A GUIDE TO

PROGRAMMING WITH

TABLE OF CONTENTS

INTRODUCTION

Welcome to Essentials of Google Go - also known as Golang - programming language. In this chapter, we introduce you to the world of the Go programming language and provide you with the knowledge you need to get started programming with this new and exciting programing language.

Are you ready?

Let's start with the biggest question of them all - what is the Google Go Programming language?

Go is an open-source programming language that was sponsored by Google in 2007. This programming language was developed by Robert Griesemer, Rob Pike, and Ken Thompson, who at the time were working for Google. Go is a snappy, expressive and concurrent programming language that simplifies development of software systems.

The Go language was originally meant for C and C++ programmers (it borrows heavily from these two programming languages) who wanted a snappy programming language that's not only expressive but also supports concurrency. Think of Go as a minimal programming (heavily redacted C and C++) language that supports polymorphism, object orientation, generic programming, and abstraction.

If you're a C or C++ programmer, who would like to learn a programming language that simplifies the coding process, then look no further than the Go programming language.

Why use the Go programming language?

Go programming fundamentally flips the way you've been programming. It doesn't matter whether you're coming from C, C++, Python or Ruby programming languages. You'll find this programming language simple in expressing polymorphism, object orientation, generic programming, and a bstraction.

Here are reasons why you should try out this programming language:

#1: It's a compiled language

In this language, the source codes you that you enter will be reduced to a set of machine-specific instructions before they are saved as executable files. Because of this features, it's fast since the executable files will run faster compared to other programming languages. But, that's not all. You can also edit and run the Go codes directly on the web when you use this programming language.

#2: There's efficient memory management

The Go language is a garbage-collected language. When you use this programming language, you'll

not worry about how memory is being managed since this will be well taken care of. If you're a Java programmer, you'll find memory management similar as opposed to C++ programming.

#3: It supports concurrency

The Go language uses go-routines to start concurrent activities and the concept of channels that promotes both communication and synchronization processes. Therefore, it's easier to implement parallelism in this language than other programming languages.

#4: It's easier to debug during compilation process

The Go language is a statically-typed language. This makes it possible for you to detect several bugs when you're compiling your program. Besides, its type inference engine allows you not to mention the variable types explicitly since this can be easily be inferred by the program's compiler.

#5: Functions are first class objects

It's easier to use the functions just like you would use the classes. This helps you to create apps that use object orientation principles such as polymorphism and abstraction.

#6: It has documentation as a feature

The Golang has documentation as its standard feature. So, if you're a developer, you'll find this language easy to develop human-readable information that's generated from the source code comments.

#7: It has a rich standard library

Just like C and C++, this programming language has a rich set of standard libraries. In fact, this is the only programming language that has the web server as part of its core libraries.

Congratulations! You have finished the first part of Go programming. You're on your way to becoming a professional Golang programmer.

GETTING STARTED WITH GOLANG

In this chapter we'll cover the basics that are necessary to get you started with Golang. By the end of this chapter, you should be in a position to set up your desktop environment ready to begin coding using Golang.

Before we start off, it's important to note that Golang is an open-source programming language. Therefore, it doesn't matter whether you're using Linux OS, Mac OS or Windows OS. So, let's dive in and find out how you can install and run Golang programs in Linux OS, Mac OS or Windows OS.

How to Install Golang on Linux OS

Whether you're using a Debian based Linux distribution - such as Ubuntu or Linux Mint - or CentOS/RHEL based distribution such as Fedora, it's pretty simple to install Golang.

1. For Debian based Linux distributions, follow the steps below:

    - Open your terminal
    - Type the following command at the command prompt: "sudo apt–get install golang"
    - Enter your root password
    - Wait for the installation process to complete

2. For CentOS/RHEL based distribution such as Fedora follows the steps below:

    - Open your terminal
    - Type the following command at the command prompt: "sudo yum install golang"
    - Enter your root password
    - Wait for the installation process to complete

Note that you can check the version of your Golang by typing the following command at the command prompt: go version. The latest version of go is 1.5.2.

How to install Golang on Mac OS

Here's how you can install Golang on your Mac OS:

- Download the Golang package file from this link.
- Open your package and follow the prompts to install the Golang on your computer.
- Your path environment variable for your Golang should be set to "/usr/local/go/bin" directory.
- Restart your computer so that the changes you've made takes effect.

How to install Golang on Windows OS

Two options are available if you want to install Golang on your Windows OS computer. You can either download and install the ".zip" file or download and install the ".msi" file.

1. For ".zip" files, here's how you can install Golang your Windows OS:

- Download the zip file from this link and extract it to the folder of your choice. For instance, you can create a Go folder in Local disk C:
- Set your GOROOT environment variable for the chosen path. This should be "C:\Go" in your case.
- Add the bin subdirectory of your GOROOT. In this case, your path environment variable will be "\Go\bin."

2. For ".msi" files, here's how you can install Golang your Windows OS:

- Download the Golang package file from this link.
- Open your package and follow the prompts to install the Golang on your computer.
- Your path environment variable for your Golang should be set to "c:\Go\bin" directory
- Restart your computer so that the changes you've made take effect.

An Overview of the Golang Application

Now that you've installed Golang on your computer, what's next? You should begin writing your Golang codes.

But first, you should understand how the Golang applications are structured.

All your Golang apps will be organized around workspaces. You can look at a workspace as directory hierarchy that contains three directories at its root. These are:

- The "src". It contains the Golang source code files that have been organized into packages.
- The "pkg". It contains the package objects – they act like namespaces. Packages can either be used from a standard library, or you can create your packages.
- The "bin". It has the executable commands.

The location of your workspace is specified by the GOPATH environment variable. The packages that are used from the Golang standard library are given short paths such as the "fmt" or the "net/http". However, if you create your packages, you should select
a base path shouldn't collide with future additions to your standard library.

So, how can you create your first application? Good question.

Here's is how you can successfully create and run a simple application in Golang:

- Select the package path and create a corresponding package directory that will be inside your workspace.
- Create a file and save it using ".go" file extension inside the directory that you've created.
- Type the code. You can use the example below:
- Build and run you program using the go tool.

So, there you have it - you've learnt how to set up the Golang environment and how to create your first application.

Next up, we look at more about Golang environments.

GETTING STARTED WITH GOLANG – MORE ABOUT THE ENVIRONMENT

In this chapter, we follow up our previous discussion on Golang environment. After this class, you should begin creating simple Golang apps.

