Backend Web Development (CSC103)

Programming

Prepared by: Ms. Tshering Lhamo

Contents

What is Go?

- **Open-source (since 2009) programming language** developed at **Google** in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson.
- Go is a **statically typed, compiled** programming language.
    - **Statically typed**: Does *type checking* at the *compile time*. Similar languages are C, C++ .
    - Once a variable is declared to a specific type it **cannot** be changed to other types unless it is removed and recreated. (See example later)

    - **Compiled language**: Code written is directly translated into machine code (byte code or executable file).

- Go has a similar syntax to C. It is developed with the vision of **high performance** and **fast development**.
- Domain name is **golang.org** which is why it is popularly named as **Golang.**
- Companies that have shifted to using Golang:
    - Google, Apple, Facebook, Docker, SoundCloud, Uber, Dropbox, BBC, and more.

Why use Go?

1. Focus on **simplicity, clarity,** and **scalability**.
    - Inspired by languages like Python.
    - Aims to provide a clean, understandable syntax.

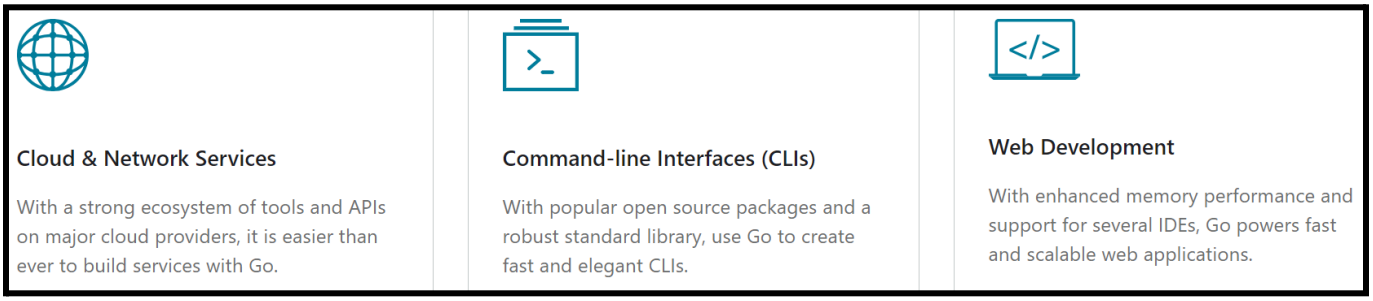
2. **High performance** and focus on **concurrency or parallelism**
    - Similar to C or C++
    - **Concurrency** is defined as the ability to execute more than one program or task simultaneously.
    - Popular for tasks that benefit from multi-threading. Go has ***Goroutines*** to achieve concurrency. Goroutines are basically ***functions*** that can run simultaneously and independently.

3. Go comes with a rich **standard library**. Includes many built-in core features. No need to use third party libraries to perform basic tasks.
4. Provides **garbage collection**: Automatic memory management.
5. **Static typing:** Go is a type-safe language.
    - Allows you to catch errors early during development instead of run time.
6. **Fast**: Code directly compiled into bytecode. Enhance the availability and reliability of services.

7. **Rigid dependency specification**:
    - Go programs are constructed from ***packages***, whose properties allow efficient management of dependencies.

- The go toolchain has a built-in system for managing versioned sets of related packages, known as ***modules***.
- ***go.mod*** file tracks dependency versions.

Popular Go use cases



Environment Setup

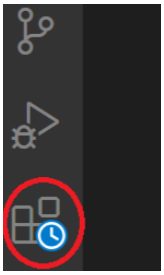
Code Editor

- A code editor is a tool developers use to make **writing code** go more smoothly. It is a standalone program into which the developer writes their code.
- Code editors have a range of features, including color-coded syntax highlighting, automatic indentation, error-checking, auto-completing with suggested code, and code snippets.
- Also, they often allow for added functionality through the use of **extensions.**
- Here, you are going to use the **Visual Studio Code** (VSCode) code editor.

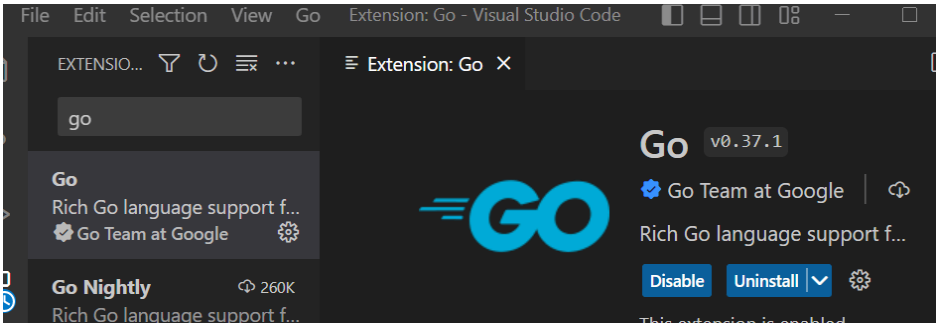
Instructions

1. First install the **VSCode** if you do not have it.
   Download VSCode installer here, *code.visualstudio.com/*

2. Configure VSCode.
   Install **Go** extension in VSCode. Click on the Extensions menu icon on the left hand side.



   Search for the “Go” extension and click on the **Install** button.



3. At the bottom of the editor window, click on the plaintext tab, change the language mode to go by typing go in the input box.

Local Environment on Windows (Optional)

The instructions given below are to set up the Go development environment on your local machine, windows. However, it is not recommended to install locally.
For now jump to the next topic. You may try to install it later on your local device.

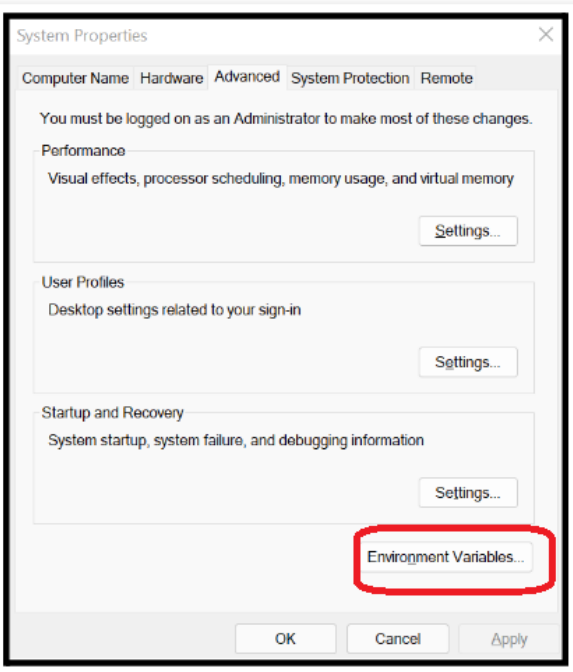
Instructions

1. Follow this link to download the Go installer. You need the Go runtime to build and execute Go programs on your local machine.
2. Open the MSI file you downloaded and follow the prompts to install Go.
   By default, the installer will install go to **Program Files or Program Files (x86)**, ***C:\Users\Username\go***
   You can change the location as needed.
3. Verify that you've installed Go.
   a. In Windows, click the **Start** menu.
   b. In the menu's search box, type **cmd**, then press the **Enter** key.
   c. In the Command Prompt window that appears, type the following command:
   $ go version

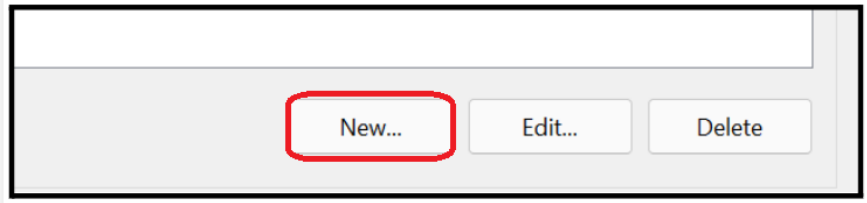
```
C:\Users\tsheringL>go version
go version go1.19.5 windows/amd64
```

4. Confirm that the command prints the installed version of Go.
   go version go1.19.5 windows/amd64 (at the time of installation)

5. Add environment variables before you start using Go.
   a. In the search box, type **env** and press the enter key
   b. Click on **Environment Variables..** button.



c. Click on the **New** button



d. Locate the path where you have installed go***(C:\Users\tsheringL\go)*** and paste it in the field as shown below. Set variable name as GOPATH.

e. Then click on **OK.** You have successfully added an environment variable.

Using Docker

Before installing Docker make sure that you have learnt about Docker. The Docker document (pdf) given to you helps you to understand Docker.

Instructions
(Skip any step if it is already completed)

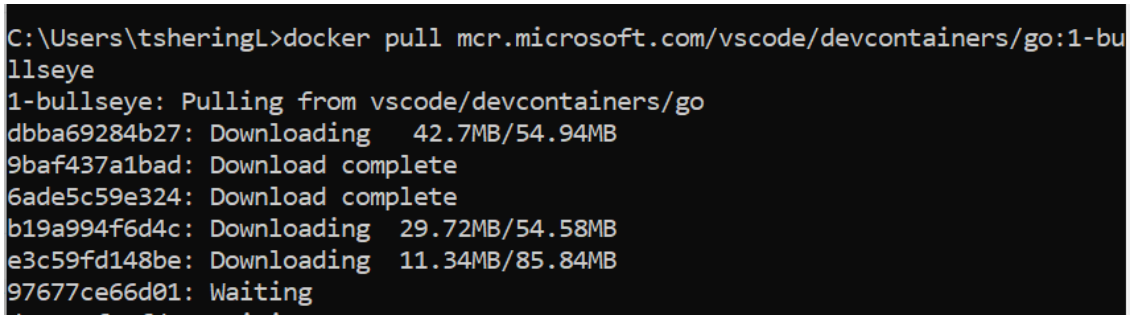
1. Install Docker
   Follow the link below to download docker installer.
   Windows: https://docs.docker.com/desktop/windows/install/
   Mac: https://docs.docker.com/desktop/mac/install/

2. Download **golang bullseye** docker image.
   The **backend language** that you are going to use is 'Go', for this purpose you need to download the Go docker image to create a container in the docker environment.

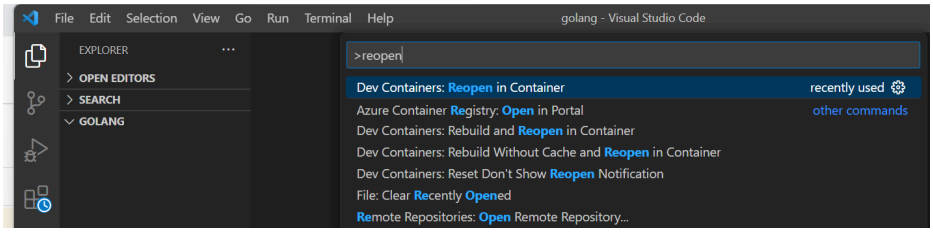
   Use the command below to download the image,

   ```
   docker pull mcr.microsoft.com/vscode/devcontainers/go:0-1-bullseye
   ```

   Note: Make sure your docker daemon is running before using docker command.

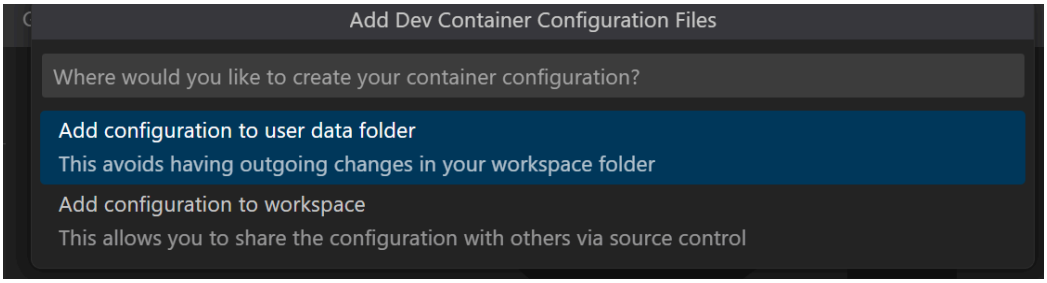
3. Create a Docker container to learn golang using VSCode.
   a. Create an empty folder(say golang) at your desired location(say desktop).
   b. Open the folder in the VSCode.
   c. Reopen this folder in Container by following the steps below:
      i. In VSCode press F1 or fn+F1, depending on your keyboard setting. Type "***reopen***" and from the options listed choose "***Remote-Containers: Reopen in Container***" and click Enter.



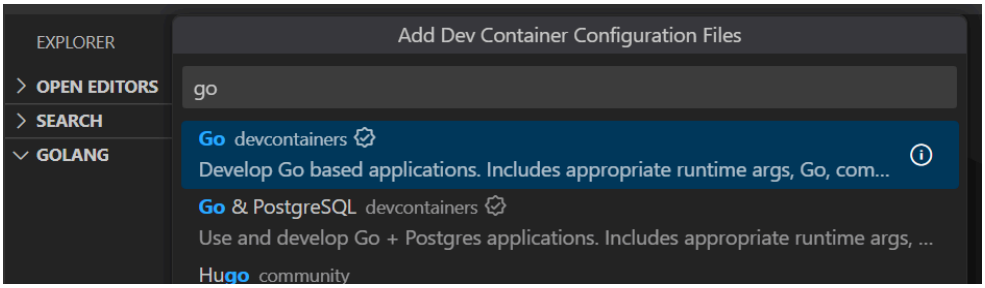
If you don't find the option in the list, go to Extensions menu on the left side and try to install extension **Dev Containers**.

After installing the extension, try to reopen in containers again.

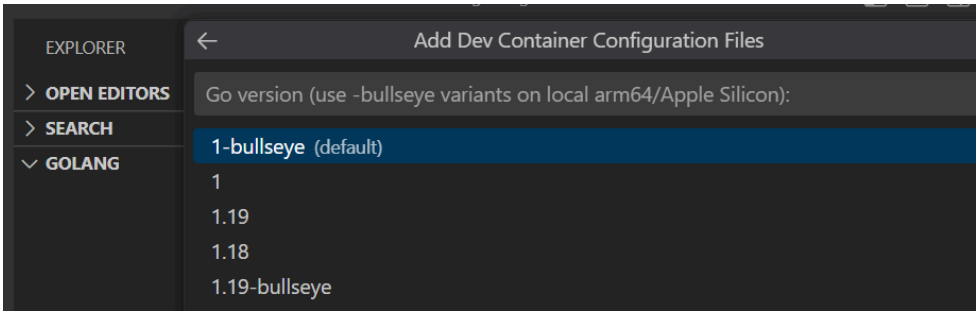
ii. Next, choose the first option.



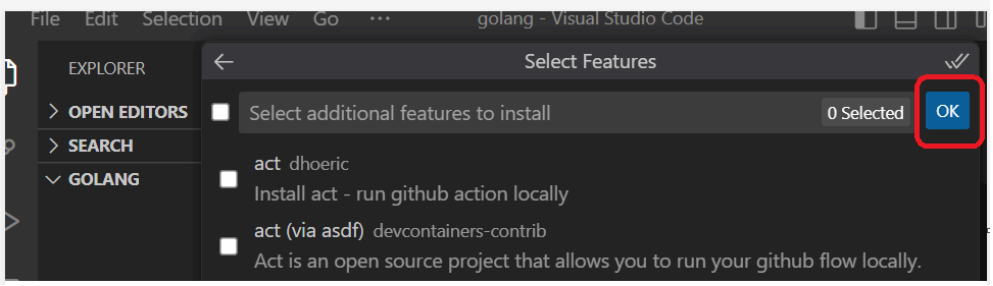
iii. Next, Choose "**Go devcontainers**" from the options listed and click. Scroll the list to find.



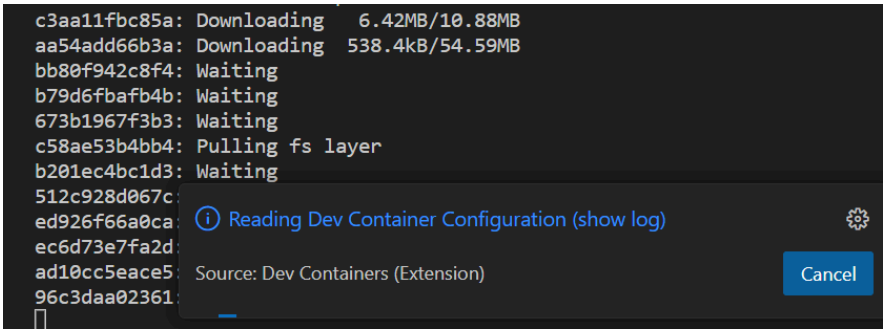
iv. Next, Choose default option "**1-bullseye (default)**"



v. Next, click OK (blue button at the right side)

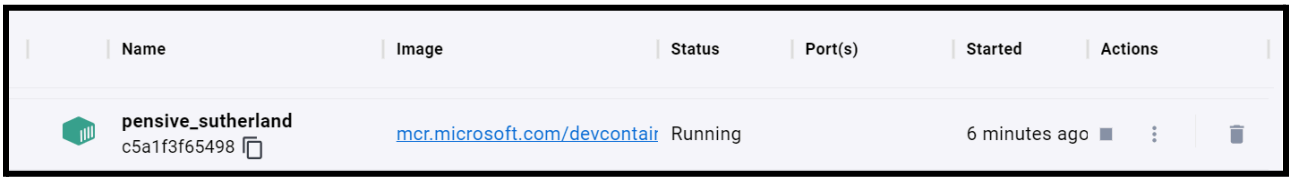


vi. Now, it takes a few minutes to create the container. Click on ***show log*** to see the logs.

