

Intro to Go Programming Language

By coding-bootcamps.com

Course Outline

1. [Golang Installation, Setup, GOPATH, and Go Workspace](#)

- Installations on Linux, Mac or Windows
- The Go tool
- GOPATH, Go Workspace, and Go Code Organization

2. [Hello Golang: Writing your First Golang Program](#)

3. [Golang Variables, Zero Values, and Type Inference](#)

- Introduction to variables and Data Types
- Declaring variables
- Zero values
- Declaring variables with initial value
- Type inference
- Short declaration

4. [Golang Basic Types, Operators and Type Conversion](#)

- Numeric Types
- Operations on Numeric Types
- Booleans
- Operations on Boolean Types
- Complex Numbers
- Operations on complex numbers
- Strings
- Type Conversion

5. [Working with Constants in Golang](#)

- Declaring a Constant
- Typed and Untyped Constants
- Constants and Type inference: Default Type
- Constant expressions
- Constant Expression Examples

6. [Golang Control Flow Statements: If, Switch and For](#)

- If Statement
- If-Else Statement
- If-Else-If chain
- If with a short statement
- Switch statement
- Switch with a short statement
- Combining multiple Switch cases
- Switch with no expression
- For Loop

7. [Introduction to Functions in Golang](#)

- Declaring and calling functions in Golang
- Functions with multiple return values
- Returning an error value from a function
- Functions with named return values
- Blank Identifier

8. [A Beginner Guide to Packages in Golang](#)

- Go Package
- The main package and main() function
- Importing Packages
- Creating and managing custom Packages
- Adding 3rd party Packages
- Manually installing Packages

9. [Working with Arrays in Golang](#)

- Declaring an Array in Golang
- Accessing array elements by their index
- Initializing an array using an array literal
- Letting Go compiler infer the length of the array
- Exploring more about Golang arrays
- Iterating over an array in Golang
- Iterating over an array using range
- Multidimensional arrays in Golang

10. [Introduction to Slices in Golang](#)

- Declaring a Slice
- Creating and Initializing a Slice
- Modifying a slice
- Length and capacity of a Slice
- Creating a slice using the built-in make() function
- Zero value of slices
- Slice functions
- Slice of slices
- Iterating over a slice

Session 1- Golang Installation, Setup, GOPATH, and Go Workspace

Go is an open source, statically typed, compiled programming language built by Google. It combines the best of both statically typed and dynamically typed languages and gives you the right mixture of efficiency and ease of programming. It is primarily suited for building fast, efficient, and reliable server side applications.

Following are some of the most noted features of Go:

- Safety : Both Type safety and Memory safety.
- Good support for Concurrency and communication.
- Efficient and latency-free Garbage Collection
- High speed compilation
- Excellent Tooling support

This is the first session of our class on Go. In this session, you'll learn how to install Go in your system and set up your development environment for Go projects.

Installing Go

Go binary distributions are available for all major operating systems like Linux, Windows, and MacOS. It's super simple to install Go from the binary distributions.

If a binary distribution is not available for your operating system, you can try [installing Go from source](#).

Mac OS X

Using Homebrew

The easiest way to install Go in Mac OS is by using [Homebrew](#) -

```
brew install go
```

Using macOS package installer

Download the latest Go package (.pkg) file from [Go's official downloads page](#). Open the package and follow the on-screen instructions to install Go. By default, Go will be installed in `/usr/local/go`.

Linux

Download the Linux distribution from [Go's official download page](#) and extract it into `/usr/local` directory.

```
sudo tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Next, add the `/usr/local/go/bin` directory to your `PATH` environment variable. You can do this by adding the following line to your `~/.bash_profile` file -

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

You can also use any other directory like `/opt/go` instead of `/usr/local` for installing Go.

Windows

Download the Windows MSI installer file from [Go's official download page](#). Open the installer and follow the on-screen instructions to install Go in your windows system. By default, the installer installs Go in `C:\Go`

The Go tool

The Go distribution comes bundled with the [go tool](#). It is a command line tool that lets you automate common tasks such as downloading and installing dependencies, building and testing your code, and much more.

After installing Go by following the instructions in the previous section, you should be able to run the *Go tool* by typing `go` in the command line -

```
$ go
```

Go is a tool for managing Go source code.

Usage:

```
go command [arguments]
```

The commands are:

build	compile packages and dependencies
clean	remove object files
doc	show documentation for package or symbol
env	print Go environment information
bug	start a bug report
fix	run go tool fix on packages
fmt	run gofmt on package sources
generate	generate Go files by processing source
get	download and install packages and dependencies
install	compile and install packages and dependencies
list	list packages
run	compile and run Go program
test	test packages
tool	run specified go tool
version	print Go version
vet	run go tool vet on packages

Use "go **help** [command]" **for more** information about a command.

Additional **help** topics:

c	calling between Go and C
---	--------------------------

buildmode description of build modes

filetype **file** types

gopath GOPATH environment variable

environment environment variables

importpath **import** path syntax

packages description of package lists

testflag description of testing flags

testfunc description of testing functions

Use "go help [topic]" for more information about that topic.

GOPATH, Go Workspace, and Go Code Organization

Go requires you to organize your code in a specific way.

By convention, all your Go code and the code you import, must reside in a single **workspace**. A workspace is nothing but a directory in your file system whose path is stored in the environment variable `GOPATH`.

Note that, after the introduction of Go modules in Go 1.11, you're no longer required to store your Go code in the Go workspace. You can create your Go project in any directory outside of `GOPATH`. The following explanation of Go workspace is given for historical reasons and the fact that it's still valid. You can skip this section if you want.

The **workspace** directory contains the following sub directories at its root:

- **src**: *contains Go source files.*

The **src** directory typically contains many version control repositories containing one or more Go packages. Every Go source file belongs to a package. You generally create a new subdirectory inside your repository for every separate Go package.

- **bin**: *contains the binary executables.*

The Go tool builds and installs binary executables to this directory. All Go programs that are meant to be executables must contain a source file with a

special package called `main` and define the entry point of the program in a special function called `main()`.

- `pkg`: contains Go package archives (`.a`).

All the non-executable packages (shared libraries) are stored in this directory. You cannot run these packages directly as they are not binary files. They are typically imported and used inside other executable packages.

Setting GOPATH

The GOPATH environment variable specifies the location of your workspace. By default, the GOPATH is assumed to be `$HOME/go` on Unix systems and `%USERPROFILE%\go` on Windows. If you're happy with this path then you don't need to do anything. You can just create your workspace directory named `go` inside the home folder and start writing Go code.

If you want to use a custom location as your workspace, you can set the GOPATH environment variable by following the instructions below -

Unix Systems (Linux and macOS)

For setting GOPATH in bash shells, add the following line to the `~/.bash_profile` file -

```
export GOPATH=$HOME/go
```

If you use **Zsh** shell, then you need to add the above line to `~/.zshrc` file.

Windows System

Let's say that you want to have your workspace directory at `C:\go-workspace`. Here is how you can set the GOPATH environment variable to use this workspace location:

- Create the workspace folder at `C:\go-workspace`.
- Right click on **Start** → click **Control Panel** → Select **System and Security** → click on **System**.
- From the menu on the left, select the **Advanced system settings**.

- Click the **Environment Variables** button at the bottom.
- Click **New** from the **User variables** section.
- Type `GOPATH` into the **Variable name** field.
- Type `C:\go-workspace` into the **Variable value** field.
- Click OK.

Note that, `GOPATH` must be different than the path of your Go installation.

Testing your Go installation with the Hello World program

First, make sure that you have created the Go workspace directory at `$HOME/go`. Next, create a new directory `src/hello` inside your workspace. Finally, create a file named `hello.go` with the following code:

Note: after Go 1.11 onwards, you can also create and run the program outside of Go workspace.

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello, World\n")
}

$ cd $HOME/go/src/hello
$ ls
hello.go
```

The easiest way to run the above program is using the `go run` command -

```
$ go run hello.go
```

```
Hello, World
```

Building an executable binary using `go build`

The `go run` command compiles and runs your program at one go. However, if you want to produce a binary from your Go source that can be run as a standalone executable without using the Go tool, then use the `go build` command -

```
$ cd $HOME/go/src/hello
```

```
$ go build
```

```
$ ls
```

```
hello hello.go
```

The `go build` command creates an executable binary with the same name as the name of your immediate package (`hello`). You can run the binary file like so -

```
$ ./hello
```

```
Hello, World
```

Installing the package into the `bin` directory using `go install`

You can use the `go install` command to build and install the executable binary into your workspace's `bin` directory -

```
$ cd $HOME/go/src/hello
```

```
$ go install
```

```
$ cd $HOME/go/bin
```

```
$ ls
```

```
hello
```

```
$ ./hello
```

```
Hello, World
```

You can also add the `$HOME/go/bin` directory to the `PATH` variable to run go executables from any location.

Don't forget to check out: `go help run`, `go help build`, `go help install`.

Session 2- Hello Golang: Writing your first Golang Program

When we start learning a new programming language, we typically start by writing the classic “Hello, World” program.

So Let's write the “Hello, World” program in Go and understand how it works. Open your favorite text editor, create a new file named `hello.go`, and type in the following code:

```
// My first Program
package main

import "fmt"

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

You can run the above program using `go run` command like so -

```
$ go run hello.go
```

```
Hello, World
```

The `go run` command does two things - It first compiles the program to machine code and then runs the compiled code.

If however, you want to produce a standalone binary executable from your Go source that can be run without using the Go tool, then use the `go build` command:

```
$ go build hello.go
```

```
$ ls
```

```
hello          hello.go
```

You may now run the built binary file like this -

```
$ ./hello
```

```
Hello, World
```

Understanding the internals of the “Hello, World” program

Let's go through each line of the “Hello, World” program one by one and understand what it does-

Line 1: The first line that starts with `//` is a comment -

```
// My first Program
```

Comments are ignored by the Go compiler. They are used to make it easier for others to understand your code.

Go supports two different styles of comments:

1. Single-line comment

```
2. // This is a Single line Comment. Everything in this line is ignored by the compiler
```

3. Multi-line comment

```
4. /*  
5.  This is a Multi line Comment.  
6.  As the name suggests, It can span multiple lines.  
7. */
```

Line 2: The second line is a package declaration:

```
package main
```

Every Go program starts with a package declaration. Packages are used to organize related go source code files into a single unit and make them reusable.

The package “main” is a special go package that is used with programs that are meant to be executable.

There are two types of programs in Go - Executable Programs and Libraries. Executable Programs are programs that can be run from the command line. Libraries are reusable pieces of code that are used by other programs to perform some task.

Line 3: The third line is an import statement:

```
import "fmt"
```

The import keyword is used to import reusable pieces of code from other packages to use in our program.

The “fmt” package contains code for dealing with I/O.

Line 4: The fourth line is a function declaration:

