Golang Cheat Sheet - Hackr.io

Before hopping directly into our Go cheat sheet, let's cover Go set-up and installation.

How to Set Up and Install Go

First, you need to download the Go Archive. The following table highlights the available archives for each system:

OS	Archive Name
Windows	go1.14.windows-amd64.msi
Linux	go1.14.linux-amd64.tar.gz
Mac	go1.14.darwin-amd64-osx10.8.pkg
FreeBSD	go1.14.freebsd-amd64.tar.gz

After downloading the appropriate archive, extract it to the /usr/local. Later, create a folder named "Go" in that folder. You can run the following command from your command prompt (cmd).

```
mkdir /usr/local/Go
tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
brew install go
```

Now, you must add the path variable /usr/local/go/bin to the environment variable. The following table highlights the path variable for various operating systems:

OS	Output
Linux	export PATH = $PATH:/usr/local/go/bin
Mac	export PATH = $PATH:/usr/local/go/bin
FreeBSD	export PATH = $PATH:/usr/local/go/bin

Execute the following command in cmd to add the path variables:

```
//  add to ~/.bash_profile
 export GOPATH="$HOME/Go"
 export PATH="/usr/local/bin:$PATH:$GOPATH/bin"
```

Create a Sample File in Windows

Here, we can create a test.go file in C:\>Go_WorkSpace.

```
package main

import "fmt"

func main() {
   fmt.Println("Hello")
}
```

To execute the test, run the following command from cmd:

```
C:\Go_WorkSpace>go run test.go
```

You will get the output:

```
Hello
```

Basic Go Syntax

Now let’s get started with some basic syntax for Golang.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World")
}
```

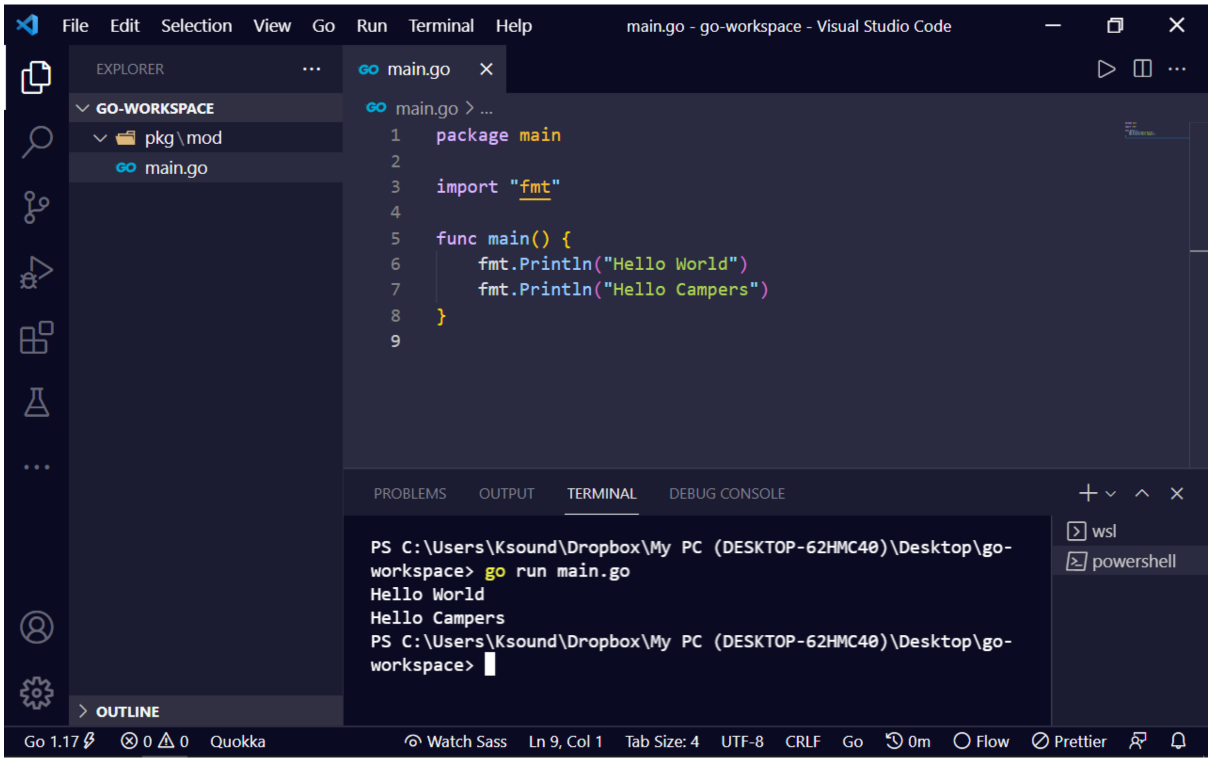
Where:

Package is a collection of the files and code. Make sure to add the “package main” at the top of your every Go program.

Fmt is a package available in the Go language’s standard library. It helps you format strings and print messages to the command line. One of the methods included is the “println” that prints the line. As per the above example, we have used it to print the “Hello World” on cmd.

“Main” function holds the code that will perform the task for you.

To run this code, type “go run main.go,” and hit enter. You can give any name to your Go program.