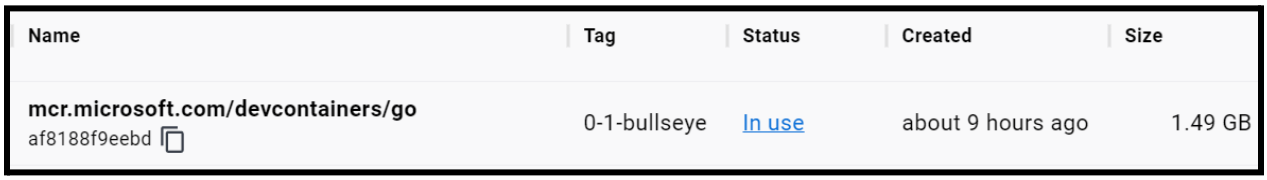


Great! You have successfully created a Container and you are now ready to work inside this container.

- You can also see your container running on Docker. Open Docker desktop app to see this. Your container is ready to use. [Status: running]



- This is the image used to create the above container. [Status: In use]
(Click on the Image side menu to see image list)



Adding SSH keys to GitLab

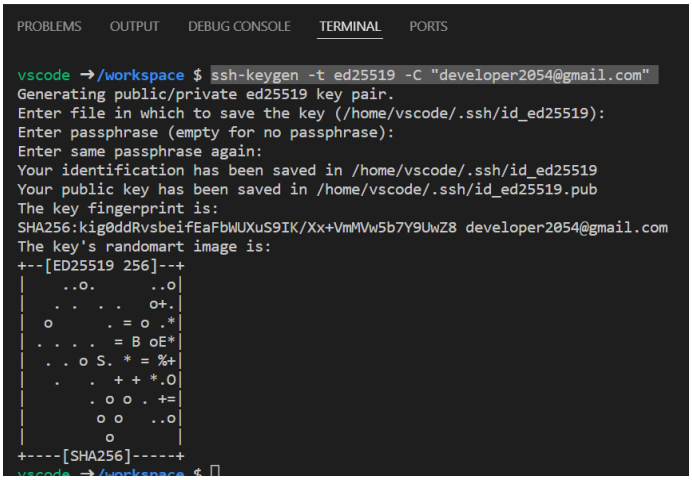
Before creating SSH keys make sure that you have learnt about Git and installed Git on your device. Also create a Gitlab account (Do not create a readme file while creating a new repository).
Refer to the Git&GitLab document (pdf) to learn about Git.

Instructions

1. Create SSH Keys.
    d. Open a new terminal. Type the command below to generate keys.

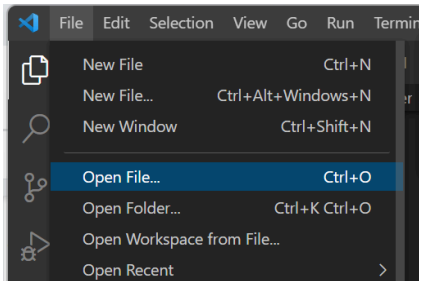
ssh-keygen -t ed25519 -C "developer2054@gmail.com"

To keep all options as default, continue to press enter on your keyboard.

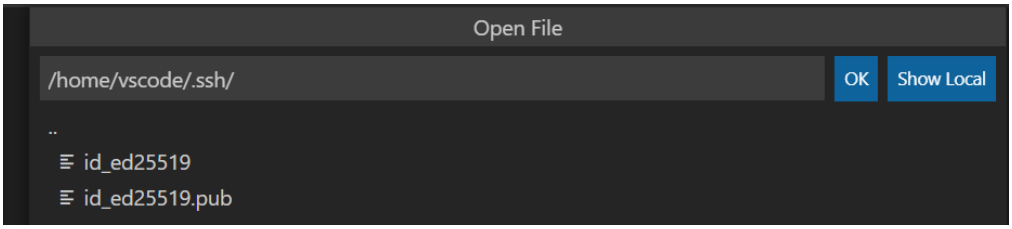


Now, the key is created and stored at the given location (/home/vscode/.ssh).

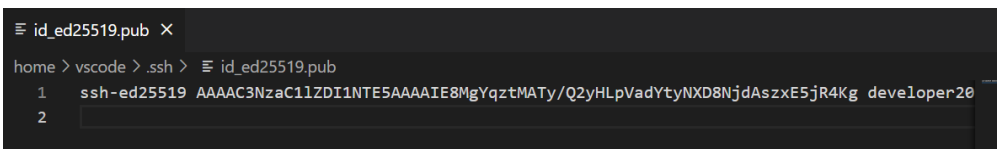
e. Next, go to the **File** menu on the top and select "**Open File**" to open the public key file.



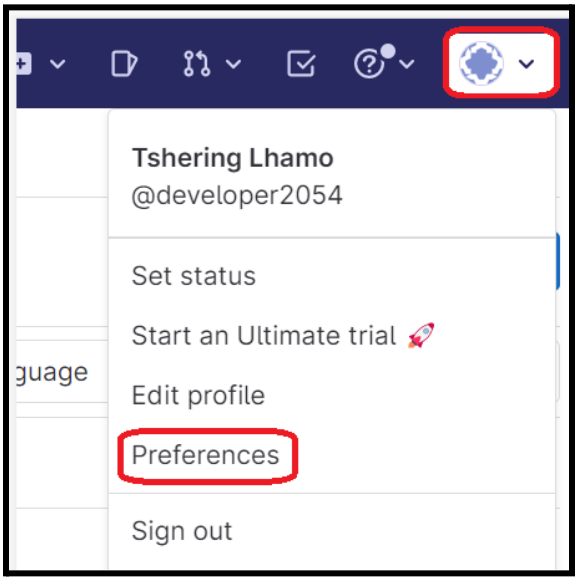
In the input field, enter the path(location of ssh keys created) as shown below and select the *.pub* (public key) file.



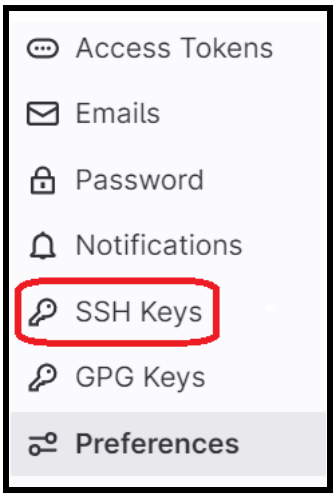
f. Now copy (ctrl+c) this key.



g. Go to your GitLab account to add (paste) the key.
Click on your profile. Select the "**Preferences**" option from the list.



Click on **"SSH Keys**" from the list on the left hand side.

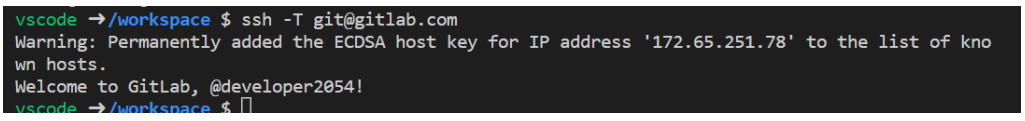


Paste the key in the “**Key”** input field.

Give a key **title** (any name) and click on the “**Add key**” button to finish.

h. Finally, confirm if the connection is successfully established.
Go to VSCode and enter the command below:

ssh -T git@gitlab.com



Great! You have successfully added the ssh key.

Git Set up

Instructions

1. First initialize the git.

```
vscode ➜/workspace $ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /workspace/.git/
vscode ➜/workspace (master ✗) $
```

2. Initialize your git global variables.

```
vscode ➜/workspace $ git config --global user.name "developer2054"
vscode ➜/workspace $ git config --global user.email "developer2054@gmail.com"
vscode ➜/workspace $
```

Use the command below to check the variables:

git config –global --list

```
vscode ➜/workspace $ git config --global --list
user.email=developer2054@gmail.com
user.name=developer2054
```

Great! You have successfully configured git.

3. Clone the Gitlab repository. See the Git&GitLab setup document for the procedure. You can add files to GitLab later on as you write programs.

Time to start learning Go!

Structure of Go Program

Before you learn the Go programming concepts firstly, let us try to understand the basic structure to write a Go program. For this you will write a simple hello world program.

Hello World Application

Instructions

1. Open the Docker Container created in the initial setup process.

In VScode press F1 or fn+F1. Type “reopen” and from the options listed choose “Remote-Containers: Reopen in Container” and click enter.
Ignore this step if it is already opened.

2. Create a root directory, **mygo.**

```
mkdir mygo
```

`mkdir` is a command to make a new directory.

You can also create folders and files using the options shown next to the project directory.

3. Create a Go module. Use the command below.

```
go mod init mygo
```

4. Inside the mygo directory, create another directory **helloWorld**.

```
cd mygo/
mkdir helloworld
```

5. Inside the helloWorld, create a new file ***main.go*** (any name).

```
cd helloworld/
touch main.go
```

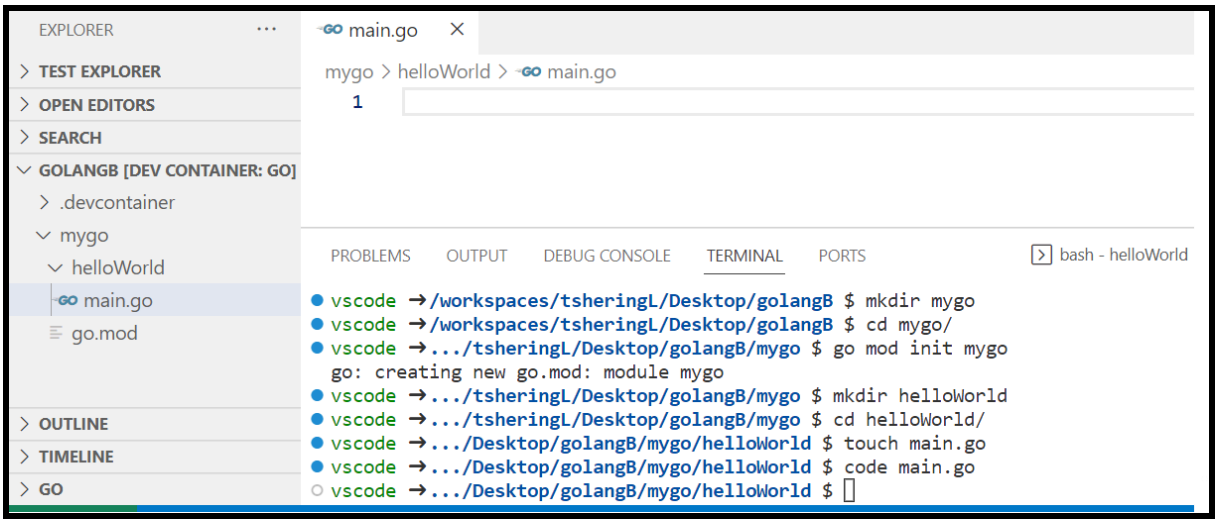
`touch` command is used to create a new file.

6. Open the main file and copy the code below in the file.

```go
package main

import "fmt"

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

This is your first Go program.

- See the steps shown below:



- `code main.go` will open the file in the editor.

Analyze the above code line by line. Discuss basic concepts such as:

1. What does ‘package main’ mean?
2. What does ‘import fmt’ mean?
3. What is that ‘func’ thing?

Let’s look into the above questions one by one.

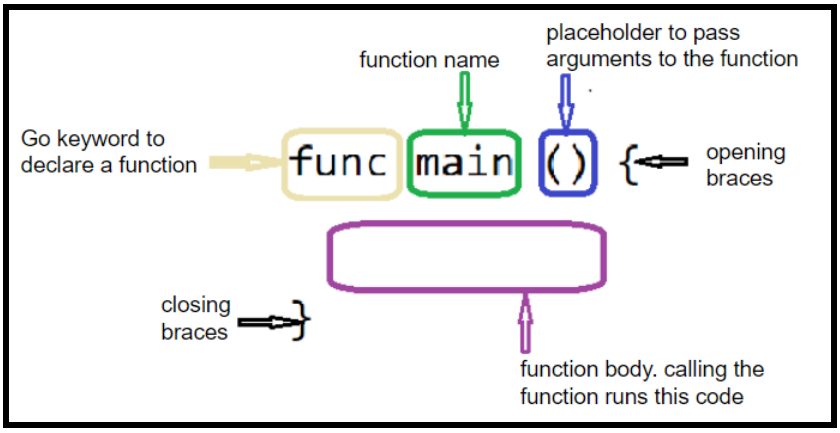
`package main`

- Every Go program is organized in packages. Each .go file first declares which package it is part of. The above program is part of package main.
- main is actually a special package name which indicates that the package contains the code for an executable application. That is, it indicates that the package contains code that can be built into a binary and run.
- The **main** package is the **entry point** of the program and identifies an **executable** program.
- A package can be composed of multiple files, or just one file.
- A program can contain multiple packages.

`import fmt`

- Use the **import** keyword to import a package.
- **`import fmt`** imports the fmt (formatting) package. The formatting package exists in Go's standard library and provides input/output utility functions.
- You can read about all the features that fmt package provides on the official documentation.

`func main(){ }`



- **`func`** is the keyword used to declare a function and **main** is the **name** of the function..

  What is a function?
  You'll see more about it later, but in the meantime let's say a function is a block of code that's assigned a name, and contains some instructions.
- The opening curly brace ({) indicates the beginning of the function code, and the closing curly brace (}) indicates the end of the function.
- The **main** function is a special function. Is the **entry point** of an executable program. Go automatically calls the main function so, there is no need to call the main function explicitly and every executable program must contain a single main package and main function.
  It doesn't contain any parameter nor return anything.
- **main function** can be only defined inside the **main package**. A main package **must** contain **only** one function main (not more than one).
- In this simple case you just have one function, main; the program starts with that and then ends.

`fmt.Println("Hello World")`

- This is the content (body) of the function defined.
- You call the Println() function defined in the fmt package that is imported earlier by passing string as a parameter.
- The "dot" syntax fmt.Println() specifies that the function (Println()) is provided by that package (fmt).

This is the basic structure of every Go program.

After the main function is executed, it has nothing else to do and the execution ends or the program terminates.

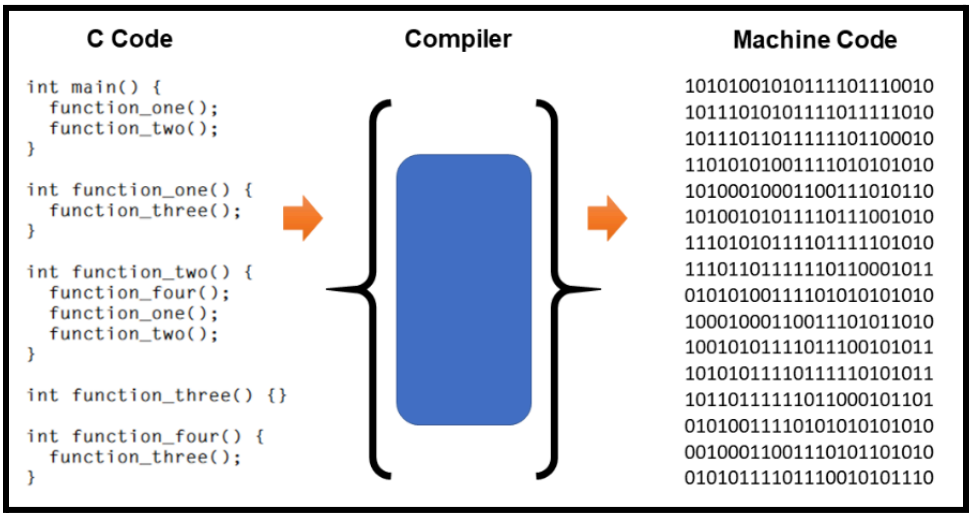
How to Compile and Run a Go program?

What does **compile** mean?
Computers need machine code which means they don't understand English or even Go. Computers don't know how to do anything unless you as a programmer tell them what to do. You need to convert high-level (Go) code into **machine language**, which is really just a set of instructions that some specific hardware can understand. In your case, your CPU.

The **Go compiler's** job is to take Go code and produce machine code. On Windows, that would be a **.exe** file. On Mac or Linux, it would be any executable file.

Go, C, and Rust are all languages where the code is first converted to machine code by the compiler before it's executed.
Machine code consists of binary zeros and ones only.

The image below shows a c program being converted to machine code using a compiler.



There are **two ways** to execute a Go program.
You can compile first and run the executable file (binary code) in two separate steps or you can compile and run at the same time.

1. `go build main.go` - compiling the code
   `./main` - executing the compiled code

2. `go run main.go` - compile and execute together

Compile and execute go program on local environment (Optional)

To compile first use the command `go build`. This command creates an executable (binary code) file which can be executed to get the output of the program.

```
go build main.go
```

Run `ls` command to see the new executable file (**main.exe)** created.

```
tsheringL@MYpc MINGW64 ~/Desktop/helloworldapp (master)
$ go build main.go

tsheringL@MYpc MINGW64 ~/Desktop/helloworldapp (master)
$ ls
main.exe*  main.go
```

Run the executable file to see the output.

