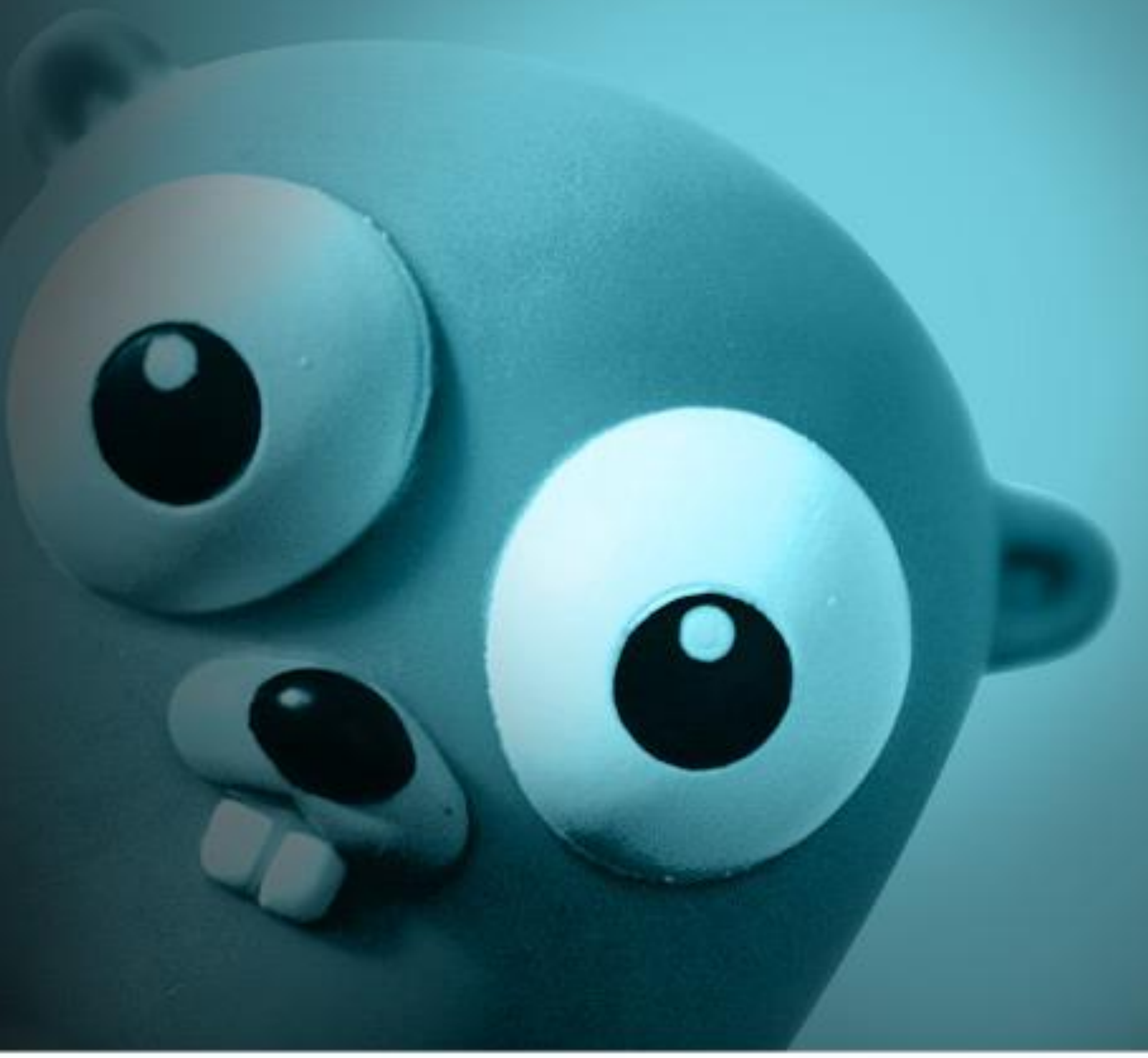
Go Programming

Programming Language



tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Go language is a programming language initially developed at Google in the year 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is a statically-typed language having syntax similar to that of C. It provides garbage collection, type safety, dynamic-typing capability, many advanced built-in types such as variable length arrays and key-value maps. It also provides a rich standard library.

The Go programming language was launched in November 2009 and is used in some of the Google's production systems.

Audience

This tutorial is designed for software programmers with a need to understand the Go programming language from scratch. This tutorial will give you enough understanding on Go programming language from where you can take yourself to higher levels of expertise.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of computer programming terminologies. If you have a good command over C, then it would be quite easy for you to understand the concepts of Go programming and move fast on the learning track.

Disclaimer & Copyright

Table of Contents

1. OVERVIEW

Go is a general-purpose language designed with systems programming in mind. It was initially developed at Google in the year 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is strongly and statically typed, provides inbuilt support for garbage collection, and supports concurrent programming.

Programs are constructed using packages, for efficient management of dependencies. Go programming implementations use a traditional compile and link model to generate executable binaries. The Go programming language was announced in November 2009 and is used in some of the Google's production systems.

Features of Go Programming

The most important features of Go programming are listed below:

- Support for environment adopting patterns similar to dynamic languages. For example, type inference (x := 0 is valid declaration of a variable x of type int)

- Compilation time is fast.

- Inbuilt concurrency support: lightweight processes (via go routines), channels, select statement.

- Go programs are simple, concise, and safe.

- Support for Interfaces and Type embedding.

- Production of statically linked native binaries without external dependencies.

Features Excluded Intentionally

To keep the language simple and concise, the following features commonly available in other similar languages are omitted in Go:

- Support for type inheritance
- Support for method or operator overloading
- Support for circular dependencies among packages
- Support for pointer arithmetic
- Support for assertions
- Support for generic programming

Go Programs

A Go program can vary in length from 3 lines to millions of lines and it should be written into one or more text files with the extension ".go". For example, hello.go.

You can use "vi", "vim" or any other text editor to write your Go program into a file.

Compiling and Executing Go Programs

For most of the examples given in this tutorial, you will find a **Try it** option, so just make use of it and enjoy your learning.

Try the following example using the **Try it** option available at the top right corner of the following sample code:

```
package main


import "fmt"


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

2. ENVIRONMENT SETUP

Try it Option Online

You really do not need to set up your own environment to start learning Go programming language. Reason is very simple, we already have set up Go Programming environment online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work.

This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using the **Try it** option available at the top right corner of the following sample code displayed on our website:

```
package main


import "fmt"


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

For most of the examples given in this tutorial, you will find a **Try it** option.

Local Environment Setup

If you are still willing to set up your environment for Go programming language, you need the following two software available on your computer:

- A text editor
- Go compiler

Text Editor

You will require a text editor to type your programs. Examples of text editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The name and version of text editors can vary on different operating systems. For example, Notepad is used on Windows, and vim or vi is used on Windows as well as Linux or UNIX.

The files you create with the text editor are called **source files**. They contain program source code. The source files for Go programs are typically named with the extension "**.go**".

Before starting your programming, make sure you have a text editor in place and you have enough experience to write a computer program, save it in a file, compile it, and finally execute it.

The Go Compiler

The source code written in source file is the human readable source for your program. It needs to be *compiled* and turned into machine language so that your CPU can actually execute the program as per the instructions given. The Go programming language compiler compiles the source code into its final executable program.

Go distribution comes as a binary installable for FreeBSD (release 8 and above), Linux, Mac OS X (Snow Leopard and above), and Windows operating systems with 32-bit (386) and 64-bit (amd64) x86 processor architectures.

The following section explains how to install Go binary distribution on various OS.

Download Go Archive

Download the latest version of Go installable archive file from Go Downloads. The following version is used in this tutorial: *go1.4.windows-amd64.msi.*

It is copied it into C:\>go folder.

OS	Archive name
Windows	go1.4.windows-amd64.msi
Linux	go1.4.linux-amd64.tar.gz
Mac	go1.4.darwin-amd64-osx10.8.pkg
FreeBSD	go1.4.freebsd-amd64.tar.gz

Installation on UNIX/Linux/Mac OS X, and FreeBSD

Extract the download archive into the folder /usr/local, creating a Go tree in /usr/local/go. For example:

tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz

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

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

Installation on Windows

Use the MSI file and follow the prompts to install the Go tools. By default, the installer uses the Go distribution in c:\Go. The installer should set the c:\Go\bin directory in Window's PATH environment variable. Restart any open command prompts for the change to take effect.

Verifying the Installation

Create a go file named test.go in C:\>Go_WorkSpace.

File: test.go

```
package main


import "fmt"


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

Now run test.go to see the result:

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

Output

```
Hello, World!
```

3. PROGRAM STRUCTURE

Before we study the basic building blocks of Go programming language, let us first discuss the bare minimum structure of Go programs so that we can take it as a reference in subsequent chapters.

Hello World Example

A Go program basically consists of the following parts:

- Package Declaration
- Import Packages
- Functions
- Variables
- Statements and Expressions
- Comments

Let us look at a simple code that would print the words "Hello World!":

```
package main

import "fmt"

func main() {
   /* This is my first sample program. */
   fmt.Println("Hello, World!")
}
```

Let us take a look at the various parts of the above program:

1. The first line of the program package *main* defines the package name in which this program should lie. It is a mandatory statement, as Go programs run in packages. The *main* package is the starting point to run the program. Each package has a path and name associated with it.

2. The next line import "fmt" is a preprocessor command which tells the Go compiler to include the files lying in the package fmt.

3. The next line func main() is the main function where the program execution begins.

4. The next line /*...*/ is ignored by the compiler and it is there to add comments in the program. Comments are also represented using // similar to Java or C++ comments.

5. The next line fmt.Println(...) is another function available in Go which causes the message "Hello, World!" to be displayed on the screen. Here fmt package has exported Println method which is used to display the message on the screen.

6. Notice the capital P of Println method. In Go language, a name is exported if it starts with capital letter. Exported means the function or variable/constant is accessible to the importer of the respective package.

Executing a Go Program

Let us discuss how to save the source code in a file, compile it, and finally execute the program. Please follow the steps given below:

1. Open a text editor and add the above-mentioned code.

2. Save the file as *hello.go*

3. Open the command prompt.

4. Go to the directory where you saved the file.

5. Type *go run hello.go* and press enter to run your code.

6. If there are no errors in your code, then you will see *"Hello World!"* printed on the screen.

