BELAJAR DENGAN JENIUS GOLANG

THEORY + VISUALIZATION + CODE

GUN GUN FEBRIANZA

“Menulis sebagai Janji Bakti, menuju tanah Air Jaya Sakti”

Gun Gun Febrianza

Bandung, 14 Maret 2020

First Published 14 ***March*** 2020,

Open Library Indonesia

Sebuah konsep Perpustakaan Digital Terbuka untuk membantu mempermudah siapapun untuk mengakses ilmu pengetahuan. OpenLibrary.id adalah sebuah gerakan dan konsep pemikiran yang penulis usung sebagai wadah tempat untuk mengabdi kepada masyarakat melalui kontribusi karya tulis. Karya tulis yang diharapkan dapat membantu agar minat baca jutaan pemuda-pemudi di Indonesia terus meningkat. Sebab penulis percaya dengan membaca peluang keberhasilan hidup seseorang kedepanya akan menjadi lebih besar dan membaca dapat membawa kita ketempat yang tidak pernah kita sangka-sangka yaitu tempat yang lebih baik dari sebelumnya.

Penulis sadar gerakan ini memerlukan penulis-penulis lainya agar tujuanya bisa tercapai dan jangkauan manfaatnya bisa lebih luas lagi. Semakin banyak penulis dari berbagai bidang keilmuan akan semakin berwarna manfaat hasil karya tulis yang bisa diberikan untuk masyarakat. Maka dari itu penulis secara terbuka mengundang siapapun yang ingin bergabung menjadi penulis di gerakan *Indonesia Open Library,* agar bisa bertemu dan saling bersilaturahmi.

Orang boleh pandai setinggi langit, tapi selama ia tidak menulis maka ia akan hilang dalam masyarakat dan sejarah

- Pramoedya Ananta Toer –

Untuk teman-teman ku, rekan-rekan sebangsaku, apapun kepercayaan kalian, penulis meminta doa dari rekan-rekan supaya selalu diberi kesehatan, keselamatan dan keberkahan dalam hidup.

Agar tetap bisa menulis dan berkarya bersama sama.

Table of Contents

Contents

Chapter 1

Computer

Subchapter 1 – Komputer & Pemograman

Subchapter* 1 – *Objectives

- Mengenal ***Computer Program***
- Mengenal ***Computation***
- Mengenal ***Computer Organization***
- Mengenal ***Data Hierarchy***
- Mengenal Klasifikasi Bahasa Pemrograman

Komputer dapat melakukan kalkulasi dan *logical decision* dengan kecepatan yang sangat tinggi melampaui kemampuan manusia. Rata-rata ***Personal Computer* (PC)** hari ini dapat mengeksekusi milyaran instruksi setiap detik, namun ***Supercomputer*** dapat mengeksekusi instruksi mencapai ***Quadrillion*** per detik atau setara dengan ribuan trilyun per detik. Sangat mencengangkan.

Salah satu *supercomputer* bernama **Tianhe-2** yang dikembangkan oleh *National University of Defense Technology*, mampu melakukan **33 *Quadrillion*** kalkulasi per detik. *Approximately*, (33,86 *Petaflops*). Namun kompetisi pembangunan *supercomputer* kembali dimenangkan oleh Amerika Serikat setelah mereka membuat *supercomputer* terbaru dengan nama Summit (OLCF-4) yang memiliki kemampuan sampai 200 *Flops*.

Flops atau *floating point operations per second* adalah salah satu ukuran yang digunakan untuk mengetahui *performance* suatu komputer. Digunakan untuk melakukan *scientific computation* yang dapat membuka potensi bisnis bernilai ratusan juta *dollar*, inovasi dan

revolusi industri yang dapat membawa suatu perusahaan atau negara membuka sejarah baru.

Computer Program

Komputer memproses sebuah program yang terdiri dari serangkaian instruksi untuk melakukan suatu komputasi (*Computation*) secara spesifik. Kata spesifik mengacu pada suatu *problem domain* atau *programming domain*, menurut Robert W.Semesta dalam bukunya yang berjudul Concept of Programming Language (2016) sebuah *programming domain* terdiri dari 4 hal yaitu untuk :

1. *Scientific Application*

   Program yang dibuat untuk keperluan sains dan penelitian.

2. *Business Application*

   Program yang dibuat untuk keperluan bisnis.

3. *Artificial Intelligence*

   Program yang dibuat memiliki kemampuan kecerdasan buatan.

4. *Web Software*

   Program yang dibuat menggunakan teknologi web.

Computation

Kata Komputasi bisa berupa *numeric computation* seperti memecahkan sesuatu dengan model matematis seperti pada *system of equation* (sistem persamaan) atau *symbolic computation* seperti melakukan pencarian pada sebuah teks, memanipulasi teks, gambar dan vidio.

Computer Organization

Komputer terdiri dari sekumpulan *logical unit* :

Input Unit

Terdiri dari sekumpulan ***input device*** untuk memproduksi informasi yaitu *keyboard, touchscreen, webcam, microphone, barcode scanner* dan *mouse devices.*

Output Unit

Terdiri dari sekumpulan ***output device*** untuk menampilkan informasi yaitu *screen monitor, speaker, printer* hingga ke *oculus rift*.

Memory Unit

Memory unit seringkali disebut *memory*, *primary memory* atau ***RAM (Random Access Memory).*** Informasi yang tersimpan dalam *memory unit* bersifat ***volatile***, artinya informasi akan hilang jika komputer dimatikan.

Memory unit menjadi tempat untuk mempertahankan informasi setelah melalui *input unit*, sehingga langsung tersedia untuk diproses oleh *processor* jika dibutuhkan untuk memproduksi hasil pada *Ouput Unit*.

Arithmetic and Logic Unit (ALU)

Fungsi dari *ALU* adalah untuk melakukan kalkulasi seperti penjumlahan, pengurangan, perkalian dan pembagian.

ALU memiliki mekanisme untuk membuat keputusan yang dapat membuat komputer misal, membandingkan dua buah data dalam *memori unit* apakah data tersebut setara (*equal*) atau tidak.

Kini ALU (*Arithmetic and Logic Unit*) dikembangkan sebagai *next logical unit* untuk *CPU*.

Central Processing Unit (CPU)

CPU (*Central Processing Unit*) akan memberi sinyal pada *input unit* saat informasi dalam *memory unit* dibutuhkan untuk diproses melakukan suatu kalkulasi dan memberikan sinyal kepada *output unit* saat informasi dalam *memory unit* siap untuk digunakan pada *output device*.

Kebanyakan komputer hari ini telah memiliki lebih dari satu *CPU* sehingga dapat melakukan banyak sekali operasi secara simultan. Sebuah *Multi-core processor* memiliki lebih dari satu *processor* dalam satu ***IC Chip*** tunggal.

Sebagai contoh *dual-core processor* artinya terdapat dua *processor* dalam 1 *IC Chip* dan *quad-core* artinya terdapat 4 *processor* dalam 1 *IC Chip*.

Secondary Storage Unit

Sebuah data atau program yang sudah tidak lagi aktif digunakan biasanya akan atau dapat disimpan kedalam *storage devices* seperti *hard drive*, sampai data tersebut dibutuhkan kembali.

Informasi yang tersimpan di dalam *secondary storage device* bersifat persisten. Informasi tetap terjaga meskipun komputer dimatikan. Informasi yang dapat disimpan dalam *hard drives* dalam *dekstop* komputer bisa sangat besar melebihi 16 *Terabyte*.

Data Hierarchy

Data yang akan diproses oleh komputer memiliki bentuk data hirarki, mulai dari yang paling kecil dan tidak dapat dibagi yaitu bits sampai ke dalam bentuk yang lebih kompleks

Bit

Bit adalah kependekan dari ***Binary Digit***, unit terkecil sebuah informasi dalam mesin komputer. Satu buah *bit* dapat menampung dua nilai diantaranya adalah **0** atau **1**.

Jika terdapat 8 *bit* maka kita dapat menyebutnya sebagai **1 *byte***.

Byte

Byte adalah kependekan dari ***Binary Term***, sebuah unit penyimpanan yang sudah memiliki kapabilitas paling sederhana untuk menyimpan sebuah karakter tunggal.

1 *byte* = sekumpulan *bit* (terdapat delapan bit). Contoh : 0 1 0 1 1 0 1 0

1 *byte* dapat menyimpan karakter contoh : `'A'` atau 'x' atau `'$'`

Bytes

Byte adalah unit yang dipat digunakan untuk menyimpan informasi, seluruh penyimpanan diukur menggunakan *bytes*.

Tabel 1 Memory Standard Metrics

Number of Bytes	***Unit***	***Representation***
1	*Byte*	*One Character*
1024	*KiloByte (Kb)*	*Small Text In notepad*
1,048,576	*MegaByte (Mb)*	*Ebook*
1,073,741,824	*GigaByte (Gb)*	*Movie*
1,099,511,627,776	*TeraByte (Tb)*	*Archive*
Approximately 10^{15}	*PetaByte (Pb)*	*Big Data*
Approximately 10^{18}	*ExaByte (Eb)*	*Big Data*
Approximately 10^{21}	*ZettaByte (Zb)*	*Big Data*

Character

Data dalam bentuk *bits* tidak mudah untuk dikelola sehingga perlu bentuk lain yang dapat digunakan manusia dan mempermudah proses pengelolaan informasi. Untuk

mewujudkan hal tersebut data dalam *bits* harus bisa direpresentasikan dalam bentuk *character*. Seperti *decimal digit* (0-9), *letter* (A-Z dan a-z), dan *special symbol* (!@#$%^&*()-=_+). *Digit, letter* dan *symbol* disebut dengan *characters*.

Characters Set adalah sekumpulan *characters* yang digunakan untuk menulis program dan merepresentasikan sebuah informasi. Dikarenakan komputer memproses informasi dalam bentuk 1 dan 0, maka sebuah *character* dapat direpresentasikan menggunakan 1 dan 0.

Character adalah unit terkecil dalam sistem teks dan memiliki makna.

Sekumpulan *character* dapat membentuk *string* yang selanjutnya dapat digunakan untuk memvisualisasikan suatu bahasa verbal secara digital. Contoh *character* adalah abjad, angka dan simbol lainya.

Gambar 1 Grapheme

ASCII

Komputer merepresentasikan sebuah data dengan *number*, di awal pengembangan komputer tepatnya sekitar tahun 1940. Penggunaan teks dalam komputer untuk disimpan dan dimanipulasi dapat dilakukan, dengan cara merepresentasikan abjad dalam alfabet menggunakan *number*. Sebagai contoh angka 65 merepresentasikan huruf A dan angka 66 merepresentasikan huruf B hingga seterusnya.

Pada tahun 1950 saat komputer sudah semakin banyak digunakan untuk berkomunikasi, standar untuk merepresentasikan *text* agar dapat difahami oleh berbagai model dan *brand* komputer diusung.

ASCII (*American Standard Code for Information Interchange*) adalah karya yang diusung, pertama dipublikasikan pada tahun 1963. Saat pertama kali dipublikasikan *ASCII* masih digunakan untuk *teleprinter technology*. *ASCII* terus direvisi hingga akhirnya *7-bit ASCII Table* diadopsi oleh *American National Standards Institute* (*ANSI*).

Dengan *7-bit* maka terdapat 128 *unique binary pattern* yang dapat digunakan untuk merepresentasikan suatu karakter. Kita dapat merepresentasikan ***alphanumeric*** (abjad a-z, A-Z, angka 0-9, dan *special character* seperti "!@#$%^&*").

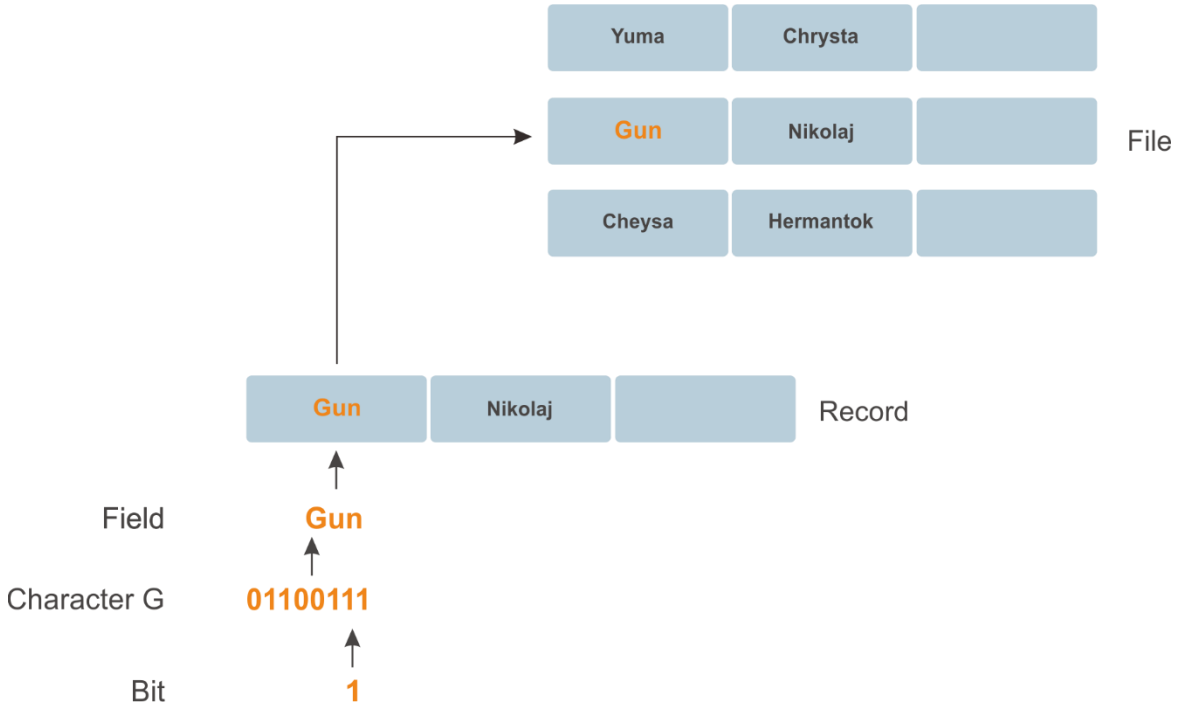
Pada gambar di bawah ini huruf kapital G memiliki representasi dalam bentuk biner 100 0111 (7 *binary digit*) dan huruf kapital F memiliki representasi dalam bentuk *binary pattern* 100 0110 :

100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H

Gambar 2 Sample ASCII Code

Pada huruf kapital G angka 107 adalah representasi dalam *octal numeral system*, angka 71 adalah representasi dalam *decimal numeral system* dan angka 46 adalah representasi dalam *hexadecimal*. Representasi tidak hanya dalam bentuk *binary*. Untuk *table ASCII* lebih lengkapnya anda dapat melihat di wikipedia.

Field



Gambar 3 Data Hierarchy

Character hanya merepresentasikan sekumpulan *bit*, *field* merepresentasikan serangkaian *character* atau *bytes* yang memiliki makna lebih luas. Misal sebuah *field* dapat digunakan untuk merepresentasikan nama seseorang atau umur seseorang.

Record

Beberapa *field* yang saling berhubungan atau memiliki kesamaan dapat digunakan untuk membangun suatu *record*. Misalkan dalam sistem kependudukan, anda tentu memiliki beberapa *fields* diantaranya adalah nama, tempat & tanggal lahir, jenis kelamin, status dan sebagainya. Ketika masing-masing *field* terisi maka kita akan membentuk suatu *record*.

Files

Secara umum sebuah *file* dapat berupa ***arbitrary data*** dengan ***arbitrary format***. Sebagai contoh dengan ***notepad.exe*** kita dapat membuat sebuah tulisan berisi ***plaintext*** kemudian menyimpanya ke berbagai format seperti, **.txt, .md, .html** atau bahkan menyimpanya dengan format yang tidak dikenali oleh program dalam suatu sistem operasi (misal dengan format .**xxxasdqwe**).

Di beberapa sistem operasi sebuah *file* dikatakan dengan serangkaian *bytes* yang membentuk suatu *file*.

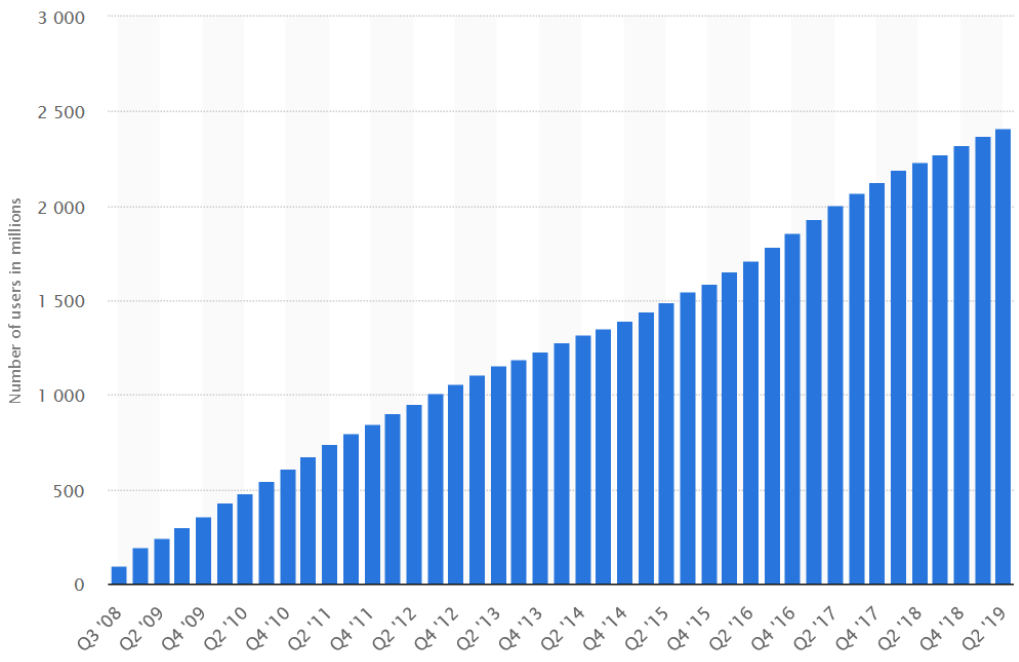
Database

Database adalah sekumpulan data yang teroganisir dengan baik agar bisa diakses dan dimanipulasi. Model pembuatan *database* yang paling terkenal adalah *relational database*, data disimpan ke dalam sebuah *table* yang berisi *record* dan *fields*.

Big Data

Big Data adalah terminologi yang menjelaskan sebuah fenomena data dengan *volume* yang sangat besar (*High Volume Data*), data dengan struktur yang bervariasi (*High Varied Data*) dan data diproduksi dengan kecepatan yang sangat tinggi (*High Velocity Data*).

Ketersediaan internet, kecepatan internet, *web application* dan produksi perangkat elektronik seperti *smartphone*, komputer, laptop dan *tablet* menciptakan ledakan data dengan pertumbuhan yang sangat cepat.



Gambar 4 Facebook Monthly Active User [1]

Dengan 2,4 milyar pengguna aktif pada kuarter kedua tahun 2019 *facebook* menjadi *platform* jejaring sosial (*social media*) terbesar didunia. Sebuah *platform social media*, dapat memproduksi berbagai jenis data seperti gambar (*image*), vidio (*video*), teks (*text*) dan suara (*voice*) dengan kecepatan yang sangat tinggi.

Operating System

Sistem operasi adalah program yang mengendalikan perangkat keras komputer dan sumber perangkat lunak yang ada di dalamnya, serta menyediakan layanan umum untuk program program yang ada di dalam komputer.

Terdapat banyak sekali sistem operasi yang ada hari ini mulai dari sistem operasi *Unix*, *Windows*, *Linux*, OS X dan sebagainya. Masing masing sistem operasi memiliki kelebihan dan kekurangan. Saat ini sistem operasi telah didominasi oleh 32 Bit dan 64 Bit *Operating System*.

Pada sistem operasi 32 bit penggunaan kapasitas RAM dibatasi sampai 4096 MB RAM, 2^{32} = 4,294,967,296 *bytes* untuk setiap *process*. Pada sistem operasi 64 bit penggunaan kapasitas RAM dibatasi sampai 16 Exabytes 2^{64} = 18,446,744,073,709,551,616 *bytes*.

Programming Language

Selalu ingat, komputer hanya memahami satu bahasa yaitu ***Machine Language***.

Bahasa pemrograman adalah bahasa formal untuk mengekspresikan suatu komputasi yang akan dikerjakan oleh sebuah mesin yaitu komputer. Bahasa formal adalah bahasa yang didesain secara khusus oleh seseorang untuk *specific applications*.

Bahasa pemrograman secara tradisional dilihat dari tiga segi aspek : [2]

1. *Syntax* sebagai struktur dari bahasa pemrograman.
2. *Semantic* sebagai makna dari bahasa pemrograman.
3. *Pragmatic* sebagai implementasi dari bahasa pemrograman.

Bahasa pemrograman di desain ***human-readable*** untuk mempermudah kita dalam memberikan instruksi kepada mesin komputer. Dalam pemograman terdapat dua cara untuk menterjemahkan ke dalam bahasa mesin yaitu :

1. Menggunakan ***Interpreter***
2. Menggunakan ***Compiler***

Programming Language Abstraction

Berdasarkan abstraksi, bahasa pemrograman bisa diurut menjari tiga kategori yaitu *Machine Language*, *Assembly Language* dan *High Level Language* [3]. Berdasarkan *translator* terdapat dua bahasa pemrograman yaitu *Compiled Language* dan *Interpreted Language*.