```
tsheringL@MYpc MINGW64 ~/Desktop/helloworldapp (master)
$ ./main
Hello World
```

To compile and execute together use command **`go run.`**

```
go run filename.go
```

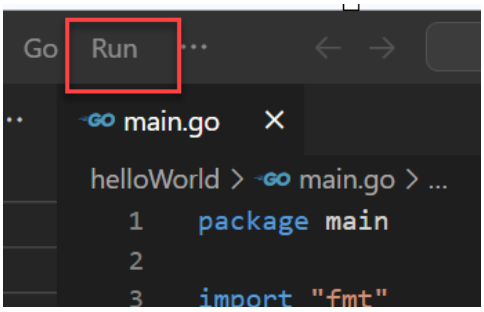
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

tsheringL@MYpc MINGW64 ~/Desktop/golang/helloworldapp (master)
$ go run main.go
Hello World
```

Compile and execute Go program in Docker environment

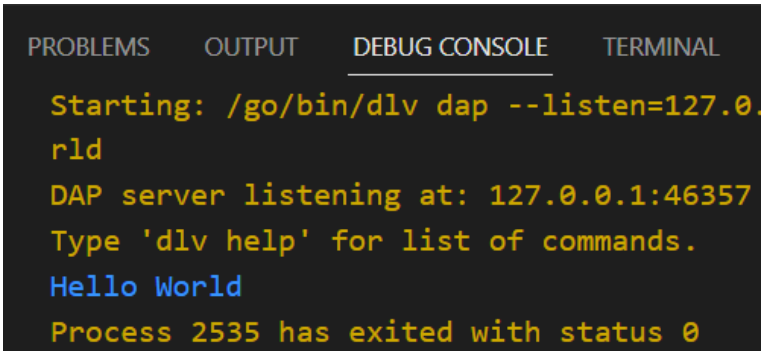
- Use command:

```
go run filename.go
```

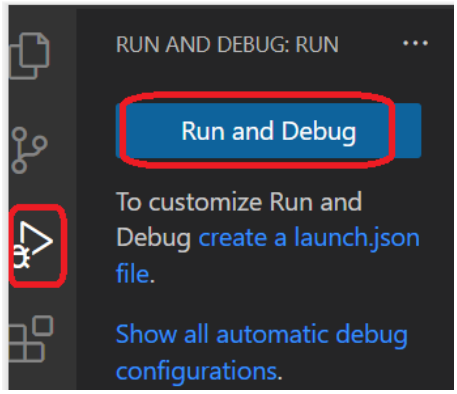
- (OR) Click on the **Run** menu on the top. Select **Start Debugging** option from the drop down list. You can use shortcut key, F5 (may be fn + F5 on your pc)



  - You will see the output in the **Debug Console** as shown below.



- (OR) Click on the **Run and Debug** menu at the left hand side and again click on **Run and Debug** button to run the code.



  - You will see the same output in the Debug Console.

Compiled vs Interpreted Languages

Every programming language is categorized as either compiled, interpreted, or both.

What does it actually mean?
Imagine you have a recipe to make a pizza, but it's written in Chinese. There are two ways you, a non-Chinese speaker, could follow these directions.

The first is if someone has already translated it into English for you. You could read the English version of the recipe and make pizza. Think of this translated recipe as the **compiled** version.

The second way is if you have a friend who knows Chinese. When you make pizza, your friend sits next to you and translates the recipe into English as you follow line by line. In this case, your friend is the interpreter for the **interpreted** version of the recipe.

[See the video here.]

Compiled Languages

- Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be faster and more efficient to execute than interpreted languages.
- Compiled languages need a "**build**" step – they need to be manually compiled first. You need to "rebuild" the program every time you need to make a change to the source code.
- Examples of compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

Interpreted Languages

- Interpreters run through a program line by line and execute each command.
- Interpreted languages were once significantly slower than compiled languages. But, with the development of just-in-time compilation, that gap is shrinking.
- Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript.

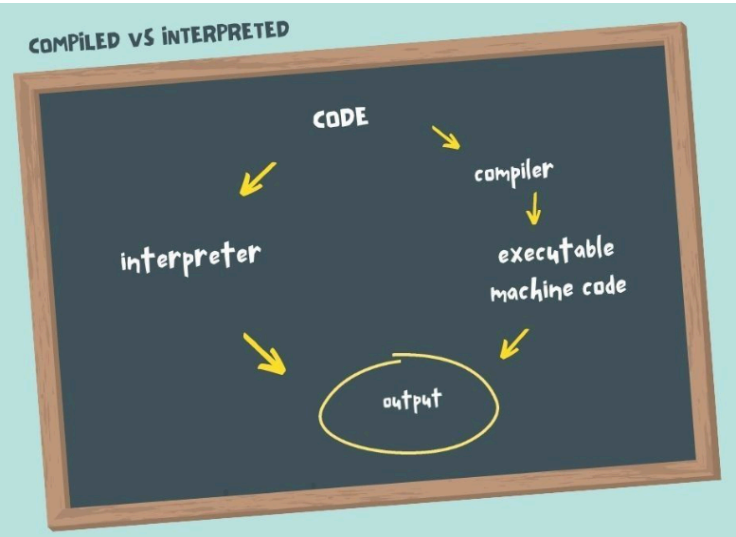
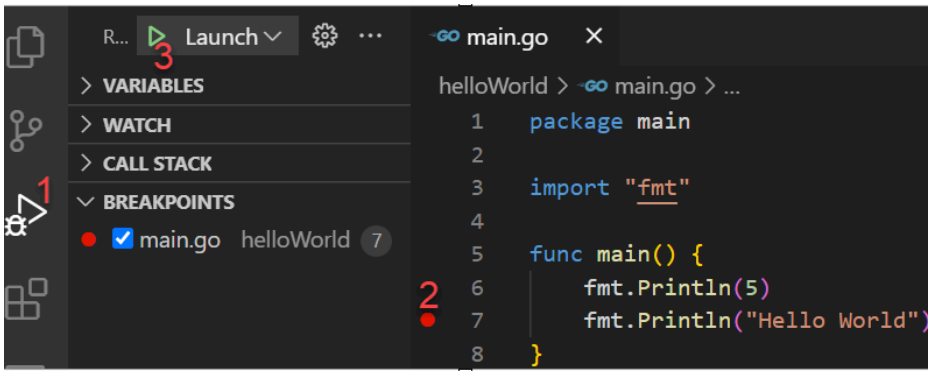


Figure: Illustration of compiled vs interpreted languages

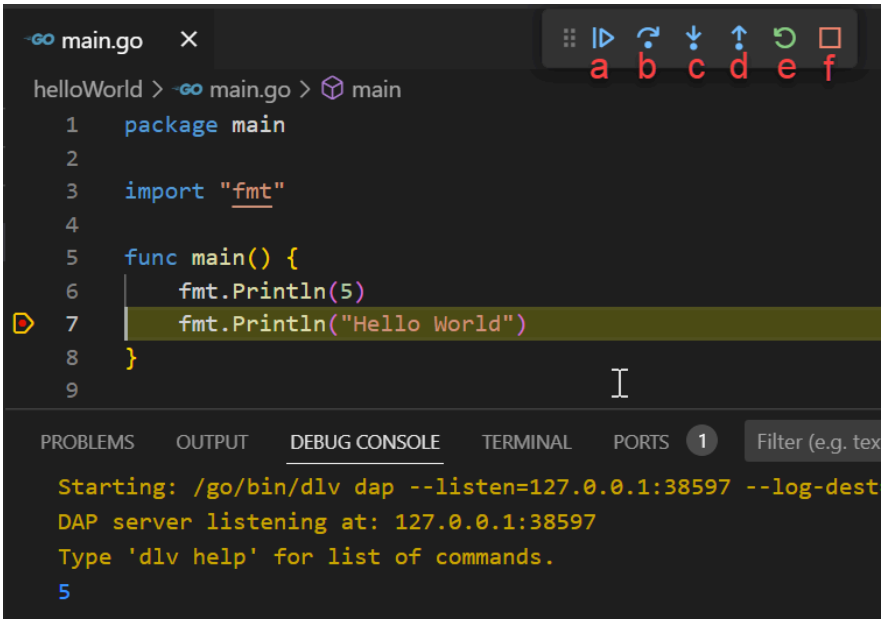
Java is considered as both a compiled and interpreted language.

Q. How to debug the code when there is an error?
Run the code in the debugging mode. Go to the run menu on the top and select the “Start Debugging” option.

➔ Place a breakpoint, red dot (line#7).
(Added a new line to print 5).
➔ Run the program. Click on the play button in green.



- The output printed is only 5. Why?
- Breakpoint **stops** the execution of the program at the line where you placed the breakpoint.



- You will notice six new buttons towards the top. Here it is labeled using alphabets.
- Use these buttons to debug the code.

- a - Continue
  Continue running the program from the breakpoint.

- b - Step Over
  Execute the current line and step over to the next line.

- c - Step Into
  Step into the underlying code.

- d - Step Out
  Step out of the underlying code.

- e - Restart
  Restart the debug process.

- f - Stop
  Terminate the debug process.

Note: Don't forget to remove the breakpoint.

Go Packages

A package in Go is essentially a named collection of one or more related **.go** files. In Go, the primary purpose of packages is to help you **isolate** and **reuse** code.

Every .go file that you write should begin with a **package {name}** statement which indicates the name of the package that the file is a part of.
For example, in your hello world application, the **package main** line declares that the **main.go** file is part of a package named **main**. The application consists of one package, with the package name **main**. The main package is made up of one file, with the filename **main.go.**

Remember that a code in a package can access and use all types, constants, variables and functions within that package — even if they are declared in a different .go file.

In the example below, there are two new files created in the application. Both the files belong to the package main (because they both start with a package main statement). Since all the files belong to the same package, you can directly call functions defined in a separate file.

hello world Application

main.go	helper.go	support.go

`package main`
 	
`import “fmt”`
	`package main`
	
	`import “fmt”`
		`package main`
		
		`import “fmt”`

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

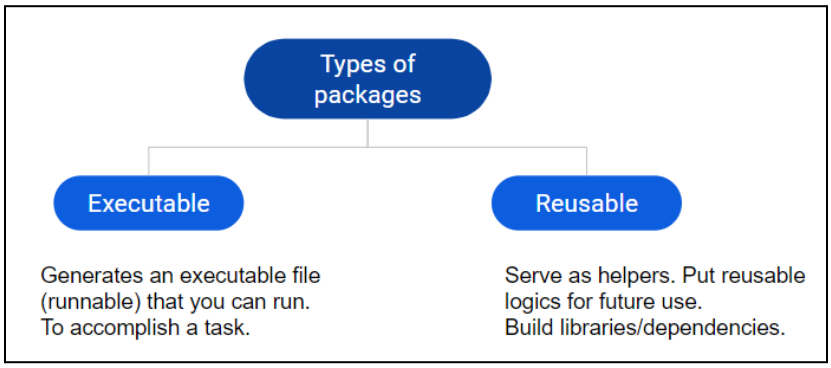
To run all three files, use the command:

```
go run *.go
```

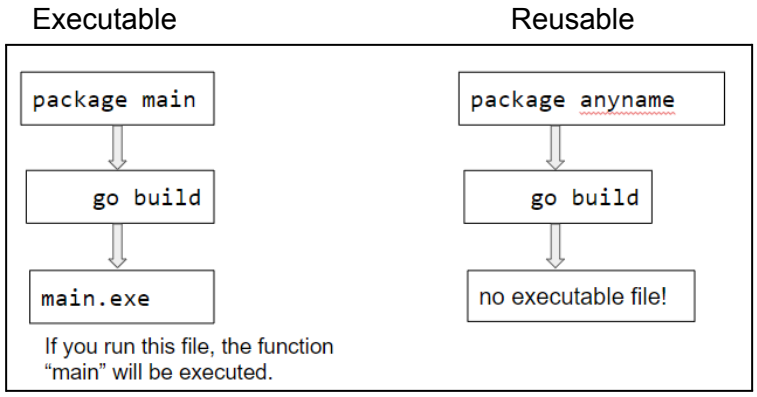
You will see output as:

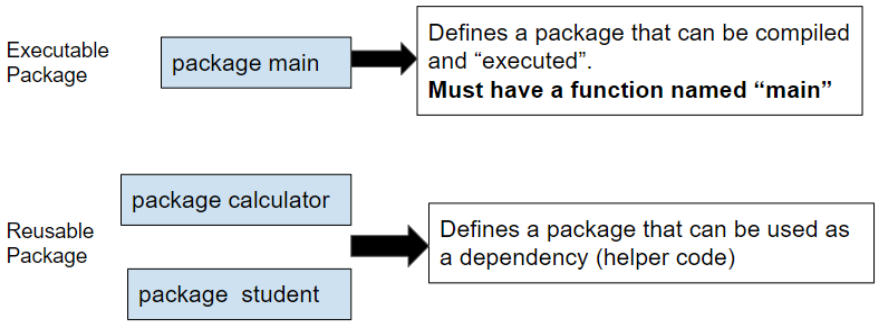
Hello World
helper here
support here

Two types of Packages in Go



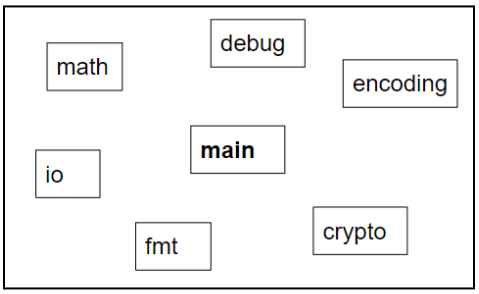
Example:



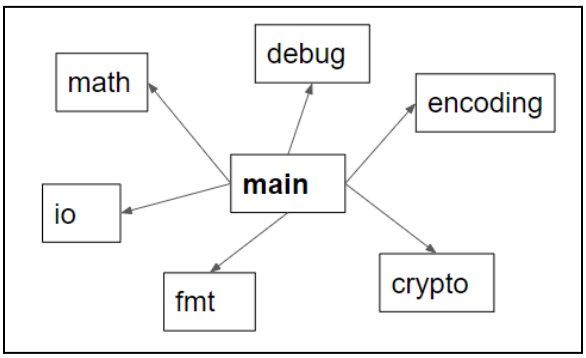


To understand how packages work, see the simple example below.

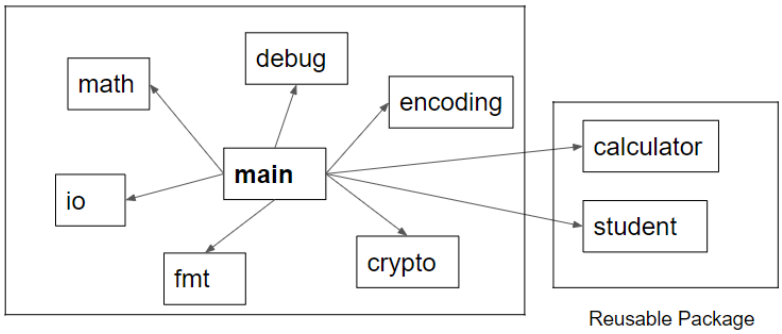
- The package **main** by default has no access to any other packages.



To get access to other packages, you have to exclusively import other packages in main. Remember that main is not a reusable package. (Cannot be imported by other packages)



Apart from the standard library packages, you can also import user defined reusable packages.



#Learn more about golang standard packages at ***golang.org/pkg***

Note:

- In Go it is not compulsory to import built-in packages explicitly. As you use the package name in your program and save the file, Go automatically imports the package name in the import list.
- Shorthand to define package main and main function,
  Type ***pkgm*** and press enter.
- It is not required to put a semicolon at the end of each statement.
- In the program, you can **hover** over the function names to see the description of functions defined by go packages.

Go Modules

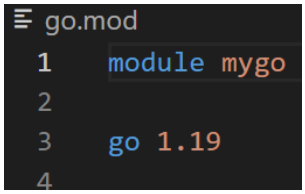
If you have a small application which only imports packages from the standard library, then no need to create modules. But if you want to import and use a third-party package — or structure your code so it's split into multiple packages — then you first need to turn your code into a Go module.

A **module** is a collection of related **Go packages** that are versioned together as a single unit. Module acts like a **root directory** for your Go packages.

In your hello world application, the mygo directory is made into a module by running the command:

```
go mod init modulePath

Ex. go mod init mygo
```

The above command creates a new file go.mod at the root of the module.



The go mod init command creates a **go.mod** file to track your code's dependencies (third-party packages). For now, the file includes only the name of your module and the Go version your code supports. But as you add dependencies, the go.mod file will list the versions your code depends on. This keeps builds reproducible and gives you direct control over which module versions to use.

Go commands

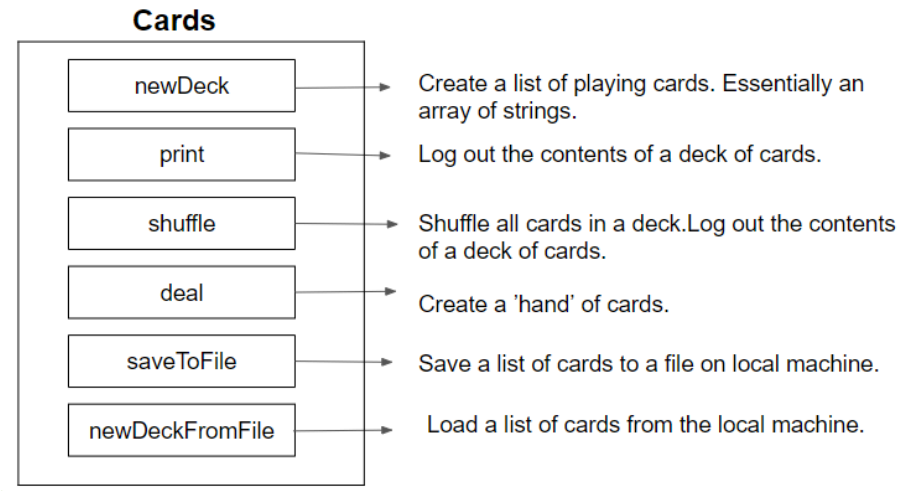
- Commonly used go commands available in **go CLI** are:
  go build - Compile only. Creates an executable file (Ex. main.exe) from the source code. Then run the executable file to see the output.

  ```
  go build filename.go
  ```
  On windows: ./main.exe
  On mac: ./main

  go fmt - Formats all the code in each file in the current directory.
  go install - Compiles and installs a package.
  go get - Downloads the raw source code of someone else's package.
  Both the commands above are to handle dependencies.
  go test - Runs any tests associated with the current project. Run and execute test files.

Go Fundamentals

You are going to learn the Go programming concepts through a simple project, **playing cards**. You can find the project overview shown below. You will create a set of functions to perform a specific task in a card game.



To start with, follow the instructions given below to set up the project.

1. Create a new project directory called **'cards'**.
2. Inside the cards directory, create a new file **main.go**.
3. Define package main, and create main function. (use shorthand `pkgm` and press enter.)