To review, your Golang apps all your Golang apps will be organized around the workspaces. Workspaces are the directories that contain three directories at its root (the "src", the "pkg" and the "bin").

The "src" contains the Golang source code files that you'll organize around the packages. The "pkg" contains the package objects - which act as namespaces - that can be generated from either the standard libraries or you can create your packages. The "bin" has all the executable commands.

Let's dive in and find out more about the structure of Golang programs.

Structure of Golang Programs

Just like C or C++ programs, the Golang are run in the package "main". As a principle, the name of the package is the same as the last element of your import path. The packages that are used from the Golang standard library are given short paths such as the "fmt" or the "net/http".

However, this doesn't mean that you'll only be using the standard packages from the standard library. You can create your packages. If you create your packages, you should select a base path that doesn't collide with future additions to your standard library.

After the main package statement, you should specify the "imports" you'll be using in your code: The "imports" are grouped into parenthesized statements. For instance, the code below illustrates how you can specify your "imports."

```
1  import "fmt"
2  import "math"
```

Alternatively, you can use the factored statements to define your "imports". Here's an example:

```
1  import (
2          "fmt"
3          "math"
4  )
```

Whenever you import a package, it's important to refer to only its exported names. You're now wondering, "What are exported names?" Well, an exported name is a name that's accessible from outside the package. In Golang, a name is exported only if it starts with a capital letter.

For instance, “Bar” is an exported name because it’s starting with a capital letter. Similarly, “BAR” is also an exported name. However, “bar” isn’t an exported name because it doesn’t begin with a capital letter.

Golang Functions

Your Golang programs will be created around functions. Functions will act as procedure or routines that perform some operations to return a value. The Golang function that you define can take zero or more parameters. For instance, the function below takes two numbers of the type “integer” and calculates their product.

```
1  func multiply (a int, b int)
2   int {
3  return a*b
4  }
```

When you declare your variables, the type should come after the variable name.

How to write the first application in Golang

In this example, we illustrate how you’ll create your first application using Golang. Here are steps that you should follow:

1. Create a new folder where you’d like to store your application. For instance, you can create a folder named “GolangApps” in ~/src/golang/ where ~ is your home directory.

2. If you’re using Linux, open the terminal and type the following commands at the command prompt:

   - mkdir src/golang
   - mkdir src/golang/GolangApps

3. Open your text editor (Notepad, Text Edit or Gedit) and type the following code:

```
1  package main
2  import "fmt"
3  //*This how comments are written in Golang
4  func main () {
5      fmt.Println("Hello World")
6  }
```

4. Save your file using the “.go” file extension. For instance, you can save it as “hello.go”.

To run the program, do the following:

- Open up a new terminal in your computer and type in the following commands at the command prompt:
- cd src/golang/GolangApps to change to the directory where the file was saved. In this case, the “hello.go” file.
- go run hello.go to execute the file.

Now you can create any application that you want in Golang.

VARIABLE AND DECLARATION

Today we will be discussing how to declare and use variables in your Golang program. Let's start by discussing the significance of variables in any programming language.

A variable can be defined as a temporary storage location in the main memory that holds data that your programs will be using. Remember, the program that you're creating is a sequence of instructions that tells a computer to perform a particular operation(s). Such a program must use data - which can either be a constant, a fixed value or a variable value - while being executed.

The ability to understand how data management occurs in any programming language - Golang included—will set you apart from other programmers. This is because you'll be able to write effective and efficient programs if you understand how data should be managed in a program.

So, how are variables used in Golang?

The Golang variables are usually given names - which act as identifiers - just like in any other programming language. Naming your variable appropriately is an important component of any programming language. In Golang, your variable name should always begin with a letter or an underscore. It can contain letters, numbers or even the underscore character.

However, your variable name can't start with a number. At the same time, you can't use special symbols when naming your variables. Also, ensure that the maximum characters for your variable is 255. We use the "var" statement to declare a list of variables in Golang. It doesn't matter where you place your "var" statement.

It can be at the package level or the function level of your Golang code. For instance, the Golang code below illustrates this:

```
1  package main
2  import "fmt"
3  var sam, janet, ismarried bool
4  func main ()
5   {
6  var i int
7  fmt.Println (i, sam, janet, ismarried)
8  }
```

Your variable declaration can include initializers, in which case you can declare one per variable. If an initializer is present, then its type can be left out. For instance, the Golang code below illustrates this:

```
package main
import "fmt"
var sam, janet, ismarried bool=true
func main () {
var i , j int=1, 2
fmt.Println (i, j, sam, janet, ismarried)
}
```

If you're inside a function, you can use the ":=" short assignment statement to declare a variable implicitly instead of using the "var." For instance, the code below shows how you can use the ":=" short assignment statement.

```
func main () {
var i , j int=1, 2
k:=3
fmt.Println (i, j, k, sam, janet, ismarried)
}
```

What about the basic data types of Golang?

Well, Golang has the following basic data types that you should know:

- The bool data type. It represents a set of Boolean truth values. It can either be true or false.
- The string data type. A Golang string is implemented as a set of bytes that store a sequence of characters, using a given character encoding scheme.
- The int data type. A Golang int data type stores whole numbers of varying sizes. They include the int8 for 8 bits, the int16 for 16 bits, the int32 for 32 bit and the int64 for 64 bits.
  The unit data types. It represents the unsigned integer types of varying lengths. For instance, the uint8 for 8 bit, the uint16 for 16 bits, the uint32 for 32 bits and the uint64 for 64 bits.
- The byte. It stores integer values for 8 bits. It's the same as the uint8.
- The float32 and the float64. They are used for storing floating point numbers for 32 bits and 64 bits respectively.
- The complex64 and the complex128 data types. It's used to represent complex numbers for 64 bits and 128 bits respectively.

Well, we've come to the end of variables. Keep on practicing with more variables.

DATA TYPES AND TYPE CASTING

In this chapter, we'll be looking at the different data types that are used in Golang and how to cast between different kinds. Before we look at different ways of type casting, let's discuss some of the most commonly used data types in Golang.

Data types in Golang

Here's a list of data types that you'll use in Golang:

- The bool data type. It's used to store a set of Boolean truth values. It can either be true or false.
- The string data type. A Golang string is implemented as a set of bytes that store a sequence of characters, using a given character encoding scheme.
- The int data type. A Golang int data type stores whole numbers (integers) of varying sizes. They include the int8 for 8 bits, the int16 for 16 bits, the int32 for 32 bit and the int64 for 64 bits.
- The uint data types. It represents the unsigned whole numbers (integer types of varying lengths. For instance, the uint8 for 8 bit, the uint16 for 16 bits, the uint32 for 32 bits and the uint64 for 64 bits.
- The byte. It stores integer values for 8 bits. It's the same as the uint8.
- The float32 and the float64. They are used for storing floating point numbers for 32 bits and 64 bits respectively.
- The complex64 and the complex128 data types. It's used to represent complex numbers for 64 bits and 128 bits respectively.