```
func main() {  
    // ...  
}
```

A function is a unit of code that contains one or more instructions to perform a task. We typically break a program into smaller functions that take some input, do some processing on the input, and produce an output.

A function in go is declared using the `func` keyword.

The `main()` function is the entry point of an executable program in Go. It is the first thing that is invoked when you run an executable program.

Line 5: This line contains a call to the `Println()` function of the `fmt` package:

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

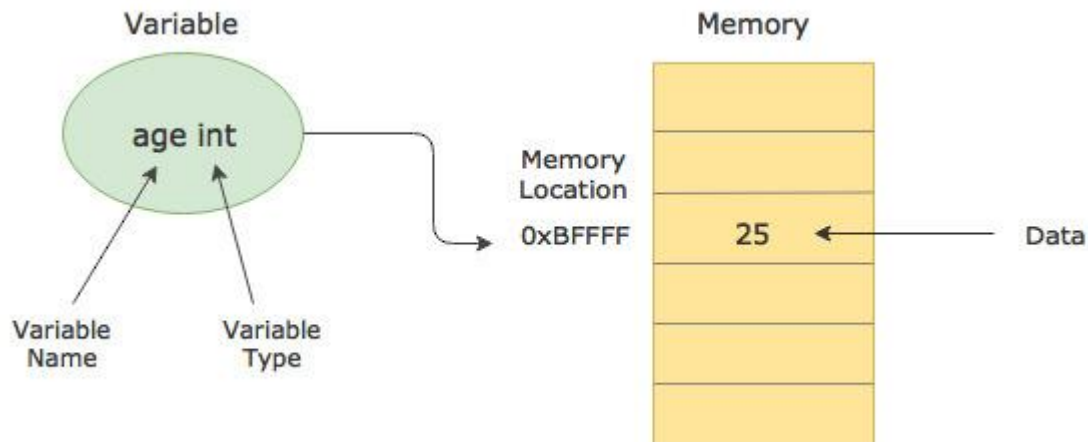
We pass the String “Hello, World” to the `Println()` function which prints it to the standard output along with a new line.

Session 3 Golang Variables, Zero Values, and Type Inference

Introduction to Variables and Data Types

Every program needs to store some data/information in memory. The data is stored in memory at a particular memory location.

A variable is just a convenient name given to a memory location where the data is stored. Apart from a name, every variable also has an associated type.



Data Types or simply Types, categorize related set of data, define the way they are stored, the range of values they can hold, and the operations that can be done on them.

For example, Golang has a data type called `int8`. It represents 8-bit integers whose values can range from -128 to 127. It also defines the operations that can be done on `int8` data type such as addition, subtraction, multiplication, division etc.

We also have an `int` data type in Golang whose size is machine dependent. It is 32 bits wide on a 32-bit system and 64 bits wide on a 64-bit system.

Other examples of data types in Golang are `bool`, `string`, `float32`, `float64` etc. You'll learn more about these data types in our other sessions. We gave a brief idea of data types here because it is necessary to understand them before we dive deep into Golang variables.

Golang Variables in depth

Declaring Variables

In Golang, We use the `var` keyword to declare variables -

```
var firstName string
var lastName string
```

```
var age int
```

You can also declare multiple variables at once like so -

```
var (  
    firstName string  
    lastName  string  
    age       int  
)
```

You can even combine multiple variable declarations of the same type with comma -

```
var (  
    firstName, lastName string  
    age           int  
)
```

Zero values

Any variable declared without an initial value will have a **zero-value** depending on the type of the variable-

Type	Zero Value
Bool	false
string	""
int, int8, int16 etc.	0

Type	Zero Value
float32, float64	0.0

The example below demonstrates the concept of zero values:

```
package main

import "fmt"

func main() {
    var (
        firstName, lastName string
        age                int
        salary              float64
        isConfirmed         bool
    )

    fmt.Printf("firstName: %s, lastName: %s, age: %d, salary: %f, isConfirmed: %t\n",
        firstName, lastName, age, salary, isConfirmed)
}
```

Output

```
firstName: , lastName: , age: 0, salary: 0.000000, isConfirmed: false
```

Declaring Variables with initial Value

Here is how you can initialize variables during declaration -

```
var firstName string = "Satoshi"
```

```
var lastName string = "Nakamoto"

var age int = 35
```

You can also use multiple declarations like this -

```
var (

    firstName string = "Satoshi"

    lastName string = "Nakamoto"

    age      int  = 35

)
```

Or even combine multiple variable declarations of the same type with comma and initialize them like so -

```
var (

    firstName, lastName string = "Satoshi", "Nakamoto"

    age int = 35

)
```

Type inference

Although Go is a statically typed language, It doesn't require you to explicitly specify the type of every variable you declare.

When you declare a variable with an initial value, Golang automatically infers the type of the variable from the value on the right-hand side. So you need not specify the type when you're initializing the variable at the time of declaration -

```
package main

import "fmt"

func main() {
```

```
var name = "Rajeev Singh" // Type declaration is optional here.

fmt.Printf("Variable 'name' is of type %T\n", name)
}

# Output

Variable 'name' is of type string
```

In the above example, Golang automatically infers the type of the variable as string from the value on the right-hand side. If you try to reassign the variable to a value of some other type, then the compiler will throw an error -

```
var name = "Raj johnson" // Type inferred as `string`

name = 1234 // Compiler Error
```

Type inference allows us to declare and initialize multiple variables of different data types in a single line like so -

```
package main

import "fmt"

func main() {

    // Multiple variable declarations with inferred types

    var firstName, lastName, age, salary = "John", "Maxwell", 28, 50000.0

    fmt.Printf("firstName: %T, lastName: %T, age: %T, salary: %T\n",
        firstName, lastName, age, salary)
}

# Output

firstName: string, lastName: string, age: int, salary: float64
```

Short Declaration

Go provides a short variable declaration syntax using `:=` operator. It is a shorthand for declaring and initializing a variable (with inferred type).

For example, the shorthand for `var name = "Raj"` is `name := "Raj"`. Here is a complete example -

```
package main

import "fmt"

func main() {

    // Short variable declaration syntax

    name := "Raj johnson"

    age, salary, isProgrammer := 35, 50000.0, true


    fmt.Println(name, age, salary, isProgrammer)

}

# Output

Raj johnson 35 50000 true
```

Note that, a Short variable declaration can only be used inside a function. Outside a function, every statement needs to begin with a keyword like `var`, `func` etc, and therefore, `:=` operator is not available.

Session 4 Golang Basic Types, Operators and Type Conversion

Go is a statically typed programming language. Every variable in Golang has an associated type.

Data types classify a related set of data. They define how the data is stored in memory, what are the possible values that a variable of a particular data type can hold, and the operations that can be done on them.

Golang has several built-in data types for representing common values like numbers, booleans, strings etc. In this session, We will look at all these basic data types one by one and understand how they work.

Numeric Types

Numeric types are used to represent numbers. They can be classified into Integers and Floating point types -

1. Integers

Integers are used to store whole numbers. Go has several built-in integer types of varying size for storing signed and unsigned integers -

Signed Integers

Type	Size	Range
int8	8 bits	-128 to 127
int16	16 bits	-2^{15} to $2^{15} - 1$
int32	32 bits	-2^{31} to $2^{31} - 1$
int64	64 bits	-2^{63} to $2^{63} - 1$
int	Platform dependent	Platform dependent

The size of the generic int type is platform dependent. It is 32 bits wide on a 32-bit system and 64-bits wide on a 64-bit system.

Unsigned Integers

Type	Size	Range
uint8	8 bits	0 to 255
uint16	16 bits	0 to $2^{16} - 1$
uint32	32 bits	0 to $2^{32} - 1$

Type	Size	Range
uint64	64 bits	0 to $2^{64} - 1$
Uint	Platform dependent	Platform dependent

The size of uint type is platform dependent. It is 32 bits wide on a 32-bit system and 64-bits wide on a 64-bit system.

When you are working with integer values, you should always use the int data type unless you have a good reason to use the sized or unsigned integer types.

In Golang, you can declare octal numbers using prefix 0 and hexadecimal numbers using the prefix 0x or 0X. Following is a complete example of integer types -

```
package main

import "fmt"

func main() {
    var myInt8 int8 = 97

    /*
        When you don't declare any type explicitly, the type inferred is `int`
        (The default type for integers)
    */

    var myInt = 1200

    var myUint uint = 500

    var myHexNumber = 0xFF // Use prefix '0x' or '0X' for declaring hexadecimal numbers
    var myOctalNumber = 034 // Use prefix '0' for declaring octal numbers
}
```

```
    fmt.Printf("%d, %d, %d, %#x, %#o\n", myInt8, myInt, myUInt, myHexNumber, myOctalNumber)
}
```

Output

97, 1200, 500, 0xff, 034

Integer Type aliases

Golang has two additional integer types called byte and rune that are aliases for uint8 and int32 data types respectively -

Type	Alias For
byte	uint8
rune	int32

In Go, the byte and rune data types are used to distinguish characters from integer values.

Golang doesn't have a char data type. It uses byte and rune to represent character values. The byte data type represents [ASCII](#) characters and the rune data type represents a more broader set of [Unicode](#) characters that are encoded in [UTF-8](#) format.

Characters are expressed in Golang by enclosing them in single quotes like this: 'A'.

The default type for character values is rune. That means, if you don't declare a type explicitly when declaring a variable with a character value, then Go will infer the type as rune -

```
var firstLetter = 'A' // Type inferred as `rune` (Default type for character values)
```

You can create a byte variable by explicitly specifying the type -

```
var lastLetter byte = 'Z'
```

Both byte and rune data types are essentially integers. For example, a byte variable with value 'a' is converted to the integer 97.

Similarly, a rune variable with a unicode value '♥' is converted to the corresponding unicode codepoint U+2665, where U+ means unicode and the numbers are hexadecimal, which is essentially an integer.

```
package main

import "fmt"

func main() {
    var myByte byte = 'a'
    var myRune rune = '♥'

    fmt.Printf("%c = %d and %c = %U\n", myByte, myByte, myRune, myRune)
}

# Output
a = 97 and ♥ = U+2665
```

In the above example, we have printed the variable myByte in character and decimal format, and the variable myRune in character and Unicode format.

2. Floating Point Types

Floating point types are used to store numbers with a decimal component (ex - 1.24, 4.50000). Go has two floating point types - float32 and float64.

- float32 occupies 32 bits in memory and stores values in single-precision floating point format.
- float64 occupies 64 bits in memory and stores values in double-precision floating point format.

The default type for floating point values is float64. So when you initialize a floating point variable with an initial value without specifying a type explicitly, the compiler will infer the type as float64 -

```
var a = 9715.635 // Type inferred as `float64` (the default type for floating-point numbers)
```

Operations on Numeric Types

Go provides several operators for performing operations on numeric types -

- Arithmetic Operators: +, -, *, /, %
- Comparison Operators: ==, !=, <, >, <=, >=
- Bitwise Operators: &, |, ^, <<, >>
- Increment and Decrement Operators: ++, --
- Assignment Operators: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=

Here is an example demonstrating some of the above operators -

```
package main

import (
    "fmt"
    "math"
)

func main() {
    var a, b = 4, 5

    var res1 = (a + b) * (a + b) / 2 // Arithmetic operations

    a++ // Increment a by 1

    b += 10 // Increment b by 10
```

```

var res2 = a ^ b // Bitwise XOR

var r = 3.5

var res3 = math.Pi * r * r // Operations on floating-point type

fmt.Printf("res1 : %v, res2 : %v, res3 : %v\n", res1, res2, res3)
}

# Output
res1 : 40, res2 : 10, res3 : 38.48451000647496

```

Booleans

Go provides a data type called `bool` to store boolean values. It can have two possible values - `true` and `false`.

```

var myBoolean = true

var anotherBoolean bool = false

```

Operations on Boolean Types

You can use the following operators on boolean types -

Logical Operators:

- `&&` (logical conjunction, “and”)
- `||` (logical disjunction, “or”)
- `!` (logical negation)

Equality and Inequality: `==`, `!=`

The operators `&&` and `||` follow short-circuiting rules. That means, in the expression `E1 && E2`, if `E1` evaluates to `false` then `E2` won’t be evaluated.

Similarly, in the expression `E1 || E2`, if `E1` evaluates to true then `E2` won't be evaluated.

Here is an example of Boolean types-

```
package main

import "fmt"

func main() {
    var truth = 3 <= 5
    var falsehood = 10 != 10

    // Short Circuiting
    var res1 = 10 > 20 && 5 == 5 // Second operand is not evaluated since first evaluates to false
    var res2 = 2*2 == 4 || 10%3 == 0 // Second operand is not evaluated since first evaluates to true

    fmt.Println(truth, falsehood, res1, res2)
}

# Output
true false false true
```

Complex Numbers

Complex numbers are one of the basic types in Golang. Go has two complex types of different sizes -

- `complex64`: both real and imaginary parts are of `float32` type.
- `complex128`: both real and imaginary parts are of `float64` type.

The default type for a complex number in golang is `complex128`. You can create a complex number like this -

```
var x = 5 + 7i // Type inferred as `complex128`
```

Go also provides a built-in function named `complex` for creating complex numbers. If you're creating a complex number with variables instead of literals, then you'll need to use the `complex` function -

```
var a = 3.57
var b = 6.23

// var c = a + bi won't work. Create the complex number like this -
var c = complex(a, b)
```

Note that, both real and imaginary parts of the complex number must be of the same floating point type. If you try to create a complex number with different real and imaginary part types, then the compiler will throw an error -

```
var a float32 = 4.92
var b float64 = 7.38

/*
The Following statement Won't compile.
(Both real and imaginary parts must be of the same floating-point type)
*/
var c = complex(a, b) // Compiler Error
```

Operations on complex numbers

You can perform arithmetic operations like addition, subtraction, multiplication, and division on complex numbers -

```
package main

import "fmt"
```

```

func main() {
    var a = 3 + 5i
    var b = 2 + 4i

    var res1 = a + b
    var res2 = a - b
    var res3 = a * b
    var res4 = a / b

    fmt.Println(res1, res2, res3, res4)
}

# Output
(5+9i) (1+1i) (-14+22i) (1.3-0.1i)

```

Strings

In Go, a string is a sequence of bytes.

Strings in Golang are declared either using double quotes as in "Hello World" or back ticks as in `Hello World` .