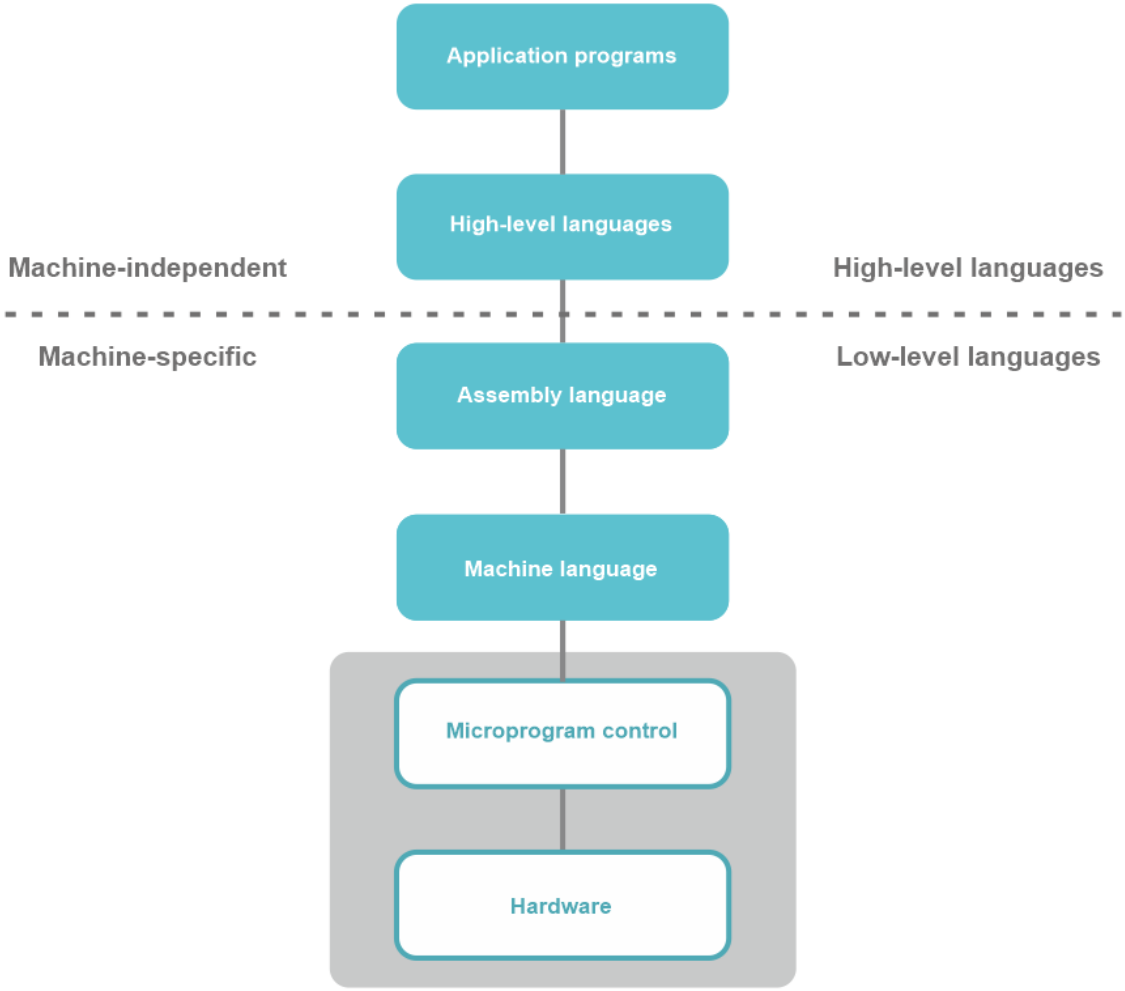
Machine Language

Machine Language adalah bahasa yang mampu difahami secara langsung oleh mesin komputer. ***Machine code*** atau *Machine Language* adalah sekumpulan instruksi atau *set of instruction* yang langsung dieksekusi oleh *CPU* (*Central Processing Unit*).

Seluruh instruksi informasinya direpresentasikan dalam bentuk angka 1 dan 0 yang diterjemahkan dengan sangat cepat oleh komputer. Sebuah program yang dibuat dengan bahasa pemrograman *high level* harus diterjemahkan terlebih dahulu kedalam *form* yang bisa difahami oleh suatu mesin komputer.

Machine Language memiliki karakteristik ***Machine Dependant***, bahasa mesin tertentu hanya dapat berjalan pada mesin komputer tertentu. Sebab setiap *processor* atau *processor family* memiliki ***Instruction Set Architecture*** yang berbeda-beda.

Machine code bersifat *platform-specific*, sehingga jika dikompilasi pada sistem operasi *windows* x86 maka program hanya akan berjalan pada sistem operasi *windows* x86.



Gambar 5 Language Abstraction

Assembly Language

Assembly Language adalah bahasa yang mempermudah para *programmer* agar mereka bisa terfokus memecahkan masalah daripada fokus pada mengingat formula 1 dan 0. Bahasa ini dikenal dengan sebutan ***Second Generation Language*** atau disingkat dengan sebutan *asm*.

Bahasa ini hampir dekat dengan bahasa mesin dengan ciri khas *mnemonic* pada setiap *syntax* bahasa *assembly*. Seperti *syntax Add* (kependekan dari *addition*), *Sub* (kependekan

dari *subtraction*), *Sum* (*summarize*) dan sebagainya berikut dengan *storage location*. Sehingga tidak lagi bermain di *level binary digit*.

Setiap instruksi memiliki tujuan spesifik untuk melaksanakan suatu tugas, seperti *load*, *jump* atau *ALU* (*Arithmetic Logic Unit*) *operation* seperti *arithmetic operation* (*ADD, SUBTRACT, INCREMENT, DECREMENT* dan lain-lain) atau *logic operation* (AND, OR, XOR dan lain lain) di dalam sebuah *CPU Register* atau *Memory*.

Sebuah *program* yang diberi nama ***Assembler*** digunakan untuk menerjemahkan *assembly language* ke *machine language*. Program yang dibuat menggunakan *assembly language* bersifat *machine dependent* yaitu selalu mengacu kepada sebuah tipe *CPU*. Setiap

CPU memiliki *machine language* sendiri dan di *level* yang lebih tinggi juga memiliki *assembly language* sendiri.

High Level Language

High Level Language disebut juga ***Machine Independant Language*** dikenal dengan sebutan *third generation language*. *High Level language* adalah bahasa pemrograman dengan *strong abstraction* dari kedetailan tentang komputer. Diciptakan untuk menyederhanakan pemrograman.

Abstraksi ini membuat proses pengembangan sebuah *program* menjadi lebih sederhana dan prosesnya mudah difahami. Jumlah abstraksi yang disediakan menjelaskan seberapa tinggi level pemrograman itu sendiri.

Pada Tahun 1960, bahasa pemrograman tingkat tinggi yang menggunakan *compiler* dikenal dengan sebutan *Autocodes* [4]. Contoh *Autocodes* adalah *Fortran* dan *Cobol*. Bahasa pemrograman tingkat tinggi pertama didunia adalah *Plankalkül* yang dibuat oleh Konrad Zuse [5].

High Level Language Programming seperti bahasa C, C++, *Python* dan *Java* mempunyai satu ke banyak (*one to many*) relasi dengan *Assembly Language* dan *Machine Language*. Disinilah kedalaman *abstraction* dari sebuah bahasa tingkat tinggi dinilai. Sebuah *Statement* satu baris yang dibuat menggunakan C++ akan mengembang jika diterjemahkan kedalam *Assembly Language* dan *Machine-language Instruction***.**

Di bawah ini adalah dua buah *statement* dengan C++ *code* yang di dalamnya terdapat *Arithmetic Operation* dan hasilnya ditetapkan kedalam sebuah *Variable* dengan asumsi X dan Y adalah *Integer* :

```
int Y;

int X = (Y + 4) * 3;
```

Jika diterjemahkan kedalam *Assembly Language* maka dibutuhkan banyak *statement code* seperti di bawah ini.

```
mov eax,Y ; Pindahkan Y ke EAX register
add eax,4 ; Tambah nilai 4 ke EAX register
mov ebx,3 ; Pindahkan 3 ke EBX register
imul ebx ; Kalikan EAX dengan EBX
mov X,eax ; Pindahkan EAX ke X
```

Selanjutnya dari bahasa *assembly* akan diterjemahkan kedalam bahasa mesin dengan relasi satu ke satu, artinya dari setiap satu instruksi yang dibuat menggunakan bahasa *assembly* mengacu pada satu *Machine-language Instruction* yang selanjutnya menjadi sinyal listrik (*Digital Signal*).

Register adalah nama sebuah lokasi di dalam *CPU* yang menyimpan sebuah hasil sementara [6].

Compiled Language

Hal yang menjadi pembeda pada sebuah bahasa pemrograman sebelum dieksekusi adalah ***translator*** yang digunakanya. Pada ***Compiled language*** proses terjemah langsung ke ***machine readable binary code*** oleh sebuah program bernama kompiler.

Hasilnya program bisa langsung dieksekusi tanpa membutuhkan kembali ***human readable source code*** [7].

Beberapa bahasa pemrograman yang termasuk kedalam ***compiled language*** adalah *C, C++, Pascal, Rust, Lisp, Julia, Go, Haskel, Basic, Fortran* dan *Algol*.

Interpreted Language

Pada ***interpreted language*** program yang ditulis langsung dieksekusi dari kode sumber. Pemrograman yang berasal dari ***interpreted language*** seringkali disebut dengan ***scripting language***.

Beberapa bahasa pemrograman yang termasuk kedalam ***interpreted language*** adalah ***ECMAScript*** (***Javascript, Actionscript*** & ***Jscript***), ***perl, ruby, php, python, smalltalk*** dan ***R Programming***.

Hybrid Language

Selain ***compiled language*** dan ***interpreted language*** juga terdapat ***hybrid language*** yaitu bahasa yang diproses secara ***compiled*** dan ***interpreted***. Bahasa yang termasuk kedalam ***hybrid language*** adalah bahasa java.

Sistem bahasa java memiliki kedua aspek sekaligus yaitu sebagai ***compiled*** dan ***interpreted language*** [8]. Sebelum ***java*** program bisa berjalan, bahasa dikompilasi terlebih dahulu kedalam ***bytecode*** yang selanjutnya proses ***interpreted bytecode*** dilakukan di atas ***Java Virtual Machine*** (***JVM***)**.**

JVM adalah ***software processor*** yang memiliki peran sebagai ***buffer*** antara ***bytecode*** dan ***microprocessor***. Manfaatnya adalah bahasa java bisa berjalan diberbagai mesin komputer.

Subchapter 2 – Kompiler & Interpreter

Subchapter 2 – Objectives

- Mengenal ***Compiler***
- Mengenal ***Interpreter***
- Mengenal ***Compilation Process***
- Mengenal ***Runtime Infrastructure***

Compiler

Kompiler adalah sebuah *program* komputer yang menerjemahkan sebuah *program* yang ditulis menggunakan suatu bahasa pemrograman (*source language*) kedalam bahasa yang *equivalent program* (*target language*) pada sebuah *platform*. Platform adalah sebuah operating system dan computer processor tertentu. Proses penerjemahan itu sendiri disebut dengan *compilation* .

Terminologi *compiler* pertama kali disebut oleh Grace Hopper seorang *inventor* A-0 *System*, COBOL dan *term compiler* itu sendiri. A-0 *System* adalah *compiler* pertama di dalam dunia komputer yang dibuat pada tahun 1952 [9].

Self-hosting Compiler

Kompiler adalah sebuah program, sama seperti program yang lainya sebelum kompiler bisa digunakan kompiler juga harus dikompilasi terlebih dahulu. Kompiler pertama dibuat dengan bahasa mesin (*machine language*). Setelah kompiler diciptakan menggunakan bahasa mesin selanjutnya bahasa yang lebih tinggi bisa dikompilasi.

Selanjutnya dalam bahasa yang lebih tinggi kita bisa menciptakan sebuah kompiler yang lebih kompleks namun bahasa yang mampu dikompilasinya semakin *human readable*. Proses ini terus dilakukan secara kontinyu sampai membentuk sebuah *advanced compiler*.

The Self Hosting compiler pertama di dunia yaitu sebuah *compiler* yang mampu melakukan kompilasi kode sumbernya sendiri ditulis menggunakan bahasa LISP pada tahun 1962. Kode tersebut ditulis oleh Hart dan Levin di MIT [10].

Adapun bahasa pemrograman yang telah mengimplementasikan *self-hosting compiler* adalah bahasa *Visual Basic* yang digunakan untuk membuat *Compiler* yang diberi nama *Microsoft Roslyn* dan bahasa *Python* yang digunakan untuk membuat sebuah *Interpreter* yang diberi nama *Pypy*.

Assembler

Sebuah *Assembler* bekerja sebagaimana sebuah *compiler* yaitu menerjemahkan sebuah *source language* ke dalam *target language*, hanya saja *assembler* bekerja di *level* yang paling bawah yaitu untuk menerjemahkan bahasa *assembly* ke dalam bahasa mesin.

Cross-compiler

Cross Compiler adalah sebuah *compiler* yang mampu berjalan disatu mesin namun mampu memproduksi *object code* untuk mesin lainya [11]. Sebuah *cross compiler* mampu memproduksi hasil kompilasi untuk berbagai *platform* dan *target machine* melebihi dari *cross compiler* itu sendiri dalam hal kemampuan *running system*.

Sebagai contoh pada GCC karya Richard Stallman terdapat banyak sekali koleksi kompiler yang bersifat *cross compiler* karena *output* kompilasinya bisa digunakan di berbagai *platform* dan *target machine*.

Just-in-Time Compiler

Just in Time (JIT) Compiler adalah sebuah *compiler* yang akan menerjemahkan bahasa *intermediate language* kedalam *machine language*. *JIT compiler* dikenal juga dengan

sebutan *Dynamic Translation*, pada *.Net Framework* sebuah *managed code* selanjutnya bisa dikompilasi menggunakan *JIT Compiler* di atas *Common Language Runtime (CLR)*.

V8 dan sebagian besar ***modern javascript engine*** sudah menggunakan ***Just-in-time compilation*.**

Decompiler

Decompiler adalah sebuah program yang mampu menerjemahkan *low level language* ke *high level language*. Biasanya digunakan untuk membaca *source code* dari sebuah program yang telah dikompilasi sehingga bisa menimbulkan resiko pada *intellectual property* dari program tersebut.

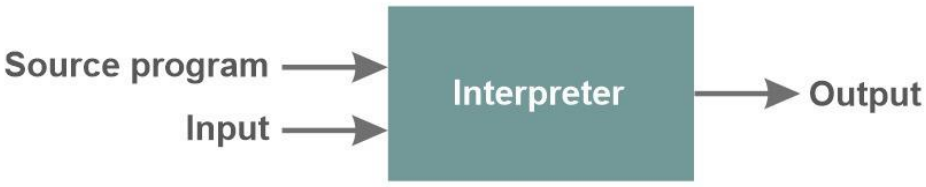
Solusi pencegahanya adalah dengan teknik *obfuscate* menggunakan program yang disebut dengan *obfucastor* dan *cyrptography* menggunakan program yang disebut dengan *crypter* agar *source code* tidak bisa dibaca ketika seseorang yang sudah *expert* dibidang *reverse engineering* mencoba melakukan *decompilation*.

Reverse engineering adalah cabang ilmu yang mempelajari bagaimana suatu *application* dalam ranah *software engineering* bisa dibuat, sehingga bisa diketahui proses pembuatanya agar bisa diduplikasi, manipulasi dan dikembangkan lebih baik lagi.

Selain *decompilation* juga terdapat istilah *dissasembly*, perbedaanya adalah pada *dissasembly* kode yang didapatkan adalah bahasa *assembly* sementara pada *decompilation* bahasa *high level* [12] atau *intermediate*.

Interpreter

Interpreter adalah sebuah *language processor* yang langsung mengeksekusi operasi yang ada di dalam *source program* dan *input* yang diberikan oleh *user*. Sebuah *interpreter* menerjemahkan satu *statement* dari *high level language* kedalam *machine code* dan langsung mengeksekusinya, kemudian menerjemahkan kembali *statement* selanjutnya sampai selesai.

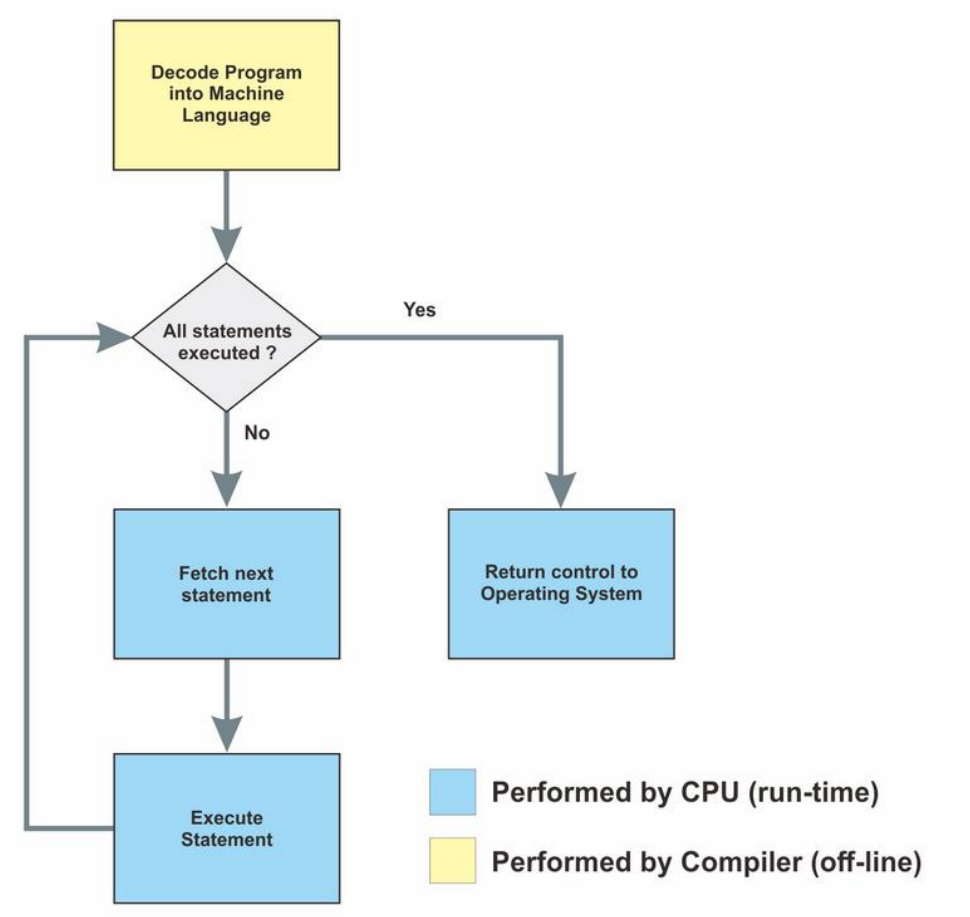


Gambar 6 Proses Intepreter

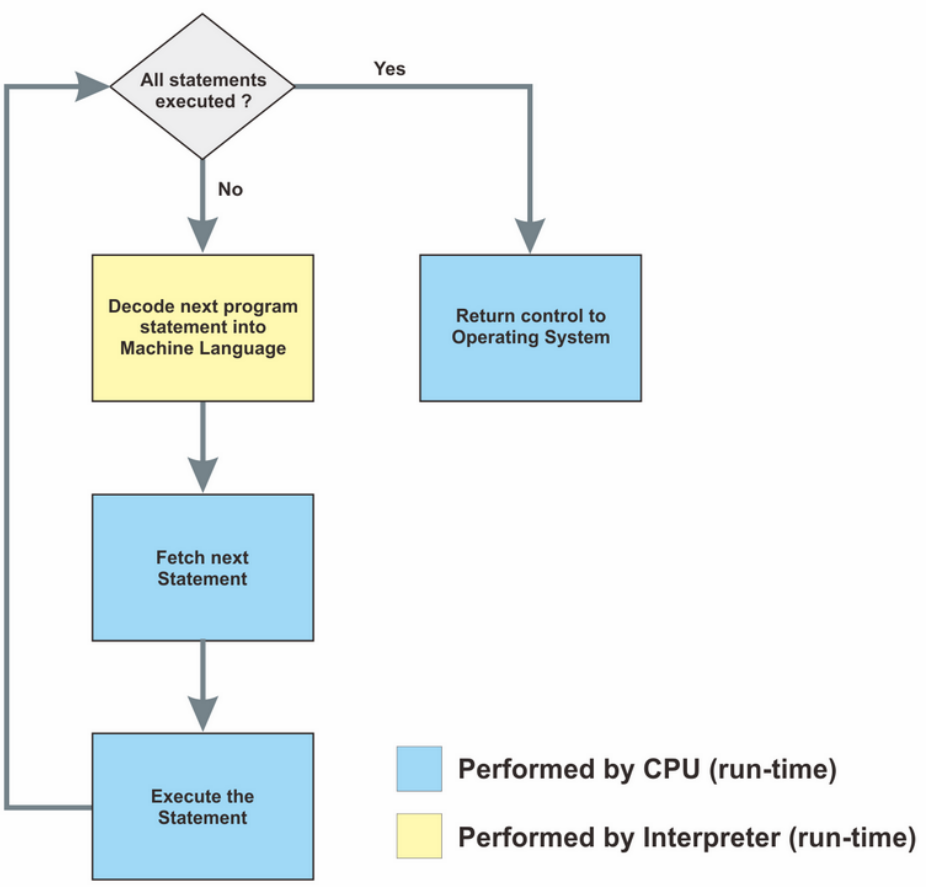
Sebuah interpreter dan kompiler secara umum akan menerjemahkan sumber kode dalam bahasa tingkat tinggi kedalam bahasa mesin agar bisa dieksekusi oleh CPU.

Kompiler menerjemahkan seluruh sumber kode sekaligus dalam satu fase kompilasi, hasil program yang telah dikompilasi dieksekusi dengan mode *fetch-execute cycle* sementara pada interpreter kode sumber harus dilakukan penerjemahan terlebih dahulu secara *statement by statements*.

Program yang dibuat menggunakan interpreter dieksekusi dengan mode *decode-fetch-execute cycle*, proses *decode* dilakukan oleh interpreter sendiri selanjutnya *fetch-execute* dilakukan oleh CPU. Proses *decode* pada interpreter membuat eksekusi program menjadi lebih lambat jika dibandingkan dengan kompiler. Cara aman untuk mengeksekusi sebuah instruksi dalam CPU adalah menggunakan interpreter dan melaksanakan tugas yang dibutuhkan saja [13]. Di bawah ini adalah *flowchart* perbandingan eksekusinya.



Gambar 7 Proses eksekusi pada kompiler



Gambar 8 Proses eksekusi pada interpreter.

Pada fakta *flowchart* di atas *interpreter* harus melewati tahap *decode* pada setiap *statement* terlebih dahulu agar bisa dieksekusi oleh *CPU*.