Identifiers in Go

In programming languages, identifiers are used for identification purposes. In other words, identifiers are the **user-defined names** of the program components.
In the Go language, an identifier can be a variable name, function name, constant, statement label, package name, or type.

In the hello world program, there are two identifiers.

main: name of the package
main: name of the function

There are certain valid rules for defining a valid Go identifier. These rules should be followed, otherwise, you will get a compile-time error.

#Explore the identifier naming rules on your own.

Go Keywords

Keywords or Reserved words are the words in a language that are used for some internal process or represent some predefined actions.
You are not allowed to use keywords as an identifier otherwise it will result in a compile-time error.

Some of the Go keywords are listed below.

```
break     default      func    interface   select
case      defer        go      map         struct
chan      else         goto    package     switch
const     fallthrough  if      range       type
continue  for          import  return      var
```

Go Comments

A comment is a text that is ignored by the compiler upon execution.
Comments can be used to explain the code, and to make it more readable.
Comments can also be used to prevent code execution when testing an alternative code.
Go supports **single-line** or **multi-line** comments.

- Single-line comments start with two forward slashes (//). Any text between // and the end of the line is ignored by the compiler.
- Multi-line comments start with /* and ends with */. Any text between /* and */ will be ignored by the compiler.

  See the example below.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       fmt.Println("Hello World")
7       // fmt.Println("Hello world 2")
8       /*
9           fmt.Println("Hello world 3")
10          fmt.Println("Hello world 4")
11      */
12  }
```

```
DEBUG CONSOLE   PROBLEMS   OUTPUT   PORTS   TERMINAL

tsheringL@MYpc MINGW64 ~/Desktop/helloworldapp (master)
$ go run main.go
Hello World
```

Go Variables

One of the first things you do in a programming language is defining (creating) a variable.

- **Variable** is a **placeholder** of any information/data used in a program.
- Variables are the **names** given to a **memory location** where the actual data is stored.

There are two ways to declare variables.

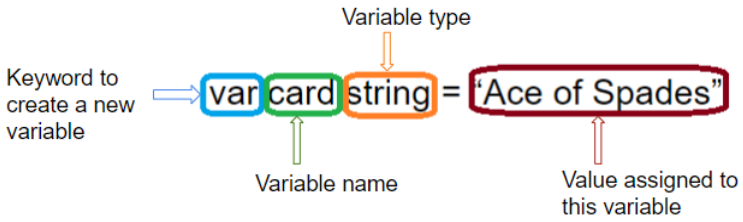
1. With the `var` keyword
   Using the var keyword there are two ways to declare a variable.

   Syntax:

   ```
   var variable_name type = expression [explicit declaration]

   var variable_name = expression [implicit declaration]
   ```

   Explicit: Type is declared explicitly
   Implicit: Type depends on what value is stored in a variable

➔ In your main.go file declare a variable card and assign a string type as card value.



Code:

```go
package main

import "fmt"

func main() {
    var card string = "Ace of Spades"
    fmt.Println(card)
}
```

- To make Go code easier to read, the variable type comes after the variable name.

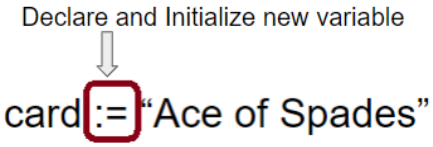
- Every variable defined **must** be used. Use the card variable by printing the value.

2. With the := sign. Also known as shorthand declaration.
   Syntax:

```
variablename := value
```

In this case, the type of the variable is **inferred** from the value (means that the compiler decides the type of the variable, based on the value). Same as implicit declaration. Commonly used for **local variables** which are declared and initialized within the functions.

➔ Declare the card variable using shorthand declaration.




```go
package main

import "fmt"

func main() {
    var card string = "Ace of Spades"
    fmt.Println(card)
    card := "Ace of Spades"
}
```

Q. What does the error say?
Same variable **name** cannot be initialized twice.

Important: Go language is **case sensitive**.

Example:

```
var Card string
var card string
```

Variables card and Card are considered as two separate variables. Same applies for function names and package names.

In summary:

var card string = “Ace of Spades”	variable declaration and initialize
var card = “Ace of Spades”	variable declaration and initialize
card := “Ace of Spades”	variable declaration and initialize
card = “Five of Diamonds”	assign new value to existing variable

Declaring Multiple variables

In Go, it is possible to declare multiple variables in the same line.

➔ Declare two new variables card1 and card2 of type string in a single line.

```
func main() {
    var card string = "Ace of Spades"
    fmt.Println(card)
    // card := "Ace of Spades"
    var card1, card2 string = "Ace of Diamond", "Six of hearts"
}
```

Q. What does the error say?

- If the variable type is not specified, you can declare different types of variables in the same line.

Example:

```
func main() {
    var card string = "Ace of Spades"
    fmt.Println(card)
    // card := "Ace of Spades"
    // var card1, card2 string = "Ace of Diamond", "Six of hearts"
    var card1, card2 = "Ace of diamond", 8
}
```

Go Constants

If a variable should have a **fixed** value that cannot be changed, you can use the **const** keyword. The const keyword declares the variable as "constant", which means that it is **unchangeable** and **read-only**.

Syntax:

```
const constname type = value
```

#Explore the Go variables naming rules on your own.

Basic Data Types

Data type is an important concept in programming. Data type specifies the size and type of variable values.
Go has three basic data types:

- Boolean: represents a boolean value and is either true or false
- Numbers: represents integer types, floating point values, and complex types
- Strings: represents a string value

Boolean Data Type

A boolean data type is declared with the **`bool`** keyword and can only take the values **true** or **false**.

The **default value** of a boolean data type is **false**.

Integer Data Type

Integer data types are used to store a whole number without decimals, like 35, -50, or 1345000.

The integer data type has two categories:

- Signed integers - can store both positive and negative values
- Unsigned integers - can only store non-negative values

Signed Integers

Signed integers, declared with one of the **`int`** keywords.

Go has five keywords/types of signed integers:

Type	Size	Range
int	Depends on platform: 32 bits in 32 bit systems and 64 bit in 64 bit systems	-2147483648 to 2147483647 in 32 bit systems and -9223372036854775808 to 9223372036854775807 in 64 bit systems
int8	8 bits/1 byte	-128 to 127
int16	16 bits/2 byte	-32768 to 32767
int32	32 bits/4 byte	-2147483648 to 2147483647
int64	64 bits/8 byte	-9223372036854775808 to 9223372036854775807

Sometimes programmers also use aliases like **rune** (int32) and **byte** (uint8).

Unsigned integers
Unsigned integers, declared with one of the **`uint`** keywords.

Go has five keywords/types of unsigned integers:

Type	Size	Range
uint	Depends on platform: 32 bits in 32 bit systems and 64 bit in 64 bit systems	0 to 4294967295 in 32 bit systems and 0 to 18446744073709551615 in 64 bit systems
uint8	8 bits/1 byte	0 to 255
uint16	16 bits/2 byte	0 to 65535
uint32	32 bits/4 byte	0 to 4294967295
uint64	64 bits/8 byte	0 to 18446744073709551615

Note: The type of integer to choose, depends on the value the variable has to store.

Float Data Types

The float data types are used to store positive and negative numbers with a decimal point, like 35.3, -2.34, or 3597.34987.

The float data type has two keywords:

Type	Size	Range
`float32`	32 bits	-3.4e+38 to 3.4e+38.
`float64`	64 bits	-1.7e+308 to +1.7e+308.

String Data Type

The **string** data type is used to store a sequence of characters (text). String values must be surrounded by double quotes.

Example:

Type	Values
bool string int float64	True (1), False (0) “Golang”, “How was your day?” 0, -10000, 99999 10.000001, 0.00009, -100.002

Zero Values

If you declare a variable without assigning any value to it, the value of the variable will be set to **zero value** of its type.

Given below are zero values of some basic types in Go.

Type	Zero Value
int bool string	0 False “”

Statically and Dynamically Typed Languages

Go is strongly typed. What does it mean?
Go enforces **strong** and **static typing**, meaning variables can only have a **single** type.
A string variable like "hello world" can not be changed to an int, such as the number 3.

On the contrary, some languages are **dynamically typed**.

Examples of the dynamic and static languages are:

Dynamic Types	Static Types
Python Javascript PHP Ruby	Java C C++ Go

Example:

Using Go

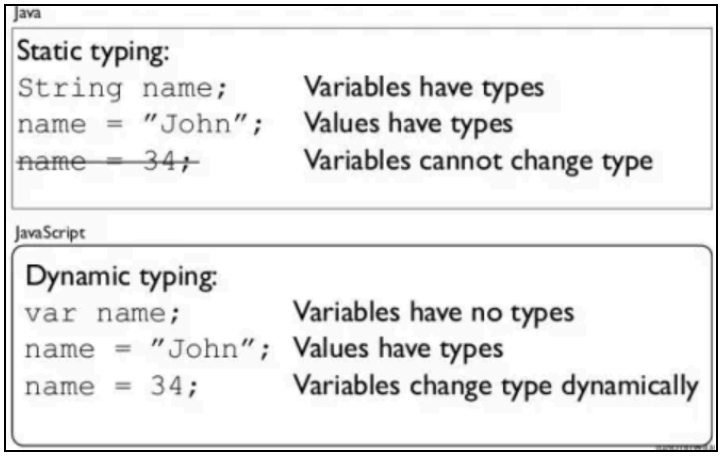
```
var card string = "Ace of Spades"
card = 8
```

What does the error say?

Using JavaScript (See demo in browser console)

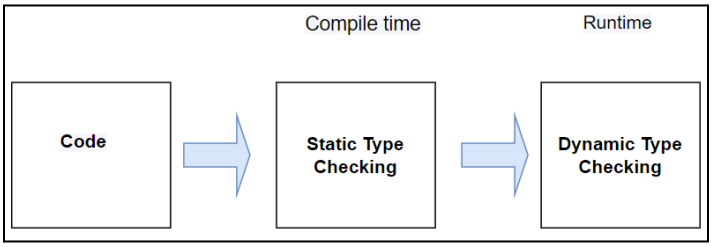
```
> var card = "Ace of Spades"
<· undefined
> console.log(card)
  Ace of Spades                    VM176:1
<· undefined
> card = 8
<· 8
> console.log(card)
  8                                VM244:1
```

Example between Java and JavaScript:



You can also differentiate them as follows:

Attribute	Dynamic Types	Static Types
Definition	If the type of the variable is checked at the **runtime** of the program then the language is called a **dynamically typed language**.	If the type of the variable is checked at the **compile-time** of the program then the language is called a **statically typed language**.
Variable declaration	Variable type is not specified.	Variable type is specified.
Type Checking	Runtime	Compile time

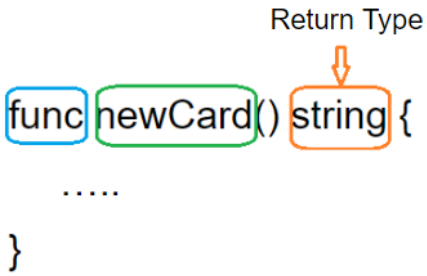


One of the biggest benefits of strong typing is that errors can be caught at "compile time". In other words, bugs are more easily caught ahead of time because they are detected when the code is compiled before it even runs.

Functions and Return Types

Function is a block of **reusable codes** that perform a **specific task.**

- ➔ Create a new function (newCard) that returns a value for a card variable rather than assigning it directly in the main.



Code:

```
package main

import "fmt"

func main() {
    card := newCard()
    fmt.Println(card)
}

func newCard() string {
    return "Ace of Spades"
}
```

- Function definition consists of the following:

```
func function_name(Parameter-list) (Return_type) {
    // function body.....
}
```

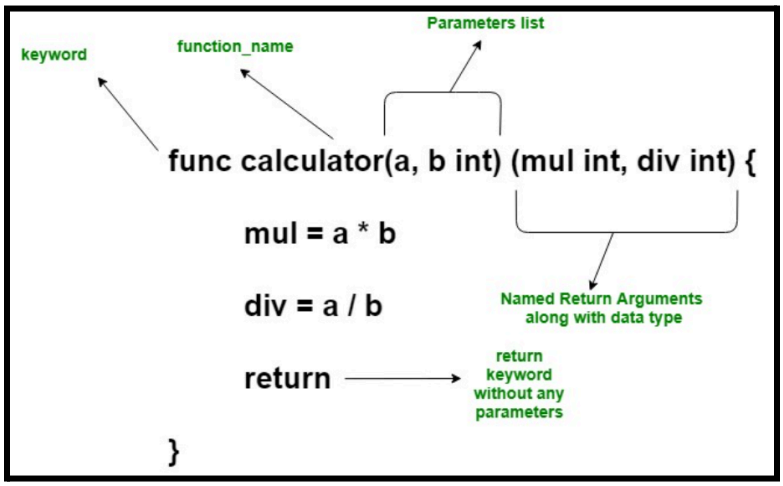
  - **`func:`** Is a keyword in Go used to create a function.
  - **function_name**: Is the name of the function.
  - **Parameter-list:** It is optional. There may be zero or more parameters.
    It must be enclosed within the angular brackets.
    Each parameter is declared with a name and the type of value it can hold.

  - **Return_type**: It is optional depending on whether you want your function to return a value or not. It contains the type of the value that function returns. A function can have **more than one** return type.

    If a function has a **return_type**, it is **mandatory** to have a **return** statement in the function.

Named return values
In Go, you can name the return values of a function.
If the return_type is **labeled (named)**, there is no need to specify the variable names explicitly in the return statement.

Example of a named return type:



If there is only one return type, no need to enclose the return type in the angular brackets.

```
func newCard() string {
    return "Ace of Spades"
}
```

`func function_name(Parameter-list) (Return_type)` is known as **function signature.**

```
Ex., func  newCard() string
```

How did the function (newCard()) get executed?
Functions are not executed immediately. They are "saved for later use", and will be executed when they are **called.**
Any defined function must be **called** to perform the task. Otherwise, the function will never be executed even when the program runs.

In the example above, we create a function named "newCard()". The function returns a string value "Ace of Spades". To call the function, just write its name followed by two parentheses ():

```
card := newCard()
```

If a function has a **return** type then only you can **store the return value in a variable** or send as an **argument** to another function.

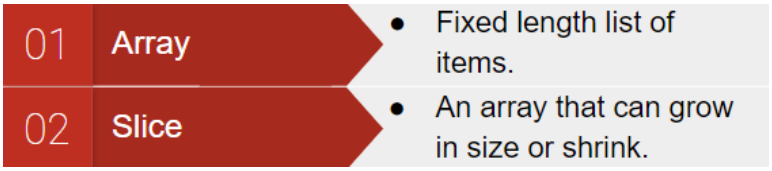
card is a variable which stores a value that is returned from the function newCard().

Q. What is the type of the value stored in the card variable?

Arrays and Slices

So far you created only one card. To work with a deck of cards how can you create 52 different cards?
It would be a waste of memory to create 52 different variables to store 52 different cards.

You can use **arrays** and **slices** in Go to store a collection of data.



Array

Is a **fixed length** sequence that is used to store **multiple** values of the **same** type in a single variable.

Like variable declarations, you also need to declare an array to use.

In Go, there are two ways to declare an array.

1. With the var keyword

   Syntax:

   ```
   var array_name[length]Type  // no values assigned only declaration.

   or

   var array_name = [length]datatype{values} // here length is defined.
   Declared and values initialized

   or

   var array_name = [...]datatype{values}  // here length is inferred
   ```

   Example:
   ```
   var myArr1[]int
   var myArr2 = [4]int{1, 2, 3, 4}
   ```

```
var myArr3 = [...]int{1, 2, 3}
```

- The number of items in an array is called **length/size** of an array.

2. With the := sign

Syntax:

```
array_name := [length]Type{} // not initialized

or

array_name := [length]datatype{values} // here length is defined

or

array_name := [...]datatype{values} // here length is inferred
```

Example:
```
myArr1 := [4]int{}
myArr2 := [4]int{1, 2, 3, 4}
myArr3 := [...]int{1, 2, 3}
```

- Each value in an array has an **index** which starts from **zero**.

Example: `myArr2 := [3]int{1, 2, 3}`

1	2	3
0	**1**	**2**

Index:

You can also find the length of the array using built-in **len()**.

The **first** index is always **zero** and the **last** index is **one less** than the length of an array.

Q. How can you access/retrieve the values stored in an array?
With the help of array index.

Syntax:

```
array_name[index]
```

Example: To print value 3 in myArr2

```
fmt.Println(myArr2[2]) —-- 1
fmt.Println(myarr2[2]) —-- 2
```

Q. Which of the above statements is correct? Why?

- Arrays do not need to be initialized explicitly; the **zero value** of an array is a ready-to-use array whose elements are themselves zeroed

Example:

```
myArr1 := [3]int{}
fmt.Println(myArr1)
```

Q. What is the output of the above print statement? How?

Slice

Slices are similar to arrays, but are more **powerful** and **flexible**.
Like arrays, slices are also used to store multiple values of the same type in a single variable.

However, unlike arrays, the length of a slice can grow and shrink as you see fit. Thus, slices are called **variable length arrays**.

See the example below.

Slice 1

“Five of Spades”	“Three of Diamonds”	“Five of Diamonds”

Slice 2

“Five of Spades”	123456	“Five of Diamonds”

Slice 2 is not a valid slice type. Why?

In Go, there are several ways to create a slice:

1. Using the [ ]datatype{values} format
2. Create a slice from an array
3. Using the make() function

Using the [ ]datatype{values} format

Syntax:

```
slice_name := []datatype{values}
```

Example: `myslice := []int{}`

- A slice is declared just like an array, except that you leave out the length.

➔ Create a slice of type string to store a list of cards.