```go
package main

import "fmt"

func main() {
	fmt.Println("Hello World")
	fmt.Println("Hello Campers")
}
```

```
PS C:\Users\Ksound\Dropbox\My PC (DESKTOP-62HMC40)\Desktop\go-
workspace> go run main.go
Hello World
Hello Campers
PS C:\Users\Ksound\Dropbox\My PC (DESKTOP-62HMC40)\Desktop\go-
workspace>
```

File Edit Selection View Go Run Terminal Help
main.go - go-workspace - Visual Studio Code
EXPLORER
GO-WORKSPACE
pkg\mod
main.go
OUTLINE
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
wsl
powershell
Go 1.17 ⊗ 0 ⚠ 0 Quokka Watch Sass Ln 9, Col 1 Tab Size: 4 UTF-8 CRLF Go 0m Flow Prettier

Operators

Every programming language comes with operators, which allows you to perform arithmetic, logical, bitwise, comparison, and other functions. Let's walk through Go operators, along with their descriptions and examples.

Arithmetic Operators

Arithmetic operators perform mathematical operations on operands.

Operator	Description	Example
+	Adds two operands	A + B gives 30
-	Subtracts second operand from the first	A - B gives -10
*	Multiplies both operands	A * B gives 200
/	Divides the numerator by the denominator.	B / A gives 2

%	Modulus operator; gives the remainder after an integer division.	B % A gives 0
++	Increment operator. increases the integer value by one.	A++ gives 11
--	Decrement operator decreases the integer value by one.	A-- gives 9

Relational Operators

You can use relational operators to compare two values.

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not; if yes, the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not; if the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the left operand value is greater than the value of right operand; if yes, the condition becomes true.	(A > B) is not true.
<	Checks if the left operand value is less than the value of the right operand; if yes, the condition becomes true.	(A < B) is true.
>=	Checks if the left operand value is greater than or equal to the value of the right operand; if yes, the condition becomes true.	(A >= B) is not true.
<=	It checks if the left operand value is less than or equal to right operand value; if yes, the condition becomes true.	(A <= B) is true.

Logical Operators

You can use logical operators to combine two or more conditions.

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both the operands are non-zero, the condition becomes true.	(A && B) is false.

||	Called Logical OR Operator. If any of the two operands is non-zero, the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

These perform bit-by-bit operations. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p | q	p ^ q
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 operand's 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 operand's value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

Assignment operators help you assign values to variables.

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, 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 assigns the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with left operand and assigns 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 assigns 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; returns the actual address of the variable ‘a’.
*	Pointer to a variable.	*a; provides a pointer to a variable.

Operators Precedence

Category	Operator	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

Data Types

Basic Data Types

Types	Description
Boolean types	They are boolean types and consists of the two predefined constants: (a) true (b) false
Numeric types	They are arithmetic types and represent a) integer types or b) floating-point values throughout the program.

String types	A string type represents the set of string values. Its value is a sequence of bytes. Strings are immutable types that, once created, cannot be changed. The predeclared string type is a string.
Derived types	These include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types f) Slice types g) Interface types h) Map types i) Channel Types

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)

Floating-Point 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

Other Data Types

Type	Description
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 uninterpreted bits of a pointer value

Variables

A variable is a name given to the storage area consistently altered during program execution. Golang supports the following types of variables:

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

Declaring a Go Variable

- Using var keyword:

var variable_name type = expression

- Using short variable declaration:

variable_name:= expression

Example

```
...
func main() {
    var age int
    age = 70
    fmt.Printf("Quantity is %d\n", quantity)
}

// You can merge the var dedclaration and assignment to one
```

```
var age int = 70

// Or you can use shorthand variable declaration operator, :=, which
// can infer type
age := 70
```

Declaring Multiple Variables (using :=)

```
func main() {
    // As long as one of the variables is new, `:=` can be used.
    // However, you can't change the type of age. It was declared
(implicitly)
    // as an integer and thus, can only be assigned integers.
    name, age := "Lemmy", 70
    fmt.Printf("%s's age is %d\n", name, age)
}
```

Constants

Constants, or literals, are fixed values that we cannot alter during program execution. Golang supports different types of constants, such as integer constant, floating constant, character constant, or string literal.

There are also enumeration constants Constants are used just like variables except you cannot change their value during program execution.

Declaring Constants Example:

```
package main

import "fmt"

const PI = 3.1412

func main() {
    const SC = "SC"
    fmt.Println("Hello", SC)

    fmt.Println("Happy", PI, "Day")
    const Correct= true
    fmt.Println( Correct)
}
```

Output:

```
Hello SC
Happy 3.14 Day
true
```

Integer Constant Examples

85   /* decimal */
0213   /* octal */
0x4b   /* hexadecimal */
30   /* int */
30u   /* unsigned int */
30l   /* long */
30ul   /* unsigned long */
212   /* Legal */
215u   /* Legal */
0xFeeL   /* Legal */
078   /* Illegal: 8 is not an octal digit */
032UU   /* Illegal: cannot repeat a suffix */

Floating Type Constant Examples

3.14159   /* Legal */
314159E-5L   /* Legal */
510E   /* Illegal: incomplete exponent */
210f   /* Illegal: no decimal or exponent */
.e55   /* Illegal: missing integer or fraction */

String Literals Examples

Syntax of string literal:

```
type _string struct {
    elements *byte // underlying bytes
    len      int   // number of bytes
}
```

Example:

```
"hello, SC"

"hello, \
SC"

"hello, " "S" "hello"
```

Const Keyword

```
const variable type = value;  //declaring
```

Example:

```
package main

import "fmt"

func main() {
   const LEN int = 4
   const WID int = 5
   var area int

   area = LEN * WID
   fmt.Printf("value of area : %d", area)
}
```

Output

```
value of area : 20
```

Escape Sequence

Escape sequence	**Meaning**
\\	\ character

\’	‘ character
\”	” character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Decision-Making Structures

The decision-making structures help programmers perform tasks based on the condition to be evaluated. If the specified condition is true, the code will run, or it will run alternate code mentioned within the program.

Golag supports the following types of decision making structure statements:

- If
- if..else
- Nested if
- Switch
- Select

If Statement

If the condition specified is true, then only the block of code executes.

Syntax:

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

Example:

```go
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 15

   /* check the boolean condition using if statement */
   if( a < 20 ) {
      /* if condition is true then print the following */
      fmt.Printf("a is less than 20\n" )
   }
   fmt.Printf("value of a is : %d\n", a)
}
```

if..else Statement

If the specified condition holds true, the 'if' block executes. Otherwise, the 'else' block executes.

Syntax:

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

Example:

```go
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100;