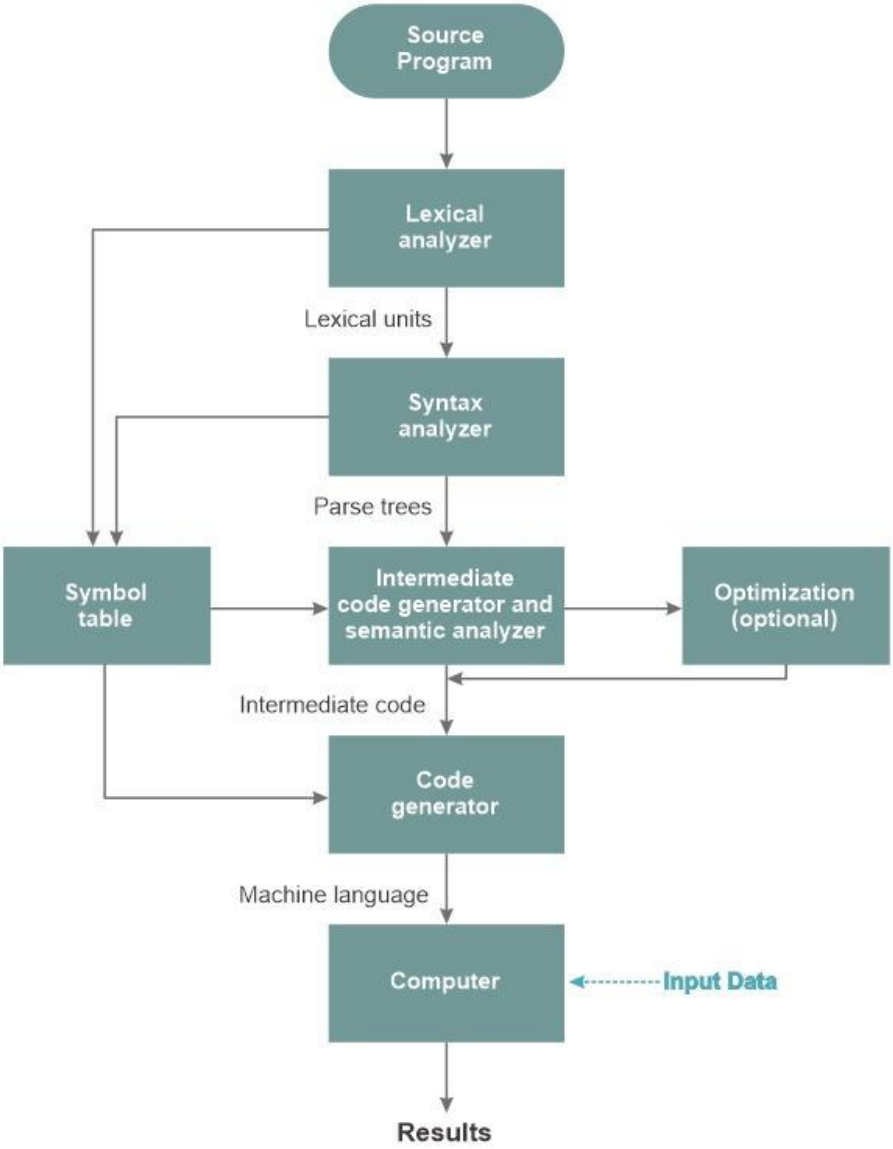
Tabel 2 Perbedaan Kompiler dan Interpreter

Kompiler	***Interpreter***
Menerjemahkan seluruh sumber kode sekaligus.	Menerjemahkan sumber kode baris perbaris.
Proses penerjemahan sumber kode cenderung lebih lama namun waktu eksekusi cenderung lebih cepat.	Proses penerjemahan sumber kode cenderung lebih cepat namun waktu eksekusi cenderung lebih lambat.

Membutuhkan penggunaan memori lebih banyak untuk menampung keluaran *intermediate object*.	Memori lebih efisien karena tidak memproduksi *intermediate object*.
Keluaran program yang dihasilkan bisa digunakan tanpa harus melakukan penerjemahan ulang.	Tidak ada keluaran program.
Kesalahan sumber kode ditampilkan setelah sumber kode diperiksa.	Kesalahan sumber kode ditampilkan setiap kali instruksi dieksekusi.

Compilation Process

Di bawah ini adalah alur dari *compilation process* yang terjadi saat kita melakukan kompilasi pada sebuah *source code*, terdapat delapan fase mulai dari desain algoritma pada *source code, lexical analyzer, syntax analyzer, intermediate code generator, symbol table, optimization* dan *code generator*.



Gambar 9 Compilation Process

Source Code

Source Code adalah serangkaian *statement* yang ditulis dengan suatu bahasa agar mudah untuk difahami dan ditulis oleh manusia sebagai *input* untuk kompilasi. Pada *source code* juga terdapat *license* yang digunakan untuk melindungi *intellectual property* dari sebuah *source code*.

Terdapat dua paradigma dalam pemberian *license* pada *source code* yaitu :

1. *Free Software* dan
2. *Proprietary Software*.

Pada *Free Software, source code* bebas untuk digunakan, didistribusikan, dimodifikasi dan dipelajari sementara pada *Proprietary Software*, *source code* bersifat rahasia, secara *private* dimiliki oleh seseorang dan biasanya dilindungi oleh hukum karena memiliki hak cipta.

Lexical Analyzer

Lexical Analyzer adalah salah satu program di dalam kompiler yang digunakan untuk melakukan analisis leksikal pada fase pertama kompilasi. *Lexical Analyzer* akan menerima masukan berupa *source code* dalam bentuk *character stream*.

Selanjutkan *character stream* akan diproses untuk memproduksi sebuah *token*, setiap *token* mempunyai *pattern* yang akan menghasilkan *lexeme* yang berbeda beda untuk menghasilkan sebuah *keyword, identifier, number, float, string* dan sebagainya.

Contoh *token* dan *lexeme* :

Tabel 3 Token & Lexeme

Token	***Lexeme***
Keyword	int, string, float64, true
Identifier	*Variable1, var, var_1*
Semicolon	;
Left Parenthesis	(
Right Parenthesis	)

Syntax Analyzer

Language adalah sekumpulan kalimat yang valid, sebuah kalimat yang terdiri dari frase, sebuah frase terdiri dari sub-frase dan kosa kata. Sebuah program yang memiliki kemampuan mengkonversi kalimat tersebut kedalam bahasa yang lain disebut sebagai *Translator*.

Agar bisa bereaksi dengan benar *Translator* harus sudah mengenali kalimat yang valid, frase, sub-frase dari bahasa tersebut. *Program* yang mengenali sebuah bahasa disebut dengan *Syntax Analyzer* [14].

Syntax Analyzer adalah program di dalam kompiler yang digunakan untuk melakukan analisis sintak dengan menerima keluaran dari *lexical analyzer* yaitu serangkaian *token*. *Syntax Analysis* seringkali disebut *parsing* [15].

Selanjutnya serangkaian *token* tersebut akan diuji susunannya, dengan mencocokan *grammar* yang telah ditetapkan untuk membangun hirarki terstruktur yang disebut dengan *parse tree* atau *syntax tree*. Jika susunanya sama maka tinggal masuk ketahap kompilasi selanjutnya diproses oleh *semantic analyzer* untuk mendeteksi *semantic error*, namun jika susunanya tidak sama maka telah terjadi *syntax error*.

Syntax analyzer digunakan untuk menerima keluaran dari hasil leksikal analisis. Hasil keluaran dari analisis leksikal akan diproses yang selanjutnya hasil dari proses sintak analisis adalah representasi *tree* dari sumber kode.

Symbol Table

Symbol Table menyediakan *database* yang akan digunakan saat proses kompilasi. Konten primer yang ada pada *symbol table* adalah *type* dan *attribute information* yang *user* gunakan di dalam *program*.

Informasi ini disimpan di dalam *symbol table* yang berasal dari *semantic analyzer* sebagai bahan untuk melakukan *semantic analysis* dan pertimbangan pada *code generator*.

Tabel 4 Contoh Symbol Table

Primitype_type	***Identifier***	***Value***
string	*variable1*	hai dunia
int	*Var_1*	666
bool	*X___Value*	benar
Float64	*_Var_1_*	88.8

Intermediate Code Generator

Intermediate Code Generator memproduksi *Intermediate Code Representation* dalam bentuk bahasa yang lain di dalam *Level Intermediate*, bisa berupa *Assembly Language* atau suatu *code* yang satu *step* lebih tinggi dari *Assembly Language*.

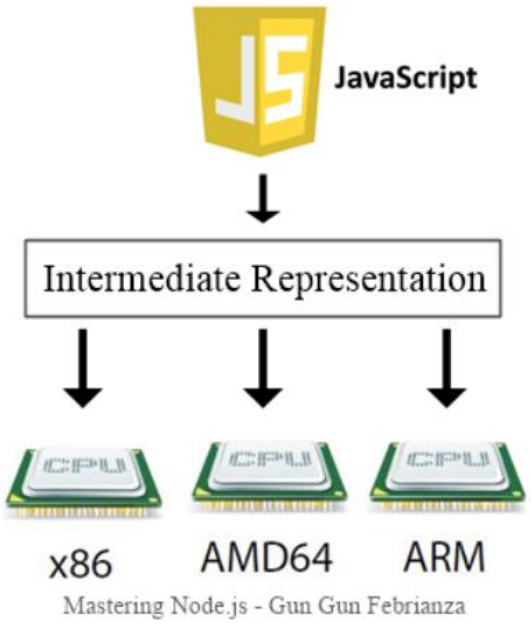
Pada pemrograman C#, *Intermediate Code Generator* akan menampung dan memproduksi *IL Assembly* sesuai dengan sintak dan semantik dari sumber kode.

Bytecode

Pada node.js, tepatnya sebelumnya *V8 Javascript Engine* masih menggunakan *crankshaft* pada versi 5.8, tim pengembang V8 harus menulis *architecture-specific code* atau bahasa *assembly* untuk 9 *platform* yang didukung seperti *windows x86, windows x64, linux x64, linux-arm x64, solaris x64* dan lain lainya yang anda bisa lihat di dokumentasi *node.js*.

Tim pengembang V8 juga harus memelihata lebih dari 10 ribu baris kode untuk masing-masing *chip architecture,* setiap kali pengembangan baru dibuat semuanya harus di *port*

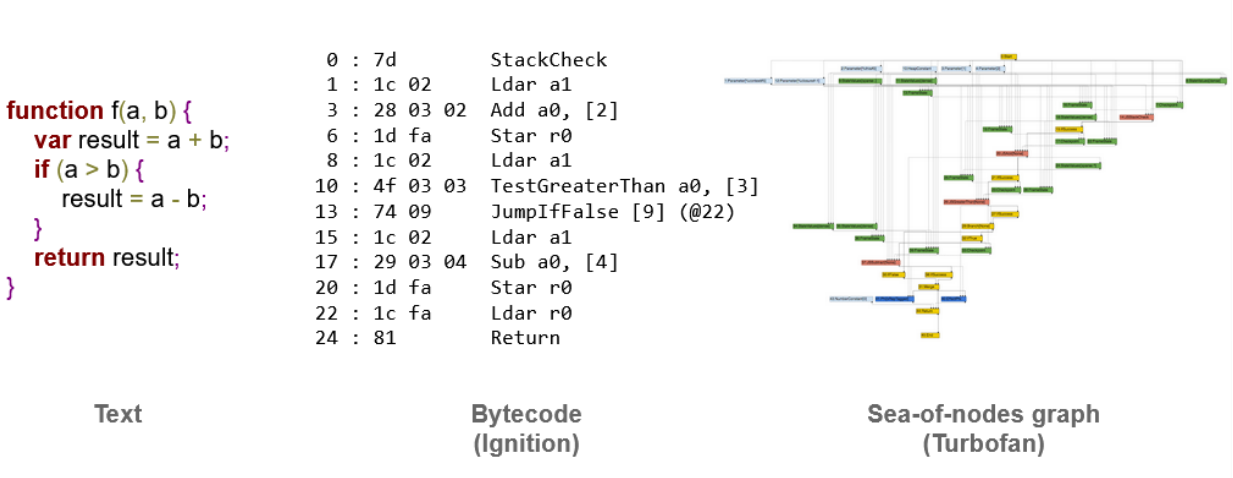
ke *architecture* yang didukung. Artinya setiap ***bahasa assembly*** untuk setiap *architecture* harus disediakan ketika pengembangan baru dibuat. Benar benar merepotkan ya?



Gambar 10 Intermediate Representation

Jadi langkah memproduksi *bytecode* sebagai *intermediate representation* sudah tepat, kita bisa dengan mudah mentargetkan *optimized machine code* untuk *platform* yang didukung oleh *V8 Javascript Engine*.

Bytecode menyediakan eksekusi model yang lebih *clean* dan tidak rentan kesalahan (*less error-prone*) saat melakukan *deoptimization*. Ukuran ***bytecode*** lebih ringan 50% sampai 25% jika dibandingkan dengan ukuran *machine code* yang diproduksi dari kode *javascript* yang ***equivalent***.



Gambar 11 Javascript Code, Bytecode, & TurboFan

Semantic Analyzer

Semantic Analyzer adalah program di dalam kompiler yang digunakan untuk melakukan analisis semantik. *Semantic Analyzer* akan melakukan cek *error* dimana *error* tersebut adalah tipe *semantic error* yang tidak bisa dideteksi oleh *syntax analyzer*.

Sebagai contoh jika terdapat sebuah *variable* dengan *identifier* yang sama hal ini tidak bisa dideteksi oleh *syntax analyzer* tapi mampu dideteksi oleh *semantic analyzer* dengan bantuan *symbol table*.

Optimization

Optimization dilakukan agar *program* memiliki peningkatan dengan membuat ukuran kapasitasnya kecil dan memiliki eksekusi yang cepat, biasanya *optimization* dilakukan pada *intermediate code* karena jika *optimization* dilakukan di *level machine language* sangat sulit, sehingga kebanyakan *optimization* dilakukan pada fase *intermediate code*.

Code Generator

Code Generator adalah program untuk melakukan *code generation* dimana pada fase ini target kode yang ingin dihasilkan tercapai. *Code Generation* dilakukan setelah fase

semantic analysis berhasil dilakukan dan dipastikan tidak memiliki *error* lagi. Selanjutnya *code generator* dieksekusi untuk memproduksi *target language*.

Runtime Infrastructure

Ada berbagai macam kompiler yang eksis hari ini, contohnya pada bahasa *java* hasil kompilasi akan diubah kedalam *bytecode* yang selanjutnya *bytecode* tersebut akan diterjemahkan oleh *Virtual Machine (Java Virtual Machine)*.

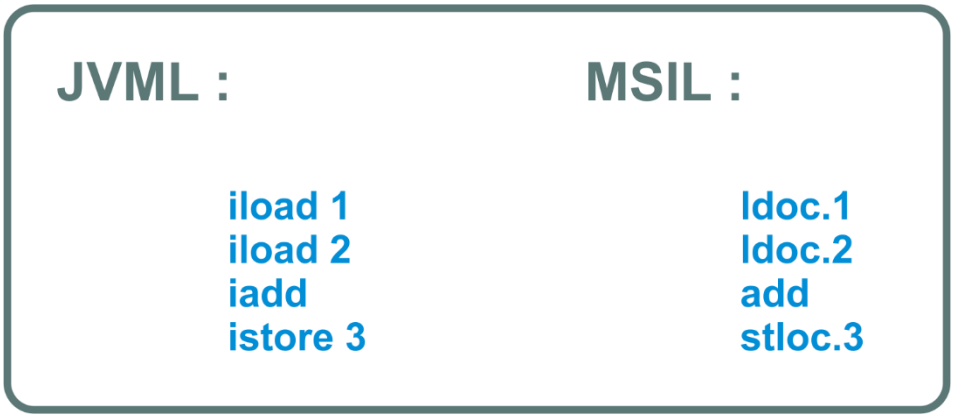
Keuntungannya adalah *platform-independent* yaitu *bytecode* yang dikompilasi disuatu sistem operasi atau mesin komputer bisa berjalan disistem operasi dan mesin komputer lainya menggunakan suatu *Cross-compiler*.

Untuk menghasilkan kecepatan dalam memproses *input* dan *output, Java Compiler* akan menggunakan *interpreter* yang akan menerjemahkan *bytecode* kedalam *Machine Language*.

JVM & CLR

Sebagai contoh pada bahasa *C#* dan *VB.Net* terdapat *CLR* (*Common Language Runtime*) yang secara konseptual dengan *JVM* keduanya adalah identik sebagai *Runtime Infrastructure*.

Tetapi *JVM* (*Java Virtual Machine*) hanya untuk bahasa *Java* dan representasi *bytecode* diterjemahkan menggunakan interpreter, sementara pada *CLR* (*Common Language Runtime*) representasi *bytecode* diterjemahkan menggunakan *compiler* dan terdapat dukungan untuk berbagai bahasa seperti *IL Assembly, IronPython, IronRuby, F#, C++/CLI, C#, VB.Net* dan sebagainya.



Gambar 12 Bytecode Representation

Pada gambar di atas terdapat representasi *bytecode* untuk *JVM* (sebelah kiri) dan *CLR* (sebalah kanan). Satu hal yang menjadi pembeda dari keduanya adalah pada *CLR* representasi *bytecode* akan di *compile* ulang menggunakan *JIT Compiler*, sementara pada *JVM bytecode* akan diterjemahkan menggunakan interpreter khusus.

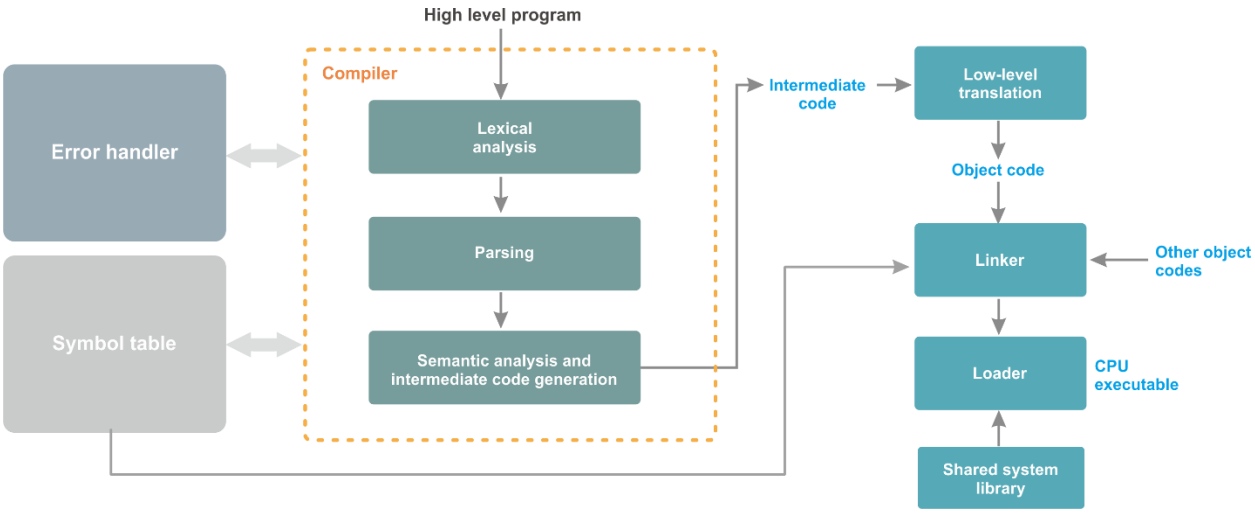
Two-stage Translation

Modern-day Compilers menggunakan *Two-stage Translation* untuk mengeksekusi *high-level language* agar bisa berjalan diberbagai arsitektur mesin komputer. Sehingga tantangan untuk menghadapi perbedaan bahasa *Assembly* bisa dihadapi, sebab setiap *Architectures* memiliki bahasa *Assembly* yang berbeda beda sebagai contoh pada *Architectures Intelx86* dan *ARM* keduanya memiliki *Assembly Language* yang berbeda.

Dengan begitu peluang besar untuk membuat kompiler yang bisa berjalan pada *Multiple Architectures* bisa diwujudkan.

Untuk diwujudkan sebuah *Intermediate Code* telah dikembangkan dalam dunia pemrograman, yaitu pada *stage* pertama kompiler menerjemahkan *high-level language* ke *Intermediate-level Code* dan *stage* kedua menerjemahkan *Intermediate-level Code* ke *Low-level Machine Code*.

Stage pertama penerjemahan terdiri dari *lexical analysis*, *parsing*, *semantic analysis* dan *code generation* dengan *target language* untuk memproduksi *intermediate language*.



Gambar 13 Two Stage Translation

Sebagai contoh kompiler yang digunakan untuk pemrograman C# menggunakan *schema Two Stage Translation, CIL Generator* akan menghasilkan *target language* berupa *IL Assembly*.

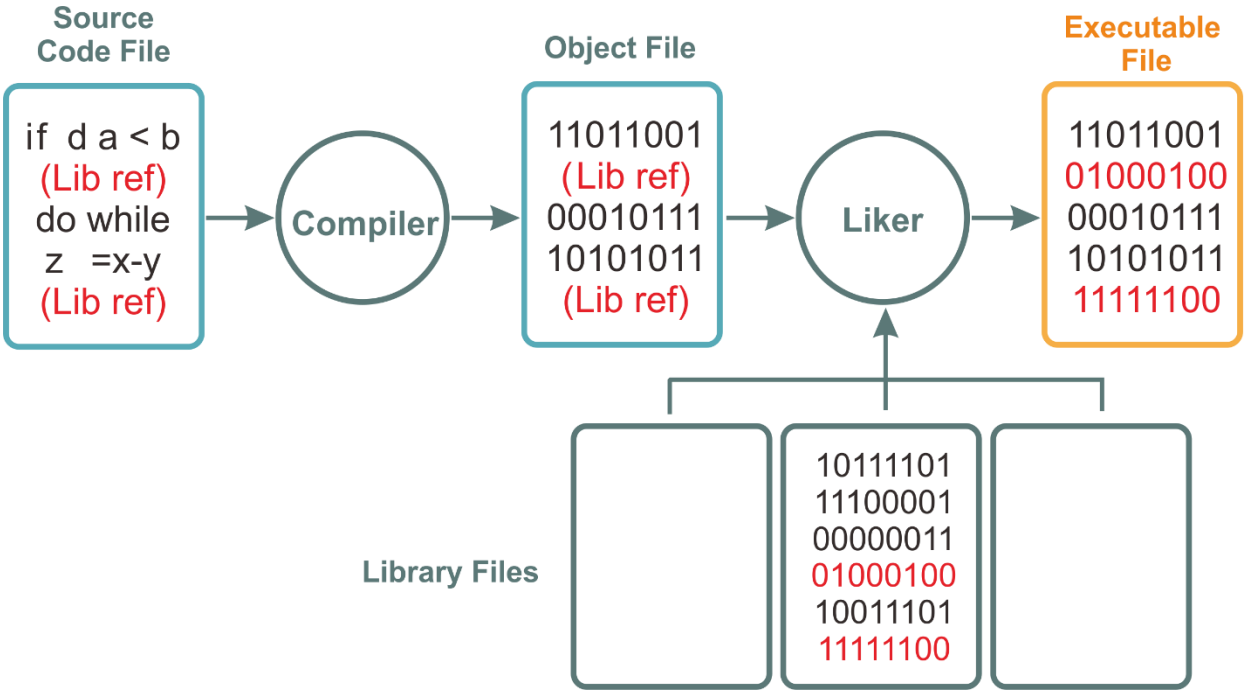
Stage selanjutnya adalah *IL Assembly* sebagai *intermediate language* diterjemahkan kedalam *Low-level Machine Code* oleh sebuah program bernama *IL Assembler*.

