write a method to display student information in descending order of percentage.

3. Write a program in go language to demonstrate working embedded interfaces.

Signature of the Instructor: Date:

Assignment Evaluation :

0: Not Done		2: Late Complete		4: Complete	
1: Complete		3: Needs Improvement		5: Well Done	

Assignment No 5 Goroutines and Channels

Objectives

- To study concurrent execution of go routines
- To study how to create go routines and channels

Reading

- You should read following topics:
  - Difference between concurrent and parallel execution of programs.
  - Go routines and channels
  - Wait groups

Ready Reference:

Go routines

In Go, each concurrently executing method or function is called a *go routine*. If you have used operating system threads or threads in other languages, then you can assume for now that a go routine is similar to a thread. The main method in any go program is a root go routine. All other go routines are executed within the main go routine. After termination main all go routines are terminated.

Create new go routine:

New go routines are created by the go statement:

- Syntactically, a go statement is an ordinary function or method call prefixed by the keyword go.
- A go statement causes the function to be called in a newly created go routine. The go statement itself completes immediately.

Example: Creating simple function

```
package main
import "fmt"
func main() {
        message()
        fmt.Print("Hi ! from main go routine")
}

func message() {
        fmt.Print("Hi ! from message go routine")
}
```

Output:

Hi ! from message go routine

Hi ! from main go routine

In the above example we created simple go function and called it from main. The main executes function message and then executes remaining part of main function. This means main waits till message function execution.

Example: Creating go routine

```
package main
import "fmt"

func main() {
        go message()
        fmt.Print("Hi ! from main go routine")
}

func message() {
        fmt.Print("Hi ! from message go routine")
}
```

Output:

Hi ! from main go routine

In this example go routine message is called by main and then starts executing next statements written after go routine. Main will not wait for completion of go routine. Go routines will be executed concurrently.

Sleep() function :

There are two ways we can pause execution of a function like sleep() function from time package and using wait groups. Sleep function pauses the execution of function for given duration,

Example:

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

```
        "time"
)
func main() {
        go f1()
        go f2()
        fmt.Print("\n Hi ! from main go routine")
        time.Sleep(time.Second)
}
func f1() {
        fmt.Print("\n In f1 go routine")
}
func f2() {
        fmt.Print("\n In f2 go routine")
}
```

Output:

```
 Hi ! from main go routine
 In f2 go routine
 In f1 go routine
```

In above example execution of main function is delayed by one second. After launching our go routines f1 and f2, we wait for a second, so our main function was sleeping (blocked) for 1 sec. In that duration all of the go routines were executed successfully.

Wait groups:

A wait group is a synchronization tool provided by the standard library. It can be used to wait for a group of go routines to finish their tasks. Another Go language's standard library primitive "**sync.WaitGroup**". **WaitGroup** is actually a type of counter which blocks the execution of function (or might say A go routine) until **its internal counter become 0**. WaitGroup is concurrency safe, so its safe to pass pointer to it as argument for Go routines.

1	var waitGroup sync.WaitGroup	Create a waitGroup
2	waitGroup.Add(int)	It increases WaitGroup counter by given integer value.
3	waitGroup.Done()	It decreases WaitGroup counter by 1, we will use it to indicate termination of a go routine.
4	waitGroup.Wait()	It Blocks the execution until it's internal counter becomes 0.

Example: Consider following example which gives output same but the program doesn't block for 1 sec .

```
package main
import (
        "fmt"
        "sync"
)
func main() {
        wg := new(sync.WaitGroup)
        wg.Add(2)
        go f1(wg)
        go f2(wg)
        fmt.Print("\n Hi ! from main go routine")
        wg.Wait()
}
func f1(wg *sync.WaitGroup) {
        defer wg.Done()
        fmt.Print("\n In f1 go routine")
}
func f2(wg *sync.WaitGroup) {
```

```
        defer wg.Done()
        fmt.Print("\n In f2 go routine")
}
```

GO Channel

In Go language, a channel is a medium through which a go routine communicates with another go routine and this communication is lock-free. Or in other words, a channel is a technique which allows to let one go routine to send data to another go routine. Channels can be :

- Send Only
- Receive only
- Bidirectional (it can send or receive)

By default channel is bidirectional, means the go routines can send or receive data through the same channel as shown in the below image:

Creating a New Channel :
We can declare a new channel type by using the chan keyword along with a datatype.
Syntax:

```
var Channel_name chan Type

OR
channel_name:= make(chan Type)
```

Example:

1. var mychannel chan int
2. mychannel1 := make(chan int)
3. ch2 := make(chan string, 3) // here 3 is the capacity of the channel.