```

// Normal String (Can not contain newlines, and can have escape characters like `\\n`, `\\t` etc)
var name = "Steve Jobs"

// Raw String (Can span multiple lines. Escape characters are not interpreted)
var bio = `Steve Jobs was an American entrepreneur and inventor.

    He was the CEO and co-founder of Apple Inc.`

```

Double-quoted strings cannot contain newlines and they can have escape characters like `\n`, `\t` etc. In double-quoted strings, a `\n` character is replaced with a newline, and a `\t` character is replaced with a tab space, and so on.

Strings enclosed within back ticks are raw strings. They can span multiple lines. Moreover, Escape characters don't have any special meaning in raw strings.

```
package main

import "fmt"

func main() {
    var website = "\thttps://coding-bootcamps.com\t\n"

    var siteDescription = `\\tCoding Bootcamps is a programming school where you can find
        practical guides and tutorials on programming languages,
        web development, and desktop app development.\\t\n`

    fmt.Println(website, siteDescription)
}

# Output

https://coding-bootcamps.com

\\tCoding Bootcamps is a programming school where you can find
    practical guides and tutorials on programming languages,
    web development, and desktop app development.\\t\n
```

That's all about Strings in this session. But there is a lot more to learn about strings which include string indexing, handling unicode characters, performing various operations like string concatenation, split, join etc. We'll learn about them in a future session.

Type Conversion

Golang has a strong type system. It doesn't allow you to mix numeric types in an expression. For example, You cannot add an int variable to a float64 variable or even an int variable to an int64 variable. You cannot even perform an assignment between mixed types -

```
var a int64 = 4

var b int = a // Compiler Error (Cannot use a (type int64) as type int in assignment)

var c int = 500

var result = a + c // Compiler Error (Invalid Operation: mismatched types int64 and int)
```

Unlike other statically typed languages like C, C++, and Java, Go doesn't provide any implicit type conversion.

.

All right! So we cannot add, subtract, compare or perform any kind of operation on two different types even if they are numeric. But what to do if we need to perform such operations?

Well, you'll need to explicitly cast the variables to the target type -

```
var a int64 = 4

var b int = int(a) // Explicit Type Conversion

var c float64 = 6.5

// Explicit Type Conversion

var result = float64(b) + c // Works
```

The general syntax for converting a value v to a type T is T(v). Here are few more examples -

```
var myInt int = 65
```

```
var myUint uint = uint(myInt)
var myFloat float64 = float64(myInt)
```

Session 5- Working with Constants in Golang

Constants

In Golang, we use the term `constant` to represent fixed (unchanging) values such as 5, 1.34, true, "Hello" etc.

Literals are constants

All the literals in Golang, be it integer literals like 5, 1000, or floating-point literals like 4.76, 1.89, or boolean literals like true, false, or string literals like "Hello", "John" are **constants**.

Constants	Examples
integer constants	1000, 67413
floating-point constants	4.56, 128.372
boolean constants	true, false
rune constants	'C', 'ä'
complex constants	2.7i, 3 + 5i
string constants	"Hello", "Rajeev"

Declaring a Constant

Literals are constants without a name. To declare a constant and give it a name, you can use the `const` keyword like so -

```
const myFavLanguage = "Python"
const sunRisesInTheEast = true
```

You can also specify a type in the declaration like this -

```
const a int = 1234
const b string = "Hi"
```

Multiple declarations in a single statement is also possible -

```
const country, code = "India", 91

const (
    employeeId string = "E101"
    salary float64 = 50000.0
)
```

Constants, as you would expect, cannot be changed. That is, you cannot re-assign a constant to a different value after it is initialized -

```
const a = 123
a = 321 // Compiler Error (Cannot assign to constant)
```

Typed and Untyped Constants

Constants in golang are special. They work differently from how they work in other languages. To understand why they are special and how they exactly work, we need some background on Go's type system. So let's jump right into it -

Background

Go is a statically typed programming language. Which means that the type of every variable is known or inferred by the compiler at compile time.

But it goes a step further with its type system and doesn't even allow you to perform operations that mix numeric types. For example, You cannot add a float64 variable to an int, or even an int64 variable to an int-

```
var myFloat float64 = 21.54
var myInt int = 562
var myInt64 int64 = 120

var res1 = myFloat + myInt // Not Allowed (Compiler Error)
var res2 = myInt + myInt64 // Not Allowed (Compiler Error)
```

For the above operations to work, you'll need to explicitly cast the variables so that all of them are of the same type -

```
var res1 = myFloat + float64(myInt) // Works
var res2 = myInt + int(myInt64) // Works
```

If you've worked with other statically typed languages like C, C++ or Java, then you must be aware that they automatically convert smaller types to larger types whenever you mix them in any operation. For example, int can be automatically converted to long, float OR double.

So the obvious question is that - why doesn't Go do the same? why doesn't it perform implicit type conversions like C, C++ or Java?

And here is what Go designers have to say about this (Quoting from [Golang's official doc](#)) -

The convenience of automatic conversion between numeric types in C is outweighed by the confusion it causes. When is an expression unsigned? How big is the value? Does it overflow? Is the result portable, independent of the machine on which it executes? It also complicates the compiler; "the usual arithmetic conversions" are not easy to implement and inconsistent across

architectures. For reasons of portability, we decided to make things clear and straightforward at the cost of some explicit conversions in the code. ([Excerpt from Golang's official doc](#))

All right! So Go doesn't provide implicit type conversions and it requires us to do explicit type casting whenever we mix variables of multiple types in an operation.

But how does Go's type system work with constants? Given that all of the following statements are valid in Golang -

```
var myInt32 int32 = 10
var myInt int = 10
var myFloat64 float64 = 10
var myComplex complex64 = 10
```

What is the type of the constant value 10 in the above examples? Moreover, if there are no implicit type conversions in Golang, then wouldn't we need to write the above statements like -

```
var myInt32 int32 = int32(10)
var myFloat64 float64 = float64(10)
// etc..
```

Well, the answers to all these questions lay in the way constants are handled in Golang. So let's find out how they are handled.

Untyped Constants

Any constant in golang, named or unnamed, is untyped unless given a type explicitly. For example, all of the following constants are untyped -

```
1 // untyped integer constant
4.5 // untyped floating-point constant
true // untyped boolean constant
```

```
"Hello" // untyped string constant
```

They are untyped even after you give them a name -

```
const a = 1  
const f = 4.5  
const b = true  
const s = "Hello"
```

Now, you might be wondering that we will be using terms like integer constant, string constant, and we are also saying that they are untyped.

Well yes, the value 1 is an integer, 4.5 is a float, and "Hello" is a string. But they are just values. They are not given a **fixed** type yet, like `int32` or `float64` or `string`, that would force them to obey Go's strict type rules.

The fact that the value 1 is untyped allows us to assign it to any variable whose type is **compatible** with integers -

```
var myInt int = 1  
var myFloat float64 = 1  
var myComplex complex64 = 1
```

Note that, although the value 1 is untyped, it is an untyped integer. So it can only be used where an integer is allowed. You cannot assign it to a string or a boolean variable for example.

Similarly, an untyped floating-point constant like 4.5 can be used anywhere a floating-point value is allowed -

```
var myFloat32 float32 = 4.5  
var myComplex64 complex64 = 4.5
```

Let's now see an example of an untyped string constant-

In Golang, you can create a type alias using the `type` keyword like so-

```
type RichString string // Type alias of `string`
```

Given the strongly typed nature of Golang, you can't assign a `string` variable to a `RichString` variable-

```
var myString string = "Hello"  
var myRichString RichString = myString // Won't work.
```

But, you can assign an untyped string constant to a `RichString` variable because it is compatible with strings -

```
const myUntypedString = "Hello"  
var myRichString RichString = myUntypedString // Works
```

Constants and Type inference: Default Type

Go supports type inference. That is, it can infer the type of a variable from the value that is used to initialize it. So you can declare a variable with an initial value, but without any type information, and Go will automatically determine the type -

```
var a = 5 // Go compiler automatically infers the type of the variable `a`
```

But how does it work? Given that constants in Golang are untyped, what will be the type of the variable `a` in the above example? Will it be `int8` OR `int16` OR `int32` OR `int64` OR `int`?

Well, it turns out that every untyped constant in Golang has a **default type**. The default type is used when we assign the constant to a variable that doesn't have any explicit type available.

Following are the default types for various constants in Golang -

Constants	Default Type
integers (10, 76)	int
floats (3.14, 7.92)	float64
complex numbers (3+5i)	complex128
characters ('a', '♠')	rune
booleans (true, false)	bool
strings ("Hello")	string

So, in the statement `var a = 5`, since no explicit type information is available, the default type for integer constants is used to determine the type of `a`, which is `int`.

Typed Constants

In Golang, Constants are typed when you **explicitly** specify the type in the declaration like this-

```
const typedInt int = 1 // Typed constant
```

Just like variables, all the rules of Go's type system applies to typed constant. For example, you cannot assign a typed integer constant to a float variable -

```
var myFloat64 float64 = typedInt // Compiler Error
```

With typed constants, you lose all the flexibility that comes with untyped constants like assigning them to any variable of compatible type or mixing them in mathematical operations. So you should declare a type for a constant only if it's absolutely necessary. Otherwise, just declare constants without a type.

Constant expressions

The fact that constants are untyped (unless given a type explicitly) allows you to mix them in any expression freely.

So you can have a constant expression containing a mix of various untyped constants as long as those untyped constants are compatible with each other -

```
const a = 5 + 7.5 // Valid
```

```
const b = 12/5 // Valid
```

```
const c = 'z' + 1 // Valid
```

```
const d = "Hey" + true // Invalid (untyped string constant and untyped boolean constant are not compatible with each other)
```

The evaluation of constant expressions and their result follows certain rules. Let's look at those rules -

Rules for constant expressions

- A comparison operation between two untyped constants always outputs an untyped boolean constant.

```
const a = 7.5 > 5 // true (untyped boolean constant)
```

```
const b = "xyz" < "uvw" // false (untyped boolean constant)
```

- For any other operation (except shift) -
 - If both the operands are of the same type (ex - both are untyped integer constants), the result is also of the same type. For example, the expression $25/2$ yields 12 not 12.5. Since both the operands are untyped integers, the result is truncated to an integer.
 - If the operands are of different type, the result is of the operand's type that is broader as per the rule: integer < rune < floating-point < complex.

```
const a = 25/2 // 12 (untyped integer constant)
```

```
const b = (6+8i)/2 // (3+4i) (untyped complex constant)
```

- Shift operation rules are a bit complex. First of all, there are some requirements -
 - The right operand of a shift expression must either have an unsigned integer type or be an untyped constant that can represent a value of type `uint`.
 - The left operand must either have an integer type or be an untyped constant that can represent a value of type `int`.

The rule - If the left operand of a shift expression is an untyped constant, the result is an untyped integer constant; otherwise the result is of the same type as the left operand.