```
cards := []string {“Ace of Diamonds”, “Five of Spades”,
                   “Six of Hearts”}
```

Creating slice from an Array

A slice can also be formed by **“slicing”** an existing slice or array. Slicing is done by specifying a half-open range with two indices separated by a colon.

Syntax:

```
var myarray = [length]datatype{values}
myslice := myarray[start:end]
```

Start and end are the **indexes**. Start represents the starting index in the array from where you want to slice. End represents the end index where you want to stop.

Start index is **inclusive** whereas the end index is **exclusive**.

Example:

```
myarr := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
myslice := myarr[1:3]
fmt.Println(myslice)
```

Guess the output?

The start and end indices of a slice expression are optional; they default to zero and the slice’s length respectively:
Which means:

```
myarr[:2] == []byte{'g', 'o'}
myarr[2:] == []byte{'l', 'a', 'n', 'g'}
myarr[:] == myarr
```

Using make() function

A slice can be created with the built-in function called **make**, which has the signature,

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

T stands for the element type of the slice to be created.
The make function takes a **type**, a **length**, and an optional **capacity**. When called, make allocates an array and returns a slice that refers to that array.

Length is the number of elements in the slice.
Capacity is the number of elements in the underlying array.

Syntax:

```
slice_name := make([]type, length, capacity)
```

Example:

```
var s []int
s = make([]int, 5, 5)
// s == []int{0, 0, 0, 0, 0}
```

Unlike array, slice is a **variable** length array, which means you can add new items to your slice.
Q. How to add a new element to a slice?

- Use the **append** function to add new items to slice.
- Append does not modify the existing slice rather adds data at the **end** of the slice and **returns** a **new** slice.

➔ Add a new card, “Six of Spades” to cards.

```
cards = append(cards, “Six of Spades”)
```

Code:

```
package main

import "fmt"

func main() {
    cards := []string{"Ace of Diamonds", newCard()}
    cards = append(cards, "Six of Spades")
    fmt.Println(cards)
}

func newCard() string {
    return "Five of Diamonds"
}
```

Q. How to access each item in the slice?
Like an array, each item in the slice has an **index**(integer) which starts from **0**.
To access items in the slice, you can use slice index.
Use **loops** to Iterate through each item in the slice.

#Add files to the gitlab repository. See the Git&GitLab setup document for the procedure.

Loops in Go

Go has only one looping construct, the **for loop**.

For Loop

The for loop iterates through a block of code a specified number of times.

Syntax:

```
for initialization; condition; post {
  // code to be executed for each iteration
}
```

Initialization: count at which the loop starts
Condition: Condition is checked before each iteration and decides whether the loop continues or ends. If the condition is true, the loop continues otherwise the loop ends.
Post: Increases or decreases the loop counter depending on the order you want. It is run **after** each iteration.

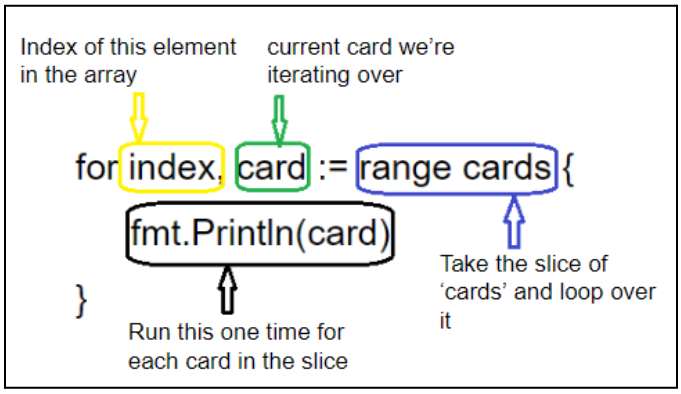
Example: Print each item in the card variable.

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

For loop using Range Keyword

- The **range** form of the for loop iterates over a slice or map (you will learn later).
- When ranging over a slice, **two** values are returned at each iteration. The first is the **index**, and the second is a copy of the **element** at that index.

➔ Iterate through the cards and print each item in the card.



Code:

```go
package main

import "fmt"

func main() {
    cards := []string{"Ace of Diamonds", newCard()}
    cards = append(cards, "Six of Spades")
    // fmt.Println(cards)
    for i, card := range cards {
        fmt.Println(i, card)
    }
}

func newCard() string {
    return "Five of Diamonds"
}
```

Output:

```
0 Ace of Diamonds
1 Five of Diamonds
2 Six of Spades
```

Loops are handy if you want to run the same code over and over again, each time with a different value.

Note: There is no parenthesis in for loop definition.
":=" operator is used to assign a new value to index and card variables in each iteration.

Custom Type Declaration

In Go, you can create "**custom types** using **type** declarations".
A custom type is a **user-defined** type based on an **existing data type**. You can create custom types to make your code more expressive, readable, and easier to maintain.

To define a custom type declaration, you can use the "**type**" keyword.

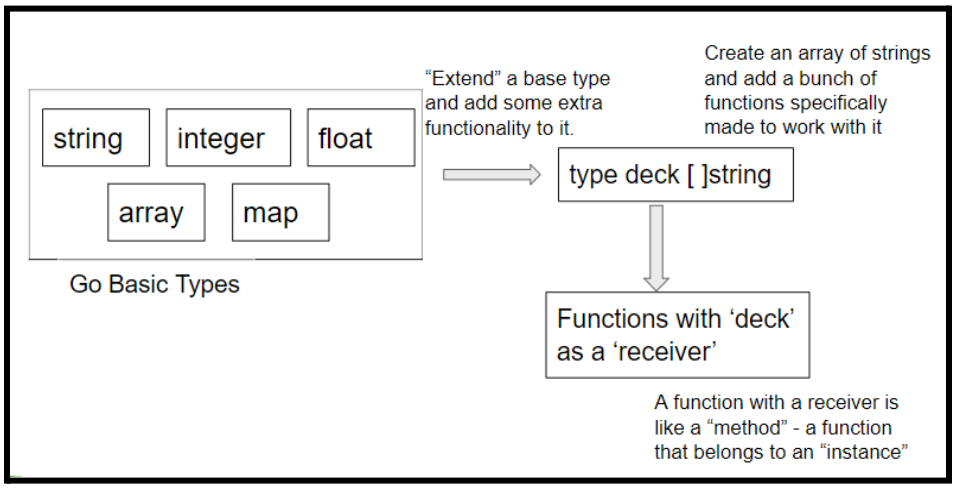
Syntax:

```
type typeName data_type
```

Example:

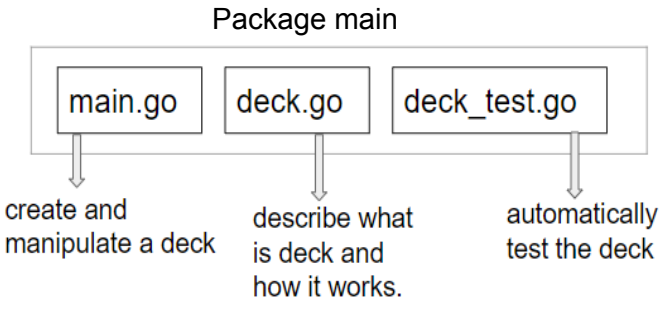
`type deck []string` —-- type name is **deck** with base type **string slice**
`type myInt int` —-- type name is **myInt** with the base type **int**

See the cards overview given below.
You are going to create a user-defined type **deck** based on string slice. Then the deck type is used to create a **receiver function** known as **methods** (see later).



- ➔ In the same project directory, create a new file **deck.go** to define **deck** type and associated receiver functions only.
- ➔ Create a new type, **deck** which is a **slice of string**.
- ➔ In order to test the deck type methods, you need to create test files. Create a **deck_test.go** file to write tests.

cards directory overview



Notice that the three files belong to the same package main.

deck.go

```
package main

type deck []string
```

➔ In the main.go file, change the cards variable type from slice of string to new type deck.

```
package main

func main() {
    // change to deck type
    cards := deck{"Ace of Diamonds", newCard()}
    cards = append(cards, "Six of Spades")

    for i, card := range cards {
        fmt.Println(i, card)
    }
}

func newCard() string {
    return "Five of Diamonds"
}
```

➔ Run the code using the command below.
(To execute both the files belonging to package main)

```
go run main.go deck.go
```

Output:

```
0 Ace of Diamonds
1 Five of Diamonds
2 Six of Spades
```

Methods

Go language support **methods**.
Go methods are similar to Go **functions** with one difference, i.e, the method contains a **receiver argument** in it. This is why it is also referred to as a **receiver function**.

A receiver function in Go is "**defined within a type**." The receiver function is associated with the type and can only be called on instances of that type.

To define a method, specify the receiver type in parentheses **before** the method name.

Syntax:

```
func(reciver_name Type) method_name(parameter_list)(return_type){
   // Code
}
```

➔ In the deck.go create a new method **print** of type deck. The print method prints each card in the cards.
Previously in the main function, you have created a for loop to iterate through cards (slice) to print each card value. But later the cards type is changed to deck type therefore, the new method print will print any cards of type deck only.

deck.go

```go
package main

import "fmt"

// create a new type of 'deck'
// which is a slice of strings
type deck []string

func (d deck) print() {
    for i, card := range d {
        fmt.Println(i, card)
    }
}
```

In the above code the receiver name is d (or any) and method name is print. This method does not have a parameter list nor return type.
Only a variable of type deck can use (call) this method since the receiver type is exclusively of type deck.

When you create a method in your code, the method and receiver type must be defined in the same package. You will get compiler error if you create a method in which the receiver type is already defined in another package including in-built types like int, string, etc.

➔ In the main.go use the cards instance to call the print method to print all the cards.

main.go

```
package main

func main() {
    cards := deck{"Ace of Diamonds", newCard()}
    cards = append(cards, "Six of Spades")
    cards.print()
}

func newCard() string {
    return "Five of Diamonds"
}
```

Output:

```
0 Ace of Diamonds
1 Five of Diamonds
2 Six of Spades
```

Notice that the method call is slightly different from the usual function call. The receiver type variable must be prefixed before the method call.

```
cards.print()
```

The cards variable is received in the print method by name d. Thus, d contains the same value as the cards variable.
d is now the copy (instance/working variable) of the deck type (cards) which can be used inside the method.
Q. How is the receiver variable used in the above code?

(Receiver variable is similar to keywords such as self (Python) or this (JavaScript).)

Creating a new Deck
Revisit the project overview image shown at the beginning of this project.
You need to create a new function to generate a deck of cards.

➔ Create a new function**, newDeck** to get a deck of cards.

deck.go

```
func newDeck() deck {


}
```

Q. Why is newDeck not created as a method?
It would be a waste of memory to create an empty deck to call the receiver function.

Q. How to add cards to the deck in newDeck()? (Avoid typing all 52 cards in the deck)

1. First create an empty deck.
2. Create a list of card suits.
3. Create another list of card values
4. Loop through both the lists to add a new card as 'value of suit' to the deck.

deck.go

```
func newDeck() deck {
    cards := deck{} // empty deck
    cardSuits := []string{"Spades", "Hearts", "Diamonds", "clubs"}
    cardValues := []string{"Ace", "Two", "Three", "Four"}

    for _, suit := range cardSuits {
        for _, value := range cardValues {
            cards = append(cards, value+" of "+suit)
        }
    }
    return cards
}
```

Underscore(_) is called the **blank identifier**.
Go is very strict for **unused variables**, you can not compile the code with any unused variables. Some functions return multiple values, and some values are not used. So, Go provides a "**blank identifier**" for replacing unused variables then the code can be compiled.

Use **underscore** to replace **unused** variables from getting compile time errors. In the above code you ignored index since it is not required to use.

➔ Now call the newDeck() to create a new deck of cards. Print the cards to see the values.

main.go

```
package main

func main() {
    cards := newDeck()
    cards.print()
}
```

Output:

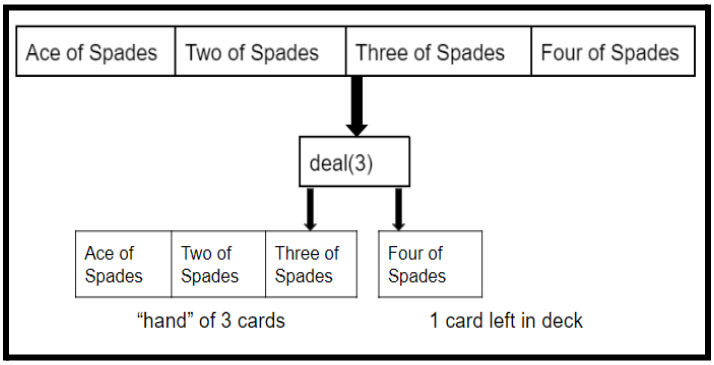
```
0 Ace of Spades
1 Two of Spades
2 Three of Spades
3 Four of Spades
4 Ace of Hearts
5 Two of Hearts
6 Three of Hearts
7 Four of Hearts
8 Ace of Diamonds
9 Two of Diamonds
10 Three of Diamonds
11 Four of Diamonds
12 Ace of clubs
13 Two of clubs
14 Three of clubs
15 Four of clubs
```

Creating deal function

In a card game, deal means to distribute cards.

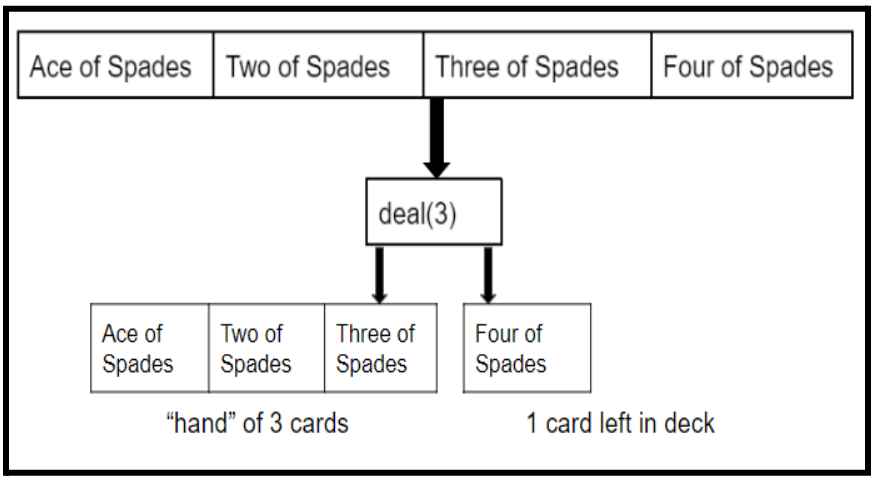
Deal function will create a hand of cards.

- The picture below shows the idea behind the implementation of the deal function.



You already have a deck of cards generated by the newDeck function. Use these cards to create a hand of the required number of cards.
The deal function **splits** the deck slice into two smaller slices. First slice contains the **cards in hand** and the second slice contains the **cards remaining** in the deck.

Use **slicing** to create a subset of the deck.



cards[:handSize]	cards[handSize:]
cards[:3]	cards[3:]
0, 1, 2	3

handSize is the number of cards required in hand.

➔ Create a new function deal. The function signature is given below.

```
func deal (d deck, handSize int) (deck, deck)
```

The deal function takes two parameters and has two return types of deck type. (You know that Go functions can return more than one value.

deck.go

```
func deal(d deck, handSize int) (deck, deck) {
    return d[:handSize], d[handSize:]

}
```

➔ Use the deal function to create a hand. You can also print the hand and remainingCards deck to see the values.

main.go

```
package main

func main() {
    cards := newDeck()
    hand, remainingCards := deal(cards, 5)

    hand.print()
    remainingCards.print()
}
```

Output:

```
0 Ace of Spades
1 Two of Spades
2 Three of Spades
3 Four of Spades
4 Ace of Hearts
0 Two of Hearts
1 Three of Hearts
2 Four of Hearts
3 Ace of Diamonds
4 Two of Diamonds
5 Three of Diamonds
6 Four of Diamonds
7 Ace of clubs
8 Two of clubs
9 Three of clubs
10 Four of clubs
```

In the above output,

Index 0 - 4 are cards in hand, and
Index 0 - 10 are remaining cards in the deck.

Comprehend the statement below with the help of deal function.

```
hand, remainingCards := deal(cards, 5)
```

```
func deal(d deck, handSize int) (deck, deck) {
    return d[:handSize], d[handSize:]

}
```

Q. How many values are passed to the deal() ?
Q. Why are there two variables defined on the left hand side of the function call?

The values that are declared within a function when the function is called are known as an **argument**.
The variables that are defined when the function is declared are known as **parameters**.
Arguments are used to **pass values** to a function and **parameters** are used to **receive (store) values** passed from a function call.
The **number** of arguments must be **equal** to the number of parameters.

In the next topic you are going to learn how to work with files.

File Input and Output

You know that we can add (store) data to a file as well as view/access the contents of a file. In computer science, storing data to a file is termed as **writing** to a file and accessing the file content is termed as **reading** from a file.
Let's see how to do this task in Go.

Writing to File
Now you are going to save the deck of cards to an external file. For this you need to create a new function **saveTofile** to save the deck contents to a file on a local machine. This involves interacting with the operating system.
Working with the operating system is a very basic requirement of any programmer. The Golang **OS** package gives us the ability to work with all things related to the OS.

The `**os**` package in Go provides a platform-independent interface to operating system functionality. It includes functions for working with files, directories, processes, and environment variables. See more about os package in Go official documentation.

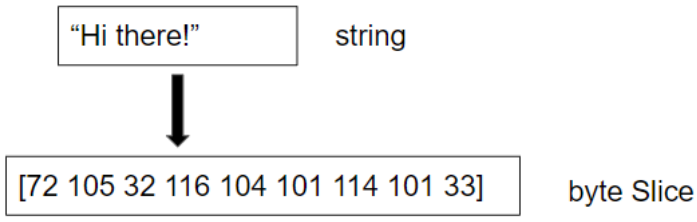
You can use the **WriteFile** function from the os package to save data to a file. The function signature is given below.