Unbuffered Channel

A channel is unbuffered when you do not specify it's capacity when you create it. A channel with a size of zero is also unbuffered.

To create an unbuffered channel you can use the following code source :

ch3 := make(chan float)

Or by specifying explicitely a size of 0 (which is equivalent to the previous notation) :

ch4 := make(chan float, 0)

Buffered Channel

A buffered channel is a channel where you specify the buffer size when you create it. A buffered channel has a queue of elements. The queue's maximum size is determined by the capacity argument to make when it is created. The statement below creates a buffered channel capable of holding three string values.

ch := make(chan String, 3)
We can send up to three values on this channel without the goroutine blocking:

ch <- "A"
ch <- "B"
ch <- "C"

Sending and receiving data through channel

In Go language, channel work with two principal operations one is sending and another one is receiving, both the operations collectively known as communication. And the direction of **<-** operator indicates whether the data is received or send. In the channel, the send and receive operation block until another side is not ready by default. It allows go routine to synchronize with each other without explicit locks or condition variables.

Send operation: The send operation is used to send data from one go routine to another go routine with the help of a channel. Values like int, float64, and bool can safe and easy to send through a channel

Syntax:

Channel_name <- value

Example:

Mychannel <- element

Mychannel <- 20

Receive operation: The receive operation is used to receive the data sent by the send operator.

Syntax:

variable := <- Channel_name

Example:

X:= <- Mychannel

The above statement indicates that the X receives data from the channel named as Mychannel.

Functions used for channels:

Function name	Use of function	Example

close()	It is an in-built function used to set a flag which indicates that no more value will send to this channel.
	Syntax: close(channel_name)
		func myfun(mychnl chan string)
		{
		for v := 0; v < 4; v++ {
		mychnl <- "Good Morning"
		}
		close(mychnl)
		}

len()	It is used to find the length of the channel which indicates the number of value queued in the channel buffer.	mychnl := make(chan string, 3) mychnl <- "Hello" mychnl <- "BCA" mychnl <- "Pune" fmt.Println("Length of the channel is: ", len(mychnl))
cap()	It is used to find the capacity of the channel which indicates indicates the size of the buffer.	mychnl := make(chan string, 3) mychnl <- "Hello" mychnl <- "BCA" mychnl <- "Pune" fmt.Println("Length of the channel is: ", cap(mychnl))

Select and case statement in Channel:

In go language, select statement is just like a switch statement without any input parameter. This select statement is used in the channel to perform a single operation out of multiple operations provided by the case block.

The general form of a select statement is shown above.

select {

```
case <-ch1:
   // ...
case x := <-ch2:
   // ...use x...
case ch3 <- y:
   // ...
default:
   // ...
}
```

Lab Assignments:

SET A

1. Write a go program using go routine and channel that will print the sum of the squares and cubes of the individual digits of a number.
   Example if number is 123 then squares = (1 * 1) + (2 * 2) + (3 * 3)
   cubes = (1 * 1 * 1) + (2 * 2 * 2) + (3 * 3 * 3).
   Output
   Sum of squares= 170
   Sum of cubes= 1366
   Final sum of squares and cubes = 1536
2. WAP in GO program that executes 5 go routines simultaneously which generates numbers from 0 to 10, waiting between 0 and 250 ms after each go routine.
3. Write a go program that creates a slice of integers, checks numbers from slice are even or odd and further sent to respective go routines through channel and display values received by go routines.

SET B

1. .WAP in Go to create buffered channel, store few values in it and find channel capacity and length. Read values from channel and find modified length of a channel.
2. WAP in Go main go routine to read and write Fibonacci series to the channel.
3. WAP in Go how to create channel and illustrate how to close a channel using for range loop and close function.

SET C

1. Write a go program to implement the checkpoint synchronization problem which is a problem of synchronizing multiple tasks. Consider a workshop where several workers assembling details of some mechanism. When each of them completes his work, they put the details together. There is no store, so a worker who finished its part first must wait for others before starting another one. Putting details together is the checkpoint at which tasks synchronize themselves before going their paths apart.