What about type casting?

Well, type casting allows programmers to change an entity from one data type into another. You may ask, "Why do I need type casting in Golang?" Well, if you need to take advantage of certain characteristics of data type hierarchies, then you have to change entities from one data type into another.

For instance, if you have a byte—or uint8—and you would like this to be converted into larger representation for arithmetic calculations, then you should cast such data types. So, are you ready to begin type casting in Golang?

Let's dive in.

In Golang, we use the expression T (v) to convert the value v to the data type T. The example below illustrates how you can convert from one data type to another for numeric data:

```
1  var amount int = 80
2  var amount2 float64 = float64 (amount)
3  var amount3 uint = uint (f)
```

This can be summarized as follows:

```
amount: = 80
amount2:= float64 (amount)
amount3:= uint (f)
```

Here's is what you should note about type casting in Golang: if you use the Go assignment between the items of different data types, you must have an explicit conversion. That's why it's important to it's vital that you differentiate between using the ":=" or the "var= expression" when type casting in Golang.

For instance, when to declare a variable without specifying as an explicit data type - by using the ":= syntax" or the "var = expression syntax" - the variable's data type will be inferred from the value that appears on the right hand side. If the right hand side of your declaration is typed, the new variable will be of the same data type.

Here's an example:

```
var amount int
amount1:= amount // In this case, amount is of int data type
```

Now what happens when the right hand side is un-typed? Well, when the right hand side has an un-typed numeric constant, then new variable - in this case amount1 - may assume an int, float64, or even complex128 depending on the precision of your constant.

The example below illustrates this:

```
i: = 42                    // This is an int data type depending on the accuracy.
pi: = 3.142      // This is a float64 data type depending on the precision.
b := 0.867 + 0.5i    // This is a complex128 depending on the precision.
```

LOCAL ENVIRONMENT WITH GOLANG LITEIDE

Let's face it - a good text editor can provide a local environment that's suitable for writing your Golang programs. However, the time you take writing such codes is lengthy, and in some cases, quite laborious and tedious - especially when testing and debugging your programs. So, what's the solution to this problem?

Well, such a problem - that involves spending so much time testing and debugging Golang apps - can be eliminated by using a LiteIDE. If you'd like to write large software application using Golang - and any other programming languages - then you must install a LiteIDE. You may be wondering, "What is a LiteIDE?"

Well, a LiteIDE is a local integrated development environment is an easy-to-use, open-source and cross-platform IDE that's precisely meant for Golang applications. This integrated development environment can help you shorten the process of testing and debugging your Golang codes.

Features of Golang's LiteIDE

Here are some of the characteristics of this integrated development environment:

- It has configurable build commands that can help you make your Golang code.
- It has a Kate formatting system that's used for auto–completion and theming of your codes.
- It provides a configurable auto–completion system that can integrate with WordApi.
- It provides Golang support for package browsers, class view and outline and document browsers.
- It provides the markdown support for live preview and synchronized scrolls, custom CSS that are integrated with themes, export Markdown as either HTML or PDF.
- It supports different platforms. You can install it on Windows OS (both 32 and 64 bits), Linux OS (both 32 and 64 bits) and Mac OS (both 32 and 64 bits).

So, how can you set up your local environment with Golang's LiteIDE? Keep reading.

For each platform - Windows OS, Mac OS or Linux OS - you will have to do three things. First, you have to download and install it on your computer. Next, you must install your Golang compiler. Third and finally, you will set up your environment ready to begin coding.

In "How to Set up a Local Environment with Golang LiteIDE", we review the procedures of setting up a local environment for Golang. So, let's dive in.

#1: Configuring the LiteIDE in Mac OS

Here are the procedures for installing LiteIDE in Mac OS environments:

- Download and install the Homebrew and Homebrew Cask on your computer. You can download the Homebrew and Homebrew Cask from this link.
- Open your Homebrew and install the LiteIDE by typing “brew cask install liteide” commands at the command prompt.
- Install the Golang compiler by using Homebrew. Type “brew install go” in the command prompt.
- Set up the environment variables and install the godoc and vet to begin using your LiteIDE. Type the command “install code.google.com/p/go.tools/cmd/godoc” and “install code.google.com/p/go.tools/cmd/vet” to install godoc and vet respectively.

#2: Configuring the LiteIDE in Windows OS

Here are the steps for downloading and installing Golang’s LiteIDE on your computer:

- Download the “.msi” file for Windows download from this link.
- Open the file and follow the quick instructions to install the LiteIDE on your computer.
- Set up the environment variables and install the godoc and vet to begin using your LiteIDE.

#3: Configuring the LiteIDE in Linux OS

Follow the steps below to install LiteIDE on your Linux OS computer:

- Download the archive file and extract it into /usr/local.
- Open the terminal and type the following commands at the command prompt: “$ git clone https://github.com/visualfc/liteide.git”.
- If you’re using Ubuntu, type “,$ sudo apt–get update” at the command prompt. Wait for the update to complete and type “sudo apt–get install qt4–dev–tools libqt4–dev libqt4–core libqt4–gui g++” to install your LiteIDE.
- If you’re using Fedora, type “yum update” at the terminal and wait for the update process to complete. Type “sudo yum install qt4–dev–tools libqt4–dev libqt4–core libqt4–gui g++”.
- Launch the LiteIDE by typing “QTDIR=/usr ./build_linux.sh” at the command prompt.

STRUCT

In this chapter, we discuss how to use structs in Golang. After the end of the lesson, you'll have learnt how to implement structs in your Golang programs.

Structs are typed collections that are used to stored fields. They are useful when you want to group your data to form the records. Why are structs important to a Golang programmer?

Now, you've learnt how to write Golang programs using various built-in data types. While using data types is crucial, there comes a time when you need to use structs instead of built-in data types, especially when you have so many data types that can be grouped together. Using so many data types and variables is tedious.