```
$ go run hello.go

Hello, World!
```

Make sure the Go compiler is in your path and that you are running it in the directory containing the source file hello.go.

4. BASIC SYNTAX

We discussed the basic structure of a Go program in the previous chapter. Now it will be easy to understand the other basic building blocks of the Go programming language.

Tokens in Go

A Go program consists of various tokens. A token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following Go statement consists of six tokens:

```
fmt.Println("Hello, World!")
```

The individual tokens are:

```
fmt
.
Println
(
"Hello, World!"
)
```

Line Separator

In a Go program, the line separator key is a statement terminator. That is, individual statements don't need a special separator like ";" in C. The Go compiler internally places ";" as the statement terminator to indicate the end of one logical entity.

For example, take a look at the following statements:

```
fmt.Println("Hello, World!")
fmt.Println("I am in Go Programming World!")
```

Comments

Comments are like helping texts in your Go program and they are ignored by the compiler. They start with /* and terminate with the characters */ as shown below:

```
/* my first program in Go */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

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

identifier = letter { letter | unicode_digit } .

Go does not allow punctuation characters such as @, $, and % within identifiers. Go is a **case-sensitive** programming language. Thus, *Manpower* and *manpower* are two different identifiers in Go. Here are some examples of acceptable identifiers:

```
mahesh    kumar    abc    move_name    a_123
myname50    _temp    j    a23b9    retVal
```

Keywords

The following list shows the reserved words in Go. These reserved words may not be used as constant or variable or any other identifier names.

break	Default	Func	interface	Select
case	Defer	Go	map	Struct
chan	Else	Goto	package	Switch
const	fallthrough	if	range	Type
continue	For	import	return	Var

Whitespace in Go

Whitespace is the term used in Go to describe blanks, tabs, newline characters, and comments. A line containing only whitespace, possibly with a comment, is known as a blank line, and a Go compiler totally ignores it.

Whitespaces separate one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:

```
var age int;
```

There must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement:

```
fruit = apples + oranges;   // get the total fruit
```

No whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

5. DATA TYPES

In the Go programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in Go can be classified as follows:

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

Array types and structure types are collectively referred to as **aggregate types**. The type of a function specifies the set of all functions with the same parameter and result types. We will discuss the basic types in the following section, whereas other types will be covered in the upcoming chapters.

Integer Types

The predefined architecture-independent integer types are:

Sr. No.	Types and Description
1	**uint8** Unsigned 8-bit integers (0 to 255)
2	**uint16** Unsigned 16-bit integers (0 to 65535)
3	**uint32** Unsigned 32-bit integers (0 to 4294967295)
4	**uint64** Unsigned 64-bit integers (0 to 18446744073709551615)
5	**int8** Signed 8-bit integers (-128 to 127)
6	**int16** Signed 16-bit integers (-32768 to 32767)
7	**int32** Signed 32-bit integers (-2147483648 to 2147483647)
8	**int64** Signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

Floating Types

The predefined architecture-independent float types are:

Sr. No.	Types and Description
1	**float32** IEEE-754 32-bit floating-point numbers
2	**float64** IEEE-754 64-bit floating-point numbers

3	**complex64** Complex numbers with float32 real and imaginary parts
4	**complex128** Complex numbers with float64 real and imaginary parts

The value of an n-bit integer is n bits and is represented using two's complement arithmetic operations.

Other Numeric Types

There is also a set of numeric types with implementation-specific sizes:

Sr. No.	Types and Description
1	**byte** same as uint8
2	**rune** same as int32
3	**uint** 32 or 64 bits
4	**int** same size as uint
5	**uintptr** an unsigned integer to store the uninterpreted bits of a pointer value

6. VARIABLES

A variable is nothing but a name given to a storage area that the programs can manipulate. Each variable in Go has a specific type, which determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Go is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types:

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

Go programming language also allows to define various other types of variables such as Enumeration, Pointer, Array, Structure, and Union, which we will discuss in subsequent chapters. In this chapter, we will focus only basic variable types.

Variable Definition in Go

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
var variable_list optional_data_type;
```

Here, **optional_data_type** is a valid Go data type including byte, int, float32, complex64, boolean or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
var    i, j, k int;
var   c, ch byte;
var  f, salary float32;
d = 42;
```

The statement **"var i, j, k;"** declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j, and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The type of variable is automatically judged by the compiler based on the value passed to it. The initializer consists of an equal sign followed by a constant expression as follows:

```
variable_name = value;
```

For example,

```
d = 3, f = 5;    // declaration of d and f. Here d and f are int
```

For definition without an initializer: variables with static storage duration are implicitly initialized with nil (all bytes have the value 0); the initial value of all other variables is zero value of their data type.

Static Type Declaration in Go

A static type variable declaration provides assurance to the compiler that there is one variable available with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail of the variable. A variable declaration has its meaning at the time of compilation only, the compiler needs the actual variable declaration at the time of linking of the program.

Example

Try the following example, where the variable has been declared with a type and initialized inside the main function:

```
package main

import "fmt"

func main() {
   var x float64
   x = 20.0
   fmt.Println(x)
   fmt.Printf("x is of type %T\n", x)
}
```

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

```
20
x is of type float64
```

Dynamic Type Declaration / Type Inference in Go

A dynamic type variable declaration requires the compiler to interpret the type of the variable based on the value passed to it. The compiler does not require a variable to have type statically as a necessary requirement.

Example

Try the following example, where the variables have been declared without any type. Notice, in case of type inference, we initialized the variable **y** with **:=** operator, whereas **x** is initialized using **=** operator.

```
package main


import "fmt"


func main() {
   var x float64 = 20.0


   y := 42
   fmt.Println(x)
   fmt.Println(y)
   fmt.Printf("x is of type %T\n", x)
   fmt.Printf("y is of type %T\n", y)
}
```

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

```
20
42
x is of type float64
y is of type int
```

Mixed Variable Declaration in Go

Variables of different types can be declared in one go using type inference.

Example

```
package main

import "fmt"

func main() {
   var a, b, c = 3, 4, "foo"

   fmt.Println(a)
   fmt.Println(b)
   fmt.Println(c)
   fmt.Printf("a is of type %T\n", a)
   fmt.Printf("b is of type %T\n", b)
   fmt.Printf("c is of type %T\n", c)
}
```

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

```
3
4
foo
a is of type int
b is of type int
c is of type string
```

The lvalues and the rvalues in Go

There are two kinds of expressions in Go:

1. **lvalue:** Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.

2. **rvalue:** The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side.

The following statement is valid:

```
x = 20.0
```

The following statement is not valid. It would generate compile-time error:

```
10 = 20
```

7. CONSTANTS

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*. There are also enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various type of Integer literals:

```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

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

Escape Sequence

When certain characters are preceded by a backslash, they will have a special meaning in Go. These are known as Escape Sequence codes which are used to represent newline (\n), tab (\t), backspace, etc. Here, you have a list of some of such escape sequence codes:

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

The following example shows how to use **\t** in a program:

```
package main


import "fmt"


func main() {
   fmt.Printf("Hello\tWorld!")
}
```

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

```
Hello   World!
```

String Literals in Go

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"


"hello, \


dear"


"hello, " "d" "ear"
```

The *const* Keyword

You can use **const** prefix to declare constants with a specific type as follows:

```
const variable type = value;
```

The following example shows how to use the **const** keyword:

```
package main

import "fmt"

func main() {
   const LENGTH int = 10
   const WIDTH int = 5
   var area int

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

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

```
value of area : 50
```

Note that it is a good programming practice to define constants in CAPITALS.

8. OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Go language is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Miscellaneous Operators

This tutorial explains arithmetic, relational, logical, bitwise, assignment, and other operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by Go language. Assume variable **A** holds 10 and variable **B** holds 20 then:

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. It increases the integer value by one.	A++ gives 11
--	Decrement operator. It decreases the integer value by one.	A-- gives 9

Example

Try the following example to understand all the arithmetic operators available in Go programming language:

```
package main

import "fmt"

func main() {

   var a int = 21
   var b int = 10
   var c int

   c = a + b
   fmt.Printf("Line 1 - Value of c is %d\n", c )
   c = a - b
   fmt.Printf("Line 2 - Value of c is %d\n", c )
   c = a * b
   fmt.Printf("Line 3 - Value of c is %d\n", c )
   c = a / b
   fmt.Printf("Line 4 - Value of c is %d\n", c )
   c = a % b
   fmt.Printf("Line 5 - Value of c is %d\n", c )
   a++
   fmt.Printf("Line 6 - Value of a is %d\n", a )
   a--
   fmt.Printf("Line 7 - Value of a is %d\n", a )
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of a is 22
Line 7 - Value of a is 21
```

Relational Operators

The following table lists all the relational operators supported by Go language. Assume variable **A** holds 10 and variable **B** holds 20, then:

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

Example

Try the following example to understand all the relational operators available in Go programming language:

```
package main


import "fmt"

```

```
func main() {
   var a int = 21
   var b int = 10

   if( a == b ) {
      fmt.Printf("Line 1 - a is equal to b\n" )
   } else {
      fmt.Printf("Line 1 - a is not equal to b\n" )
   }
   if ( a < b ) {
      fmt.Printf("Line 2 - a is less than b\n" )
   } else {
      fmt.Printf("Line 2 - a is not less than b\n" )
   }

   if ( a > b ) {
      fmt.Printf("Line 3 - a is greater than b\n" )
   } else {
      fmt.Printf("Line 3 - a is not greater than b\n" )
   }
   /* Lets change value of a and b */
   a = 5
   b = 20
   if ( a <= b ) {
      fmt.Printf("Line 4 - a is either less than or equal to  b\n" )
   }
   if ( b >= a ) {
      fmt.Printf("Line 5 - b is either greater than  or equal to b\n" )
   }
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to  b
Line 5 - b is either greater than  or equal to b
```

Logical Operators

The following table lists all the logical operators supported by Go language. Assume variable **A** holds 1 and variable **B** holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

The following table shows all the logical operators supported by Go language. Assume variable **A** holds true and variable **B** holds false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are false, then the condition becomes false.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is true, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Example

Try the following example to understand all the logical operators available in Go programming language:

```
package main

import "fmt"

func main() {
   var a bool = true
   var b bool = false
   if ( a && b ) {
      fmt.Printf("Line 1 - Condition is true\n" )
   }
   if ( a || b ) {
      fmt.Printf("Line 2 - Condition is true\n" )
   }
   /* lets change the value of  a and b */
   a = false
   b = true
   if ( a && b ) {
      fmt.Printf("Line 3 - Condition is true\n" )
   } else {
      fmt.Printf("Line 3 - Condition is not true\n" )
   }
   if ( !(a && b) ) {
      fmt.Printf("Line 4 - Condition is true\n" )
   }
}
```

When you compile and execute the above program, it produces the following result:

```
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation. 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

Assume A = 60; and B = 13. In binary format, they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

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

	number of bits specified by the right operand.	
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Example

Try the following example to understand all the bitwise operators available in Go programming language:

```
package main

import "fmt"

func main() {

   var a uint = 60     /* 60 = 0011 1100 */
   var b uint = 13     /* 13 = 0000 1101 */
   var c uint = 0

   c = a & b       /* 12 = 0000 1100 */
   fmt.Printf("Line 1 - Value of c is %d\n", c )

   c = a | b       /* 61 = 0011 1101 */
   fmt.Printf("Line 2 - Value of c is %d\n", c )

   c = a ^ b       /* 49 = 0011 0001 */
   fmt.Printf("Line 3 - Value of c is %d\n", c )

   c = a << 2     /* 240 = 1111 0000 */
   fmt.Printf("Line 4 - Value of c is %d\n", c )

   c = a >> 2     /* 15 = 0000 1111 */
   fmt.Printf("Line 5 - Value of c is %d\n", c )
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is 240

Line 5 - Value of c is 15
```

Assignment Operators

The following table lists all the assignment operators supported by Go language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns the value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Example

Try the following example to understand all the assignment operators available in Go programming language:

```
package main

import "fmt"

func main() {
   var a int = 21
   var c int

   c =  a
   fmt.Printf("Line 1 - =  Operator Example, Value of c = %d\n", c )

   c +=  a
   fmt.Printf("Line 2 - += Operator Example, Value of c = %d\n", c )

   c -=  a
   fmt.Printf("Line 3 - -= Operator Example, Value of c = %d\n", c )

   c *=  a
   fmt.Printf("Line 4 - *= Operator Example, Value of c = %d\n", c )

   c /=  a
   fmt.Printf("Line 5 - /= Operator Example, Value of c = %d\n", c )

   c  = 200;

   c <<=  2
   fmt.Printf("Line 6 - <<= Operator Example, Value of c = %d\n", c )

   c >>=  2
   fmt.Printf("Line 7 - >>= Operator Example, Value of c = %d\n", c )

   c &=  2
   fmt.Printf("Line 8 - &= Operator Example, Value of c = %d\n", c )

   c ^=  2
   fmt.Printf("Line 9 - ^= Operator Example, Value of c = %d\n", c )

   c |=  2
   fmt.Printf("Line 10 - |= Operator Example, Value of c = %d\n", c )

}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - =  Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - <<= Operator Example, Value of c = 800
Line 7 - >>= Operator Example, Value of c = 200
Line 8 - &= Operator Example, Value of c = 0
Line 9 - ^= Operator Example, Value of c = 2
Line 10 - |= Operator Example, Value of c = 2
```

Miscellaneous Operators

There are a few other important operators supported by Go Language including **sizeof** and **?:**.

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

Example

Try following example to understand all the miscellaneous operators available in Go programming language:

```
package main

import "fmt"

func main() {
   var a int = 4
   var b int32
   var c float32
   var ptr *int

   /* example of type operator */
   fmt.Printf("Line 1 - Type of variable a = %T\n", a );
   fmt.Printf("Line 2 - Type of variable b = %T\n", b );
   fmt.Printf("Line 3 - Type of variable c= %T\n", c );

   /* example of & and * operators */
   ptr = &a     /* 'ptr' now contains the address of 'a'*/
   fmt.Printf("value of a is  %d\n", a);
   fmt.Printf("*ptr is %d.\n", *ptr);
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Type of variable a = int
Line 2 - Type of variable b = int32
Line 3 - Type of variable c= float32
value of a is  4
*ptr is 4.
```

Operators Precedence in Go

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, and those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

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
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Example

Try the following example to understand the operator precedence available in Go programming language:

```
package main

import "fmt"

func main() {
   var a int = 20
   var b int = 10
   var c int = 15
   var d int = 5
   var e int;

   e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   fmt.Printf("Value of (a + b) * c / d is : %d\n",  e );

   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   fmt.Printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );

   e = (a + b) * (c / d);   // (30) * (15/5)
   fmt.Printf("Value of (a + b) * (c / d) is  : %d\n",  e );

   e = a + (b * c) / d;     //  20 + (150/5)
   fmt.Printf("Value of a + (b * c) / d is  : %d\n" ,  e );
}
```

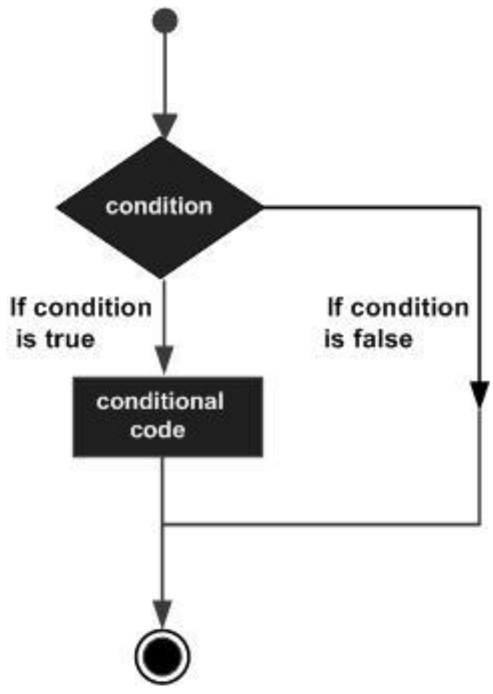
When you compile and execute the above program, it produces the following result:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50
```

9. DECISION MAKING

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



Go programming language provides the following types of decision making statements. Click the following links to check their detail.

Statement	Description
if statement	An if statement consists of a boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).

switch statement	A switch statement allows a variable to be tested for equality against a list of values.
select statement	A select statement is similar to switch statement with difference that case statements refers to channel communications.

The *if* Statement

An **if** statement consists of a boolean expression followed by one or more statements.

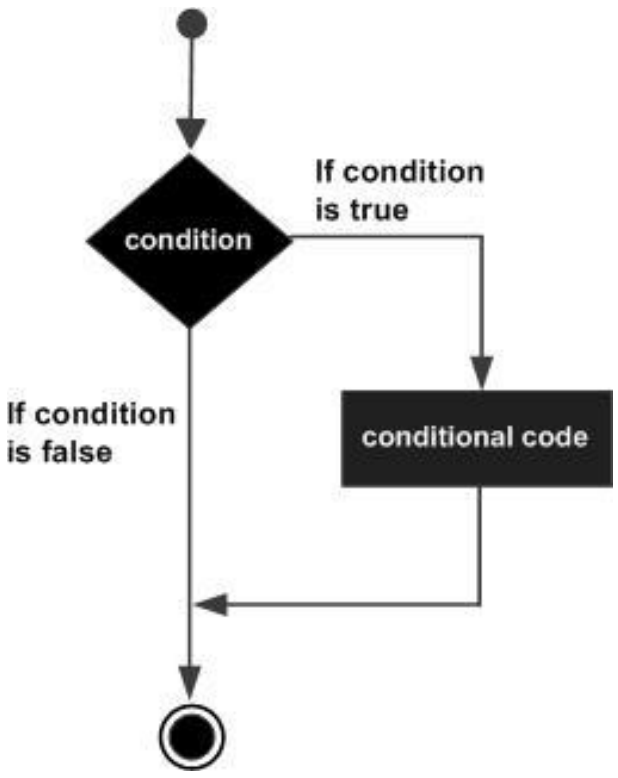
Syntax

The syntax of an **if** statement in Go programming language is:

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

If the boolean expression evaluates to **true**, then the block of code inside the if statement is executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) is executed.

Flow Diagram



Example

```
package main

import "fmt"

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

   /* 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)
}
```

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