Object Code

Object Code adalah *machine language* yang telah dikompilasi atau sebuah *Object module* diproduksi oleh *assembler* di dalam proses *translation*.

Secara umum *object code* terdiri dari sekumpulan *statement* atau *instruction* bisa berupa *machine language* atau *intermediate language* tergantung tipe *compiler* dan *target language* yang diproduksi suatu *compiler*.

Sekumpulan *Object Code* disebut dengan *object files* yang selanjutnya akan dibentuk menjadi sebuah *executable file* atau *library file* oleh program yang disebut *Linker* [16].



Gambar 14 Contoh Representasi Object Code

Linker

Linker adalah program yang menggabungkan (*linking process*) sekumpulan *object files* yang telah dikompilasi oleh sebuah *assembler* untuk menjadi sebuah *executable*.

Linker akan mencari *library* yang digunakan di dalam program, selain itu *Object files* mengandung kombinasi *machine instructions*, *data*, dan informasi yang dibutuhkan untuk menempatkan instruksi di dalam *memory*.

Loader

Loader adalah sebuah program yang memuat sebuah *executable* ke dalam sebuah main *memory* [17].

Loader memuat *linked code* agar bisa dieksekusi di dalam *memory segment*, sebuah daerah *memory* yang diberikan untuk *executable code* agar bisa dieksekusi di dalam sistem operasi.

Chapter 2 ✓

Setup Learning Environment ✓

Sebelum memulai belajar kita harus mengenal lingkungan belajar yang akan digunakan, ada beberapa *software* yang harus kita fahami agar proses belajar kita maksimal dan pengembangan *application* yang ingin kita buat bisa optimal.

Subchapter 1 – Visual Studio Code ✓

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

— Martin Fowler

Subchapter* 1 – *Objectives

- Mempelajari Cara ***Install Programming Language***
- Mempelajari Cara ***Install Keybinding***
- Mempelajari Cara ***Install & Change Theme Editor***
- Mempelajari Cara ***Install Extension***
- Mempelajari Cara ***Install Fira Font***

Visual studio code adalah ***code editor*** yang dibangun menggunakan **node.js** di atas ***base*** **electron.js** agar bisa berjalan di dalam ***dekstop environment***.

Sederhananya, ***Electron.js*** adalah ***framework*** yang dapat membuat aplikasi *web* menjadi aplikasi ***dekstop*** agar menjadi ***cross-platform application*** yang berjalan di semua sistem operasi.

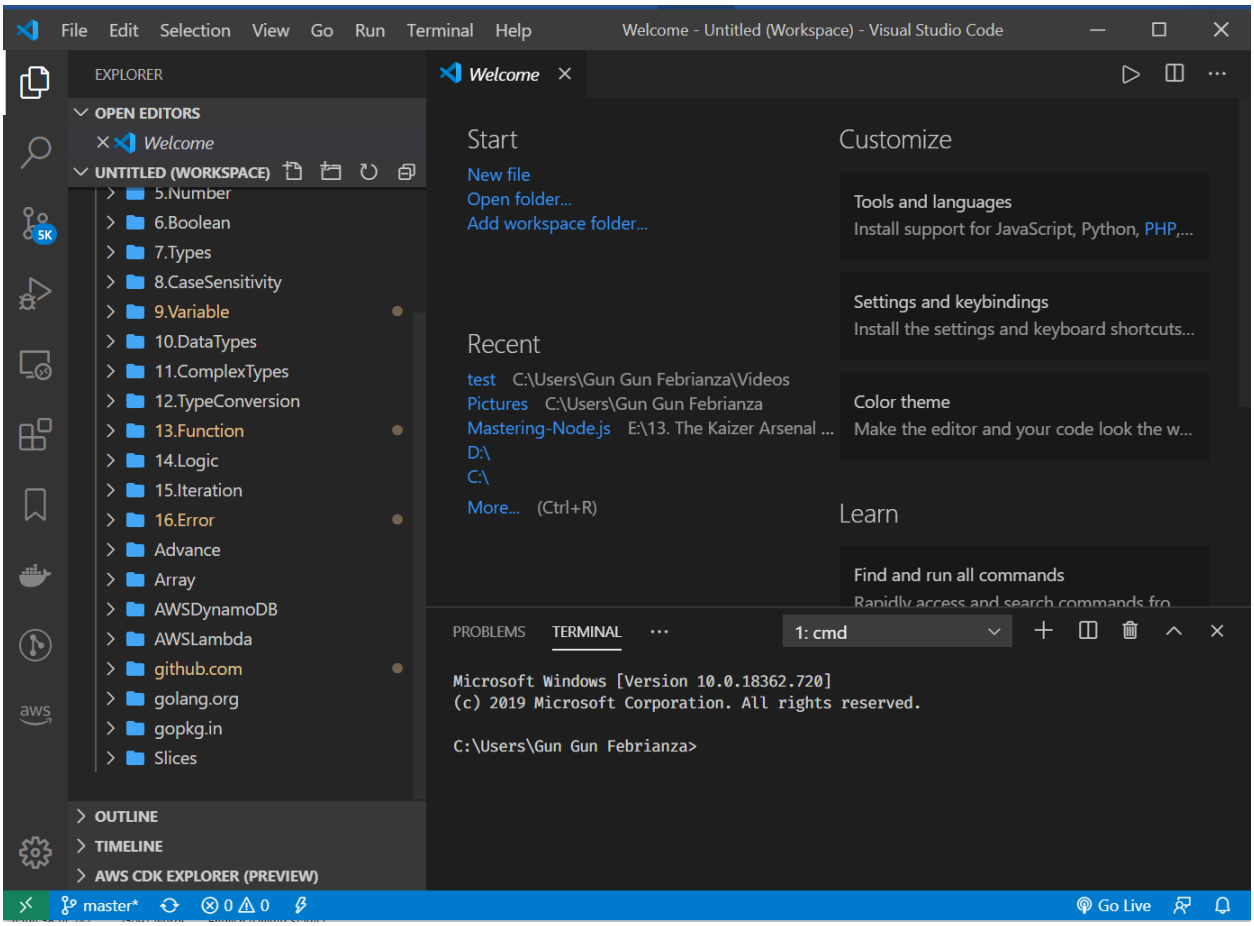
Link Project Visual Studio Code :

https://github.com/microsoft/vscode

Untuk mengetahui update fitur-fitur yang telah dikembangkan silahkan cek di :

https://code.visualstudio.com/updates/

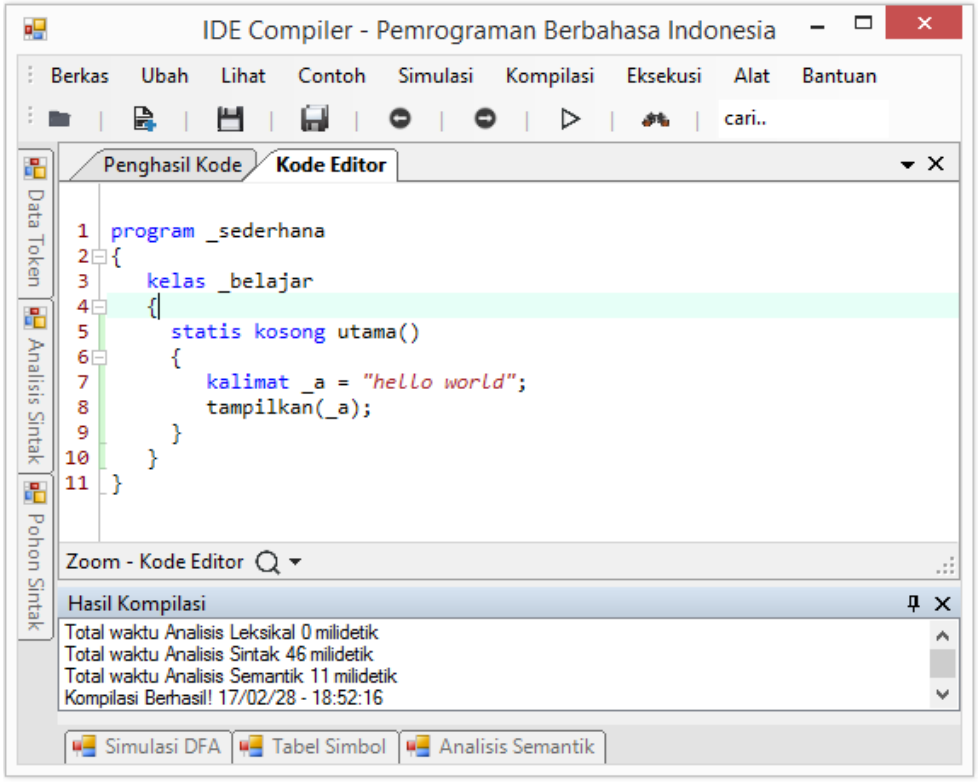
Di bawah ini adalah tampilan *user-interface* dari *viscode* :



Gambar 15 Visual Studio Code Interface

Saat duduk di bangku kulih, penulis pernah membuat IDE (*Integrated Development Environment*) yang di dalamnya terdapat *code editor*, skripsi penulis adalah pembangunan *compiler* dan pembuatan bahasa pemrograman berbahasa Indonesia.

Penulis masih ingat pembimbing memaksa untuk membangun *code editor* yang bisa mempermudah pengguna dalam menulis kode pemrograman berbahasa indonesia. Sungguh permintaan yang berat 😊 di bawah ini penampakanya :



Gambar 16 IDE untuk Pemograman berbahasa Indonesia

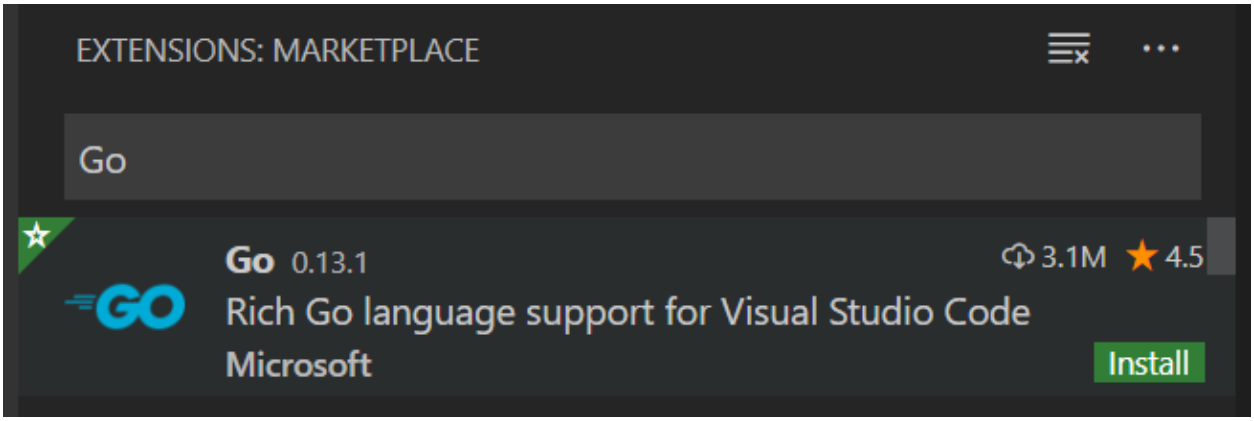
Jadi diri sini, dari pengalaman ini penulis begitu percaya diri membahas kajian seputar *code editor*.

Jika teman anda bertanya mengapa anda memilih suatu *code editor* tentu anda harus memiliki jawaban yang jelas dan menginspirasi. Di bawah ini adalah beberapa alasan mengapa kita memilih *code editor* :

1. Memiliki mekanisme untuk membuat performa dari *code editor* ringan.
2. Tersedia fitur ***Intellisense*** yang terdiri dari ***Syntax Highlighting*** & ***Autocomplete***.
3. Tersedia fitur untuk melakukan ***Debugging***.
4. Tersedia fitur untuk berinteraksi dengan ***Git***.
5. Tersedia fitur ***Add-ons*** untuk *code editor* yang dikembangkan oleh komunitas aktif dan pengembang *expert*.

1. *Install Programming Language Support*

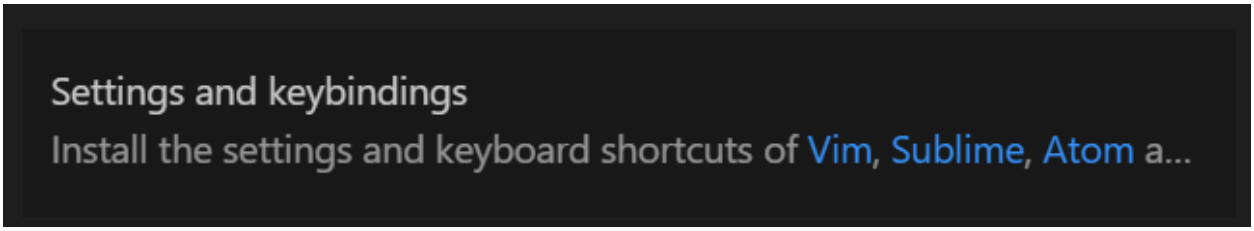
Tekan tombol **CTRL + SHIFT + X** untuk memasuki ***menu Extensions***, kemudian pada kolom pencarian ketik ***Go***, sampai muncul ***icon Go*** seperti pada gambar di bawah ini :



Gambar 17 Install Go Language Addons

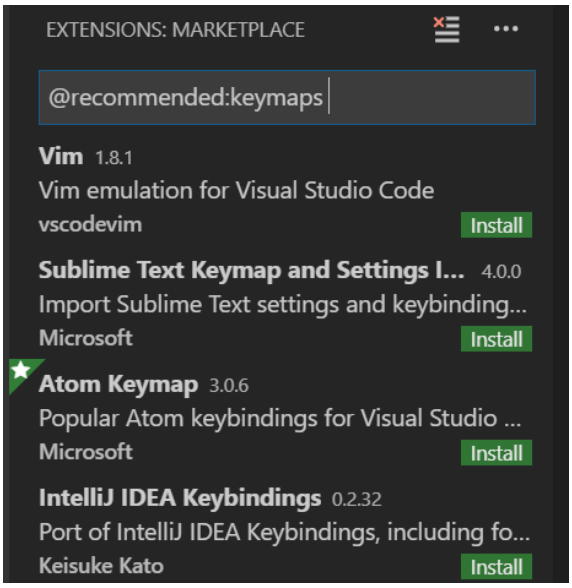
2. *Install Keybinding*

Keyboard Shortcut memiliki peran vital untuk penulisan kode, perubahan ***keyboard shortcut*** pada ***code editor*** baru tentu akan menyulitkan. Untuk mengatasi permasalahan ini pada menu ***keybindings*** kita bisa melakukan instalasi ***keymap extension***,



Gambar 18 Key Bindings

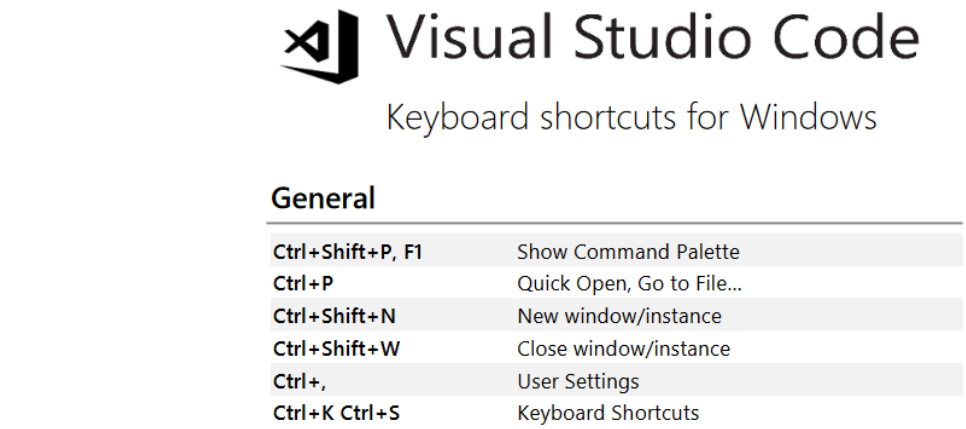
Terdapat beberapa ***keymaps*** yang bisa kita gunakan termasuk ***Atom Keymap***.



Gambar 19 Keymap Extension

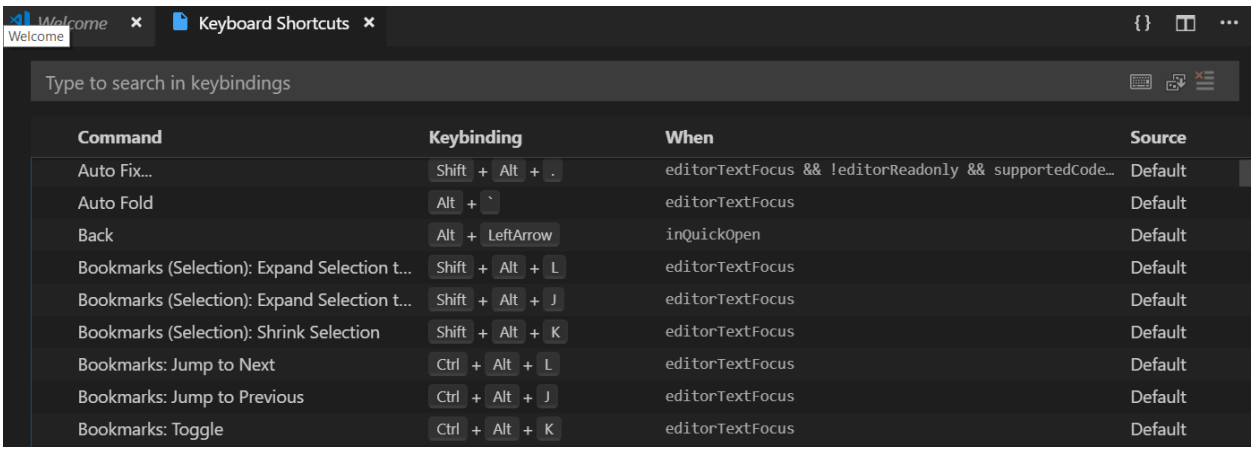
Saat ini penulis masih menggunakan ***Keyboard Shortcut Default*** bawaan dari *viscode*. Untuk ***Cheatsheet*** lengkapnya bisa di *print* dan tempel di tembok. Silahkan lihat disini :

https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf



Gambar 20 Visual Studio Code Keyboard Shortcut

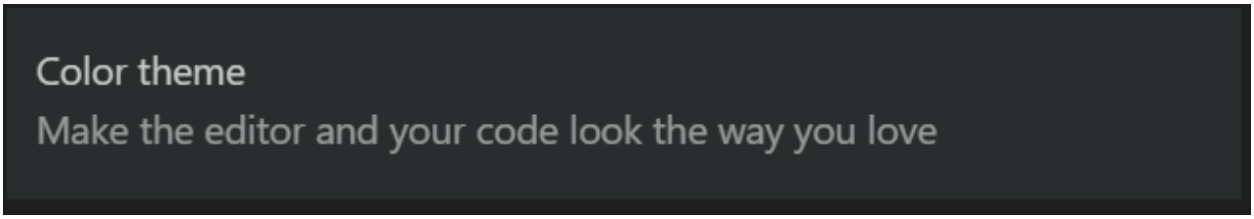
Jika anda ingin mengubah *keyboard shortcut* silahkan memilih menu ***File → Preferences → Keyboard Shortcuts*** atau tekan tombol ***CTRL+K*** kemudian ***CTRL+S***.