Consider the Golang program below:

```
1  package main
2  import ("fmt"; "math")
3  func coordinates(a, b, c, d float32) float32{
4  m := b - a
5  n := d-c
6  return math.Sqrt (m*m - n*n)
7  }
```

It's difficult to keep track of all coordinates and see how the program is running especially if you have so many variables to work with. So, how can structs simplify such a program?

First, structs are specified using type—that groups related fields into form records—and the "struct" keywords. Here's the syntax of using structs in Golang:

```
1  type name_of_the_struct  struct { }
```

In this case, the name_of_the_struct is an identifier—that must adhere to naming convention rules. The example below illustrates how you can create a struct in Golang:

```
1  type myshape struct { }
```

In the above case, the name of the struct is "myshape". What's placed in "{ }" are the implementations of the struct.

Now, just like classes in object oriented programming, structs can contain some other data—that means you've to declare the various data types that you wish to have in your struct. That means that you'll be representing contents of any real world object into the computer's memory. *Here's an example:*

```
type shape1 struct {
a int32
b int32
c string
}
type shape2 struct {
d, e int
g float32
}
```

Next up, how can you use structs in your Golang code? Keep reading.

Initialization of Struct

You can create an instance of the "shape1" in so many ways. First, you can use the syntax below to create an instance of "shape1":

```
var h shape1
```

The above initialization creates a local shape1 variable that will be set to zero, by default. This means the values for variables a and b will all be set to 0 since they are integers. However, the variable c will be set to "nil" since it's a string.

Secondly, you can initialize an instance of "shape1" using the syntax below:

```
i := new(shape1)
```

The above initialization allocates memory for all the fields—a, b and c—and sets each of fields to 0 value and returns a pointer.

Thirdly, if you want to set different values for your fields instead of 0, you can use the syntax below:

```
i := shape1{a: 5, b: 16, c: "This an example of structs in Golang"}
```

Finally, you can use the syntax below to initialize your structs:

```
i := shape1{5, 16, "This an example of structs in Golang"}
```

What about displaying fields in Golang?

Well, to display the field in Golang, we use the "." operator.

Here's an example of how to display fields while using structs in Golang:

```
fmt.Println (i.a, i.b, i.c)
```

Finally, let's have an example that illustrates Golang's struct:

```
package main
import "fmt"
type Rect struct {
l, w int
}
func main() {
    r := Rect{}
    fmt.Println("The default values of the rectangle are: ", r)
}
```

Now you know how to use structs in Golang programs. It's important to adhere to the syntactic rules when using structs to have efficient Golang programs.

FUNCTIONS

In this chapter, we discuss Golang Functions. By the end of the class, you'll be expected to master how to use Golang functions. But first, let's start by defining what functions are and their importance in programming.

A function—also known as procedures or subroutines in some programming languages—is a section of code that's independent of the main code that helps to map zero or more input arguments. Ideally, the main purpose of using a function in a code is to assist group program statements to perform specific tasks.

Why do we need functions in Golang programs?

Well, functions are very useful constructs in helping you to organize your code. If you don't use functions, you may end up writing so many lines of codes that may not scale up well. One common function that we've been using so far in our earlier discussions is:

```
1 | func main () {}
```

This is the core function of every Golang program that you'll be creating. It's the function that initializes execution of your code. It invokes other functions—both user defined and in-built functions—in your Golang code.

For you to learn how to create your functions, you need to first, understand how to declare functions.

All Golang functions must start with the keyword "func" which is followed by a function's name. What follows after the function name is a list of arguments—parameters or inputs that the function will use. Here's a syntax:

```
1 | func function_name (list of arguments)
```

You'll define the arguments by indicating their name type. Here's an example:

```
1 | func add(a float32, b float32)
```

The above function has two arguments namely: "an" and "b" which are floating point numbers. Collectively all the arguments and their return type are called function's signature. In a nutshell, all Golang functions must begin with the keyword "func" followed by the function name and the function signature enclosed in brackets.

When two or more logical function arguments share the data type, you can leave the data type from all but the last argument. Here's an example:

```
1 | a float32, b float32
```

Can be shortened to:

```
a, b float32
```

How to implement functions in Golang

We implement functions by declaring any other variable(s) that may not have been declared in the function signature followed by program statements that will depend on what you want your function to do.

Here's an example:

```
func add(a float32, b float32) float32 {
return a + b
}
```

The return statement causes the execution of a function to leave the current procedure or subroutine and resume in the main function. Sometime, functions can also be valued. That means that they can be passed around just like all the other values—that may be used as function parameters.

The return statement that has no arguments returns the named return values in Golang. This is referred to as "naked" return. The naked return statements should only be used with small functions. Otherwise they can make readability of longer functions to be difficult.

Functions in Golang can return a different number of results.

Here's an example:

```
func example1 (a, b string) (string, string) {
return a, b
}
```

Next up, we explore how to call functions in your main program.

Functions are called by their names in the main function "func main ( )." For instance, here's an example that shows how the function above "example1" can be called into the main function.

```
package main
import "fmt"
func example1 (a, b string) (string, string) {
return a, b
}

func main() {
c, d:= example1("Hallo my dear!!", "I have just returned from Iowa")
fmt.Println(c, d)
}
```

By following the rules of using functions that we've just learnt in this lesson—you'll be on your way to becoming a professional in Golang!

DEFER FUNCTION

In this class, we follow up our discussions on functions by looking at the special statement “defer” in Golang.

Let’s start by discussing what the defer statement does in Golang programs.

The defer statement is used to schedule a function call so that it can be run after the function completes. Take a look at the example below:

```
1  package main
2  import "fmt"
3  func example1 () {
4  fmt.Println ("Welcome to Golang")
5  }
6  func example2 () {
7  fmt.Println ("This is an example of using Defer in Golang")
8  }
9  func main() {
10 defer example2()
11 example1 ()
12 }
```

What do you think will be the output of this Golang code?

This Golang code will print “Welcome to Golang” followed by “This is an example of using Defer in Golang.” The defer statement has been used to move the function call to the second so that it executes at the end of the function. Here’s how it’s executed:

```
1  func main () {
2  example1 ()
3  example2 ()
4  }
```

You might be thinking, “When should I use the defer statement with Golang functions?”

Well, you’ll be forced to use defer statement when you need to free the resources in your computer. For instance, if you open a file, and you wish to close it later, and then you can defer. Here’s an example:

```
1  f, _:= os.Open (filename)
2  defer f.Close ()
```

The arguments that are passed to the deferred function must be evaluated first when to defer function executes and not when the function call runs. This means that you shouldn’t worry much about wheth-

er the variables will be changing when your function executes—because there will only be a single deferred function call. Here's an example:

```
for number := 0; i < 40; i++ {
defer fmt.Printf ("The number that you want is %d \n", number)
}
```

You should note that deferred functions are executed based on last in first out (LIFO) order. The advantages of using defer statement with Golang functions are threefold. First, it will help you to keep your close calls near your open call so that your code becomes easier to understand.

