1... Introduction

Learning Objectives ...

Students 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.

1.1 INTRODUCTION

- **Go** is a statically typed, compiled programming language designed at Google by Robert Griesemer (Swiss computer scientist), Rob Pike (Canadian Programmer), and Ken Thompson (American pioneer of computer science).
- Go is syntactically same as C, but with memory safety, garbage collection, structural typing, and CSP-style concurrency. The language is often referred to as **Golang** because of its domain name, golang.org, but the proper name is Go.
  - **Statistically Typed**: Static type checking is the process of verifying the type safety of a program based on analysis of a program's text (source code). If a program passes a static type checker, then the program is guaranteed to satisfy some set of type safety properties for all possible inputs.
  - **Compiled programming language**: A compiled language is a programming language whose implementations are typically compilers not interpreters.
  - **Google**: Google LLC is an American Multinational company that specializes in internet related services and products, includes online advertising technologies, search engine, cloud computing, software and hardware.
  - **Memory safety**: This is the state of being protected from various software bugs and security vulnerabilities (weakness) when dealing with memory access (buffer overflows, dangling pointers).

- **Garbage Collection:** It is a form of automatic memory management. It is trying to collect the memory which was previously allocated but currently dereferenced.
- **Structural typing**: A structural type system is a major class of type systems in which type compatibility and equivalence is determined by the type's actual structure or definition and not by other characteristics (its name and place of declaration).
- **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. The language designers to address criticism of other languages in use at Google, but keep their useful characteristics as follows:
  - Static typing (set of rules set by that language which assigns a property like variables, expressions, functions or modules) and runtime (final phase of program life cycle) efficiency (like C programming language).
  - Readability (easy way so that reader can understand reader text) and usability (system performs tasks, safely, effectively, efficiently) (like Python or JavaScript).
  - High-performance networking and multiprocessing.
  - Go was publicly announced in November 2009, and its first version is released in March 2012. Go is widely used in production at Google and in many other organizations and open-source projects.

Difference between C++ and GO

	C++	GO
Design	It is Object Oriented and procedural programming language.	It was designed and influenced by C programming language with special features.
Inheritance	Allows multiple inheritances.	Does not allow multiple inheritance as it does not support class based declaration.
Optimization	It provides SIMD optimizations during its compilation process.	It does not provide optimizations in its compiler.
Classes	Structs and classes are same in C++.	Go does not support class based declaration but instead has interfaces to support objects.
License	Open source project 2.0.	It is licensed under BSD license.
Type	Static typing.	It is static typing and strong typing discipline.
Polymorphism	It is checked at compile time.	Does not exist.
Boolean operators	Uses true, false and bool.	It has logical operators instead of bool.

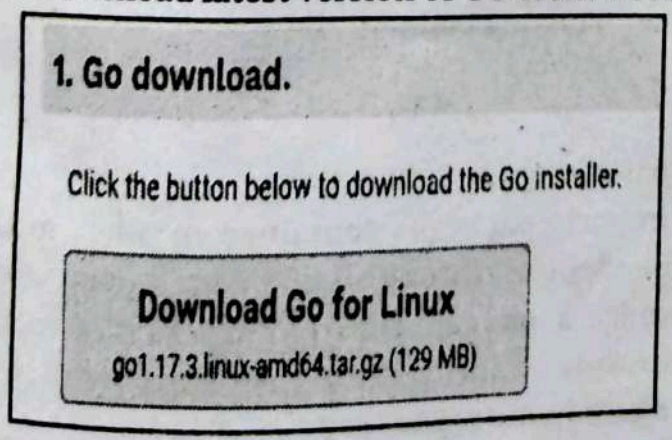
contd. ...

Templates	It has STL (Standard Template Library).	Does not have Template libraries.
Supports	Files and Streams, Exception Handling, Dynamic memory, Namespaces, Templates, Pre-processors, and Multi-threading.	An inbuilt concurrency feature with channels and lightweight processes.
Features	Namespaces, References, Templates, implicit method overloading.	Pointers, Structures, Slice, Range, Maps, Recursion, Functions, Interfaces, Type Embedding and Error Handling.
Operator overloading	Supports.	Does not support operator overloading as it will increase complexity, and the same can be implemented over structures to avoid complexity by using pointer receivers.
Memory allocation	Runtime at the heap of the objects.	Heap, Stack, and Data Segments and also uses garbage collection.

- Go is used widely in production inside Google. One easy example is the server behind golang.org. It is just the godoc document server running in a production configuration on Google App Engine.
- A more significant instance is Google's download server, dl.google.com, which delivers chrome binaries and other large installable such as apt-get packages.
- Go is not the only language used at Google, but it is a key language for a number of areas including Site Reliability Engineering (SRE) and large-scale data processing.
- A couple of major cloud infrastructure projects written in Go are Docker and Kubernetes, but there are many more. You can refer to the page that many companies using GO language: GoUsers · golang/go Wiki · GitHub.

1.1.1 Installation of GO

Step 1 : Download latest version of GO from website https://golang.org/dl/.



Step 2 : Install:

Select the tab for your computer's operating system below, then follow its installation instructions.

(i) Extract the archive you downloaded into /usr/local, creating a Go tree in /usr/local/go.

Important: This step will remove a previous installation at /usr/local/go, if any, prior to extracting. Please back up any data before proceeding.

For example, run the following as root or through sudo:

```
rm -rf /usr/local/go && tar -C /usr/local -xzf go1.17.3.linux-amd64.tar.gz
```

(ii) Add /usr/local/go/bin to the PATH environment variable.

You can do this by adding the following line to your $HOME/.profile or /etc/profile (for a system-wide installation):

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

Note: Changes made to a profile file may not apply until the next time you log into your computer. To apply the changes immediately, just run the shell commands directly or execute them from the profile using a command such as source $HOME/.profile.

(iii) Verify that you have installed Go by opening a command prompt and typing the following command:

```
$ go version
```

(iv) Confirm that the command prints the installed version of Go.

Step 3 : Start Coding

(i) Open a command prompt and cd to your home directory.

On Linux or Mac:

```
cd
```

On Windows:

```
cd %HOMEPATH%
```

(ii) Create a hello directory for your first Go source code.

For example, use the following commands:

```
mkdir hello
cd hello
```

(iii) Enable dependency tracking for your code

When your code imports packages contained in other modules, you manage those dependencies through your code's own module. That module is defined by a go.mod file that tracks the modules that provide those packages. That go.mod file stays with your code, including in your source code repository.

To enable dependency tracking for your code by creating a go.mod file, run the go mod init command, giving it the name of the module your code will be in. The name is the module's module path.

In actual development, the module path will typically be the repository location where your source code will be kept. For example, the module path might be github.com/mymodule.

If you plan to publish your module for others to use, the module path must be a location from which Go tools can download your module. For more about naming a module with a module path, see Managing dependencies.

For the purposes of this tutorial, just use example/hello.

```
$ go mod init example/hello
go: creating new go.mod: module example/hello
```

(iv) In your text editor, create a file hello.go in which to write your code.

Paste the following code into your hello.go file and save the file.

```
package main
import "fmt"
func main() {
fmt.Println("Hello, World!")
}
```

(v) This is your Go code. In this code, you:

(a) Declare a main package (a package is a way to group functions, and it is made up of all the files in the same directory).

(b) Import the popular fmt package, which contains functions for formatting text, including printing to the console. This package is one of the standard library packages you got when you installed Go.

(c) Implement a main function to print a message to the console. A main function executes by default when you run the main package.

(vi) Run your code to see the greeting.

```
$ go run.
Hello, World!
```

The go run command is one of many go commands you'll use to get things done with Go. Use the following command to get a list of the others:

```
$ go help
```

1.2 GO RUNTIME AND COMPILATIONS

- GO program language structure:
  - Package Declaration.
  - Import Packages.
  - Functions Variables.
  - Statements and Expressions.
  - Comments.

Explanation:

- **Package Declaration:** First line int Go program is package name. It defines the package name into which program can be written. It is main package. It is also a mandatory statement. Main package is the starting point to run any go program. Each package has a path and a name associated with it.
- **Import Packages:** import fmt is a preprocessor directive command which tells go compiler to include the files lying in the package fmt.
- **Functions Variables:** func main is main function where program execution begins.
- **Statements and Expressions:** They are included in { }.
- **Comments:** Comments are represented by /*....*/ and //.

Example:

```
package main
import "fmt"
func main()
{
    /* Comment This is first GO Programme*/
    fmt.Println("Welcome to Go Programming Language")
}
```

Println:

- It is function variable in GO which prints message given in double inverted commas on screen. Note here Pin Println is in capital letter. In Go language a name is exported if it starts with capital letter. Exported means the function or variable/constant is accessible to the importer of the respective package.

1.2.1 Steps to Execute Go Program

- Follow the following steps to execute go program.
  - Open a text editor and type the program.
  - Save the file as program_name.go
  - Open the command prompt.
  - Go to the directory where you saved the file.
  - Type go run program_name.go and press enter to run your code.
  - If there are no errors in your code, then you will see the output printed on the screen.

- For example if above program will have name as welcome.go (on windows)

```
$ go run welcome.go
Welcome to Go Programming Language
```

1.2.2 Basic Building Blocks of Go Programming Language

Tokens:

- A token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following Go statement consists of six tokens.
- For example: fmt.Println("Welcome to Go Programming")
- The tokens of above statements are as follows:

```
fmt
.
Println
(
   "Welcome to Go Programming"
)
Line Separator
```

- In a Go program, the line separator key is a statement terminator.

Different ways to import packages:

- **import format "fmt":** Creates an alias of fmt. Precede all fmt package content with format instead of fmt.
- **import. "fmt":** Allows content of the package to be accessed directly, without the need for it to be preceded with fmt.
- **import_"fmt":** Suppresses compiler warnings related to fmt if it is not being used, and executes initialization functions if there are any. The remainder of fmt is inaccessible.

1.3 KEYWORDS AND IDENTIFIERS

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.
- Go is case sensitive programming language. For example abc and Abc are two different identifiers.

- Following are some examples of identifiers.

Identifier	Valid/Invalid	Identifier	Valid/Invalid
mahesh	Valid	a_123	Valid
Movie_name	valid	A123	Valid
movie@name	Invalid		
$abc	Invalid		
_abc	valid		

- **List of Keywords in go programming language:** There are 25 keywords in go programming language.

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

Whitespace:

- A line containing only whitespace, possibly with a comment, is known as a blank line, and a Go compiler totally ignores it.
- Whitespaces separate one part of a statement from another.
- Whitespace enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:
  var a int;(at least one whitespace is needed to distinguish variable name and datatype)

Types in Go programming language:

Sr. No.	Type	Description
1.	Boolean types	They consists of the two predefined constants: (a) true (b) false
2.	Numeric types	They represent (a) integer types or (b) floating point values throughout the program.
3.	String types	They represent the set of string values. Its value is a sequence of bytes. Strings are immutable type that is once created, it is not possible to change the contents of a string. The predeclared string type is string.
4.	Derived types	They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types, (e) Function types, f) Slice types, g) Interface types, h) Map types, i) Channel Types.
5.	Aggregate types	They are array and structure. (include other types)

Integer types:

Types	Description
uint8	Unsigned 8-bit integers (0 to 255)
uint16	Unsigned 16-bit integers (0 to 65535)
uint32	Unsigned 32-bit integers (0 to 4294967295)
uint64	Unsigned 64-bit integers (0 to 18446744073709551615)
int8	Signed 8-bit integers (-128 to 127)
int16	Signed 16-bit integers (-32768 to 32767)
int32	Signed 32-bit integers (-2147483648 to 2147483647)
int64	Signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

Float types:

float32	IEEE-754 32-bit floating-point numbers
float64	IEEE-754 64-bit floating-point numbers
complex64	Complex numbers with float32 real and imaginary parts
complex128	Complex numbers with float64 real and imaginary parts

Numeric types:

byte	same as uint8
rune	same as int32
uint	32 or 64 bits
int	same size as uint
uintptr	an unsigned integer to store the uninterrupted bits of a pointer value

1.4 CONSTANTS AND VARIABLES

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:

```
const PI= 3.142
const (
a = 1
b = 2
c = 3
)
```

```
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)
}
```

(in case of Variable Large: range of integer is extended)

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 srno int
var sname string
var saddress string
```

Example for initializing variables: Variables can be declared and initialized as well.

For Example 1:

```
var sname string="Minal"
```

For Example 2:

```
var {
srno int =101
sname string ="Minal"
}
```

If an initializer is not present the type can be omitted. In this case the variable will take the type of the initializer.

```
var {
srno          =101
sname  ="Minal"
saddress      ="S.B. Road"
}
```

Variables can be initialized on the same line also.

```
var {
srno, sname, saddress = 101, "Minal","S.B. Road"
}
```

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

```
func main() {
sname,saddress:= "Minal","S.B. Road"
srno:=101
}
A variable can contain any type, including functions:
func main() {
add:=func() {
//statements
}
add()
}
```

How to print constants and variables?

- Print or Println will be used to print constants and variables. The ideal way is to use fmt package.

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 Six"
   alis:= fmt.Sprintf("Number %d", 6)
   fmt.Printf("%s is also known as %s",name, alis)
}
```

Output: Number Six is also known as Number 6

Scope of a variable:

1. **Inside a function:** In this type variable is not accessed by another function.

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

Output:

```
Welcome
```

2. **Outside a function:** This means that other functions can access this variable. Go is lexically scoped using blocks.

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

Output:

```
Welcome
Welcome
```

3. **Defining multiple variables:** Following is another way of defining multiple variables.

```
var(
a=5
b=10
c=15
)
```

Format specifiers for Printf

1. **For general:**

%v	the value in a default format when printing structs, the plus flag (%+v) adds field names
%#v	a Go-syntax representation of the value
%T	a Go-syntax representation of the type of the value
%%	a literal percent sign; consumes no value

2. **For Boolean:**

%t	the word true or false

3. **Integer:**

%b	base 2
%c	the character represented by the corresponding Unicode code point
%d	base 10
%o	base 8
%O	base 8 with 0o prefix
%q	a single-quoted character literal safely escaped with Go syntax.
%x	base 16, with lower-case letters for a-f
%X	base 16, with upper-case letters for A-F
%U	Unicode format: U+1234; same as "U+%04X"

4. **Floating point and constant constituents:**

%b	decimal less scientific notation with exponent a power of two, in the manner of strconv. FormatFloat with the 'b' format, e.g. -123456p-78
%e	scientific notation, e.g. -1.234456e+78
%E	scientific notation, e.g. -1.234456E+78
%f	decimal point but no exponent, e.g. 123.456
%F	synonym for %f
%g	%e for large exponents, %f otherwise. Precision is discussed below.
%G	%E for large exponents, %F otherwise
%x	hexadecimal notation (with decimal power of two exponent), e.g. 0x1.23abcp+20
%X	upper-case hexadecimal notation, e.g. -0X1.23ABCP+20

5. **String and slice of bytes (treated equivalently with these verbs):**

%s	the uninterpreted bytes of the string or slice
%q	a double-quoted string safely escaped with Go syntax
%x	base 16, lower-case, two characters per byte
%X	base 16, upper-case, two characters per byte

6. **Slice:**

%p	address of 0^{th} element in base 16 notation, with leading 0x

7. **Pointer:**

%p	base 16 notation, with leading 0x The %b, %d, %o, %x and %X verbs also work with pointers, formatting the value exactly as if it were an integer.

1.5 OPERATORS AND EXPRESSIONS

- Operators are used to perform specific mathematical or logical manipulations. GO language provides following types of operators.

Arithmetic Operators:

Operator	Description	Example A=10, B=2
+	Adds two operands	A + B = 12
-	Subtracts second operand from the first	A - B = 8
*	Multiplies both operands	A * B = 20
/	Divides the numerator by the denominator.	A/B = 5
%	Modulus operator; gives the remainder after an integer division.	A%B = 0
++	Increment operator. It increases the integer value by one.	A ++ A = 11
--	Decrement operator. It decreases the integer value by one.	A -- A = 9

Relational Operators:

Operator	Description	Example A=10, B=2
==	**Equals:** It checks whether the values of two operands are equal or not; if yes, then the condition becomes true.	if(A= =B) Answer: False
!=	**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.	if(A!=B) Answer: 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.	if(A>B) Answer: 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.	if(A<B) Answer: False
>=	**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.	if(A>=B) Answer: 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.	if (A<=B) Answer: False

Logical Operators:

Operator	Description	Example A=10, B=2
&&	Logical AND operator. If both the operands are non-zero, then condition becomes true.	if (A&&B) Answer: False
||	Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	if (A||B) Answer: 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.	if(A!=B) Answer: True

Bitwise Operators:

- These operators perform bit by bit operation.

a	b	Bitwise and a & b	Bitwise Or a | b	Bitwise Xor a ^ b
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Program 1.1: Write a program to illustrate use of bitwise operators.

```
package main
import "fmt"
func main()
{
    p:= 3
    q:= 4
        // & (bitwise AND Operator)
    fmt.Printf("p=%d,q=%d",p,q)
      result1:= p & q
    fmt.Printf("\nResult of p & q = %d", result1)
        // | (bitwise OR Operator)
      result2:= p | q
    fmt.Printf("\nResult of p | q = %d", result2)
        // ^ (bitwise XOR Operator)
      result3:= p ^ q
    fmt.Printf("\nResult of p ^ q = %d", result3)
        // << (left shift Operator)
      result4:= p << 1
    fmt.Printf("\nResult of p << 1 = %d", result4)
        // >> (right shift Operator)
      result5:= p >> 1
    fmt.Printf("\nResult of p >> 1 = %d", result5)
        // &^ (AND NOT Operator)
      result6:= p &^ q
    fmt.Printf("\nResult of p &^ q = %d", result6)
}
```

Output:

```
p=3,q=4
Result of p & q = 0
Result of p | q = 7
Result of p ^ q = 7
Result of p << 1 = 6
Result of p >> 1 = 1
Result of p &^ q = 3
```

Assignment Operators:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Miscellaneous Operators:–

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

Operators Precedence in Go:

Operator	Description	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* &sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

1.6 LOCAL ASSIGNMENTS

- Short variable declarations may appear only inside functions. In some contexts such as the initializers for "if", "for", or "switch" statements, they can be used to declare local temporary variables.

1.7 BOOLEANS, NUMERIC, CHARACTERS

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.
- The values true and false will always be with a lowercase t and f respectively, as they are pre-declared identifiers in Go.

Example, user can store Boolean value in a variable.

```
K:=5>8
```

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

```
fmt. Println(K)
```

as 5 is not greater than 8 we receive following output.

Output:

```
false
```

Numeric:

1. **Integers**: Integers in programming language are whole numbers i.e positive, negative or zero. In GO Language integers are known as int.

   For example

   ```
   var n int =4
   fmt.Println(n)
   ```

2. **Float**: Float is used to represent floating point number i.e. real number which are not expressed as real 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.

   ```
   var n float =4.2
   fmt.Println(n)
   ```

   With integers and floating-point numbers, it is important to keep in mind that $3 \neq 3.0$, as 3 refers to an integer while 3.0 refers to a float.

   As there is distinction between integers and floats, Go has two types of numeric data that are distinguished by the static or dynamic nature of their sizes.

   The first type is an **architecture-independent** type, which means that the size of the data in bits does not change, regardless of the machine that the code is running on.

   In all today systems architectures are either 32 bit or 64 bit. For instance, On Laptop having OS Windows, (Operating system runs on a 64-bit architecture). However, if you are developing for a device like a fitness watch, you may be working with a 32-bit architecture. If you use an architecture-independent type like int32, regardless of the architecture you compile for, the type will have a constant size.

   The second type is an **implementation-specific** type. In this type, the bit size can vary based on the architecture the program is built on. For instance, if we use the int type, when Go compiles for a 32-bit architecture, the size of the data type will be 32 bits. If the program is compiled for a 64-bit architecture, the variable will be 64 bits in size.

   Go have the following architecture-independent integer types:

uint8	unsigned 8-bit integers (0 to 255)
uint16	unsigned 16-bit integers (0 to 65535)
uint32	unsigned 32-bit integers (0 to 4294967295)
uint64	unsigned 64-bit integers (0 to 18446744073709551615)
int8	signed 8-bit integers (-128 to 127)
int16	signed 16-bit integers (-32768 to 32767)
int32	signed 32-bit integers (-2147483648 to 2147483647)
int64	signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

Floats and complex numbers also come in varying sizes:

float32	IEEE-754 32-bit floating-point numbers
float64	IEEE-754 64-bit floating-point numbers
complex64	complex numbers with float32 real and imaginary parts
complex128	complex numbers with float64 real and imaginary parts

Picking Numeric Data Types:

- As there are architecture-independent types, and implementation-specific types in GO Programming language. For integer data, it is common in Go to use the implementation types like int or uint instead of int64 or uint64. This will typically result in the fastest processing speed for your target architecture.
- For instance, if you use an int64 and compile to a 32-bit architecture, it will take at least twice as much time to process those values as it takes additional CPU cycles to move the data across the architecture. If you used an int instead, the program would define it as a 32-bit size for a 32-bit architecture, and would be significantly faster to process.
- If you know you won't exceed a specific size range, then picking an architecture-independent type can both increase speed and decrease memory usage. For example, if you know your data won't exceed the value of 100, and will only be a positive number, then choosing a uint8 would make your program more efficient as it will require less memory.
- Now, what will happen if we exceed those ranges in program?

Overflow vs Wraparound:

- Go has the potential to both *overflow* a number and *wraparound* a number when you try to store a value larger than the data type was designed to store, depending on if the value is calculated at compile time or at runtime. A compile time error happens when the program finds an error as it tries to build the program. A runtime error happens after the program is compiled, while it is actually executing.
- Consider the following example. We can set variable a to its maximum value.

```
package main
import "fmt"
func main()
{
   var a   uint32=4294967295// Max uint32 size
   fmt.Println(a)
}
```

Output:

```
4294967295
```

If we add 1 to the value at runtime, it will wraparound to 0:

Output:

```
0
```

- On the other hand, let's change the program to add 1 to the variable when we assign it, before compile time:

```
package main
import "fmt"
funcmain()
{
   var a   uint32=4294967295 + 1// Max uint32 size
   fmt.Println(a)
}
```

- At compile time, if the compiler can determine a value will be too large to hold in the data type specified, it will throw an overflow error. This means that the value calculated is too large for the data type you specified.
- Because the compiler can determine it will overflow the value, it will now throw an error:

Output:

```
prog.go:6:36: constant 4294967296 overflows uint32
```

This will help user to avoid potential bugs in program in the future.

Characters:

- Golang does not have any data type of '**char**'. Therefore
  - **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.
  - **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.
  - 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
- To declare either a **byte** or a **rune** we use single quotes. While declaring **byte** we have to specify the type, if we don't specify the type, then the default type is meant as a **rune.**
- To declare a **string**, we use double quotes or backquotes. Double quotes string honors escape character while back quotes string is a raw literal string and doesn't honor any kind of escaping.
- Consider following example. It shows:
  - A byte representing the character 'a'
  - A rune representing the pound sign '£'
  - A string having one character micro sign 'µ'

```
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
```

Program 1.2: Write a Program in GO to check if a string contains alphabetic characters.

```
package main
import
{
   "fmt"
}
func checkAlphabetinstring(str string) bool
{
   for _, charVariable:= range str
   {
      if (charVariable< 'a' || charVariable> 'z') && (charVariable< 'A'
      || charVariable> 'Z')
      {
         return false
      }
   }
   return true
}
func main()
{
   fmt.Println(checkAlphabetinstring("Program")) // true
   fmt.Println(checkAlphabetinstring("1234"))    // false
}
```

Output:

```
true
false
```

1.8 POINTERS AND ADDRESSES

- Go pointers are easy to learn. Pointers are mostly used for call by reference. Pointer is a variable which stores memory address of another variable. Every location in memory is having some address that is denoted by '&'.

  Syntax: `var var_name*var_type`

- Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.

Example:

```
Var ip *int        /* pointer to an integer */
Var fp *float32    /* pointer to a float */
```

Program 1.3: Write a Program in GO to print address of a variable.

```
package main
import "fmt"
func main()
{
   var a int = 10
   fmt.Printf("Address of a variable: %x\n", &a)
}
```

Output:

```
Address of a variable: c000016058
```

Program 1.4: Write a Program in GO to illustrate the use of pointer variable.

```
package main
import "fmt"
func main()
{
   var a int = 20   /* declaring actual variable */
   var ip *int      /* declaring pointer variable */
   ip = &a  /* pointer variable will store address of variable*/
   fmt.Printf("Address of a variable: %x\n", &a)
      /* address stored in pointer variable */
   fmt.Printf("Address stored in ip variable: %x\n", ip)
      /* access the value using the pointer */
   fmt.Printf("Value of *ip variable: %d\n", *ip)
}
```

Output will be:

```
Address of a variable: c42000e1f8
Address stored in ip variable: c42000e1f8
Value of *ip variable: 20
```

Nil pointers in GO:

- Go compiler assign a Nil value to a pointer variable when user do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned nil is called a nil pointer.
- The nil pointer is a constant with a value of zero defined in several standard libraries.
- Consider the following example

```
package main
import "fmt"
func main()
{
   var  ptr *int
   fmt.Printf("The value of ptr is: %x\n", ptr  )
}
```

Output of above program will be:

```
The value of ptr is: 0
```

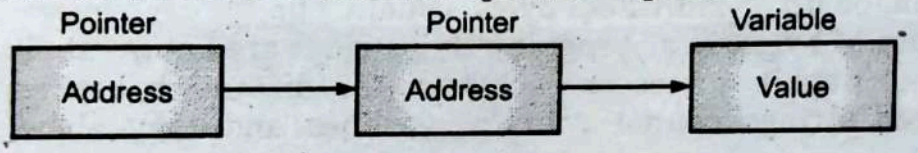
- On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the nil (zero) value, it is assumed to point to nothing.

  To check for a nil pointer you can use an if statement as follows:

```
if(ptr != nil)     /* succeeds if p is not nil */
if(ptr == nil)     /* succeeds if p is null */
```

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;`

Program 1.5: Write a Program in go to illustrate pointer to pointer concept.

```
package main
import "fmt"
funcmain()
{
   var a int
   var p1 *int
   var p2 **int
   a = 3
   p1 = &a // p1 store address of variable a
   p2 = &p1// p2 will store address of variable p1
   fmt.Printf("Value of a = %d\n", a )
   fmt.Printf("Value available at *p1 = %d\n", *p1 )
   fmt.Printf("Value available at **p2 = %d\n", **p2)
}
```

- 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.

Program 1.6: Write a Program in GO to explain use of new function.

```
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
```

1.9 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. Strings are read only slice of bytes. In go programming strings are known as slices. Strings are created using double quotes or backticks. The difference between these is that double-quoted strings cannot contain newlines and they allow special escape sequences.
- For example, \n gets replaced with a newline and \t gets replaced with a tab 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"
```

- Whenever compiler encounters a string literal in code the compiler creates a string object with its value. In above example it is "Welcome".
- A string can hold arbitrary bytes. A string literal holds a valid UTF-8 sequence called runes.

Program 1.7: Write a Program in GO to accept a string and print it.

```
package main
import "fmt"
func main()
{
   var s1="Welcome"
   fmt.Printf("%s",s1)
}
```

Output:

```
Welcome
```

String operations:

1. **Length of the string:** len function is used to print length of the string.
   Example:

```
package main
import "fmt"

func main()
{
   var s1 =  "Welcome to GO"
   fmt.Printf("String Length is: ")
   fmt.Println(len(s1))
}
```

   Output: String Length is: 13

2. **Concatenation of strings:** join method from strings package will join multiple strings.
   Syntax: strings.join(sample, " ")
   Second parameter is the separate to join the two strings.

```
package main
import ("fmt"
        "strings"
       )
func main()
{
   s1:=  []string{"Welcome","Programming"}
   fmt.Println(strings.Join(s1, " "))
}
```

   Output: Welcome Programming
   Same program will be written with another separator as follows. For example, can also work as separator.

```
package main
import ("fmt"
        "strings"
       )
funcmain()
{
   s1:=  []string{"Welcome","Programming"}
   fmt.Println(strings.Join(s1, ","))
}
```

   Output will be: Welcome,Programming
   + character is also used to concatenate two strings.
   For example "Welcome, " + to GO Programming" concatenates two strings

3. **Accessing a particular character of a string:** "Welcome"[1] accesses particular character in the string. In this example it access second character.

Program 1.8: Write a program to illustrate the use of various string functions.