   /* check the boolean condition */
   if( a < 20 ) {
      /* if condition is true then print the following */
      fmt.Printf("a is less than 20\n" );
   } else {
      /* if condition is false then print the following */
      fmt.Printf("a is not less than 20\n" );
   }
   fmt.Printf("value of a is : %d\n", a);
}
```

Nested “If” Statement

The nested if statement implies one ‘if’ statement inside another.

Syntax:

```
if( boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2) {
      /* Executes when the boolean expression 2 is true */
   }
}
```

Example:

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int = 200

   /* check the boolean condition */
   if( a == 100 ) {
      /* if condition is true then check the following */
      if( b == 200 )  {
         /* if condition is true then print the following */
         fmt.Printf("Value of a is 100 and b is 200\n" );
      }
   }
   fmt.Printf("Exact value of a is : %d\n", a );
   fmt.Printf("Exact value of b is : %d\n", b );
}
```

Switch Statement

A switch statement is a multi-way branch statement. You can specify multiple conditionals across many branches.

```
switch(boolean-expression or integral type){
   case boolean-expression or integral type :
      statement(s);
   case boolean-expression or integral type :
      statement(s);

   /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

Example:

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var grade string = "B"
   var marks int = 90

   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 );
}
```

Select Statement

The select statement is analogous to the switch statement. However, here the case statement refers to communication.

Syntax:

```
select {
   case communication clause  :
      statement(s);
   case communication clause  :
      statement(s);
   /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

Example:

```
package main

import "fmt"

func main() {
   var c1, c2, c3 chan int
   var i1, i2 int
   select {
      case i1 = <-c1:
         fmt.Printf("received ", i1, " from c1\n")
```

```
        case c2 <- i2:
            fmt.Printf("sent ", i2, " to c2\n")
        case i3, ok := (<-c3):  // same as: i3, ok := <-c3
            if ok {
                fmt.Printf("received ", i3, " from c3\n")
            } else {
                fmt.Printf("c3 is closed\n")
            }
        default:
            fmt.Printf("no communication\n")
    }
}
```

Loops

If you want to execute some part of the code several times, you can implement loops. Read on for more details about the loops Golang supports.

“for” Loop

When you need to execute a specific block of code for a particular number of times, you can use the ‘for’ loop.

Syntax:

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

Example:

```
package main

import "fmt"

func main() {
    var b int = 15
    var a int
    numbers := [6]int{1, 2, 3, 5}

    /* for loop execution */
    for a := 0; a < 10; a++ {
       fmt.Printf("value of a: %d\n", a)
    }
    for a < b {
       a++
       fmt.Printf("value of a: %d\n", a)
    }
    for i,x:= range numbers {
       fmt.Printf("value of x = %d at %d\n", x,i)
    }
}
```

Nested Loops

The nested loop implies the loops inside another loop.

Syntax:

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

Example:

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var i, j int

   for i = 2; i < 100; i++ {
      for j = 2; j <= (i/j); j++ {
         if(i%j==0) {
            break; // if factor found, not prime
         }
      }
      if(j > (i/j)) {
         fmt.Printf("%d is prime\n", i);
      }
   }
}
```

Simple Range In For Loop

```
for i, j:= range rvariable{
   // statement..
}
```

Using for loop for Strings

A for loop can iterate over the Unicode code point for a string.

```
for index, chr:= range str{
     // Statement..
}
```

For Maps

A for loop can iterate over the key and value pairs of the map.

```
for key, value := range map {
     // Statement..
}
```

For Channel

A for loop can iterate over the sequential values sent on a channel until it closes.

```
for item := range Chnl {
     // statements..
}
```

Loop Control Statements

There are three loop control statements in Go:.

- Break
- Continue
- Goto statement

Break

Whenever you use a break statement within a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

Example:

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 10

   /* for loop execution */
   for a < 20 {
      fmt.Printf("value of a: %d\n", a);
      a++;
      if a > 15 {
         /* terminate the loop using break statement */
         break;
      }
   }
}
```

Continue

The continue statement works like a break statement; however, instead of performing a forced termination, a continue statement starts the next iteration of the loop, skipping any code in between.

But if you use “continue” with the “for” loop, it causes the conditional test and increment portions of the loop to execute.

Example:

```go
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 10

   /* do loop execution */
   for a < 20 {
      if a == 15 {
         /* skip the iteration */
         a = a + 1;
         continue;
      }
      fmt.Printf("value of a: %d\n", a);
      a++;
   }
}
```

Goto Statement

This statement performs an unconditional jump from the goto to a labeled statement in the same function.

Example:

```go
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 10

   /* do loop execution */
   LOOP: for a < 20 {
      if a == 15 {
         /* skip the iteration */
         a = a + 1
         goto LOOP
      }
      fmt.Printf("value of a: %d\n", a)
      a++
   }
}
```

Functions

As with any other object-oriented programming language, Golang allows you to create functions.

Syntax:

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

Where:

- **func:** To declare a function, you need to use the **func** keyword.
- **function_name:** It specifies the name of the function and you need to call it using this name with proper arguments.
- **Parameter list:** It defines the parameters used (if any) within the function, and these values are provided by the user during runtime. This value is referred to as an actual parameter or argument. The parameter list consists of the type, order, and number of the parameters.
- **return type:** A function may return a list of values. The return_types defines the type of that returned value.
- **body of function:** This is where you place the code to be executed.

Example:

```
func max(num1, num2 int) int {
   /* local variable declaration */
   result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

How to Call a Function

```go
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int = 200
   var ret int

   /* calling a function to get max value */
   ret = max(a, b)

   fmt.Printf( "Max value is : %d\n", ret )
}

/* function returning the max between two numbers */
func max(num1, num2 int) int {
   /* local variable declaration */
   var result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

How to Return Multiple Values from a Function

```go
package main

import "fmt"

func swap(x, y string) (string, string) {
   return y, x
}
func main() {
   a, b := swap("Mahesh", "Kumar")
   fmt.Println(a, b)
}
```

Function Arguments

Call by Value

The Go programming language uses the call by value method to pass arguments. This means that code within a function cannot alter the arguments used to call the function.

Example:

```
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;
}
```

Call by Reference

This method copies the argument address into the formal parameter. Inside the function, the address is used to access the actual argument used in the call.

Example:

```
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 */
}
```

Scope

The scope defines the area in the program where the defined variable can be used. In Golang, you can declare variables with three scopes.

- Local variables
- Global variables
- Formal variables

Local variables

These variables have scope within the function or code block where they have been declared. They cannot be accessed outside that code block.

Example:

```
package main

import "fmt"

func main() {
   /* local variable declaration */
   var a, b, c int

   /* actual initialization */
   a = 10
   b = 20
   c = a + b

   fmt.Printf ("value of a = %d, b = %d and c = %d\n", a, b, c)
}
```

Global Variables

These variables are declared outside the function or code block and can be accessed from functions within the program.

Example:

```go
package main

import "fmt"

/* global variable declaration */
var g int

func main() {
   /* local variable declaration */
   var a, b int

   /* actual initialization */
   a = 10
   b = 20
   g = a + b

   fmt.Printf("value of a = %d, b = %d and g = %d\n", a, b, g)
}
```

Formal Parameters

Formal parameters are treated as local variables within that function and they take preference over the global variables.

Example:

```go
package main

import "fmt"

/* global variable declaration */
var a int = 20;

func main() {
   /* local variable declaration in main function */
   var a int = 10
   var b int = 20
   var c int = 0

   fmt.Printf("value of a in main() = %d\n",  a);
   c = sum( a, b);
   fmt.Printf("value of c in main() = %d\n",  c);
}
/* function to add two integers */
func sum(a, b int) int {
   fmt.Printf("value of a in sum() = %d\n",  a);
   fmt.Printf("value of b in sum() = %d\n",  b);

   return a + b;
}
```

Arrays

An array is a data structure that stores elements of the same data type. It has a fixed-length. All array elements are stored at contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. For example, an array can store the marks of students in a specific subject.

Declaring an Array

Syntax: `var variable_name [SIZE] variable_type`
Example: `var balance [10] float32`

Initializing an Array

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

Accessing the Elements of an Array

```
float32 salary = balance[9]
```

Example:

```
package main

import "fmt"

func main() {
   var n [10]int /* n is an array of 10 integers */
   var i,j int

   /* initialize elements of array n to 0 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
   }
}
```

Pointers

Pointers are used to make complex tasks easier, such as call by reference, and cannot be performed without using pointers. As you know, every variable is a memory location having its address defined and that can be accessed using ampersand (&) operator. In short, a pointer is a variable that stores the address of another variable.

Syntax:

```
var var_name *var-type
```

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

Example:

```
package main
import "fmt"
func main() {
   var a int = 20   /* actual variable declaration */
   var ip *int      /* pointer variable declaration */

   ip = &a  /* store address of a in pointer 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 )
}
```

Nil Pointers

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

Defer Keyword

You can create a deferred method, or function, or anonymous function by using the defer keyword.

- // 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
}()
```

Structure (struct)

With structures, you can create user-defined data types with information about various data types. Unlike arrays, structures can store data items of *different* data types. For example, a structure can store the name, class, age, and marks of students of a specific class.

Syntax:

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

Accessing a Structure’s Members

```
package main

import "fmt"

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

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

   /* book 2 specification */
   Book2.title = "Telecom Billing"
   Book2.author = "Zara Ali"
   Book2.subject = "Telecom Billing Tutorial"
   Book2.book_id = 6495700

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

   /* print Book2 info */
```

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

Pointers to Structure

```
var struct_pointer *Books
struct_pointer = &Book1;
```

Example:

```
package main

import "fmt"

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

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

    /* book 2 specification */
    Book2.title = "Telecom Billing"
    Book2.author = "Zara Ali"
    Book2.subject = "Telecom Billing Tutorial"
    Book2.book_id = 6495700

    /* print Book1 info */
```

```
   printBook(&Book1)

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

Slice

Go Slice is an abstraction over Go Array. Go Array allows you to define variables to hold several data items of the same kind but it does not provide any inbuilt method to increase its size dynamically or get a sub-array of its own. But Slice in Go can overcome this problem.

Defining a Slice

```
var numbers []int /* a slice of unspecified size */
/* numbers == []int{0,0,0,0,0}*/
numbers = make([]int,5,5) /* a slice of length 5 and capacity 5*/
```

len() and cap() Function

The len() function returns the elements presents in the slice, whereas the cap() function returns the capacity of the slice.

```
package main

import "fmt"

func main() {
   var numbers = make([]int,3,5)
   printSlice(numbers)
}
func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

Nil Slice

```go
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   if(numbers == nil){
      fmt.Printf("slice is nil")
   }
}
func printSlice(x []int){
   fmt.Printf("len = %d cap = %d slice = %v\n", len(x), cap(x),x)
}
```

append() and copy() Functions

```go
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* append allows nil slice */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* add one element to slice*/
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* add more than one element at a time*/
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* create a slice numbers1 with double the capacity of earlier slice*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* copy content of numbers to numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)
}
func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

Range

The range keyword is used in for loop to iterate over items of an array, slice, channel, or map. With array and slices, it returns the index of the item as integer. With maps, it returns the key of the next key-value pair.

Range expression	**1st Value**	**2nd Value(Optional)**
Array or slice a [n]E	index i int	a[i] E
String s string type	index i int	rune int
map m map[K]V	key k K	value m[k] V
channel c chan E	element e E	none

Example:

```
package main

import "fmt"

func main() {
   /* create a slice */
   numbers := []int{0,1,2,3,4,5,6,7,8}

   /* print the numbers */
   for i:= range numbers {
      fmt.Println("Slice item",i,"is",numbers[i])
   }
```

```go
   /* create a map*/
   countryCapitalMap := map[string] string
{"France":"Paris","Italy":"Rome","Japan":"Tokyo"}