Gambar 21 Default Keyboard Shortcut

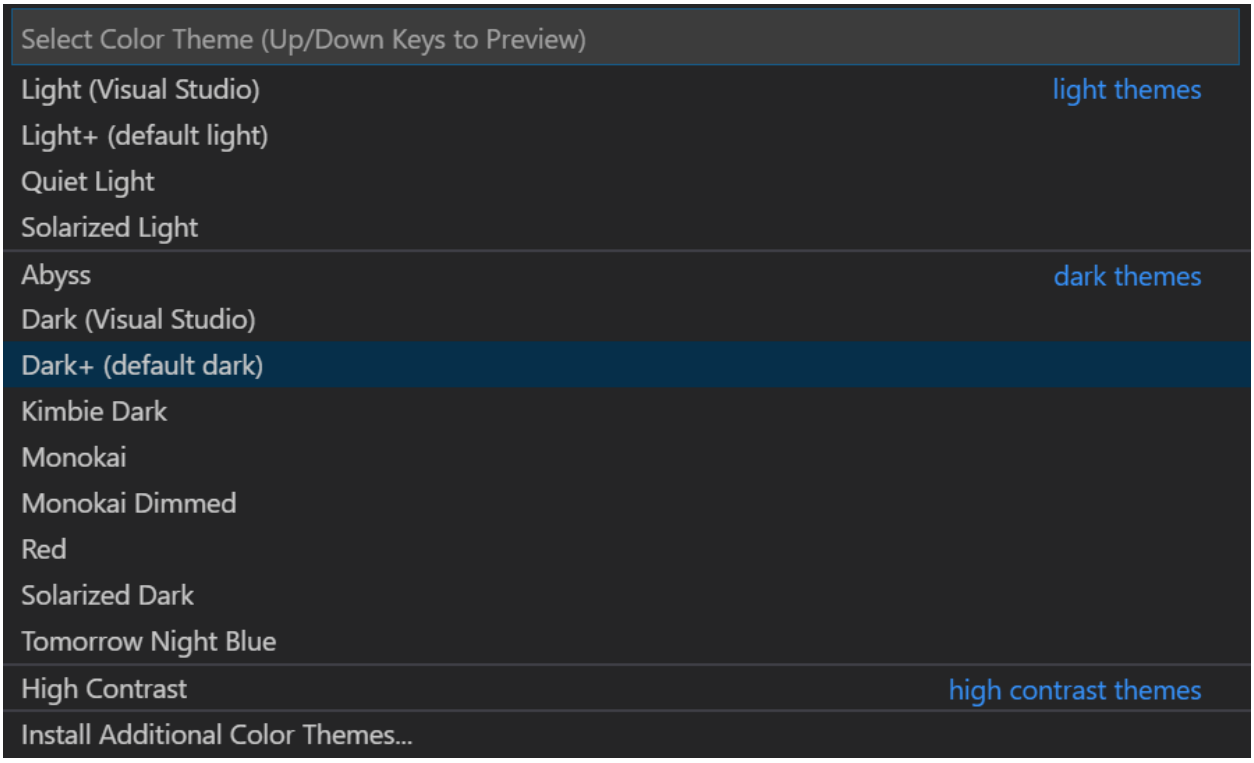
3. *Install & Change Theme Editor*

Pada ***Color Theme*** kita bisa memilih *theme editor* yang tersedia. Ada banyak pilihan dan kita juga bisa melakukan instalasi *Theme* yang telah disediakan komunitas, dibuat oleh *expert*.



Gambar 22 Color Theme Menu

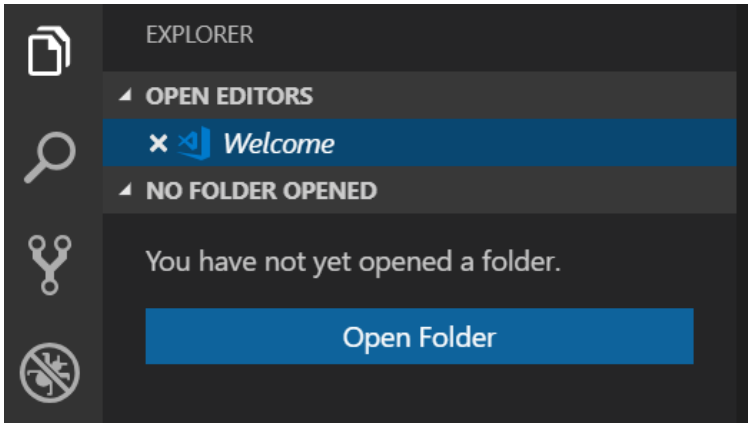
Jika anda kurang puas dengan *theme* yang tersedia anda bisa mencari *theme* lainya dengan memilih menu paling bawah **Install Additional Color Themes**.



Gambar 23 Install New Themes

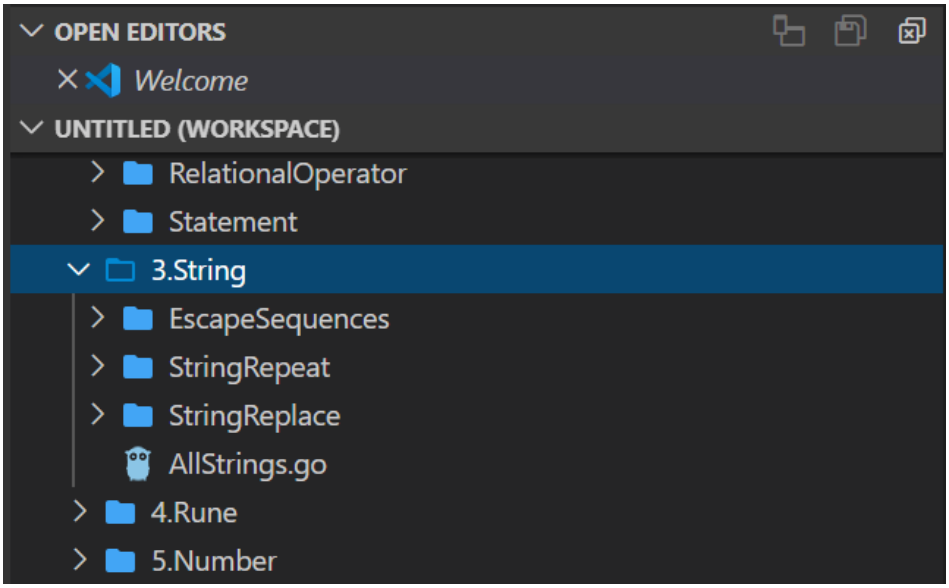
4. *The File Explorer*

Sekarang kita akan mempelajari ***file explorer*** yang disediakan oleh *viscode*. Klik **Open Folder** untuk membuka *folder* proyek yang pernah anda buat.



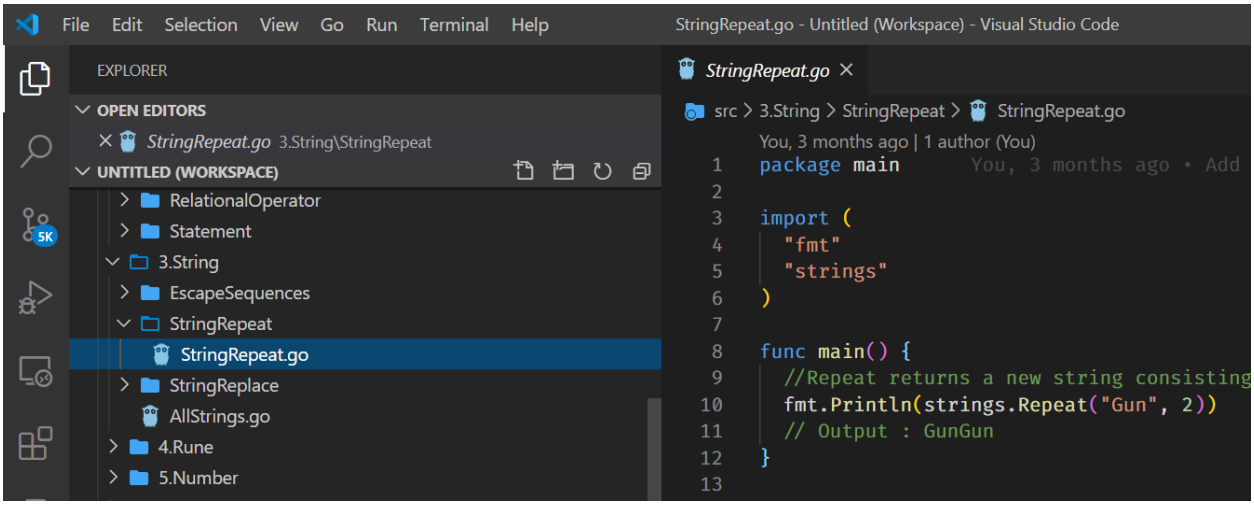
Gambar 24 File Explorer

Di bawah ini adalah daftar *folder* dan *file* yang telah penulis muat ke dalam *viscode explorer* :



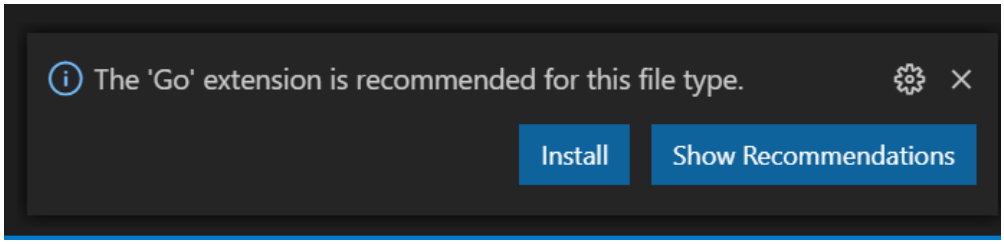
Gambar 25 Display Folder in File Explorer

Jika kita ingin membuka salah satu *file* ke dalam *code editor*, klik *file* tersebut :



Gambar 26 Display File in The Code Editor

Jika muncul pesan di bawah ini klik ***install*** :

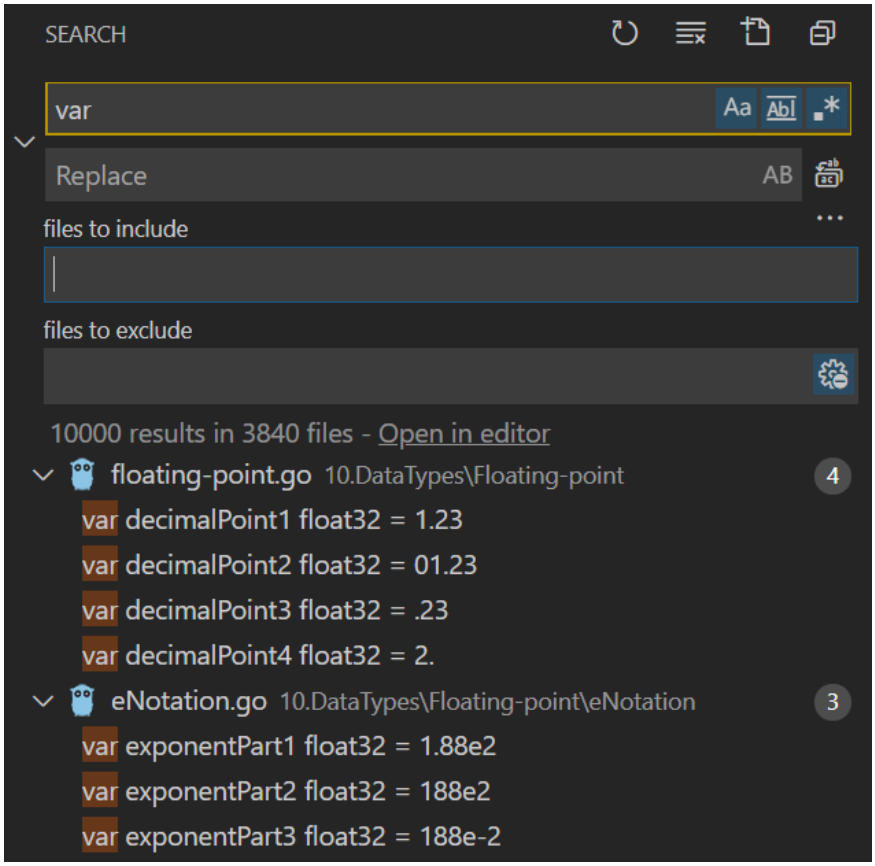


Gambar 27 Extension Recommendation

5. *Search Feature*

Viscode memiliki fitur yang bisa kita gunakan untuk mencari suatu ***string*** dalam ***files*** proyek yang kita muat. Klik gambar kaca pembesar, kemudian masukan ***string*** yang ingin kita cari.

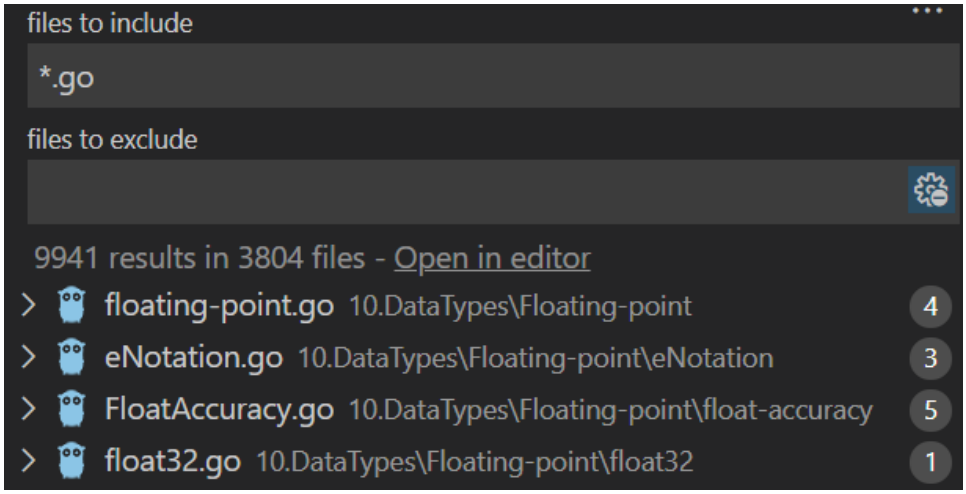
Pada kasus ini penulis memasukan ***keyword*** **var** :



Gambar 28 Search String

Pada gambar di atas kita bisa melihat ada banyak *file* yang memiliki ***string*** **var**, kita bisa melakukan operasi ***replace*** pada seluruh *file* atau hanya pada satu *file* saja. Untuk membuka *file* yang memiliki ***string*** **var** anda tinggal klik daftar *file* yang muncul dalam kolom pencarian.

Kita juga bisa mengatur *file* dengan ekstensi apa saja yang akan kita cari dan membatasi *file* dan *folder* mana saja yang tidak ingin kita cari.

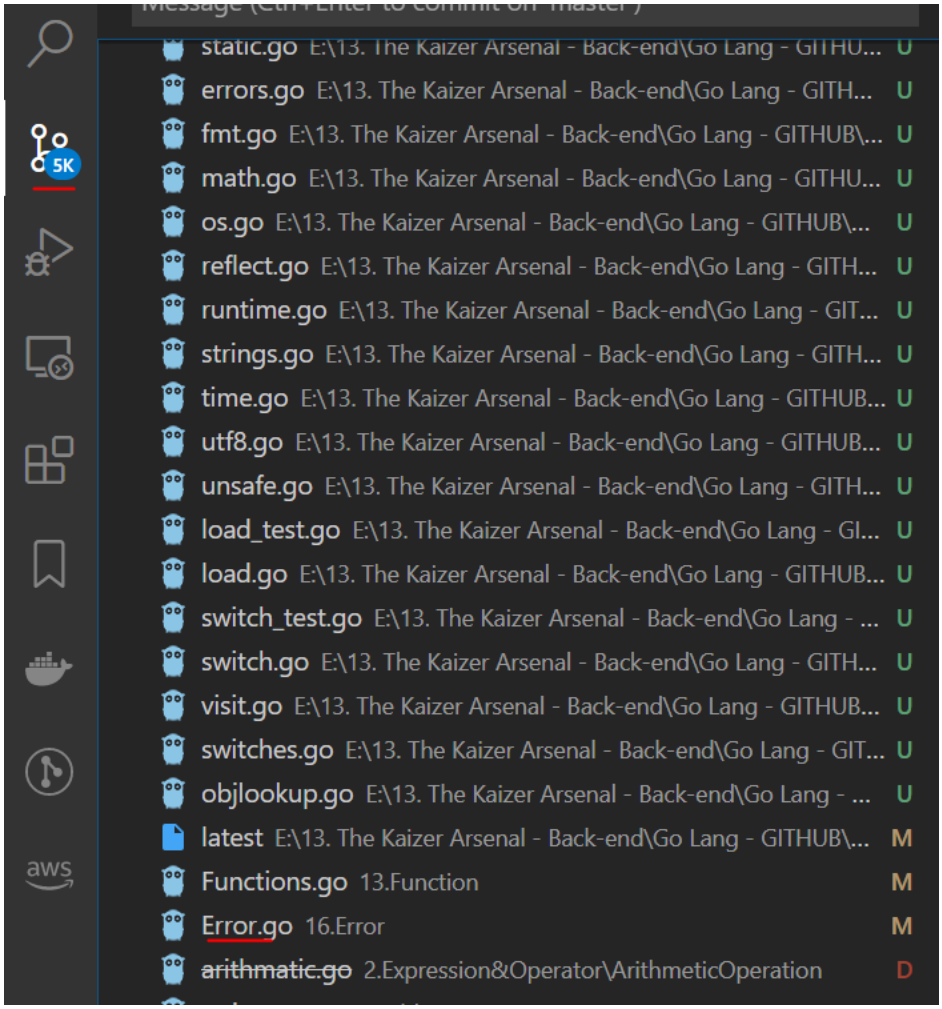


Gambar 29 Include & Exclude on Search String

Untuk melakukan pencarian dengan ***filter*** klik *icon* yang diberi garis merah, pada gambar di atas kita membatasi pencarian *string* hanya untuk *file* .go saja

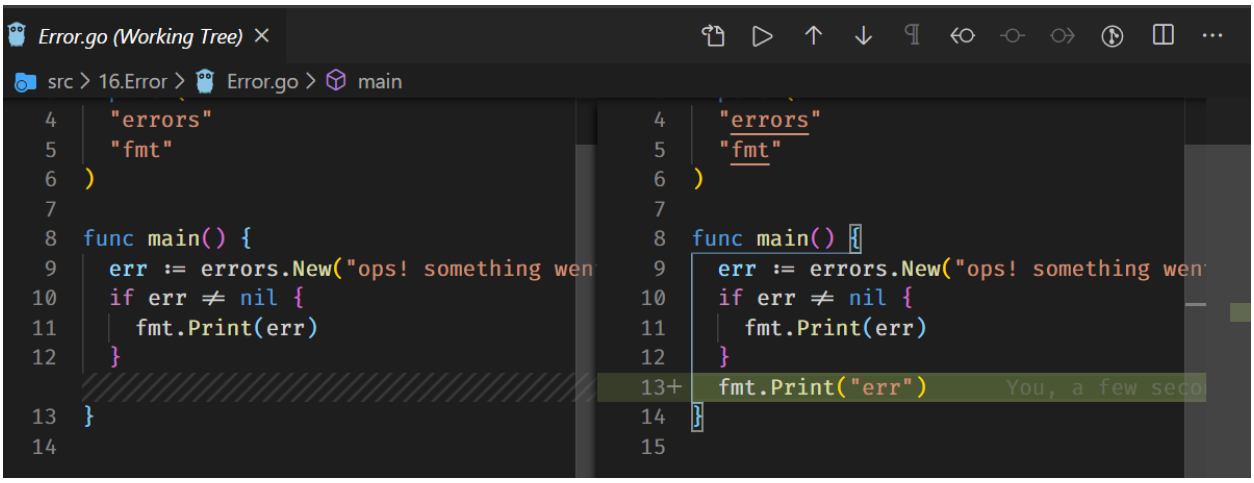
6. *Source Control*

Viscode mendukung git sehingga kita bisa melacak setiap perubahan yang terjadi. Untuk melihatnya klik ikon *working tree*, pada kasus ini penulis telah mengubah salah satu *file* di dalam proyek :



Gambar 30 Source Control

Jika kita klik *file* tersebut maka kita bisa melihat perubahan yang telah kita lakukan sebelum dan sesudahnya :



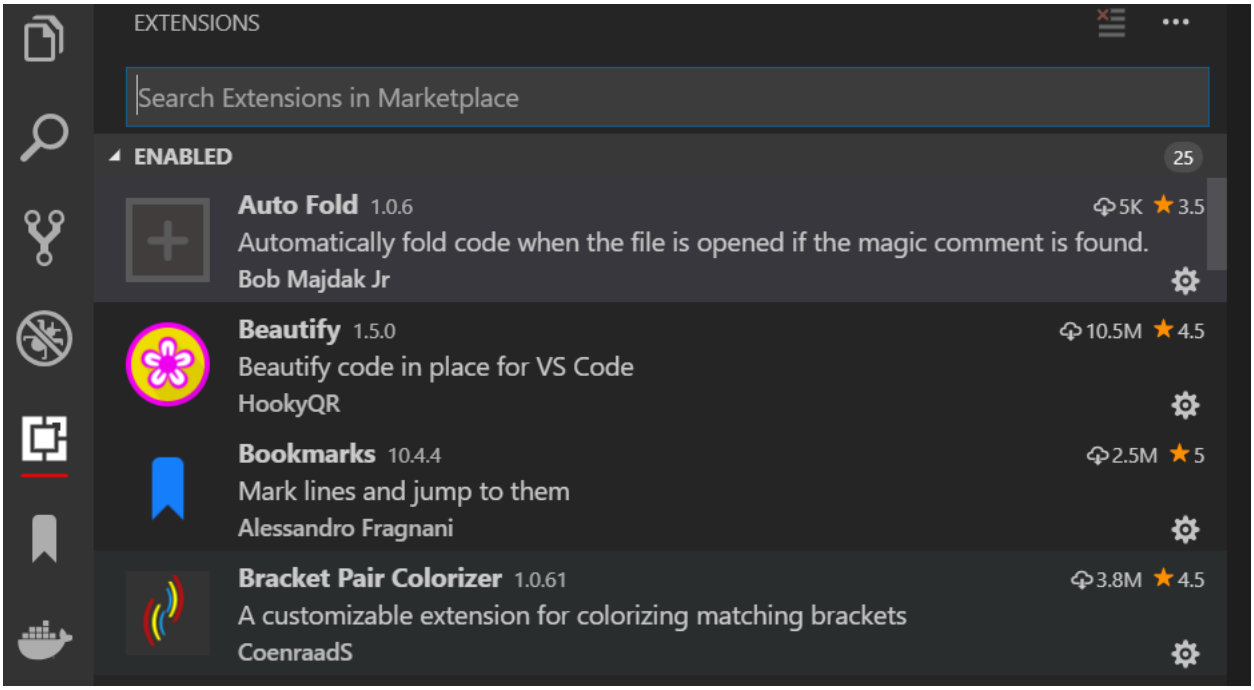
Gambar 31 Diff Mode

7. *Debugger*

Viscode menyediakan *debugger* untuk berbagai bahasa pemrograman, kita akan mempelajari cara melakukan *debugging* dalam *Go* pada *chapter* selanjutnya.

8. *Extension*

Extension adalah tempat untuk menambahkan berbagai fitur yang dapat mempermudah dan mempercepat kita menulis kode. Produktivitas kita menjadi lebih masimal dengan *tools* yang telah disediakan dan dikembangkan oleh expert di komunitas *viscode*.