```
package main
import (
    "fmt"
    s "strings"
)
var p = fmt.Println
func main()
{
    p("Contains:  ", s.Contains("test", "es"))
    p("Count:     ", s.Count("test", "t"))
    p("HasPrefix: ", s.HasPrefix("test", "te"))
    p("HasSuffix: ", s.HasSuffix("test", "st"))
    p("Index:     ", s.Index("test", "e"))
    p("Join:      ", s.Join([]string{"a", "b"}, "-"))
    p("Repeat:    ", s.Repeat("a", 5))
    p("Replace:   ", s.Replace("foo", "o", "0", -1))
    p("Replace:   ", s.Replace("foo", "o", "0", 1))
    p("Split:     ", s.Split("a-b-c-d-e", "-"))
    p("ToLower:   ", s.ToLower("TEST"))
    p("ToUpper:   ", s.ToUpper("test"))
    p("Len: ", len("hello"))
    p("Char:", "hello"[1])
}
```

Output:

```
Contains:  true
Count:     2
HasPrefix: true
HasSuffix: true
Index:     1
Join:      a-b
Repeat:    aaaaa
Replace:   f00
Replace:   f0o
Split:     [a b c d e]
ToLower:   test
ToUpper:   TEST
Len:  5
Char: 101
```

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.

(note p is fmt.Println means it is a shortcut for Println)

1.10 if-else, switch, for loop

1.10.1 if else

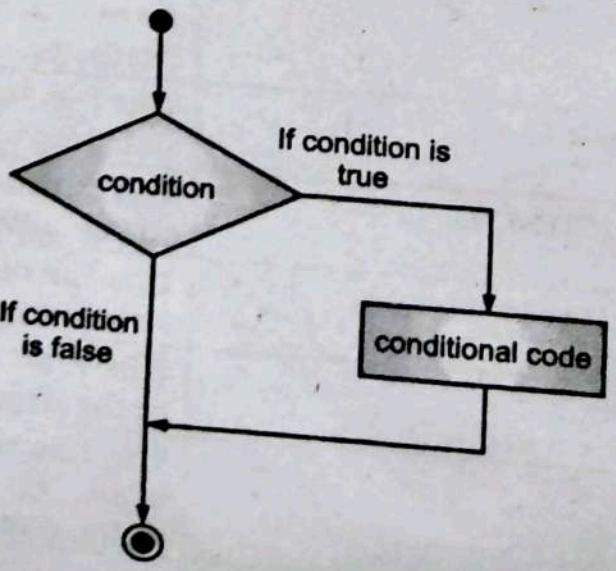
- Following types of decision-making statements are provided by GO programming language.

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 */
}
```



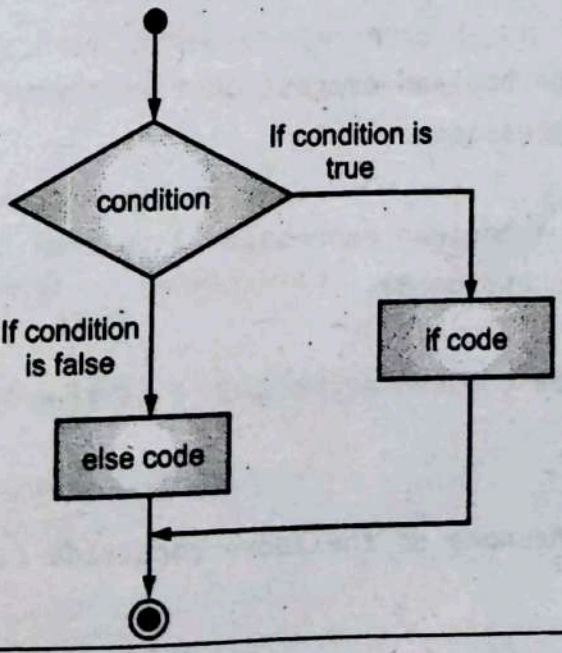
Program 1.9: Write a program in Go to check whether number positive negative or zero.

```
package main
import (
   "fmt"
)
func main()
{
   var a int
   fmt.Printf("Enter the Number:")
   fmt.Scanf("%d" ,&a)
   if(a==0)
   {
      fmt. Printf(" Accepted number %d is ZERO",a)
   }
   if(a>0)
   {
      fmt. Printf(" Accepted number %d is POSITIVE",a)
   }
  if(a<0)
   {
      fmt. Printf(" Accepted number %d is NEGATIVE",a)
   }
}
```

Output:

```
Enter the Number:0
Accepted number 0 is ZERO
```

if else statement:



Program 1.10: Write a program to check whether number is even or odd.

```
package main
import "fmt"
func main()
{
    var a int
    fmt.Printf("\nEnter the number:")
    fmt.Scanf("%d",&a)
    if( (a % 2 )== 0 )
    {
        fmt.Printf("%d is Even  number\n" ,a)
    }    else
    {
        fmt.Printf("%d is Odd number\n", a)
    }
}
```

Output:

```
Enter the number:3
3 is Odd number
```

---

if ...else if ..if .. else statement

- An if statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.
- When using if, else if, else statements there are few points to keep in mind:
  - An if can have zero or one else's and it must come after any else if's.
  - An if can have zero to many else if's and they must come before the else.
  - Once an else if succeeds, none of the remaining else if's or else's will be tested.

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 */
}
```

Program 1.11: Write a GO program to check whether number is positive, negative or zero.

```
package main
import "fmt"
funcmain()
{
   var a int = 0
   if(a<0)
   {
      fmt.Printf("%d Number is negative\n" ,a)
   } else if(a>0)
   {
      fmt.Printf("%d Number is positive\n", a)
   }  else
   {
      fmt.Printf("%d is ZERO\n", a)
   }
}
```

Output:

```
0 is ZERO
```

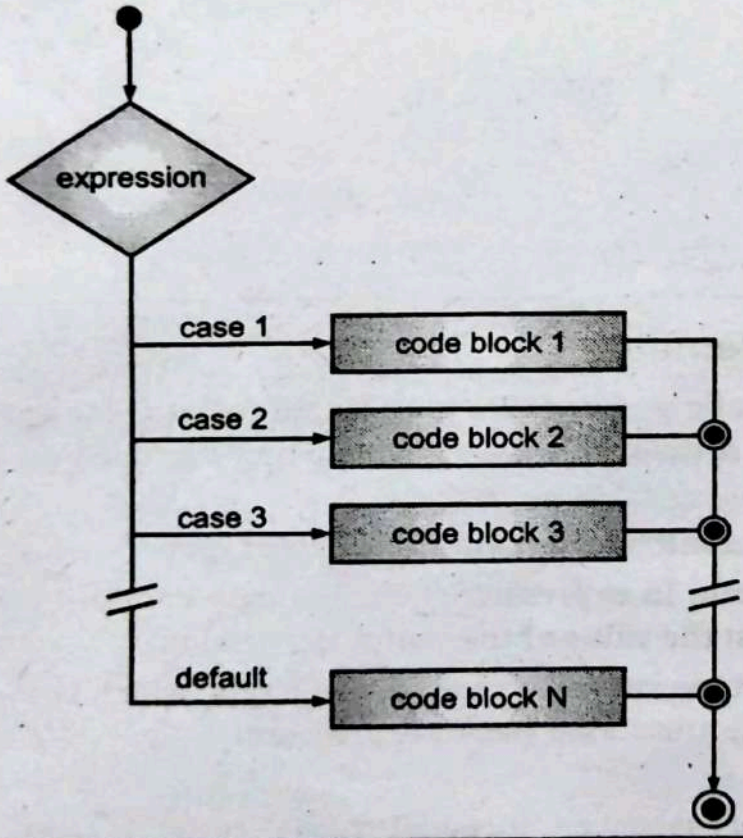
1.10.2 Switch statement

- **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);
}
```

- The following rules apply to a **switch** statement:
  - The **expression** used in a **switch** statement must have an integral or boolean expression, or be of a class type in which the class has a single conversion function to an integral or boolean value. If the expression is not passed then the default value is true.
  - You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
  - The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
  - When the variable being switched on is equal to a case, the statements following that case will execute. No **break** is needed in this case statement.
  - A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.



Program 1.12: Write a program in GO illustrate switch statement.

```
package main
import "fmt"
funcmain()
{
   /* local variable definition */
   var grade string = "K"
   var marks int
    fmt.Printf("\nEnter the makrs:");
    fmt.Scanf("%d",&marks)
```

```
    switch marks
    {
       case 90: grade = "A"
       case 80: grade = "B"
       case 50,60,70: grade = "C"
       default: grade = "D"
    }
    switch
    {
          case grade == "A":
       fmt.Printf("Excellent!\n" )
          case grade == "B", grade == "C":
       fmt.Printf("Well done\n" )
          case grade == "D":
       fmt.Printf("You passed\n" )
          case grade == "F":
       fmt.Printf("Better try again\n" )
          default:
       fmt.Printf("Invalid grade\n" );
    }
    fmt.Printf("Your grade is  %s\n", grade );
}
```

(The above program will give output for given numbers only)

Output:

```
Enter the makrs:80
Well done
Your grade is  B
```

Program 1.13: Write a GO program to print number name of a number for single digit. If number is other than single digit, then display the message accordingly.

```
package main
import "fmt"
func main() {
   var d int = 9
switch  d {
      case 0  :
fmt.Printf("ZERO")
      case 1:
fmt.Printf("ONE\n" )
      case 2:
fmt.Printf("TWO\n" )
      case 3:
fmt.Printf("Three\n" )
```

```
        case 4:
    fmt.Printf("Four\n" )
        case 5:
    fmt.Printf("Five\n" )
        case 6:
    fmt.Printf("Six\n" )
        case 7:
    fmt.Printf("Seven\n" )
        case 8:
    fmt.Printf("Eight\n" )
        case 9:
    fmt.Printf("Nine\n" )
        default:
    fmt.Printf("Not a single digit number\n" );
      }
    }
```

Output:

```
Nine
```

1.10.3 For loop

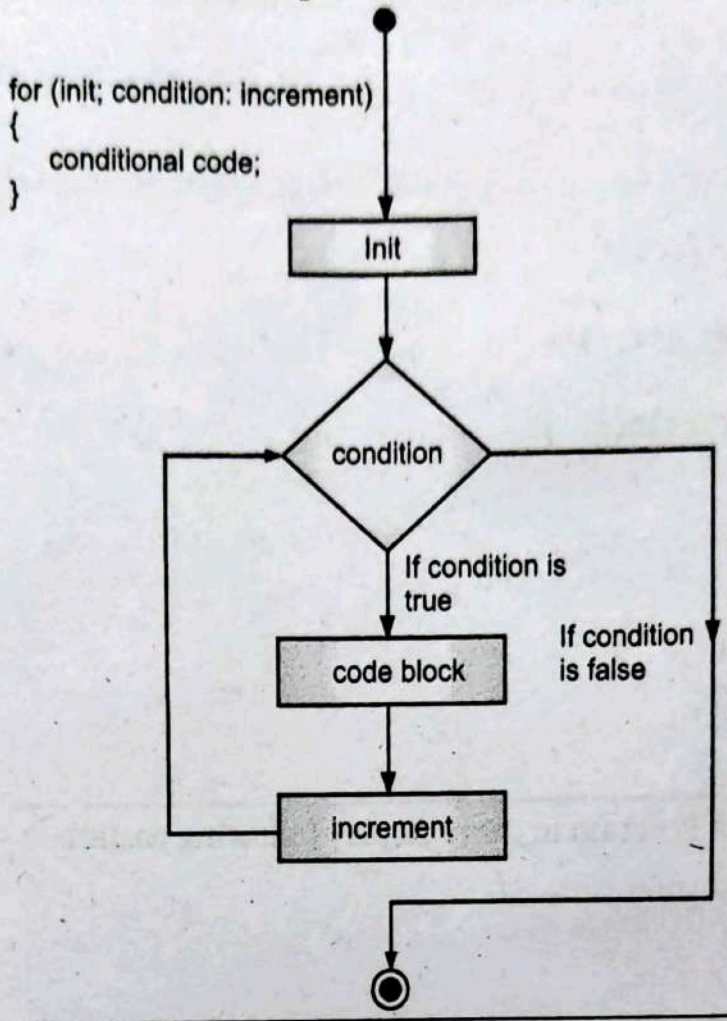
- For loop is used to run Go language script n number of times.

  Syntax:

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

- The flow of control in a **for** loop is as follows:
- If a **condition** is available, then for loop executes as long as condition is true.
- If a **for** clause **(init; condition; increment)** is present then:
  - The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
  - Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the **for** loop.
  - After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
  - The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again the condition). After the condition becomes false, the for loop terminates.

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



Program 1.14: Write a Program in GO language 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
Nested For Loop
```

Program 1.15: Write a Program in GO language to illustrate the concept of Nested For loop

```
package main
import "fmt"
funcmain()
{
    for i:= 0; i<2; i++
    {
        for j:= 0; j<2; j++
        {
            fmt.Println(i, j)
        }
    }
}
```

Output:

```
0 0
0 1
1 0
1 1
```

Program 1.16: Write a Program in GO to display following pattern

```
******
******
******
******
******
package main
import "fmt"
func main()
{
    for i:= 0; i< 5; i++
    {
        fmt.Printf("\t\t\t\t")
        for j:= 0; j < 5; j++
        {
            fmt. Printf("* ")
        }
        fmt.Printf("\n")
    }
}
```

Output:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

1.11 ITERATIONS

- It can be useful to skip to the next iteration of a loop early. This can be a good pattern for **guard** clauses within a loop. Guard clause is the ability to return early from function or loop.
- Following example will clear this concept. Break and continue will be used to execute this concept.

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

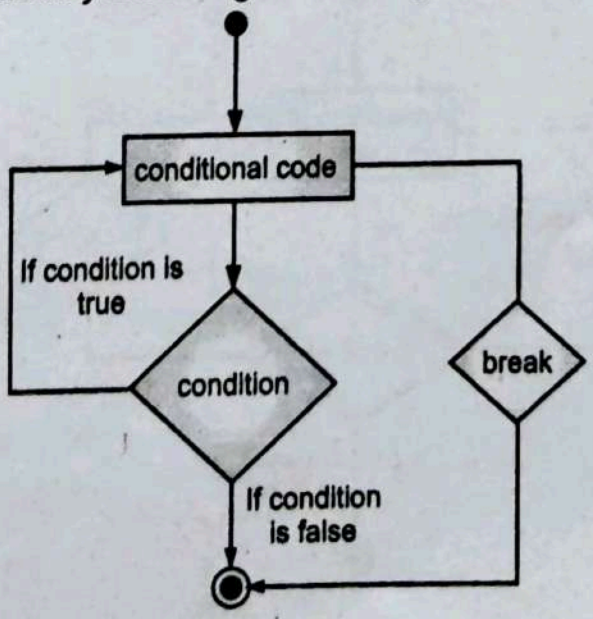
Output:

```
1 is odd
3 is odd
5 is odd
7 is odd
9 is odd
```

1.12 USING BREAK AND CONTINUE

1. **Break statement:**

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



Program 1.17: Write a program in GO to illustrate the use of break statement.

```
package main
import "fmt"
func main()
{
    var a int
    a=11
    for   a<20
    {
        fmt.Printf("Value of a=%d\n",a)
        a++;
        if (a>15) {  break; }
    }
}
```

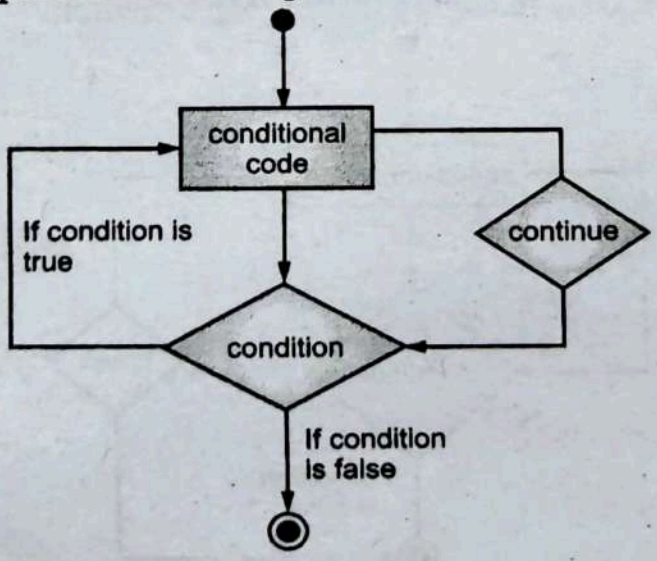
Output:

```
Value of a=11
Value of a=12
Value of a=13
Value of a=14
Value of a=15
```

- In the above example when value of a becomes 15 loop will be break though value of a is <20.

2. 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.



```
package main
import "fmt"
func main()
{
   for i:= 1; i<= 5; i++
   {
      if (i == 3)
      {// condition for continue
         continue;
      }
      fmt.Printf("%d",i)
   }
}
```

Output:

```
1245
```

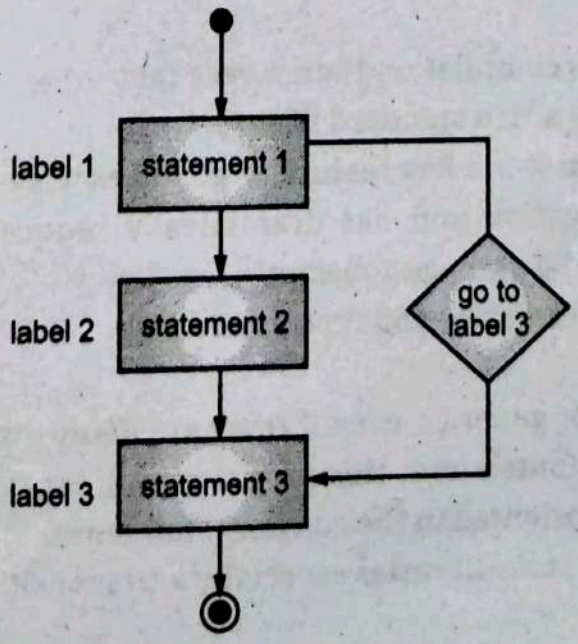
- In above program the number 3 will not get printed as loop will be continued for number value 3.

3. 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;
```



Program 1.18: Write a Program in GO to illustrate the working of goto statement.

```
package main
import "fmt"
func main()
{
   var a int = 10
   LOOP: for a < 20
   {
      if a == 15
      {
         a = a + 1
         goto LOOP
      }
      fmt.Printf("value of a: %d\n", a)
      a++
   }
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Advantages:

1. **Flexible:** It is concise, simple and easy to read.
2. **Concurrency:** It allows multiple processes running simultaneously and effectively.
3. **Quick Outcome:** Its compilation time is very fast.
4. **Library:** It provides a rich standard library.
5. **Garbage collection:** It is a key feature of go. Go excels in giving a lot of control over memory allocation and has dramatically reduced latency in the most recent versions of the garbage collector.
6. It validates for the interface and type embedding.

Disadvantages:

1. It has no support for generics, even if there are many discussions about it.
2. The packages distributed with this programming language is quite useful but Go is not so object-oriented in the conventional sense.
3. There is absence of some libraries especially a UI tool kit.

Some popular Applications developed in Go Language:

- **Docker:** A set of tools for deploying linux containers.
- **Openshift:** A cloud computing platform as a service by Red Hat.
- **Kubernetes:** The future of seamlessly automated deployment processes.
- **Dropbox:** Migrated some of their critical components from Python to Go.
- **Netflix:** For two part of their server architecture.
- **InfluxDB:** Is an open-source time series database developed by InfluxData.
- **Golang:** The language itself was written in Go.

Summary

- Go is new language which introduces many interesting features which make many aspects of programming much easier, particularly in implementing parallel programs.
- Go makes use of a simple memory model with go routines to automatically manage multiple threads of execution, which can be much easier than the manual approach that is required in C.
- Go is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson.
- **Println:** It is function variable in GO which prints message given in double inverted commas on screen.
- Tokens keywords identifiers, constants, identifiers are basic building blocks of go language.
- Go language supports different arithmetic, relational, logical, bitwise operators.
- Go strings are made up of individual bytes, usually one for each character.
- Decision making statements like if-else, switch and loop constructs like for work as same in c programming language.

Sample Programs for Study Purpose

1. WAP in GO language to print Student name, rollno, division and college name.

```
package main
import "fmt"
func main()
{
   name, CollegeName, div:= "Datta Kulkarni", "Modern College", "A"
   Rollno:= 32
   fmt.Printf("Name:%s\nRollNo:%d\nDiv:%s\nCollegeName:%s", name, Rollno,
   div, CollegeName)
}
```

Output:

```
Name:Datta Kulkarni
RollNo:32
Div:A
CollegeName:Modern College
```

2. WAP in GO language to print whether number is even or odd.

```
package main
import "fmt"
func main()
{
    var a int
    fmt.Printf("Enter  number:")
    fmt.Scanf("%d", &a)
    if((a % 2)== 0 )
    {
       fmt.Printf("%d is Even  number\n" ,a)
    }
    fmt.Printf("%d is Odd number\n", a)
}
```

Output:

```
Enter number:5
5 is Odd number
```

---

3. WAP in GO language to swap the number without temporary variable.

```
package main
import "fmt"
func main()
{
    var a,b int
    fmt.Printf("Enter Two Numbers:")
    fmt.Scanf("%d %d", &a,&b)
    fmt.Printf("\nBefore Swapping:")
    fmt.Printf("a= %d b=%d", a,b)
            a = a + b;
            b = a - b;
            a = a - b;
    fmt.Printf("\nAfter Swapping:")
    fmt.Printf("a= %d b=%d", a,b)
}
```

Output:

```
Enter   Two Numbers:2 3

Before Swapping:  a= 2 b=3
After Swapping:  a= 3 b=2
```

---

4. WAP in GO to print table of given number.

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

```
func main()
{
   var  a int
   fmt.Printf("\nEnter the Number:")
   fmt.Scanf("%d",&a)
   for i:= 1; i<= 10; i++
   {
      fmt.Printf("%d * %d = %d \n", a,i,i*a)
   }
}
```

Output:

```
Enter the Number:3
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
```

---

5. WAP in GO language to print PASCALS triangle

```
package main
import "fmt"
func main()
{
   var n,a int
   fmt.Printf("Enter the number of rows: ")
   fmt.Scanf("%d",&n)
   for row:=1; row<=n; row++
   {
      a=1
      for i:=1; i<=40-row; i++
      {
         fmt.Printf(" ")
      }
      for i:=1;i<=row;i++
      {
         fmt.Printf("%d ",a)
         a = a * (row-i)/i
      }
      fmt.Printf("\n")
   }
}
```

Output:

```
Enter the number of rows: 5
    1
   11
  121
 1331
14641
```

6. WAP in GO language to print Fibonacci series of n terms.

```
package main
import "fmt"
func main()
{
    var  t1,t2, n,nt int
    t1 = 0
    t2 = 1
    nt = t1 + t2
    fmt.Printf("Enter the number of terms: ")
    fmt.Scanf("%d", &n);
    fmt.Printf("Fibonacci Series: %d, %d, ", t1, t2)
    for i:= 3; i<= n; i++
    {
        fmt.Printf("%d, ", nt)
        t1 = t2;
        t2 = nt;
        nt = t1 + t2;
    }
}
```

Output:

```
Enter the number of terms: 6
Fibonacci Series: 0, 1, 1, 2, 3, 5,
```

7. WAP in GO language to accept two strings and compare them.

```
package main
import "fmt"
func main()
{
    str1:= "Wels"
    str2:= "Wel"
    str3:= "WelcometoGO"
    str4:= "Wels"
    // Checking the string are equal or not using  == operator
    result1:= str1 == str2
```

```
    result2:= str2 == str3
    result3:= str3 == str4
    result4:= str1 == str4
    fmt.Println("Result 1: ", result1)
    fmt.Println("Result 2: ", result2)
    fmt.Println("Result 3: ", result3)
    fmt.Println("Result 4: ", result4)
    // Checking the string are not equal or not using != operator
    result5:= str1 != str2
    result6:= str2 != str3
    result7:= str3 != str4
    result8:= str1 != str4
    fmt.Println("\nResult 5: ", result5)
    fmt.Println("Result 6: ", result6)
    fmt.Println("Result 7: ", result7)
    fmt.Println("Result 8: ", result8)
}
```

Output:

```
Result 1:  false
Result 2:  false
Result 3:  false
Result 4:  true

Result 5:  true
Result 6:  true
Result 7:  true
Result 8:  false
```

8. WAP in GO language to accept user choice and print answer of using arithmetical operators.

```
package main
import (
      "fmt"
)
func main()
{
    fmt.Println("Hello, playground")
    var a,b,z int
    fmt.Printf("Enter two numbers")
    fmt.Scanf("%d%d", &a, &b)
    fmt.Printf("\n1:Addition       \n2:Subtraction       \n3:Multiplication
    \n4:Division \n5:Remainder")
    fmt.Printf("\nEnter Your Choice:")
    fmt.Scanf("%d" , &z)
```

```
    switch  z
    {
        case 1:
            fmt.Printf("\nAddition=%d",a+b)
                case 2:
            fmt.Printf("\nSubtraction=%d",a-b)
                case 3:
            fmt.Printf("\nMultiplication=%d",a*b)
                case 4:
            fmt.Printf("\nDivision=%d",a/b)
                case 5:
            fmt.Printf("\nRemainder=%d",a%b)
                default:
            fmt.Printf("Wrong Choice\n" );
    }
}
```

Output:

```
Hello, playground
Enter two numbers3 4

1:Addition
2:Subtraction
3:Multiplication
4:Division
5:Remainder
Enter Your Choice:1

Addition=7
```

Check Your Understanding

1. Which of the following is true about Go programming language?
   (a) Go is a general-purpose language designed with systems programming in mind.
   (b) It is strongly and statically typed, provides inbuilt support for garbage collection.
   (c) It supports concurrent programming.
   (d) All of the Above
2. Which of the following is true about select statement in Go?
   (a) You can have any number of case statements within a select. Each case is followed by the value to be compared to and a colon.
   (b) The type for a case must be the communication channel operation.
   (c) When the channel operation occurred the statements following that case will execute. No break is needed in the case statement.
   (d) All of the above.

3. Which of the following is true about break statement in Go?
   (a) When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
   (b) It can be used to terminate a case in the switch statement.
   (c) Both of the above.
   (d) None of the above.
4. Which of the following is true about global variables in Go?
   (a) Global variables are defined outside of a function, usually on top of the program.
   (b) The global variables will hold their value throughout the lifetime of your program.
   (c) A global variable is available for use throughout your entire program after its declaration.
   (d) All of the above.
5. Which of the following is correct about ranges in Go?
   (a) The range keyword is used in for loop to iterate over items of an array, slice, channel or map.
   (b) With array and slices, it returns the index of the item as integer.
   (c) With maps, it returns the key of the next key-value pair.
   (d) All of the above.
6. Which of the following method of parameter passing, copies the actual value of an argument into the formal parameter of the function?
   (a) call by value (b) call by reference
   (c) Both of the above (d) None of the above
7. What's another name for Go?
   (a) Golang (b) Gooang
   (c) Godong (d) Gobong
8. Which of the following is initial value (zero value) for interfaces, slice, pointers, maps, channels and functions?
   (a) 0 (b) " "
   (c) Nil (d) False
9. Where was the language created?
   (a) Grocery store (b) Google
   (c) Apple (d) Restaurant
10. When was the language created?
    (a) 2007 (b) 2008
    (c) 2009 (d) 2010

11. One of the following had a hand in creating the language?
    (a) Micheal Rodney (b) Rob Pike
    (c) Harry Lobman (d) Carrick Mcdon
12. In which quarter of 2009 was the programming language created?
    (a) 4th quarter (a) 2nd quarter
    (c) 3rd quarter (d) 1st quarter
13. Which version was released in February, 2018?
    (a) 1.50 (b) 1.20
    (c) 1.10 (d) 1.30
14. Is it recommended to use Global Variables in a program that implements go routines?
    (a) It is designed by Google.
    (b) It is statistically typed programming language.
    (c) GOLang is syntactically similar in java.
    (d) "Testbook" is a valid declaration /initialization.
15. In GoLang, which of the following transfers control to the labelled statement?
    (a) enum (b) goto
    (c) jump (d) return
16. Which of the following terminates the for loop or switch statement and transfers execution to the statement immediately following the for loop or switch in Go?
    (a) goto (b) break
    (c) Continue (d) switch

Answers

1. (d)	2. (d)	3. (c)	4. (d)	5. (d)	6. (a)	7. (a)	8. (c)	9. (b)	10. (c)
11. (b)	12. (a)	13. (c)	14. (c)	15. (b)	16. (b)				

Trace the Output

1.
```
package main
import "fmt"
func main()
{
    var X uint8 = 225
    fmt.Println(X+1, X)
    var Y int16 = 32767
    fmt.Println(Y+2, Y-2)

    a:= 20.45
    b:= 34.89
    var m complex128 = complex(6, 2)
    var n complex64 = complex(9, 2)
    c:= b - a
```

```
    fmt.Printf("Result is: %f\n", c)
    fmt.Printf("The type of c is: %T\n", c)
    fmt.Println(m)
    fmt.Println(n)
    fmt.Printf("The type of m is %T and "+
    "the type of n is %T", m, n)
}
```

2.
```
package main
import "fmt"
func main()
{
    str1:= "Keyboard"
    str2:= "Is input Device"
    result1:= str1 == str2
    fmt.Println(result1)
    fmt.Printf("The type of result1 is %T\n", result1)
    str:= "Hello"
    fmt.Printf("Length of the string is: %d", len(str))
    fmt.Printf("\nString is: %s", str)
    fmt.Printf("\nType of str is: %T", str)
}
```

3.
```
package main
import "fmt"
func main()
{
    var b1 =5
    fmt.Printf("The value of b1 is: %d\n", b1)
    fmt.Printf("The type of b1 is: %T\n", b1)
}
```

4.
```
import "fmt"
const PI = 3.14
funcmain()
{
    const GFG = "LoveCoding"
    fmt.Println("Hello", GFG)
    fmt.Println("Happy", PI, "Maths Coding")
    const Correct = true
    fmt.Println("Go rules to be verified?", Correct)
}
```

Practice Questions

Q. I Answer the following questions in short.

1. What is the purpose of Println?
2. Who is known as father of Go programming language?
3. Does Go support generic programming?
4. Is Go a case sensitive language?
5. What is a string literal in Go programming?
6. What is workspace in Go?
7. What is the default value of type bool in Go programming?
8. What is GOPATH environment variable in go programming?
9. What are the advantages/ benefits of Go programming language?
10. What are the several built-in supports in Go?
11. What is the usage of break statement in Go programming language?
12. What is the usage of continue statement in Go programming language?
13. What is the usage of goto statement in Go programming language?
14. How a pointer is represented in Go?
15. Is it true that short variable declaration:= can be used only inside a function?
16. How can you format a string without printing?
17. What is the default value of a local variable in Go?
18. What is default value of a global variable in Go?
19. What is the default value of a pointer variable in Go?

Q. II Answer the following questions.

1. Explain switch statement with syntax.
2. Explain if else statement with syntax.
3. Explain the syntax for 'for' loop.
4. What is goroutine in Go programming language?
5. How to write multiple strings in Go programming?
6. Write the syntax to create a function in Go programming language?
7. Explain static type declaration of variable in Go programming language?
8. Explain dynamic type declaration of a variable in Go programming language?
9. How would you print type of variable in Go?
10. How can you check a variable type at runtime in Go programming language?
11. Is it recommended to use Global Variables in a program that implements go routines?

Q. III Define the terms.

1. Pointer
2. String

■■■

2...

Functions

Learning Objectives ...

Students 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.

2.1 INTRODUCTION

- A function is a group of statements performs some tasks. Every GO program has at least one function, That is main().
- A function declaration tells the compiler about a function name, return type and parameters. Functions are also known as method, sub-routine, or procedure.

Defining a Function:

- A function definition provides the actual body of the function. 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 add(num1, num2 int) int
{
    var result int// local variables in function
    result = num1 + num2
    return result //returning value
}
```

Calling Function:

- To use a function, user must call a function to complete the defined task. When program will call a function the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.
- To call a function user must pass the required parameters along with its function name. If the function returns a value, then you can store the returned value.
- For example, to call above function;

```
z=add(a,b)
```

2.2 PARAMETERS AND RETURN VALUES

- Following are the three types how functions can be used in a program.

