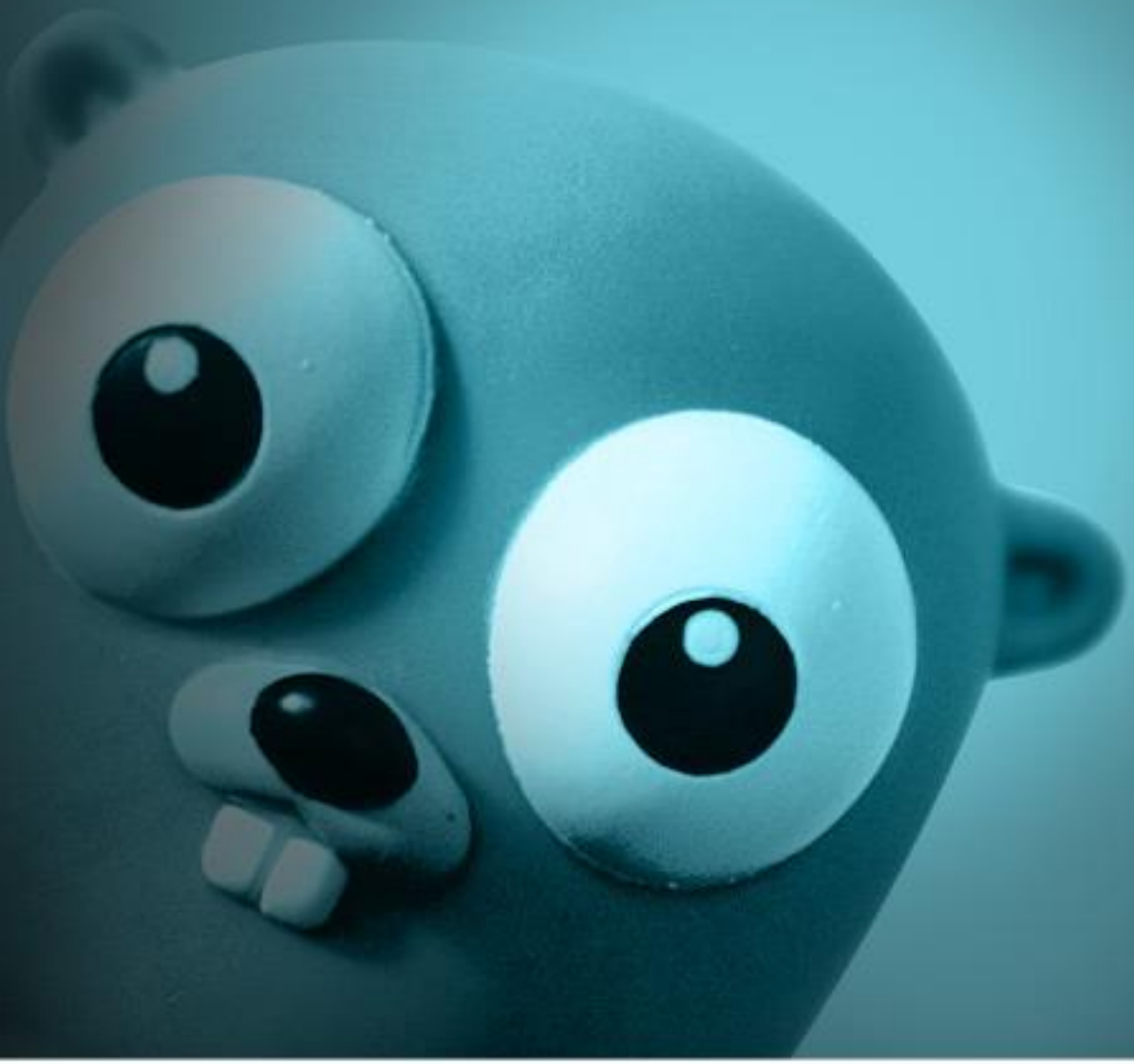
Go Programming

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. GO PROGRAMMING – 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. GO PROGRAMMING – 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. GO PROGRAMMING – 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. GO PROGRAMMING – 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. GO PROGRAMMING – 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. GO PROGRAMMING – 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. GO PROGRAMMING – 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. GO PROGRAMMING – 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 number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

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	C *= A is equivalent to C = C * A

	operand and assign the result to left operand	
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	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
```

End of ebook preview
If you liked what you saw…
Buy it from our store @ **https://store.tutorialspoint.com**