   /* print map using keys*/
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }

   /* print map using key-value*/
   for country,capital := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",capital)
   }
}
```

Maps

Golang offers another important data type called map, which maps unique keys to values. Here, a key is an object used to retrieve a value. You can store the value (key and value) in a Map object.

Defining a Map

Syntax:

```go
/* declare a variable, by default map will be nil*/
var map_variable map[key_data_type]value_data_type

/* define the map as nil map can not be assigned any value*/
map_variable = make(map[key_data_type]value_data_type)
```

Example:

```go
package main
import "fmt"
func main() {
   var countryCapitalMap map[string]string
   /* create a map*/
   countryCapitalMap = make(map[string]string)
   /* insert key-value pairs in the map*/
   countryCapitalMap["France"] = "Paris"
   countryCapitalMap["Italy"] = "Rome"
   countryCapitalMap["Japan"] = "Tokyo"
   countryCapitalMap["India"] = "New Delhi"

   /* print map using keys*/
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }

   /* test if entry is present in the map or not*/
   capital, ok := countryCapitalMap["United States"]

   /* if ok is true, entry is present otherwise entry is absent*/
   if(ok){
      fmt.Println("Capital of United States is", capital)
   } else {
      fmt.Println("Capital of United States is not present")
   }
}
```

Delete() Function

```
package main
import "fmt"
func main() {
   /* create a map*/
   countryCapitalMap := map[string] string
{"France":"Paris","Italy":"Rome","Japan":"Tokyo","India":"New Delhi"}

   fmt.Println("Original map")

   /* print map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }

   /* delete an entry */
   delete(countryCapitalMap,"France");
   fmt.Println("Entry for France is deleted")

   fmt.Println("Updated map")

   /* print map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
}
```

Recursion

Through the recursion process, you can repeat items and apply the same concept. When one function calls another function inside it, it is called a recursive function call.

Syntax:

```
func recursion() {
   recursion() /* function calls itself */
}
func main() {
   recursion()
}
```

Example (calculating factorial):

```
package main

import "fmt"

func factorial(i int)int {
   if(i <= 1) {
      return 1
   }
   return i * factorial(i - 1)
}
func main() {
   var i int = 15
   fmt.Printf("Factorial of %d is %d", i, factorial(i))
}
```

Type Conversion

This converts a variable from one data type to another data type.

Syntax:

```
type_name(expression)
```

Example:

```
package main

import "fmt"

func main() {
   var sum int = 17
   var count int = 5
   var mean float32

   mean = float32(sum)/float32(count)
   fmt.Printf("Value of mean : %f\n",mean)
}
```

Interfaces

Interfaces represent a set of method signatures.

Syntax:

```
/* define an interface */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   method_name3 [return_type]
   ...
   method_namen [return_type]
}

/* define a struct */
type struct_name struct {
   /* variables */
}
```

```
/* implement interface methods*/
func (struct_name_variable struct_name) method_name1() [return_type] {
   /* method implementation */
}
...
func (struct_name_variable struct_name) method_namen() [return_type] {
   /* method implementation */
}
```

Example:

```
package main

import ("fmt" "math")

/* define an interface */
type Shape interface {
   area() float64
}

/* define a circle */
type Circle struct {
   x,y,radius float64
}

/* define a rectangle */
type Rectangle struct {
   width, height float64
}

/* define a method for circle (implementation of Shape.area())*/
func(circle Circle) area() float64 {
   return math.Pi * circle.radius * circle.radius
}

/* define a method for rectangle (implementation of Shape.area())*/
func(rect Rectangle) area() float64 {
   return rect.width * rect.height
}
```

```
/* define a method for shape */
func getArea(shape Shape) float64 {
   return shape.area()
}

func main() {
   circle := Circle{x:0,y:0,radius:5}
   rectangle := Rectangle {width:10, height:5}

   fmt.Printf("Circle area: %f\n",getArea(circle))
   fmt.Printf("Rectangle area: %f\n",getArea(rectangle))
}
```

Error Handling

Error handling implies response and recovery procedures from various error conditions.

Syntax:

```
type error interface {
   Error() string
}
```

Example:

```
package main

import "errors"
import "fmt"
import "math"

func Sqrt(value float64)(float64, error) {
   if(value < 0){
      return 0, errors.New("Math: negative number passed to Sqrt")
   }
   return math.Sqrt(value), nil
}
func main() {
   result, err:= Sqrt(-1)

   if err != nil {
      fmt.Println(err)
```

```go
    } else {
       fmt.Println(result)
    }

    result, err = Sqrt(9)

    if err != nil {
       fmt.Println(err)
    } else {
       fmt.Println(result)
    }
}
```

Embedding

Go does not support subclassing. So, it uses embedding for interface and struct.

Example:

```go
// ReadWriter implementations must satisfy both Reader and Writer
type ReadWriter interface {
    Reader
    Writer
}

// Server exposes all the methods that Logger has
type Server struct {
    Host string
    Port int
    *log.Logger
}

// initialize the embedded type the usual way
server := &Server{"localhost", 80, log.New(...)}

// methods implemented on the embedded struct are passed through
server.Log(...) // calls server.Logger.Log(...)

// the field name of the embedded type is its type name (in this case
Logger)
var logger *log.Logger = server.Logger
```

Goroutines

Goroutines allow the functions to run independent of each other. Goroutines are basically functions that are run concurrently. You can use the “go” statement to create goroutines.

```
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
```

The go keyword makes the function call to return immediately, while the function starts running in the background as a goroutine and the rest of the program continues its execution. The goroutine starts the main function.

Example:

```
// just a function (which can be later started as a goroutine)
func doStuff(s string) {
}

func main() {
    // using a named function in a goroutine
    go doStuff("foobar")

    // using an anonymous inner function in a goroutine
    go func (x int) {
        // function body goes here
    }(42)
}
```

Channels

Channels share data between goroutines. When you execute a concurrent activity as a goroutine, it shares resources between goroutines. Channels act as a pipe between the goroutines to guarantee a synchronous exchange.

There are two types of channels based on their behavior of data exchange: unbuffered channels and buffered channels.

Syntax:

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

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

Example:

```go
ch := make(chan int) // create a channel of type int
ch <- 42             // Send a value to the channel ch.
v := <-ch            // Receive a value from ch

// Non-buffered channels block. Read blocks when no value is available,
write blocks until there is a read.