Addition of two numbers:

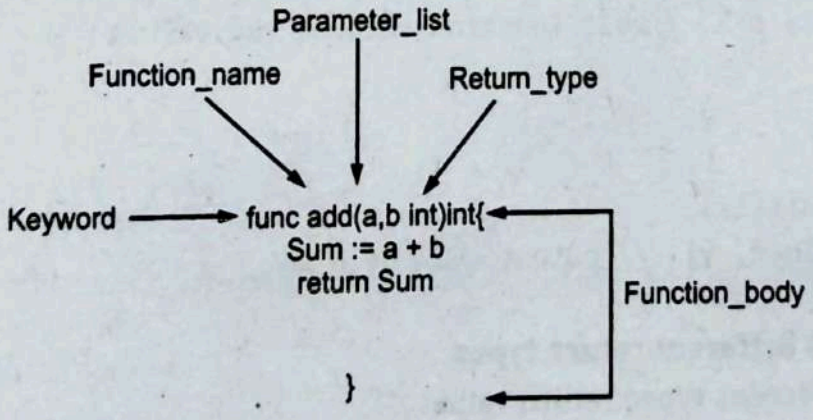
Program without function	Program with using function which does not accept any parameters and does not return any values	Program with using function which does not accept parameters but return values	Program with using function which accept any parameters and return values

package main
import "fmt"
func main()
{
var a,b,c int
fmt.Printf("Enter two numbers:")
fmt.Scanf("%d%d", &a, &b)
c = a + b
fmt.
Println("Addition =",c)
}
	package main
	import "fmt"
	func main()
	{
	add()
	}
		package main
		import "fmt"
		func main()
		{
		var a,b int
		fmt.Printf("Enter two numbers:")
		fmt.Scanf("%d%d", &a, &b)
		add(a,b)
		}
			package main
			import "fmt"
			func main()
			{
			var a,b,c int
			fmt.Printf("Enter two numbers:")
			}

contd. ...

	func add( )
	{
	var a,b,c int
	fmt.Printf("Enter two numbers:")
	fmt.Scanf("%d%d", &a, &b)
	c = a + b
	fmt.
	Println("Addition =",c)
	}
		func add(num1, num2 int)
		{
		var c int
		c = num1 + num2
		fmt.
		Println("Addition =",c)
		}
			fmt.Scanf("%d%d", &a, &b)
			c = add(a, b)
			fmt.Printf("Addition is : %d\n", c)
			}
			func add(num1, num2 int) int
			{
			var result int
			result = num1 + num2
			return result
			}

	No any formal and actual parameters	a and b are actual parameters num1 and num2 are formal parameters	a and b are actual parameters num1 and num2 are formal parameters



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 mult(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 mult(a int) (int, int)
{
   return a/2, a*2  //mult function returns two values
}
func main()
{
   x, y := mult(12)
   fmt.Println(x, y)  // output will be 6 24
}
```

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
}
```

Benefits of multiple return values:

- Multiple return values are extremely useful when checking errors as you can see in following example. It makes the program flow easier since you are checking the errors in a step-by-step manner. Thus omits the need for complicated and nested error-checking and other problems.
- Multiple return values return each of them separately so it reduces the needs of unpacking data from a packed structure. It also reduces the needs of returning pointers to data.

Skipping a return value:

- Any number of return values can be ignored completely. Go has a literal, the hyphen (-) which can be used in place of the data and the data will be skipped completely. Here we can see how to skip return values.

For Example:

```
package main
import "fmt"
func abc(a int) (int, int)
{
    return a*2, a*4
}
func main()
{
    x, _ := abc(2)
    /*observe the skip operator is used to ignore the compiler doesn't
    throw error*/
    fmt.Println(x)  // 4
}
```

Needs for skipping return values:

- In Go programming user can initialize a variable but do not use it, Go compiler will give errors. The error will be very annoying if you store all return values in variables but do not use it. This problem can be entirely avoided by skipping the variables.

For Example:

- 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!!!
}
```

2.3 CALL BY VALUE AND REFERENCE

2.3.1 Call by Value

- 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. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
/* function definition to swap the values */
func swap(int x, int y) int
{
   var temp int
   temp = x /* save the value of x */
   x = y    /* put y into x */
   y = temp /* put temp into y */
   return temp;
}
```

- If the same code will be put into program for example, consider the following code:

```
package main
import "fmt"
func main()
{
   var a int = 1 //local variable
   var b int = 2 //local variable
   fmt.Printf("Before swap, value of a = %d ,b= %d \n", a, b)
   swap(a, b)
   fmt.Printf("After swap, value of a = %d b = %d\n", a, b)
}
func swap(x, y int) int
{
   var temp int
   temp = x
   x = y
   y = temp
   return temp;
}
```

Output:

```
Before swap, value of a = 1 b= 2
After swap, value of a = 1 b = 2
```

2.3.2 Call by Reference

- 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. Accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

For example:

```
/* function definition to swap the values */
func swap(x *int, y *int)
{
   var temp int
   temp = *x    /* save the value at address x */
   *x = *y      /* put y into x */
   *y = temp    /* put temp into y */
}
```

- To learn more about pointers in Go programming, please go through Go- Pointers.
- For now, let us call the function swap() by passing values by reference as in the following example:

```
package main
import "fmt"
func main()
{
   var a int = 1 //local variable
   var b int = 2 //local variable
   fmt.Printf("Before swap, value of a = %d b=%d\n", a, b )
   swap(&a, &b)
   fmt.Printf("After swap, value of a= %d b=%d\n", a ,b )
}
func swap(x *int, y *int)
{
   var temp int
   temp = *x
   *x = *y
   *y = temp
}
```

2.4 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
}
```

2.5 BLANK IDENTIFIERS

- In some situations, some values need to be ignored for multiple reasons. GO language provide an identifier for this. If any variable is declared but not used or any import has the same issue, the Go compiler throws an error.
- **Blank Identifier**: The blank identifier is the single underscore (_) operator. It is used to ignore the values returned by functions or import for side-effects.

Need of Blank Identifier:

- Go compiler throws an error whenever it encounters a variable declared but not used. Now we can simply use the blank identifier and not declare any variable at all.

Uses of Blank Identifier:

- The blank identifier can ignore any value. The main use case for this identifier is to ignore some of the values returned by a function or for import side-effects.

2.5.1 Ignore Values

- The blank identifier ignores any value returned by a function. It can be used any number of times in a Go program. The code below shows how to use the blank identifier.

```
package main
import "fmt"
func abc() (int, int)
{
   return 42, 53
}
func main()
{
   v, _ := abc()    // no error from compiler
   fmt.Println(v) // 42
}
```

2.5.2 Side Effects of Import

- Sometimes, a package in Go needs to be imported solely for side effects i.e. initialization. The blank identifier if used before does that. It allows us to import the package without using anything from it. It also stops the compiler from throwing error messages like "**unused import**".

```
Import _ "fmt"
```

2.5.3 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
}
```

2.6 VARIABLE ARGUMENT PARAMETERS

- In Go, a function that can accept a dynamic number of arguments is called a Variadic function. Below is the syntax for variadic function. Three dots are used as a prefix before type.

  For example:

  ```
  func add(numbers ... int)
  ```

- The above function can be called with zero or more arguments.

  For example:

  ```
  add()
  add(1,3)
  add(1,4, 7,20 )
  ```

- The variadic param which is **numbers**, in this case, gets converted into a slice inside the function which can be iterated using **range.**

  For example:

  ```
  func add(numbers ... int )int
  {
     sum:=0
     for __,num:=range numbers
     {
        sum+=num
     }
  return sum
  }
  ```

- If you already have a slice and you need to pass it as a variadic param then that can be done by adding three dots (...) after the argument while calling the function.

  ```
  var numbers:=[]int{1,2,3}
  add( numbers ...)
  ```

- In case there is a need to pass variadic as well as non-variadic arguments to a function, then the non-variadic needs to be passed as initial arguments and variadic argument need to be the last.

```
func add(val string, numbers ...int)
```

- The best example of a variadic function used in GO library is the **fmt.Println()** function. This is the signature of the function will be

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

- While using variadic functions there can be a couple of cases related to the variadic argument.

2.6.1 Different Number of Parameters and of same type

- This case can be handled using both variadic function and empty interface.
- The above case can easily be handled using variadic functions. Notice in below code the parameter is of one type i.e. int.

For Example:

```
package main
import "fmt"
func main()
{
   fmt.Println(add(1, 2))
   fmt.Println(add(1, 2, 3))
   fmt.Println(add(1, 2, 3, 4))
}
func add(numbers ...int) int
{
   sum := 0
   for _, num := range numbers
   {
      sum += num
   }
   return sum
}
```

Output:

```
3
6
10
```

2.6.2 Different Number of Parameters and of different types

- This case can be handled using both variadic function and empty interface.

For Example:

```
package main
import "fmt"
func main()
{
    fun(1, "abc")
    fun("GO", "C++", 3)
    fun(1, 2, 3, 4)
}
func fun(params ...interface{})
{
    fmt.Println("Fun function called with parameters:")
    for _, param := range params
    {
        fmt.Printf("%v\n", param)
    }
}
```

Output:

```
Fun function called with parameters:
1
abc
Fun function called with parameters:
GO
C++
3
Fun function called with parameters:
1
2
3
4
```

- We can also use a switch case to get the exact parameters and use them accordingly.

For example:

```
package main
import "fmt"
type person struct
{
    name string
    gender string
    age int
}
```

```
func main()
{
   err := addPerson("Minal", "Female", 40)
   if err != nil
   {
      fmt.Println("PersonAdd Error: " + err.Error())
   }
   err = addPerson("Satish", "Male")
   if err != nil
   {
      fmt.Println("PersonAdd Error: " + err.Error())
   }
   err = addPerson("Pallawi", 2, 3)
   if err != nil
   {
      fmt.Println("PersonAdd Error: " + err.Error())
   }
}
func addPerson(args ...interface{}) error
{
   if len(args) > 3
   {
      return fmt.Errorf("Wrong number of arguments passed")
   }
   p := &person{}
   //0 is name
   //1 is gender
   //2 is age
   for i, arg := range args
   {
      switch i
      {
         case 0: // name
         name, ok := arg.(string)
         if !ok
         {
            return fmt.Errorf("Name is not passed as string")
         }
         p.name = name
         case 1:
            gender, ok := arg.(string)
            if !ok
            {
               return fmt.Errorf("Gender is not passed as string")
            }
            p.gender = gender
```

```
            case 2:
                age, ok := arg.(int)
                if !ok
                {
                    return fmt.Errorf("Age is not passed as int")
                }
                p.age = age
                default:
                return fmt.Errorf("Wrong parametes passed")
            }
        }
        fmt.Printf("Person struct is %+v\n", p)
        return nil
    }
```

Output:

```
Person struct is &{name:Minal gender:Female age:40}
Person struct is &{name:Satish gender:Male age:0}
PersonAdd Error: Gender is not passed as string
```

- Kindly note wherever the arg is not passed it is substituted as default.

2.7 USING DEFER STATEMENTS

- Defer is a special statement in Go. The defer statement schedules a function to be called after the current function has completed.
- Go will run all statements in the function then do the function call specified by defer after.
- Defer statement is used to execute a function call just before the surrounding function where the defer statement is present returns.
- In Go language, defer statements delay the execution of the function or method or an anonymous method until the nearby functions returns. In other words, defer function or method call arguments evaluate instantly, but they don't execute until the nearby functions returns. You can create a deferred method, or function, or anonymous function by using the defer keyword.

Syntax:

```
// Function
defer func func_name(parameter_list Type)return_type
{
    // Code
}
// Method
defer func (receiver Type) method_name(parameter_list)
{
    // Code
}
```

```
defer func (parameter_list)(return_type)
{
   // code
}
```

Kindly Note:

- In Go language, multiple defer statements are allowed in the same program and they are executed in LIFO(Last-In, First-Out) order as shown in following example(B).
- In the defer statements, the arguments are evaluated when the defer statement is executed, not when it is called.
- Defer statements are generally used to ensure that the files are closed when their need is over, or to close the channel, or to catch the panics in the program.
- Let us see above concepts with the help of:

Example (A):

```
package main
import "fmt"
func mul(a, b int) int
{
   res := a * b
   fmt.Println("Result: ", res)
   return 0
}
func show()
{
   fmt.Println("Hello, Go Programming")
}
func main()
{
   mul(2, 4) // mul function behaves like a normal function
   defer mul(10, 20)// mul function with defer keyword
   show()
}
```

Output:

```
Result:  8
Hello, Go Programming
Result:  200
```

- In above program we have two functions named mul() and show(). Where show() function is called normally in the main() function, mul() function is called in two different ways:
  1. We call mul() function normally (without the defer keyword), i.e, mul(2, 4) and it executes when the function is called (Output: Result : 8).
  2. We call mul() function as a defer function using defer keyword, i.e, defer mul(10, 20) and it executes (Output: Result: 200) when all the surrounding methods return.

Example (B):

```
package main
import "fmt"
func add(a, b int) int
{
   res := a + b
   fmt.Println("Result: ", res)
   return 0
}
func main()
{
   fmt.Println("Start")
   // Multiple defer statements Executes in LIFO order
   defer fmt.Println("End")
   defer add(2, 4)
   defer add(10, 20)
}
```

Output:

```
Start
Result: 30
Result: 6
End
```

2.8 RECURSIVE FUNCTIONS

- 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 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.

- In case of recursion programmer need to be careful about exit condition written in function. If exit condition is not written properly then program will go into infinite loop.

Program 2.1: Write a Program in GO language to calculate factorial of given number.

```
package main
import "fmt"
func fact(i int)int
{
   if(i<= 1)
   {
      return 1
   }
   return i * fact(i - 1)
}
func main()
{
   var n int
   fmt.Println("Enter the Number:")
   fmt.Scanf("%d",&n)
   fmt.Printf("Factorial of %d is %d", n, fact(n))
}
```

Output:

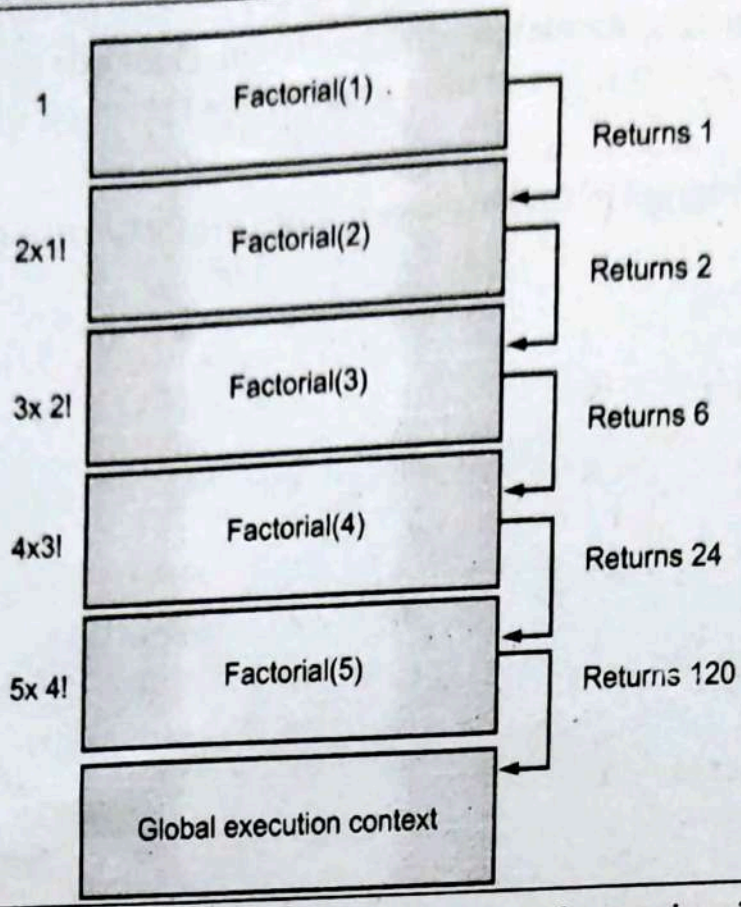
```
Enter the Number:
5
Factorial of 5 is 120
```

Example of factorial calculations:

$1! = 1$		$1! = 1$
$2! = 2 \times 1 = 2$		$2! = 2 \times 1! = 2$
$3! = 3 \times 2 \times 1 = 6$	OR	$3! = 3 \times 2! = 6$
$4! = 4 \times 3 \times 2 \times 1 = 24$		$4! = 4 \times 3! = 24$
$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$		$5! = 5 \times 4! = 120$

The concept of calculating the factorial of 5

Explanation how a recursion functions of factorial will get executed.



Program 2.2: Write a Program in GO Language to print Fibonacci series using recursion

```
package main
import "fmt"
func fibo(i int) (ret int)
{
    if i == 0
    {
        return 0
    }
    if i == 1
    {
        return 1
    }
    return fibo(i-1) + fibo(i-2)
}
func main()
{
    var n int
    fmt.Println("Enter the Number:")
    fmt.Scanf("%d",&n)
    for n = 0; n < 10; n++
    {
        fmt.Printf("%d ", fibo(n))
    }
}
```

What is better? Recursion or Iteration?

- Recursive code is short, elegant, and sometimes hard to understand and implement.

In case of efficiency:

- Iterative solutions are generally more efficient than recursive solutions. That's because recursion requires more memory - each recursive call consumes extra space on the stack frame. The potential of stack overflow problems will arise if the recursion is too deep or infinite.

How stack overflow problems can be avoided in Recursion in GO Language?

- In Golang, stacks grow as needed. It makes the effective recursion limits relatively large. The initial setting is 1 GB on 64-bit systems and 250 MB on 32-bit systems. The default can be changed by SetMaxStack in the runtime/debug package.
- (SetMaxStack sets the maximum amount of memory that can be used by a single goroutine stack. If any goroutine exceeds this limit while growing its stack, the program crashes.)

2.9 FUNCTIONS AS PARAMETERS

- User can pass function as parameter to a Go function. Following program illustrate this concept. fn1 and fn2 are two normal functions printing values of variables. testfunction is a function in which function can be passes as parameter. User has to pass values for function parameters first (if needed) then that function can be taken as parameter for another function.

For Example:

```
package main
import "fmt"
type fn func(int)

func fn1(i int)
{
   fmt.Printf("\ni is %v", i)
}
func fn2(i int)
{
   fmt.Printf("\ni is %v", i)
}
func testfunction(f fn, val int)
{
   f(val)
}
func main()
{
   testfunction(fn1, 10)
   testfunction(fn2, 30)
}
```

Summary

- A function is a group of statements performs some tasks. Every GO program has at least one function i.e. main().
- There are two types of parameters associated with function formal and actual parameters. Formal parameters represent actual parameters. Actual parameters are passed to function as an argument when it is called.
- In GO Language function can return multiple return values of same or different types.
- Go has excellent error handling support for multiple return values. We simply can return an error when doing any operation.
- Functions can be called by call by value or call by reference.
- The benefit of using the named return types is that it allows us to not explicitly return each one of the values.
- GO Language provides blank identifiers to ignore some value.
- In Go, a function that can accept a dynamic number of arguments is called a Variadic function.
- Defer is a special statement in Go. The defer statement schedules a function to be called after the current function has completed.
- User can pass function as parameter to a Go function.

Sample Programs for Study Purpose

1. Write a Program in go language to print addition of two number using functions.

```
package main
import "fmt"
func main()
{
    var a int
    var b int
    var c int
    fmt.Printf("Enter two numbers: ")
    fmt.Scanf("%d%d", &a, &b)
    c = add(a, b)
    fmt.Printf( "Addition value is: %d\n", c )
}
func add(num1, num2 int) int
{
    var result int
    result = num1 + num2
    return result
}
```

Output:

```
Enter two numbers: 2 4
Addition value is: 6
```

2. Write a Program in go language to print recursive sum of digits of given number.

```
package main
import "fmt"
func SumOfDigits(num int) int
{
   if(num == 0)
   {
      return 0
   }
   else
   {
      return ((num % 10) + SumOfDigits(num / 10))
   }
}
func main()
{
   var n int
   fmt.Printf("\nEnter your number: ")
   fmt.Scanf("%d",&n)
   fmt.Printf("Recursive Sum of Digit %d is %d", n, SumOfDigits(n))
}
```

Output:

```
Enter your number: 568
Recursive Sum of Digit 568 is 19
```

---

3. Write a Program in go language using function to check whether accepted number is palindrome or not.

```
package main
import "fmt"
func cpallindrome(num int )int
{
   var reversed, remainder, original int
   reversed = 0
   original = num
   for (num != 0)
   {
      remainder = num % 10;
      reversed = reversed * 10 + remainder;
      num /= 10;
   }
   if (original == reversed)
   {
      return 1
   }
```

```
    else
    {
        return 0
    }
}
func main()
{
    var n, r int
    fmt. Println("\nEnter the Number:")
    fmt.Scanf("%d",&n)
    r=cpallindrome(n);
    if(r==1)
    {
        fmt.Printf("The number %d is Pallindrome", n)
    }
    else
    {
        fmt.Printf("The number %d is not Pallindrome", n)
    }
}
```

Output:

```
Enter the Number:
4554
The number 4554 is Pallindrome
```

4. Write a Program in go language to swap two numbers using call by reference concept.

```
package main
import "fmt"
func main()
{
    var a, b int
    fmt. Println("\nEnter two numbers:")
    fmt.Scanf("%d%d",&a,&b)
    fmt.Printf("Before swap, value of a = %d b=%d \n", a,b )
    swap(&a, &b)
    fmt.Printf("After swap, value of a = %d b=%d \n", a,b )
}
func swap(x *int, y *int)
{
    var temp int
    temp = *x
    *x = *y
    *y = temp
}
```

Output:

```
Enter two numbers:
23 67
Before swap, value of a = 23 b=67
After swap, value of a = 67 b=23
```

5. Write a Program in go language to demonstrate use of names returns variables.

```
package main
import "fmt"
func main()
{
   // calling the function, here function returns two values m, d :=
   calculator(105, 7)
   fmt.Println("105 x 7 = ", m)
   fmt.Println("105 / 7 = ", d)
}
// function having named arguments
func calculator(a, b int) (mul int, div int)
{
   // here, simple assignment will initialize the values to it
   mul = a * b
   div = a / b
   // here you have return keyword without any resultant parameters
   return
}
```

Output:

```
105 x 7 =  735
105 / 7 =  15
```

6. Write a Program in go language to show the compiler throws an error if a variable is declared but not used.

```
package main
import (
    "fmt" // imported and not used: "fmt"
)
func main()
{
   i := 1 // i declared and not used
}
```

Output:

```
# command-line-arguments
./main.go:10:5: imported and not used: "fmt"
```

7. Write a Program in go language to illustrate the concept of call by value.

```
package main
import "fmt"
// function which swap values
func swap(a, b int)int
{
    var t int
    t= a
    a=b
    b=t
    return t
}
func main()
{
    var p int = 10
    var q int = 20
    fmt.Printf("p = %d and q = %d", p, q)
    // swap function call by values concept
    swap(p, q)
    fmt.Printf("\np = %d and q = %d",p, q)
}
```

Output:

```
p = 10 and q = 20
p = 10 and q = 20
```

8. Write a Program in go language to illustrate the concept of returning multiple values from a function.

```
package main
import "fmt"
// myfunction return 3 values of int type
func myfunction(a, b int)(int, int, int)
{
    return a - b, a * b, a + b
}
func main()
{
    // The return values are assigned into three different variables
    var v1,v2,v3 int
    fmt.Println("Enter three values:")
    fmt.Scanf("%d%d%d",&v1,&v2,&v3)
    v1, v2, v3 = myfunction(v1, v2)
    // Display the values
```

```
    fmt.Printf("Result is: %d", v1 )
    fmt.Printf("\nResult is: %d", v2)
    fmt.Printf("\nResult is: %d", v3)
}
```

Output:

```
Enter three values:
2 8 99
Result is: -6
Result is: 16
Result is: 10
```

9. Write a Program in go language to find the largest and smallest number in an array.

```
package main
import "fmt"
func main()
{
   var s, i  int
   fmt.Print("Enter the Array Size to find Smallest & Largest = ")
   fmt.Scan(&s)
   Arr1 := make([]int, s)
   fmt.Print("Enter the Array Items  = ")
   for i = 0; i< s; i++
   {
      fmt.Scan(&Arr1[i])
   }
   lar := Arr1[0]
   sma :=Arr1[0]
   for i = 0; i< s; i++
   {
      if lar < Arr1[i]
      {
         lar = Arr1[i]
      }
      if sma> Arr1[i]
      {
         sma = Arr1[i]
      }
   }
   fmt.Println("\nThe Largest Number in Array    = ", lar)
   fmt.Println("\nThe Smallest Number in Array   = ", sma)
}
```

Output:

```
Enter the Array Size to find Smallest & Largest = 5
Enter the Array Items = 34 56 78 9 1
The Largest Number in Array = 78
The Smallest Number in Array = 1
```

10. Write a Program in go language to accept the book details such as BookID, Title, Author, Price. Read and display the details of n number of books. (Not running well)

```
package main
import "fmt"
type Book struct
{
    BID int
    Btitle, Bauthor string
    Bprice float64
}
func main()
{
    var n,i int
    var b1[10] Book
    fmt.Println("Enter The value of n:")
    fmt.Scanf("%d",&n)
    for i=0;i<n;i++
    {
        fmt.Println("\nEnter Book ID:")
        fmt.Scanf("%d",&b1[i].BID)
        fmt.Println("\nEnter Book Title:")
        fmt.Scanf("%s",&b1[i].Btitle)
        fmt.Println("\nEnter Book Author:")
        fmt.Scanf("%s",&b1[i].Bauthor)
        fmt.Println("\nEnter Book Price:")
        fmt.Scanf("%f",&b1[i].Bprice)
    }
    for i=0;i<n;i++
    {
        fmt.Println("Book: ", b1[i])
    }
}
```

Output:

```
Enter The value of n:
2

Enter Book ID:
11
```

```
Enter Book Title:
Java
Enter Book Author:
abc
Enter Book Price:
90
Enter Book ID:
12
Enter Book Title:
OOSE
Enter Book Author:
xyz
Enter Book Price:
80
Book:  {11 Java abc 90}
Book:  {12 OOSE xyz 80}
```

With make method:

```
package main
import "fmt"
func main()
{
   type book struct
   {
      BookID int
      Title string
      Author string
      Price float32
   }
   var n, i int;
   fmt.Println("Enter value of n")
   fmt.Scan(&n)
   var Bookarr=make([]book,n);
   for i = 0; i< n; i++
   {
      fmt.Println("Enter details (Book Id, Title, Author, Price) of
      book",i)
      fmt.Scan(&Bookarr[i].BookID,&Bookarr[i].Title,&Bookarr[i].Author,
                                                    &Bookarr[i].Price)
   }
```

```
    for i = 0; i< n; i++
    {
        fmt.Println("Books details (Book Id, Title, Author, Price) of Book ",i);
        fmt.Println("BookID=",Bookarr[i].BookID)
        fmt.Println("Title=",Bookarr[i].Title)
        fmt.Println("Author=",Bookarr[i].Author)
        fmt.Println("Price=",Bookarr[i].Price)
    }
}
```

Output:

```
Enter value of n
2
Enter details (Book Id, Title, Author, Price) of book 0
11
C++
abc
70
Enter details (Book Id, Title, Author, Price) of book 1
12
C
xyz
80
Books details (Book Id, Title, Author, Price) of Book  0
BookID= 11
Title= C++
Author= abc
Price= 70
Books details (Book Id, Title, Author, Price) of Book  1
BookID= 12
Title= C
Author= xyz
Price= 80
```

Check Your Understanding

1. A function can return _____ values.
   (a) Many (b) cannot say
   (c) -1 (d) infinity
2. Go compiler throws an error whenever it encounters a variable _____ but not used.
   (a) Initialized (b) not declared
   (c) declared (d) no initialized

3. Every go program has at least _____ function.
   (a) 0 (b) 9
   (c) -1 (d) 0
4. func is a _____ in Go language.
   (a) Identifier (b) keyword
   (c) Constant (d) Parameter
5. _____ parameters are representing actual parameters.
   (a) Constants (b) Keywords
   (c) Formal (d) identifiers
6. _____ parameters are passed to the function from where it has been called as an argument.
   (a) Actual (b) Keywords
   (c) Formal (d) identifiers
7. A Go function can return _____ values
   (a) Single (b) multiple
   (c) negative (d) positive
8. Multiple return values reduce the needs of returning _____ to data.
   (a) Pointers (b) identifiers
   (c) keywords (d) constants
9. Go language uses _____ which can be used in place of the data and the data will be skipped completely.
   (a) hyphen (b) semicolon
   (c) colon (d) inverter comma
10. The blank identifier in GO Language is the single _____ operator.
    (a) underscore (b) semicolon
    (c) colon (d) constant
11. The blank identifier can _____ any value.
    (a) predict (b) add
    (c) multiple (d) ignore
12. The blank identifier helps to _____ the code.
    (a) run (b) trace
    (c) debug (d) execute
13. In Go language a function that can accept a dynamic number of arguments is called a _____ function.
    (a) simple (b) normal
    (c) recursive (d) Variadic

14. The _____ statement schedules a function to be called after the current function has completed.
    (a) Constant (b) identifier
    (c) keyword (d) defer

Answers

1. (a)	2. (c)	3. (a)	4. (b)	5. (c)	6. (a)	7. (b)	8. (a)	9. (a)	10. (a)
11. (d)	12. (c)	13. (d)	14. (d)						

Trace The Output

1.
```
package main
import "fmt"
func funify(word string) string
{
    return word + " is really fun!"
}
func main()
{
    fmt.Println(funify("cheese"))
}
```
Ans.: cheese is really fun! is printed.
Explanation: The funify() function takes a string as an argument and returns a string.
funify() appends the "is really fun" string to the argument that's invoked with the function.