Second, if your function has many return statements—which in most cases makes your code appear "unclean"—then you can determine what to close. Third and finally, the deferred functions are easily run even during run-time panic times.

The Golang's defer function is used in conjunction with panic and recover functions to handle run-time panic and recovery options. When used with both panic and recover functions, defer function can help you to free resources in your computer and make your program effective and efficient.

Next up, we look at both panic and recovery functions and how we can use them to have effective and efficient programs.

Golang's panic and recover functions

The Golang's panic function is used to define programmer error—such as accessing an index of an array that's outside of bounds or forgetting to initialize the map—that 's hard to recover from. Here's an example of Golang code that demonstrates how the panic function works in Golang:

```
package main
import "fmt"
func main() {
panic ("This is an error that's difficult to recover from")
a := recover()
fmt.Println (a)
}
```

In the above example, the call to recover may never occur because it immediately stops the execution of the function call. However, when used with deferring, it changes the execution of the function call. Here's an example of such a code:

```
package main
import "fmt"
func main() {
defer func () {
a := recover( )
fmt.Println ("The value of a is:", a)
}( )
panic ("This is an error that's difficult to recover from ")
}
```

Now you can start using defer function to help you manage errors in Golang.

POINTERS

In this class, we discuss how to use pointers in Golang. By the end of the class, you'll have learnt how to define and use pointers in Golang.

Let's start the discussion by first defining pointers.

A pointer is an object whose value is used to refer to another value that's stored elsewhere in a computer's memory using its address. When you define a pointer in Golang, it will reference to a particular location in your memory and get the value that's stored at that memory location through a process known as "dereferencing the pointer."

Let's take a look at the Golang code below that doesn't use pointers:

```
1  package main
2  import "fmt"
3  func total (number int32) {
4  number = number + 1000
5  }
6  func main() {
7  number := 2000
8  total (number)
9  fmt.Println ("The total is:", number)
10 }
```

What do you think will be the output of these Golang code? Is it 2000 or 3000?

Well, the output of this program will be:

```
1 The total is: 2000
```

I bet that you'll have expected the code to print 3000. Why is this the case? By default, all arguments are copied by value when passing in an argument. Therefore, the copy of the number variable is used to by the total function. That's why the program prints 2000 instead of 3000.

How can you fix this problem? Good question.
You can either use the call by reference or use pointers. Here's is how you can use the call by reference:

```
1  package main
2  import "fmt"
3  func total (number *int32) {
4  *number = * number + 1000
5  }
6  func main() {
7  number: = 2000
8  total (&number)
9  fmt.Println ("number:", val)
10 }
```

Notice the use of "*" and the ampersand character"&" in the above code. The total function has been made to accept an integer—which is a number—that references some memory. This pointer points to a memory location and allows you to change its value. When you call the total function in the main function, all you need is to send the reference—which is an address of the number variable.

The use of pointers in Golang helps to pass references to values and records within the Golang program. Therefore, the output of such a program becomes:

```
1 | The total is: 3000
```

What you should know about Golang's pointers

Here's what you need to know when using Golang's pointers:

- The Golang's pointer is used to store the memory address of a given variable. The type *P means a pointer to the P value. By default, it will be set to zero value if it's numeric or "nil" if it's a string.
- We declare pointers using the syntax below:

```
1 | var name_of_the_pointer *data type
```

  In the above declaration, name_of_the_pointer is the name of the pointer. This is followed by an asterisk (*) and the data type—which can be numeric or non–numeric.

- Pointers can be initialized using the syntax below: name_of_the_pointer:=some value. Here's an example:

```
1 | p=: 3830
```

- The "&" operator creates a pointer and pins it to its operand. For instance, the code "p= &M"denotes the pointer's underlying value. This is referred to as dereferencing.
- The & M syntax allocates the memory address of P, which is a pointer to M.
- Pointers can also be created using the "new" function. The new function takes a data type as an argument and allocates it enough memory so that it fits a value of that data type—and returns a pointer to the same data type. Here's an example:

```
1 | func example3 (p *int32) {
2 | *p = 1000
3 | }
4 | func main () {
5 | p: = new(int)
6 |   example3 (p)
7 |   fmt.Println (*p)
8 | }
```

Now you know how to use pointers in Golang. Use the knowledge you've learnt in this chapter to create programs that use pointers.

PROGRAMMING OPERATORS

Operators are symbols that tell the compiler to perform specific operations that may be mathematical, relational or logical. The Golang programming language has a rich in in-built operators that can be grouped into: Arithmetic Operators, Relational Operators, Logical Operators, Bitwise Operators, Assignment Operators and the Misc Operators.

So, let's dive in and find out about these operators:

#1: Arithmetic Operators

Here are examples of Golang's arithmetic operators and their uses:

- "+" It adds two operands. For instance if A=20 and B=20 then A+ gives 40 as the answer.
- "–" It subtracts the second operand from the first operand. For example, if A=20 and B=20 then A – B will yield 0.
- "*" It's used for multiplication of operands. For example, A*B produces 400.
- "/" It's used for division. For instance, A/B produces 1.
- "%" It's a modulus operator—it displays the remainder of a number after an integer division. For instance, B % A yields 0 as a remainder.
- "++" It's an increment operator—it increases integer value by one. For instance, A++ yields 21.
- "—" It's a decrements operator—it decreases an integer value by one.

#2: Relational Operators

Here are examples of Golang's relational operators and their uses:

- "==" It checks whether the values of two operands are equal or not. If they are equal, the condition becomes true. For instance A==B is true if A=20 and B=20.
- "!=" It checks whether the values of two operands are equal or not. For instance, A!=B is false.
- ">" It checks whether the value of the left operand is greater than what is on the right operand.
- "<" It checks whether the value of the left operand is less than what is on the right operand.
- ">=" It checks whether the value of the left operand is greater than or equal to what is on the right operand.
- "<=" It checks whether the value of the left operand is less than or equal to what is on the right operand.

#3: Logical Operators

Here are examples of Golang's logical operators and their uses:

- "&&" the logical AND operator. If both the operands are not zero, then condition is true.
- "||" the logical OR operator. If any of two operands is not zero, then condition is true.