```
const a = 1 << 5      // 32 (untyped integer constant)

const b = int32(1) << 4 // 16 (int32)

const c = 16.0 >> 2    // 4 (untyped integer constant) - 16.0 can represent a value of type `int`

const d = 32 >> 3.0    // 4 (untyped integer constant) - 3.0 can represent a value of type `uint`

const e = 10.50 << 2   // ILLEGAL (10.50 can't represent a value of type `int`)

const f = 64 >> -2     // ILLEGAL (The right operand must be an unsigned int or an untyped constant compatible with `uint`)
```

Constant Expression Examples

Let's see some examples of constant expressions -

```
package main

import "fmt"

func main() {
    var result = 25/2

    fmt.Printf("result is %v which is of type %T\n", result, result)
```



```
}
```

```
# Output
```

```
result is 12 which is of type int
```

Since both 25 and 2 are untyped integer constants, the result is truncated to an untyped integer 12.

To get the correct result, you can do one of the following:

```
// Use a float value in numerator or denominator
```

```
var result = 25.0/2
```

```
// Explicitly cast the numerator or the denominator
```

```
var result = float64(25)/2
```

Let's see another example -

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var result = 4.5 + (10 - 5) * (3 + 2)/2
```

```
    fmt.Println(result)
```

```
}
```

What will be the result of the above program?

Well, it's not 17. The actual result of the above program is 16.5. Let's go through the evaluation order of the expression to understand why the result is 16.5

```
4.5 + (10 - 5) * (3 + 2)/2
```

↓

```
4.5 + (5) * (3 + 2)/2
```

↓
4.5 + (5) * (5)/2
↓
4.5 + (25)/2
↓
4.5 + 12
↓
16.5

You got it right? The result is wrong because the expression 25/2 is evaluated to 12.

To get the correct result, you can do one of the following:

```
// Use a float value in the numerator or denominator  
var result = 4.5 + (10 - 5) * (3 + 2)/2.0  
  
// Explicitly cast numerator or the denominator  
var result = 4.5 + float64((10 - 5) * (3 + 2))/2
```

Session 6- Golang Control Flow Statements: If, Switch and For

If Statement

If statements are used to specify whether a block of code should be executed or not depending on a given condition.

Following is the syntax of if statements in Golang -

```
if(condition) {  
    // Code to be executed if the condition is true.  
}
```

Here is a simple example -

```
package main  
  
import "fmt"  
  
func main() {  
    var x = 25  
    if(x % 5 == 0) {  
        fmt.Printf("%d is a multiple of 5\n", x)  
    }  
}  
  
# Output  
25 is a multiple of 5
```

Note that, You can omit the parentheses () from an if statement in Golang, but the curly braces {} are mandatory -

```
var y = -1  
if y < 0 {  
    fmt.Printf("%d is negative\n", y)  
}
```

You can combine multiple conditions using short circuit operators && and || like so -

```
var age = 21
```

```
if age >= 17 && age <= 30 {  
    fmt.Println("My Age is between 17 and 30")  
}
```

If-Else Statement

An if statement can be combined with an else block. The else block is executed if the condition specified in the if statement is false -

```
if condition {  
    // code to be executed if the condition is true  
} else {  
    // code to be executed if the condition is false  
}
```

Here is a simple example -

```
package main  
  
import "fmt"  
  
func main() {  
    var age = 18  
    if age >= 18 {  
        fmt.Println("You're eligible to vote!")  
    } else {  
        fmt.Println("You're not eligible to vote!")  
    }  
}  
  
# Output
```

You're eligible to vote!

If-Else-If Chain

if statements can also have multiple else if parts making a chain of conditions like this -

```
package main

import "fmt"

func main() {
    var BMI = 21.0
    if BMI < 18.5 {
        fmt.Println("You are underweight");
    } else if BMI >= 18.5 && BMI < 25.0 {
        fmt.Println("Your weight is normal");
    } else if BMI >= 25.0 && BMI < 30.0 {
        fmt.Println("You're overweight")
    } else {
        fmt.Println("You're obese")
    }
}
```

Output

Your weight is normal

If with a short statement

An if statement in Golang can also contain a short declaration statement preceding the conditional expression -

```
if n := 10; n%2 == 0 {
    fmt.Printf("%d is even\n", n)
}
```

The variable declared in the short statement is only available inside the if block and it's else or else-if branches -

```
if n := 15; n%2 == 0 {
    fmt.Printf("%d is even\n", n)
} else {
    fmt.Printf("%d is odd\n", n)
}
```

Note that, If you're using a short statement, then you can't use parentheses. So the following code will generate a syntax error -

```
// You can't use parentheses when `if` contains a short statement
if (n := 15; n%2 == 0) { // Syntax Error

}
```

Switch Statement

A Switch statement takes an expression and matches it against a list of possible cases. Once a match is found, it executes the block of code specified in the matched case.

Here is a simple example of switch statement -

```
package main

import "fmt"
```

```

func main() {
    var dayOfWeek = 6
    switch dayOfWeek {
        case 1: fmt.Println("Monday")
        case 2: fmt.Println("Tuesday")
        case 3: fmt.Println("Wednesday")
        case 4: fmt.Println("Thursday")
        case 5: fmt.Println("Friday")
        case 6: {
            fmt.Println("Saturday")
            fmt.Println("Weekend. Yaay!")
        }
        case 7: {
            fmt.Println("Sunday")
            fmt.Println("Weekend. Yaay!")
        }
        default: fmt.Println("Invalid day")
    }
}

```

Output

Saturday

Weekend. Yaay!

Go evaluates all the switch cases one by one from top to bottom until a case succeeds. Once a case succeeds, it runs the block of code specified in that case and then stops (it doesn't evaluate any further cases).

This is contrary to other languages like C, C++, and Java, where you explicitly need to insert a `break` statement after the body of every case to stop the evaluation of cases that follow.

If none of the cases succeed, then the default case is executed.

Switch with a short statement

Just like if, switch can also contain a short declaration statement preceding the conditional expression. So you could also write the previous switch example like this -

```
switch dayOfWeek := 6; dayOfWeek {  
    case 1: fmt.Println("Monday")  
    case 2: fmt.Println("Tuesday")  
    case 3: fmt.Println("Wednesday")  
    case 4: fmt.Println("Thursday")  
    case 5: fmt.Println("Friday")  
    case 6: {  
        fmt.Println("Saturday")  
        fmt.Println("Weekend. Yaay!")  
    }  
    case 7: {  
        fmt.Println("Sunday")  
        fmt.Println("Weekend. Yaay!")  
    }  
    default: fmt.Println("Invalid day")  
}
```

The only difference is that the variable declared by the short statement (dayOfWeek) is only available inside the switch block.

Combining multiple Switch cases

You can combine multiple switch cases into one like so -


```

package main

import "fmt"

func main() {
    switch dayOfWeek := 5; dayOfWeek {
        case 1, 2, 3, 4, 5:
            fmt.Println("Weekday")
        case 6, 7:
            fmt.Println("Weekend")
        default:
            fmt.Println("Invalid Day")
    }
}

# Output
Weekday

```

This comes handy when you need to run a common logic for multiple cases.

Switch with no expression

In Golang, the expression that we specify in the `switch` statement is optional. A `switch` statement without an expression is same as `switch true`. It evaluates all the cases one by one, and runs the first case that evaluates to true -

```

package main

import "fmt"

func main() {
    var BMI = 21.0

    switch {

```

```

    case BMI < 18.5:
        fmt.Println("You're underweight")

    case BMI >= 18.5 && BMI < 25.0:
        fmt.Println("Your weight is normal")

    case BMI >= 25.0 && BMI < 30.0:
        fmt.Println("You're overweight")

    default:
        fmt.Println("You're obese")

}

```

Switch without an expression is simply a concise way of writing if-else-if chains.

For Loop

A loop is used to run a block of code repeatedly. Golang has only one looping statement - the for loop.

Following is the generic syntax of for loop in Go -

```

for initialization; condition; increment {

    // loop body

}

```

The **initialization** statement is executed exactly once before the first iteration of the loop. In each iteration, the **condition** is checked. If the condition evaluates to true then the body of the loop is executed, otherwise, the loop terminates. The **increment** statement is executed at the end of every iteration.

Here is a simple example of a for loop -

```

package main

import "fmt"

```

```
func main() {
    for i := 0; i < 10; i++ {
        fmt.Printf("%d ", i)
    }
}
```

Output

0 1 2 3 4 5 6 7 8 9

Unlike other languages like C, C++, and Java, Go's for loop doesn't contain parentheses, and the curly braces are mandatory.

Note that, both initialization and increment statements in the for loop are optional and can be omitted

- Omitting the initialization statement

```
• package main
• import "fmt"
•
• func main() {
•     i := 2
•     for ; i <= 10; i += 2 {
•         fmt.Printf("%d ", i)
•     }
• }
• # Output
• 2 4 6 8 10
```

- Omitting the increment statement

- `package main`
- `import "fmt"`
-
- `func main() {`
- `i := 2`
- `for ;i <= 20; {`
- `fmt.Printf("%d ", i)`
- `i *= 2`
- `}`
- `}`
- # Output
- `2 4 8 16`

Note that, you can also omit the semicolons from the `for` loop in the above example and write it like this -

```
package main

import "fmt"

func main() {
    i := 2
    for i <= 20 {
        fmt.Printf("%d ", i)
        i *= 2
    }
}
```

The above for loop is similar to a while loop in other languages. Go doesn't have a while loop because we can easily represent a while loop using for.

Finally, You can also omit the condition from the for loop in Golang. This will give you an infinite loop -

```
package main

func main() {
    // Infinite Loop
    for {
    }
}
```

break statement

You can use break statement to break out of a loop before its normal termination. Here is an example -

```
package main

import "fmt"

func main() {
    for num := 1; num <= 100; num++ {
        if num%3 == 0 && num%5 == 0 {
            fmt.Printf("First positive number divisible by both 3 and 5 is %d\n", num)
            break
        }
    }
}

# Output
```

First positive number divisible by both 3 and 5 is 15

continue statement

The `continue` statement is used to stop running the loop body midway and continue to the next iteration of the loop.

```
package main

import "fmt"

func main() {
    for num := 1; num <= 10; num++ {
        if num%2 == 0 {
            continue;
        }
        fmt.Printf("%d ", num)
    }
}
```

Output

1 3 5 7 9

Session 7- Introduction to Functions in Golang

A function is a block of code that takes some input(s), does some processing on the input(s) and produces some output(s).



Functions help you divide your program into small reusable pieces of code. They improve the readability, maintainability, and testability of your program.

Declaring and Calling Functions in Golang

In Golang, we declare a function using the `func` keyword. A function has a **name**, a list of comma-separated **input parameters** along with their types, the **result type(s)**, and a **body**.

Following is an example of a simple function called `avg` that takes two input parameters of type `float64` and returns the average of the inputs. The result is also of type `float64` -

```
func avg(x float64, y float64) float64 {  
    return (x + y) / 2  
}
```

Now, calling a function is very simple. You just need to pass the required number of parameters to the function like this -

```
avg(6.56, 13.44)
```

Here is a complete example -

```
package main  
  
import "fmt"  
  
func avg(x float64, y float64) float64 {
```

```

        return (x + y) / 2
    }

func main() {
    x := 5.75
    y := 6.25

    result := avg(x, y)

    fmt.Printf("Average of %.2f and %.2f = %.2f\n", x, y, result)
}

# Output
Average of 5.75 and 6.25 = 6.00

```

Function parameters and return type(s) are optional

The input parameters and return type(s) are optional for a function. A function can be declared without any input and output.

The `main()` function is an example of such a function -

```

func main() {
}

```

Here is another example -

```

func sayHello() {
    fmt.Println("Hello, World")
}

```

You need to specify the type only once for multiple consecutive parameters of the same type

If a function has two or more consecutive parameters of the same type, then it suffices to specify the type only once for the last parameter of that type.

For example, we can declare the `avg` function that we saw in the previous section like this as well -

```
func avg(x, y float64) float64 { }  
  
// Same as - func avg(x float64, y float64) float64 { }
```

Here is another example -

```
func printPersonDetails(firstName, lastName string, age int) { }  
  
// Same as - func printPersonDetails(firstName string, lastName string, age int) { }
```

Functions with multiple return values

Go functions are capable of returning multiple values. That's right! This is something that most programming languages don't support natively. But Go is different.

Let's say that you want to create a function that takes the *previous price* and the *current price* of a stock, and returns the amount by which the price has changed and the percentage of change.