2.
```
func add(a int, b int) int
{
    return a + b
}
func main()
{
    fmt.Println(add(3, 5))
}
```
Ans.: 8 is printed.
Explanation: The add() function adds two integers.

3. What does the following code print?
(i)
```
func average(nums []float64) float64
{
    total := 0.0
    for _, v := range nums
    {
        total += v
```

```
    }
    return total / float64(len(nums))
}
func main()
{
    scores := []float64{10, 11, 12, 13}
    fmt.Println(average(scores))
}
```

Ans.: 11.5 is printed.

Explanation: The average() function loops through a slice of floats and calculates the sum of all the numbers. The function returns the sum divided by the total number of elements in the slice, or the average.

The float64() function is used to convert an integer into a floating point number. You can only divide a floating point number by another floating point number.

(ii)
```
func product(a int, b int) (res int)
{
    res = a * b
    return
}
func main()
{
    fmt.Println(product(5, 3))
}
```

Ans.: 15 is printed.

Explanation: The product() function has a named return type.

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

Naked returns should only be used in short functions.

(iii)
```
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(2)
    fmt.Println(x, y)
}
```

Ans.: 6 -2

Practice Questions

Q. I Answer the following questions in short.

1. What is the default way of passing parameters to a function?
2. What do you mean by functions as value in GO?
3. What are the function closures?
4. What is default value of variable in GO?
5. Explain the purpose of printf function in go.

Q. II Answer the following questions.

1. Explain the syntax to create a function in GO.
2. Can user return multiple values from a function? How?
3. In how many ways user can pass parameters to a function? Explain.
4. Explain the concept function returning multiple values.
5. "Sometimes there is need to skip return values" explain with example.
6. When recursion is better than iteration?
7. When iteration is better than recursion?
8. Explain concept of function with simple example.
9. Explain the concept of formal and actual parameters in function with suitable example.
10. Briefly explain the concept of same return types in function with suitable example.
11. Briefly explain the concept of different return types in function with suitable example.
12. What are different benefits of multiple return values in function?
13. What is the need for skipping return value in function?
14. Explain with example call by value concept.
15. Explain with example call be reference concept.
16. What do you mean by named return variables?
17. What is the need and use of blank identifier?
18. Explain with example use of defer statement in GO language.
19. What are advantages and disadvantages of recursion?
20. Does recursion is better over iteration?
21. How to pass functions as parameters in GO Language?

Q. III Define the terms.

1. Function.
2. Formal Parameters.
3. Actual Parameters.
4. Blank Identifiers.
5. Recursion.

■■■

3... Working with Data

Learning Objectives ...

Students will be able to:

- Learn different array concepts and single and multidimensional arrays in Go programming Language.
- Learn different ways of slicing for single and multidimensional arrays.
- Learn structures and way of writing programmes using structures in GO Language.

3.1 ARRAY LITERALS

- An **array** is a fixed-length sequence of data items of the same type. Array is homogeneous type of data structure. This type can be anything from primitive types like integers or strings to self-defined types. The length of an array is always a constant expression that always evaluates a non-negative integer value.
- Go language arrays holds fixed number of elements and they can grow or shrink to any size. Array elements are always stored sequentially and accessed by their index value. Array indexing always starts from zero and means first array element is at index 0 position then the second element is at 1st position and so on so forth.
- The number of items in an array is called as array length (len) or size of an array. The length of an array is fixed when array is declared i.e. at compile time. The memory is allocated to array at compile time.

Syntax for array declaration:

```
var identifier [len] type
```

or

```
var array_name[length]Type{item1, item2, item3, ...itemN}
```

Example:

```
Create an array arr of int type and can hold 3 values.
var arr[3] int
```

Program 3.1: Write a program in go to declare an array of 3 elements and print it.

```
package main
import "fmt"
func main() {
    var a [3]int
    fmt.Println(a)
}
```

Output:

```
[0 0 0]
```

Note: All array elements are initialized to value zero by default automatically to individual element.

Assigning value to an Array Element:

- In Go Language arrays are declared by individual elements.
  Syntax: Assigning a value to an array-item at index i, is done with:
  ```
  arr[i]=value
  ```
  Arrays are mutable, which means that they can be changed.
- Only valid array indexes can be used. If user will give array index greater than or equal to len(arr) then compiler throws an error message that **index out of bounds**. Or otherwise runtime error will be thrown as **index out of range.**
- **Panic:** A runtime error in programming language means the program will crash and give a certain message. In Go terminology, this is called a **panic.**

Multiple Assignments in an Array:

- Instead of assigning one value at a time in a single line in an array, user can assign multiple values in a single line.

Program 3.2: Write a program in GO to declare an array of 5 elements to some predefined value and print it.

```
package main
import "fmt"
func main() {
    x := [5]int{10, 20, 30, 40, 50}
    fmt.Println(x)
}
```

Output:

```
10 20 30 40 50
```

Program 3.3: Write a program in GO Language to print the exact type of the array, as to print the length as part of the array itself.

```
package main
import {
    "fmt"
    "reflect"
}
```

```
func main() {
    var a [3]int
    fmt.Println(a)
    fmt.Println(reflect.TypeOf(a))
}
```

Output:

```
[0 0 0]
[3]int
```

Program 3.4: Write a program to assign values to array elements and print value at second index.

```
package main
import "fmt"
func main() {
    var a [3]int
    a[0] = 1
    a[1] = 2
    a[2] = 3
    fmt.Println(a)
    fmt.Println(a[2])
}
```

Output:

```
[1 2 3]
3
```

Array Literals:

- In Go Language, as user can declare an array and can also set the entries in the array as well with the help of the literal values. Literal values are those values that are passed when declaring an array. This approach of array declaration is only useful when user know what exact elements user wants in his array and how many of them are there in total.

  For example: Array depicts an integer array where we are making use of the literals to declare the array and set the values as well.

```
package main
import "fmt"
func main() {
    a := [3]int{1,2,3}
    fmt.Println(a)
    stringA := [3]string{"College","Go","Language"}
    fmt.Println(stringA)
}
```

  Output:

```
[1 2 3]
[College Go Language]
```

- In above example we declared two literal arrays, where one holds the integer values and the other holds the string values and then we are printing those arrays.

Arrays with implicit Length:

- In Go language user can provide the implicit length of the array with the help of **ellipsis** which is also a valid syntax for array declaration.
- Ellipsis is nothing but **three dots(...)** which can replace the array length keyword when we are declaring an array.

Program 3.5: Write a program to declare an array (Integer and array of strings) with implicit length via ellipsis.

```
package main
import "fmt"
func main() {
    a := [...]int{1,2,3} //integer array
    fmt.Println(a)
    b := [...]string{"Go","Programming","Language"}//  array of strings
    fmt.Println(b)
}
```

Output:

```
[1 2 3]
[Go Programming Language]
```

- In case of ellipsis array is printed without the length property. In case of normal number of elements in array declaration will be used to print an array.

Printing Array Elements:

- Println function is used to print array elements. Printf function is also used to print array elements. In Printf %v is used for value. Whenever user is not sure what type of values an array is holding then make use of the **%v** verb to print the values of the array.

For Example:

```
package main
import "fmt"
func main() {
    a := [...]int{1,2,3}
    fmt.Println(a)
    fmt.Printf("The array is %v\n", a)
}
```

Output:

```
[1 2 3]
The array is [1 2 3]
```

Printing Arrays Elements using Loop:

- There are three ways to print array elements as follows:
  1. Use of simple for loop.
  2. Use of range clause in for loop (which acts like while loop).
  3. Reflects package function.

Program 3.6: Write a program to print array elements using all three techniques.

```
package main
import {
   "fmt"
   "reflect"
}
func main() {
   a := [...]int{1, 2, 3}// a is integer array
   for i := 0; i<len(a); i++ {
      fmt.Print(a[i], " ")
   }
   fmt.Println()
   var i int
   for i<len(a) {
      fmt.Print(a[i], " ")
      i++
   }
   fmt.Println()
   for _, val := range a {
      fmt.Print(val , " ")
   }
   fmt.Println(reflect.ValueOf(a))
}
```

Output:

```
1 2 3
1 2 3
1 2 3 [1 2 3]
```

Reflect Package TypeOf() and Kind() on Arrays:

- Sometimes it is necessary to know what kind of values is present in an array i.e. what type of array it is.

Arrays are Value type:

- In Go Language, when arrays are assigned to a new variable then instead of passing address of an array user can pass copy of the array. User can make any changes to new array. The changes to new array will not reflect the previous or original array at all.
- Following example will clear the idea. Array a will contain 3 integer values. Array a will be assigned to new variable b.

```
package main
import    "fmt"
func main() {
   a := [...]int{1, 2, 3}
```

```
    b := a //array a is assigned to array b
    println("The Original array a is...")
    fmt.Println(a)
    println("The copied array b is...")
    fmt.Println(b)

    b[0] = 100
    // zeroth index value of array b is changed which does not reflected to
    array a
    println("The Original array a is...")
    fmt.Println(a)
    println("The copied array b is...")
    fmt.Println(b)
}
```

Output:

```
The Original array a is...
[1 2 3]
The copied array b is...
[1 2 3]
The Original array a is...
[1 2 3]
The copied array b is...
[100 2 3]
```

Filtering Array Values:

- Due to indexing techniques, we can extract the values from the array. This is known as filtering array values. For this starting, ending index, or any one can be used. Colon can be used to separate starting index and ending index. Starting index is included and ending index is excluded.
- There are three cases:
  1. Only starting index is there - in this situation user will get all the elements from the starting index (including) to the last element of the array.
  2. Only ending index is there - in this situation user will get all the elements from the starting element of the array to the ending index (excluding).
  3. No starting index and no ending index - in this situation user will get entire array.
- When user will filter array values, he will get a slice in return and he can also confirm that by using the TypeOf() function of the reflect package.

For example:

```
package main
import ("fmt"
   "reflect"
)
```

```
func main() {
    a := [...]int{1, 2, 3, 4, 5, 6}
    b := a[:2]
    c := a[1:3]
    d := a[2:]
    e := a[:]
    fmt.Println(a, reflect.TypeOf(a))
    fmt.Println(b, reflect.TypeOf(b))
    fmt.Println(c, reflect.TypeOf(c))
    fmt.Println(d, reflect.TypeOf(d))
    fmt.Println(e, reflect.TypeOf(e))
}
```

Output:

```
[1 2 3 4 5 6] [6]int
[1 2] []int
[2 3] []int
[3 4 5 6] []int
[1 2 3 4 5 6] []int
```

3.2 MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays are basically arrays containing others arrays as elements.

  Syntax: `[len1][len2].....[lenN] T {}`

  len1, len2 and lenN are length of each dimensions, T is data type of an array.
- All rules applied to one dimensional array are applied to multidimensional array also. It is also possible to specify the array elements during the declaration. In case, the array elements are not specified during declaration, then all the array elements are allocated the default zero value of the **<data_type>**.

Two- dimensional array:

Example: 2-dimensional array is declared with the help of array literals.

```
package main
import "fmt"
func main() {
    a := [2][3]int{{10, 20, -3}, {4, 8, 16}}// a is two dimensional array
    fmt.Println("Rows in array a are:", len(a))
    fmt.Println("Columns in array a are:", len(a[0]))
    fmt.Println("Total elements in array a are:", len(a)*len(a[0]))
    // Traversing the array
    for _, row := range a {
        for _, val := range row {
```

```
            fmt.Print(val," ")
        }
    }
}
```

Output:

```
Rows in array a are: 2
Columns in array a are: 3
Total elements in array a are: 6
10 20 -3 4 8 16
```

Note: Negative indexing cannot be used in an array. If user can do this with a constant or a literal index, it is a compile-time error.

Following example will show invalid array index.

```
package main
import "fmt"
func main() {
   A := [2][3]int{{11, 12, 83}, {4, -5, 6}}
   fmt.Println(A[1][4])
}
```

Output:

```
# command-line-arguments
.\sadguru.go:7:18: invalid array index 4 (out of bounds for 3-element
array)
```

3.3 ARRAY PARAMETERS

- Arrays can be passed as parameters to function. For this create a formal parameter using following syntax:

```
// For sized array
func function_name(variable_name [size]type){
// Code
}
```

- Using this syntax user can pass 1 or multiple dimensional arrays to the function.
- There are two ways to do this task.

1. Formal parameters as a sized array as follows:

```
void Function(param [10]int)
{
    .
    .
    .
}
```

2. Formal parameters as an unsized array as follows:

```
void Function(param []int)
{
    .
    .
    .
}
```

Program 3.7: Write a program to illustrate the concept of array parameters.

```
package main
import "fmt"
// This function accept an array as an argument
func fun(a [3]int, size int) int {
   var i, val, r int
   for i = 0; i< size; i++ {
      val += a[i]
   }
   r = val / size
   return r
}
func main() {
   var arr = [3]int{1, 9, 2}  // Creating and initializing an array
   var res int
   res = fun(arr, 3)          // Passing an array as an argument to
         function             name fun
   fmt.Printf("Final result is: %d ", res)
}
```

Output:

```
Final result is: 4
```

3.4 SLICES AND SLICE PARAMETERS

- Slice is a variable-length sequence which stores elements of a similar type users are not allowed to store different type of elements in the same slice. It is just like an array having an index value and length, but the size of the slice is resized they are not in fixed-size just like an array.
- Internally, slice and an array are connected with each other, a slice is a reference to an underlying array. It is allowed to store duplicate elements in the slice. The first index position in a slice is always 0 and the last one will be (length of slice – 1).
- Following points will explain the slices of array and its parameters.

1. **Declare an Array Variable with a specific size:**

```
var str [5]string    //character array str(string) of having 5
                     characters length
var b[5] int//integer array b having size 5
```

2. **Assign a value to a specific element in an array:**

```
Assigning values to array str
str[0]= "Learning"
str[1]= "Go"
str[2]= "Programming"
str[3]= "Language"
str[4]= "With Examples"
Assigning values to integer array b
b[0]=10
b[1]=20
b[2]=30
b[3]=40
b[4]=50
```

Note: Array indexing starts from 0. str[0] or b[0] refers to the first element in an array and str[4] or b[4] refers to the last element of an array. If user will try to assign value to array index beyond its size then it will an error message i.e. user cannot add new element to an existing array.

For example: `str[5]= "New"`

Above statement will generate an error message as invalid array index 5(out of bounds for 5 element array).

3. **Initialize a Slice using Multi-Line Syntax:**

- For better clarification and readability purpose multiline syntax can be used while initializing and assigning values to array elements.

Program 3.8: Write a program in go language to initialize a Slice using Multi-Line Syntax and display.

```
package main
import "fmt"
func main() {
    a := [4] int {
        9,
        65,
        82,
        0,
    }
    fmt.Println(a)
}
```

Output:

```
[9 65 82 0]
```

Note: In multi-line syntax comma should be specified after each element. One advantage of using this syntax is that user can comment any value in the slice without removing a comma. Let us consider above example. Last value is commented and do not worry about removing comma removing the comma after last element.

Example:

```
package main
import "fmt"
func main() {
   a := [4] int {
      9,
      65,
      82,
      //0,
   }
   fmt.Println(a)
}
```

Output:

```
[9 65 82 0]
```

Another example

```
package main
import "fmt"
func main() {
   str := []string {
      "Radha",
      "Pallawi",
      "Minal",
      // "Kusum",
   }
fmt.Println(str)
}
```

Output:

```
[Radha Pallawi Minal]
```

4. **Assign a Value to a Specific Element in a Slice:**

- Assigning a value to a slice is really assigning value to the array that is behind the slice.

  For example:

  ```
  str[0]= "Satish"
  b[0]=12
  ```

5. **Access a particular Element in a Slice:**

- := will declare and initialize the variable at the same time. Using := operator any particular element in a slice can be accessed.

```
P :=str[2]//accessing third element
Q :=b[1]//accessing first element
```

6. **Display All or Specific Elements from a Slice:**

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

Output: The above statements will display all the elements inside square parentheses.

```
["Radha","Pallawi","Minal"]
[10,20,30,40,50]
```

The above two statements will display all the elements form an array.

```
fmt.Println("second element=",str[1])
fmt.Println("second element=",b[1])
```

Output:

```
Pallawi
20
```

Note: While using Println make sure that fmt package is included.

7. **Create Slice with Low and High values using Colon Syntax:**

Syntax:

```
[low:high]
low: It indicates where the slice starts.
high: It indicates where the slice ends(i.e. high-1).
P:=str[0:3] // 0 indicate low value 3 indicates high value
fmt.Println(P)
```

Output:

```
["Radha","Pallawi","Minal"]
P:=str[0:2] // 0 indicate low value 2 indicates high value
fmt.Println(P)
```

Output:

```
["Radha","Pallawi"]
```

Note: low and high are optional and default value of low is zero. Default high value is same as length of an array.

- Following statements are same:

```
P:=[:]
P = str[0:len(str)]
fmt.Println(P)// will print ["Radha","Pallawi","Minal"]
```

```
P = str[0:3]
fmt.Println(P)//will print ["Radha","Pallawi","Minal"]
User can either specify low or high value like the following.
P:=str[:3]// low will be taken as zer0
P:=str[0:]// high value is array length -1
```

- So all of the following are same:

```
str[0:5]
st[:5]
str[0:]
str[:]
```

8. **Slice Refers to an Array underneath:**

- Implication of Changing Slice Element Value.
- Changing a value of an element is slice changes the value of the original array. Because this is sharing the copy of the value and get reflected into original array.

```
P := str[0:2]
P[0] = "Kusum"
fmt.Println(P) //second array
fmt.Println(str)//original array
Q := str[0:3]
Q[1]="Priya"
fmt.Println(P) //second array
fmt.Println(str)//original array
```

Output:

```
[Kusum Pallawi]
[Kusum Pallawi Minal]
[Kusum Priya]
[Kusum Priya Minal]
```

9. **Slice of Structs:**

- User can create slice of struct and it can be initialized at the same time.

For example:

```
student := []struct
{
    id int
    name string
    vaccinated bool
}
{
    {10, "Pallawi", true},
    {30, "Minal", false},
    {50, "Shreya", true},
}
```

10. **Identify Length of a Slice:**

- len function is used to return total number of elements in size.

```
l := len(str)
fmt.Println("Length of string=",l)
```

Output:

```
Length of string= 3
```

11. **Slice Capacity: Length vs Capacity**

- A slice has both length and capacity:
  - **Length:** It indicates number of elements in a slice.
  - **Capacity:** Total number of elements in the slice's underlying array from the first element in the slice.
  - For the same slice, both the length and the capacity can be same or different.
- Both of the lines below are exactly the same with length 5 and capacity 5.

```
str = make([]string, 5)
str = make([]string, 5, 5)
str = make([]string, 1, 5)// capacity of str array is 5 and length is 1
```

- To find out length of the slice len function is used.

```
fmt.Println("length = ", len(str))
```

Output:

```
length =  3
```

- To find out capacity of a slice cap function is used.

```
fmt.Println("capacity = ", cap(str))
capacity= 3
```

12. **Empty Slice without an Array Underneath:**

- A nil slice has a length and capacity of 0 and has no underlying array as shown below.

```
var S1 []string
if S1 == nil
{
   fmt.Println("Slice S1 is empty")
}
```

Output:

```
Slice S1 is empty
```

- The following is not a nil slice, as we have allocated an array, eventhough the length and capacity is 0.

```
S2 := make([]string, 0)
if S2 != nil
{
   fmt.Println("S2 is not empty")
}
```

Output:

```
S2 is not empty
```

13. **Append one or more Elements to an existing Slice:**

- Append function is used to append the elements to existing slice.
- Following statement will append(add) one more element to slice str and assign the result to S1.

```
S1 := append(str, "Satish")
fmt.Println(S1)
```

Note: New element will be appended to str slice and it will get assigned to a variable S1.

Program 3.9: Write a program to illustrate the use of append function.

```
package main
import "fmt"
func main() {
   str := []string {
      "Radha",
      "Pallawi",
      "Minal",
      "Kusum",
   }
   fmt.Println(str)
   S1 := append(str, "Satish")
   fmt.Println(S1)
   fmt.Println(str)
}
```

Output:

```
[Radha Pallawi Minal Kusum]
[Radha Pallawi Minal Kusum Satish]
[Radha Pallawi Minal Kusum]
```

- User can append to same slice as well.

Example:

```
package main
import "fmt"
func main() {
   str := []string {
      "Radha",
      "Pallawi",
      "Minal",
      "Kusum",
   }
   fmt.Println(str)
   str = append(str, "Satish")
   fmt.Println(str)
}
```

Output:

```
[Radha Pallawi Minal Kusum]
[Radha Pallawi Minal Kusum Satish]
```

- User can add any number of elements to existing slice as well.

For example:

```
package main
import "fmt"
func main() {
    str := []string {
        "Radha",
        "Pallawi",
        "Minal",
        "Kusum",
    }
    fmt.Println("Before Append")
    fmt.Println(str)
    str = append(str, "Satish","Niket","Dipali")
    fmt.Println("After Append")
    fmt.Println(str)
}
```

Output:

```
Before Append
[Radha Pallawi Minal Kusum]
After Append
[Radha Pallawi Minal Kusum Satish Niket Dipali]
```

- **Copy elements from one Slice to another (Duplicate Slice):**

Copy function is used to copy the elements from one slice to another as shown below.

```
package main
import "fmt"
func main() {
    str := []string {
        "Radha",
        "Pallawi",
        "Minal",
        "Kusum",
    }
    S1 := make([]string, len(str))
    copy(S1, str)//copy str into S1. S1 is duplicate slice
    fmt.Println("Original Array")
    fmt.Println(str)
    fmt.Println("Slice S1")
    fmt.Println(S1)
}
```

Output:

```
Original Array
[Radha Pallawi Minal Kusum]
Slice S1
[Radha Pallawi Minal Kusum]
```

Two Dimensional Slices:

- Go's arrays and slices are one-dimensional. To create the equivalent of a 2D array or slice, it is necessary to define an array-of-arrays or slice-of-slices. Following program illustrates the array of (4 x 2 multidimensional array)

```
package main
import "fmt"
func main() {
   count := 1
   var a = make([][]int, 4) // number of rows
   for i := 0; i< 4; i++ {// loop number of rows
      a[i] = make([]int, 2)// each row two column
      for j := 0; j < 2; j++ {// loop for number of columns
         a[i][j] = count
         count++
      }
   }
   fmt.Println(a)
}
```

Output:

```
[[1 2] [3 4] [5 6] [7 8]]
```

Loop through Slice Elements using For and Range:

- This is similar to loop through elements in an Array using for and range as shown below.

```
package main
import "fmt"
func main() {
   count := 1
   var a = make([][]int, 4) // number of rows
   for i := 0; i< 4; i++ {// loop number of rows
      a[i] = make([]int, 2)// each row two column
      for j := 0; j < 2; j++ {// loop for number of columns
         a[i][j] = count
         count++
      }
   }
```

```
    for index, value := range a {
        fmt.Println(index, " = ", value)
    }
}
```

Output:

```
0  =  [1 2]
1  =  [3 4]
2  =  [5 6]
3  =  [7 8]
```

- **range str:** range is a go command which will return the index number and the value of an element.
- User can assign the output of the range command to a for loop and loop through all the elements.
- **for i, value:** Here specify two variable to the for command. The variable name can be anything. It just happened for clarity we are using index and value as the variable name here.

Loop through a Slice and get only the Values (Ignore Slice Index):

- When user want to use the array element and not the index, then specifying the index variable as shown in following program will give error message.
- So the correct program will be as follows:

```
package main
import "fmt"
func main() {
    total := 0
    ids := make([]int, 5)
    /*for _ index, value := range ids {
        total = total + value
    }*/
    for _, value := range ids {
        total = total + value
    }
    fmt.Println("total of all ids = ", total)
}
```

Output:

```
total of all ids =  0
```

3.5 MULTIDIMENSIONAL SLICES

- It is nothing but multidimensional array except that slice does not contain the size.

Syntax:

```
[len1][len2][len3]....[lenN]T{}
```

Where

`len 1,len 2,… ,len N are length of each dimensions.`

T is the data type.

- All the rules that apply to the one-dimensional array also apply to the multidimensional array as well. It is also possible to specify the array elements during the declaration. In case the array elements are not specified during declaration, then all the array elements are allotted the default zero value of the **<data_type>**.
- Format for declaring two dimensional array

```
var sample = [len1][len2]T{{a11, a12 .. a1y},
   {a21, a22 .. a2y},
   {.. },
   {ax1, ax2 ..axy}}
```

Where

- **len1** denotes the number of rows.
- **len2** denotes the number of columns.
- **aij** denotes an element present at i row and j column.
- **T** is the data type.

Program 3.10: Write a program in Go language to print 2-Array.

```
package main
import "fmt"
func main() {
   a := [3][2]int{{1, 2}, {3,4},{ 5, 6}}
   fmt.Printf("Number of rows in array: %d\n", len(a))
   fmt.Printf("Number of columns in array: %d\n", len(a[0]))
   fmt.Printf("Total number of elements in array: %d\n", len(a)*len(a[0]))
   fmt.Println("Traversing Array")
   for _, row := range a {
      for _, val := range row {
         fmt.Println(val)
      }
   }
}
```

Output:

```
Number of rows in array: 3
Number of columns in array: 2
Total number of elements in array: 6
Traversing Array
1
2
3
4
5
6
```

Same program in another way

```
package main
import "fmt"
func main() {
    // Creating multi-dimensional slice
    a := [][]int {
        {1, 2},
        {3, 4},
        {5, 6},
        {7, 8},
    }
    // Accessing multi-dimensional slice
    fmt.Println("Slice 1 : ", a)
}
```

Output:

```
Slice 1 :  [[1 2] [3 4] [5 6] [7 8]]
```

2D array of string

Program 3.11: Write a program to create multidimensional slice using 2D array of string.

```
package main
import "fmt"
func main() {
    // Creating multi-dimensional slice
    str1 := [][]string {
        []string{"C Prog", "FYBCS"},
        []string{"CPP Prog", "SYBCS"},
        []string{"Go Prog", "TYBCS"},
    }
    // Printing every slice inside str1
    fmt.Println("MultiDimensional Slice str1:")
        for i := 0; i<len(str1); i++ {
        fmt.Println(str1[i])
    }
    // Printing elements in slice as matrix
    fmt.Println("Slice str1 Like Matrix:")
    // number of rows in str1
    n := len(str1)
    // number of columns in str1
    m := len(str1[0])
    for i := 0; i< n; i++ {
        for j := 0; j < m; j++ {
```

```
            fmt.Print(str1[i][j] + " ")
        }
        fmt.Print("\n")
    }
}
```

Output:

```
MultiDimensional Slice str1:
[C Prog FYBCS]
[CPP Prog SYBCS]
[Go Prog TYBCS]
Slice str1 Like Matrix:
C Prog FYBCS
CPP Prog SYBCS
Go Prog TYBCS
```

Accessing elements of a multi-dimensional array:

- An element of a multi-dimensional array can be accessed using the index of each of the dimensions.