```
a is less than 20;
value of a is : 10
```

The *if…else* Statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

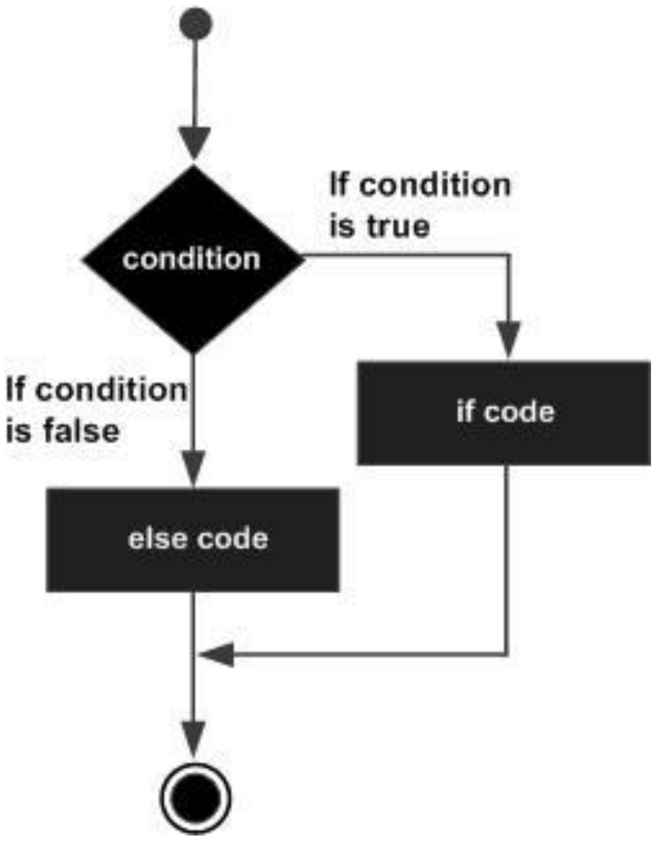
Syntax

The syntax of an **if...else** statement in Go programming language is:

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

If the boolean expression evaluates to **true**, then the **if block** of code is executed, otherwise **else block** of code is executed.

Flow Diagram



Example

```
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);

}
```

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

```
a is not less than 20;
value of a is : 100
```

Nested *if* Statement

It is always legal in Go programming to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows:

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

You can nest **else if...else** in the similar way as you have nested *if* statement.

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 );
}
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

The *Switch* Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

In Go programming, switch statements are of two types:

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

- **Type Switch** - In type switch, a case contain type which is compared against the type of a specially annotated switch expression.

Expression Switch

The syntax for **expression switch** statement in Go programming is as follows:

```
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);
}
```

The following rules apply to a **switch** statement:

- The **expression** used in a **switch** statement must have an integral or boolean expression, or be of a class type in which the class has a single conversion function to an integral or boolean value. If the expression is not passed, then the default value is true.

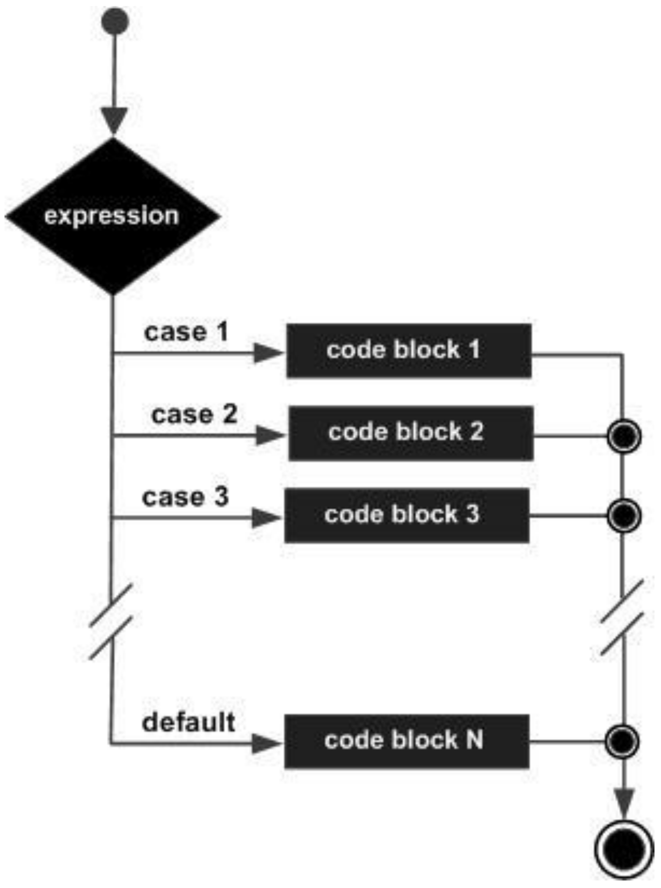
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

- When the variable being switched on is equal to a case, the statements following that case will execute. No **break** is needed in the case statement.

- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram



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 );
}
```

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

```
Well done
Excellent!
Your grade is  A
```

Type Switch

The syntax for a **type switch** statement in Go programming is as follows:

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

The following rules apply to a **switch** statement:

- The expression used in a switch statement must have an variable of interface{} type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The type for a case must be the same data type as the variable in the switch, and it must be a valid data type.
- When the variable being switched on is equal to a case, the statements following that case will execute. No break is needed in the case statement.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example

```
package main

import "fmt"

func main() {
   var x interface{}

   switch i := x.(type) {
      case nil:
         fmt.Printf("type of x :%T",i)
      case int:
         fmt.Printf("x is int")
      case float64:
         fmt.Printf("x is float64")
      case func(int) float64:
         fmt.Printf("x is func(int)")
      case bool, string:
         fmt.Printf("x is bool or string")
      default:
         fmt.Printf("don't know the type")
   }
}
```

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

```
type of x :<nil>
```

The *Select* Statement

The syntax for a **select** statement in Go programming language is as follows:

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

The following rules apply to a **select** statement:

- You can have any number of case statements within a select. Each case is followed by the value to be compared to and a colon.
- The **type** for a case must be the communication channel operation.
- When the channel operation occurs, the statements following that case is executed. No **break** is needed in the case statement.
- A **select** statement can have an optional **default** case, which must appear at the end of the select. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

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")
   }
}
```

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

```
no communication
```

The *if...else if...else* Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

While using **if...else if...else** statements, there are a few points to keep in mind:

- An **if** can have zero or one **else**'s and it must come after any **else if**'s.
- An **if** can have zero to many **else if**'s and they must come before the **else**.
- Once an **else if** succeeds, none of the remaining else **if**'s or **else**'s will be tested.

Syntax

The syntax of **if...else if...else** statement in Go programming language is:

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

Example

```
package main

import "fmt"

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

   /* check the boolean condition */
   if( a == 10 ) {
       /* if condition is true then print the following */
       fmt.Printf("Value of a is 10\n" )
   } else if( a == 20 ) {
       /* if else if condition is true */
       fmt.Printf("Value of a is 20\n" )
   } else if( a == 30 ) {
       /* if else if condition is true  */
       fmt.Printf("Value of a is 30\n" )
   } else {
       /* if none of the conditions is true */
       fmt.Printf("None of the values is matching\n" )
   }
   fmt.Printf("Exact value of a is: %d\n", a )
}
```

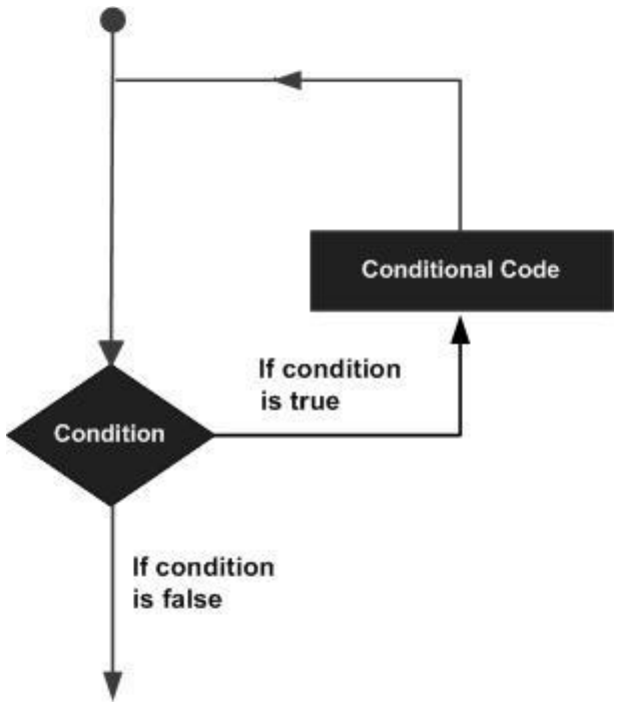
When the above code is compiled and executed, it produces the following result:

```
None of the values is matching
Exact value of a is: 100
```

10. LOOPS

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: the first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Go programming language provides the following types of loops to handle looping requirements.

Loop Type	Description
for loop	It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	These are one or multiple loops inside any for loop.

for Loop

A **for** loop is a repetition control structure. It allows you to write a loop that needs to execute a specific number of times.

Syntax

The syntax of **for** loop in Go programming language is:

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

The flow of control in **a for** loop is as follows:

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

- If a **for** clause that is **(init; condition; increment)** is present, then:

    - The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

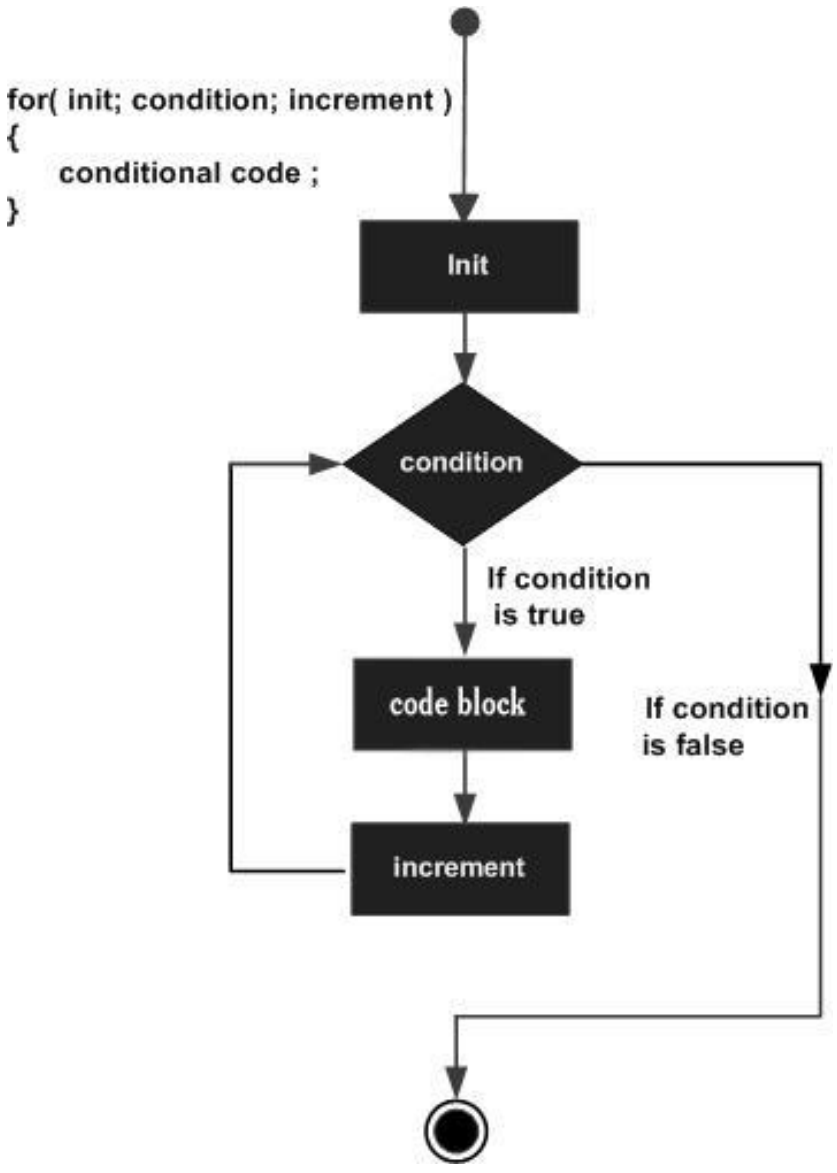
    - Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the **for** loop.

    - After the body of the **for** loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

    - The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again the condition). After the condition becomes false, the **for** loop terminates.

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

Flow Diagram



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)
   }
}
```

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

```
value of a: 0
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of x = 1 at 0
value of x = 2 at 1
value of x = 3 at 2
value of x = 5 at 3
value of x = 0 at 4
value of x = 0 at 5
```

Nested *for* Loops

Go programming language allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept:

Syntax

The syntax for a **nested for loop** statement in Go is as follows:

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

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
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);
      }
   }
}
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
```

```
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Loop Control Statements

Loop control statements change an execution from its normal sequence. When an execution leaves its scope, all automatic objects that were created in that scope are destroyed.

Go supports the following control statements:

Control Statement	Description
break statement	It terminates a **for loop** or **switch** statement and transfers execution to the statement immediately following the **for** loop or **switch**.
continue statement	It causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	It transfers control to the labeled statement.

The **break** statement in Go programming language has the following two usages:

1. When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

2. It can be used to terminate a case in a **switch** statement.

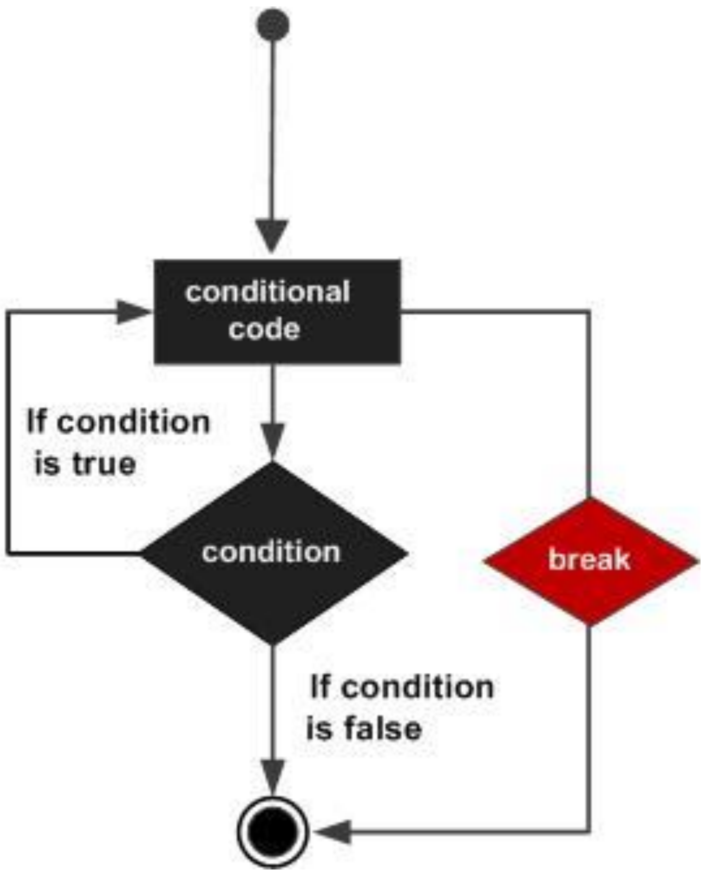
If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in Go is as follows:

```
break;
```

Flow Diagram



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;
      }
   }
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

The *continue* Statement

The **continue** statement in Go programming language works somewhat like a **break** statement. Instead of forcing termination, a **continue** statement forces the next iteration of the loop to take place, skipping any code in between.

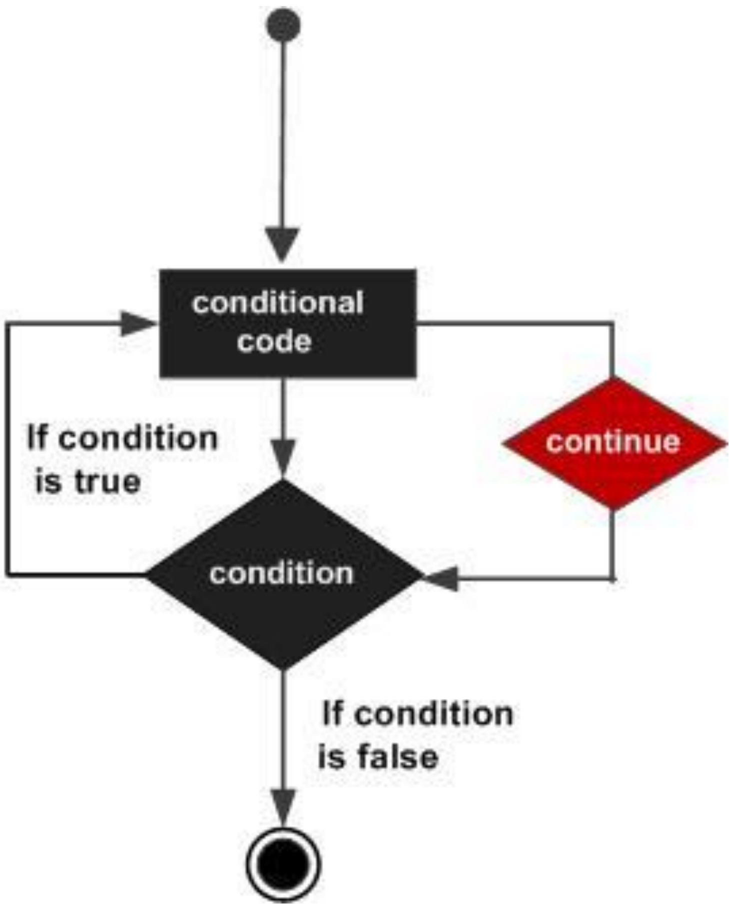
In case of the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute.

Syntax

The syntax for a **continue** statement in Go is as follows:

```
continue;
```

Flow Diagram



Example

```
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++;
   }
}
```

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

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

The *goto* Statement

A **goto** statement in Go programming language provides an unconditional jump from the goto to a labeled statement in the same function.

Note: Use of **goto** statement is highly discouraged in any programming language because it becomes difficult to trace the control flow of a program, making the program difficult to understand and hard to modify. Any program that uses a goto can be rewritten using some other construct.