// Create a buffered channel. Writing to a buffered channels does not block
if less than <buffer size> unread values have been written.
ch := make(chan int, 100)

close(ch) // closes the channel (only sender should close)

// read from channel and test if it has been closed
v, ok := <-ch

// if ok is false, channel has been closed

// Read from channel until it is closed
for i := range ch {
    fmt.Println(i)
}

// select blocks on multiple channel operations, if one unblocks, the
corresponding case is executed
func doStuff(channelOut, channelIn chan int) {
    select {
    case channelOut <- 42:
        fmt.Println("We could write to channelOut!")
    case x := <- channelIn:
        fmt.Println("We could read from channelIn")
    case <-time.After(time.Second * 1):
        fmt.Println("timeout")
    }
}
```

Logs

Golang has a standard library package 'log' for log management. It traces the details, location, and time for what's happening in the program. Logs help you find potential bugs and understand the program's functioning.

Syntax:

```
import (
      "log"
)
```

Example:

```
package main
import (
      "log"
)
func init(){
      log.SetPrefix("LOG: ")
      log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
      log.Println("init started")
}
func main() {
// Println writes to the standard logger.
      log.Println("main started")

// Fatalln is Println() followed by a call to os.Exit(1)
      log.Fatalln("fatal message")

// Panicln is Println() followed by a call to panic()
      log.Panicln("panic message")
}
```

Files and Directories

Golang offers an "os" package to manipulate with files and directories.

Creating an Empty File

```
package main
```

```go
import (
	"log"
	"os"
)

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

Creating a Directory

```go
package main

import (
	"log"
	"os"
)

func main() {
	_, err := os.Stat("test")

	if os.IsNotExist(err) {
		errDir := os.MkdirAll("test", 0755)
		if errDir != nil {
			log.Fatal(err)
		}
	}
}
```

Renaming a File

```go
package main
import (
	"log"
	"os"
)

func main() {
	oldName := "test.txt"
	newName := "testing.txt"
	err := os.Rename(oldName, newName)
	if err != nil {
		log.Fatal(err)
	}
}
```

Copying File to Destination

```go
package main

import (
	"io"
	"log"
	"os"
)

func main() {

	sourceFile, err := os.Open("/var/www/html/src/test.txt")
	if err != nil {
		log.Fatal(err)
	}
	defer sourceFile.Close()

	// Create new file
	newFile, err := os.Create("/var/www/html/test.txt")
	if err != nil {
		log.Fatal(err)
	}
	defer newFile.Close()

	bytesCopied, err := io.Copy(newFile, sourceFile)
	if err != nil {
		log.Fatal(err)
	}
	log.Printf("Copied %d bytes.", bytesCopied)
}
```

Getting Metadata of a File

```go
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()
}
```

Deleting a File

```go
package main

import (
      "log"
      "os"
)

func main() {
      err := os.Remove("/var/www/html/test.txt")
      if err != nil {
            log.Fatal(err)
      }
}
```

Reading Characters from a File

```go
package main

import (
      "bufio"
      "fmt"
      "io/ioutil"
      "os"
      "strings"
)

func main() {
      filename := "test.txt"

      filebuffer, err := ioutil.ReadFile(filename)
      if err != nil {
            fmt.Println(err)
            os.Exit(1)
      }
      inputdata := string(filebuffer)
      data := bufio.NewScanner(strings.NewReader(inputdata))
      data.Split(bufio.ScanRunes)

      for data.Scan() {
            fmt.Print(data.Text())
      }
}
```

Truncating the Content of a File

```go
package main

import (
      "log"
      "os"
)

func main() {
	err := os.Truncate("test.txt", 100)

	if err != nil {
		log.Fatal(err)
	}
}
```

Appending Content to a File

```
package main

import (
	"fmt"
	"os"
)

func main() {
	message := "Add this content at end"
	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)
}
```

Compressing Several Files

```
package main

import (
	"archive/zip"
	"fmt"
	"io"
	"log"
	"os"
)

func appendFiles(filename string, zipw *zip.Writer) error {
	file, err := os.Open(filename)
	if err != nil {
		return fmt.Errorf("Failed to open %s: %s", filename, err)
	}
	defer file.Close()

	wr, err := zipw.Create(filename)
	if err != nil {
		msg := "Failed to create entry for %s in zip file: %s"
		return fmt.Errorf(msg, filename, err)
	}

	if _, err := io.Copy(wr, file); err != nil {
		return fmt.Errorf("Failed to write %s to zip: %s", filename,
err)
	}

	return nil
}

func main() {
	flags := os.O_WRONLY | os.O_CREATE | os.O_TRUNC
	file, err := os.OpenFile("test.zip", flags, 0644)
	if err != nil {
		log.Fatalf("Failed to open zip for writing: %s", err)
	}
	defer file.Close()

	var files = []string{"test1.txt", "test2.txt", "test3.txt"}

	zipw := zip.NewWriter(file)
	defer zipw.Close()

	for _, filename := range files {
		if err := appendFiles(filename, zipw); err != nil {
```

```
                  log.Fatalf("Failed to add file %s to zip: %s", filename, 
err)
            }
      }
}
```

Golang Regex Cheat Sheet

It is a tool that is used to describe a search pattern for matching the text. Regex is nothing but a sequence of some characters that defines a search pattern.