Here is how you can implement such a function in Go -

```
func getStockPriceChange(prevPrice, currentPrice float64) (float64, float64) {  
    change := currentPrice - prevPrice  
    percentChange := (change / prevPrice) * 100  
    return change, percentChange  
}
```

Simple! isn't it? You just need to specify the return types separated by comma inside parentheses, and then return multiple comma-separated values from the function.

Let's see a complete example with the `main()` function -

```
package main

import (
    "fmt"
    "math"
)

func getStockPriceChange(prevPrice, currentPrice float64) (float64, float64) {
    change := currentPrice - prevPrice
    percentChange := (change / prevPrice) * 100
    return change, percentChange
}

func main() {
    prevStockPrice := 75000.0
    currentStockPrice := 100000.0

    change, percentChange := getStockPriceChange(prevStockPrice, currentStockPrice)

    if change < 0 {
        fmt.Printf("The Stock Price decreased by $%.2f which is %.2f%% of the prev price\n", math.Abs(change), math.Abs(percentChange))
    } else {
        fmt.Printf("The Stock Price increased by $%.2f which is %.2f%% of the prev price\n", change, percentChange)
    }
}

# Output

The Stock Price increased by $25000.00 which is 33.33% of the prev price
```

Returning an error value from a function

Multiple return values are often used in Golang to return an error from the function along with the result.

Let's see an example - The `getStockPriceChange` function that we saw in the previous section will return $\pm\text{Inf}$ (Infinity) if the `prevPrice` is 0. If you want to return an error instead, you can do so by adding another return value of type `error` and return the error value like so -

```
func getStockPriceChangeWithError(prevPrice, currentPrice float64) (float64, float64, error) {  
    if prevPrice == 0 {  
        err := errors.New("Previous price cannot be zero")  
        return 0, 0, err  
    }  
    change := currentPrice - prevPrice  
    percentChange := (change / prevPrice) * 100  
    return change, percentChange, nil  
}
```

The `error` type is a built-in type in Golang. Go programs use `error` values to indicate an abnormal situation. Don't worry if you don't understand about errors for now. You'll learn more about error handling in a future session.

Following is a complete example demonstrating the above concept with a `main()` function -

```
package main  
  
import (  
    "errors"  
    "fmt"  
    "math"
```

```

)

func getPriceChangeWithError(prevPrice, currentPrice float64) (float64, float64, error) {
    if prevPrice == 0 {
        err := errors.New("Previous price cannot be zero")
        return 0, 0, err
    }
    change := currentPrice - prevPrice
    percentChange := (change / prevPrice) * 100
    return change, percentChange, nil
}

func main() {
    prevStockPrice := 0.0
    currentStockPrice := 100000.0

    change, percentChange, err := getPriceChangeWithError(prevStockPrice,
currentStockPrice)

    if err != nil {
        fmt.Println("Sorry! There was an error: ", err)
    } else {
        if change < 0 {
            fmt.Printf("The Stock Price decreased by $%.2f which is %.2f%% of the
prev price\n", math.Abs(change), math.Abs(percentChange))
        } else {
            fmt.Printf("The Stock Price increased by $%.2f which is %.2f%% of the
prev price\n", change, percentChange)
        }
    }
}

```

```
}
```

```
# Output
```

```
Sorry! There was an error: Previous price cannot be zero
```

Functions with named return values

The return values of a function in Golang may be named. Named return values behave as if you defined them at the top of the function.

Let's rewrite the `getStockPriceChange` function that we saw in the previous section with named return values -

```
// Function with named return values
```

```
func getNamedStockPriceChange(prevPrice, currentPrice float64) (change, percentChange float64) {  
    change = currentPrice - prevPrice  
    percentChange = (change / prevPrice) * 100  
    return change, percentChange  
}
```

Notice how we changed `:=` (short declarations) with `=` (assignments) in the function body. This is because Go itself defines all the named return values and makes them available for use in the function. Since they are already defined, you can't define them again using short declarations.

Named return values allow you to use the so-called **Naked return** (a return statement without any argument). When you specify a return statement without any argument, it returns the named return values by default. So you can write the above function like this as well -

```
// Function with named return values and naked return
```

```
func getNamedStockPriceChange(prevPrice, currentPrice float64) (change, percentChange float64) {  
    change = currentPrice - prevPrice  
    percentChange = (change / prevPrice) * 100  
    return  
}
```

```
        return  
    }  
}
```

Let's use the above function in a complete example with the `main()` function and verify the output -

```
package main  
  
import (  
    "fmt"  
    "math"  
)  
  
func getNamedStockPriceChange(prevPrice, currentPrice float64) (change, percentChange float64) {  
    change = currentPrice - prevPrice  
    percentChange = (change / prevPrice) * 100  
    return  
}  
  
func main() {  
    prevStockPrice := 100000.0  
    currentStockPrice := 90000.0  
  
    change, percentChange := getNamedStockPriceChange(prevStockPrice, currentStockPrice)  
  
    if change < 0 {  
        fmt.Printf("The Stock Price decreased by $%.2f which is %.2f%% of the prev  
price\n", math.Abs(change), math.Abs(percentChange))  
    } else {  
        fmt.Printf("The Stock Price increased by $%.2f which is %.2f%% of the prev price\n",  
change, percentChange)  
    }  
}
```

```
}  
  
}
```

Output

The Stock Price decreased by \$10000.00 which is 10.00% of the prev price

Named return values improve the readability of your functions. Using meaningful names would let the consumers of your function know what the function will return just by looking at its signature.

The naked return statements are good for short functions. But don't use them if your functions are long. They can harm the readability. You should explicitly specify the return values in longer functions.

Blank Identifier

Sometimes you may want to ignore some of the results from a function that returns multiple values.

For example, Let's say that you're using the `getStockPriceChange` function that we defined in the previous section, but you're only interested in the amount of change, not the percentage change.

Now, you might just declare local variables and store all the values returned from the function like this -

```
change, percentChange := getStockPriceChange(prevStockPrice, currentStockPrice)
```

But in that case, you'll be forced to use the `percentChange` variable because Go doesn't allow creating variables that you never use.

So what's the solution? Well, you can use a *blank identifier* instead -

```
change, _ := getStockPriceChange(prevStockPrice, currentStockPrice)
```

The blank identifier is used to tell Go that you don't need this value. The following example demonstrates this concept -

```

package main

import (
    "fmt"
    "math"
)

func getStockPriceChange(prevPrice, currentPrice float64) (float64, float64) {
    change := currentPrice - prevPrice
    percentChange := (change / prevPrice) * 100
    return change, percentChange
}

func main() {
    prevStockPrice := 80000.0
    currentStockPrice := 120000.0

    change, _ := getStockPriceChange(prevStockPrice, currentStockPrice)

    if change < 0 {
        fmt.Printf("The Stock Price decreased by $%.2f\n", math.Abs(change))
    } else {
        fmt.Printf("The Stock Price increased by $%.2f\n", change)
    }
}

# Output
The Stock Price increased by $40000.00

```


Session 8- A Beginner Guide to Packages in Golang

Go was designed to encourage good software engineering practices. One of the guiding principles of high-quality software is the DRY principle -

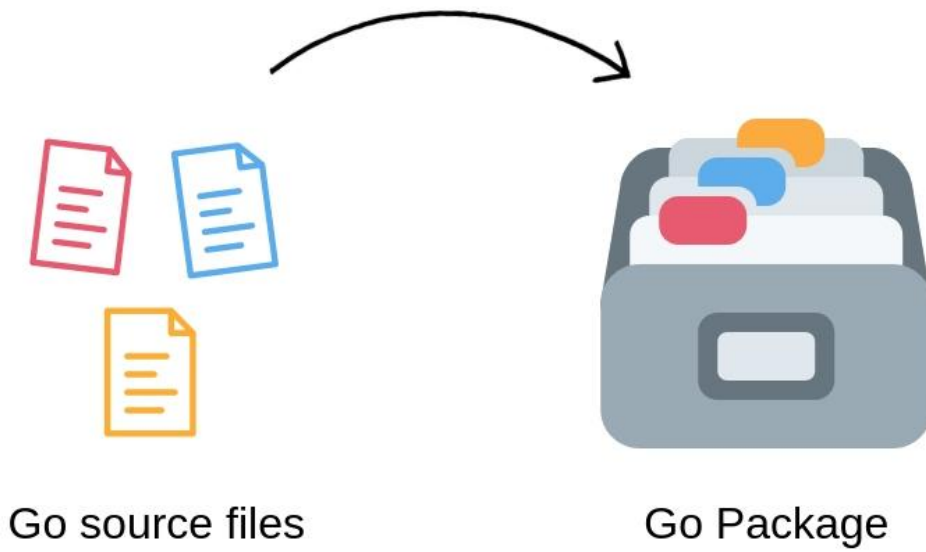
Don't Repeat Yourself, which basically means that you should never write the same code twice. You should reuse and build upon existing code as much as possible.

Functions are the most basic building blocks that allow code reuse. **Packages** are the next step into code reusability. They help you organize related Go source files together into a single unit, making them modular, reusable, and maintainable.

In this session, you'll learn how to organize Go code into reusable packages, how to import a package, how to export a function, type, or variable to outside packages, and how to install 3rd party packages.

Go Package

In the most basic terms, A package is nothing but a directory inside your Go workspace containing one or more Go source files, or other Go packages.



Every Go source file belongs to a package. To declare a source file to be part of a package, we use the following syntax -

```
package <packagename>
```

The above package declaration must be the first line of code in your Go source file. All the functions, types, and variables defined in your Go source file become part of the declared package.

You can choose to export a member defined in your package to outside packages, or keep them private to the same package. Other packages can import and reuse the functions or types that are exported from your package.

Let's see an example

Almost all the code that we have seen so far in so far include the following line -

```
import "fmt"
```

`fmt` is a core library package that contains functionalities related to formatting and printing output or reading input from various I/O sources. It exports functions like `Println()`, `Printf()`, `Scanf()` etc, for other packages to reuse.

Packaging functionalities in this way has the following benefits -

- It reduces naming conflicts. You can have the same function names in different packages. This keeps our function names short and concise.
- It organizes related code together so that it is easier to find the code you want to reuse.
- It speeds up the compilation process by only requiring recompilation of smaller parts of the program that has actually changed. Although we use the `fmt` package, we don't need to recompile it every time we change our program.

The main package and main() function

Go programs start running in the `main` package. It is a special package that is used with programs that are meant to be executable.

By convention, Executable programs (the ones with the `main` package) are called *Commands*. Others are called simply *Packages*.

The `main()` function is a special function that is the entry point of an executable program. Let's see an example of an executable program in Go -

```
// Package declaration
package main

// Importing packages
import (
    "fmt"
    "time"
    "math"
    "math/rand"
)

func main() {
    // Finding the Max of two numbers
```

```

    fmt.Println(math.Max(73.15, 92.46))

    // Calculate the square root of a number
    fmt.Println(math.Sqrt(225))

    // Printing the value of  $\pi$ 
    fmt.Println(math.Pi)

    // Epoch time in milliseconds
    epoch := time.Now().Unix()
    fmt.Println(epoch)

    // Generating a random integer between 0 to 100
    rand.Seed(epoch)
    fmt.Println(rand.Intn(100))
}

$ go run main.go

# Output

92.46

15

3.141592653589793

1538045386

40

```

Importing Packages

There are two ways to import packages in Go -

```
// Multiple import statements
```

```
import "fmt"
import "time"
import "math"
import "math/rand"

// Factored import statements
import (
    "fmt"
    "time"
    "math"
    "math/rand"
)
```

Go's convention is that - *the package name is the same as the last element of the import path*. For example, the name of the package imported as `math/rand` is `rand`. It is imported with path `math/rand` because it is nested inside the `math` package as a subdirectory.

Exported vs Unexported names

Anything (variable, type, or function) that starts with a capital letter is exported, and visible outside the package.

Anything that does not start with a capital letter is not exported, and is visible only inside the same package.