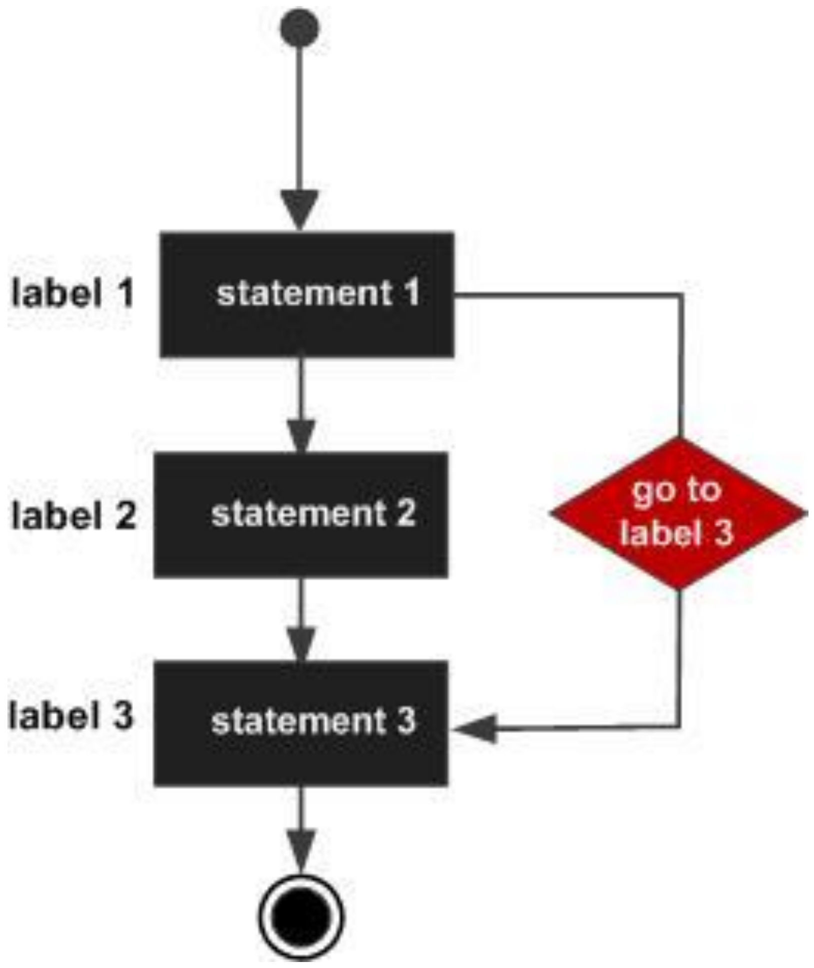
Syntax

The syntax for a **goto** statement in Go is as follows:

```
goto label;
..
.
label: statement;
```

Here, **label** can be any plain text except Go keyword and it can be set anywhere in the Go program above or below to **goto** statement.

Flow Diagram



Example

```
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++
    }
}
```

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

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

The Infinite Loop

A loop becomes an infinite loop if its condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty or by passing true to it.

```
package main

import "fmt"

func main() {
   for true  {
       fmt.Printf("This loop will run forever.\n");
   }
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the **for(;;)** construct to signify an infinite loop.

Note: You can terminate an infinite loop by pressing Ctrl + C keys.

11. FUNCTIONS

A function is a group of statements that together perform a task. Every Go program has at least one function, which is **main().** You can divide your code into separate functions. How you divide your code among different functions is up to you, but logically, the division should be such that each function performs a specific task.

A function **declaration** tells the compiler about a function name, return type, and parameters. A function **definition** provides the actual body of the function.

The Go standard library provides numerous built-in functions that your program can call. For example, the function **len()** takes arguments of various types and returns the length of the type. If a string is passed to it, the function returns the length of the string in bytes. If an array is passed to it, the function returns the length of the array.

Functions are also known as **method, sub-routine**, or **procedure**.

Defining a Function

The general form of a function definition in Go programming language is as follows:

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

A function definition in Go programming language consists of a *function header* and a *function body*. Here are all the parts of a function:

- **Func:** It starts the declaration of a function.

- **Function Name:** It is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Return Type:** A function may return a list of values. The return_types is the list of data types of the values the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the not required.

- **Function Body:** It contains a collection of statements that define what the function does.

Example

The following source code shows a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

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

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

Calling a Function

While creating a Go function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with its function name. If the function returns a value, then you can store the returned value. For example:

```
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
}
```

We have kept the max() function along with the main() function and compiled the source code. While running the final executable, it would produce the following result:

```
Max value is : 200
```

Returning Multiple Values from Function

A Go function can return multiple values. For example:

```
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)
}
```

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

```
Kumar Mahesh
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, Go uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. The above program, while calling the max() function, used the same method.

Call by Value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, Go programming language uses *call by value* method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows:

```
/* function definition to swap the values */
func swap(int x, int y) int {
   var temp int

   temp = x /* save the value of x */
   x = y    /* put y into x */
   y = temp /* put temp into y */

   return temp;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example:

```
package main

import "fmt"

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

   fmt.Printf("Before swap, value of a : %d\n", a )
   fmt.Printf("Before swap, value of b : %d\n", b )

   /* calling a function to swap the values */
   swap(a, b)

   fmt.Printf("After swap, value of a : %d\n", a )
   fmt.Printf("After swap, value of b : %d\n", b )
}
func swap(x, y int) 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;
}
```

Put the above code in a single C file, and then compile and execute it. It will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

Call by Reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. Accordingly, you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

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

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

```
package main

import "fmt"

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

   fmt.Printf("Before swap, value of a : %d\n", a )
   fmt.Printf("Before swap, value of b : %d\n", b )

   /* calling a function to swap the values.
   * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b)

   fmt.Printf("After swap, value of a : %d\n", a )
   fmt.Printf("After swap, value of b : %d\n", b )
}

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 */
}
```

Put the above code in a single C file, and then compile and execute it. It produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

Function Usage

A function can be used in the following ways:

Function Usage	Description
Function as Value	Functions can be created on the fly and can be used as values.
Function Closures	Functions closures are anonymous functions that can be used in dynamic programming.
Method	Methods are special functions with a receiver.

Go programming language provides the flexibility to create functions on the fly and use them as values. In the following example, we've initialized a variable with a function definition. Purpose of this function variable is just to use inbuilt math.sqrt() function. For example:

```
package main

import (
   "fmt"
   "math"
)

func main(){
   /* declare a function variable */
   getSquareRoot := func(x float64) float64 {
      return math.Sqrt(x)
   }

   /* use the function */
   fmt.Println(getSquareRoot(9))

}
```

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

```
3
```

Function Closures

Go programming language supports anonymous functions which can act as function closures. Anonymous functions are used when we want to define a function inline without passing any name to it.

In our example, we created a function getSequence() which returns another function. The purpose of this function is to close over a variable i of upper function to form a closure. For example:

```
package main

import "fmt"

func getSequence() func() int {
   i:=0
   return func() int {
      i+=1
       return i
   }
}

func main(){
   /* nextNumber is now a function with i as 0 */
   nextNumber := getSequence()

   /* invoke nextNumber to increase i by 1 and return the same */
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())

   /* create a new sequence and see the result, i is 0 again*/
   nextNumber1 := getSequence()
   fmt.Println(nextNumber1())
   fmt.Println(nextNumber1())
}
```

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

```
1
2
3
1
2
```

Method

Go programming language supports special types of functions called methods. In method declaration syntax, a "receiver" is present to represent the container of the function. This receiver can be used to call a function using "." operator. For example:

Syntax

```
func (variable_name variable_data_type) function_name() [return_type]{
   /* function body*/
}
package main

import (
   "fmt"
   "math"
)

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

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

func main(){
   circle := Circle(x:0, y:0, radius:5)
   fmt.Printf("Circle area: %f", circle.area())
}
```

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

```
Circle area: 78.539816
```

12. SCOPE RULES

A scope in any programming is a region of the program where a defined variable can exist and beyond that the variable cannot be accessed. There are three places where variables can be declared in Go programming language:

1. Inside a function or a block (**local** variables)
2. Outside of all functions (**global** variables)
3. In the definition of function parameters (**formal** parameters)

Let us find out what are **local** and **global** variables and what are **formal** parameters.

Local Variables

Variables that are declared inside a function or a block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example uses local variables. Here all the variables a, b, and c are local to the main() function.

```
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)
}
```

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

```
value of a = 10, b = 20 and c = 30
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. Global variables hold their value throughout the lifetime of the program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout the program after its declaration. The following example uses both global and local variables:

```
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)
}
```

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

```
value of a = 10, b = 20 and c = 30
```

A program can have the same name for local and global variables but the value of the local variable inside a function takes preference. For example:

```
package main

import "fmt"
```

```

/* global variable declaration */
var g int = 20

func main() {
   /* local variable declaration */
   var g int = 10

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

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

```
value of g = 10
```

Formal Parameters

Formal parameters are treated as local variables with-in that function and they take preference over the global variables. For example:

```
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;
}
```

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

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

Local and global variables are initialized to their default value, which is 0; while pointers are initialized to nil.

Data Type	Initial Default Value
int	0
float32	0
pointer	Nil

13. STRINGS

Strings, which are widely used in Go programming, are a read-only slice of bytes. In the Go programming language, strings are **slices**. The Go platform provides various libraries to manipulate strings:

- unicode
- regexp
- strings

Creating Strings

The most direct way to create a string is to write:

```
var greeting = "Hello world!"
```

Whenever it encounters a string literal in your code, the compiler creates a string object with its value. In this case, it is "Hello world!'.

A string literal holds valid UTF-8 sequences called **runes**. A string holds arbitrary bytes.

```
package main
import "fmt"
func main() {
   var greeting =  "Hello world!"

   fmt.Printf("normal string: ")
   fmt.Printf("%s", greeting)
   fmt.Printf("\n")
   fmt.Printf("hex bytes: ")
   for i := 0; i < len(greeting); i++ {
       fmt.Printf("%x ", greeting[i])
   }
   fmt.Printf("\n")

   const sampleText = "\xbd\xb2\x3d\xbc\x20\xe2\x8c\x98"
```

```
    /*q flag escapes unprintable characters, with + flag it escapes non-
    ascii characters as well to make output unambiguous  */

    fmt.Printf("quoted string: ")
   fmt.Printf("%+q", sampleText)
   fmt.Printf("\n")
}
```

It would produce the following result:

```
normal string: Hello world!
hex bytes: 48 65 6c 6c 6f 20 77 6f 72 6c 64 21
quoted string: "\xbd\xb2=\xbc \u2318"
```

Note: The string literal is immutable. Once it is created, a string literal cannot be changed.

String Length

len(str) method returns the number of bytes contained in a string literal.