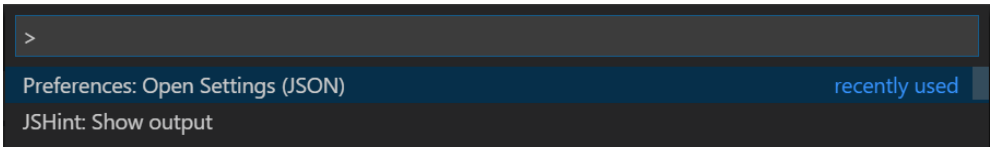
Gambar 32 Extension

Pada gambar di atas penulis sudah melakukan instalasi beberapa *extension* di antaranya adalah :

Auto Fold

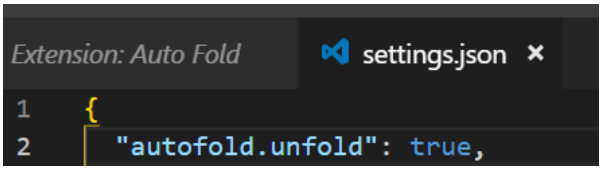
Ekstensi ini berguna agar *viscode* langsung membuka seluruh baris kode yang tertutup oleh *bracket* {...}, sehingga anda tidak perlu membukanya secara manual.

Install ekstensi ini kemudian tekan tombol **F1** pada *viscode* hingga muncul kolom perintah kemudian Ketik **Open Settings** seperti pada gambar di bawah ini :



Gambar 33 Viscode Settings

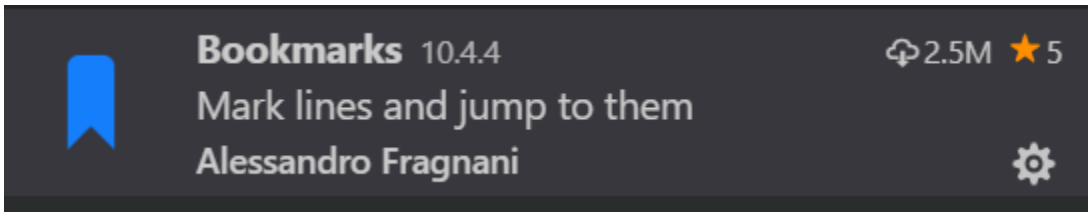
Pada **setting.json** masukan kode **"autofold.unfold": true**, agar ekstensi *auto fold* yang kita gunakan bekerja. Perhatikan gambar di bawah ini :



Gambar 34 Autofold Configuration

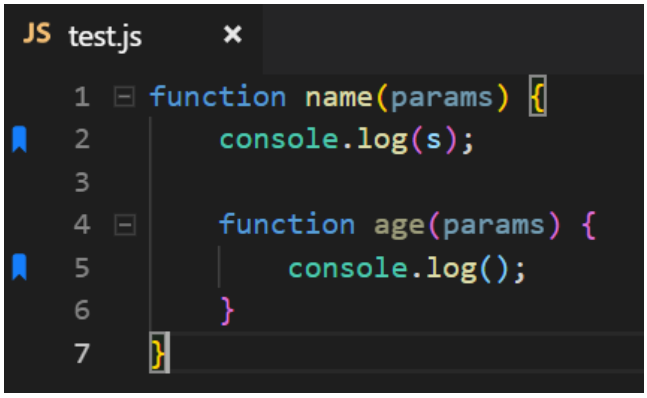
Bookmarks

Ekstensi ini sangat membantu jika jumlah kode yang telah kita buat sudah sangat panjang, baik itu ratusan ataupun ribuan baris. ***Scrolling code*** menjadi salah satu hal yang memakan waktu, untuk itu ***bookmarks code*** sangat membantu agar kita bisa meloncat ke alamat kode yang kita inginkan.



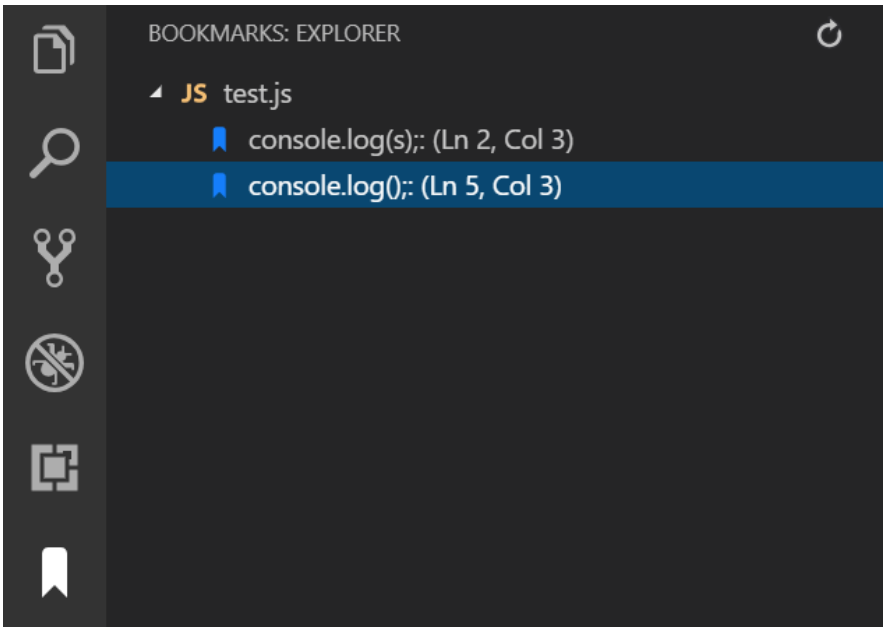
Gambar 35 Bookmarks Extension

Kita bisa melakukan *bookmark code* dengan cara menekan tombol **CTRL+ALT+K**, perhatikan gambar di bawah ini :



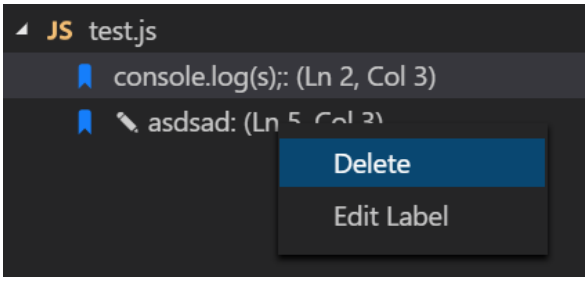
Gambar 2.2.29 *Bookmarked Codes*

Pada gambar di atas kita bisa melihat terdapat dua baris kode yang telah kita *bookmark*. Baris kode kedua dan baris kode kelima, jika kita ingin loncat kesetiap *bookmark* maka tekan tombol **CTRL+ALT+L**. Silahkan fahami terlebih dahulu.



Gambar 2.2.30 *Bookmark Explorer*

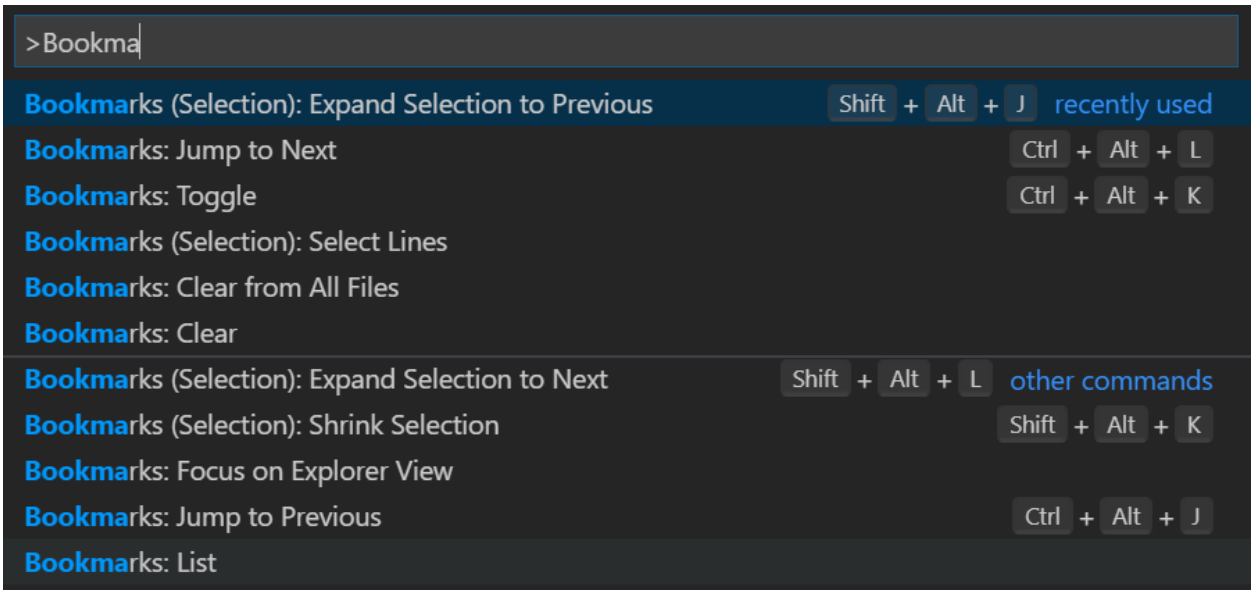
Pada *bookmark explorer* kita bisa melihat *file* apa saja dan alamat kode mana saja yang telah kita *bookmark*. Untuk mempermudah memahami kode yang telah kita *bookmark* kitapun bisa memberikan *label* pada kode yang telah kita *bookmark*.



Gambar 2.2.31 *Bookmark – Delete & Edit Label*

Kita juga bisa menghapus kode yang telah kita *bookmark*, atau dengan cara menekan kembali **CTRL+ALT+K** pada baris kode yang telah kita berikan *bookmark*.

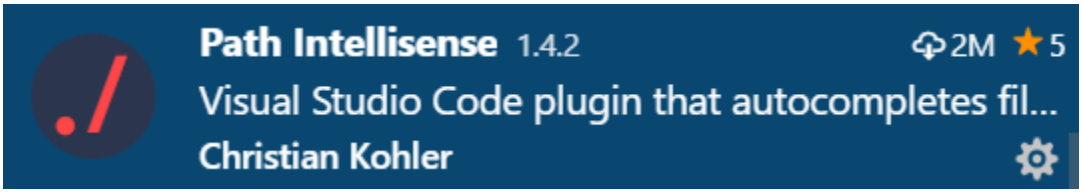
Untuk mengetahui ada fitur *bookmark* apa saja, tekan tombol **F1** kemudian ketikan ***bookmarks*** seperti pada gambar di bawah ini :



Gambar 36 Bookmark – List Commands

Path Intellisense

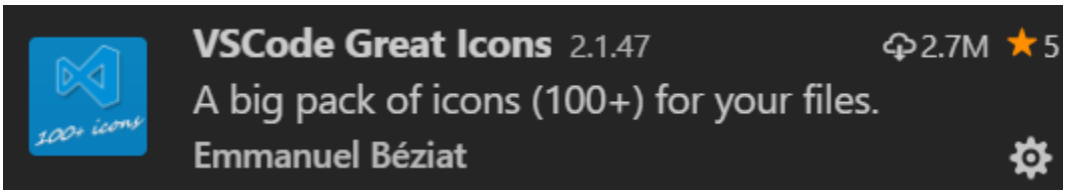
Ekstensi ini membantu kita saat kita berhadapan dengan *path*, ekstensi ini akan memberikan fitur *autocomplete* ketika ingin mengeksplorasi suatu *path* di dalam *code editor*.



Gambar 37 Path Intellisense Extension

VSCode Great Icons

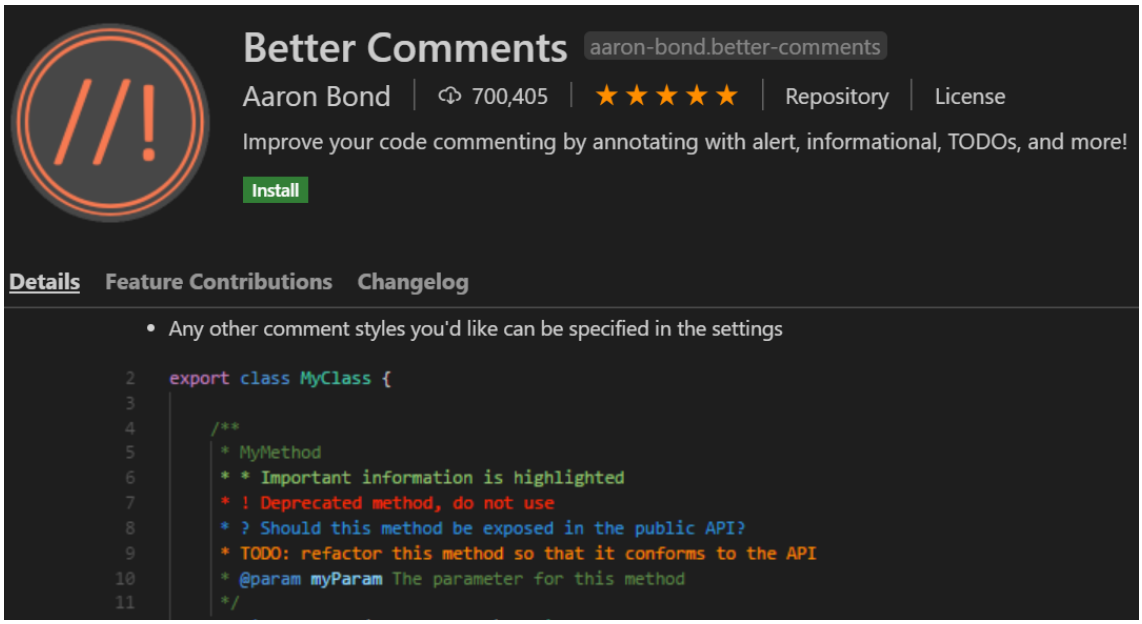
Ekstensi ini sangat membantu untuk menampilkan *icon* untuk berbagai macam ekstensi agar menjadi lebih menarik lagi dan klasifikasi *file* menjadi mudah untuk dikenal.



Gambar 38 VSCode Great Icons

Better Comment

Ekstensi ***better comment*** membantu kita membedakan jenis komentar dengan warna menarik yang mudah diingat.



Gambar 39 Better Comment Extension

Ada banyak *Extension* yang bisa anda gunakan, silahkan berdiskusi, bertanya di komunitas dan forum online jika anda ingin mendapatkan insight lebih banyak lagi.

9. *The Terminal*

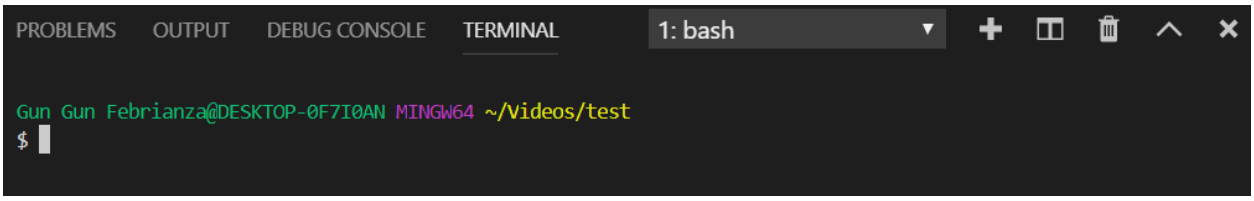
Fitur terminal dalam *code editor* membantu kita untuk bisa mengeksekusi berbagai macam perintah. Seperti mengeksekusi *file go*, *install package* atau mengeksekusi program lainya di dalam direktori proyek yang sedang anda bangun.

Pada ***Settings.json*** tambahkan kode di bawah ini agar terminal kita terintegrasi dengan ***bash*** :

Gambar 40 Shell Integration

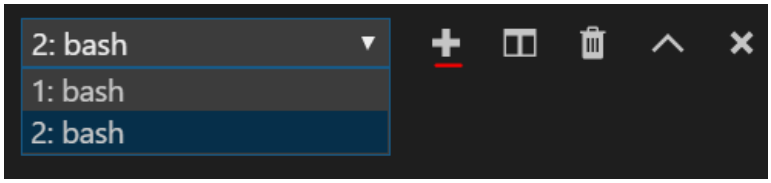
Notes
Pastikan anda sudah melakukan instalasi Git !

Untuk menampilkan terminal dalam *viscode* tekan tombol **CTRL+`**, maka *terminal* akan tampil seperti pada gambar di bawah ini :



Gambar 41 Terminal Visual Studio Code

Menambah *Terminal* Baru



Gambar 42 Add New Terminal

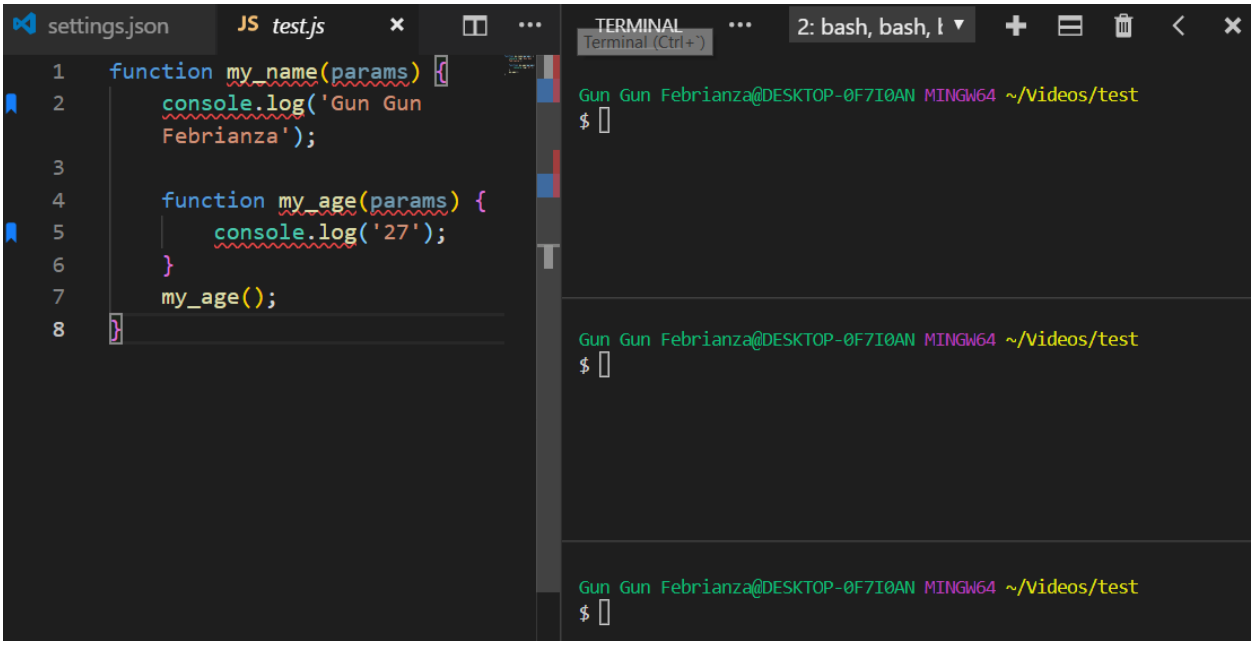
Tekan tombol tambah jika kita ingin menambahkan *terminal* yang baru, anda bisa melakukan *switch* pada *terminal* yang pertama dan kedua.

Melakukan *Split Terminal*

Gambar 43 Add New Terminal

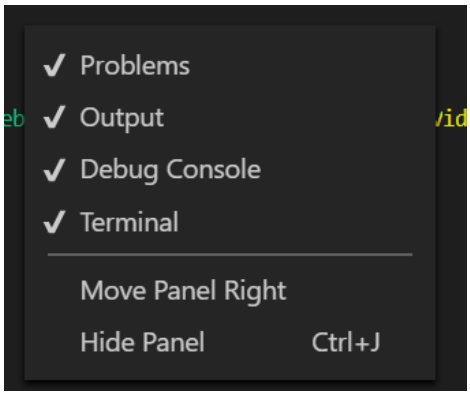
Bagi yang memiliki layar dengan lebar yang sangat panjang fitur ini sangat membantu. Sehingga kita tidak perlu menggunakan ***drop down*** untuk mengganti *channel terminal*.