```
WriteFile(filename string, data []byte, perm FileMode) error
```

WriteFile **writes** data to the **filename**, creating it if necessary. If the file does not exist, WriteFile **creates** it with permissions perm; otherwise WriteFile truncates it before writing, without changing permissions.

The function requires data of type **byte slice** to add to the file.

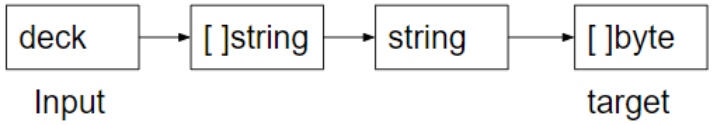
Q. What is a byte slice?
Byte Slice is a computer friendly representation of string. Computer understands every string as a byte slice.
Example below shows byte slice representation for string Hi there!



Each number corresponds to the ascii character code (from ascii table).

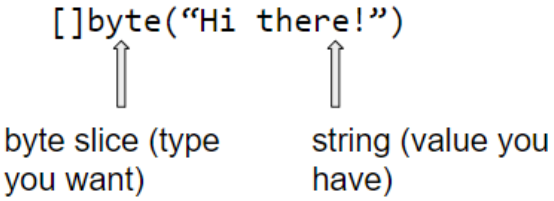
Convert deck type to string
The data you want to save to file is of deck type (string slice). However, to use the WriteFile function it expects a byte slice.

Q. How to convert your data (deck type) to byte slice (actually a string)?



The deck type cannot be directly converted to byte slice. First deck type which is based on string slice should be converted to string and then convert the string to byte slice. Then you are ready to use the WriteFile function.

Example below shows how to obtain a byte slice from a string.



Code:

```
func main() {
    greeting := "Hi there!"
    fmt.Println([]byte(greeting))
}
```

Output:

```
[72 105 32 116 104 101 114 101 33]
```

➔ Before you implement the saveToFile function, create a helper function **toString** that returns a **string** from a **deck**. (Avoid putting this logic in the saveToFile function.)

deck.go

```
func (d deck) toString() string {
    return []string(d)
}
```

Q. Why does the above code show error?

- The above function only converts deck type to string slice; however, you want all the cards in the deck as a single string. For this you need to **join** the items in the slice to make one string.
- The **strings** package in Go provides a function called **Join** that creates a string from a string slice. The function signature is given below.

```
Join(a []string,sep string) string
```

Join function has two parameters and one return type string. The first parameter is a **string slice** which you want to join and the second is a **string** which is used as a **separator** of items in the resultant string.

➔ Now **import** the **strings** package in deck.go file. Update the toString function body as shown below.

```
func (d deck) toString() string {
    return strings.Join([]string(d), ",")
}
```

➔ Call the toString() to see whether the function returns the desired output.

main.go

```
package main

import "fmt"

func main() {
    cards := newDeck()
    fmt.Println(cards.toString())
}
```

Command: `go run main.go deck.go`
Output:

```
Ace of Spades,Two of Spades,Three of Spades,Four of Spades,Ace of Hearts,Two of Hear
ts,Three of Hearts,Four of Hearts,Ace of Diamonds,Two of Diamonds,Three of Diamonds,
Four of Diamonds,Ace of clubs,Two of clubs,Three of clubs,Four of clubs
Process 26108 has exited with status 0
```

Now your helper function is ready to use, so start implementing the saveToFile function.

SaveToFile function

➔ Create a new function **saveTofile** in the deck.go file. Define the function signature as shown below.

```
func (d deck) saveTofile(filename string) error
```

The function is a receiver function of type deck. It accepts a filename (string) as an input parameter and returns an error value.

➔ Now use the WriteFile function from the os package to save the deck of cards to a given filename.

deck.go

```
func (d deck) saveTofile(filename string) error {
    return os.WriteFile(filename, []byte(d.toString()), 0666)
}
```

File permission 0666 means anyone can read and write to this newly created file.

- Make sure you have the following packages imported in deck.go.

```
import (
    "fmt"
    "os"
    "strings"
)
```

➔ Call the saveTofile function to store the **cards** in the filename **my_cards**.

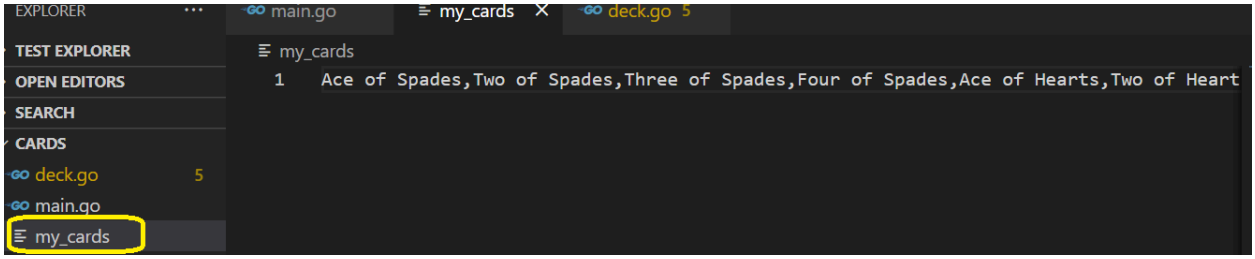
main.go

```
package main

func main() {
    cards := newDeck()
    // pass name of the new file you create
    cards.saveToFile("my_cards")
}
```

Here you have not checked the **error value** returned from the saveTofile function.

Output: `go run main.go deck.go`

A new text file “my_cards” is created and contains the cards in the deck.



Notice that the separator (,) specified in the Join() is used to separate each card in the resultant string.

You have successfully created a file and added the data to a file. Now let's look into how to read the data from the file.

Reading From a File

Here you will read the data from my_cards file and create a deck of cards. Now it is the opposite process.

To read the data from a file use **ReadFile()** from the os package. The function signature is shown below.

```
ReadFile(filename string) ([]byte, error)
```

Q. Describe the function signature.

ReadFile reads the filename and returns the byte slice and error value.
For now ignore the error. You will learn how to handle errors in the next topic.

➔ Create a new function **newDeckFromFile** in deck.go file to **read** the contents from the file saved in the above task. Define the function as shown below.

```
func newDeckFromFile(filename string) deck
```

The function accepts the name of the file to read and returns a deck type.

Q. Why is the above function not created as a method?

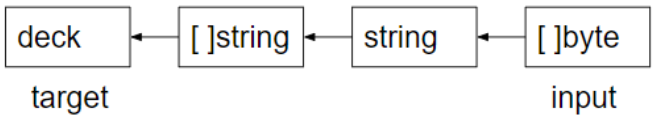
➔ Use the ReadFile() from the os package to read the file content. Ignore the error value.

deck.go

```
func newDeckFromFile(filename string) deck {
    byteSlice, _ := os.ReadFile(filename)
}
```

Now you have read the file data and stored it in a variable named byteSlice which is of type byte slice. But how can you create a deck type from byte slice data?

Convert byte slice to deck
You can follow the opposite process of the saveToFile function.



First you can convert the byte slice to string using type casting.

```
str := string(byteSlice)
```

Once you have the string, use the Split() from the strings package to create a string slice. The Split function signature is shown below.

```
func Split(s string, sep string) []string
```

The function takes string data as the first parameter and string separator as the second parameter. And returns a string slice as an output.

`stringSlice := strings.Split(string, “,”)`

The typecast the string slice directly to deck type. See the complete function below.

deck.go

```
func newDeckFromFile(filename string) deck {
    byteSlice, _ := os.ReadFile(filename)
    // convert byteSlice to deck
    stringSlice := strings.Split(string(byteSlice), ",")
    return deck(stringSlice) // convert string slice to deck type

}
```

➔ Now call the newDeckFromFile() to create a new deck of cards from a file. Update the main.go as shown below.

main.go

```
package main

func main() {
    cards := newDeckFromFile("my_cards")
    cards.print()
}
```

Output: `go run main.go deck.go`

```
0 Ace of Spades
1 Two of Spades
2 Three of Spades
3 Four of Spades
4 Ace of Hearts
5 Two of Hearts
6 Three of Hearts
7 Four of Hearts
8 Ace of Diamonds
9 Two of Diamonds
10 Three of Diamonds
11 Four of Diamonds
12 Ace of clubs
13 Two of clubs
14 Three of clubs
15 Four of clubs
```

The function works successfully.

Q. What happens if you read a file which does not exist?

```
package main

func main() {
    cards := newDeckFromFile("cards")
    cards.print()
}
```

Output: The program simply terminates successfully (status 0). You don't know whether you have read the file successfully or not.

```
DAP server listening at: 127.0.0.1:51824
Type 'dlv help' for list of commands.
0
Process 11260 has exited with status 0
```

If the filename is wrong, your program must let you know that the filename is incorrect so that you know what to do next.
To improve the above code, do not ignore the error value from the ReadFile().

Error Handling

In the above two functions we simply ignored the error value returned from the function. If the file operation is **successful** the error value returned from the function is **nil**. In this case you do not have to worry much. However, if it is not nil (some error), your program needs to handle the error for the correct result.

There are two ways to handle the error if the error value is not nil.

Option 1: Log (print) the error and return a call to newDeck(). newDeck() always returns a deck to work with.

```
fmt.Println(err)
return newDeck()
```

Option 2: Log the error and quit the program.
Conventionally status code **zero** indicates **success, non-zero** an **error.**

➔ Using option 2, update the readFromFile() as shown below.

```go
func newDeckFromFile(filename string) deck {
    byteSlice, err := os.ReadFile(filename)

    // error handling
    if err != nil {
        // fmt.Println("error:", err)
        log.Fatal(err)
    }
    // convert byteSlice to deck
    stringSlice := strings.Split(string(byteSlice), ",")
    return deck(stringSlice) // convert string slice to deck type
}
```

➔ Call the newDeckFromFile() using filename as cards and see the difference in output.

```go
package main

func main() {
    cards := newDeckFromFile("cards")
    cards.print()
}
```

Output:

```
Type 'dlv help' for list of commands.
2024/02/02 14:59:23 open file: The system cannot find the file specified.
Process 9404 has exited with status 1
```

Using Option 1

```go
func newDeckFromFile(filename string) deck {
    byteSlice, err := os.ReadFile(filename)

    // error handling
    if err != nil {
        fmt.Println("error:", err)
        return newDeck()
    }
    // convert byteSlice to deck
    stringSlice := strings.Split(string(byteSlice), ",")
    return deck(stringSlice) // convert string slice to deck type
}
```

Output:

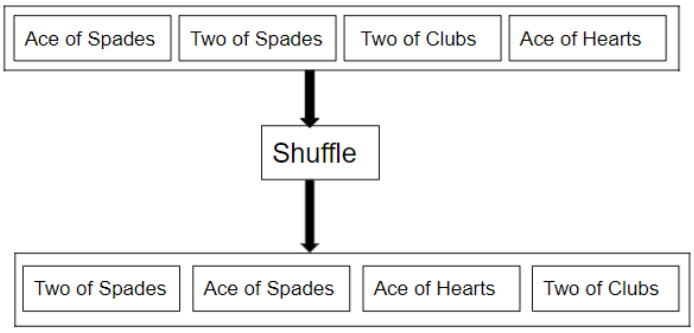
```
error: open file: The system cannot find the file specified.
0 Ace of Spades
1 Two of Spades
2 Three of Spades
3 Four of Spades
4 Ace of Hearts
5 Two of Hearts
```

Next, you will look at card shuffling using the standard library package, rand.

Shuffling a Deck

Card shuffling changes the order of cards in the deck in some random order.

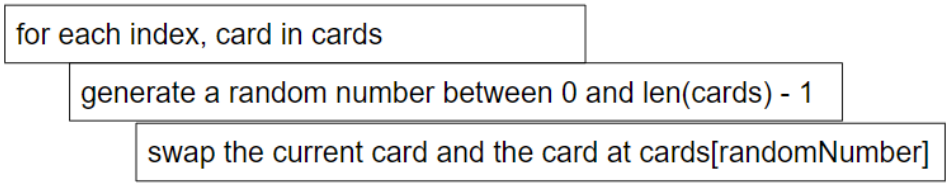
> Example: A deck of cards is passed to a shuffle function whereby it returns a deck of cards in a random order.



Q. How to implement the shuffle function?

Example:

In this example you have a deck containing four cards. You can use a for loop to iterate through each card. In each iteration swap the card at the current index (say 0), Ace of Spades with a randomly generated index (say 2), Two of Clubs. Continue the same process until the loop terminates.



Q. How to generate a random number in Go?
Go has a standard library package **rand** which is used to generate random numbers.

```
import math/rand
```

From the rand package, use the Intn function to generate the random number. The function signature is given below.

```
func Intn(n int) int
```

Parameter n is the upper bound to generate a random number. It panics if n <= 0.

Panic in Go is one of the mechanisms to handle errors. It is used to abort a function execution.
#Explore more about panic yourself.

Q. What should be the value of n for your random number generator?
n = len(deck of cards) - 1

➔ Define a method shuffle. The receiver type is deck.
You need to shuffle a deck of cards which is already created; therefore, you can create shuffle as a receiver function of type deck.

deck.go

```
func (d deck) shuffle() {
    for i := range d {
        // generate new random index
        newIndex := rand.Intn(len(d) - 1)
        // swap
        d[i], d[newIndex] = d[newIndex], d[i]
    }
}
```

➔ Call the shuffle method to shuffle the cards.

main.go

```
package main

func main() {
    cards := newDeck()
    cards.shuffle()
    cards.print()
}
```

Output: `go run main.go deck.go`

Before shuffle

Find the complete deck.go code below.

```go
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strings"
    "time"
)

// create a new type of 'deck'
// which is a slice of strings
type deck []string // slice

func newDeck() deck {
    cards := deck{} // empty deck
    cardSuits := []string{"Spades", "Hearts", "Diamonds", "clubs"}
    cardValues := []string{"Ace", "Two", "Three", "Four"}

    for _, suit := range cardSuits {
        for _, value := range cardValues {
            cards = append(cards, value+" of "+suit)
        }
    }
    return cards
}
```

```go
func (d deck) print() {
    for i, card := range d {
        fmt.Println(i, card)
    }
}

func deal(d deck, handSize int) (deck, deck) {
    return d[:handSize], d[handSize:]
}

func (d deck) toString() string {
    return strings.Join([]string(d), ",")
}

func (d deck) saveToFile(filename string) error {
    return os.WriteFile(filename, []byte(d.toString()), 0666)
}
```

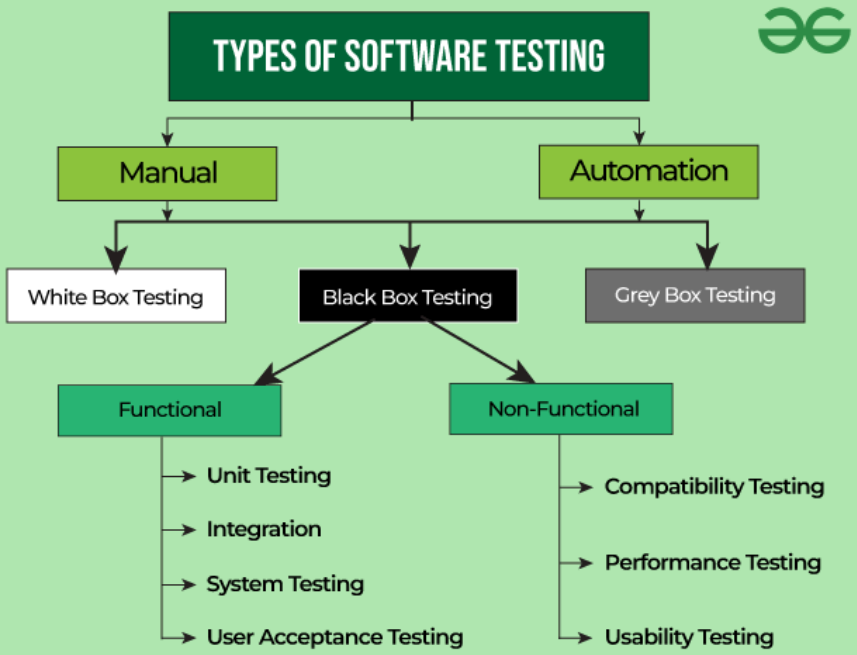
```go
func newDeckFromFile(filename string) deck {
    byteSlice, err := os.ReadFile(filename)

    // error handling
    if err != nil {
        // fmt.Println("error:", err)
        log.Fatal(err)
    }
    // convert byteSlice to deck
    stringSlice := strings.Split(string(byteSlice), ",")
    return deck(stringSlice) // convert string slice to deck type
}
```

#Add files to the GitLab repository. See the Git&GitLab setup document for the procedure.

Testing

Testing is part of the software development process and the purpose is to produce better software, more robust, with fewer bugs, and more stable.
There are different kinds of testing that can be done to improve software quality. Starting with unit testing, integration testing, end to end testing, and so on.

In this course you will focus on **unit testing** only.



Unit Testing

Q. What is Unit Testing?

Unit testing is the most popular level of testing implemented by Go developers. It involves writing individual-function tests, cross-function tests, and other intra-package tests to make sure that a distinct package and its functions perform according to their requirements.

The goal of unit testing is only to **verify** the **performance** of an **individual unit**.

Go's built-in support for unit testing makes it easier to test as you go. Specifically, using **naming conventions**, Go's **testing** package, and the **go test** command, you can quickly write and execute tests.

In Go, the **testing** package provides a framework for writing and running tests. It includes functions and types for test management, error reporting, and more.

Some of the popular testing frameworks available in Go are:

To create unit testing in Go using the built-in **testing** package, you should follow these conventions:

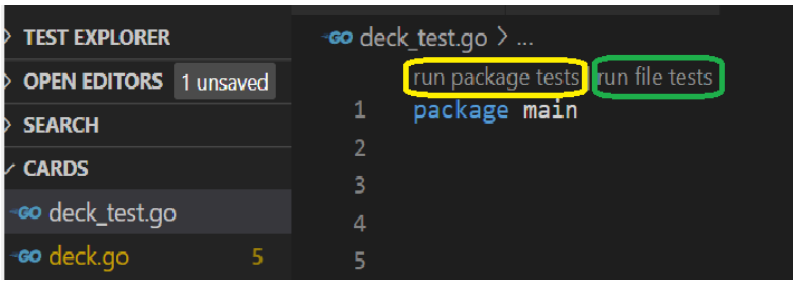
1. Create a file with a name that ends with **_test.go**. The best practice is to name the file following the file it's testing. For example, if we have **math.go** the unit test file should be **math_test**.go. And it's also best practice to put the test file and the implementation in the same package.

2. Create a function in the test file and start the name with **Test**. The function should take ***testing.T** type from the Golang testing package. The *testing.T type is a testing handle that provides methods for reporting test failures and logging test status. The test function signature is TestXXX(t *testing.T) with XXX as the name of the function you are testing.

Implementation

➔ Create a test file **ending** with **_test.go** inside the cards directory. (Ignore this step if the file is already created.)
The file name is **deck.go** so keep the test file name as **deck_test.go**.

➔ As always, define the package name main.
You must implement test functions in the same package as the code you're testing.



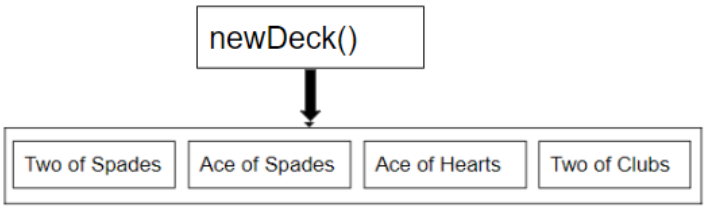
You will notice two buttons on the top,

```
run package tests
run file tests
```

As the name suggests, the first command is to run all the tests in the package and the second command is to run the tests inside the current file.

Firstly, you are going to test the **newDeck** function. What components to test in the function?

Example:



Output

You can write unit tests to verify that the function behaves correctly by checking the output component. For example:

Test case 1: Verify that the function generates four cards in the deck.
Test case 2: Verify that the first card in the deck is Two of Spades.
Test case 3: Verify that the last card in the deck is Two of Clubs.

➔ In the deck_test.go, define a new function **TestNewDeck**.