```
package main
import "fmt"
func main() {
   var greeting =  "Hello world!"

   fmt.Printf("String Length is: ")
   fmt.Println(len(greeting))
}
```

It would produce the following result:

```
String Length is : 12
```

Concatenating Strings

The strings package includes a method **join** for concatenating multiple strings:

```
strings.Join(sample, " ")
```

Join concatenates the elements of an array to create a single string. Second parameter is a separator which is placed between the elements of the array.

Let us look at the following example:

```
package main
import (
 "fmt"
 "strings"
)
func main() {
   greetings :=  []string{"Hello","world!"}
   fmt.Println(strings.Join(greetings, " "))
}
```

It would produce the following result:

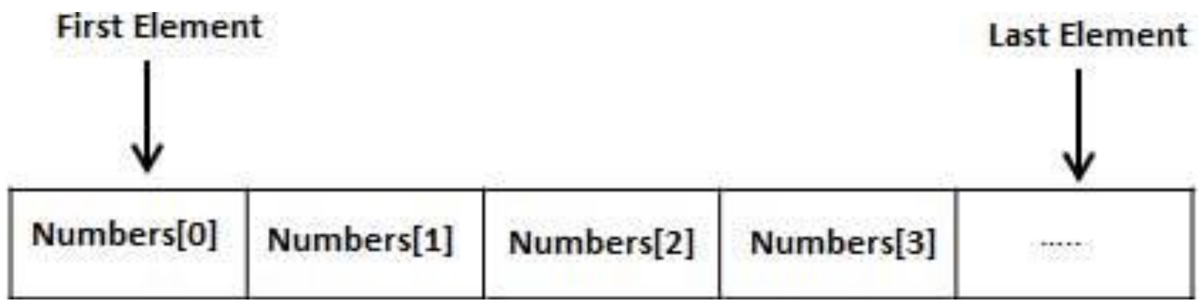
```
Hello world!
```

14. ARRAYS

Go programming language provides a data structure called the **array**, which can store a fixed-size sequential collection of elements of the same data type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in Go, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
var variable_name [SIZE] variable_type
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid Go data type. For example, to declare a 10-element array called **balance** of type float32, use this statement:

```
var balance [10] float32
```

Here, **balance** is a variable array that can hold up to 10 float numbers.

Initializing Arrays

You can initialize an array in Go, either one by one or using a single statement as follows:

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

The number of values between the braces { } cannot be larger than the number of elements that we declare for the array between the square brackets [ ].

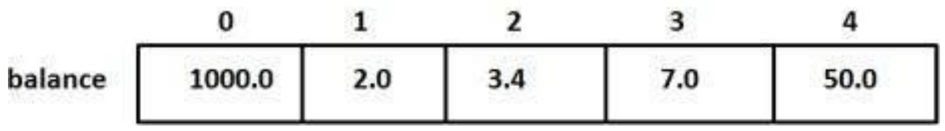
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

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

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

```
balance[4] = 50.0
```

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

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

The above statement will take the 10th element from the array and assign the value to salary variable. Following is an example which will use all the above-mentioned three concepts, viz. declaration, assignment, and accessing arrays:

```
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] )
   }
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Go Arrays in Detail

There are important concepts related to array which should be clear to a Go programmer:

Concept	Description
Multi-dimensional arrays	Go supports multidimensional arrays. The simplest form of a multidimensional array is the two-dimensional array.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.

Multidimensional Arrays in Go

Go programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type
```

For example, the following declaration creates a three-dimensional [5][10][4] integer array:

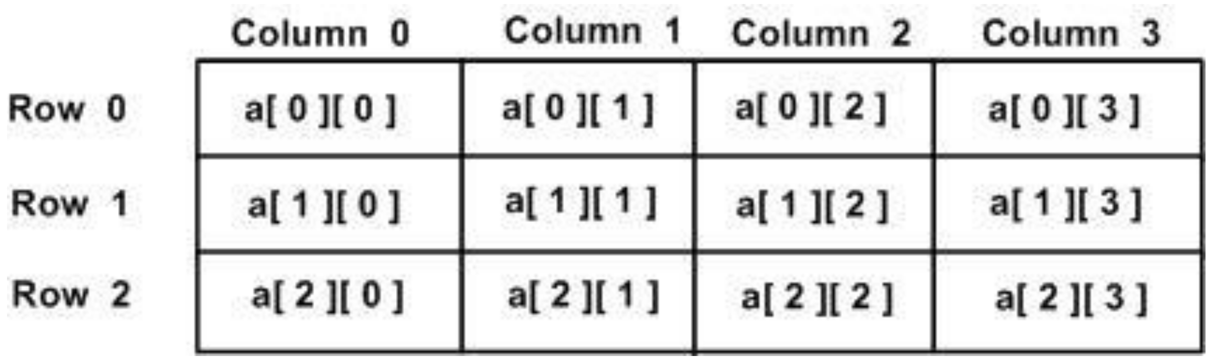
```
var threedim [5][10][4]int
```

Two-Dimensional Arrays

A two-dimensional array is the simplest form of a multidimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x, y], you would write something as follows:

```
var arrayName [ x ][ y ] variable_type
```

Where **variable_type** can be any valid Go data type and **arrayName** can be a valid Go identifier. A two-dimensional array can be taken as a table having **x** number of rows and **y** number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name as **a[i][j]**, where **a** is the name of the array, and **i** and **j** are the subscripts that uniquely identify each element in **a**.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. The following array is with 3 rows and each row has 4 columns.

```
a = [3][4]int{
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
}
```

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3]
```

The above statement takes the 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check the following program where we have used nested loop to handle a two-dimensional array:

```
package main

import "fmt"

func main() {
   /* an array with 5 rows and 2 columns*/
   var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}
   var i, j int

   /* output each array element's value */
   for  i = 0; i < 5; i++ {
      for j = 0; j < 2; j++ {
         fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )
      }
   }
}
```

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

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
```

```
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Passing Arrays to Functions

If you want to pass a one-dimensional array as an argument in a function, you would have to declare a function formal parameter in one of following two ways. Both the declaration methods produce similar results because each tells the compiler that an integer array is going to be received. Similarly, you can pass a multidimensional array as a formal parameter.

Way-1

Formal parameters as a sized array as follows:

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

Way-2

Formal parameters as an unsized array as follows:

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

Example

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return the average of the numbers passed through the array as follows:

```
func getAverage(arr []int, int size) float32
{
   var i int
   var avg, sum float32

   for i = 0; i < size; ++i {
      sum += arr[i]
   }

   avg = sum / size

   return avg;
}
```

Now, let us call the above function as follows:

```
package main

import "fmt"

func main() {
   /* an int array with 5 elements */
   var  balance = []int {1000, 2, 3, 17, 50}
   var avg float32

   /* pass array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value */
   fmt.Printf( "Average value is: %f ", avg );
}
func getAverage(arr []int, size int) float32 {
   var i,sum int
   var avg float32

   for i = 0; i < size;i++ {
      sum += arr[i]
   }

   avg = float32(sum / size)

   return avg;
}
```

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

```
Average value is: 214.400000
```

As you can see, the length of the array doesn't matter as far as the function is concerned because Go performs no bounds checking for formal parameters.

15. POINTERS

Pointers in Go are easy and fun to learn. Some Go programming tasks are performed more easily with pointers, and other tasks, such as call by reference, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect Go programmer.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

```
package main


import "fmt"


func main() {
   var a int = 10


   fmt.Printf("Address of a variable: %x\n", &a  )
}
```

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

```
Address of a variable: 10328000
```

So you understood what is memory address and how to access it. Now let us see what pointers are.

What Are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

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

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

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

The actual data type of the value of all pointers, whether integer, float, or otherwise., is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we frequently perform with pointers: (a) we define pointer variables, (b) assign the address of a variable to a pointer, and (c) access the value at the address stored in the pointer variable.

All these operations are carried out using the unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example demonstrates how to perform these operations:

```
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 )
}
```

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

```
Address of var variable: 10328000
Address stored in ip variable: 10328000
Value of *ip variable: 20
```

Nil Pointers in Go

Go compiler assign a nil value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned nil is called a **nil** pointer.