Traversal of a multidimensional array:

- Multidimensional arrays can be traversed using for range loop and for loop.

Program 3.12: Write a program to display array using for range and for loop.

```
package main
import "fmt"
func main() {
    a := [2][3]int{{10, 20, 30}, {40, 50, 60}}
    fmt.Println("Using for-range")
    for _, row := range a {
        for _, val := range row {
            fmt.Println(val)
        }
    }
    fmt.Println("\nUsing for loop")
    for i := 0; i< 2; i++ {
        for j := 0; j < 3; j++ {
            fmt.Println(a[i][j])
        }
    }
    fmt.Println("\nUsing for loop - Second way")
    for i := 0; i<len(a); i++ {
        for j := 0; j <len(a[i]); j++ {
            fmt.Println(a[i][j])
        }
    }
}
```

Output:

```
Using for-range
10
20
30
40
50
60

Using for loop
10
20
30
40
50
60

Using for loop - Second way
10
20
30
40
50
60
```

Storage of multidimensional in memory

- Memory allocated for array is contiguous irrespective of whether an array is one dimensional or two dimensional. For example, in case of two dimension array the second row starts in memory where the first row ends. Following program illustrates this concept:

Program 3.13: Write a program in GO language to illustrate storage of multidimensional array

```
package main
import "fmt"

func main() {
    a := [2][3]byte{}
   fmt.Println("First row")
   fmt.Println(&a[0][0])
   fmt.Println(&a[0][1])
   fmt.Println(&a[0][2])
   fmt.Println("\nSecond row")
   fmt.Println(&a[1][0])
   fmt.Println(&a[1][1])
   fmt.Println(&a[1][2])
}
```

Output:

```
First row
0xc000088010
0xc000088011
0xc000088012

Second row
0xc000088013
0xc000088014
0xc000088015
```

- As the multidimensional array is an array of arrays, similarly multi-dimensional slice is a slice of slices. To understand this, let's first look at the definition of a slice. A slice points to an underlying array and is internally represented by a slice header. A slice header is a struct which looks like as follows:

```
type SliceHeader struct
{
   Data uintptr
   Len int
   Cap int
}
```

- Data field in slice header is pointer to the underlying array. For a one dimensional slice, we have declaration below:

```
oneDSlice := make([]int, 2)
```

- To declare a two dimensional slice the declaration would be:

```
twoDSlice := make([][]int, 2)
```

- Above declaration means that we want to create a slice of 2 slices. But wait a second here, we haven't specified the second dimension here, meaning what is the length of each of the inner 2 slices. In case of slice, each of the inner slices has to be explicitly initialized like below:

```
for i := range twoDSlice
{
   twoDSlice[i] = make([]int, 3)
}
```

- So using range on the original slice, we specify the length each of 2 slices using make. Below is one other way of doing the same but with slice elements specified

```
var twoDSlice = make([][]int, 2)
twoDSlice[0] = []int{1, 2, 3}
twoDSlice[1] = []int{4, 5, 6}
```

- Basically, with the above declaration, we create a slice of 2*3 dimensions which is a two-dimensional slice. The same idea can be extended to two-dimension, three-dimension, and so on.

Program 3.14: Write a program on Go language to explain multidimensional slices

```
package main
import "fmt"
func main() {
   twoDSlice1 := make([][]int, 3)
   for i := range twoDSlice1 {
      twoDSlice1[i] = make([]int, 3)
}
fmt.Printf("Number of rows in slice: %d\n", len(twoDSlice1))
fmt.Printf("Number of columns in arsliceray: %d\n", len(twoDSlice1[0]))
fmt.Printf("Total number of elements in slice: %d\n",
len(twoDSlice1)*len(twoDSlice1[0]))
fmt.Println("First Slice")
   for _, row := range twoDSlice1 {
      for _, val := range row {
         fmt.Println(val)
      }
   }
   twoDSlice2 := make([][]int, 2)
   twoDSlice2[0] = []int{1, 2, 3}
   twoDSlice2[1] = []int{4, 5, 6}
fmt.Println()
fmt.Printf("Number of rows in slice: %d\n", len(twoDSlice2))
fmt.Printf("Number of columns in arsliceray: %d\n", len(twoDSlice2[0]))
fmt.Printf("Total number of elements in slice: %d\n",
len(twoDSlice2)*len(twoDSlice2[0]))
fmt.Println("Second Slice")
   for _, row := range twoDSlice2 {
      for _, val := range row {
         fmt.Println(val)
      }
   }
}
```

Output:

```
Number of rows in slice: 3
Number of columns in arsliceray: 3
Total number of elements in slice: 9
First Slice
0
0
0
0
```

```
0
0
0
0
0
Number of rows in slice: 2
Number of columns in arsliceray: 3
Total number of elements in slice: 6
Second Slice
1
2
3
4
5
6
```

Note: It is possible to have different lengths for inner slices. Unlike arrays which have inner arrays of the same length, in case of slice since we initialize each of the inner slices individually, it is possible to have different length for inner slices.

Program 3.15: Write a program in GO language to illustrate the concept of different length for inner slices in multidimensional array.

```
package main
import "fmt"
func main() {
   a := make([][]int, 2)
   a[0] = []int{10, 20, 30}
   a[1] = []int{40, 50}

   fmt.Printf("Number of rows in slice: %d\n", len(a))
   fmt.Printf("Len of first row: %d\n", len(a[0]))
   fmt.Printf("Len of second row: %d\n", len(a[1]))
   fmt.Println("Traversing slice")
      for _, row := range a {
         for _, val := range row {
            fmt.Println(val)
         }
      }
   }
```

Output:

```
Number of rows in slice: 2
Len of first row: 3
Len of second row: 2
```

```
Traversing slice
10
20
30
40
50
```

3.6 STRUCTURES AND STRUCTURE PARAMETERS

- A structure is a collection of data fields with declared data types. Golang has the ability to declare and create own data types by combining one or more types, including both built-in and user-defined types. Each data field in a struct is declared with a known type, which could be a built-in type or another user-defined type.
- Structs are the only way to create concrete user-defined types in Golang. Struct types are declared by composing a fixed set of unique fields. Structs can improve modularity and allows to create and pass complex data structures around the system. You can also consider Structs as a template for creating a data record, like an employee record or an e-commerce product.
- The declaration starts with the keyword **type**, then a name for the new struct, and finally the keyword **struct**. Within the curly brackets, a series of data fields are specified with a name and a type.

Syntax:

```
type identifier struct
{
    field1 data_type
    field2 data_type
    field3 data_type
}
```

Example create structure of student

```
type student struct
{
    rollno int
    name string
}
```

Program 3.16: Write a program to declare structure of student and display it.

```
package main
import "fmt"
type student struct
{
    srno int
    sname string
}
```

```
func main()
{
   fmt.Println(student{10, "Pallawi"})
}
```

Output:

```
{10 Pallawi}
```

Creating instances of structure

- Var keyword is used to create structure. Use dot notation to access members of structure.

For example:

```
type student struct
{
   srno int
   sname string
}
var student s1
s1.srno=10;
s1.sname= "Pallawi"
```

Program 3.17: Write a program in GO language to accept data of one student and print it.

```
package main
import "fmt"
type student struct
{
   srno  int
   sname string
}
func main()
{
   var s1 student
   fmt.Println("Enter student Roll No:")
   fmt.Scanf("%d",&s1.srno)
   fmt.Println("Enter student Name:")
   fmt.Scanf("%s",&s1.sname)
   fmt.Println("\nRoll No:",s1.srno)
   fmt.Println("\nName:",s1.sname)
}
```

Output:

```
Enter student Roll No: 10
Enter student Name: Pallawi
Roll No: 10
Name: Pallawi
```

Creation of struct instance using struct literal:

Syntax: `var variable name=structure name(structure member values)`

Program 3.18: Write a program in Go language to illustrate the concept of creation of struct instance using struct literal.

```
package main
import "fmt"
type employee struct
{
    eno int
    esal float64
    ename string
}
func main()
{
    var  e1= employee{10,87797.76,"Pallawi"}
    fmt.Println(e1)
}
```

Output:

```
{10 87797.76 Pallawi}
```

Creation of struct instance using new keyword:

- An instance of a **struct** can also be created with the new keyword. It is then possible to assign data values to the data fields using dot notation.

Syntax: `Variable name:= new (structure name)`

Program 3.19: Write a program in Go Language to illustrate the concept of creation of structure using new keyword

```
package main
import "fmt"
type employee struct
{
    eno  int
    esal float64
    ename string
}
func main()
{
    e1:= new (employee)
    e1.eno=10
    e1.esal=87797.76
    e1.ename="Pallawi"
    fmt.Println(e1)
}
```

Output:

```
&{10 87797.76 Pallawi}
```

Creation of struct instance using pointer address operator

Syntax: `var structure variable = &structure name{member values}`

Program 3.20: Write a program in GO Language to illustrate the concept of creation of instance of structure using pointer to address operator.

```
package main
import "fmt"
type employee struct
{
    eno  int
    esal float64
    ename string
}
func main()
{
    var e1 = employee{10, 22220.30, "Pallawi"}
    fmt.Println(e1)
    var e2 = employee{}
    e2.eno = 20
    e2.esal = 32224.76
    e2.ename="Meenal"
    fmt.Println(e2)
}
```

Output:

```
{10 22220.3 Pallawi}
{20 32224.76 Meenal}
```

Nested Structures:

- Struct can be nested by creating a Struct type using other Struct types as the type for the fields of Struct. Nesting one struct within another can be a useful way to model more complex structures.

Program 3.21: Write a program in GO Language to illustrate the concept of nested structures

```
package main
import "fmt"
type Salary struct {
    Basic, HRA, TA float64
}
type Employee struct {
    Fname, Lname, Email string
    Age              int
    MonthlySalary    []Salary
}
```

```
func main() {
    e := Employee{
        Fname: "Pallawi",
        Lname: "Kale",
        Email: "Pal21@gmail.com",
        Age:   25,
        MonthlySalary: []Salary {
            Salary {
                Basic: 15000.00,
                HRA:   5000.00,
                TA:    2000.00,
            },
            Salary {
                Basic: 16000.00,
                HRA:   5000.00,
                TA:    2100.00,
            },
            Salary {
                Basic: 17000.00,
                HRA:   5000.00,
                TA:    2200.00,
            },
        },
    }
    fmt.Println(e.Fname, e.Lname)
    fmt.Println(e.Age)
    fmt.Println(e.Email)
    fmt.Println(e.MonthlySalary[0])
    fmt.Println(e.MonthlySalary[1])
    fmt.Println(e.MonthlySalary[2])
}
```

Output:

```
Pallawi Kale
25
Pal21@gmail.com
{15000 5000 2000}
{16000 5000 2100}
{17000 5000 2200}
```

Summary

- Arrays are fixed-length sequence of data items of the same type. They are homogeneous types of data structure.
- Arrays are mutable, which means that they can be changed.
- All array elements are initialized to value zero by default automatically to individual element.
- In Go language user can provide the implicit length of the array with the help of ellipsis.
- Println and Printf both functions are used to print array elements.
- Arrays can be passed as parameters to function.
- Slice is a variable-length sequence which stores elements of a similar type. Users are not allowed to store different type of elements in the same slice.
- Multidimensional arrays can be traversed using for range loop and for loop.
- Memory allocated for array is contiguous irrespective of whether an array is one dimensional or two dimensional.
- A structure is a collection of data fields with declared data types.

Sample Programmes for Study Purpose

1. Write a program in go language to create and print multidimensional Slice.

(For integer)

```
package main
 import "fmt"
func main() {
   // Creating multi-dimensional slice
   s1 := [][]int {
      {1, 2},
      {3, 4},
      {5, 6},
      {7, 8},
   }
   // Accessing multi-dimensional slice
fmt.Println("Slice 1 : ", s1)
}
```

Output:

```
Slice 1 :  [[1 2] [3 4] [5 6] [7 8]]
```

(For string)

```
package main
import "fmt"
func main() {
   // Creating multi-dimensional slice
```

```
    s1 := [][]string {
        []string{"Learning", "GO"},
        []string{"Programming", "on"},
          []string{"OS", "Linux"},
    }
    // Accessing multi-dimensional slice
    fmt.Println("Slice 1 : ", s1)
    // Printing every slice inside s1
    fmt.Println("MultiDimensional Slice s1:")
    for i := 0; i<len(s1); i++ {
       fmt.Println(s1[i])
    }
}
```

Output:

```
Slice 1 :  [[Learning GO] [Programming on] [OS Linux]]
MultiDimensional Slice s1:
[Learning GO]
[Programming on]
[OS Linux]
```

2. Write a program in go language to sort array elements in ascending order.

```
package main
import "fmt"
func main() {
   var temp, s, i, j int
   fmt.Print("Enter the size of Array= ")
   fmt.Scan(&s)
   a := make([]int, s)
   fmt.Print("Enter the Array Items  = ")
   for i = 0; i< s; i++ {
      fmt.Scan(&a[i])
   }
   for i := 0; i< s; i++ {
      for j := i+1; j < s; j++ {
         if(a[i] > a[j]) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         }//if
      }//forj
   }//for i
   fmt.Print("Array after ascending order: \n")
   for j = 0; j < s; j++ {
      fmt.Printf(" %d\n", a[j] )
   }
}
```

Output:

```
Enter the size of Array= 5
Enter the Array Items  = 3 6 2 8 1
Array after ascending order:
 1
 2
 3
 6
 8
```

9. Write a program to print matrix addition.

```
package main
import "fmt"
func main() {
   var i,j, m,n int
   var a[5][5] int
   var b[5][5] int
   var c[5][5]  int
   fmt.Print("Enter the order of matrix = ")
   fmt.Scanf("%d%d ", &m, &n)

   fmt.Print("Enter the First Matrix..\n")
   for i = 0; i< m; i++ {
      for j=0; j<n; j++ {
         fmt.Printf("Enter the element:" )
         fmt.Scanln(&a[i][j])
      }
   }
   fmt.Print("Enter the Second Matrix..\n")
   for i = 0; i< m; i++ {
      for j=0; j<n; j++ {
         fmt.Printf("Enter the element:")
         fmt.Scanln(&b[i][j])
      }
   }
   for i = 0; i< m; i++ {
      for j=0; j<n; j++ {
         c[i][j]=a[i][j]+b[i][j]
      }
}
fmt.Printf("\nFirst Matrix is..\n")
for i = 0; i< m; i++ {
```

```
    for j=0; j<n; j++ {
        fmt.Printf("%d ",a[i][j])
    }
    fmt.Printf("\n")
}
fmt.Printf("\nSecond Matrix is..\n")
for i = 0; i< m; i++ {
        for j=0; j<n; j++ {
            fmt.Printf("%d ",b[i][j])
        }
        fmt.Printf("\n")
    }
    fmt.Printf("\nResualtant Matrix is..\n")
    for i = 0; i< m; i++ {
        for j=0; j<n; j++ {
            fmt.Printf("%d ",c[i][j])
        }
        fmt.Printf("\n")
    }
}
```

Output:

```
Enter the order of matrix = 2 2
Enter the First Matrix..
Enter the element:10
Enter the element:20
Enter the element:30
Enter the element:40
Enter the Second Matrix..
Enter the element:3
Enter the element:2
Enter the element:4
Enter the element:5

First Matrix is..
10 20
30 40

Second Matrix is..
3 2
4 5

Resualtant Matrix is..
13 22
34 45
```

4. Write a program 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.

```
package main
import "fmt"
type student struct {
   srno  int
   m1,m2,m3 int
   sname string
}
func main() {
   var  s1 student
   var total float64
   var avg float64
   fmt.Println("Enter student Roll No:")
   fmt.Scanf("%d",&s1.srno)
   fmt.Println("Enter student Name:")
   fmt.Scanf("%s",&s1.sname)
   fmt.Println("Enter student marks for 3 subjects:")
   fmt.Scanf("%d %d %d",&s1.m1,&s1.m2,&s1.m3)
   fmt.Println("\nRoll No:",s1.srno)
   fmt.Println("\nName:",s1.sname)
   fmt.Println("Marks for 3 Subjects:",s1.m1,s1.m2,s1.m3)
   total=float64(s1.m1+s1.m2+s1.m3)
   avg=float64(total/3.0)
   fmt.Println("Total=",total)
   fmt.Println("Average=",avg)
}
```

Output:

```
Enter student Roll No:
11
Enter student Name:
Pallawi
Enter student marks for 3 subjects:
67 44 22

Roll No: 11

Name: Pallawi
Marks for 3 Subjects: 67 44 22
Total= 133
Average= 44.333333333333336
```

5. Write a program in go language to accept two matrices and display it's multiplication.

```
package main
import "fmt"
func main() {
    var i, j, m, n, p, q, total, k int
    var a[5][5] int
    var b[5][5] int
    var c[5][5] int
    fmt.Print("Enter the order of first matrix = ")
    fmt.Scanf("%d%d ", &m, &n)

    fmt.Print("Enter the order of second matrix = ")
    fmt.Scanf("%d%d ", &p, &q)
    if n != p {
       fmt.Println("Error: The matrix cannot be multiplied") }
    else {
          fmt.Print("Enter the First Matrix..\n")
          for i = 0; i< m; i++ {
              for j=0; j<n; j++ {
                  fmt.Printf("Enter the element:" )
                  fmt.Scan(&a[i][j])
              }
          }
          fmt.Print("Enter the Second Matrix..\n")
          for i = 0; i< p; i++ {
              for j=0; j<q; j++ {
                  fmt.Printf("Enter the element:")
                  fmt.Scan(&b[i][j])
              }
          }
          for i = 0; i< m; i++  {
              for j = 0; j < q; j++ {
                  for k = 0; k < p; k++ {
                     total = total + a[i][k]*b[k][j]
                  }
                  c[i][j] = total
                  total = 0
              }
          }
          fmt.Printf("\nFirst Matrix is..\n")
          for i = 0; i< m; i++ {
```

```
            for j=0; j<n; j++ {
                fmt.Printf("%d ",a[i][j])
            }
            fmt.Printf("\n")
        }
        fmt.Printf("\nSecond Matrix is..\n")
        for i = 0; i< m; i++ {
            for j=0; j<n; j++ {
                fmt.Printf("%d ",b[i][j])
            }
            fmt.Printf("\n")
        }
        fmt.Printf("\nResualtant Matrix is..\n")
        for i = 0; i< m; i++ {
            for j=0; j<n; j++ {
                fmt.Printf("%d ",c[i][j])
            }
            fmt.Printf("\n")
        }
    }
}
```

Output:

```
Enter the order of first matrix = 2 2
Enter the order of second matrix = 2 2
Enter the First Matrix..
Enter the element:2
Enter the element:3
Enter the element:5
Enter the element:7
Enter the Second Matrix..
Enter the element:6
Enter the element:2
Enter the element:8
Enter the element:5

First Matrix is..
2 3
5 7

Second Matrix is..
6 2
8 5

Resultant Matrix is..
36 19
86 45
```

6. Write a program in go language to accept n records of employee information (eno,ename,salary) and display record of employees having maximum salary.

```
package main
import "fmt"
type employee struct {
    eno int
    esal float64
    ename string
}
func main() {
    var e1[10] employee
    var n,k int
    var max float64
    fmt.Println("Enter No of EMployee You Want:")
    fmt.Scanf("%d",&n)
    for i := 0; i<n; i++ {
        fmt.Println("Enter Employee No:")
        fmt.Scanf("%d",&e1[i].eno)
        fmt.Println("Enter Employee Name:")
        fmt.Scanf("%s",&e1[i].ename)
        fmt.Println("Enter Employee Salary:")
        fmt.Scanf("%s",&e1[i].esal)
    }
    max=e1[0].esal
    for i := 0; i<n; i++ {
        if(e1[i].esal>max) {
            max=e1[i].esal
            k=i
        }
    }
    fmt.Println("....Employee Having Maximum Salary...")
    fmt.Println("Employee No:",e1[k].eno)
    fmt.Println("Employee Name:",e1[k].ename)
    fmt.Println("Employee Salary:",e1[k].esal)
}
```

Check Your Understanding

1. Arrays are _____ length data structure.
   (a) Fixed (b) Variable
   (c) undetermined (d) determined
2. Array elements are always stored _____.
   (a) sequentially (b) non linear fashion
   (c) incremental fashion (d) ziczac fashion

3. Array elements are by default initialized to _____.
   (a) Zero (b) 1
   (c) -1 (d) undefined value
4. Ellipsis has _____ dots.
   (a) 4 (b) 3
   (c) 0 (d) 11
5. _____ function is used to print array elements.
   (a) Reflects package (b) package
   (c) fmt package (d) string package
6. _____ indexing cannot be used in an array.
   (a) Positive (b) Up
   (c) Down (d) Negative

Answers

1. (a)	2. (a)	3. (a)	4. (b)	5. (a)	6. (d)

Trace The Output

1.
```
package main
import "fmt"
func main()
{
   var a [3]int
   fmt.Println(a)
}
```
Output:
```
[0 0 0]
```
2.
```
package main
import "fmt"
func main()
{
   x := [2]int{-10,- 20}
   fmt.Println(x)
}
```
Output:
```
[-10 -20]
```
3.
```
package main
import "fmt"
func main()
{
   A := [3][3]int{{11, 12, 83}, {4, -5, 6}}
   fmt.Println(A[1][4])
}
```

Output:

```
# command-line-arguments
./main.go:12:18: invalid array index 4 (out of bounds for 3-element
array)
```

4.
```
P := str[0:2]
P[0] = "University"
fmt.Println(P)
fmt.Println(str)
```

Output:

```
#command-line-arguments
./main.go:11:7: undefined: str
./main.go:14:13: undefined: str
```

Practice Questions

Q. I Answer the following questions in short:

1. What is an Array? Explain with example.
2. What are different types of arrays in Go Language?
3. How to print array elements?
4. What are three ways to print array elements?
5. How to access particular element from array slice?
6. How to identify length of slice of an array?
7. How multidimensional arrays are traversed?
8. Explain structures in GO Language.
9. Explain different ways of creating instance of a structure.
10. What are nested structures?

Q. II Answer the following questions:

1. Explain the concept of arrays with implicit length with program.
2. Explain ways to print array elements with proper example.
3. Write a short note on multidimensional arrays.
4. Explain method of initializing slices using Multi line syntax.
5. How to create array Slice with Low and High values using Colon Syntax?
6. Explain length and capacity of array slices.
7. Explain how to append one or more elements to existing array slice.
8. How to copy elements of one slice to another slice.
9. Explain how multidimensional arrays are stored in memory.

Q. III Define the following:

1. Panic
2. Literal
3. Ellipsis
4. Slice
5. Structure

4...

Methods and Interfaces

Learning Objectives ...

Students will be able to:

- Go programming language from Google will provide learners with an overview of Go's programming features.
- Learners will have gained the knowledge and skills needed to create concise, efficient, and clean applications using Go.

4.1 INTRODUCTION

- 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 ultimately saves the excessive use of memory, acts as a time saver and more importantly, provides 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.

4.1.1 Function Declaration

- Function declaration means 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 this purpose. 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.

Program 4.1: Write a 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(12, 10))
}
```

Output:

```
Area of rectangle is : 120
```

4.2 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.

4.2.1 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:

Program 4.2: Write a 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:   "Sona",
      branch: "CSE",
      particles: 203,
      salary: 34000,
   }
   // Calling the method
   res.show()
}
```

Output:

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

4.2.2 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.

Program 4.3: Write a 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(23)
    value2 := data(20)
    res := value1.multiply(value2)
    fmt.Println("Final result: ", res)
}
```

Output:

```
Final result: 460
```

4.2.3 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
}
```

Program 4.4: Write a 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
```

4.2.4 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 vice versa. 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:

Program 4.5: Write a 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 = "Gourav"
    fmt.Println("Author's Name(Before) : ", a.name)
}
// Main function
func main() {
   // Initializing the values of the author structure
   res := author{
      name:   "Sona",
      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) :  Gourav
Author's Name(After):  Sona
```

4.2 FUNCTIONS VS. METHODS

- **Functions** are an important element of every programming language because they allow you to break big programs into smaller and more manageable parts. Functions must be as independent from each other as possible and must do only one job well. So, if you find yourselves writing functions that do multiple things, you must consider replacing them by multiple functions instead. The single most popular Go function is main(), which is used in every independent Go program. You should already know that function definitions begin with the func keyword.
- **Anonymous functions** can be defined inline without the need for a name, and they are usually used for implementing things that require a small amount of code. In Go, a function can return an anonymous function or take an anonymous function as one of its arguments.
- Additionally, anonymous functions can be attached to Go variables. Note that anonymous functions are also called **closures**, especially in functional programming terminology.
- It is considered a good practice for anonymous functions to have a small implementation and a local focus. If an anonymous function does not have a local focus, then you might need to consider making it a regular function. When an anonymous function is suitable for a job, then it is extremely convenient and makes your work easier; but do not use too many anonymous functions in your programs without having a valid reason.

Functions that return multiple values

- Go functions can return multiple distinct values, which saves you from having to create a dedicated structure in order to be able to receive multiple values at once from a function. You can declare a function that returns four values, two int values, a float64 value, and a string, as follows:

```
func Function() (int, int, float64, string)
{
}
```

- There are anonymous functions and functions that return multiple values in more detail using the Go code of functions. The relevant code will be presented in five parts.

- The first code portion of functions.go is as follows:

```
package main
import (
"fmt"
"os"
"strconv"
)
```

- The second code segment from functions.go is shown in the following Go code:

```
func doubleSquare(x int) (int, int)
{
   return x * 2, x * x
}
```

- Here you can see the definition and implementation of a function named doubleSquare(), which requires a single int parameter and returns two int values.
- The third part of the functions.go program is as follows:

```
func main()
{
   arguments := os.Args
   if len(arguments) != 2
   {
      fmt.Println("The program needs 1 argument!")
      return
   }
   y, err := strconv.Atoi(arguments[1])
   if err != nil
   {
      fmt.Println(err)
      return
   }
}
```

- The preceding code deals with the command line arguments of the program.
- The fourth portion of the functions.go program contains the following Go code:

```
square := func(s int) int
{
   return s * s
}
fmt.Println("The square of", y, "is", square(y))
double := func(s int) int
{
   return s + s
}
fmt.Println("The double of", y, "is", double(y))
```

- Each one of the square and double variables holds an anonymous function. You are allowed to change the value of square, double, or any other variable that holds an anonymous function afterwards, which means that the meaning of those variables can change and calculate something else instead.

Difference between Method and Function:

Sr. No.	Method	Function
1.	It contains a receiver.	It does not contain a receiver.
2.	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.
3.	It cannot be used as a first-order object.	It can be used as first-order objects and can be passed.

4.2 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.

```
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.

4.4.1 Difference between the Value Receiver and Pointer Receiver

- **Value receiver** makes a copy of the type and passes 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 remains 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.

Program 4.6:

```
package main
import (
  "fmt"
)
type Person struct {
    Name string
    Age int
}
func ValueReceiver(p Person) {
    p.Name = "John"
    fmt.Println("Inside ValueReceiver : ", p.Name)
}
func PointerReceiver(p *Person) {
    p.Age = 24
    fmt.Println("Inside PointerReceiver model: ", p.Age)
}
func main() {
    p := Person{"Tom", 28}
    p1:= &Person{"Patric", 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 :  John
Inside Main after value receiver :  Tom
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 Tom to John by ValueReceiver method, but this change was not reflected in the main method. On the other hand, the method with PointerReceivers 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 ValueReceiver.
- If you want to change the state of the receiver in a method, manipulating the value of it, **use a PointerReceiver**. It's not possible with a ValueReceiver, which copies by value. Any modification to a ValueReceiver is local to that copy. If you don't need to manipulate the receiver value, **use a ValueReceiver**.
- The PointerReceiver avoids copying the value on each method call. This can be more efficient if the receiver is a large struct, ValueReceivers are concurrency safe, while pointer receivers are not concurrency safe. Hence a programmer needs to take care of it.
- The Go code of ptrFun.go will illustrate the use of pointers as function parameters.
- The first part of ptrFun.go follows next:

```
package main
import (
"fmt"
)
func getPtr(v *float64) float64
{
   return *v * *v
}
```

So, the getPtr() function accepts a pointer parameter that points to a float64 value.