- "!" the logical NOT Operator. It reverses the logical state of its initial operand.

#4: Bitwise Operators

The bitwise operators perform bit-by-bit operations in programming. Here are examples of Golang's bitwise operators and their uses:

- "&" the binary AND operator. It copies a bit to the result if it is existing in both the operands.
- "|" the binary OR operator. It copies a bit if it is existing in either operand.
- "^" the binary XOR operator. It copies the bit if it is set in only one operand and not both operators.
- "<<" the binary left shift operator. It moves the left side by a number of bits that are specified by the right operand.
- ">>" the binary right shift operator. It moves the right side by a number of bits that are specified by the right operand.

#5: Assignment Operators

Here are examples of Golang's assignment operators and their uses:

- "=" It assigns values from right side operands to the left side operand.
- "+=" It adds the right operand to left operand and assign the answer to left operand.
- "–=" It subtracts the right operand to left operand and assign the answer to left operand.
- "*=" It multiplies the right operand to left operand and assign the answer to left operand.
- "/=" It divides the right operand to left operand and assign the answer to left operand.

#6: Misc Operators

Here are examples of Golang's misc operators and their uses:

- "&" It returns the address of a given variable.
- "*" It's a pointer to a particular variable.

Now you can begin using Golang's operators.

CONDITIONALS

In this class, we'll be covering how you can use conditionals in Golang. So, let's dive in and find out how to use conditionals in Golang.

You might be thinking, "What are conditionals?"

Well, conditionals are expressions that result to either true or false when evaluated. The main purpose of using conditionals is to help in controlling the program flow during execution. Two conditionals are employed in Golang. These are:

- If else statement
- Switch statement

Let's dive in and find out how you can use these conditionals.

#1: If else statement

In Golang, a simple if statement has the following form:

```
if Boolean expression
{
Statements
}
```

Here's an example of a code that tests if the average score is greater than or equal to 80 and displays grade "A".

```
if average>=80
{
grade="A."
}
```

An if statement can only be executed when the Boolean expression is true. Otherwise, the statement is skipped. In some cases, if statement can be followed by an else statement that is optional. When you have an optional if statement, the optional statement executes only when the Boolean expression is false. Here's the syntax:

```
if Boolean expression
{
Statement(s) /* they can only execute when the Boolean expressions are true*/
   }
else
{
Statement(s) /* they can only execute when the Boolean expressions are false*/
}
```

Note that the Boolean expression is not enclosed in brackets. Here's an example of if statement that has optional else statement:

```
if average>=50
{
remark=" Passed."
}
else
{
remark=" Failed."
}
```

An if else statement can also be nested. For instance, the Golang code below shows how the if else statement can nested:

```
if average>=70
{
grade="A."
}
else if
average>=60
{
grade=" B."
}
else if
average>=50
{
grade=" C."
}
else if
average>=40
{
grade=" D."
}
else
grade=" F."
}
```

#2: The Switch Statement

The switch statement enables a variable to be tested for equivalence against a set of values where each value—commonly referred to as the case—is the variable that's checked for each switch case. There are two types of defining the switch statement. These are:

- The expression switch.
- The type switch.

While the expression switch statement has expressions that are compared against the value for each of the switch expressions, the type switch contains the types that are compared against the type of special annotated switch expressions.

The syntax for the expression switch statement is as follows:

```
switch Boolean expression {
    case Boolean expression:
       Golang statement(s)
    case Boolean expression
       Golang statement(s);
    default :
      Golang  statement(s);
}
```

Note that the Boolean expression can also be substituted with integral value. Here's an example of a Golang code that uses expression switch:

```
switch marks {
      case 90: grade = "A"
      case 80: grade = "B"
      case 50, 60, 70: grade = "C"
      Default: grade = "D"
The syntax for the type switch is as follows:
switch y. (type) {
    case type:
       Golang statement(s);
    case type:
       Golang statement(s);
        default:
      Golang statement(s);
}
```

Here's what you should note about the syntax of the type switch statement:

- The expression that's used in a type switch statement should have variable of interface {} type.
- The type of your case should be of the same data type as that of variables in the kind switch—which must be a valid data type.
- You can have an optional default statement in a type switch statement.

Here's an example of Golang code that demonstrates use of type switch statement:

```
var y interface{}
   switch m := y.(type) {
      case int32:
         fmt.Printf ("y is integer that has 32 bits")
      case float64:
         fmt.Printf ("y is floating point number with 64 bits")
      case bool, string:
         fmt.Printf ("y is either a Boolean or string")
      default:
         fmt.Printf("I don't know the type")
```

ARRAY AND SLICES

In this class, we'll discuss how to use arrays and slices in Golang. By the end of the tutorial, you'll be in a position to declare and use arrays and slices in Golang.

Let's start by looking at the differences between arrays and slices in Golang.

An *array* can be defined as a data structure that's used to store a fixed-size sequential collection of data elements that are of the same data type. Think of arrays as collections of variables that are of the same data type. For instance, if you have variables num1, num2, num3, num4 and num5 that are all integers (int32), we can declare these variables as one array instead of declaring five different variables.

On the other hand, *slices* are can be defined as abstractions that can take place over the Golang's arrays. Golang's arrays help you to define a variable that can hold multiple data items of the same type. However, it doesn't provide you with in-built methods that can promote interaction with the array. This is a limitation that slices provide—they are many utility methods that slices provide to help you interact with arrays.

So, how can you define and use arrays and slices in Golang? Let's dive in and learn tips for effective programming.

Golang's Arrays

Here's how you can declare arrays in Golang:

```
1 | var name_of_array [SIZE] data type
```

The size of the array should be an integer constant that's greater than 0. Here's an example of how you can declare an array named balance that has 10 data elements of type float64:

```
1 | var balance [10] float64
```

Here's how you can initialize an array by using a single statement:

```
1 | var balance = [10]float64{1000.0, 300.73, 33003.40, 7730.0, 5004.0,57603.0,57602.5,56570.0,6806.0,4049.7}
```

Array elements are accessed by indexing their names—where the index of the data element is placed within square brackets after the array name. Here's an example:

```
1 | float64 mybalance= balance [6]
```

Here's an example of Golang code that uses arrays:

```
var balance [5] int32
    var a, b, c int32

   for a = 0; a < 5; i++ {
      balance [1] = i + 100   }
      fmt.Printf ("balance[a])
   }
```

What about Go Slices?