When you import a package, you can only access its exported names.

```
package main

import (
    "fmt"
    "math"
)
```

```
func main() {
    // MaxInt64 is an exported name
    fmt.Println("Max value of int64: ", int64(math.MaxInt64))

    // Phi is an exported name
    fmt.Println("Value of Phi ( $\phi$ ): ", math.Phi)

    // pi starts with a small letter, so it is not exported
    fmt.Println("Value of Pi ( $\pi$ ): ", math.pi)
}
```

Output

```
./exported_names.go:16:38: cannot refer to unexported name math.pi
./exported_names.go:16:38: undefined: math.pi
```

To fix the above error, you need to change `math.pi` to `math.Pi`.

Creating and managing custom Packages

Until now, we have only written code in the `main` package and used functionalities imported from Go's core library packages.

Let's create a sample Go project that has multiple custom packages with a bunch of source code files and see how the same concept of package declaration, imports, and exports apply to custom packages as well.

Fire up your terminal and create a directory for our Go project:

```
$ mkdir packer
```

Next, we'll create a [Go module](#) and make the project directory the root of the module.

Note: Go module is Go's new dependency management system. A module is a collection of Go packages stored in a directory with a **go.mod** file at its root. The go.mod file defines the module's path, which is also the import path used while importing packages that are part of this module.

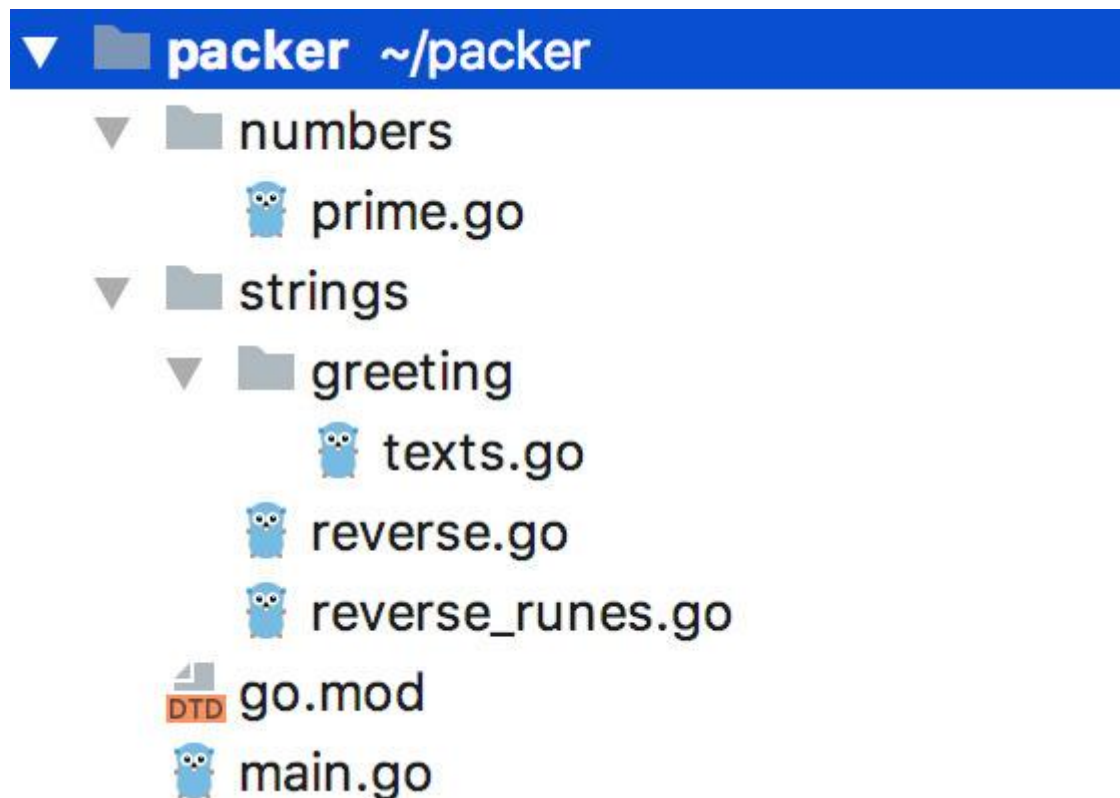
Before Go module got introduced in Go 1.11, every project needed to be created inside the so-called **GOPATH**. The path of the project inside GOPATH was considered its import path.

We'll learn more about Go modules in a separate session.

Let's initialize a Go module by typing the following commands:

```
$ cd packer
$ go mod init URL/packer
```

Let's now create some source files and place them in different packages inside our project. The following image displays all the packages and the source files:



Here is the code inside every source file of our project -

numbers/prime.go

```

package numbers

import "math"

// Checks if a number is prime or not
func IsPrime(num int) bool {
    for i := 2; i <= int(math.Floor(math.Sqrt(float64(num)))); i++ {
        if num%i == 0 {
            return false
        }
    }
    return num > 1
}

```

strings/reverse.go

```

package strings

// Reverses a string
/*
    Since strings in Go are immutable, we first convert the string to a mutable array of runes
    ([]rune),
    perform the reverse operation on that, and then re-cast to a string.
*/
func Reverse(s string) string {
    runes := []rune(s)
    reversedRunes := reverseRunes(runes)
    return string(reversedRunes)
}

```



```
}
```

strings/reverse_runes.go

```
package strings

// Reverses an array of runes
// This function is not exported (It is only visible inside the `strings` package)
func reverseRunes(r []rune) []rune {
    for i, j := 0, len(r)-1; i < j; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return r
}
```

strings/greeting/texts.go (Nested package)

```
// Nested Package
package greeting

// Exported
const (
    WelcomeText = "Hello, World to Golang"
    MorningText = "Good Morning"
    EveningText = "Good Evening"
)

// Not exported (only visible inside the `greeting` package)
var loremIpsumText = `Lorem ipsum dolor sit amet, consectetur adipiscing elit,
```

sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.`

main.go (The main package: entry point of our program)

```
package main

import (
    "fmt"
    str "strings" // Package Alias

    "URL/packer/numbers"
    "URL/packer/strings"
    "URL/packer/strings/greeting" // Importing a nested package
)

func main() {
    fmt.Println(numbers.IsPrime(19))

    fmt.Println(greeting.WelcomeText)

    fmt.Println(strings.Reverse("callicoder"))

    fmt.Println(str.Count("Go is Awesome. I love Go", "Go"))
}

# Building the Go module

$ go build
```

The above command will produce an executable binary. Let's execute the binary file to run the program:

```
# Running the executable binary
```

```
$ ./packer
```

```
true
```

```
Hello, World to Golang
```

```
redocillac
```

```
2
```

Things to note

- **Import Paths**

All import paths are relative to the module's path URL/packer.

```
import (  
    "URL/packer/numbers"  
    "URL/packer/strings"  
    "URL/packer/strings/greeting"  
)
```

- **Package Alias**

You can use package alias to resolve conflicts between different packages of the same name, or just to give a short name to the imported package

```
import (  
    str "strings" // Package Alias  
)
```

- **Nested Package**

You can nest a package inside another. It's as simple as creating a subdirectory -

```
packer
  strings      # Package
    greeting   # Nested Package
      texts.go
```

A nested package can be imported similar to a root package. Just provide its path relative to the module's path URL/packer -

```
import (
    "URL/packer/strings/greeting"
)
```

Adding 3rd party Packages

Adding 3rd party packages to your project is very easy with Go modules. You can just import the package to any of the source files in your project, and the next time you build/run the project, Go automatically downloads it for you -

```
package main

import (
    "fmt"
    "rsc.io/quote"
)

func main() {
    fmt.Println(quote.Go())
}

$ go run main.go

go: finding rsc.io/quote v1.5.2
```

```
go: downloading rsc.io/quote v1.5.2
go: extracting rsc.io/quote v1.5.2
go: downloading rsc.io/sampler v1.3.0
go: extracting rsc.io/sampler v1.3.0
go: downloading golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
go: extracting golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
go: finding rsc.io/sampler v1.3.0
go: finding golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
```

Don't communicate by sharing memory, share memory by communicating.

Go will also add this new dependency to the `go.mod` file.

Manually installing packages

You can use `go get` command to download 3rd party packages from remote repositories.

```
$ go get -u github.com/jinzhu/gorm
```

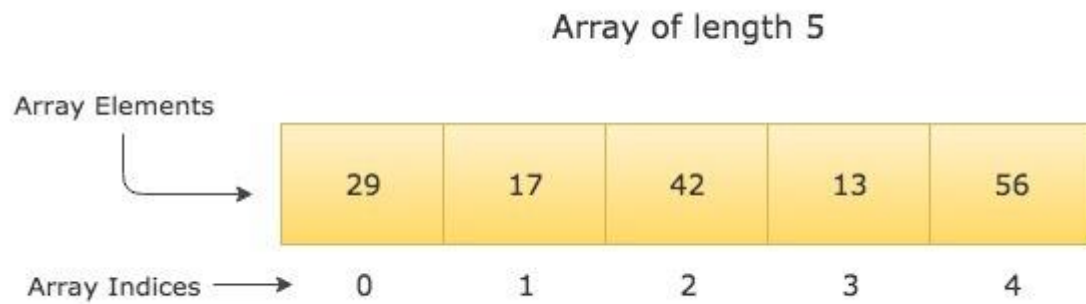
The above command fetches the `gorm` package from Github and adds it as a dependency to your `go.mod` file.

That's it. You can now import and use the above package in your program like this -

```
import "github.com/jinzhu/gorm"
```

Session 9- Working with Arrays in Golang

An array is a fixed-size collection of elements of the same type. The elements of the array are stored sequentially and can be accessed using their `index`.



Declaring an Array in Golang

You can declare an array of length n and type T like so -

```
var a[n]T
```

For example, here is how you can declare an array of 10 integers -

```
// An array of 10 integers  
var a[10]int
```

Now let's see a complete example -

```
package main  
  
import "fmt"  
  
func main() {  
    var x [5]int // An array of 5 integers  
    fmt.Println(x)  
  
    var y [8]string // An array of 8 strings  
    fmt.Println(y)  
}
```

```

var z [3]complex128 // An array of 3 complex numbers

fmt.Println(z)

}

# Output

[0 0 0 0]

[ ]

[(0+0i) (0+0i) (0+0i)]

```

By default, all the array elements are initialized with the zero value of the corresponding array type.

For example, if we declare an integer array, all the elements will be initialized with 0. If we declare a string array, all the elements will be initialized with an empty string "", and so on.

Accessing array elements by their index

The elements of an array are stored sequentially and can be accessed by their index. The index starts from zero and ends at length - 1.

```

package main

import "fmt"

func main() {

    var x [5]int // An array of 5 integers

    x[0] = 100
    x[1] = 101
    x[3] = 103
    x[4] = 105
}

```

```

    fmt.Printf("x[0] = %d, x[1] = %d, x[2] = %d\n", x[0], x[1], x[2])

    fmt.Println("x = ", x)

}

```

Output

```

x[0] = 100, x[1] = 101, x[2] = 0
x = [100 101 0 103 105]

```

In the above example, Since we didn't assign any value to `x[2]`, it has the value 0 (The zero value for integers).

Initializing an array using an array literal

You can declare and initialize an array at the same time like this -

```

// Declaring and initializing an array at the same time

var a = [5]int{2, 4, 6, 8, 10}

```

The expression on the right-hand side of the above statement is called an array literal.

Note that we do not need to specify the type of the variable `a` as in `var a [5]int`, because the compiler can automatically infer the type from the expression on the right hand side.

You can also use Golang's short variable declaration for declaring and initializing an array. The above array declaration can also be written as below inside any function -

```

// Short hand declaration

a := [5]int{2, 4, 6, 8, 10}

```

Here is a complete example -


```

package main

import "fmt"

func main() {
    // Declaring and initializing an array at the same time
    var a = [5]int{2, 4, 6, 8, 10}
    fmt.Println(a)

    // Short declaration for declaring and initializing an array
    b := [5]int{2, 4, 6, 8, 10}
    fmt.Println(b)

    // You don't need to initialize all the elements of the array.
    // The un-initialized elements will be assigned the zero value of the corresponding array type
    c := [5]int{2}
    fmt.Println(c)
}

# Output

[2 4 6 8 10]
[2 4 6 8 10]
[2 0 0 0 0]

```

Letting Go compiler infer the length of the array

You can also omit the size declaration from the initialization expression of the array, and let the compiler count the number of elements for you -

```

package main

import "fmt"

```

```
func main() {
    // Letting Go compiler infer the length of the array
    a := [...]int{3, 5, 7, 9, 11, 13, 17}
    fmt.Println(a)
}

# Output
[3 5 7 9 11 13 17]
```

Exploring more about Golang arrays

1. Array's length is part of its type

The length of an array is part of its type. So the array `a[5]int` and `a[10]int` are completely distinct types, and you cannot assign one to the other.