- The second part of the program is shown in the following Go code:

```
func main()
{
   x := 12.2
   fmt.Println(getPtr(&x))
   x = 12
   fmt.Println(getPtr(&x))
}
```

- The difficult part here is that you need to pass the address of the variable to the getPtr() function because it requires a pointer parameter, which can be done by putting an ampersand in front of a variable (&x).
- Executing ptrFun.go will generate the following kind of output:

```
148.83999999999997
144
```

- If you try to pass a plain value such as 12.12 to getPtr() and call it like getPtr(12.12), the compilation of the program will fail, as shown in the following with an error message:

```
./ptrFun.go:15:21: cannot use 12.12 (type float64) as type *float64 in
argument to getPtr
```

Functions that accept other functions as parameters:

- Go functions can accept other Go functions as parameters, which is a feature that adds versatility to what you can do with a Go function. The single most common use of this functionality is for sorting elements. However, in the example presented here, which is named funFun.go, we will implement a much simpler case that deals with integer values.
- The relevant code will be presented in three parts.
- The first code segment of funFun.go is shown in the following Go code:

```
package main
import "fmt"
func function1(i int) int
{
   return i + i
}
func function2(i int) int
{
   return i * i
}
```

- What we have here is two functions that both accept an int and return an int. These functions will be used as parameters to another function in a short while.
- The second code segment of funFun.go contains the following code:

```
func funFun(f func(int) int, v int) int
{
   return f(v)
}
```

- The funFun() function accepts two parameters, a function parameter named f and an int value. The f parameter should be a function that takes one int argument and returns an int value.
- The last code segment of funFun.go follows next:

```
func main()
{
   fmt.Println("function1:", funFun(function1, 123))
   fmt.Println("function2:", funFun(function2, 123))
   fmt.Println("Inline:", funFun(func(i int) int {return i * i *i}, 123))
}
```

- The first fmt.Println() call uses funFun() with function1 without any parentheses as its first parameter, whereas the second fmt.Println() call uses funFun() with function2 as its first parameter.
- In the last fmt.Println() statement, the implementation of the function parameter is defined inside the call to funFun(). Although this method works fine for simple and small function parameters, it might not work that well for functions with many lines of Go code.
- Executing funFun.go will produce the following output:

```
function1: 246
function2: 15129
Inline: 1860867
```

4.3 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 heart of regular expressions because, without a grammar, you cannot define or use a regular expression.
- Although regular expressions allow you to solve problems that would be extremely difficult to solve otherwise, do not try to solve every problem you face with a regular expression.

4.4 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. Put simply, 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. If you are defining an interface and its implementation in the same Go package, then you might have been using interfaces in the wrong manner.
- 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 satisfy 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 satisfy the io.Writer interface, you will 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.

4.5 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 this way, 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 tells you whether the type assertion was successful or not.
- The second thing a type assertion is to allow you to use the 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.
- Check the Go code of the assertion.go program, which will be presented in two parts. The first part contains the following Go code:

Program 4.7:

```
package main
import (
"fmt"
)
func main() {
   var myInt interface{} = 123
   k, ok := myInt.(int)
   if ok {
      fmt.Println("Success:", k)
   }
   v, ok := myInt.(float64)
   if ok {
      fmt.Println(v)
   } else {
      fmt.Println("Failed without panicking!")
   }
}
```

Output:

```
Success: 123
Failed without panicking!
```

- First, you declare the myInt variable, which has dynamic type int and value 123. Then, you use type assertion two times to test the interface of the myInt variable once for int and once for float64.

- As the myInt variable does not contain a float64 value, the myInt.(float64) type assertion will fail unless handled properly. Fortunately, in this case the correct use of the ok variable will save your program from panicking.
- The second part of the assertion.go program is shown in the following Go code:

```
{
    i := myInt.(int)
        fmt.Println("No checking:", i)
    j := myInt.(bool)
        fmt.Println(j)
}
```

- There are two type assertions taking place here. The first type assertion is successful, so there will be no problem with that. But let's review this particular type assertion a bit further. The type of variable i will be int, and its value will be 123, which is the value stored in myInt. So, as int satisfies the myInt interface, which in this case occurs because the myInt interface requires no functions in order to be implemented, the value of myInt.(int) is an int value.
- However, the second type assertion, which is myInt.(bool), will trigger a panic because the underlying value of myInt is not Boolean (bool). Therefore the output after executing assertion.go by adding the second part will be:

```
Success: 123
Failed without panicking!
No checking: 123
panic: interface conversion: interface {} is int, not bool
goroutine 1 [running]:
main.main()
```

- The reason for panicking: interface {} is int, not bool. When you are working with interfaces, expect to use type assertions as well.

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 and straightforward, as it requires that you 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, you will need to install the myInterface.go package and make it available to the current user. As you 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
```

4.6 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 (that is, it 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.

4.7 EMBEDDED INTERFACES

- An interface T may use an interface type name E in place of a method specification. This is called embedding interface E in T. The method set of T is the union of the method sets of T's explicitly declared methods and of T'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, embed 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()
}
```

Program 4.8: Write a 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:    "Mickey",
        branch:    "Computer science",
        college:   "XYZ",
        year:      2012,
        salary:    50000,
        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: Mickey
Branch: Computer science and passing year: 2012
College Name: XYZ
Salary: 50000
Published articles: 209
Pending articles: 100
```

- As shown in the above example we have three interfaces. Interface 1 and 2 are simple interfaces and interface 3 is the embedded interface which holds both 1 and 2 interfaces in it. So if any changes take place in the interface 1 and interface 2 will reflect in interface 3. And interface 3 can access all the methods present in interface 1 and 2.

Program 4.9: Write a 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()
    picked()
}
```

```
// Interface 3
// Interface 3 embedded with interface
// 1's method and interface 2
// And also contain its own method
type FinalDetails interface {
   details()
   AuthorArticles
   cdeatils()
}
// Structure
type author struct {
   a_name string
   branch string
   college string
   year  int
   salary int
   particles int
   tarticles int
   cidint
   post   string
   pick  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 methods
// of the interface 2
func (a author) articles() {
   pendingarticles := a.tarticles - a.particles
   fmt.Printf("\nPending articles: %d", pendingarticles)
}
func (a author) picked() {
   fmt.Printf("\nTotal number of picked articles: %d", a.pick)
}
```

```
    // Implementing the method
    // of the embedded interface
    func (a author) cdeatils() {
        fmt.Printf("\nAuthor Id: %d", a.cid)
        fmt.Printf("\nPost: %s", a.post)
    }
    // Main value
    func main() {
        // Assigning values to the structure
        values := author {
            a_name: "Mickey",
            branch: "Computer science",
            college: "XYZ",
            year:   2012,
            salary: 50000,
            particles: 209,
            tarticles: 309,
            cid:    3087,
            post:   "Technical content writer",
            pick:   58,
        }
        // Accessing the methods
        // of the interface 1 and 2
        // UsingFinalDetails interface
        var f FinalDetails = values
        f.details()
        f.articles()
        f.picked()
        f.cdeatils()
    }
```

Output:

```
Author Name: Mickey
Branch: Computer science and passing year: 2012
College Name: XYZ
Salary: 50000
Published articles: 209
Pending articles: 100
Total number of picked articles: 58
Author Id: 3087
Post: Technical content writer
```

- As shown in the above example we have three interfaces. Interface 1 and 2 are simple interfaces and interface 3 is the embedded interface which holds both interface 1's method signatures, interface 2 and its own method in it. So if any changes take place in the interface 1's methods and interface 2 it will reflect in interface 3. And interface 3 can access all the methods present in it including interface 1, 2, and its own.

4.8 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.

Summary

- Go is not an object oriented programming language, but that it can mimic some of the functionality offered by object-programming languages, such as Java and C++.
- In Golang, we declare a function using the func keyword.
- The input parameters and return type(s) are optional for a function.
- A function can be declared without any input and output.
- Anonymous functions can be defined inline without the need for a name.
- Anonymous functions are also called closures.
- An empty interface may hold values of any type.

Check Your Understanding

1. _____ are an important element of every programming language because they allow you to break big programs into smaller and more manageable parts.
   (a) Arrays (b) Functions
   (c) Interface (d) Exception
2. _____ can be defined inline without the need for a name.
   (a) Array (b) Package
   (c) Class (d) Anonymous functions
3. Anonymous functions are also called _____.
   (a) closures (b) inner
   (c) Noname (d) None of the above
4. Go functions can accept other Go functions as parameters.
   (a) True (b) False

5. Function can take pointer parameters provided that their signature allows it.
   (a) True (b) False
6. The interface type that specifies one method is known as the empty interface.
   (a) True (b) False
7. An empty interface may hold values of any type.
   (a) True (b) False
8. Go interface does not support embedding.
   (a) True (b) False
9. In embedding, an interface can embed other interfaces or an interface can embed other interface's method signatures in it.
   (a) True (b) False
10. In Go language, the interface is a custom type that is used to specify a set of one or more method signatures and the interface is abstract.
    (a) True (b) False

Answers

1. (b)	2. (d)	3. (a)	4. (a)	5. (a)	6. (b)	7. (a)	8. (b)	9. (a)	10. (a)

Trace the Output

1.
```
package main
import "fmt"
// Creating an interface
type tank interface {
    // Methods
    Tarea() float64
    Volume() float64
}
type myvalue struct {
    radius float64
    height float64
}
// Implementing methods of the tank interface
func (m myvalue) Tarea() float64 {
    return 2*m.radius*m.height + 2*3.14*m.radius*m.radius
}
func (m myvalue) Volume() float64 {
    return 3.14 * m.radius * m.radius * m.height
}
```

```
// Main Method
func main() {
    // Accessing elements of the tank interface
    var t tank
    t = myvalue{10, 14}
    fmt.Println("Area of tank :", t.Tarea())
    fmt.Println("Volume of tank:", t.Volume())
}
```

Output:

```
Area of tank: 908
Volume of tank: 4396
```

2.
```
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
```

Practice Questions

Q. I Answer the following questions in short.

1. What is a method in Go?
2. Describe an Interface in Go?
3. What are Method Values and Expressions?
4. Which are the Interface Types?
5. Write a note on Type Assertions and Type Switches

Q. II Answer the following questions.

1. Explain Method Sets with Interfaces?
2. What are Embedded Interfaces?
3. What are Empty Interfaces?

Q. III Define the following terms:

1. Interface
2. Assertions
3. Expressions

Programs for Practice

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.
4. Write a program in go language to demonstrate working type switch in interface.
5. Write a program in go language to copy all elements of one array into another using method.
6. Write a program in go language to create an interface and display its values with the help of type assertion.
7. 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.
8. Write a program in go language to demonstrate working embedded interfaces.

■■■

5...

Goroutines and Channels

Learning Objectives ...

Students will be able to:

- Understanding Goroutines and Channels.
- Implementing Goroutines and Channels.

5.1 INTRODUCTION

- A **goroutine** is the smallest Go entity that can be executed concurrently. Goroutines are not autonomous entities like Unix processes - goroutines live in threads that live in Unix processes. The main advantage of goroutines is that they are extremely lightweight and running hundreds or thousands of them is not a problem.
- A **channel** is a communication mechanism that allows goroutines to exchange data with others. However, there are certain definite rules to be followed. Firstly, each channel allows the exchange of a particular data type, which is also called the **element type** of the channel, and secondly, for a channel to operate properly, you will need someone to receive what is sent via the channel.

5.2 CONCURRENCY VS. PARALLELISM

- It is a very common misconception that **concurrency** is the same thing as **parallelism**. However, there is a difference between the two. Parallelism is the simultaneous execution of multiple entities of some kind, whereas concurrency is a way of structuring your components so that they can be executed independently when required.
- It is only when you build software components concurrently that you can safely execute them in parallel if the operating system and hardware permit. The **Erlang** programming language did this a long time ago - long before CPUs had multiple cores and computers had lots of RAM.

- In a valid concurrent design, adding concurrent entities makes the whole system run faster because more things can run in parallel. So, the desired parallelism comes from a better concurrent expression and implementation of the problem. The developer is responsible for taking concurrency into account during the design phase of a system and benefit from a potential parallel execution of the components of the system. So, the developer should not think about parallelism, but about breaking things into independent components that solve the initial problem when combined.

5.3 GOROUTINE FUNCTIONS AND LAMBDAS

- A Goroutine is a function or method which executes independently and simultaneously in connection with any other Goroutines present in the program. Every concurrently executing activity in Go language is called as a Goroutine. You can consider a Goroutine like a light weighted thread. The cost of creating Goroutines is very small as compared to the thread. Every program contains at least a single Goroutine and that Goroutine is known as the **main Goroutine**. All the Goroutines are working under the main Goroutines if the main Goroutine is terminated then all the Goroutines present in the program also get terminated. Goroutine always works in the background.
- Concurrency in Golang is the ability for functions to run independent of each other. Goroutines are functions that are run concurrently. Golang provides Goroutines as a way to handle operations concurrently. New Goroutines are created by the go statement. To run a function as a Goroutine, call that function prefixed with the go statement. Here is the example code block:

```
sum()     // A normal function call that executes sum synchronously and
          waits for completing it.
go sum()  // A goroutine that executes sum asynchronously and doesn't wait
          for completing it.
```

5.3.1 Creating Goroutines

- The 'go' keyword is added before each call of function **responseSize**. The three **responseSize** Goroutines start up concurrently and three calls to http.Get are made concurrently as well. The program doesn't wait until one response comes back before sending out the next request. As a result the three response sizes are printed much sooner using Goroutines.

Program 5.1:

```
package main
import(
   "fmt"
   "io/ioutil"
   "log"
   "net/http"
   "time"
)
```

```
func responseSize(url string) {
    fmt.Println("Step1: ",url)
    response, err :=http.Get(url)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Step2: ",url)
    defer response.Body.Close()
    fmt.Println("Step3: ",url)
    body, err :=ioutil.ReadAll(response.Body)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Step4: ",len(body))
}
func main() {
    go responseSize("https://www.golangprograms.com")
    go responseSize("https://coderwall.com")
    go responseSize("https://stackoverflow.com")
    time.Sleep(10*time.Second)
}
```

Output:

```
Step 1 : https://www.golangprograms.com
Step 1 : https://stackoverflow.com
Step 1 : https://coderwall.com
Step 2 : https://stackoverflow.com
Step 3 : https://stackoverflow.com
Step 4 : 116749
Step 2 : https://www.golangprograms.com
Step 3 : https://www.golangprograms.com
Step 4 : 79551
Step 2 : https://coderwall.com
Step 3 : https://coderwall.com
Step 4 : 203842
```

5.4 WAIT GROUPS

A Wait Group provides **a Goroutine synchronization mechanism in Golang**, and is used for waiting for a collection of Goroutines to finish. This Wait Group is used to wait for all the Goroutines launched here to finish. If a Wait Group is explicitly passed into functions, it should be done by pointer. This would be important if we had to launch additional Goroutines.

- Wait Group exports 3 methods:

1.	**Add(int)**	It increases Wait Group counter by given integer value.
2.	**Done()**	It decreases Wait Group counter by 1, we will use it to indicate termination of a Goroutine.
3.	**Wait()**	It blocks the execution until its internal counter becomes 0.

Note: Wait Group is concurrency safe, so it is safe to pass pointer to it as an argument for Goroutines.

Program 5.2:

```
package main
import (
    "fmt"
    "sync"
)
func runner1(wg *sync.WaitGroup) {
    defer wg.Done() // This decreases counter by 1
    fmt.Print("\nI am first runner")
}
func runner2(wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Print("\nI am second runner")
}
func execute() {
    wg := new(sync.WaitGroup)
    wg.Add(2)
    // We are increasing the counter by 2
    // because we have 2 goroutines
    go runner1(wg)
    go runner2(wg)
    // This Blocks the execution
    // until its counter become 0
    wg.Wait()
}
func main() {
    // Launching both the runners
    execute()
}
```

Output:

```
I am second runner
I am first runner
```

5.5 CHANNELS

- Go provides a mechanism called a channel that is used to share data between Goroutines. When you execute a concurrent activity as a Goroutine a resource or data needs to be shared between Goroutines, channels act as a pipe between the Goroutines and provide a mechanism that guarantees a synchronous exchange.
- Data type needs to be specified at the time of declaration of a channel. We can share Values and pointers of built-in, named, struct, and reference types. Data are passed around on channels only one Goroutine has access to a data item at any given time.
- There are two types of channels based on their behaviour of data exchange: unbuffered channels and buffered channels. An unbuffered channel is used to perform synchronous communication between Goroutines while a buffered channel is used for performing asynchronous communication. An unbuffered channel provides assurance that an exchange between two Goroutines is performed at the instant the send and receives takes place. A buffered channel has no such guarantee.
- A channel is created by the make function, which specifies the 'chan' keyword and a channel's element type.

Syntax:

```
Unbuffered := make(chan int) // Unbuffered channel of integer type
Buffered := make(chan int,10) // Buffered channel of integer type
```

- The use of the built-in function makes to create both an unbuffered and buffered channel. The first argument to make requires the keyword chan and then the type of data the channel will allow to be exchanged.

Program 5.3: Simple program to demonstrate use of Buffered Channel.

```
package main
import(
   "fmt"
   "math/rand"
   "sync"
   "time"
)
var goRoutine sync.WaitGroup
func main() {
   rand.Seed(time.Now().Unix())

   // Create a buffered channel to manage the employee vs project load.
   projects := make(chan string,10)

   // Launch 5 goroutines to handle the projects.
   goRoutine.Add(5)
   for i := 1; i <= 5; i++.{
      go employee(projects, i)
   }
```

```
    for j :=1; j <= 10; j++ {
        projects <- fmt.Sprintf("Project :%d", j)
    }
    // Close the channel so the goroutines will quit
    close(projects)
    goRoutine.Wait()
}
func employee(projects chan string, employee int) {
    defer goRoutine.Done()
    for {
        // Wait for project to be assigned.
        project, result := <-projects
        if result==false {
            // This means the channel is empty and closed.
            fmt.Printf("Employee : %d : Exit\n", employee)
            return
        }
        fmt.Printf("Employee : %d : Started   %s\n", employee, project)
        // Randomly wait to simulate work time.
        sleep := rand.Int63n(50)
        time.Sleep(time.Duration(sleep) * time.Millisecond)
        // Display time to wait
        fmt.Println("\nTime to sleep",sleep,"ms\n")
        // Display project completed by Student.
        fmt.Printf("Employee : %d : Completed %s\n", employee, project)
    }
}
```

Output:

```
Student : 4 : Started   Project : 4
Student : 2 : Started   Project : 1
Student : 5 : Started   Project : 5
Student : 1 : Started   Project : 2
Student : 3 : Started   Project : 3

Time to sleep 3 ms

Student : 4 : Completed Project : 4
Student : 4 : Started   Project : 6

Time to sleep 13 ms

Student : 4 : Completed Project : 6
Student : 4 : Started   Project : 7
```

```
Time to sleep 16 ms

Student : 3 : Completed Project : 3
Student : 3 : Started   Project : 8

Time to sleep 20 ms

Student : 1 : Completed Project : 2
Student : 1 : Started   Project : 9

Time to sleep 42 ms

Student : 5 : Completed Project : 5
Student : 5 : Started   Project : 10

Time to sleep 47 ms

Student : 2 : Completed Project : 1
Student : 2 : Exit

Time to sleep 10 ms

Student : 5 : Completed Project : 10
Student : 5 : Exit

Time to sleep 41 ms

Student : 3 : Completed Project : 8
Student : 3 : Exit

Time to sleep 46 ms

Student : 4 : Completed Project : 7
Student : 4 : Exit

Time to sleep 43 ms

Student : 1 : Completed Project : 9
Student : 1 : Exit
```

- In above program, a buffered channel of type string is created with a capacity of 10. Wait Group is given the count of 5, one for each Goroutine. 10 strings are sent into the channel to simulate or replicate project for the Goroutines. Once the last string is sent into the channel, the channel is going to be closed and the main function waits for all the projects to be completed.

5.6 SENDING AND RECEIVING WITH CHANNEL

- **A channel** is a communication mechanism that allows Goroutines to exchange data, among other things. However, there are some definite rules here. First, each channel allows the exchange of a particular data type, which is also called the **element type** of the channel, and second, for a channel to operate properly, you will need someone to receive what is sent via the channel. You should declare a new channel using the chan keyword, and you can close a channel using the close() function.

- When using a channel as a function parameter, you can specify its direction, that is, whether it is going to be used for sending or receiving. In my opinion, if you know the purpose of a channel in advance, you should use this capability because it will make your programs more robust as well as safer because you will not be able to send data accidentally to a channel from which you should only receive data or receive data from a channel to which you should only be sending data.

5.7 UNBUFFERED AND BUFFERED CHANNELS

5.7.1 Unbuffered Channels

- In unbuffered channel there is no capacity to hold any value before it is received. In this type of channel both a sending and receiving Goroutine should be ready at the same instance before any send or receive operation can complete. If the two Goroutines are not ready at the same instance, the channel makes the Goroutine that performs its respective send or receive operation first wait. Synchronization is fundamental in the interaction between the send and receive on the channel.

5.7.2 Buffered Channels

- In buffered channel there is a capacity to hold one or more values before they are received. In this type of channels it is not expected that Goroutines be ready at the same instance to perform sends and receives. There are also different conditions for when a send or receive does block. A receive will block only if there's no value in the channel to receive. A send will block only if there's no available buffer to place the value being sent.

5.8 DIRECTIONAL CHANNELS

- Channels can be directional - which means that you can restrict a channel to only send or receive data. This is specified by the arrow (<-) accompanied with the channel declaration.
- For example, take a look at the type definition of the out argument of the multiplyByTwo function:

```
out chan<- int
```

- The chan<- declaration tells us that you can only send data into the channel, but not receive data from it.
- The int declaration tells us that the channel will only accept int datatypes.
- Although they look like separate parts, chan <- int can be thought of as one datatype, that describes a "send-only" channel of integers.
- Similarly, an example of a "receive-only" channel declaration would look like:

```
out<-chanint
```

- You can also declare a channel without giving directionality, which means it can send or receive data.

5.9 MULTIPLEXING WITH SELECT

GoLang select statement is like the switch statement, which is used for multiple channels operation. This statement blocks until any of the cases provided are ready. The syntax of the select statement is like the switch statements. It is really easy to work with.

```
select
{
   case case1:
      // case 1...
   case case2:
      // case 2...
   case case3:
      // case 3...
   case case4:
      // case 4...
   default:
      // default case...
}
```

Program 5.4: Instead of having concrete cases it has operations that are either sending or receiving using channels.

```
package main
import (
   "fmt"
)
func g1(ch chan int) {
   ch<- 12
}
func g2(ch chan int) {
   ch<- 32
}
func main() {
   ch1 := make(chan int)
   ch2 := make(chan int)
   go g1(ch1)
   go g1(ch2)
   select {
      case v1 := <-ch1:
         fmt.Println("Got: ", v1)
```

```
        case v2 := <-ch2:
            fmt.Println("Got: ", v2)
    }
}
```

Output:

```
Got:  12
```

- The output we got is totally dependent on what was executed then. It is simply random. We cannot predict the output since select works in a very different way. It chooses any output if all of the statements are ready for execution.

5.10 TIMERS AND TICKERS

- Timers and Tickers let write code that executes in the future, once or repeatedly.

Timers vs Tickers:

- **Timers:** These are used for one-off tasks. It represents a single event in the future. You tell the timer how long you want to wait, and it provides a channel that will be notified at that time.
- **Tickers**: Tickers are exceptionally helpful when you need to perform an action repeatedly at given time intervals. We can use tickers, in combination with Goroutines to run these tasks in the background of our applications.

A Simple Timer:

- The execution of time is associated with a Goroutine. The purpose of using **done** is to control the execution of the program only.

Summary

- A **Goroutine** is the minimum Go entity that can be executed concurrently.
- A **channel** is a communication mechanism that allows Goroutines to exchange data, among other things.
- In unbuffered channel there is no capacity to hold any value before it's received.
- In buffered channel there is a capacity to hold one or more values before they're received.
- Channels can be directional - which means that you can restrict a channel to only send or receive data.
- Timers and Tickers let you write code that executes in the future, once or repeatedly.
- Tickers are exceptionally helpful when you need to perform an action repeatedly at given time intervals.

Check Your Understanding

1. Which of the following is initial value (zero value) for interfaces, slice, pointers, maps, channels and functions?

   (a) 0 (b) ""

   (c) Nil (d) False

2. _____ is the minimum Go entity that can be executed concurrently.
   (a) Goroutine (b) Channel
   (c) Concurrency (d) None of the above
3. A _____ is a communication mechanism that allows Goroutines to exchange data.
   (a) Channel (b) Pipe
   (c) Subroutine (d) None of these
4. You should declare a new channel using the _____ keyword.
   (a) chain (b) chan
   (c) channel (d) None of the above
5. _____ is the simultaneous execution of multiple entities of some kind.
   (a) Parallelism (b) Concurrency
   (c) Semaphore (d) None of the above
6. The cost of creating Goroutines is very small as compared to the thread.
   (a) True (b) False
7. New Goroutines are created by the go statement.
   (a) True (b) False
8. A _____ group provides a Goroutine synchronization mechanism in Golang.
   (a) Wait (b) Watch
   (c) subroutine (d) Golang
9. Channels can be directional.
   (a) True (b) False
10. GoLang select statement is like the _____ statement
    (a) goto (b) switch
    (c) continue (d) return

Answers

1. (c)	2. (a)	3. (a)	4. (b)	5. (a)	6. (a)	7. (a)	8. (a)	9. (a)	10. (b)

Trace the Output

1. package main

```
import(
    "fmt"
    "strconv"
)
func main(){
    ch:=make(chan string)

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

```
        go func(i int) {
            for j := 0; j < 10; j++ {
                ch <- "Goroutine : " + strconv.Itoa(i)
            }
        }(i)
    }
    for k := 1; k <= 10; k++ {
        fmt.Println(k, <-ch)
    }
}
```

Output:

```
1 Goroutine : 0
2 Goroutine : 0
3 Goroutine : 0
4 Goroutine : 4
5 Goroutine : 1
6 Goroutine : 2
7 Goroutine : 3
8 Goroutine : 6
9 Goroutine : 5
10 Goroutine : 7
```

Explanation: This is an example to Launch multiple Goroutines and each Goroutine adds values to a Channel.

The program starts with 10 Goroutines. We created a ch string channel and writing data to that channel by running 10 Goroutines concurrently. The direction of the arrow with respect to the channel specifies whether the data is sent or received. The arrow point towards ch specifies we are writing to channel ch. The arrow point outwards from ch specifies we are reading from channel ch.

2.
```
package main
import(
    "fmt"
)
func main() {
    c:=generator()
    receiver(c)
}
```

```
func receiver(c <-chan int) {
    for v := range c {
        fmt.Println(v)
    }
}
func generator()<-chan int {
    c:=make(chan int)
    go func() {
        for i:=0;i<10;i++ {
            c <-i
        }
        close(c)
    }()
    return c
}
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Explanation: This is an example of Send and Receive values from Channel. The main function has two functions generator and receiver. We create a c int channel and return it from the generator function. The for loop running inside anonymous Goroutines writing the values to the channel c.

Practice Questions

Q.I Answer the following questions in short.

1. Compare Concurrency and Parallelism.
2. What are Goroutine functions?
3. What are Wait Groups?
4. What are buffered channels and unbuffered channels?
5. What are directional channels?
6. What is the use of Timers and Tickers?