We declare the Golang slices as an array that's specified without specifying its size. Alternatively, you can use the "make" function to create one. Here's an example:

```
var balance [] float64
Or
balance = make ([] float64, 5, 5)
```

If you declare your slice with no inputs, then it will be initialized as "nil". This means that its length and capacity will be set to zero.

There two commonly used functions with Golang's slices are "len ( )" and "cap ()" functions. The len () function returns the number of elements in a slice while the cap () function returns the capacity of a slice. Here's an example:

```
func main ()
{
var balance = make ([] float64, 5, 5)
myslice (balance)
}
func myslice (a [] int32){
   fmt.Printf ("The length and the capacity of slice are: ", len (a), cap (a))
}
```

The "append ( )" and "copy () functions allows you to increase the capacity of your slice. We use the copy () function to copy the contents of a source slice into destination slice while append () function adds data elements at the end of a Golang slice. Here's an example:

```
package main
import "fmt"
func main () {
var balance [] int32
myslice (balance)
balance = append (balance, 0)
myslice (balance
balance = append (balance, 1)
myslice (balance)
balance = append (balance, 2, 3, 4)
myslice (balance)
balance1:= make ([] int32, len (balance), (cap (balance))*2)
copy (balance1, balance)
myslice(balance1)
}
func myslice(a []int32){
fmt.Printf ("The length and the capacity of slice are: ", len (a), cap (a))
}
```

There you have it—keep on practicing to perfect the use of arrays and slices in Golang.

MAPS

In our previous chapters, we've discussed Golang's data types, arrays and slices. In this tutorial, we present a one more in-built type—the maps. By the end of the class, you should have learnt how to use maps in your Golang codes.

Are you ready?

A map is an unordered combination of key-value pairs that helps to look-up values by using their associated keys. You should recall that a hash table is one of the most important data structure that programmers use to look-up values using associated keys. The merits of using a hash table in a program are that it helps in providing fast lookups, additions, and deletes.

Well, Golang provides an in-built map data type that contributes to implementing a hash table. So, let's dive in and find out how you can implement a hash table using Golang.

How to declare and initialize Golang maps

Golang maps are declared using the syntax below:

```
1 | var n map[string]int
```

In this declaration, the map data type is specified by the keyword "map" which is followed by the key data type in brackets and finally the value type. Such a declaration can be read as "n is a map of strings to integers." You can figure this as a hash table that can help you provide fast look-ups using the keys (indexes) through referencing.

The map types will act as reference types, just like the pointers or slices. Therefore, the value of n above is set to "nil" since it hasn't been initialized. Nil maps are empty when they are read.

Maps are initialized using the in-built make function. Here's an example that initializes a map n that has been declared above:

```
1 | n = make (map [string] int)
```

The main function of the make function is to allocate and initialize the hash map data structure. It also returns the map value that point to the map.

Here's what you should note about Golang's maps:

- They aren't good when you use them for concurrent use. If you want to read and write to a map at the same time, the accesses must be moderated by synchronization mechanisms such as the sync.RWMutex.
- The map keys that you specify under the declaration can be of any data type that is comparable. For instance, you can use Boolean, numeric or string data types that you want in your maps.

So, how can work with maps in your Golang code? Keep reading.

How to work with Maps in Golang

Just like arrays and slices, maps can be accessed using the brackets. Here's an example:

```
var n map [string]int
n ["key"] = 100
fmt.Println (n ["key"])
```

You can also create maps using the key type of integer. Here's an example:

```
n: = make (map [int] int)
n [1] = 200
fmt.Println (n [1])
```

In the above example, the map has been declared and initialized just like Golang arrays.

Golang Map Functions

Here are examples of Golang map functions:

1. The delete () function

It's used for deleting entries in Golang maps. For you to delete an entry from a map, you need to have the map and the corresponding key that should be deleted. Here's an example that shows how to use the delete function if you want to delete an entry from a map:

```
delete (n,"Yaoundé")
fmt.Println (" The entry for Yaoundé has been deleted")
fmt.Println ("The map has been updated after the delete process")
```

2. The len () function

It's used to specify the number of items on a map. Here's an example of Golang code that uses the len () function:

```
n: = make (map [int] int)
n [1] = 200
x := len(n)
fmt.Println (n [1])
fmt.Println ("The length of this map is:", x)
```

You can now begin using maps when writing Golang maps.

LOOPING ARRAYS AND SLICES

We've so far discussed how you can declare and initialize arrays and slices in Golang. In this chapter, we discuss how you can use loops to initialize your arrays and slices.

Why do we need loops when initializing some arrays and slices?

Well, there may be a situation, when you need to initialize your array using a block of several values in some times. In such a scenario, you need to use Golang loops to simplify your code. Here's an example of initializing an array in Golang where:

```
1 | var balance = [10]float64{1000.0, 1000.0, 1000.0, 1000.0, 1000.0, 1000.0, 1000.0, 1000.01000.0, 1000.0}
```

In the above example, an array named balance has been initialized with 10 data elements of float64 type. Such a code can be simplified using loops. A loop statement enables you to execute a program statement or group of program statements multiple times. In the above example, a single loop can be written to initialize the balance ten times since the number of elements is ten.

Structure of Golang Loops

The Golang has for loop that is a repetition control structure that allows programmers to write efficiently loops that execute multiple times while initializing Golang's arrays and slices. Here's the syntax for specifying for loop in Golang:

```
1 | for [condition |  ( init; condition; increment ) | range]
2 | {
3 |    Golang statement(s);
4 | }
```

Here's what you should know about this syntax: If the condition is available, then the loop will execute as long as the condition remains true. If the for clause that is (init; condition; increment) here's how the loop will be executed:

- The init step will be executed first, and only once. You can declare and initialize any of your loop control variables. Program statements are placed in this section as long as the semicolon exists.
- Next, the condition will be evaluated. If the condition is true, then the body of for loop will be executed. On the other hand, if the condition is false, the body of the loop will not run and flow of the program will jump to the next program statement just after the loop.
- After the body of for loop has executed the flow of program will jump back up to the increment statement.
- The condition of your loop will now be evaluated again. If it's true, then the loop will execute, and the process repeats itself until the condition becomes false, after which for loop is terminated.
- If the range is specified, then for loop will execute for each data item that's in the range.

Examples of looping with arrays and slices in Golang

Here's an example of how you can use loops for your arrays in Golang:

```
func main () {
var a [5] float64
a [0] = 100
a [1] = 200
a [2] = 300
a [3] = 350
a [4] = 500
var sum float64 = 0
for count := 0; count < 5; count++ {
sum += a[count]
}
fmt.Println(sum / 5)
}
```

The above Golang program calculates the average of 5 numbers which have been implemented in arrays. Notice how for loop has been used to compute the sum.