This also means that you cannot resize an array, because resizing an array would mean changing its type, and you cannot change the type of a variable in Golang.

```
package main

func main() {
    var a = [5]int{3, 5, 7, 9, 11}
    var b [10]int = a // Error, a and b are distinct types
}
```

2. Arrays in Golang are value types

Arrays in Golang are value types unlike other languages like C, C++, and Java where arrays are reference types.

This means that when you assign an array to a new variable or pass an array to a function, the entire array is copied. So if you make any changes to this copied array, the original array won't be affected and will remain unchanged.

Here is an example -

```
package main

import "fmt"

func main() {
    a1 := [5]string{"English", "Japanese", "Spanish", "French", "Hindi"}
    a2 := a1 // A copy of the array `a1` is assigned to `a2`

    a2[1] = "German"

    fmt.Println("a1 = ", a1) // The array `a1` remains unchanged
    fmt.Println("a2 = ", a2)
}

# Output
a1 = [English Japanese Spanish French Hindi]
a2 = [English German Spanish French Hindi]
```

Iterating over an array in Golang

You can use the for loop to iterate over an array like so -

```
package main

import "fmt"

func main() {
    names := [3]string{"Mark Zuckerberg", "Bill Gates", "Larry Page"}

    for i := 0; i < len(names); i++ {
```

```

        fmt.Println(names[i])
    }
}

```

Output

```

Mark Zuckerberg
Bill Gates
Larry Page

```

The `len()` function in the above example is used to find the length of the array.

Let's see another example. In the example below, we find the sum of all the elements of the array by iterating over the array, and adding the elements one by one to the variable `sum` -

```

package main

import "fmt"

func main() {
    a := [4]float64{3.5, 7.2, 4.8, 9.5}
    sum := float64(0)

    for i := 0; i < len(a); i++ {
        sum = sum + a[i]
    }

    fmt.Printf("Sum of all the elements in array %v = %f\n", a, sum)
}

```

Output

```

Sum of all the elements in array [3.5 7.2 4.8 9.5] = 25.000000

```

Iterating over an array using range

Golang provides a more powerful form of for loop using the `range` operator. Here is how you can use the `range` operator with for loop to iterate over an array -

```
package main

import "fmt"

func main() {
    daysOfWeek := []string{"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"}

    for index, value := range daysOfWeek {
        fmt.Printf("Day %d of week = %s\n", index, value)
    }
}
```

Output

```
Day 0 of week = Mon
Day 1 of week = Tue
Day 2 of week = Wed
Day 3 of week = Thu
Day 4 of week = Fri
Day 5 of week = Sat
Day 6 of week = Sun
```

Let's now write the same `sum` example that we wrote with normal for loop using `range` form of the for loop -

```
package main

import "fmt"
```

```

func main() {
    a := [4]float64{3.5, 7.2, 4.8, 9.5}
    sum := float64(0)

    for index, value := range a {
        sum = sum + value
    }

    fmt.Printf("Sum of all the elements in array %v = %f", a, sum)
}

```

When you run the above program, it'll generate an error like this -

Output

```
./array_iteration_range.go:9:13: index declared and not used
```

Go compiler doesn't allow creating variables that are never used. You can fix this by using an `_` (underscore) in place of `index` -

```

package main

import "fmt"

func main() {
    a := [4]float64{3.5, 7.2, 4.8, 9.5}
    sum := float64(0)

    for _, value := range a {
        sum = sum + value
    }
}

```

```
    fmt.Printf("Sum of all the elements in array %v = %f", a, sum)
}
```

The underscore (_) is used to tell the compiler that we don't need this variable. The above program now runs successfully and outputs the sum of the array -

Output

Sum of all the elements in array [3.5 7.2 4.8 9.5] = 25.000000

Multidimensional arrays in Golang

All the arrays that we created so far in this post are one dimensional. You can also create multi-dimensional arrays in Golang.

The following example demonstrates how to create multidimensional arrays -

```
package main

import "fmt"

func main() {
    a := [2][2]int{
        {3, 5},
        {7, 9},    // This trailing comma is mandatory
    }

    fmt.Println(a)

    // Just like 1D arrays, you don't need to initialize all the elements in a multi-dimensional array.
    // Un-initialized array elements will be assigned the zero value of the array type.

    b := [3][4]float64{
```

```

        {1, 3},
        {4.5, -3, 7.4, 2},
        {6, 2, 11},
    }

    fmt.Println(b)
}

```

Output

```

[[3 5] [7 9]]
[[1 3 0 0] [4.5 -3 7.4 2] [6 2 11 0]]

```

Session 10- Introduction to Slices in Golang

A Slice is a segment of an array. Slices build on arrays and provide more power, flexibility, and convenience compared to arrays.

Just like arrays, Slices are indexable and have a length. But unlike arrays, they can be resized.

Internally, A Slice is just a reference to an underlying array. In this session, we'll learn how to create and use slices, and also understand how they work under the hood.

Declaring a Slice

A slice of type `T` is declared using `[]T`. For example, here is how you can declare a slice of type `int` -

```

// Slice of type `int`
var s []int

```


The slice is declared just like an array except that we do not specify any size in the brackets [].

Creating and Initializing a Slice

1. Creating a slice using a slice literal

You can create a slice using a slice literal like this -

```
// Creating a slice using a slice literal  
var s = []int{3, 5, 7, 9, 11, 13, 17}
```

The expression on the right-hand side of the above statement is a slice literal. The slice literal is declared just like an array literal, except that you do not specify any size in the square brackets [].

When you create a slice using a slice literal, it first creates an array and then returns a slice reference to it.

Let's see a complete example -

```
package main  
  
import "fmt"  
  
func main() {  
    // Creating a slice using a slice literal  
    var s = []int{3, 5, 7, 9, 11, 13, 17}  
  
    // Short hand declaration  
    t := []int{2, 4, 8, 16, 32, 64}  
  
    fmt.Println("s = ", s)  
    fmt.Println("t = ", t)  
}
```

Output

```
s = [3 5 7 9 11 13 17]
```

```
t = [2 4 8 16 32 64]
```

2. Creating a slice from an array

Since a slice is a segment of an array, we can create a slice from an array.

To create a slice from an array `a`, we specify two indices `low` (lower bound) and `high` (upper bound) separated by a colon -

```
// Obtaining a slice from an array `a`  
a[low:high]
```

The above expression selects a slice from the array `a`. The resulting slice includes all the elements starting from index `low` to `high`, but excluding the element at index `high`.

Let's see an example to make things more clear -

```
package main  
  
import "fmt"  
  
func main() {  
    var a = [5]string{"Alpha", "Beta", "Gamma", "Delta", "Epsilon"}  
  
    // Creating a slice from the array  
    var s []string = a[1:4]  
  
    fmt.Println("Array a = ", a)  
    fmt.Println("Slice s = ", s)  
}
```

Array a = [Alpha Beta Gamma Delta Epsilon]

Slice s = [Beta Gamma Delta]

The low and high indices in the slice expression are optional. The default value for low is 0, and high is the length of the slice.

```
package main

import "fmt"

func main() {

    a := [5]string{"C", "C++", "Java", "Python", "Go"}

    slice1 := a[1:4]
    slice2 := a[:3]
    slice3 := a[2:]
    slice4 := a[:]

    fmt.Println("Array a = ", a)
    fmt.Println("slice1 = ", slice1)
    fmt.Println("slice2 = ", slice2)
    fmt.Println("slice3 = ", slice3)
    fmt.Println("slice4 = ", slice4)

}
```

Output

Array a = [C C++ Java Python Go]

slice1 = [C++ Java Python]

slice2 = [C C++ Java]

slice3 = [Java Python Go]

```
slice4 = [C C++ Java Python Go]
```

3. Creating a slice from another slice

A slice can also be created by slicing an existing slice.

```
package main

import "fmt"

func main() {
    cities := []string{"New York", "London", "Chicago", "Beijing", "Delhi", "Mumbai", "Bangalore",
        "Hyderabad", "Hong Kong"}

    asianCities := cities[3:]
    indianCities := asianCities[1:5]

    fmt.Println("cities = ", cities)
    fmt.Println("asianCities = ", asianCities)
    fmt.Println("indianCities = ", indianCities)
}
```

Output

```
cities = [New York London Chicago Beijing Delhi Mumbai Bangalore Hyderabad Hong Kong]
```

```
asianCities = [Beijing Delhi Mumbai Bangalore Hyderabad Hong Kong]
```

```
indianCities = [Delhi Mumbai Bangalore Hyderabad]
```

Modifying a slice

Slices are reference types. They refer to an underlying array. Modifying the elements of a slice will modify the corresponding elements in the referenced array. Other slices that refer the same array will also see those modifications.

```

package main

import "fmt"

func main() {
    a := [7]string{"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"}

    slice1 := a[1:]
    slice2 := a[3:]

    fmt.Println("----- Before Modifications -----")
    fmt.Println("a = ", a)
    fmt.Println("slice1 = ", slice1)
    fmt.Println("slice2 = ", slice2)

    slice1[0] = "TUE"
    slice1[1] = "WED"
    slice1[2] = "THU"

    slice2[1] = "FRIDAY"

    fmt.Println("\n----- After Modifications -----")
    fmt.Println("a = ", a)
    fmt.Println("slice1 = ", slice1)
    fmt.Println("slice2 = ", slice2)
}

```

Output

```

----- Before Modifications -----

a = [Mon Tue Wed Thu Fri Sat Sun]
slice1 = [Tue Wed Thu Fri Sat Sun]

```

```
slice2 = [Thu Fri Sat Sun]
```

----- After Modifications -----

```
a = [Mon TUE WED THU FRIDAY Sat Sun]
```

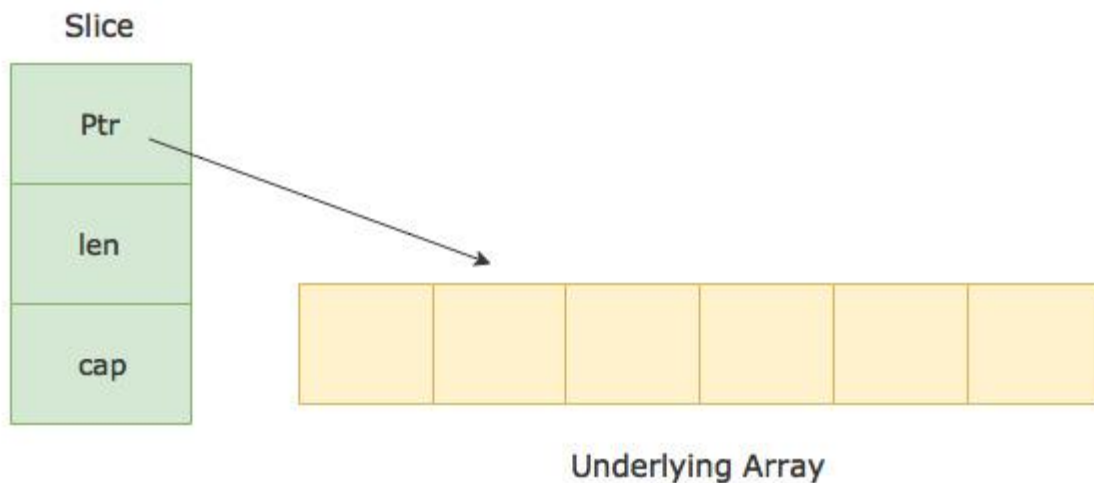
```
slice1 = [TUE WED THU FRIDAY Sat Sun]
```

```
slice2 = [THU FRIDAY Sat Sun]
```

Length and Capacity of a Slice

A slice consists of three things -

- A **pointer** (reference) to an underlying array.
- The **length** of the segment of the array that the slice contains.
- The **capacity** (the maximum size up to which the segment can grow).