Extracting Text Between Square Brackets

```
package main

import (
      "fmt"
      "regexp"
      "strings"
)

func main() {
      str1 := "this is a [sample] [[string]] with [SOME] special words"

      re := regexp.MustCompile(`\[([^\[\]]*)\]`)
      fmt.Printf("Pattern: %v\n", re.String())      // print pattern
      fmt.Println("Matched:", re.MatchString(str1)) // true

      fmt.Println("\nText between square brackets:")
      submatchall := re.FindAllString(str1, -1)
      for _, element := range submatchall {
            element = strings.Trim(element, "[")
            element = strings.Trim(element, "]")
            fmt.Println(element)
      }
}
```

Find DNS Records

DNS records are mapping files that associate with DNS server whichever IP addresses each domain is associated with, and they handle requests sent to each domain. Golang provides the “net” package offering various methods to get information of DNS records.

net.LookupIP()

Returns a slice of net.IP objects that contains host's IPv4 and IPv6 addresses.

```go
package main

import (
	"fmt"
	"net"
)

func main() {
	iprecords, _ := net.LookupIP("facebook.com")
	for _, ip := range iprecords {
		fmt.Println(ip)
	}}
```

Canonical Name

CNAMEs are essentially domain and subdomain text aliases to bind traffic.

```go
package main

import (
	"fmt"
	"net"
)

func main() {
	cname, _ := net.LookupCNAME("m.facebook.com")
	fmt.Println(cname)
}
```

PTR

These records provide the reverse binding from addresses to names.

```go
package main

import (
	"fmt"
	"net"
)

func main() {
	ptr, _ := net.LookupAddr("6.8.8.8")
	for _, ptrvalue := range ptr {
		fmt.Println(ptrvalue)
	}
}
```

Name Server

The NS records describe the authorized name servers for the zone.

```
package main

import (
	"fmt"
	"net"
)

func main() {
	nameserver, _ := net.LookupNS("facebook.com")
	for _, ns := range nameserver {
		fmt.Println(ns)
	}
}
```

MX Records

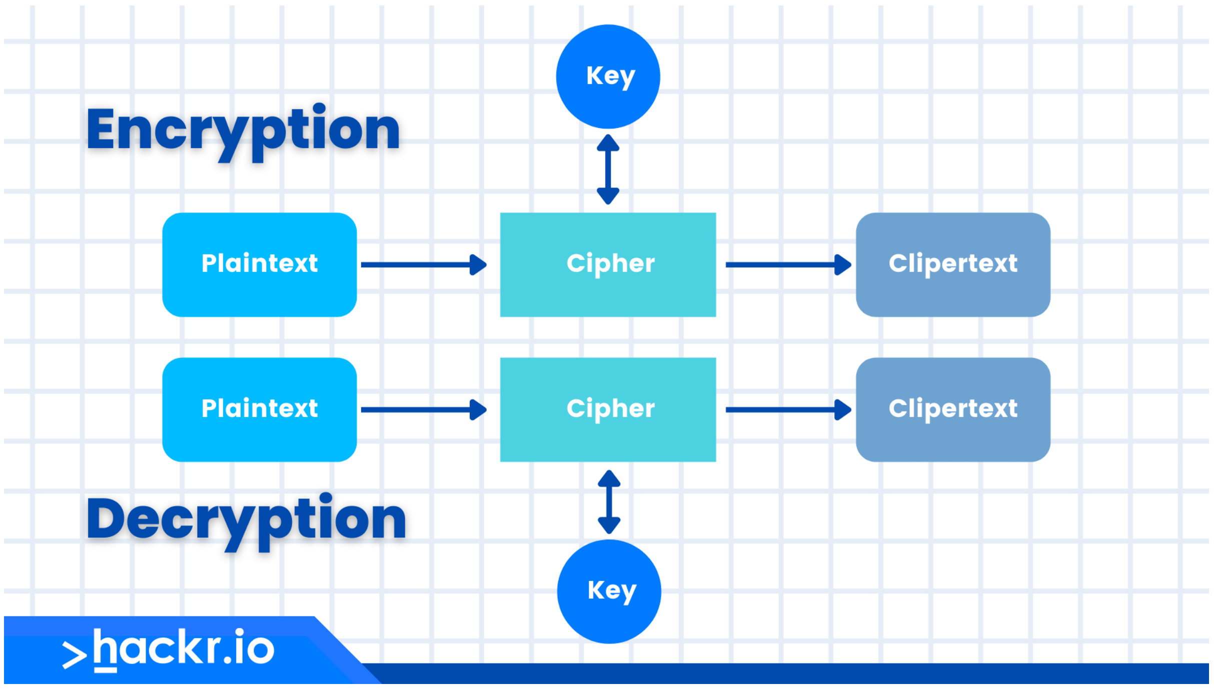
These records identify servers that can exchange emails.

```
package main

import (
	"fmt"
	"net"
)

func main() {
	mxrecords, _ := net.LookupMX("facebook.com")
```

```
      for _, mx := range mxrecords {
            fmt.Println(mx.Host, mx.Pref)
      }
}
```

Cryptography

Cryptography is the process of encrypting plain text into the cipertext so that its meaning is hidden from hackers.



Example:

Classical Cipher

```
package main

import (
    "fmt"
    "unicode"
)

// Cipher encrypts and decrypts a string.
type Cipher interface {
```

```go
    Encryption(string) string
    Decryption(string) string
}

// Cipher holds the key used to encrypts and decrypts messages.
type cipher []int

// cipherAlgorithm encodes a letter based on some function.
func (c cipher) cipherAlgorithm(letters string, shift func(int, int) int)
string {
    shiftedText := ""
    for _, letter := range letters {
        if !unicode.IsLetter(letter) {
            continue
        }
        shiftDist := c[len(shiftedText)%len(c)]
        s := shift(int(unicode.ToLower(letter)), shiftDist)
        switch {
        case s < 'a':
            s += 'z' - 'a' + 1
        case 'z' < s:
            s -= 'z' - 'a' + 1
        }
        shiftedText += string(s)
    }
    return shiftedText
}

// Encryption encrypts a message.
func (c *cipher) Encryption(plainText string) string {
    return c.cipherAlgorithm(plainText, func(a, b int) int { return a + b
})
}

// Decryption decrypts a message.
func (c *cipher) Decryption(cipherText string) string {
    return c.cipherAlgorithm(cipherText, func(a, b int) int { return a - b
})
}

// NewCaesar creates a new Caesar shift cipher.
func NewCaesar(key int) Cipher {
    return NewShift(key)
```

```
}

// NewShift creates a new Shift cipher.
func NewShift(shift int) Cipher {
    if shift < -25 || 25 < shift || shift == 0 {
        return nil
    }
    c := cipher([]int{shift})
    return &c
}

func main() {
    c := NewCaesar(1)
    fmt.Println("Encrypt Key(01) abcd =>", c.Encryption("abcd"))
    fmt.Println("Decrypt Key(01) bcde =>", c.Decryption("bcde"))
    fmt.Println()

    c = NewCaesar(10)
    fmt.Println("Encrypt Key(10) abcd =>", c.Encryption("abcd"))
    fmt.Println("Decrypt Key(10) klmn =>", c.Decryption("klmn"))
    fmt.Println()

    c = NewCaesar(15)
    fmt.Println("Encrypt Key(15) abcd =>", c.Encryption("abcd"))
    fmt.Println("Decrypt Key(15) pqrs =>", c.Decryption("pqrs"))
}
```