Q. II Answer the following questions.

1. Write a note on Multiplexing with select.
2. Explain Timer with an example.

Q. III Define the following terms:

1. Wait Group
2. Timers
3. Tickers

Programs for Practice

1. **Write a program in Go to illustrate how to create an anonymous Goroutine.**

```
package main
import (
    "fmt"
    "time"
)
// Main function
func main() {
   fmt.Println("Welcome!! To Main function")
   // Creating Anonymous Goroutine
   go func() {
      fmt.Println("Welcome !!  In Function ")
   }()
   time.Sleep(1 * time.Second)
   fmt.Println("GoodBye!! To Main function")
}
```

Output:

```
Welcome!! To Main function
Welcome !! In Function
GoodBye!! To Main function
```

2. **Write a program in Go how to create channel and illustrate how to close a channel using for range loop and close function.**

```
package main
import "fmt"
// Function
func myfun(mychnl chan string) {
   for v := 0; v < 4; v++ {
      mychnl <- "Hello"
   }
   close(mychnl)
}
```

```
// Main function
func main() {
    // Creating a channel
    c := make(chan string)
    // calling Goroutine
    go myfun(c)
    // When the value of ok is set to true means the
    // channel is open and it can send or receive data
    // When the value of ok is set to false means the channel is closed
    for {
        res, ok := <-c
        if ok == false {
            fmt.Println("Channel Close ", ok)
            break
        }
        fmt.Println("Channel Open ", res, ok)
    }
}
```

Output:

```
Channel Open Hello true
Channel Open Hello true
Channel Open Hello true
Channel Close false
```

3. **Write a program in Go main Goroutine computes the 10th Fibonacci number using an Inefficient recursive algorithm.**

```
package main
import (
    "fmt"
    "strconv"
)
func FibonacciLoop(n int) int {
    f := make([]int, n+1, n+2)
    if n < 2 {
        f = f[0:2]
    }
```

```
    f[0] = 0
    f[1] = 1
    for i := 2; i<= n; i++ {
        f[i] = f[i-1] + f[i-2]
    }
    return f[n]
}
func FibonacciRecursion(n int) int {
    if n <= 1 {
        return n
    }
    return FibonacciRecursion(n-1) + FibonacciRecursion(n-2)
}
func main() {
    for i := 0; i<= 10; i++ {
        fmt.Print(strconv.Itoa(FibonacciLoop(i)) + " ")
    }
    fmt.Println("")
    for i := 0; i<= 10; i++ {
        fmt.Print(strconv.Itoa(FibonacciRecursion(i)) + " ")
    }
    fmt.Println("")
}
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55
0 1 1 2 3 5 8 13 21 34 55
```

4. **Write a program in GO prints out the numbers from 0 to 10, waiting between 0 and 250 ms after each one using delay function.**

```
package main
import (
    "fmt"
    "time"
)
func numbers() {
    for i := 1; i <= 5; i++ {
        time.Sleep(250 * time.Millisecond)
        fmt.Printf("%d ", i)
    }
}
```

```
func alphabets() {
   for i := 'a'; i <= 'e'; i++ {
      time.Sleep(400 * time.Millisecond)
      fmt.Printf("%c ", i)
   }
}
func main() {
   go numbers()
   go alphabets()
   time.Sleep(3000 * time.Millisecond)
   fmt.Println("main terminated")
}
```

Output:

```
1 a 2 3 b 4 c 5 d e main terminated
```

5. **Write a program in GO implement multiple Goroutine function and schedule is determined by the scheduler.**

```
package main
import (
    "fmt"
    "time"
)
// For gl-1
func namesGLine() {
   arr1 := [4]string{"Rohit", "Suman", "Aman", "Ria"}
   for t1 := 0; t1 <= 3; t1++ {
      time.Sleep(150 * time.Millisecond)
      fmt.Printf("%s\n", arr1[t1])
   }
}
// gl-2 function
func idsGline() {
   arr2 := [4]int{300, 301, 302, 303}
   for t2 := 0; t2 <= 3; t2++ {
      time.Sleep(500 * time.Millisecond)
      fmt.Printf("%d\n", arr2[t2])
   }
}
```

```
func main() {
    fmt.Println("!...Main Go-routine Start...!")
        // calling Goroutine 1
    go namesGLine()
        // calling Goroutine 2
    go idsGline()
    time.Sleep(3500 * time.Millisecond)
    fmt.Println("\n!...Main Go-routine End...!")
}
```

Output:

```
!...Main Go-routine Start...!
Rohit
Suman
Aman
300
Ria
301
302
303
!...Main Go-routine End...!
```

6. Write a program in Go to illustrate channels buffering.

```
package main
import "fmt"
func main() {
    // Here we 'make' a channel of strings buffering up to
    // 2 values.
    messages := make(chan string, 2)
    // Because this channel is buffered, we can send these
    // values into the channel without a corresponding
    // concurrent receive.
    messages<- "buffered"
    messages<- "channel"

    // Later we can receive these two values as usual.
    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

Output:

```
buffered
channel
```

SETC:

7. Write a program in GO. Lambda function handler using structured types you can also pass in structured events to your function handler.

```
package main
import (
    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/lambda"
    "encoding/json"
    "fmt"
    "os"
    "strconv"
)
type getItemsRequest struct {
   SortBy string
   SortOrder string
   ItemsToGet int
}
type getItemsResponseError struct {
   Message string 'json:"message"'
}
type getItemsResponseData struct {
   Item string 'json:"item"'
}
type getItemsResponseBody struct {
   Result string                  'json:"result"'
   Data   []getItemsResponseData 'json:"data"'
   Error  getItemsResponseError  'json:"error"'
}
type getItemsResponseHeaders struct {
   ContentType string 'json:"Content-Type"'
}
type getItemsResponse struct {
   StatusCode int                          'json:"statusCode"'
   Headers    getItemsResponseHeaders   'json:"headers"'
   Body       getItemsResponseBody      'json:"body"'
}
func main() {
   // Create Lambda service client
   sess := session.Must(session.NewSessionWithOptions(session.Options{
```

```
    SharedConfigState: session.SharedConfigEnable,}))
    client := lambda.New(sess, &aws.Config{Region: aws.String("us-west-2")})
    // Get the 10 most recent items
    request := getItemsRequest{"time", "descending", 10}
    payload, err := json.Marshal(request)
    if err != nil {
        fmt.Println("Error marshalling MyGetItemsFunction request")
        os.Exit(0)
    }
    result, err := client.Invoke(&lambda.InvokeInput{FunctionName:
    aws.String("MyGetItemsFunction"), Payload: payload})
    if err != nil {
        fmt.Println("Error calling MyGetItemsFunction")
        os.Exit(0)
    }
    var resp getItemsResponse
    err = json.Unmarshal(result.Payload, &resp)
    if err != nil {
        fmt.Println("Error unmarshallingMyGetItemsFunction response")
        os.Exit(0)
    }
    // If the status code is NOT 200, the call failed
    if resp.StatusCode != 200 {
        fmt.Println("Error getting items, StatusCode:"
                                        + strconv.Itoa(resp.StatusCode))
        os.Exit(0)
    }
    // If the result is failure, we got an error
    if resp.Body.Result == "failure" {
        fmt.Println("Failed to get items")
        os.Exit(0)
    }
    // Print out items
    if len(resp.Body.Data) > 0 {
        for i := range resp.Body.Data {
            fmt.Println(resp.Body.Data[i].Item)
        }
    } else {
        fmt.Println("There were no items")
    }
}
```

Output:

```
{
   "statusCode": 200,
   "body":
   {
      "result": "success",
      "error": ""
      "data": [
      {
         "item": "item1"
      },
      {
         "item": "item2"
      }
      ]
   }
}
```

8. **Write a program in Go program such that the squares are calculated in a separate Goroutine, cubes in another Goroutine and the final summation happens in the main Goroutine.**

```
package main
import (
    "fmt"
)
func calcSquares(number int, squareop chan int) {
    sum := 0
    for number != 0 {
        digit := number % 10
        sum += digit * digit
        number /= 10
    }
    squareop<- sum
}
func calcCubes(number int, cubeop chan int) {
    sum := 0
    for number != 0 {
        digit := number % 10
        sum += digit * digit * digit
        number /= 10
    }
    cubeop<- sum
}
```

```
func main() {
    number := 589
    sqrch := make(chan int)
    cubech := make(chan int)
    go calcSquares(number, sqrch)
    go calcCubes(number, cubech)
    squares, cubes := <-sqrch, <-cubech
    fmt.Println("Final output", squares + cubes)
}
```

Output:

```
Final output 1536
```

■■■

6... Packages and Files

Learning Objectives ...

Students will be able to:

- Understanding the use of Packages and Files in Go.

6.1 INTRODUCTION

- Packages are an important and powerful part of the Go language. The work of a package is to design and maintain a large number of programs by grouping together related features 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 code.
- In Go language, every package is defined with a different name and that name is close to their purpose like "strings" package which contains methods and functions that are only related to strings.
- Go programs are also organized into packages. It can be considered as a **collection of source files in the same directory that are compiled together**. Functions, types, variables, and constants defined in one source file are visible to all other source files within the same package. A repository contains one or more modules.

6.2 PACKAGES AND WORKSPACES

- A package is a collection of source files in the same directory that are compiled together. A workspace is a directory hierarchy with two directories at its root: src contains Go source files, and bin contains executable commands.

6.2.1 Packages

- Everything in Go is delivered in the form of packages. A Go **package** is a Go source file that begins with the package keyword followed by the name of the package. Some packages have a structure.

- For example, the net package has several subdirectories named http, mail, rpc, smtp, textproto, and url which should be imported as net/http, net/mail, net/rpc, net/smtp, net/textproto, and net/url respectively.
- Packages are mainly used for grouping related functions, variables, and constants so that you can transfer them easily and use them in your own Go programs. Note that except the main package, Go packages are not autonomous programs and cannot be compiled into executable files. This means that they need to be called directly or indirectly from a main package in order to be used.
- To illustrate the above, check the following command in Linux:

```
$ go run aPackage.go
go run: cannot run non-main package
```

6.2.2 Workspace Configuration

- After installing Go, you have to configure your workspace. A Go workspace is a directory stored on your local machine that contains code specific to Go. Any code written in Go should be saved to the workspace in order for the compiler to operate at optimal efficiency. The path to your workspace will be later stored in an environment variable named $GOPATH.
- Inside of the workspace you must create 3 subdirectories named by convention:
  1. **$GOPATH/src: src** is where all of your Go projects and programs are located. It handles name spacing package management for all your Go repos.
  2. **$GOPATH/pkg: pkg** stores archived files of packages installed in programs. This essentially helps to save compilation time based on whether the packages being used have been modified.
  3. **$GOPATH/bin: bin** stores all of your compiled binaries.

6.2.3 Exporting Package Names

- A utility package is supposed to provide some **variables** to a package who imports it. Like export syntax in JavaScript, Go exports a variable if a variable name starts with **Uppercase**. All other variables not starting with an uppercase letter is private to the package.

6.3 IMPORT PATHS AND NAMED IMPORTS

- An import path is a string that uniquely identifies a package. A package's import path corresponds to its location inside a workspace or in a remote repository.

6.3.1 Import Paths

- The packages from the standard library are given short import paths such as "fmt" and "net/http". For your own packages, you must choose a base path that should not collide with future additions to the standard library or other external libraries.

If you keep your code in a source repository somewhere, then you should use the root of that source repository as your base path. For instance, if you have a GitHub account at github.com/user, that should be your base path. Note that one does not need to publish your code to a remote repository before you can build it. It is a good practise to organize your code as if you will publish it someday. In practice you can choose any arbitrary path name, as long as it is unique to the standard library.

6.3.2 Named Imports

While importing multiple packages in a go file, there is a possibility that the custom or remote package name may be the same and thus it results in ambiguity. To resolve this ambiguity, Golang Named Imports are used.

Example:

```
import (
        "fmt"
        myfmt "format/fmt" // Custom Package
)
```

The above example uses fmt package that comes as a part of the standard library and the second imported package name is also fmt that is created by the user. If we try to access the exported functions from both fmt packages we cannot do it as both have the same name.

6.4 PACKAGE INITIALIZATION

Package-level variables are initialized in declaration order, but after any of the variables they depend on. Initialization of variables declared in multiple files is done in lexical file name order. Variables declared in the first file are declared before any of the variables declared in the second file. Initialization cycles are not allowed.

Dependency analysis is performed per package only references referring to variables, functions and methods declared in the current package are considered.

Variables may also be initialized using init functions:

```
func init() { … }
```

Such multiple functions may be defined. They cannot be called from inside a program.

A package with no imports is initialized:

- By assigning initial values to all its package-level variables.
- Followed by calling all init functions in the order they appear in the source.

Imported packages are initialized before the package itself.

Each package is initialized once, regardless if it is imported by multiple other packages.

It follows that there can be no cyclic dependencies.

Package initialization happens in a single goroutine, sequentially, one package at a time.

- For example, given the following statement

```
var (
a = c + b
b = f()
c = f()
d = 3
)
func f() int
{
    d++
    return d
}
```

The initialization sequence is: d, b, c, a.

6.5 BLANK IMPORTS

- Blank identifier in importing packages means specifying a blank import for the imported package. The syntax for it is

```
import _
```

- Go doesn't allow any unused variable. Any unused variable can be replaced by a blank identifier ('_').
- A blank import of a package is used when imported package is not being used in the current program. But we intend to import that package so that init function in the GO source files belonging to that package can be called and initialization of variables in that package can be done properly.

6.6 UNIT TESTING WITH TEST FUNCTIONS

- Unit testing is very common in Go (Golang) code. Many of the types of applications that Go is frequently used for, are very easy to write good unit tests. However, if you are new to the language or new to unit testing in general, then there are a few things that may help to steer you in the right direction. This article focuses on helping get you started writing unit tests and learning about some of the best practices related to testing.

6.7 TABLE TESTS AND RANDOM TESTS

- Testing is one of the main and important activities in programming, but often it's neglected or left as an afterthought. Go provides basic testing capabilities that seem to be primitive. Go supplies the tools to create the automated tests one needs as a programmer. Table Tests and Random Tests are more commonly used.

6.7.1 Table tests

Decision table testing is a software testing technique used to test system behaviour for different input combinations. This is a systematic approach where the different input combinations and their corresponding system behaviour (Output) are captured in a tabular form.

A decision table is an important technique used for testing and requirements management.

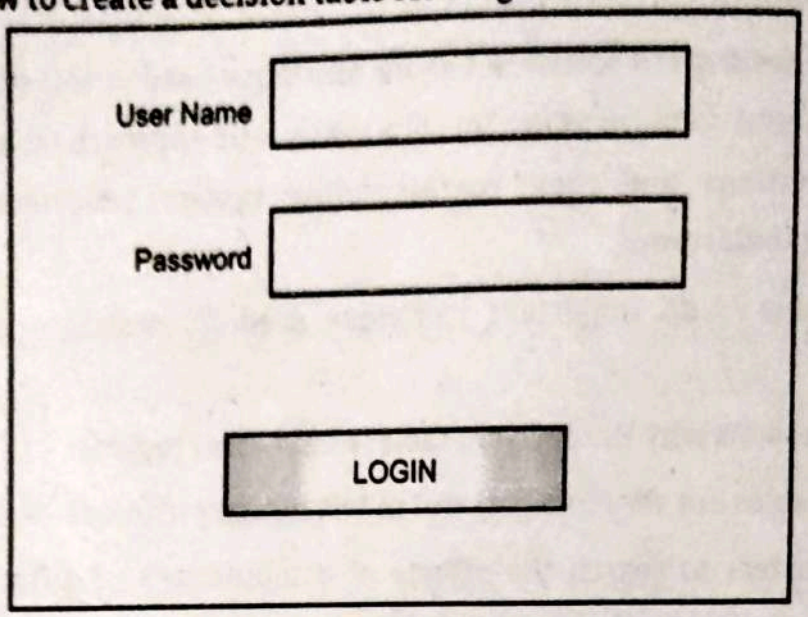
Some of the reasons why the decision table is important include:

- Decision tables are very much useful in test design technique
- It helps testers to search the effects of combinations of different inputs and other software states that implement business rules.
- It provides a regular way of stating complex business rules which benefits the developers as well as the testers.
- It assists in the development process with the developer to do a better job. Testing with all combination might be impractical.
- It is the most preferable choice for testing and requirements management.
- It is a structured exercise to prepare requirements when dealing with complex business rules.
- It is also used to model complicated logic.

Following are the Advantages of using Decision Table in Software Testing:

- Any complex business flow can be easily converted into the test scenarios & test cases using this technique.
- Decision tables work in an iterative manner. Therefore, the table created at the first iteration is used as the input table for the next tables. The iteration is done only if the initial table is not satisfactory.
- Simple to understand and everyone can use this method to design the test scenarios & test cases.
- It provides complete coverage of test cases which help to reduce the rework on writing test scenarios and test cases.
- These tables guarantee that we consider every possible combination of condition values. This is known as its completeness property.

Example: How to create a decision table for a login screen?



The condition states that if the user provides the correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username	F	T	F	T
Password	F	F	T	T
Output	E	E	E	H

In the above example,

- T – Correct username/password
- F – Wrong username/password
- E – Error message is displayed
- H – Home screen is displayed

- Now let's understand the interpretation of the above cases:
  - **Case 1:** Username and password both were wrong. So the user is shown an error message.
  - **Case 2:** Username was correct, but the password was wrong. So the user is shown an error message.
  - **Case 3:** Username was wrong, but the password was correct. So the user is shown an error message.
  - **Case 4:** Username and password both were correct so the user is navigated to the homepage.

6.7.2 Random Tests

There are basically two methods to generate a random number using two functions of the go language, they are int() and int31()

```
rand.Intn(n          int)        int          //example          rand.intn(4)
rand.Int31() int31 //example rand.int31()
```

Before going further to understand the core concept of the working of the random number in go language firstly we need to understand the uses of the random number. You have seen many applications used to send OTP for registrations and login to that application, so these applications use the logic of random number generation.

In case of a very large number of users, it may be possible that the generated random number can be duplicated with any already sent random number so to avoid such type of situation we are using the go language system used to define random number generation logic.

Working of the generation of the random number in the go language is explained with the help of the below steps:

- Before using random generation in go language we have to import the math/rand package into the application.
- After importing the package we will have two options to generate the random number one is by using the function int() and another is with the help of the function int31().
- Both the function used to generate the random number for us, the only difference is that in case of the int() we can pass some number where we can define the specification of the number length and in case of int31() function we cannot pass any custom value to it, it will automatically generate the number with a length of 31 bit.
- We can also customize the length of the random number generated by these functions by adding and overriding the logic of generation of the number.

Program 6.1: Write a simple go program to generate the random number using int31().

```
package main
import (
"fmt"
"math/rand"
)
func main()
{
    random_no1 := rand.Int31()
    random_no2 := rand.Int31()
```

```
    random_no3 := rand.Int31()
    random_no4 := rand.Int31()
    fmt.Println("The first random number is: ", random_no1)
    fmt.Println("The second random number is: ", random_no2)
    fmt.Println("The third random number is: ", random_no3)
    fmt.Println("The fourth random number is: ", random_no4)
}
```

Output:

```
The first random number is:  1298498081
The second random number is:  2019727887
The third random number is:  1427131847
The fourth random number is:  939984059
```

6.8 BENCHMARKING

- Benchmarking can give you information about the performance of a function or a program in order to understand better how much faster or slower a function is compared to another function, or compared to the rest of the application. Using that information, you can easily reveal the part of the Go code that needs to be rewritten in order to improve its performance.
- Never benchmark your Go code on a busy Unix machine that is currently being used for other, more important purposes unless you have a very good reason to do so.
- Go follows certain conventions regarding benchmarking. The most important convention is that the name of a benchmark function must begin with Benchmark.
- Once again, the Go test subcommand is responsible for benchmarking a program. As a result, you still need to import the testing standard Go package and include benchmarking functions in Go files that end with _test.go.

A simple benchmarking example

- We will see a basic benchmarking example that will measure the performance of three algorithms that generate numbers belonging to the **Fibonacci sequence**. The good news is that such algorithms require lots of mathematical calculations, which make them perfect candidates for benchmarking.

Program 6.2: To generate numbers belonging to the **Fibonacci sequence.**

For the purposes of this section, we have to create a new main package, which will be saved as benchmarkMe.go and presented in three parts.

The first part of benchmarkMe.go is as follows:

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

```
func fibo1(n int) int
{
   if n == 0
   {
      return 0
   } else if n == 1
   {
      return 1
   } else
   {
   return fibo1(n-1) + fibo1(n-2)
   }
}
```

- The preceding code contains the implementation of the fibo1() function that uses recursion in order to calculate the numbers of the Fibonacci sequence. Although the algorithm works fine, this is a relatively simple and somehow slow approach.
- The second code segment of benchmarkMe.go is shown in the following Go code:

```
func fibo2(n int) int
{
   if n == 0 || n == 1
   {
      return n
   }
   return fibo2(n-1) + fibo2(n-2)
}
```

- In this part, you see the implementation of the fibo2() function that is almost identical to the fibo1() function that you saw earlier. However, it will be interesting to see whether a small code change a single if statement as opposed to if else if block has any impact to the performance of the function.
- The third code portion of benchmarkMe.go contains yet another implementation of a function that calculates numbers that belong to the Fibonacci sequence:

```
func fibo3(n int) int
{
   fn := make(map[int]int)
   for i := 0; i<= n; i++
   {
      var f int
      if i<= 2
      {
         f = 1
      } else
```

```
        {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

- The fibo3() function presented here uses a totally new approach that requires a Go map and has a for loop. It remains to be seen whether this approach is indeed faster than the other two implementations.
- The remaining code of benchmarkMe.go follows next:

```
func main()
{
    fmt.Println(fibo1(40))
    fmt.Println(fibo2(40))
    fmt.Println(fibo3(40))
}
```

- Executing benchmarkMe.go will generate the following output:

```
102334155
102334155
102334155
```

- As the Go rules require, the version of benchmarkMe.go containing the benchmark functions will be saved as benchmarkMe_test.go. This program is presented in five parts.

Program 6.3:

The first code segment of benchmarkMe_test.go contains the following Go code:

```
package main
import (
"testing"
)
var result int
func benchmarkfibo1(b *testing.B, n int)
{
    var r int
    for i := 0; i<b.N; i++
    {
        r = fibo1(n)
    }
    result = r
}
```

In the preceding code, you can see the implementation of a function with a name that begins with the benchmark string instead of the Benchmark string. As a result, this function will not run automatically because it begins with a lowercase b instead of an uppercase B.

The reason for storing the result of fibo1(n) in a variable named r and using another global variable named result afterwards is tricky. This technique is used for preventing the compiler from performing any optimizations that will exclude the function that you want to measure because its result is never used. The same technique will be applied in functions benchmarkfibo2() and benchmarkfibo3(), which will be presented next.

The second part of benchmarkMe_test.go is shown in the following Go code:

```
func benchmarkfibo2(b *testing.B, n int)
{
   var r int
   for i := 0; i<b.N; i++
   {
      r = fibo2(n)
   }
   result = r
}
func benchmarkfibo3(b *testing.B, n int)
{
   var r int
   for i := 0; i<b.N; i++
   {
      r = fibo3(n)
   }
   result = r
}
```

The preceding code defines two more benchmark functions that will not run automatically because they begin with a lowercase b instead of an uppercase B.

The functions BenchmarkFibo1(), BenchmarkFibo2(), and BenchmarkFibo3(), would not have been invoked automatically by the go test command because their signature is not func(*testing.B). So this is the reason for naming them with a lowercase b. However, there is nothing that prevents you from invoking them from other benchmark functions afterwards, as you will see shortly.

The third part of benchmarkMe_test.go follows next:

```
func Benchmark30fibo1(b *testing.B)
{
   benchmarkfibo1(b, 30)
}
```

- This is a correct benchmark function with the correct name and the correct signature, which means that it will get executed by go tool. Note that although Benchmark30fibo1() is a valid benchmark function name, benchmarkfibo1() is not because there is no uppercase letter or a number after the Benchmark string. This is very important because a benchmark function with an incorrect name will not get executed automatically.
- The fourth code segment of benchmarkMe_test.go contains the following Go code:

```
func Benchmark30fibo2(b *testing.B)
{
    benchmarkfibo2(b, 30)
}
func Benchmark30fibo3(b *testing.B)
{
    benchmarkfibo3(b, 30)
}
```

- Both the Benchmark30fibo2() and Benchmark30fibo3() benchmark functions are similar to Benchmark30fibo1().
- The last part of benchmarkMe_test.go is as follows:

```
func Benchmark50fibo1(b *testing.B)
{
    benchmarkfibo1(b, 50)
}
func Benchmark50fibo2(b *testing.B)
{
    benchmarkfibo2(b, 50)
}
func Benchmark50fibo3(b *testing.B)
{
    benchmarkfibo3(b, 50)
}
```

- In this part, you see three additional benchmark functions that calculate the 50th number in the Fibonacci sequence.
- Remember that each benchmark is executed for at least 1 second by default. If the benchmark function returns in a time that is less than 1 second, the value of b.N is increased and the function is run again. The first time the value of b.N is 1, it becomes 2, 5, 10, 20, 50, and so on.
- This happens because the faster the function, the more times you need to run it to get accurate results.

Executing benchmarkMe_test.go in Linux will generate the following output:

```
$ go test -bench=. benchmarkMe.gobenchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fibo1-8 300 4494213 ns/op
Benchmark30fibo2-8 300 4463607 ns/op
Benchmark30fibo3-8 500000 2829 ns/op
Benchmark50fibo1-8 1 67272089954 ns/op
Benchmark50fibo2-8 1 67300080137 ns/op
Benchmark50fibo3-8 300000 4138 ns/op
PASS
ok command-line-arguments 145.827s
```

There are two important points to be noted here: first, the value of the -bench parameter specifies the benchmark functions that will be executed. The value used is a regular expression that matches all valid benchmark functions. The second point is that if you omit the bench parameter, no benchmark function will be executed.

As you can see, the fibo1() and fibo2() functions are really slow compared to the fibo3() function. To include memory allocation statistics in the output, we can execute the following command:

```
$ go test -benchmem -bench=.benchmarkMe.gobenchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fibo1-8 300 4413791 ns/op 0 B/op 0 allocs/op
Benchmark30fibo2-8 300 4430097 ns/op 0 B/op 0 allocs/op
Benchmark30fibo3-8 500000 2774 ns/op 2236 B/op 6 allocs/op
Benchmark50fibo1-8 1 71534648696 ns/op 0 B/op 0 allocs/op
Benchmark50fibo2-8 1 72551120174 ns/op 0 B/op 0 allocs/op
Benchmark50fibo3-8 300000 4612 ns/op 2481 B/op 10 allocs/op
PASS
ok command-line-arguments 150.500s
```

The preceding output is similar to the one without the - benchmem command-line parameter, but it includes two more columns in its output. The fourth column shows the amount of memory that was allocated on an average in each execution of the benchmark function. The fifth column shows the number of allocations used for allocating the memory value of the fourth column. So, Benchmark50fibo3() allocated 2481 bytes in 10 allocations on average.

The functions fibo1() and fibo2() do not need any special kind of memory apart from the expected one, because they are not using any kind of data structure, which is not the case with fibo3() which uses a map variable; hence the larger than zero values in both the fourth and fifth columns of the output of Benchmark10fibo3-8.

A wrong benchmark function:

- Look at the Go code of the following benchmark function:

```
funcBenchmarkFiboI(b *testing.B)
{
    for i := 0; i<b.N; i++
    {
        _ = fibo1(i)
    }
}
```

- The BenchmarkFibo() function has a valid name and the correct signature. The point to note is that this benchmark function is wrong, and you will not get any output from it after executing the go test command. The reason for this is that as the b.N value grows according to the way described before, the run time of the benchmark function will also increase.

6.9 WORKING WITH FILES

The io.Reader and io.Writer interfaces

- Agreement with the io.Reader interface requires the implementation of the Read() method, whereas if you want to satisfy the io.Writer interface guidelines, you will need to implement the Write() method. Both these interfaces are very important in Go.

Buffered and unbuffered file input and output

- Buffered file input and output happens when there is a buffer for temporarily storing data before reading data or writing data. Thus, instead of reading a file byte by byte, you read many bytes at once. You put it in a buffer and wait for someone to read it in the desired way. Unbuffered file input and output happens when there is no buffer to temporarily store data before actually reading or writing it.
- The next question that you might ask is how to decide when to use buffered and when to use unbuffered file input and output. When dealing with critical data, unbuffered file input and output is generally a better choice because buffered reads might result in out-of-date data and buffered writes might result in data loss when the power of your computer is interrupted. However, most of the time, there is no definitive answer to that question. This means that you can use whatever makes your tasks easier to implement.

The bufio package

- As the name suggests, the bufio package is about buffered input and output. However, the bufio package still uses (internally) the io.Reader and io.Writer objects, which it wraps in order to create the bufio.Reader and bufio.Writer objects, respectively.
- As you will see in the forthcoming sections, the bufio package is very popular for reading text files.

Reading text files

A text file is the most common kind of file that you can find on a Unix system. In this section, you will learn how to read text files in three ways: line by line, word by word, and character by character. As you will see, reading a text file line by line is the easiest method to access a text file, while reading a text file word by word is the most difficult method of all.

If you look closely at the byLine.go, byWord.go, and byCharacter.go programs, you will see many similarities in their Go code. First, all three utilities read the input file line by line. Second, all three utilities have the same main() function, with the exception of the function that is called in the for loop of the main() function. Last, the three functions that process the input text files are almost identical, except for the part that implements the actual functionality of the function.

Reading a text file line by line

Going line by line is the most common method of reading a text file. This is the main reason it is being shown first. The Go code of byLine.go, which will be presented in three parts, will help you understand the technique.

Program 6.4: The first code segment from byLine.go is shown in the following Go code:

```
package main
import (
"bufio"
"flag"
"fmt"
"io"
"os"
)
```

As you can see by the presence of the bufio package, we will use buffered input.