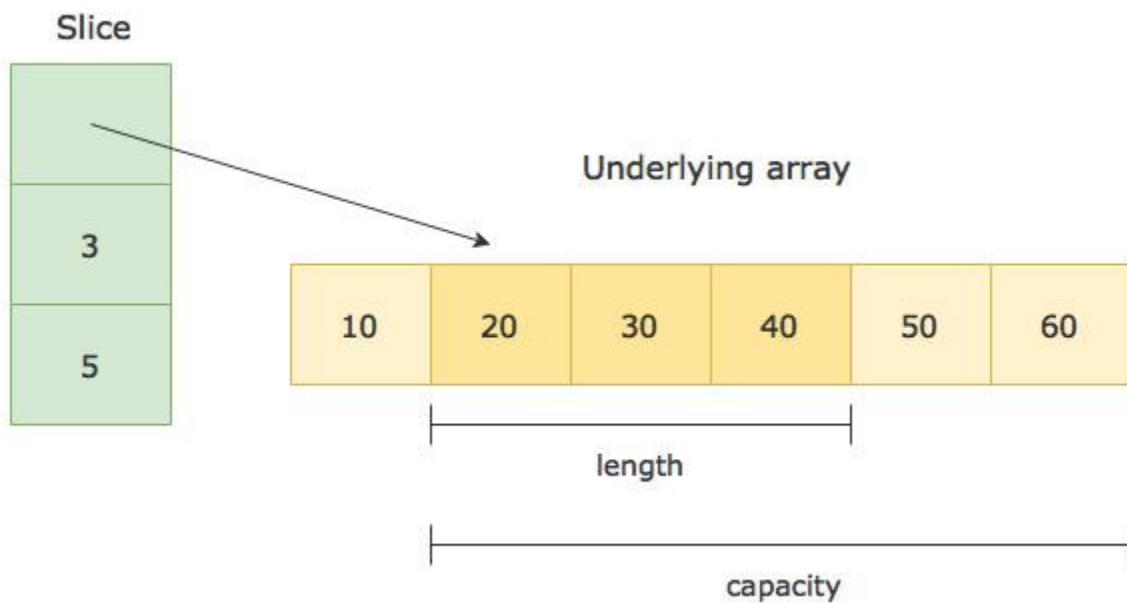


Let's consider the following array and the slice obtained from it as an example -

```
var a = [6]int{10, 20, 30, 40, 50, 60}
```

```
var s = [1:4]
```

Here is how the slice `s` in the above example is represented -



The length of the slice is the number of elements in the slice, which is 3 in the above example.

The capacity is the number of elements in the underlying array starting from the first element in the slice. It is 5 in the above example.

You can find the length and capacity of a slice using the built-in functions `len()` and `cap()` -

```
package main

import "fmt"

func main() {
    a := [6]int{10, 20, 30, 40, 50, 60}
    s := a[1:4]

    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))
}

# Output
s = [20 30 40], len = 3, cap = 5
```

A slice's length can be extended up to its capacity by re-slicing it. Any attempt to extend its length beyond the available capacity will result in a runtime error.

Check out the following example to understand how re-slicing a given slice changes its length and capacity -

```
package main

import "fmt"

func main() {
    s := []int{10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

    fmt.Println("Original Slice")

    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))

    s = s[1:5]

    fmt.Println("\nAfter slicing from index 1 to 5")

    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))

    s = s[:8]

    fmt.Println("\nAfter extending the length")

    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))

    s = s[2:]

    fmt.Println("\nAfter dropping the first two elements")

    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))
}
```

Output

Original Slice

s = [10 20 30 40 50 60 70 80 90 100], len = 10, cap = 10

After slicing from index 1 to 5


```
s = [20 30 40 50], len = 4, cap = 9
```

After extending the length

```
s = [20 30 40 50 60 70 80 90], len = 8, cap = 9
```

After dropping the first two elements

```
s = [40 50 60 70 80 90], len = 6, cap = 7
```

Creating a slice using the built-in make() function

Now that we know about the length and capacity of a slice. Let's look at another way to create a slice.

Golang provides a library function called `make()` for creating slices. Following is the signature of `make()` function -

```
func make([]T, len, cap) []T
```

The `make` function takes a type, a length, and an optional capacity. It allocates an underlying array with size equal to the given capacity, and returns a slice that refers to that array.

```
package main

import "fmt"

func main() {
    // Creates an array of size 10, slices it till index 5, and returns the slice reference
    s := make([]int, 5, 10)
    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))
}

# Output
```

```
s = [0 0 0 0 0], len = 5, cap = 10
```

The capacity parameter in the `make()` function is optional. When omitted, it defaults to the specified length -

```
package main

import "fmt"

func main() {
    // Creates an array of size 5, and returns a slice reference to it
    s := make([]int, 5)
    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))
}
```

Output

```
s = [0 0 0 0 0], len = 5, cap = 5
```

Zero value of slices

The zero value of a slice is `nil`. A `nil` slice doesn't have any underlying array, and has a length and capacity of 0 -

```
package main

import "fmt"

func main() {
    var s []int
    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))

    if s == nil {
```

```

        fmt.Println("s is nil")
    }
}

```

Output

s = [], len = 0, cap = 0

s is nil

Slice Functions

1. The `copy()` function: copying a slice

The `copy()` function copies elements from one slice to another. Its signature looks like this -

```
func copy(dst, src []T) int
```

It takes two slices - a destination slice, and a source slice. It then copies elements from the source to the destination and returns the number of elements that are copied.

The number of elements copied will be the minimum of `len(src)` and `len(dst)`.

```

package main

import "fmt"

func main() {
    src := []string{"Sublime", "VSCode", "IntelliJ", "Eclipse"}
    dest := make([]string, 2)

    numElementsCopied := copy(dest, src)
}

```

```

    fmt.Println("src = ", src)

    fmt.Println("dest = ", dest)

    fmt.Println("Number of elements copied from src to dest = ", numElementsCopied)
}

```

Output

src = [Sublime VSCode IntelliJ Eclipse]

dest = [Sublime VSCode]

Number of elements copied from src to dest = 2

2. The append() function: appending to a slice

The `append()` function appends new elements at the end of a given slice. Following is the signature of `append` function.

```
func append(s []T, x ...T) []T
```

It takes a slice and a variable number of arguments `x ...T`. It then returns a new slice containing all the elements from the given slice as well as the new elements.

If the given slice doesn't have sufficient capacity to accommodate new elements then a new underlying array is allocated with bigger capacity. All the elements from the underlying array of the existing slice are copied to this new array, and then the new elements are appended.

However, if the slice has enough capacity to accommodate new elements, then the `append()` function re-uses its underlying array and appends new elements to the same array.

Let's see an example to understand things better -

```

package main

import "fmt"

```

```
func main() {
    slice1 := []string{"C", "C++", "Java"}
    slice2 := append(slice1, "Python", "Ruby", "Go")

    fmt.Printf("slice1 = %v, len = %d, cap = %d\n", slice1, len(slice1), cap(slice1))
    fmt.Printf("slice2 = %v, len = %d, cap = %d\n", slice2, len(slice2), cap(slice2))

    slice1[0] = "C#"
    fmt.Println("\nslice1 = ", slice1)
    fmt.Println("slice2 = ", slice2)
}
```

Output

```
slice1 = [C C++ Java], len = 3, cap = 3
slice2 = [C C++ Java Python Ruby Go], len = 6, cap = 6

slice1 = [C# C++ Java]
slice2 = [C C++ Java Python Ruby Go]
```

In the above example, since `slice1` has capacity 3, it can't accommodate more elements. So a new underlying array is allocated with bigger capacity when we append more elements to it.

So if you modify `slice1`, `slice2` won't see those changes because it refers to a different array.

But what if `slice1` had enough capacity to accommodate new elements? Well, in that case, no new array would be allocated, and the elements would be added to the same underlying array.

Also, in that case, changes to `slice1` would affect `slice2` as well because both would refer to the same underlying array. This is demonstrated in the following example

-

```
package main
```

```

import "fmt"

func main() {
    slice1 := make([]string, 3, 10)
    copy(slice1, []string{"C", "C++", "Java"})

    slice2 := append(slice1, "Python", "Ruby", "Go")

    fmt.Printf("slice1 = %v, len = %d, cap = %d\n", slice1, len(slice1), cap(slice1))
    fmt.Printf("slice2 = %v, len = %d, cap = %d\n", slice2, len(slice2), cap(slice2))

    slice1[0] = "C#"
    fmt.Println("\nslice1 = ", slice1)
    fmt.Println("slice2 = ", slice2)
}

```

Output

slice1 = [C C++ Java], len = 3, cap = 10

slice2 = [C C++ Java Python Ruby Go], len = 6, cap = 10

slice1 = [C# C++ Java]

slice2 = [C# C++ Java Python Ruby Go]

Appending to a nil slice

When you append values to a nil slice, it allocates a new slice and returns the reference of the new slice.

```

package main

import "fmt"

```

```
func main() {
    var s []string

    // Appending to a nil slice

    s = append(s, "Cat", "Dog", "Lion", "Tiger")

    fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))
}
```

Output

s = [Cat Dog Lion Tiger], len = 4, cap = 4

Appending one slice to another

You can directly append one slice to another using the ... operator. This operator expands the slice to a list of arguments. The following example demonstrates its usage -

```
package main

import "fmt"

func main() {
    slice1 := []string{"Jack", "John", "Peter"}
    slice2 := []string{"Bill", "Mark", "Steve"}

    slice3 := append(slice1, slice2...)

    fmt.Println("slice1 = ", slice1)
    fmt.Println("slice2 = ", slice2)
    fmt.Println("After appending slice1 & slice2 = ", slice3)
}
```

Output

slice1 = [Jack John Peter]

slice2 = [Bill Mark Steve]

After appending slice1 & slice2 = [Jack John Peter Bill Mark Steve]

Slice of slices

Slices can be of any type. They can also contain other slices. The example below creates a slice of slices -

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    s := []string{
        {"India", "China"},
        {"USA", "Canada"},
        {"Switzerland", "Germany"},
    }
```

```
    fmt.Println("Slice s = ", s)
```

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

```
    fmt.Println("capacity = ", cap(s))
```

```
}
```

Output

Slice s = [[India China] [USA Canada] [Switzerland Germany]]

length = 3


```
capacity = 3
```

Iterating over a slice

You can iterate over a slice in the same way you iterate over an array. Following are two ways of iterating over a slice:

1. Iterating over a slice using for loop

```
package main

import "fmt"

func main() {
    countries := []string{"India", "America", "Russia", "England"}

    for i := 0; i < len(countries); i++ {
        fmt.Println(countries[i])
    }
}
```

Output

India

America

Russia

England

2. Iterating over a slice using the range form of for loop

```
package main
```

```
import "fmt"

func main() {
    primeNumbers := []int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}

    for index, number := range primeNumbers {
        fmt.Printf("PrimeNumber(%d) = %d\n", index+1, number)
    }
}
```

Output

```
PrimeNumber(1) = 2
PrimeNumber(2) = 3
PrimeNumber(3) = 5
PrimeNumber(4) = 7
PrimeNumber(5) = 11
PrimeNumber(6) = 13
PrimeNumber(7) = 17
PrimeNumber(8) = 19
PrimeNumber(9) = 23
PrimeNumber(10) = 29
```

Ignoring the index from the range form of for loop using Blank identifier

The range form of for loop gives you the index and the value at that index in each iteration. If you don't want to use the index, then you can discard it by using an underscore `_`.

The underscore (`_`) is called the blank identifier. It is used to tell Go compiler that we don't need this value.

```
package main
```

```
import "fmt"

func main() {
    numbers := []float64{3.5, 7.4, 9.2, 5.4}

    sum := 0.0
    for _, number := range numbers {
        sum += number
    }

    fmt.Printf("Total Sum = %.2f\n", sum)
}
```

Output

Total Sum = 25.50