Signature of the Instructor: Date:

Assignment Evaluation :

0: Not Done		2: Late Complete		4: Complete	
1: Complete		3: Needs Improvement		5: Well Done	

Assignment 6: Packages and File

Go was designed to be a language that encourages good software engineering practices. An important part of high quality software is code reuse – embodied in the principle "Don't Repeat Yourself."

Objectives

- How to implement Package and file program.

Reading

You should read the following topics before starting this exercise.

- Core packages.
- How to create package and file.

Ready Reference

Go was designed to be a language that encourages good software engineering practices. An important part of high quality software is code reuse – embodied in the principle "Don't Repeat Yourself."

Packages in Golang:

Packages are the most powerful part of the Go language. The purpose of a package is to design and maintain a large number of programs by grouping related features together into single units so that they can be easy to maintain and understand and independent of the other package programs. This modularity allows them to share and reuse. In Go language, every package is defined with a different name and that name is close to their functionality like "strings" package and it contains methods and functions that are only related to strings.

Important Points

In Go language, every package is defined by a unique string and this string is known as import path. With the help of an import path, you can import packages in your program. For example:

```
import "fmt"
```

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

Bundling code in this way serves 3 purposes:

- It reduces the chance of having overlapping names. This keeps our function names short and succinct
- It organizes code so that its easier to find code you want to reuse.
- It 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.

Creating Packages

Packages only really make sense in the context of a separate program which uses them. Without this separate program we have no way of using the package we create. Let's create an application that will use a package we will write. Create a folder in ~/src/golang-book called goprogram . Inside that folder create a file called main.go which contains this:

```
package main
import "fmt"
import "golang-book/goprogram /math"
func main() {
  xs := []float64{1,2,3,4}
  avg := math.Average(xs)
  fmt.Println(avg)
}
```

Now create another folder inside of the goprogram folder called math. Inside of this folder create a file called math.go that contains this:

```
package math

func Average(xs []float64) float64 {
  total := float64(0)
  for _, x := range xs {
    total += x
  }
  return total / float64(len(xs))
}
```

Using a terminal in the math folder you just created run go install. This will compile the math.go program and create a linkable object file: ~/pkg/os_arch/golang-book/goprogram/math.a. (where os is something like windows and arch is something like amd64)

Giving Names to the Packages

In Go language, when you name a package you must always follow the following points:

1. When you create a package the name of the package must be short and simple. For example strings, time, flag, etc. are standard library packages.

2. The package name should be descriptive and unambiguous.

3. Always try to avoid choosing names that are commonly used or used for local relative variables.

4. The name of the package is generally in the singular form. Sometimes some packages are named in plural form like strings, bytes, buffers, etc. Because to avoid conflicts with the keywords.

5. Always avoid package names that already have other connotations.

The Core Packages

Strings

Go includes a large number of functions to work with strings in the strings package:

```
package main


import (
  "fmt"
  "strings"
)


func main() {
  fmt.Println(
    // true
    strings.Contains("test", "es"),


    // 2
    strings.Count("test", "t"),


    // true
    strings.HasPrefix("test", "te"),


    // true
    strings.HasSuffix("test", "st"),
```

```
    // 1
    strings.Index("test", "e"),

    // "a-b"
    strings.Join([]string{"a","b"}, "-"),

    // == "aaaaa"
    strings.Repeat("a", 5),

    // "bbaa"
    strings.Replace("aaaa", "a", "b", 2),

    // []string{"a","b","c","d","e"}
    strings.Split("a-b-c-d-e", "-"),

    // "test"
    strings.ToLower("TEST"),

    // "TEST"
    strings.ToUpper("test"),
  )
}
```

Sometimes we need to work with strings as binary data. To convert a string to a slice of bytes (and vice-versa) do this:

```
arr := []byte("test")
str := string([]byte{'t','e','s','t'})
```

Input / Output

Before we look at files we need to understand Go's io package. 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)

Errors

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:

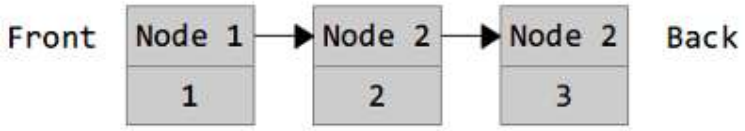
```
package main
import "errors"
func main() {
  err := errors.New("error message")
```

Containers & Sort

In addition to lists and maps Go has several more collections available underneath the container package. We'll take a look at the container/list package as an example.

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

Sort

The sort package contains functions for sorting arbitrary data. There are several predefined sorting functions (for slices of ints and floats).