MD5 and SHA-1

```
package main

import (
      "crypto/md5"
      "crypto/sha1"
      "crypto/sha256"
      "crypto/sha512"
      "fmt"
)

func main() {
      fmt.Println("\n----------------Small Message----------------\n")
      message := []byte("Today web engineering has modern apps adhere to
what is known as a single-page app (SPA) model.")

	fmt.Printf("Md5: %x\n\n", md5.Sum(message))
	fmt.Printf("Sha1: %x\n\n", sha1.Sum(message))
	fmt.Printf("Sha256: %x\n\n", sha256.Sum256(message))
	fmt.Printf("Sha512: %x\n\n", sha512.Sum512(message))

	fmt.Println("\n\n----------------Large Message----------------\n")
	message = []byte("Today web engineering has modern apps.")

	fmt.Printf("Md5: %x\n\n", md5.Sum(message))
	fmt.Printf("Sha1: %x\n\n", sha1.Sum(message))
	fmt.Printf("Sha256: %x\n\n", sha256.Sum256(message))
	fmt.Printf("Sha512: %x\n\n", sha512.Sum512(message))
}
```

Hash-based MAC

```
package main

import (
	"crypto/hmac"
	"crypto/rand"
	"crypto/sha256"
	"crypto/sha512"
	"encoding/base64"
	"fmt"
	"io"
)

var secretKey = "4234kxzjcjj3nxnxbcvsjfj"

// Generate a salt string with 16 bytes of crypto/rand data.
func generateSalt() string {
	randomBytes := make([]byte, 16)
	_, err := rand.Read(randomBytes)
	if err != nil {
		return ""
	}
	return base64.URLEncoding.EncodeToString(randomBytes)
}
```

```
func main() {
	message := "Today web engineering has modern apps adhere to what is
known as a single-page app (SPA) model."
	salt := generateSalt()
	fmt.Println("Message: " + message)
	fmt.Println("\nSalt: " + salt)

	hash := hmac.New(sha256.New, []byte(secretKey))
	io.WriteString(hash, message+salt)
	fmt.Printf("\nHMAC-Sha256: %x", hash.Sum(nil))

	hash = hmac.New(sha512.New, []byte(secretKey))
	io.WriteString(hash, message+salt)
	fmt.Printf("\n\nHMAC-sha512: %x", hash.Sum(nil))
}
```

Advanced Encryption Standard (AES)

```
package main

import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
	"encoding/pem"
	"fmt"
	"io/ioutil"
	"log"
)

const (
	keyFile       = "aes.key"
	encryptedFile = "aes.enc"
)

var IV = []byte("1234567812345678")

func readKey(filename string) ([]byte, error) {
	key, err := ioutil.ReadFile(filename)
	if err != nil {
		return key, err
	}
	block, _ := pem.Decode(key)
	return block.Bytes, nil
}

func createKey() []byte {
	genkey := make([]byte, 16)
	_, err := rand.Read(genkey)
	if err != nil {
		log.Fatalf("Failed to read new random key: %s", err)
	}
	return genkey
}

func saveKey(filename string, key []byte) {
	block := &pem.Block{
		Type:  "AES KEY",
		Bytes: key,
	}
	err := ioutil.WriteFile(filename, pem.EncodeToMemory(block), 0644)
	if err != nil {
		log.Fatalf("Failed in saving key to %s: %s", filename, err)
	}
}

func aesKey() []byte {
	file := fmt.Sprintf(keyFile)
	key, err := readKey(file)
	if err != nil {
		log.Println("Creating a new AES key")
		key = createKey()
		saveKey(file, key)
	}
	return key
}

func createCipher() cipher.Block {
	c, err := aes.NewCipher(aesKey())
	if err != nil {
		log.Fatalf("Failed to create the AES cipher: %s", err)
	}
	return c
}
```

```go
func encryption(plainText string) {
	bytes := []byte(plainText)
	blockCipher := createCipher()
	stream := cipher.NewCTR(blockCipher, IV)
	stream.XORKeyStream(bytes, bytes)
	err := ioutil.WriteFile(fmt.Sprintf(encryptedFile), bytes, 0644)
	if err != nil {
		log.Fatalf("Writing encryption file: %s", err)
	} else {
		fmt.Printf("Message encrypted in file: %s\n\n", encryptedFile)
	}
}

func decryption() []byte {
	bytes, err := ioutil.ReadFile(fmt.Sprintf(encryptedFile))
	if err != nil {
		log.Fatalf("Reading encrypted file: %s", err)
	}
	blockCipher := createCipher()
	stream := cipher.NewCTR(blockCipher, IV)
	stream.XORKeyStream(bytes, bytes)
	return bytes
}

func main() {

	var plainText = "AES is now being used worldwide for encrypting
digital information, including financial, and government data."
	encryption(plainText)

	fmt.Printf("Decrypted Message: %s", decryption())
}
```