Mengubah Posisi *Terminal*



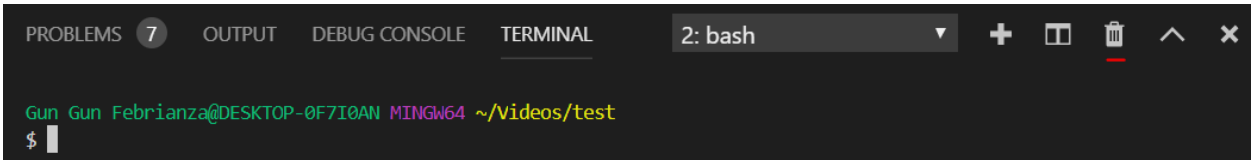
Gambar 44 Change Terminal Position

Kita bisa mengubah posisi terminal menjadi di sebelah kanan dengan cara melakukan klik kanan pada *terminal*, kemudian pilih menu *Move Panel Right* seperti pada gambar di bawah ini :



Gambar 45 Move Panel

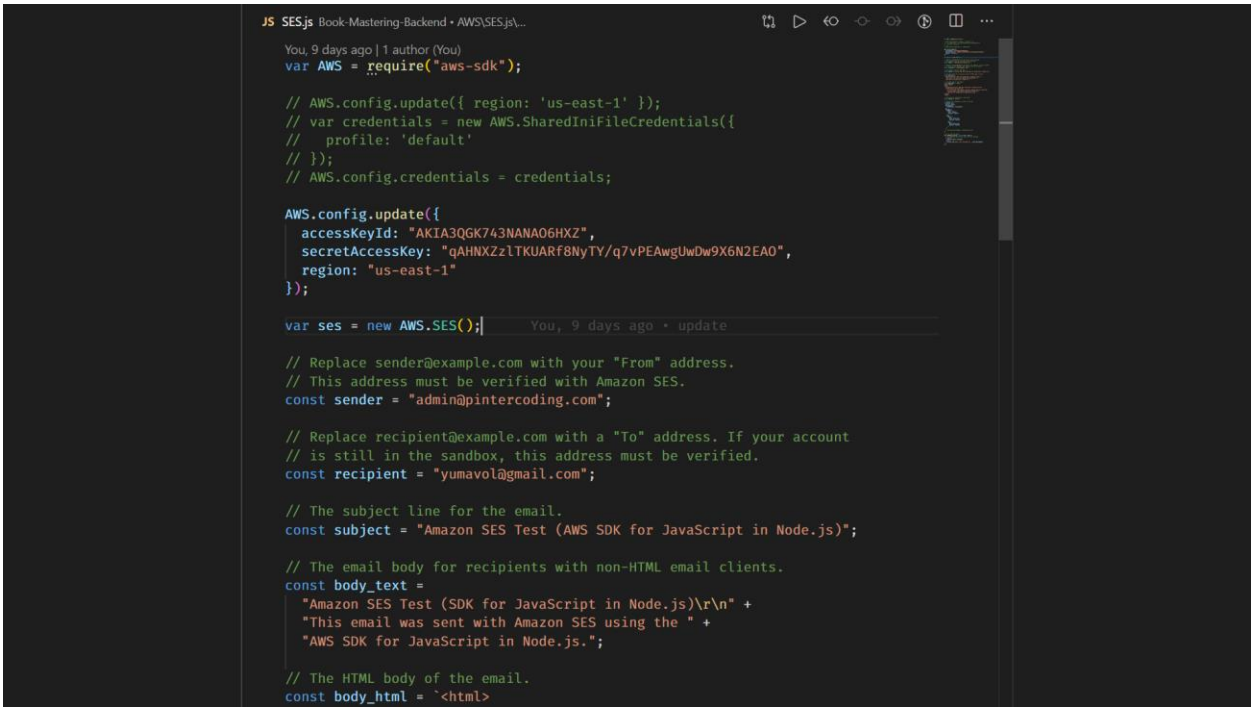
Menghapus *Terminal*



Gambar 46 Delete Terminal

10. *Zen Mode*

Agar kita dapat fokus pada kode yang kita tulis, kita bisa memasuki ***zen mode*** dengan menekan tombol **CTRL+K** sekali kemudian klik tombol **Z** :

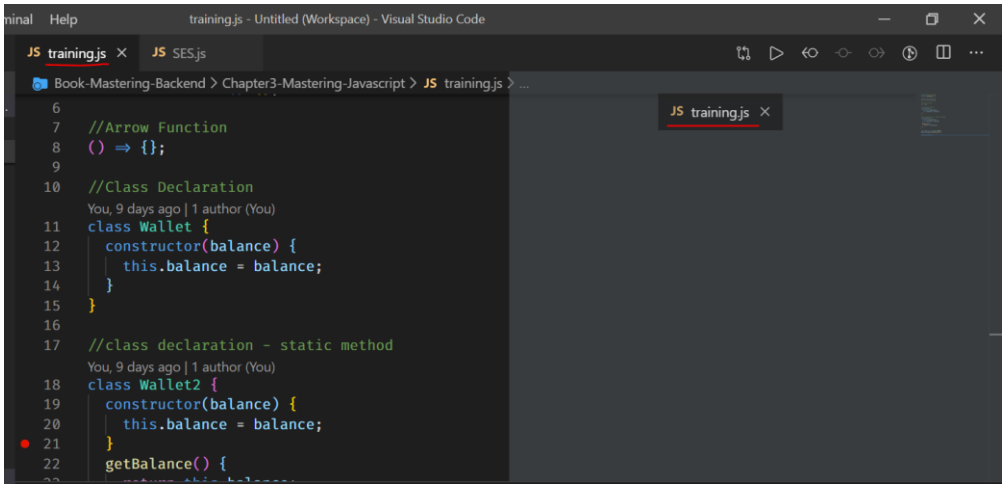


Gambar 47 Zen Mode

Untuk keluar dari ***zen mode*** tekan tombol **CTRL+K** sekali kemudian klik lagi tombol **Z**.

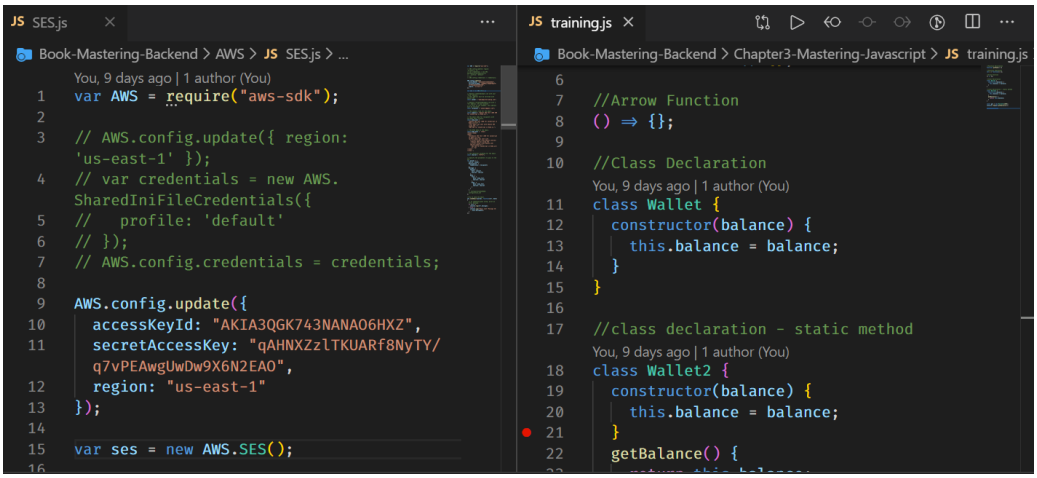
11. *Display Multiple File*

Terkadang ada saatnya kita ingin menampilkan dua kode sekaligus dalam satu ***code editor***, untuk melakukannya *drag* salah satu *file* ke arah kanan sampai muncul ***gradient*** warna yang berbeda membelah ***code editor*** :



Gambar 48 Display Multiple File

Jika sudah kita dapat melihat ***code editor*** kedua untuk mempermudah kita memahami kode, jika layar anda cukup lebar anda dapat menampilkan ***code editor*** ketiga dan seterusnya :



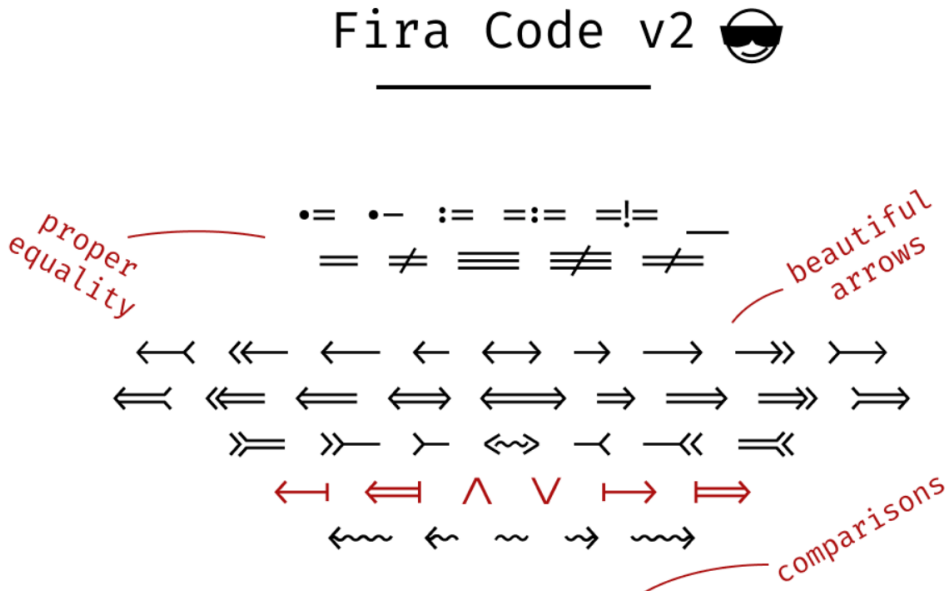
Gambar 49 Display Double File

12. *Font Ligature*

Terdapat font yang bagus untuk *code editor* yaitu ***Fira Code***, bisa anda cek disini :

https://github.com/tonsky/FiraCode

Jika kita menggunakan ***font* Fira code** kita akan memiliki efek ***symbol*** yang lebih mudah difahami untuk menulis kode :

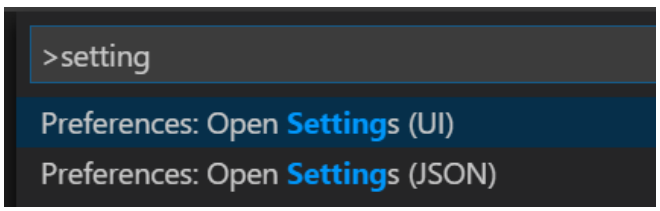


Gambar 50 Font Fira Code

Untuk cara instalasi Font dapat dibaca disini :

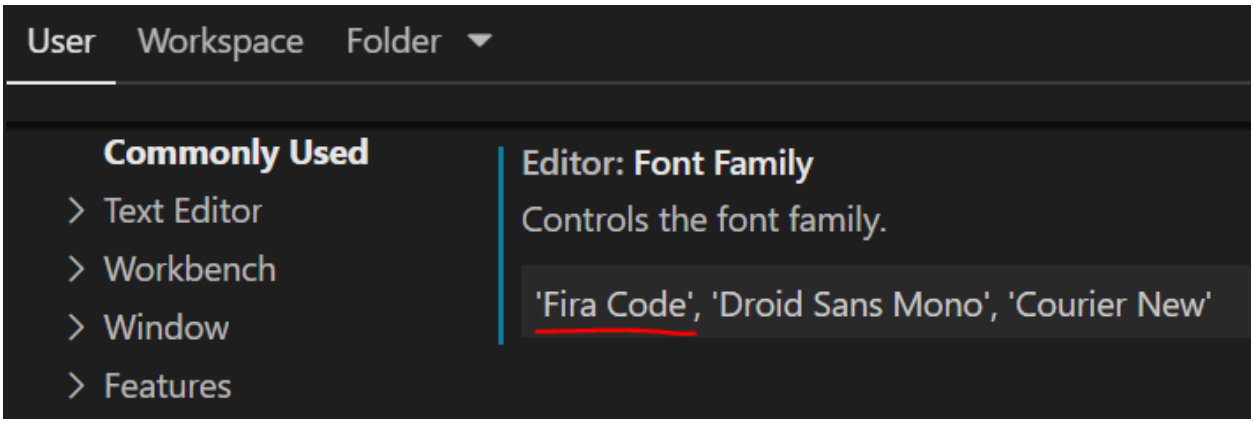
https://github.com/tonsky/FiraCode/wiki/Installing

Setelah anda melakukan instalasi font selanjutnya kita harus melakukan konfigurasi, tekan tombol **F1** kemudian ketik **settings**. Pilih **Open Settings (UI)** :



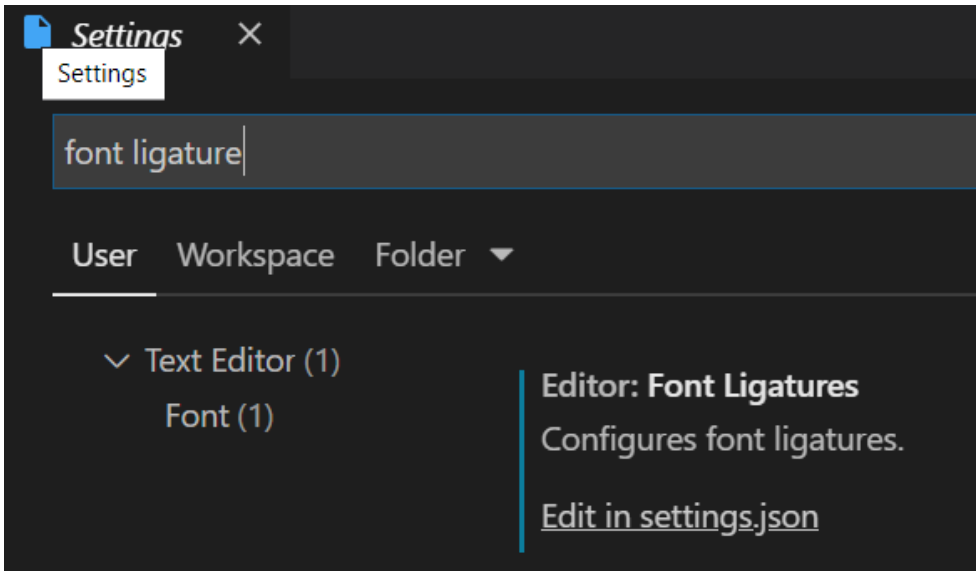
Gambar 51 Open Settings

Pada menu **Font Family** tambahkan **'Fira Code'** seperti gambar di bawah ini :



Gambar 52 Add Fira Code

Selanjutnya pada kolom pencarian ketik **font ligature**, klik **Edit in setting.json** :

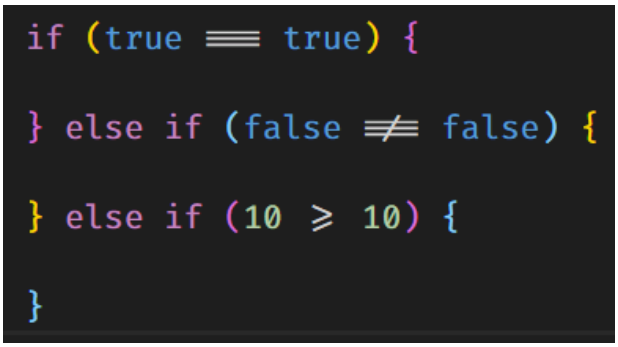


Gambar 53 Configure Font Ligature

Tambahkan pengaturan di bawah ini :

```
"editor.fontFamily": "'Fira Code', 'Droid Sans Mono', 'Courier New'",
"editor.fontLigatures": true,
```

Jika kita menulis kode seperti di bawah ini maka anda dapat melihat efeknya pada operator di dalam **logic if** dan **else if** :



Gambar 54 Sample

Subchapter 2 – Go Lang ✓

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

— Martin Fowler

Subchapter 2 – Objectives

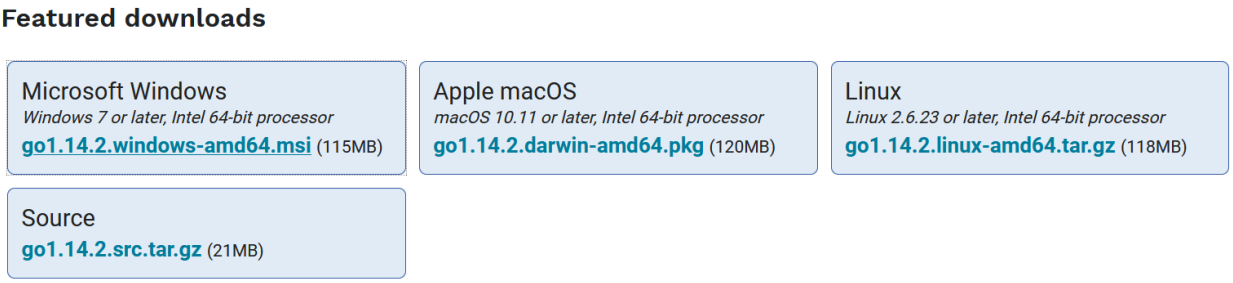
- Mempelajari Cara ***Install Go Lang***
- Mempelajari Cara ***Install Keybinding***
- Mempelajari Cara *Install & Change Theme Editor*

Go Lang Installation

Saat buku ini ditulis, versi *stable Golang* yang digunakan adalah *Golang* versi 1.14.2, untuk mendapatkanya kunjungi halaman *download* dari situs resmi *Golang* :

https://golang.org/dl/

Terdapat pilihan sistem operasi untuk instalasi go lang, baik itu sistem operasi *linux, mac* dan *windows* :



Gambar 55 Download Golang

Check Golang Version

Untuk memastikan ***Golang*** sudah terpasang dalam sistem operasi anda, buka ***terminal*** kemudian ketik :

```
 $ go version

go version go1.14.2 windows/amd64
```

Chapter 3 ✓

Mastering Go Lang ✓

Subchapter 1 – *Introduction to Go Lang* ✓

Subchapter* 1 – *Objectives

- Mengenal *Go Language*
- Mengenal Apa itu *Static Linking?*
- Mengenal Apa itu *Statically Typed & Type-safe memory?*
- Mengenal Apa itu *Garbage Collection?*
- Mempelajari Apa itu *Multicore Programming?*

Go atau *Golang* adalah bahasa pemrograman ***open source*** yang dibuat oleh **Robert Griesemer**, **Rob Pike** dan **Ken Thompson** tahun 2007 di *Google*. Robert Griesemer adalah pegawai *Google* yang memiliki pemahaman sangat dalam tentang ***Code Generation*** untuk ***Google V8 Javascript Engine***. Ken Thompson adalah seorang pujangga yang menciptakan sistem operasi UNIX.

Rob Pike bekerja sama dengan Ken Thompson untuk membangun skema *encoding UTF-8*. Bahasa pemograman *Go* didesain oleh para *pioneer* & *innovator* terbaik di dunia *computer science*.

Semua ini berawal pada tahun 2007, saat perangkat lunak *Search Engine* yang dimiliki ***Google*** bermasalah. Terdapat jutaan baris kode yang harus dipelihara, setiap kali mereka menambahkan fitur baru dan menguji fitur tersebut proses kompilasi harus dilakukan.

Proses kompilasi bisa memakan waktu berjam-jam, sangat buruk untuk produktivitas *developer*.

Alasan inilah yang menginspirasi mereka untuk membuat bahasa pemrograman yang bisa ditulis dengan cepat, dikompilasi dengan cepat dan dieksekusi dengan cepat. *Go lang* adalah jawaban atas semua permasalahan yang mereka hadapi.

> "A language that was fast to write code for and produced programs that were fast to compile and run."

Pada tahun 2009, bahasa pemrograman *Go* dibuat *open source* agar bisa dikembangkan oleh *developer* dari seluruh dunia.

1. *Go is Compiled Language*

Go adalah bahasa pemograman ***high-level*** yang perlu dikompilasi (*compilation*) terlebih dahulu agar dapat dimengerti oleh mesin.

Go adalah ***Compiled Language***, artinya *Go* adalah bahasa pemograman yang menggunakan ***Compiler*** untuk melakukan ***Compilation***. Program yang dibuat menggunakan *go*, kode sumbernya (*source code*) harus diterjemahkan terlebih dahulu kedalam *format executable*. *Go* akan memproduksi *file executable* yang dapat berjalan pada sistem operasi *windows, linux* dan *mac*.

Static Linking

Compilation adalah sebuah proses untuk menerjemahkan kode sumber yang bersifat ***human-readable code*** ke dalam kode biner, serangkaian instruksi yang difahami oleh komputer.

Jika anda lupa atau belum membacanya anda bisa membaca ***chapter*** 1 pertama tentang *compiler*.

Proses ***compilation*** pada bahasa pemrograman ***Go*** akan memproduksi kode biner (***binary code***) dalam format ***executable***. ***File Executable*** bersifat ***self-contained*** dapat langsung dieksekusi tanpa membutuhkan lagi *dependencies* pada ***library*** tertentu.

Go menggunakan ***Static Linking*** sehingga ***binary files*** yang diproduksi dapat di transfer ke sistem operasi lainya yang sama dengan mudah. Ketika program yang dibuat dengan ***Go*** telah dikompilasi, *developer* tidak perlu lagi memikirkan permasalahan tentang *libraries* dan *dependencies*.

Saat kompilasi terjadi *compiler* dapat memeriksa *error*, melakukan *optimization* dan memproduksi kode biner sesuai dengan target *platform* yang dituju. Kita dapat

memproduksi *executable* agar dapat berjalan di sistem operasi *Windows, Linux* dan *MacOS*. Bahasa pemograman *Go* juga mendukung jargon :

"write your code once and run it anywhere"

Go Compiler

Sebagian besar *compiler* untuk bahasa pemograman ***Go***, ditulis dengan bahasa pemograman ***Go*** sendiri. Nama program *compiler* untuk bahasa pemograman *Go* adalah ***gc*** atau ***Go Compiler***.

Program tersebut sudah disisipkan kedalam *Go SDK* (*Software Development Kit*).

2. Go is Safe Language

Ada tiga hal dari hasil pengamatan penulis kenapa bahasa pemrograman *Go* termasuk ke dalam *safe language* :

Statically Typed & Type-safe Memory

Go adalah bahasa pemograman yang mendukung ***Statically typed*** dan ***Type-safe memory model***, melindungi *developer* dari sekumpulan *bug* umum yang sering muncul dan *security flaw* dari kode yang kita tulis.

Dengan begitu pengembangan *software* menjadi lebih aman.

Garbage Collection

Go adalah bahasa pemograman yang mendukung ***Garbage Collection*** sehingga kita tidak perlu memikirkan permasalah alokasi dan dealokasi memori.

Unicode

Go adalah bahasa pemograman yang mendukung ***Unicode*** agar dapat mencetak semua sistem bahasa manusia diseluruh dunia.

Pada bahasa pemrograman yang mendukung *dynamically typed language* seperti *javascript*, *python, php* dan *ruby*, mereka berpendapat produktivitas pengembangan akan meningkat jika kita tidak perlu khawatir dan berfokus pada *types* dan *memory*.

Namun pada keunggulan tersebut terdapat *trade-off*, yaitu berupa *downside* pada sisi *performance*, efisiensi *memory* dan *bug* seputar *type-mismatch*.

Go mendeklarasikan dirinya dapat memberikan produktivitas yang sama seperti pada bahasa pemrograman yang mendukung *dynamically typed language*, namun tidak menjual sisi *performance* dan efisiensi.

3. *Go is Multicore Programming*

Ada fakta yang menarik, seluruh bahasa pemrograman pupolar yang ada saat ini tidak didesain untuk bisa memanfaatkan ***multiple CPU Core (Multicore)***. Bahasa pemrograman go telah di desain oleh para Robert, Rob dan Ken agar kita dapat menulis *parallel* dan *concurrent code* secara aman. Sehingga kita dapat memanfatkan keuntungan *modern multicore CPU* dan *Cloud Computing*.

Transformasi *performance computer* terus mengalami perubahan, dengan kondisi *status quo* dunia komputer saat ini. Hari ini berdiskusi mengenai kecepatan artinya bagaimana kita bisa memanfaatkan konsep *parallel* atau *concurrency*. Kenapa bisa begitu?