```
package main

import "testing"

func TestNewDeck(t *testing.T) {
    d := newDeck()
    if len(d) != 16 {
        t.Errorf("Expected deck length of 16 but got %d", len(d))
    }
}
```

Errorf method is used to print a message to the console and end execution.

- This function will be automatically called by the **go test runner** with the argument **t *testing.T** . T is a struct type defined in the package testing.
- In the test file all the function names must start with an **uppercase letter**.
- To run the test file use the command '`go test`' or directly click on the '**run test**' button right above the function.

Output:

```
PASS
ok      _/c_/Users/tsheringL/Desktop/golang/cards    0.431s
```

- If the test fails, you will get an error message as shown below.

Say deck length value given is 20.

```
package main

import "testing"

func TestNewDeck(t *testing.T) {
    d := newDeck()
    if len(d) != 20 {
        t.Errorf("Expected deck length of 20 but got %d", len(d))
    }
}
```

Output:

```
--- FAIL: TestNewDeck (0.00s)
    deck_test.go:8: Expected deck length of 20 but got 16
FAIL
exit status 1
FAIL    _/c_/Users/tsheringL/Desktop/golang/cards   0.484s
```

The remaining two tests for newDeck() are shown below.

First card: Ace of Spades

Last card: Four of Clubs

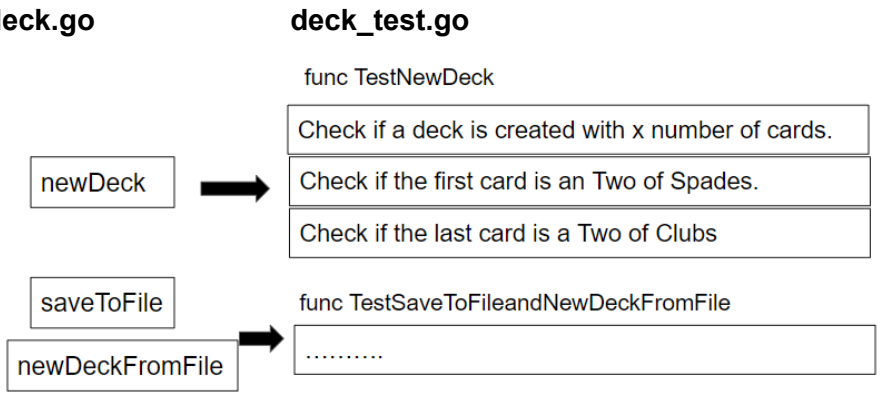
Code:

```
func TestNewDeck(t *testing.T) {
    d := newDeck()
    // test 1
    if len(d) != 16 {
        t.Errorf("Expected deck length of 16 but got %d", len(d)) // or %v
    }
    // test 2
    if d[0] != "Ace of Spades" {
        t.Errorf("Expected Ace of Spades but found %v", d[0])
    }
    // test 3
    if d[len(d)-1] != "Four of Clubs" {
        t.Errorf("Expected Four of Clubs but found %v", d[len(d)-1])
    }
}
```

Using command line:

```
go test deck_test.go deck.go
```

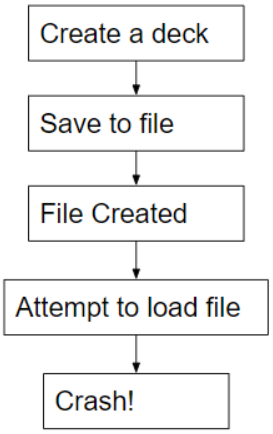
Other functions you need to test are,



Testing File IO (Optional)
Write a test for writing to file (saveTofile()) and reading from a file (newDeckFromFile()).

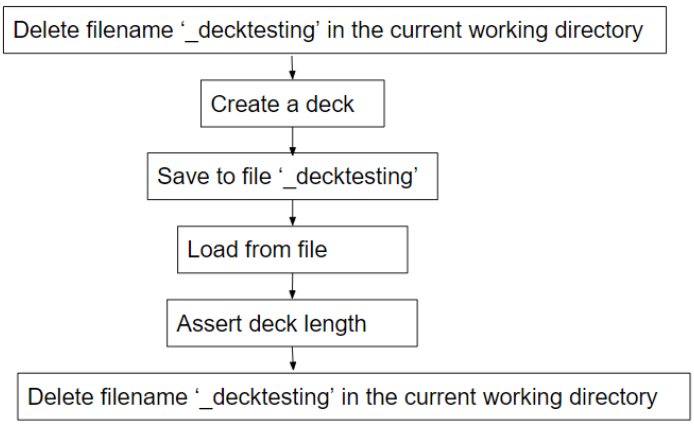
- Consider the situation (edge case) given below.
  When you run this test file, you have created a new deck and successfully saved the file in the current directory. However, while reading from the file, if the test crashes due to some error, the file that was created remains in the current directory. In this case, you should **remove** the file which was created just before. So, how can you do this **clean up** while performing tests in go?

- See the diagram given below to understand the process.



- You can follow the steps below to perform automatic clean up.

➔ Create a new function **TestWriteandRead.**
The diagram below shows the implementation process to write this test.



Q. How to delete a file?

- Import package **os** to use **Remove** function.

```
func Remove(name string) error
```

```go
func TestWriteAndRead(t *testing.T) {
    os.Remove("_decktesting")
    deck := newDeck()
    deck.saveToFile("_decktesting")

    loadedDeck := newDeckFromFile("_decktesting")
    if len(loadedDeck) != 16 {
        t.Errorf("Expected 16 cards in deck but got %v",
len(loadedDeck))
    }
}
```

➔ Run the test.
Use the `go test` command or click the **run tests** button.

```
PASS
ok      _/c_/Users/tsheringL/Desktop/golang/cards    0.517s
```

Go fundamentals recap

- Go Program Structure:
  - Package main
  - Imports
  - Main function

- Functions
  Function syntax

- Methods
- Slice
  Slicing

- Function without receiver but with arguments. Why?
  In the deal function, there is no receiver because we don't have to modify the original deck of cards. If you put a receiver, it seems like you are removing the cards(hand size) from the deck. But we simply want to create two different decks.

(See more about about receiver variable and * pointer in the next section).

Structs in Go

A struct (short for structure) is used to create a collection of members of **different** data types, into a single variable.

While arrays are used to store multiple values of the same data type into a single variable, **structs** are used to store multiple values of **different** data types into a single variable.

To declare a structure in Go, use the **type** and **struct** keywords:

Syntax:

```
type struct_name struct {
  member1 datatype;
  member2 datatype;
  member3 datatype;
  ...
}
```

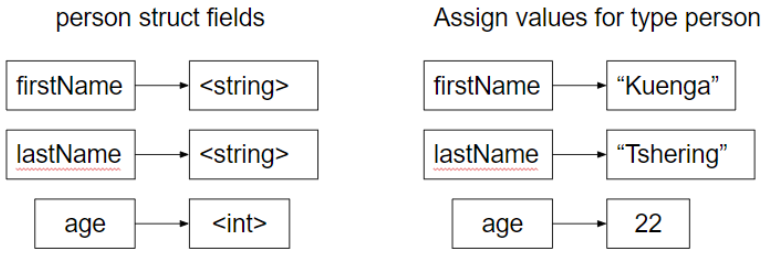
Before you define a struct you need to identify the struct properties (members). Properties are those that **collectively** describe the struct.

Example for a struct type Person the members are: name, age, job and salary.

```
type Person struct {
  name string
  age int
  job string
  salary int
}
```

Data type for each struct property must be specified in struct declaration.

Example:



One instance(copy) of a person. You can create as many instances as required.

The values assigned to the struct properties must match the data type defined in the struct declaration otherwise you will get a compile error.

In the above Cards project, you have represented cards as a deck type which is actually a string slice.

Example:

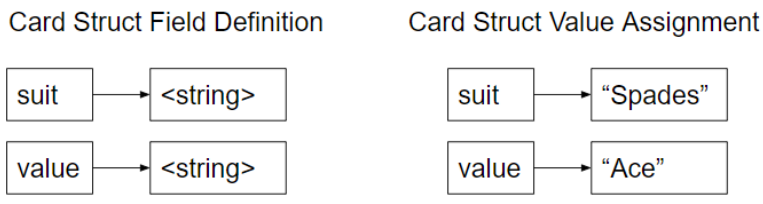
[“Ace of Spades”, “Two of Clubs”, “Three of Diamond”]

Now that you have some idea about the struct, you can think of representing a card as a struct type.

A card can be identified by two properties, **suit** and **value**.

In the above implementation, to know the suit and value for a card, you need to do string manipulation.
One of the solutions is to split the string based on white space and get the first and last item from a list. This is a complex task and may not be accurate too.

Card example:



Let us look into details about the struct implementation.

- ➔ Create a new project, **structs** at the root directory.
- ➔ Inside this project, create a new file **main.go** and define package main in the file.

➔ Define a struct type **person** having the following properties/fields.

main.go

```go
package main

type person struct {
    firstName string
    lastName  string
}

func main() {
}
```

Note: Comma not required while defining multiple properties for a struct.

➔ Assign values by creating a person instance.

There are two different ways to declare/assign values to the struct fields.

Example:

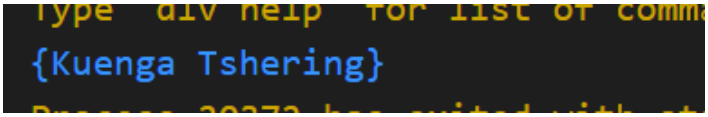
```go
person1 := person{"Kuenga", "Tshering"} // 1
person1 := person{firstName: "Kuenga", lastName: "Tshering"} // 2
```

In option 1 you have to follow a **strict sequence** of fields defined in the struct. On the other hand, the second option has a **flexible order.**

Code: main.go

```go
package main

type person struct {
    firstName string
    lastName  string
}

func main() {
    // person1 := person{"Kuenga", "Tshering"} // 1
    person1 := person{firstName: "Kuenga", lastName: "Tshering"} // 2
    fmt.Println(person1)
}
```

Output: `go run main.go`



Updating Struct Values

How can you update the values for a struct instance?

➔ Create a new variable, kuenga of struct type person.

```
var kuenga person
```

➔ Print variable kuenga.

```
fmt.Println(kuenga)
```

```
import "fmt"

type person struct {
    firstName string
    lastName  string
}

func main() {
    var kuenga person
    fmt.Println(kuenga)
}
```

Output:

Q. What is the output? Why?

- Alternative way to print the value to the console is using **Printf().** It allows you to specify a certain formatting.

```
fmt.Printf("%+v", kuenga)
```

Output:

```
{firstName: lastName:}
```

Some Go output format verbs are:

%v	the value in a default format when printing structs, the plus flag (%+v) adds field names
%T	a Go-syntax representation of the type of the value
%t	the word true or false
%d	base 10 (int and int8)
%s	the uninterpreted bytes of the string or slice

#Explore the difference between **Println()** and **Printf()** yourself.

➔ Assign new values to the new struct variable.
Use the **dot operator** syntax.

```
variableName.Field = newValue
```

Code:

```
package main

import "fmt"

type person struct {
    firstName string
    lastName  string
}

func main() {
    var kuenga person

    kuenga.firstName = "kuenga"
    kuenga.lastName = "Tshering"

    fmt.Println(kuenga)
    fmt.Printf("%+v", kuenga)
}
```

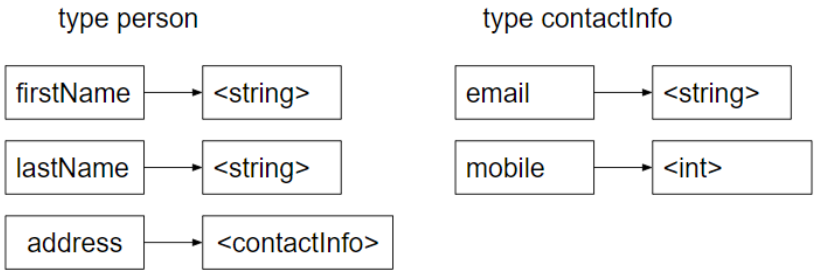
Output:

```
{kuenga Tshering}
{firstName:kuenga lastName:Tshering}
```

More advanced features of Structs

Embedding Structs (More complex uses of structs)
Nesting one struct type to another struct type. (Nested struct)

See the example given below.
In the person type, the address field is of type **contactInfo** which is another custom type. Here, the **contactInfo** type is **embedded** into the **person** type.



Implementation

- ➔ Create another struct type contactInfo with fields as shown above.
- ➔ Create an instance of a person struct.
  Do not forget the comma after each field while creating an instance.

main.go

```go
package main

import "fmt"

type contactInfo struct {
    email  string
    mobile int
}
type person struct {
    firstName string
    lastName  string
    address   contactInfo
}
func main() {
    kuenga := person{
        firstName: "kuenga",
        lastName:  "Tshering",
        address: contactInfo{
            email:  "ku@gmail.com",
            mobile: 12345678,
        },
    }
    fmt.Printf("%+v", kuenga)
}
```

Output:

```
{firstName:kuenga lastName:Tshering contactInfo:{email:ku@gmail.com mobile:12345678}}
```

- It is not mandatory to specify **field names** for **embedded** fields rather you can mention only the **type** in struct definition.

  Code:

```
type person struct {
    firstName string
    lastName  string
    contactInfo
}
```

Output:

```
{firstName:kuenga lastName:Tshering contactInfo:{email:ku@gmail.com mobile:12345678}}
```

Structs with Receiver Functions (Methods)

Receiver variable of type struct.

In the cards project, you have created methods having a receiver type deck. You can also create methods having receiver type **struct**.

Let's look at the example below.

➔ Create a new method **print** having receiver type as struct person to print the details of a person.

```
person.print()
```

Code:

```
func (p person) print() {
    fmt.Printf("%+v", p)
}
```

```
kuenga.print()
```

Output:

```
{firstName:kuenga lastName:Tshering contactInfo:{email:ku@gmail.com mobile:12345678}}
```

➔ Create another receiver function **updateName** to update the name of a person.

```
func (p person) updateName(newfirstName string) {
    p.firstName = newfirstName
}
```

```
kuenga.print()
kuenga.updateName("Deki")
kuenga.print()
```

Q. What is the output? Is the first name updated to Deki ?

Output:

```
{firstName:kuenga lastName:Tshering contactInfo:{email:ku@gmail.com mobile:12345678}}{firstName:
kuenga lastName:Tshering contactInfo:{email:ku@gmail.com mobile:12345678}}
```

No. Why?

Q. Why did this happen?!
To understand this, you have to know the concept called “Pass By Value” when passing arguments to a function/method.

Pointers

In programming there are two ways to pass **arguments** to a function/method: **call by value** and **call by reference**.

Pass by value and Pass by reference are an important thing that needs to be careful of when working with Programming Language that supports us to access a ‘pointer’ value like Java, C#, C/C++, Go, etc.
The terms are used to describe **how variables are passed** on. When you create a method/function with parameters, the data type of the parameters can be set with a normal data type (int, string, etc.) or a pointer. This will make the difference in the **argument** that is passed:

Pass by value will pass the **value** of the variable into the method, or we can say that the original variable ‘**copy**’ the value into another memory location and pass the newly created one into the method. So, anything that happens to the variable inside the method will not affect the original variable value.

Pass by reference will pass the **memory location** instead of the value. In other words, it passes the **‘container**’ of the variable to the method so, anything that happens to the variable inside the method will affect the original variable.

In a nutshell,

Pass by value means the actual value is passed on. Pass by reference means a number (called an address) is passed on which defines where the value is stored.

For better and clearer understanding of the above concept let's see how data is stored in **RAM** (Random Access Memory) when the program runs. Imagine RAM as a series of slots that can store a value.
RAM stores data (variables, function calls) of a program which is in execution.

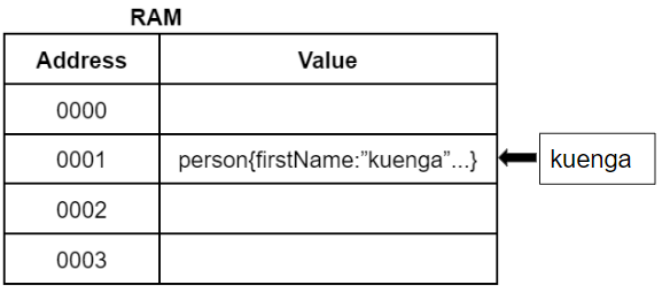
RAM

Address	Value
0000	
0001	
0002	
0003	

Each value stored in RAM has something called an **address**. When you try to access the value, the compiler checks the address first and then retrieves the value.

Suppose you create a new variable of struct type `person` and is stored in memory address 0001 in the RAM.

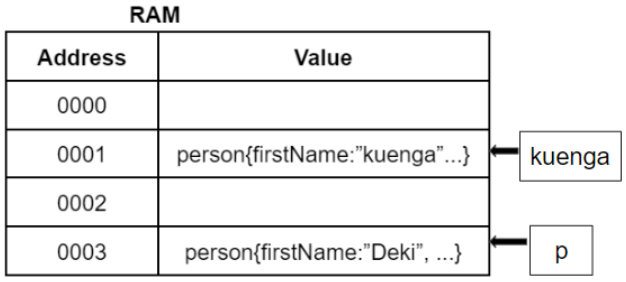
```
kuenga := person {firstName: "kuenga", lastName: "Tshering"}
```



Now, call the update function with an argument value "Deki". The `updateName()` has a receiver `p` of type `person`.