The nil pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
package main


import "fmt"


func main() {
   var  ptr *int


   fmt.Printf("The value of ptr is : %x\n", ptr  )
}
```

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

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

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

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

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

Go Pointers in Detail

Pointers have many but easy concepts and they are very important to Go programming. The following concepts of pointers should be clear to a Go programmer:

Concept	Description
Go – Array of pointers	You can define arrays to hold a number of pointers.
Go – Pointer to pointer	Go allows you to have pointer on a pointer and so on.
Passing pointers to functions in Go	Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.

Go – Array of Pointers

Before we understand the concept of arrays of pointers, let us consider the following example, which makes use of an array of 3 integers:

```
package main

import "fmt"

const MAX int = 3

func main() {

   a := []int{10,100,200}
   var i int

   for i = 0; i < MAX; i++ {
      fmt.Printf("Value of a[%d] = %d\n", i, a[i] )
   }
}
```

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

```
Value of a[0] = 10
Value of a[1] = 100
Value of a2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or string or any other data type available. The following statement declares an array of pointers to an integer:

```
var ptr [MAX]*int;
```

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. The following example makes use of three integers, which will be stored in an array of pointers as follows:

```
package main


import "fmt"

const MAX int = 3

func main() {

   a := []int{10,100,200}

   var i int

   var ptr [MAX]*int;


   for  i = 0; i < MAX; i++ {

      ptr[i] = &a[i] /* assign the address of integer. */

   }


   for  i = 0; i < MAX; i++ {

      fmt.Printf("Value of a[%d] = %d\n", i,*ptr[i] )

   }

}
```

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

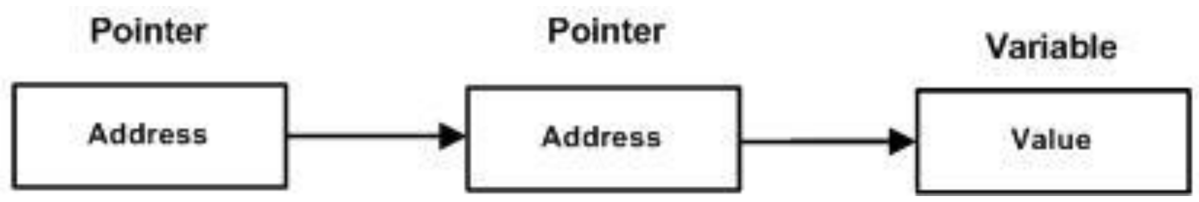
```
Value of a[0] = 10

Value of a[1] = 100

Value of a[2] = 200
```

Go – Pointer to Pointer

A pointer to a pointer is a form of chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following statement declares a pointer to a pointer of type int:

```
var ptr **int;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown in the following example:

```
package main

import "fmt"

func main() {

   var a int
   var ptr *int
   var pptr **int

   a = 3000

   /* take the address of var */
   ptr = &a

   /* take the address of ptr using address of operator & */
   pptr = &ptr

   /* take the value using pptr */
   fmt.Printf("Value of a = %d\n", a )
   fmt.Printf("Value available at *ptr = %d\n", *ptr )
   fmt.Printf("Value available at **pptr = %d\n", **pptr)
}
```

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

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Go – Passing Pointers to Functions

Go programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

In the following example, we pass two pointers to a function and change the value inside the function which reflects back in the calling function:

```
package main

import "fmt"

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

   fmt.Printf("Before swap, value of a : %d\n", a )
   fmt.Printf("Before swap, value of b : %d\n", b )

   /* calling a function to swap the values.
   * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   fmt.Printf("After swap, value of a : %d\n", a )
   fmt.Printf("After swap, value of b : %d\n", b )
}

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 */
}
```

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

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

16. STRUCTURES

Go arrays allow you to define variables that can hold several data items of the same kind. **Structure** is another user-defined data type available in Go programming, which allows you to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of the books in a library. You might want to track the following attributes of each book:

- Title
- Author
- Subject
- Book ID

In such a scenario, structures are highly useful.

Defining a Structure

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

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

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

```
Variable_name := structure_variable_type {value1, value2…valuen}
```

Accessing Structure Members

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

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* Declare Book1 of type Book */
   var Book2 Books        /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "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)
}
```

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

```
Book 1 title : Go Programming
Book 1 author : Mahesh Kumar
Book 1 subject : Go Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

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

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* Declare Book1 of type Book */
   var Book2 Books        /* Declare Book2 of type Book */
   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "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);
}
```

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

```
Book title : Go Programming
Book author : Mahesh Kumar
Book subject : Go Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable as follows:

```
var struct_pointer *Books
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the "&" operator before the structure name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the "." operator as follows:

```
struct_pointer.title;
```

Let us re-write the above example using structure pointer:

```
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);
}
```

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

```
Book title : Go Programming
Book author : Mahesh Kumar
Book subject : Go Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

17. SLICES

Go Slice is an abstraction over Go Array. Go Array allows you to define variables that can 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. Slices overcome this limitation. It provides many utility functions required on Array and is widely used in Go programming.

Defining a slice

To define a slice, you can declare it as an array without specifying its size. Alternatively, you can use **make** function to create 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() functions

A slice is an abstraction over array. It actually uses arrays as an underlying structure. The **len()** function returns the elements presents in the slice where **cap()** function returns the capacity of the slice (i.e., how many elements it can be accommodate). The following example explains the usage of 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)
}
```

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

```
len=3 cap=5 slice=[0 0 0]
```

Nil slice

If a slice is declared with no inputs, then by default, it is initialized as nil. Its length and capacity are zero. For example:

```
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)
}
```

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

```
len=0 cap=0 slice=[]
slice is nil
```

Subslicing

Slice allows lower-bound and upper bound to be specified to get its subslice using **[lower-bound:upper-bound]**. For example:

```
package main

import "fmt"

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

   /* print the original slice */
   fmt.Println("numbers ==", numbers)

   /* print the sub slice starting from index 1(included) to index
4(excluded)*/
   fmt.Println("numbers[1:4] ==", numbers[1:4])

   /* missing lower bound implies 0*/
   fmt.Println("numbers[:3] ==", numbers[:3])

   /* missing upper bound implies len(s)*/
   fmt.Println("numbers[4:] ==", numbers[4:])

   numbers1 := make([]int,0,5)
   printSlice(numbers1)

   /* print the sub slice starting from index 0(included) to index
2(excluded) */
   number2 := numbers[:2]
   printSlice(number2)

   /* print the sub slice starting from index 2(included) to index
5(excluded) */
   number3 := numbers[2:5]
   printSlice(number3)
}
```

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

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

```
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=0 cap=5 slice=[]
len=2 cap=9 slice=[0 1]
len=3 cap=7 slice=[2 3 4]
```

append() and copy() Functions

One can increase the capacity of a slice using the **append()** function. Using **copy()** function, the contents of a source slice are copied to a destination slice. For example:

```
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)
}
```

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

```
len=0 cap=0 slice=[]
len=1 cap=2 slice=[0]
len=2 cap=2 slice=[0 1]
len=5 cap=8 slice=[0 1 2 3 4]
len=5 cap=16 slice=[0 1 2 3 4]
```

18. RANGE

The **range** keyword is used in **for** loop to iterate over items of an array, slice, channel or map. With arrays and slices, it returns the index of the item as integer. With maps, it returns the key of the next key-value pair. Range either returns one value or two. If only one value is used on the left of a range expression, it is the 1st value in the following table.

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

The following paragraph shows how to use **range**:

```
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])
   }

   /* create a map*/
   coutryCapitalMap := 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)
   }
}
```

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

```
Slice item 0 is 0
Slice item 1 is 1
Slice item 2 is 2
Slice item 3 is 3
Slice item 4 is 4
Slice item 5 is 5
Slice item 6 is 6
Slice item 7 is 7
Slice item 8 is 8
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
```

19. MAPS

Go provides another important data type named **map** which maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

Defining a Map

You must use **make** function to create a map.

```
/* 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

The following example illustrates how to create and use a map:

```
package main

import "fmt"

func main {
   var coutryCapitalMap map[string]string
   /* create a map*/
   coutryCapitalMap = 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*/
   captial, 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")
   }
}
```

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

```
Capital of India is New Delhi
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of United States is not present
```

delete() Function

delete() function is used to delete an entry from a map. It requires the map and the corresponding key which is to be deleted. For example:

```
package main

import "fmt"

func main {
   /* create a map*/
   coutryCapitalMap := 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])
   }
}
```

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

```
Original Map
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Entry for France is deleted
Updated Map
Capital of India is New Delhi
Capital of Italy is Rome
Capital of Japan is Tokyo
```

20. RECURSION

Recursion is the process of repeating items in a self-similar way. The same concept applies in programming languages as well. If a program allows to call a function inside the same function, then it is called a recursive function call. Take a look at the following example:

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

func main() {
   recursion()
}
```

The Go programming language supports recursion. That is, it allows a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go on to become an infinite loop.

Examples of Recursion in Go

Recursive functions are very useful to solve many mathematical problems such as calculating factorial of a number, generating a Fibonacci series, etc.

Example 1: Calculating Factorial Using Recursion in Go

The following example calculates the factorial of a given number using a recursive function:

```
package main

import "fmt"

func factorial(i int) {
   if(i <= 1) {
      return 1
   }
   return i * factorial(i - 1)
}
```

```
func main {
    var i int = 15
    fmt.Printf("Factorial of %d is %d\n", i, factorial(i))
}
```

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

```
Factorial of 15 is 2004310016
```

Example 2: Fibonacci Series Using Recursion in Go

The following example shows how to generate a Fibonacci series of a given number using a recursive function:

```
package main

import "fmt"

func fibonaci(i int) {
   if(i == 0) {
      return 0
   }
   if(i == 1) {
      return 1
   }
   return fibonaci(i-1) + fibonaci(i-2)
}

func main() {
    var i int
    for i = 0; i < 10; i++ {
       fmt.Printf("%d\t%n", fibonaci(i))
    }
}
```

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

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

21. TYPE CASTING

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another using the **cast operator**. Its syntax is as follows:

```
type_name(expression)
```

Example

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating number operation.

```
package main

import "fmt"

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

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

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

```
Value of mean : 3.400000
```

22. INTERFACES

Go programming provides another data type called **interfaces** which represents a set of method signatures. The struct data type implements these interfaces to have method definitions for the method signature of the interfaces.

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))
}
```

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

```
Circle area: 78.539816
Rectangle area: 50.000000
```

23. ERROR HANDLING

Go programming provides a pretty simple error handling framework with inbuilt error interface type of the following declaration:

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

Functions normally return error as last return value. Use **errors.New** to construct a basic error message as following:

```
func Sqrt(value float64)(float64, error) {
   if(value < 0){
      return 0, errors.New("Math: negative number passed to Sqrt")
   }
   return math.Sqrt(value)
}
```

Use return value and error message.

```
result, err:= Sqrt(-1)

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

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)
}

func main() {
   result, err:= Sqrt(-1)

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

   result, err = Sqrt(9)

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

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

```
Math: negative number passed to Sqrt
3
```