Next up, let's have a look at how you can use for loops to implement Golang slices:

```
func main () {
var a =[5] float64 {100, 200, 300, 350, 500}
var sum float64 = 0
for count := 0; count < 5; count++ {
sum += a[count]
}
fmt.Println ("The sum of 5 data elements in the slice is:", sum / 5)
}
```

There you have it—you can now begin using loops to implement Golang's arrays and slices.

LOOPING WITH MAPS

We've so far discussed how you can declare and initialize maps in Golang. In this chapter, we discuss how you can use loops to implement your Golang maps.

We use loops in maps to initialize a block of several values where we want to simplify our Golang code. Have a look at the following sample code where the map has been initialized:

```
n: = make (map [int] int)
n [1] = 200
n [2] = 300
n [3] = 350
n [4] = 450
n [5] = 600
```

If we were to print the individual elements of such a map, we can either print each of elements or use for loop. Here's how you can use the print fmt function to print the individual elements:

```
func main () {
n: = make (map [int] int)
n [1] = 200
n [2] = 300
n [3] = 350
n [4] = 450
n [5] = 600
fmt.Println ("the First element is:", n[1])
fmt.Println ("Second element is:", n[2])
fmt.Println ("Third element is:", n[3])
fmt.Println ("the First element is:", n[4])
fmt.Println ("Fifth element is:", n[5])
}
```

Now, such a code is too bulky. You need for the loop to help you simplify it.

Here's the syntax of for loop:

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

What's the significance of this syntax? Well, here's what you should know about for loop syntax that you want to use with Golang maps:

If the condition is available, then the loop will execute as long as the condition remains true. If the for clause that is (init; condition; increment) here's how the loop will be executed:

- The init step will be executed first, and only once. You can declare and initialize any of your loop control variables. Program statements are placed in this section as long as the semicolon exists.
- Next, the condition will be evaluated. If the condition is true, then the body of for loop will be executed. On the other hand, if the condition is false, the body of the loop will not run and flow of the program will jump to the next program statement just after the loop.
- After the body of for loop has executed the flow of program will jump back up to the increment statement.
- The condition of your loop will now be evaluated again. If it's true, then the loop will execute, and the process repeats itself until the condition becomes false, after which for loop is terminated.
- If the range is specified, then for loop will execute for each data item that's in the range. Other wise, if it's not specified then your loop will execute in the normal way.

So, how can you modify the above code to use for loops? Here's how your code should look like if you wish to use for loops in Golang maps:

Here's an example of how you can use loops for your arrays in Golang:

```
func main () {
n: = make (map [int] int)
n [1] = 200
n [2] = 300
n [3] = 350
n [4] = 450
n [5] = 600
for count := 0; count < 5; count++ {
fmt.Println ("The element of the map is:", n [count])
}
```

Here's another example that illustrates how you can use loops in your maps:

```
import "sort"
var a map[int]string
var index []int
for i := range a {
index = append(index, i)
}
sort.ints (index)
for _, i: = range index {
fmt.Println ("The key that you wish to find is:", i, "and the value that you are looking for is :", a[i])
}
```

There you have it—you can now begin using loops to implement Golang's maps. Keep on practicing to be perfect.

INTERFACE

Welcome to Golang certification class number 19—How to use interfaces in Golang programming. Today, we'll be discussing how you can define and use interfaces in Golang programming.

Let's begin by discussing why interfaces are necessary in programming.

An *interface* is can be defined as a set of methods that helps programmers to model real world objects in a simplified manner. The value of type can be used to hold any data item that implements any of the named methods and collections. We use interfaces in daily life situations.

For instance, if you bank your money in a fixed deposit account, you expect it to earn some interest. However, how this money makes you interest is of little concern to you. All you want at the end of interest period is your amount which includes the principal plus interest. The same concept can be applied in programming.

If you define an interface in Golang, the interface helps you to hide the implementation details of your objects through abstraction while only providing high level functions. For instance, high level functions could be the interest that your money accrues after some period. Interfaces also help to make an extensible and robust application based on principles of object orientation.

So, are you ready to learn how you can implement interfaces in Golang? In "How to use Interfaces in Golang" we learn how you can define and use interfaces in Golang.

Defining interfaces in Golang

Just like structs, all interfaces are defined using the key word "type" which must be followed by the interface name and the keyword "interface." However, instead of representing the fields—as case of structs—you'' have to the "method set" in interfaces.

The method set is simply a list of all your methods of a particular type that will help you to implement your interface. An interface can be viewed through the prism of object oriented programming as a class with class members that helps you to interact with data items.

Here's an example of how an interface named "myshape" can be defined in Golang:

```
1  type myshape interface {
2    area() float32
3  }
```

In the above example, myshape is the name of the interface that we've created. This interface has one method called, "area ()" which is of type float32.

Once you've defined the interface methods, you should go ahead and provide the implementation of

these methods. For instance, the method area () can be implemented as follows:

```
type Rect struct {
length, width int
}
func (a Rect) area() float32 {
return a.length * a.width
}
```

The first section of the code creates a "Rect" struct that should implement the area () method of the interface we defined earlier. The second code is a function that implements how the area of a given rectangle should be computed.

In a nutshell, here's what we have defined:

- The name of the interface is myshape.
- It has one method called area ().
- There's one shape implementation that has been defined called rectangle.
- To calculate the area of the rectangle we use the formula: return a.length * a.width.

So, how can you implement such an interface in Golang?

Working with interfaces in Golang

To apply interfaces in Golang, we use the same concept we learnt in structs. For instance, we'll use the "." operator to access fields in a particular object. Here's an example of how the above interface definition can be implemented in Golang:

```
package main
import "fmt"
type myshape interface {
  area() float32
}
type Rect struct {
length, width int32
}
func (a Rect) area() float32 {
return a.length * a.width
}
func main () {
r := Rect{length:10.4, width:5.3}
fmt.Println ("Rectangle r's area is: ", r.area())
b: = myshape(r)
fmt.Println ("The Area of the myshape r is: ", b.area())
}
```

There you have it—start defining and implementing interfaces in Golang. Remember, we're here to help you. Get in touch if you have queries that you'd like us to respond to.

Thank you for reading!

We invite you to share your thoughts and reactions

STONE RIVER eLEARNING