```
kuenga.updateName("Deki")
func (p person) updateName(s string) {
    ...
}
```

Since Go language is of type pass by value, it first creates a copy of kuenga and passes it to the updateName function and it is stored in p. p is then stored in a new location (0003) in the RAM.



So, when you pass the new value to update, the changes take place only in p (copy) and the value of the original variable kuenga remains unchanged in the memory.

This is an example of pass by value.

Q. How can you fix this issue?
Using Pointers (Pass by Reference).

Q. What is a Pointer?
In Go, all function parameters are **passed by value** by default. This means that a **copy** of the parameter value is passed to the function. However, you can use **pointers** to pass parameters by **reference**, which means that the function can modify the original parameter value.

Pointers are one of the most powerful and fundamental features of the Go programming language. They allow you to manipulate memory directly.

Pointer is a **variable** that stores the **memory address** of another variable.

Syntax to declare pointer: `var pointer_name *type`
Ex., `var p *int`

Pointers are represented by the * symbol.
In the above example, p is a pointer variable that can store the memory address of another int variable. The pointer variable is not assigned any value so the **zero value** is **nil**.

See an example to declare and initialize a pointer variable.

```
var x int = 10
var ptr *int = &x  // or prt:= &x    declare and initialize
fmt.Println(x)  // output: 10
fmt.Println(ptr) // output: 0xc0000140a8
fmt.Println(*ptr) // output: 10
```

In this example, we declare an integer variable **x** and initialize it with the value 10. We also declare a pointer variable **ptr** of type **int** and initialize it with the **memory address** of x using the **address** (&) operator. We then print the value of x, ptr, and the value stored in the memory address of ptr.

&variable	get the memory address of the value this variable is pointing at.
*pointer	get the value stored at this memory address.

Structs Pointers
Let us fix the above updateName() using a struct pointer.

➔ Update the updateName function as shown below.

```
func (pPointer *person) updateName(newfirstName string) {
    (*pPointer).firstName = newfirstName
}
```

Here you have changed the receiver variable from **struct value** type to **struct pointer** and used the **pointer variable** to update the name.

➔ Call the updateName method from the main function to update the first name.
Since the receiver type has changed to a pointer variable you need to create a pointer first.
You want to change the first name of the variable kuenga so create a pointer (kuengaPointer) that points to variable kuenga.
Now use the kuengaPointer variable to call the updateName().

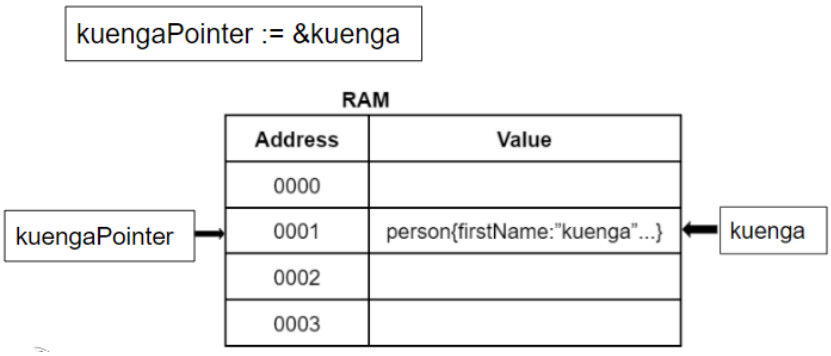
➔ Print kuenga variable before and after update to see the difference in values.

```go
func main() {
    kuenga := person{
        firstName: "kuenga",
        lastName:  "Tshering",
        contactInfo: contactInfo{
            email:  "ku@gmail.com",
            mobile: 12345678,
        },
    }
    kuengaPointer := &kuenga
    kuenga.print()
    kuengaPointer.updateName("Deki")
    kuenga.print()
}

func (p person) print() {
    fmt.Printf("%+v", p)
}

func (pPointer *person) updateName(newfirstName string) {
    (*pPointer).firstName = newfirstName
}
```

Output:

```
{firstName:kuenga lastName:Tshering contactInfo:{email:ku@gmail.com mobile:12345678}}{firstName:
Deki lastName:Tshering contactInfo:{email:ku@gmail.com mobile:12345678}}
```

Learn more about & below:



Here, `KuengaPointer` is a **pointer variable** that points to the memory address of variable `kuenga` which is a struct `person` type. It does not directly point to the struct person type (value) rather, it points to (stores) the address only.
To understand better, you can print `kuengaPointer` to see what it stores.
Use formatting **%p** to print a pointer variable.

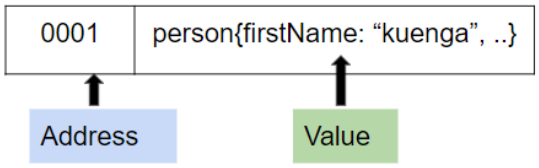
Ampersand(&) is an **address operator** which can be used to access the memory location of another variable. (creating a pointer)

To understand about the * operator, let's look at the code below:

```
kuengaPointer := &kuenga
kuengaPointer.updateName("Deki")

func (pPointer *person) updateName(newfirstName string) {
      (*pPointer).firstName = newfirstName
}
```

`updateName() is` called using a pointer(`kuengaPointer`) which points to a struct `person` type (`kuenga`).
This is why the function has a receiver as a **person pointer** (`pPointer`) which points to the `person` type. Now you use * in front of the pointer variable to access the firstName property value stored in that address and update the **value** of to a **new** name of the **same** memory address.

Summary:



Turn address into value with *address
Turn value into address with &value
(Address or pointer refers the same thing)

While implementing pointers you must remember that * can be used for **two** purposes.
Use of * as a **type description** (*type) and as an **operator** (*pointer) while working with pointers:

*type	*person	type description: type of a variable that pointer points to.
*pointer	*pPointer	operator: access the value of the variable that pointer is referencing

Some pointer hacks in Go:

```
kuengaPointer := &kuenga
kuengaPointer.updateName("Deki")
```

- The above two lines can be reduced to a single line of code as follows.
  `kuenga.updateName("Deki")`
  This is because Go can **internally convert** the **person** type to a **pointer** variable (value to reference) since the function **receiver** type is explicitly mentioned as a pointer pointing to person type.

  Try in go playground (online Go compiler)

- While working with **slice**, to **update** values in a slice, you do **not** need to pass the slice by reference. Just pass slice value and you can directly update the slice items.

  Ex., main.go

```go
package main

import "fmt"

func main() {
    mySlice := []string{"hello", "how", "are", "you"}
    updateSlice(mySlice)
    fmt.Println(mySlice)
}

func updateSlice(s []string) {
    s[0] = "hi"
}
```

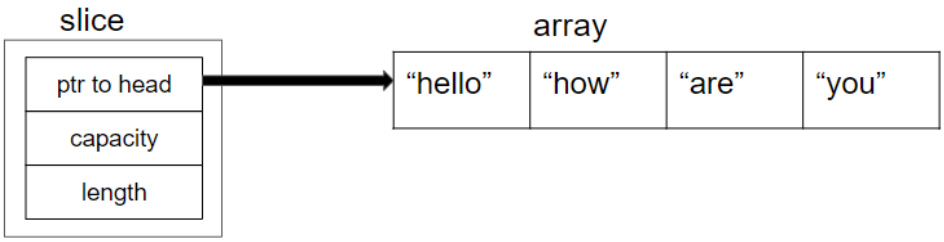
Output:

```
[hi how are you]
```

Reference Vs Value Type

When you create a slice, Go internally creates two types of data structure, slice and array.

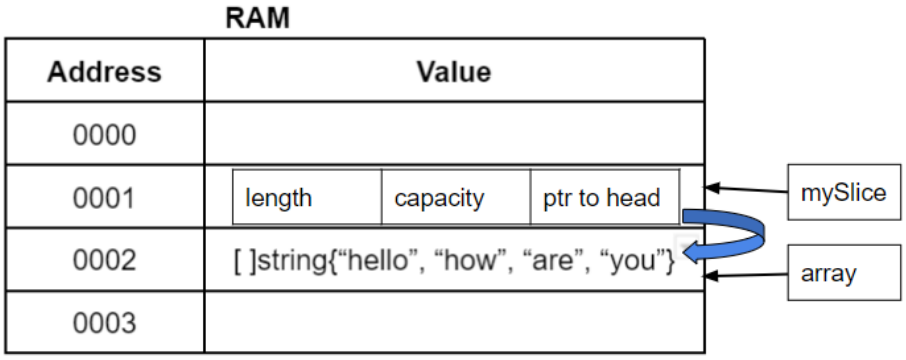
`mySlice := []string{"hello", "how", "are", "you"}`



Length: how many elements currently exist inside the slice.
Capacity: how many elements are there in the underlying array.
Ptr to head: pointer reference to an underlying array that contains the slice data.

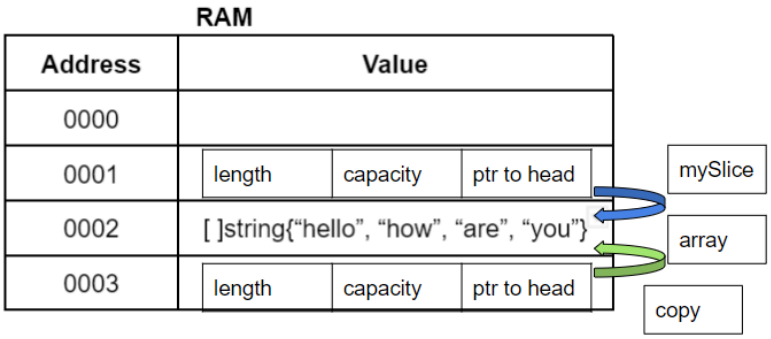
mySlice and array are stored in a separate memory location. The table below shows the memory location.

```
mySlice := []string{"hello", "how", "are", "you"}
```

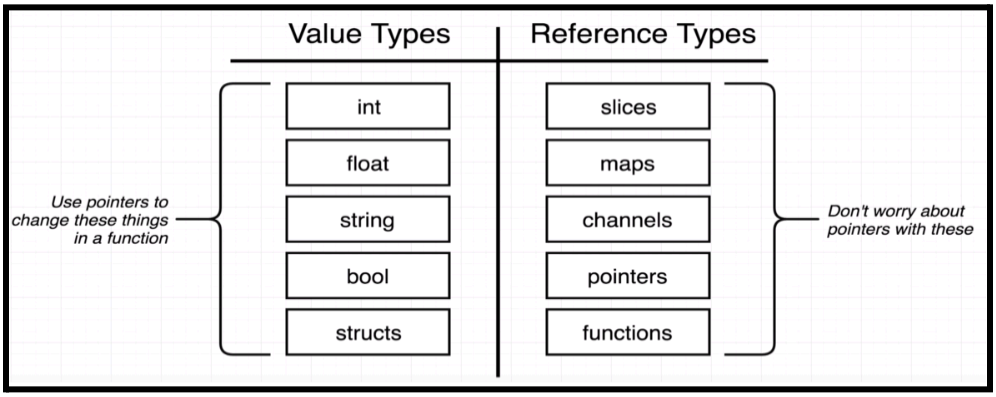



```
func updateSlice(s []string)
```

- When the `updateSlice()` is called by passing the slice, Go (pass by value) creates a **copy** of the slice (s) in a new memory location. Although it is a copy, the **pointer** reference contained in the slice still points to the **same** data stored in the array.
  This is how the data gets updated despite being in a separate memory location and not passing the pointer reference of mySlice.



If you can differentiate data types as **reference types** or **value type**s then you can easily decide whether to use pointers or not.



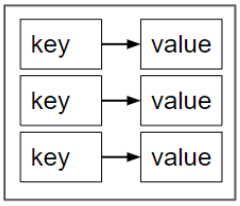
- **Slice** in Go are **reference types** therefore, you do not need to pass the reference to update the slice values in a function/method. This is why the values were successfully updated in the above slice example.

Maps

Q. What is a Map?
Similar to struct.

Is a collection of key value pairs.

Map



Map is a **data structure** that stores data in the form of **key value** pairs.
Key set should be the **same type** and of course the value set.

Syntax:

```
mapName := map[keyType]ValueType { }
```

Let's look at the implementation to understand how to declare maps and manipulate data inside maps.

- ➔ Create a new project, maps.
- ➔ Create a new file main.go and define package main.
- ➔ Create a map.

See the code below.

```
package main

import "fmt"

func main() {
   // declare and assign values to map
    colors := map[string]string{
        "red":   "danger",
        "green": "nature",
    }
    fmt.Println(colors)
}
```

Output:

```
map[green:nature red:danger]
```

Each key value pair inside the map must be separated by a **comma**.

There are two other ways to declare a map:

1. Using a **var** keyword.
   `var colors map[string]string`

   Declares an empty map.

2. Using **make** function.
   `colors := make(map[string]string)`

   .

   To assign value to a map, you can use **square brackets** ([ ]) syntax. Unlike structs you **cannot** use a **dot** operator. Why?

   `colors[“green”] = “live”`

Q. How to delete a key value pair in a map?
Use a built-in function **delete**.

Example: `delete(colors, “green”)`

```
colors := make(map[string]string)
colors["red"] = "danger"
delete(colors, "red")
fmt.Println(colors)
```

Output:

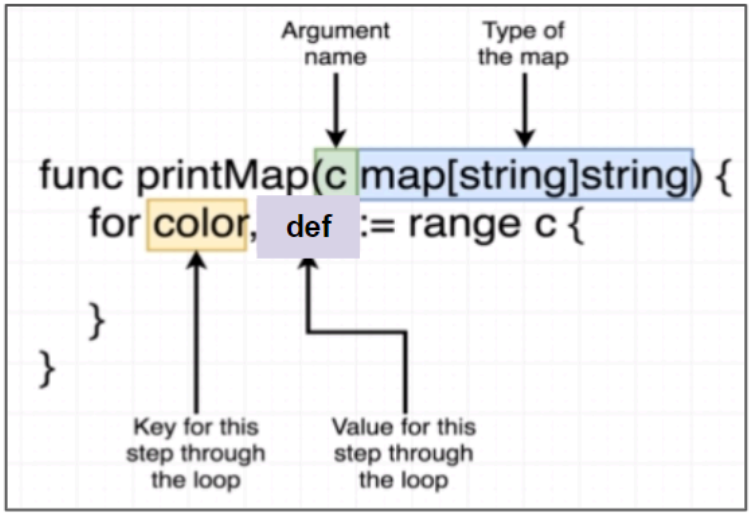
```
map[]
```

When you call the delete function, pass the map as the first argument and the key as the second argument.

Q. How to **iterate** over a map?
Same as slice. Using a for loop.

Implement the example below.

➔ Create a new function **printMap** which accepts a map. The function iterates over the map and prints every key value pair.



Code:

```go
package main

func main() {
    // 1
    colors := map[string]string{
        "red":   "danger",
        "green": "nature",
        "blue":  "sky",
    }

    printMap(colors)
}

func printMap(c map[string]string) {
    for color, def := range c {
        println("definition of " + color + " is " + def)
    }
}
```

Output:

```
definition of red is danger
definition of green is nature
definition of blue is sky
```

Difference between structs and map:

Map	Struct
All keys and values must be of the same type.	Values (fields) can be of different types.
Keys are indexed, iterable.	Does not support indexing. Not iterable.
Used to represent a collection of **related** properties.	Used to represent a thing with a lot of **different** properties.
Not compulsory to know all keys at the compile time.	You need to know all the different fields at the compile time.
Reference type	Value type
Values statically typed. That's why you can only use [ ].	Values not strongly typed. That's why you can use the dot operator.

Q. When to use one?
No framed rules as such. Just some general observations,

1. Fixed and known properties: struct
2. Unknown properties when writing the code (compile time): map
   Can add and delete later.
3. Depends on the type of application you build.

**** The End ***

Reference

Retrieved from:

https://dit.udemy.com/course/go-the-complete-developers-guide/learn/lecture/7797300#overview
https://www.baeldung.com/cs/statically-vs-dynamically-typed-languages
https://medium.com/@kellyrschroeder/the-static-around-dynamical-languages-b52a5d083192
https://speedscale.com/blog/golang-testing-frameworks-for-every-type-of-test/#:~:text=Unit%20testing%20is%20the%20most,perform%20according%20to%20their%20requirements.
https://david-yappeter.medium.com/golang-pass-by-value-vs-pass-by-reference-e48aac8b2716