Testing

Programming is not easy; even the best programmers are incapable of writing programs that work exactly as intended every time. Therefore an important part of the software development process is testing. Writing tests for our code is a good way to ensure quality and improve reliability.

Go includes a special program that makes writing tests easier, so let's create some tests for the package we made in the last chapter. In the math folder from goprogram 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
```

You should see this:

```
$ go test
PASS
ok      golang-book/goprogram/math      0.032s
```

The go test command will look for any tests in any of the files in the current folder and run them. Tests are identified by starting a function with the word Test and taking one argument of type *testing.T. In our case since we're testing the Average function we name the test function TestAverage.

Once we have the testing function setup we write tests that use the code we're testing. In this case we know the average of [1,2] should be 1.5 so that's what we check. It's probably a good idea to test many different combinations of numbers so let's change our test program a little:

```
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,
      )
    }
  }
}
```

This is a very common way to setup tests (abundant examples can be found in the source code for the packages included with Go). We create a struct to represent the inputs and outputs for the function. Then we create a list of these structs (pairs). Then we loop through each one and run the function.

Files

The most important package that allows us to manipulate files and directories as entities is the os package.

The io package has the io.Reader interface to reads and transfers data from a source into a stream of bytes. The io.Writer interface reads data from a provided stream of bytes and writes it as output to a target resource.

Create an empty file

ex.

```
package main

import (
        "log"
        "os"
)


func main() {
        emptyFile, err := os.Create("empty.txt")
        if err != nil {
                log.Fatal(err)
        }
        log.Println(emptyFile)
        emptyFile.Close()

}
```

Reading XML file

The xml package includes Unmarshal() function that supports decoding data from a byte slice into values. The xml.Unmarshal() function is used to decode the values from the XML formatted file into a Notes struct.

Sample XML file:

The notes.xml file is read with the ioutil.ReadFile() function and a byte slice is returned, which is then decoded into a struct instance with the xml.Unmarshal() function. The struct instance member values are used to print the decoded data.

ex.

```
<note>
<to>Tove</to>
```

```
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

```
package main
 import (
        "encoding/xml"
        "fmt"
        "io/ioutil"
)

type Notes struct {
        To      string `xml:"to"`
        From    string `xml:"from"`
        Heading string `xml:"heading"`
        Body    string `xml:"body"`
}

func main() {
        data, _ := ioutil.ReadFile("notes.xml")

        note := &Notes{}

        _ = xml.Unmarshal([]byte(data), &note)

        fmt.Println(note.To)
        fmt.Println(note.From)
        fmt.Println(note.Heading)
        fmt.Println(note.Body)
}
```

Writing XML file

The xml package has an Marshal() function which is used to serialized values from a struct and write them to a file in XML format.

The notes struct is defined with an uppercase first letter and ″xml″ field tags are used to identify the keys. The struct values are initialized and then serialize with the xml.Marshal() function. The serialized XML formatted byte slice is received which then written to a file using the ioutil.WriteFile() function.

```
package main
 import (
        "encoding/xml"
        "io/ioutil"
)
 type notes struct {
        To      string `xml:"to"`
        From    string `xml:"from"`
        Heading string `xml:"heading"`
        Body    string `xml:"body"`
}

func main() {
        note := ¬es{To: "Nicky",
                From:    "Rock",
                Heading: "Meeting",
                Body:    "Meeting at 5pm!",
        }
        file, _ := xml.MarshalIndent(note, "", " ")

        _ = ioutil.WriteFile("notes1.xml", file, 0644)
 }
```

Lab Assignments:

SET A

1. WAP to create student struct with student name and marks and sort it based on student marks using sort package

2. WAP in Go language using user defined package calculator that performs one calculator operation as per the user's choice.

3. WAP in Go language to create an user defined package to find out the area of a rectangle.

SET B

1. WAP in Go language to add two integers and write code for unit test to test this code.

2. WAP in Go language to subtract two integers and write code for table test to test this code.

3. Write a function in Go language to find the square of a number and write a benchmark for it.

SET C

1. WAP in Go language to read a XML file into structure and display structure

2. WAP in Go language to print file information.

3. WAP in Go language to add or append content at the end of a text file.

Signature of the Instructor: Date:

Assignment Evaluation :

0: Not Done		2: Late Complete		4: Complete	
1: Complete		3: Needs Improvement		5: Well Done	