The second part of byLine.go contains the following Go code:

```
func lineByLine(file string) error
{
   var err error
   f, err := os.Open(file)
   if err != nil
   {
      return err
   }
   Defer f.Close()
   r := bufio.NewReader(f)
```

```
    for
    {
        line, err := r.ReadString('\n')
        if err == io.EOF
        {
            break
        }
        else if err != nil
        {
            fmt.Printf("error reading file %s", err)
            break
        }
        fmt.Print(line)
    }
    return nil
}
```

- The main part is done in the lineByLine() function. After making sure that we can open the given filename for reading, we create a new reader using bufio.NewReader(). Then you use that reader with bufio.ReadString() in order to read the input file line by line. The trick is done by the parameter of bufio.ReadString(), which is a character that tells bufio.ReadString() to keep reading until that character is found. Constantly calling bufio.ReadString() when that parameter is the newline character results in reading the input file line by line. Note that the use of fmt.Print() for printing the read line instead of fmt.Println() shows that the newline character is included in each input line.
- The third part of byLine.go follows next:

```
func main()
{
    flag.Parse()
    if len(flag.Args()) == 0
    {
        fmt.Printf("usage: byLine<file1> [<file2> ...]\n")
        return
    }
    for _, file := range flag.Args()
    {
        err := lineByLine(file)
        if err != nil
        {
            fmt.Println(err)
        }
    }
}
```

- Executing byLine.go and processing its output with wc(1) will generate the following type of output:

```
$ go run byLine.go /tmp/swtag.log /tmp/adobegc.log | wc
4761 88521 568402
```

- The following command will verify the accuracy of the preceding output:

```
$ wc /tmp/swtag.log /tmp/adobegc.log
131 693 8440 /tmp/swtag.log
4630 87828 559962 /tmp/adobegc.log
4761 88521 568402 total
```

Reading a text file word by word

- The technique presented in this subsection will be demonstrated by the byWord.go file, which is shown in four parts. As you will see in the Go code, separating the words of a line can be tricky.

Program 6.5: The first part of this utility is as follows:

```
package main
import (
"bufio"
"flag"
"fmt"
"io"
"os"
"regexp"
)
```

The second code portion of byWord.go is shown in the following Go code:

```
func wordByWord(file string) error
{
   var err error
   f, err := os.Open(file)
   if err != nil
   {
      return err
   }
   defer f.Close()
   r := bufio.NewReader(f)
   for
   {
      line, err := r.ReadString('\n')
      if err == io.EOF
      {
         break
      }
```

```
        else if err != nil
        {
            fmt.Printf("error reading file %s", err)
            return err
        }
```

This part of the wordByWord() function is the same as the lineByLine() function of the byLine.go utility.

The third part of byWord.go is as follows:

```
        r := regexp.MustCompile("[^\\s]+")
        words := r.FindAllString(line, -1)
        for i := 0; i<len(words); i++
        {
            fmt.Println(words[i])
        }
    }
    return nil
}
```

The remaining code of the wordByWord() function is totally new, and it uses regular expressions to separate the words found in each line of the input. The regular expression defined in the regexp.MustCompile("[^\\s]+") statement states that empty characters will separate one word from another.

The last code segment of byWord.go is as follows:

```
func main()
{
    flag.Parse()
    if len(flag.Args()) == 0
    {
        fmt.Printf("usage: byWord<file1> [<file2> ...]\n")
        return
    }
    for _, file := range flag.Args()
    {
        err := wordByWord(file)
        if err != nil
        {
            fmt.Println(err)
        }
    }
}
```

Executing byWord.go will produce the following type of output:

```
$ go run byWord.go /tmp/adobegc.log
01/08/18
20:25:09:669
|
```

You can verify the validity of byWord.go with the help of the wc(1) utility:

```
$ go run byWord.go /tmp/adobegc.log | wc
91591 91591 559005
$ wc /tmp/adobegc.log
4831 91591 583454 /tmp/adobegc.log
```

- As you can see, the number of words calculated by wc(1) is the same as the number of lines and words that you took from the execution of byWord.go.

Reading a text file character by character

- In this section, you will learn how to read a text file character by character, which is a pretty rare requirement unless you want to create a text editor. The relevant Go code will be saved as byCharacter.go, which will be presented in four parts.

Program 6.6:

The first part of byCharacter.go is shown in the following Go code:

```
package main
import (
"bufio"
"flag"
"fmt"
"io"
"os"
)
```

As you can see, you will not need to use regular expressions for this task.

The second code segment from byCharacter.go is as follows:

```
func charByChar(file string) error
{
   var err error
   f, err := os.Open(file)
   if err != nil
   {
      return err
   }
   defer f.Close()
   r := bufio.NewReader(f)
```

```
    for
    {
        line, err := r.ReadString('\n')
        if err == io.EOF
        {
            break
            www.EBooksWorld.ir
            Telling a Unix System What to Do Chapter 8
        } else if err != nil
        {
            fmt.Printf("error reading file %s", err)
            return err
        }
```

The third part of byCharacter.go is where the logic of the program is found:

```
        for _, x := range line
        {
            fmt.Println(string(x))
        }
    }
    return nil
}
```

Here you take each line that you read and split it using range. The range returns two values: You discard the first, which is the location of the current character in the line variable, and you use the second. However, that value is a not a character that is the reason why you have to convert it into a character using the string() function.

Note that due to the fmt.Println(string(x)) statement, each character is printed in a distinct line, which means that the output of the program will be large. If you want a more compressed output, you should use the fmt.Print() function instead.

The last part of byCharacter.go contains the following Go code:

```
func main()
{
    flag.Parse()
    if len(flag.Args()) == 0
    {
        fmt.Printf("usage: byChar<file1> [<file2> ...]\n")
        return
    }
    for _, file := range flag.Args()
    {
        err := charByChar(file)
```

```
        if err != nil
        {
            fmt.Println(err)
        }
    }
}
```

The execution of byCharacter.go will generate the following type of output:

```
$ go run byCharacter.go /tmp/adobegc.log
0
1
/
0
8
/
1
8
```

- Note that the Go code presented here can be used for counting the number of characters found in the input file, which can help you implement a Go version of the handy wc(1) command-line utility.

Reading from /dev/random:

- In this section, you will learn how to read from the /dev/random system device. The purpose of the /dev/random system device is to generate random data, which you might use for testing your programs or, in this case, you will plant the seed for a random number generator. Getting data from /dev/random can be a difficult task.
- On a macOS High Sierra machine, the /dev/random file has the following permissions:

```
$ ls -l /dev/random
crw-rw-rw- 1 root wheel 14, 0 Jan 8 20:24 /dev/random
```

- Similarly, on a Debian Linux machine, the /dev/random system device has the following Unix file permissions:

```
$ ls -l /dev/random
crw-rw-rw- 1 root root 1, 8 Jan 13 12:19 /dev/random
```

- This means that the /dev/random file has analogous file permissions on both Unix variants. The only difference between these two Unix variants is the group that owns the file, which is wheel on macOS and root on Debian Linux, respectively.
- The name of the program for this topic is devRandom.go, and it will be presented in three parts.

Program 6.7: The first part of the program is as follows:

```
package main
import (
"encoding/binary"
"fmt"
"os"
)
```

In order to read from /dev/random, you will need to import the encoding/binary standard Go package, because /dev/random returns binary data that needs to be decoded.

The second code portion of devRandom.go follows next:

```
func main()
{
    f, err := os.Open("/dev/random")
    defer f.Close()
    if err != nil
    {
        fmt.Println(err)
        return
    }
```

You open /dev/random as usual because everything in Unix is a file.

The last code segment of devRandom.go is shown in the following Go code:

```
    var seed int64
    binary.Read(f, binary.LittleEndian, &seed)
    fmt.Println("Seed:", seed)
}
```

You need the binary.Read() function, which requires three parameters, in order to read from the /dev/random system device. The value of the second parameter (binary.LittleEndian) specifies that you want to use the little endian byte order. The other option is binary.BigEndian, which is used when your computer is using the big endian byte order.

Executing devRandom.go will generate the following type of output:

```
Seed: -2044736418491485077
```

Reading the amount of data you want from a file

- In this section, you will learn how to read exactly the amount of data you want. This technique is particularly useful when reading binary files, where you have to decode the data you read in a particular way. Nevertheless, this technique still works with text files. The logic behind this technique is not that difficult: You create a byte slice with the size you need and use that byte slice for reading. To make this more

interesting, this functionality is going to be implemented as a function with two parameters. One parameter will be used for specifying the amount of data that you want you read, and the other parameter, which will have the *os.File type, will be used for accessing the desired file. The return value of that function will be the data you have read.

The name of the Go program for this topic will be readSize.go, and it will be presented in four parts. The utility will accept a single parameter, which will be the size of the byte slice. This particular program, when used with the present technique, can help you copy any file using the buffer size you want.

Program 6.8: The first part of readSize.go has the expected preamble:

```
package main
import (
"fmt"
"io"
"os"
"strconv"
)
```

The second part of readSize.go contains the following Go code:

```
func readSize(f *os.File, size int) []byte
{
   buffer := make([]byte, size)
   n, err := f.Read(buffer)
   if err == io.EOF
   {
      return nil
   }
   if err != nil
   {
      fmt.Println(err)
      return nil
   }
   return buffer[0:n]
}
```

This is the function discussed earlier. Although the code is straightforward, there is one point that needs an explanation. The io.Reader.Read() method returns two parameters: the number of bytes read as well as an error variable. The readSize() function uses the former return value of io.Read() in order to return a byte slice of that size. Although this is a tiny detail, and it is only significant when you reach the end of the file, it ensures that the output of the utility will be same as the input and that it will not contain any extra characters. Finally, there is code that checks for io.EOF, which is an error that signifies that you have reached the end of a file. When that kind of error occurs, the function returns.

The third code portion of this utility is as follows:

```
func main()
{
    arguments := os.Args
    if len(arguments) != 3
    {
        fmt.Println("<buffer size><filename>")
        return
    }
    bufferSize, err := strconv.Atoi(os.Args[1])
    if err != nil
    {
        fmt.Println(err)
        return
    }
    file := os.Args[2]
    f, err := os.Open(file)
    if err != nil
    {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

The last code segment of readSize.go is as follows:

```
    for
    {
        readData := readSize(f, bufferSize)
        if readData != nil
        {
            fmt.Print(string(readData))
        } else
        {
            break
        }
    }
}
```

So, here we can keep reading the input file until readSize() returns an error or nil. Executing readSize.go by telling it to process a binary file and handling its output with wc(1) will validate the correctness of the program:

```
$ go run readSize.go 1000 /bin/ls | wc
80 1032 38688
$ wc /bin/ls
80 1032 38688 /bin/ls
```

- In the previous section, the readSize.go utility illustrated how you can read a file byte by byte, which is a technique that best applies to binary files. Imagine that you want to store the number 20 as a string to a file. It is easy to understand that you will need two bytes for storing 20 using ASCII characters, one for storing 2 and another for storing 0. Storing 20 in binary format requires just one byte, since 20 can be represented as 00010100 in binary or as 0x14 in hexadecimal.
- This difference might look insignificant when you are dealing with small amounts of data, but it could be pretty substantial when dealing with data found in applications such as database servers.

Reading CSV files:

- **CSV files** are plain text files with a format. In this section, you will learn how to read a text file that contains points of a plane, which means that each line will contain a pair of coordinates. Additionally, you are also going to use an external Go library named Glot, which will help you create a plot of the points that you read from the CSV file. Note that **Glot** uses **Gnuplot**, which means that you will need to install Gnuplot on your Unix machine in order to use Glot.
- The name of the source file for this topic is CSVplot.go, and it is going to be presented in five parts.

Program 6.9: The first code segment is as follows:

```
package main
import (
"encoding/csv"
"fmt"
"github.com/Arafatk/glot"
"os"
"strconv"
)
```

The second part of CSVplot.go is shown in the following Go code:

```
func main()
{
   if len(os.Args) != 2
   {
      fmt.Println("Need a data file!")
      return
   }
   file := os.Args[1]
   _, err :=os.Stat(file)
   if err != nil
   {
      fmt.Println("Cannot stat", file)
      return
   }
```

In this part, you see a technique for checking whether a file already exists or not using the powerfulos.Stat() function.

The third part of CSVplot.go is as follows:

```
f, err := os.Open(file)
if err != nil
{
    fmt.Println("Cannot open", file)
    fmt.Println(err)
    return
}
defer f.Close()
reader := csv.NewReader(f)
reader.FieldsPerRecord = -1
allRecords, err := reader.ReadAll()
if err != nil
{
    fmt.Println(err)
    return
}
```

The fourth code segment of CSVplot.go is shown in the following Go code:

```
xP := []float64{}
yP := []float64{}
for _, rec := range allRecords
{
    x, _ := strconv.ParseFloat(rec[0], 64)
    y, _ := strconv.ParseFloat(rec[1], 64)
    xP = append(xP, x)
    yP = append(yP, y)
}
points := [][]float64{}
points = append(points, xP)
points = append(points, yP)
fmt.Println(points)
```

Here you convert the string values you read into numbers and put them into a slice with two dimensions named points.

The last part of CSVplot.go contains the following Go code:

```
dimensions := 2
persist := true
debug := false
plot, _ := glot.NewPlot(dimensions, persist, debug)
```

```
    plot.SetTitle("Using Glot with CSV data")
    plot.SetXLabel("X-Axis")
    plot.SetYLabel("Y-Axis")
    style := "circle"
    plot.AddPointGroup("Circle:", style, points)
    plot.SavePlot("output.png")
}
```

In the preceding Go code, we have seen how to create a PNG file with the help of the Glot library and its glot.SavePlot() function.

We need to download the Glot Go library before being able to compile and execute the CSVplot.go source code, which requires the execution of the following command from the Unix shell:

```
$ go get github.com/Arafatk/glot
```

You can see the results of the preceding command by looking inside ~/go directory:

```
$ ls -l ~/go/pkg/darwin_amd64/github.com/Arafatk/
total 240
-rw-r--r-- 1 mtsouk staff 119750 Jan 22 22:12 glot.a
$ ls -l ~/go/src/github.com/Arafatk/glot/
total 120
-rw-r--r-- 1 mtsouk staff 1818 Jan 22 22:12 README.md
-rw-r--r-- 1 mtsouk staff 6092 Jan 22 22:12 common.go
-rw-r--r-- 1 mtsouk staff 552 Jan 22 22:12 common_test.go
-rw-r--r-- 1 mtsouk staff 3162 Jan 22 22:12 core.go
-rw-r--r-- 1 mtsouk staff 138 Jan 22 22:12 core_test.go
-rw-r--r-- 1 mtsouk staff 3049 Jan 22 22:12 function.go
-rw-r--r-- 1 mtsouk staff 511 Jan 22 22:12 function_test.go
-rw-r--r-- 1 mtsouk staff 4955 Jan 22 22:12 glot.go
-rw-r--r-- 1 mtsouk staff 220 Jan 22 22:12 glot_test.go
-rw-r--r-- 1 mtsouk staff 10536 Jan 22 22:12 pointgroup.go
-rw-r--r-- 1 mtsouk staff 378 Jan 22 22:12 pointgroup_test.go
```

The CSV data file containing the points that will be plotted has the following format:

```
$ cat /tmp/dataFile
1,2
2,3
3,3
4,4
5,8
6,5
-1,12
-2,10
-3,10
-4,10
```

Executing CSVplot.go will generate the following kind of output:

```
$ go run CSVplot.go /tmp/doesNoExist
Cannot stat /tmp/doesNoExist
$ go run CSVplot.go /tmp/dataFile
[[1 2 3 4 5 6 -1 -2 -3 -4] [2 3 3 4 8 5 12 10 10 10]]
```

We can see the results of CSVplot.go in a much better format in the following figure:

The type of graphical output you can get from Glot.

Writing to a file:

- Generally speaking, you can use the functionality of the io.Writer interface for writing data to files on a disk. Nevertheless, the Go code of save.go will show you five ways to write data to a file. The save.go program will be presented in six parts.

Program 6.10: The first part of save.go is as follows:

```
package main
import (
"bufio"
"fmt"
"io"
"io/ioutil"
"os"
)
```

The second code portion of save.go is shown in the following Go code:

```
func main()
{
   s := []byte("Data to write\n")
   f1, err :=os.Create("f1.txt")
   if err != nil
   {
      fmt.Println("Cannot create file", err)
   return
   }
   defer f1.Close()
   fmt.Fprintf(f1, string(s))
```

Understand that the 's' byte slice will be used in every line that involves writing presented in this

Go program. Additionally, the fmt.Fprintf() function used here can help you write data to your own log files using the format you want. In this case, fmt.Fprintf() writes your data to the file identified by f1.

The third part of save.go contains the following Go code:

```
f2, err :=os.Create("f2.txt")
if err != nil
{
   fmt.Println("Cannot create file", err)
   return
}
defer f2.Close()
n, err := f2.WriteString(string(s))
fmt.Printf("wrote %d bytes\n", n)
```

In this case, f2.WriteString() is used for writing your data to a file.

The fourth code segment of save.go follows next:

```
f3, err :=os.Create("f3.txt")
if err != nil
{
   fmt.Println(err)
   return
}
w := bufio.NewWriter(f3)
n, err = w.WriteString(string(s))
fmt.Printf("wrote %d bytes\n", n)
w.Flush()
```

In this case, bufio.NewWriter() opens a file for writing and bufio.WriteString() writes the data.

The fifth part of save.go will teach you another method for writing to a file:

```
f4 := "f4.txt"
err = ioutil.WriteFile(f4, s, 0644)
if err != nil
{
   fmt.Println(err)
   return
}
```

This method needs just a single function call named ioutil.WriteFile() for writing your data, and it does not require the use of os.Create().

The last code segment of save.go is as follows:

```
f5, err :=os.Create("f5.txt")
if err != nil
{
   fmt.Println(err)
   return
}
n, err = io.WriteString(f5, string(s))
```

```
    if err != nil
    {
        fmt.Println(err)
        return
    }
    fmt.Printf("wrote %d bytes\n", n)
}
```

The last technique uses io.WriteString() to write the desired data to a file.

Executing save.go will create the following type of output:

```
$ go run save.go
wrote 14 bytes
wrote 14 bytes
wrote 14 bytes
$ ls -l f?.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f1.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f2.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f3.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f4.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f5.txt
$ cat f?.txt
Data to write
Data to write
Data to write
Data to write
Data to write
```

The next section will show you how to save data to a file with the help of a specialized function of a package that is in the standard Go library

File permissions

- An important topic in Unix systems programming is Unix file permissions. In this section, you will learn how to print the permissions of any file, provided that you have adequate Unix permission to do so. The name of the program is permissions.go, and it will be presented in three parts.

Program 6.11: The first part of permissions.go contains the following Go code:

```
package main
import (
"fmt"
"os"
)
```

The second code segment of permissions.go is shown in the following Go code:

```
func main()
{
   arguments := os.Args
   if len(arguments) == 1
   {
      fmt.Printf("usage: permissions filename\n")
      return
   }
```

The last part of this utility follows next:

```
   filename := arguments[1]
   info, _ := os.Stat(filename)
   mode := info.Mode()
   fmt.Println(filename, "mode is", mode.String()[1:10])
}
```

The call to os.Stat(filename) returns a big structure with lots of data. As we are only interested in the permissions of the file, we will call the Mode() method and print its output. Actually, we are printing a part of the output denoted by mode.String()[1:10] because this is where the data that interests us is found.

Executing permissions.go will create the following type of output:

```
$ go run permissions.go /tmp/adobegc.log
/tmp/adobegc.log mode is rw-rw-rw-
$ go run permissions.go /dev/random
/dev/random mode is crw-rw-rw
```

The output of the ls(1) utility verifies the correctness of permissions.go:

```
$ ls -l /dev/random /tmp/adobegc.log
crw-rw-rw- 1 root wheel 14, 0 Jan 8 20:24 /dev/random
-rw-rw-rw- 1 root wheel 583454 Jan 16 19:12 /tmp/adobegc.log
```

Summary

- In this chapter, you have studied about Go packages and Files. We have provided ample discussion about developing good applications to demonstrate Go packages and Files.
- Packages are an important and powerful part of the Go language.
- In Go language, every package is defined with a different name and that name is close to their purpose like "strings" package which contains methods and functions that are only related to strings.
- A package is a collection of source files in the same directory that are compiled together. A workspace is a directory hierarchy with two directories at its root: src contains Go source files, and bin contains executable commands.

- An import path is a string that uniquely identifies a package. A package's import path corresponds to its location inside a workspace or in a remote repository.
- Blank identifier in importing packages means specifying a blank import for the imported package.
- Unit testing is very common in Go (Golang) code. Many of the types of applications that Go is frequently used for, are very easy to write good unit tests.
- Testing is one of the main and important activities in programming, but often it's neglected or left as an afterthought.
- Benchmarking can give you information about the performance of a function or a program in order to understand better how much faster or slower a function is compared to another function, or compared to the rest of the application.
- Buffered file input and output happens when there is a buffer for temporarily storing data before reading data or writing data.

Check Your Understanding

1. What is workspace in GO?
   (a) src contains GO source files organized into packages
   (b) pkg contains package objects
   (c) bin contains executable commands
   (d) All of these
2. Which of the following is a derived type in Go?
   (a) Interface types (b) Map types
   (c) Channel types (d) All of the above
3. What are the advantages of GO?
   (a) GO compiles very quickly
   (b) Go supports concurrency at the language level
   (c) Functions are first class objects in GO
   (d) All of these
4. What are the benefits of using Go Programming?
   (a) Support for environment adopting patterns similar to dynamic languages.
   (b) Compilation time is fast
   (c) In Built concurrency support: light-weight processes (via goroutines), channels, select statement.
   (d) All of the above.

Answers

1. (d)	2. (d)	3. (d)	4. (d)

Trace The Output (only for programming languages)

1. Golang program to read and write the files.

```
package main
// importing the packages
import (
   "fmt"
   "io/ioutil"
   "log"
   "os"

)
func CreateFile()
{
   // fmt package implements formatted
   // I/O and has functions like Printf
   // and Scanf
   fmt.Printf("Writing to a file in Go lang\n")
   // in case an error is thrown it is received
   // by the err variable and Fatalf method of
   // log prints the error message and stops
   // program execution
   file, err := os.Create("test.txt")
   if err != nil
   {
      log.Fatalf("failed creating file: %s", err)
   }
   // Defer is used for purposes of cleanup like
   // closing a running file after the file has
   // been written and main //function has
   // completed execution
   defer file.Close()
   // len variable captures the length
   // of the string written to the file.
   len, err := file.WriteString("Welcome to Go Programming."+
         "This program demonstrates reading and writing"+
                  "operations to a file in Go lang.")
   if err != nil
   {
      log.Fatalf("failed writing to file: %s", err)
   }
```

```
    // Name() method returns the name of the
    // file as presented to Create() method.
    fmt.Printf("\nFile Name: %s", file.Name())
    fmt.Printf("\nLength: %d bytes", len)
}
func ReadFile()
{
    fmt.Printf("\n\nReading a file in Go lang\n")
    fileName := "test.txt"
    // The ioutil package contains inbuilt
    // methods like ReadFile that reads the
    // filename and returns the contents.
    data, err := ioutil.ReadFile("test.txt")
    if err != nil
    {
        log.Panicf("failed reading data from file: %s", err)
    }
    fmt.Printf("\nFile Name: %s", fileName)
    fmt.Printf("\nSize: %d bytes", len(data))
    fmt.Printf("\nData: %s", data)
}
// main function
func main()
{
    CreateFile()
    ReadFile()
}
```

Practice Questions

Q. I Answer the following questions in short.

1. What is a Package in Go Programming?
2. Briefly explain the process of reading from a Text file.
3. Briefly explain how package names are imported.
4. What are named Imports?
5. What are blank Imports?
6. What is a Workspace?

Q. II Answer the following questions.

1. Write a note on benchmarking?
2. Write a note on Table Tests and Random Tests?
3. What is Unit testing with Test functions?

Q. III Define the following terms:

1. Benchmarking
2. Go package
3. Blank imports
4. Unit Testing
5. Packages
6. Workspaces

Programs for Practice

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

```
package main
import (
   "fmt"
)
type Calculator struct
{
   a int
   b int
}
func (c *Calculator)Add()
{
   fmt.Println("Addition of two numbers: ", c.a + c.b)
}
func (c *Calculator)Mul()
{
   fmt.Println("Multiplication of two numbers: ", c.a * c.b)
}
func (c *Calculator)Div()
{
   fmt.Println("Division of two numbers: ", c.a / c.b)
}
func (c *Calculator)Sub()
{
   fmt.Println("Subtraction of two numbers: ", c.a - c.b)
}
```

```
func main()
{
   var a, b int
   fmt.Print("Enter the first number: ")
   fmt.Scanf("%d", &a)
   fmt.Print("Enter the second number: ")
   fmt.Scanf("%d", &b)
   cal := Calculator
   {
      a: a,
      b: b,
   }
   c:=1
   for c>=1
   {
      fmt.Println("Enter 1 for Addition: ")
      fmt.Println("Enter 2 for Multiplication: ")
      fmt.Println("Enter 3 for Division: ")
      fmt.Println("Enter 4 for Subtraction: ")
      fmt.Print("Enter 5 for Exit: ")
      fmt.Scanf("%d", &c)
      switch c
      {
         case 1:
            cal.Add()
         case 2:
            cal.Mul()
         case 3:
            cal.Div()
         case 4:
            cal.Sub()
         case 5:
            c = 0
            break
         default:
         fmt.Println("Enter valid number.")
      }
   }
}
```

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

```
package main
import (
   "fmt"
)
type Rectangle struct
{
   breadth int
   len int
}
func (r *Rectangle)Area() int
{
   return r. len * r.breadth
}
func main()
{
   rectangle := Rectangle
   {
      breadth: 10,
      len: 8,
   }
   fmt.Println("Area of the rectangle:", rectangle, " is: ",
   rectangle.Area())
}
```

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

```
package math
import "testing"
func TestAdd(t *testing.T)
{
    got := Add(4, 6)
    want := 10
    if got != want
    {
        t.Errorf("got %q, wanted %q", got, want)
    }
}
```

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

```
package math
import "testing"
// arg1 means argument 1 and arg2 means argument 2, and the expected
stands for the 'result we expect'
type addTest struct
{
    arg1, arg2, expected int
}
var addTests = []addTest
{
    addTest{2, 3, 5},
    addTest{4, 8, 12},
    addTest{6, 9, 15},
    addTest{3, 10, 13},
}
func TestAdd(t *testing.T)
{
   for _, test := range addTests
   {
      if output := Add(test.arg1, test.arg2); output != test.expected
      {
         t.Errorf("Output %q not equal to expected %q", output,
         test.expected)
      }
   }
}
```

5. Write a function in Go language which computes all of the prime numbers between one and an integer and write a benchmark for it.

```
// main.go
func primeNumbers(max int) []int
{
   var primes []int
   for i := 2; i < max; i++
   {
      isPrime := true
      for j := 2; j <= int(math.Sqrt(float64(i))); j++
      {
```

```
            if i%j == 0
            {
                isPrime = false
                break
            }
        }
        if isPrime
        {
            primes = append(primes, i)
        }
    }
    return primes
}
```

The function above determines if a number is a prime number by checking whether it is divisible by a number between two and its square root. Next we write a benchmark for this function in main_test.go:

```
package main
import (
    "testing"
)
var num = 1000
func BenchmarkPrimeNumbers(b *testing.B)
{
    for i := 0; i < b.N; i++
    {
        primeNumbers(num)
    }
}
```

Like unit tests in Go, benchmark functions are placed in a _test.go file, and each benchmark function is expected to have func BenchmarkXxx(*testing.B) as a signature, with the testing.B type managing the benchmark's timing.

- b.N specifies the number of iterations; the value is not fixed, but dynamically allocated, ensuring that the benchmark runs for at least one second by default.
- In the BenchmarkPrimeNumbers() function above, the primeNumbers() function will be executed b.N times until the developer is satisfied with the stability of the benchmark.

Running a benchmark in Go

- To run a benchmark in Go, we'll append the -bench flag to the go test command. The argument to -bench is a regular expression that specifies which benchmarks should be run, which is helpful when you want to run a subset of your benchmark functions.
- To run all benchmarks, use -bench=., as shown below:

```
$ go test -bench=.
goos: linux
goarch: amd64
pkg: github.com/ayoisaiah/random
cpu: Intel(R) Core(TM) i7-7560U CPU @ 2.40GHz
BenchmarkPrimeNumbers-4     14588     82798 ns/op
PASS
ok     github.com/ayoisaiah/random     2.091s
```

goos, goarch, pkg, and cpu describe the operating system, architecture, package, and CPU specifications, respectively. BenchmarkPrimeNumbers-4 denotes the name of the benchmark function that was run. The -4 suffix denotes the number of CPUs used to run the benchmark, as specified by GOMAXPROCS

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

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:

Example:

```
<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)
}
```

7. **WAP in Go language to print file information.**

```
package main
import (
    "fmt"
    "log"
    "os"
)
func main()
{
    fileStat, err := os.Stat("test.txt")
    if err != nil
    {
        log.Fatal(err)
    }
    fmt.Println("File Name:", fileStat.Name()) // Base name of the file
    fmt.Println("Size:", fileStat.Size())
                                        // Length in bytes for regular files
    fmt.Println("Permissions:", fileStat.Mode())      // File mode bits
    fmt.Println("Last Modified:", fileStat.ModTime())
                                                  // Last modification time
    fmt.Println("Is Directory: ", fileStat.IsDir())
                                          // Abbreviation for Mode().IsDir()
}
```

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

```
package main
import (
    "fmt"
    "os"
)
func main()
{
    message := "Add this content at end of Go file"
    filename := "test.txt"
    f, err := os.OpenFile(filename, os.O_RDWR|os.O_APPEND|os.O_CREATE,
    0660)
    if err != nil
    {
        fmt.Println(err)
        os.Exit(-1)
    }
    defer f.Close()
    fmt.Fprintf(f, "%s\n", message)
}
```

■■■