Di bawah ini adalah opini penulis, ada beberapa faktor kenapa *multi-core programming* akan membawa standar dan tren pemrograman di masa depan :

1. Berdasarkan ***Moore Law*** setiap 18 bulan sekali total *transistor* dalam *Central Processing Unit* (*CPU*) meningkat dua kali lipat. *Central Processing Unit* (*CPU*) mampu menampung lebih banyak *transistor* dan ukuranya semakin kecil.
2. Setiap 18 bulan sekali kecepatan *Central Processing Unit* (*CPU*) terus meningkat dua kali lipat, dimulai dari semenjak tahun 1950.
3. Namun setelah 50 tahun lebih, tepatnya pada tahun 2002 terdapat batasan dalam desain sirkuit *Central Processing Unit* (*CPU*), menuntut adanya arsitektur baru.
4. Solusi atas permasalahan di atas adalah *multi-core processor*, setelah tahun 2002 dalam satu *chip* bisa terdapat 2 *core*, 4 *core*, 8 *core* lebih *processor*.
5. Namun pada masing-masing core tidak mengalami peningkatan kecepatan, hanya kuantitas *core* saja yang bertambah. Hukum *moore law* masih berjalan, bukan lagi dengan menambah *transistor* dalam *Central Processing Unit* (*CPU*) tetapi menambah *core* dalam 1 chip tunggal.

6. *Chip manufacturer* mulai berhenti menambah *transistor*, *trend* mulai berfokus menambahkan *core* dan *cache*. Namun menambah jumlah *core* juga terbatas pada *cost* secara *economic*.
7. Menambah jumlah *cache* juga pada akhirnya akan menemukan batasan, *the bigger the cache, the slower it gets*.
8. Dari permasalahan di atas, jelas kita tidak bisa lagi mengandalkan inovasi peningkatan *hardware* melainkan bagaimana cara membangun *software development* yang efisien.
9. Berdasarkan keadaan *status quo* industri *chip* saat ini, para *software engineer* harus memanfaatkan *multicore processor* dengan konsep *concurrency* untuk memaksimalkan kapabilitas *hardware*.

Subchapter 2 – Setup Go Lang ✓

Subchapter 2 – Objectives

- Memahami ***Go Workspace***
- Memahami Konfigurasi ***GOPATH***
- Memahami ***Compilation*** pada ***Go***
- Memahami ***Execution*** pada ***Go***
- Memahami ***Documentation*** pada ***Go***
- Memahami ***Go Playground***

Sebelum kita memulai praktik menulis kode menggunakan ***Go language*** kita harus memahami terlebih dahulu apa itu ***GOPATH***. Konfigurasi ***GOPATH*** perlu dibuat terlebih dahulu agar kita memiliki sebuah ***Workspace***.

Go Workspace adalah tempat ***Go*** memanajemen ***source files, compiled binaries*** dan ***objects*** yang di ***cache*** agar kompilasi menjadi lebih cepat.

1. *Configure GOPATH*

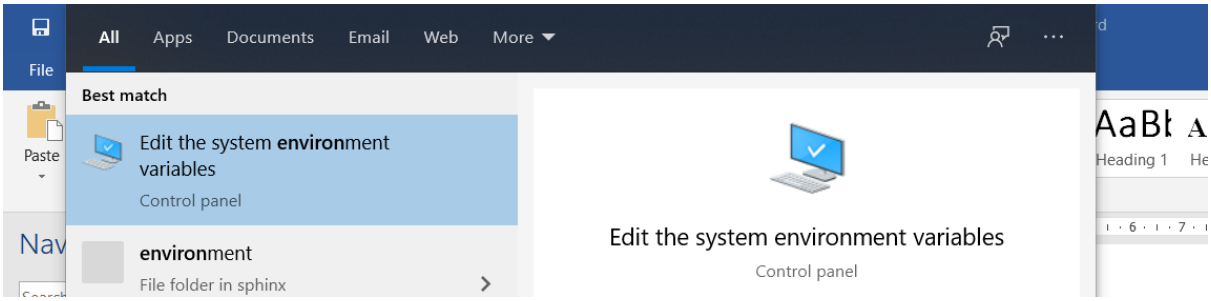
Konfigurasi ***GOPATH environment variable*** dibuat fungsinya untuk menentukan lokasi tempat kita akan membuat proyek dengan ***Golang***.

Jika anda tidak membuat ***custom GOPATH*** sendiri, maka di bawah ini adalah konfigurasi ***default*** lokasi GOPATH berdasarkan sistem operasi :

1. Lokasi **GOPATH** berdasarkan sistem operasi ***Windows* :**
   %USERPROFILE%\go atau **C:\Users\Gun Gun Febrianza\go**
2. Lokasi **GOPATH** berdasarkan sistem operasi ***Unix-based*** :
   $HOME/go

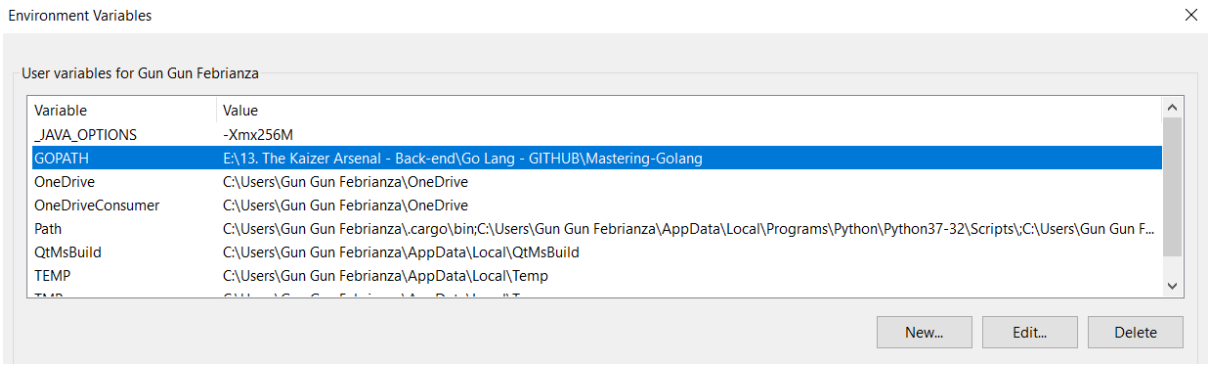
Setup GOPATH for Windows

Untuk membuat ***custom GOPATH environment variable*** pada sistem operasi *windows* ketik *environment* pada kolom pencarian di pojok kiri bawah, tunggu sampai *editor environment variable* muncul:



Gambar 56 Edit Environment Variable

Sebelum memulai penulisan kode buatlah ***Environment Variables*** dengan nama **GOPATH**. Isi ***Value*** dengan lokasi tempat kita akan menyimpan *project* yang akan di buat, di bawah ini adalah alamat ***custom GOPATH environment variable*** Penulis :



Gambar 57 Add Environment Variable

Setup GOPATH for MacOS

Untuk pengguna ***MacOS***, ***export path*** ke **~/.bash_profile** dengan perintah berikut :

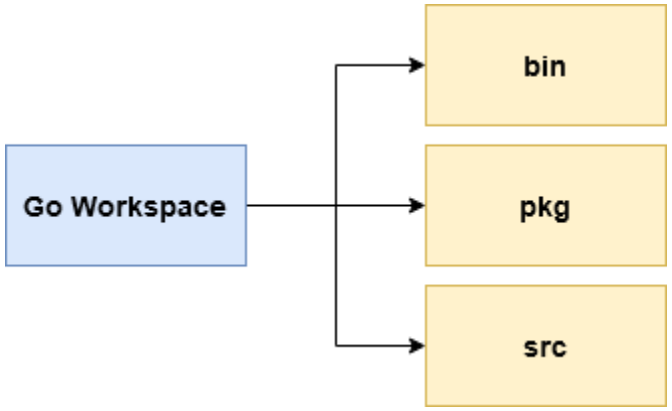
```
$ echo "export GOPATH=$HOME/Documents/go" >> ~/.bash_profile

$ source ~/.bash_profile
```

Setup GOPATH for Linux

Untuk pengguna ***Linux***, ***export path*** ke **~/.bashrc** dengan perintah berikut :

```
$ echo "export GOPATH=$HOME/Documents/go" >> ~/.bash_profile

$ source ~/.bash_profile
```

Setelah itu buatlah di dalam ***GOPATH*** buatlah 3 ***folder*** :



Gambar 58 Go Workspace Structure

Struktur penamaan ***folder*** untuk ***workspace*** telah ditentukan oleh ***Go***, adapun penjelasan ketiga ***folder*** tersebut adalah :

Folder bin

Folder ini digunakan untuk menyimpan ***binary file*** yang akan diproduksi oleh ***go program***.

Folder pkg

Folder ini digunakan untuk menyimpan ***package files*** yang dikompilasi.

Folder src

Folder ini digunakan untuk menyimpan seluruh kode yang kita buat. Setiap kali kita ingin membuat sebuah program yang baru, maka anda cukup masuk kedalam ***folder*** **src** kemudian buat lagi ***folder*** baru sesuai dengan nama program yang ingin anda buat.

2. Go Compilation

Sudah menjadi tradisi jika kita berkenalan dengan suatu bahasa pemrograman kita akan membuat program yang paling sederhana yaitu **Hello World**.

Kita akan belajar melakukan kompilasi ***Go program*** untuk menampilkan pesan *hello world*, untuk memulainya di dalam ***folder*** **src** buatlah sebuah folder dengan nama 1.Intro. Kemudian buat *file* bernama **helloworld.go**, lalu tulis kode di bawah ini :

```
package main

import (
    "fmt"
)

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

Untuk melakukan kompilasi eksekusi perintah di bawah ini :

```
$ go build helloworld.go
```

Jika berhasil maka dalam direktori yang sama akan muncul *binary file* bernama **helloworld.exe**

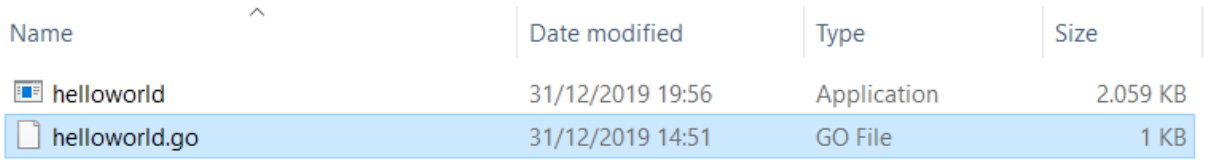
Untuk mendapatkan informasi lebih detail lagi, eksekusi perintah di bawah ini :

```
$ file helloworld

helloworld: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
```

Selanjutnya kita dapat mengeksekusi ***file executable*** yang telah di buat :

```
$ helloworld

Hello World!
```

Jika kita lihat lagi ukuran ***executable*** menjadi lebih besar, hal karena ***executable*** sudah mendukung karakteristik *statically linked yang* artinya tidak memerlukan lagi *libraries external* untuk berjalan.



Gambar 59 Compilation Output

3. *Go Execution*

Selain melakukan kompilasi juga terdapat cara lain untuk mengeksekusi program yang ditulis dengan ***Go*** tanpa harus memproduksi ***executable***. Untuk melakukanya eksekusi perintah di bawah ini :

```
$ go run helloworld.go
Hello World!
```

4. *Go Documentation*

Go menyediakan program bernama ***go doc*** untuk memudahkan kita mendapatkan sebuah dokumentasi tanpa harus terhubung ke internet.

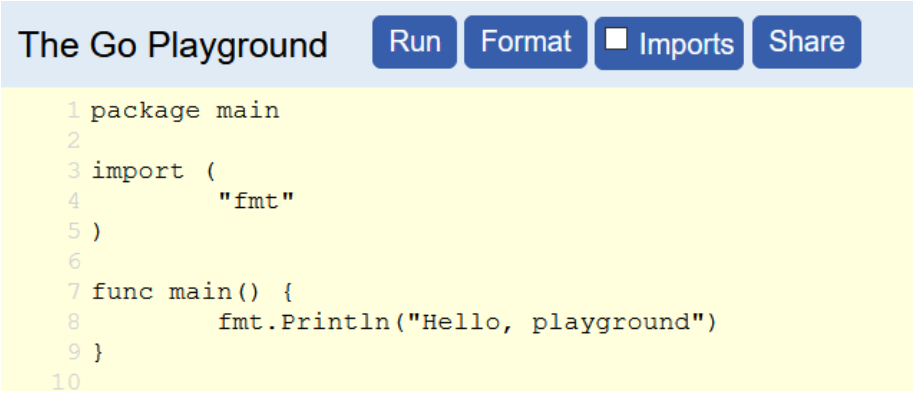
Sebagai contoh jika kita ingin menampilkan informasi mengenai ***function*** **Printf()** dari ***fmt package***, eksekusi perintah di bawah ini :

```
$ go doc fmt Printf
```

5. *Go Playground*

Selain menulis kode menggunakan ***Editor*** yang kita sukai, ***Go*** juga menyediakan ***editor*** versi ***online*** untuk menguji kode ***golang*** yang kita tulis di :

https://play.golang.org/



Gambar 60 Go Playground

Subchapter 3 – Go Program ✓

The goals of the Go project were to eliminate
the slowness and clumsiness of software development
at Google.

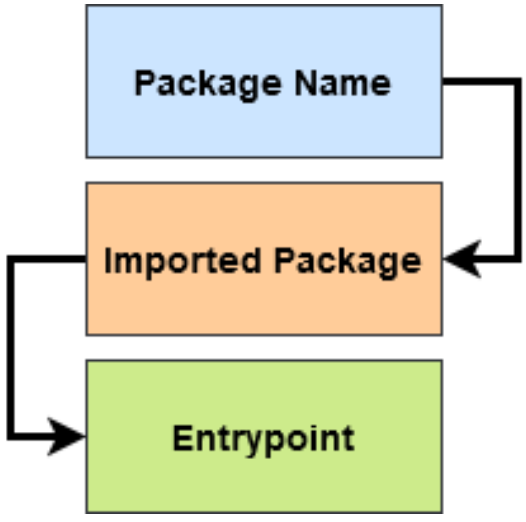
Subchapter 3 – Objectives

- Mempelajari ***Basic Structure Go Program***
- Mempelajari ***Comment*** dalam ***Go***
- Mempelajari ***Expression & Operator*** dalam ***Go***
- Mempelajari ***Arithmetic Operator & Operation*** dalam ***Go***
- Mempelajari ***Comparison Operator & Operation*** dalam ***Go***
- Mempelajari ***Logical Operator & Operation*** dalam ***Go***
- Mempelajari ***Assignment Operator & Operation*** dalam ***Go***
- Mempelajari ***Variable Declaration*** dalam ***Go***
- Mempelajari ***Reserved Words*** dalam ***Go***

1. *Basic Structure*

Di bawah ini adalah struktur paling dasar ***Go program***, terdapat 3 komponen utama yaitu ***Package Name***, ***Imported Package*** dan ***Entrypoint***:

```go
package main

import (
  "fmt"
)

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



Package Name

Setiap program yang ditulis menggunakan ***Go*** pasti memiliki ***keyword*** **package**. Seluruh kode di atas adalah bagian dari **package main**. Kita menggunakan **package** spesial bernama **main**, sebuah **package** yang dibutuhkan agar program bisa berjalan ketika dieksekusi.

Imported Package

Selanjutnya terdapat sebuah ***statement*** yang di dalamnya terdapat ***keyword*** **import**, *keyword* tersebut digunakan ketika kita ingin menggunakan sebuah **package** *built-in* yang telah disediakan ***Go***.

Pada kode sumber di atas kita menggunakan **package fmt**. Kegunaan dari **package fmt** adalah kita dapat berinteraksi dengan *input* dan *output*, pada kode sumber di atas kita menggunakan fungsi **Println** dari **package fmt** untuk menampilkan teks pada *standard output*.

Entrypoint

Pada kode Selanjutnya di atas kita memiliki fungsi **main()** yang bekerja sebagai *entrypoint* tempat pertama kali kode akan di baca saat program dieksekusi.

2. Comment

Dalam prinsip dasar ***software engineering***, kita harus membuat ***code documentation*** yang baik melalui komentar.

Manfaat penggunaan komentar adalah ketika program yang kita buat semakin kompleks dan banyak, maka ***comments*** yang kita buat bisa mempermudah kita untuk memahaminya kembali.

Jika anda bekerja dalam tim maka kode yang anda buat juga dapat mempermudah tim lainya untuk memahami kode yang anda buat.

Comment yang kita buat akan diabaikan oleh ***Go compiler***.

Di bawah ini adalah contoh pembuatan *single line* dan *multi line comment* :

```
package main

/*
1 multiline comment
2 multiline comment
n multiline comment
*/
import (
  "fmt"
)

//entrypoint – single line comment
func main() {
  fmt.Println("Hello World!")
}
```

3. *Expression & Operator*

Statement

Apa itu ***Statement***?

Statement adalah perintah untuk melakukan suatu aksi. Sebuah program terdiri dari sekumpulan ***statement***. Sebuah ***statement*** bisa berupa :

Declaration Statement

Declaration statement untuk membuat sebuah variabel,

Assignment Statement

Assignment statement untuk memberikan nilai pada sebuah variabel,

Invocation Statement

Invocation statement untuk memanggil sebuah ***block of code***,

Conditional Statement

Conditional statement untuk mengeksekusi kode berdasarkan kondisi,

Iteration Statement

Iteration statement untuk mengeksekusi kode secara berulang,

Disruptive Statement

Disruptive statement untuk keluar dari sebuah *control flow*.

Expression Statement

Expression statement untuk evaluasi sebuah *expression*,

Sebuah *statement* dapat memiliki sebuah *expression* atau tidak sama sekali (*non-expression statement*), tapi apa itu ***Expression***?

Expression

Expression adalah suatu ***statement*** yang digunakan untuk melakukan komputasi agar memproduksi suatu nilai. ***Expression*** adalah kombinasi dari ***literal***, ***variable***, ***operand*** dan ***operator*** yang dievaluasi untuk memproduksi sebuah *value*.

Di bawah ini terdapat dua buah ***statement*** :

```go
package main

import (
  "fmt"
)

func main() {
  var A int  // declaration statement
  A = 2 * 10 // expression statement
  fmt.Println(A)
}
```

Pada ***statement*** pertama di dalam ***function*** **main()** terdapat perintah untuk melakukan deklarasi variabel. ***Statement*** berikutnya di sebut dengan ***expression statement*** karena terdapat operasi untuk memproduksi suatu nilai.

Operator Precedence

Ketika sebuah ***expression*** memiliki lebih dari satu ***operator*** maka terdapat aturan untuk mengatur ***operator precedence***. Pada contoh kode di bawah ini ***compiler*** mengetahui mana operasi aritmetika yang harus dilakukan pertama kali :

```go
func main() {
  var A int  // declaration statement
  A = 2 + 2 * 10 // expression statement
  fmt.Println(A) // 22
}
```

Operasi perkalian akan dilakukan terlebih dahulu sebelum operasi penjumlahan.

Block of Code

Sebuah ***block*** terdiri dari serangkaian ***statements*** atau ***zero statement*** yang berada di sekup (***scope***) dalam ***curly braces***, dan secara ***semantic*** beraksi sebagai sebuah ***single syntactic statement***.

Pada contoh kode di bawah ini fungsi **main()** bekerja menjadi *single syntactic statement* yang membungkus setiap *statement* di dalamnya :

```go
func main() {
  var A int  // first block of code
  A = 2 * 10
  fmt.Println(A) // last block of code
}
```

Operator & Operand

Operator yang digunakan dalam *expression* kode sumber sebelumnya adalah *multiplication* (*), angka 2 dan 10 adalah sebuah *operand*.

```go
A = 2 * 10 // expression statement
```

Dalam sebuah bahasa pemrograman terdapat sekumpulan operator yang digunakan untuk mengekspresikan suatu *statement*. Salah satunya adalah *simple arithmetic operator*.

Arithmetic Operator

Di bawah ini adalah *arithmetic operator* dalam *Go*. Tidak hanya penjumlahan, pengurangan, perkalian dan pembagian. *Go* menyediakan *operator* untuk *remainder*, *unary, pre & post increment* dan *pre & post decrement* :

Tabel 5 Arithmetic Operator

Operator	***Description***	***Example***
+	*Addition (String Concat)* atau penjumlahan	`2+1 //3`
-	*Subtraction* atau pengurangan	`2-1 //1`
/	*Division* atau pembagian	`6/2 //3`
*	*Multiplication* atau perkalian	`3*2 //6`
%	*Remainder*	`3%2 //1`
-	*Unary Negation*	`-x // negative x`
+	*Unary Plus*	`+x // positive x`
++	*Post-Increment*	`X++`
--	*Post-Decrement*	`x--`

Arithmetic Operation

Contoh *arithmetic operation* dalam *Go* :

```
package main
```

```go
import "fmt"

func main() {
  a := 3
  var result int
  result = (100 + 50) * a
  fmt.Println(result) //450
}
```

Modulus Operation

Contoh *modulus operation* dalam *Go* :

```go
package main

import "fmt"

func main() {
  fmt.Println(12 % 5) //2
  fmt.Println(-1 % 2); // -1
  fmt.Println(1 % -2); // 1
  fmt.Println(1 % 2); // 1
  fmt.Println(2 % 3); // 2
  fmt.Println(-4 % 2); // 0
}
```

Pada baris pertama terdapat 12 mod 5 hasilnya adalah 2, karena angka *integer* 5 maksimum hanya dapat di kalikan dua kali saja (5*2) atau (5+5) yaitu sama dengan 10.

Angka *integer* 5 tidak bisa dikalikan tiga kali karena tidak bisa lebih dari 12. Angka *integer* hanya dapat dikalikan dua kali saja maka hasil mod adalah 12-10 = 2

Comparison Operator

Ada beberapa ***Comparison Operator / Relational Operator*** yang bisa kita gunakan untuk mengatur kondisi dan *control flow*. Bisa kita lihat pada tabel *Comparison Operator* di bawah ini, sebagai contoh diketahui nilai x = 5 maka :

Tabel 6 Relational Operator

Operator	***Description***	***Example***	***Return***
==	Membandingkan dua buah *operand* apakah setara atau tidak, jika setara atau sama maka hasilnya adalah *true*	x == 9	*False*
!=	Membandingkan dua buah *operand* apakah setara atau tidak, jika tidak setara atau tidak sama maka hasilnya adalah *true*	x != 8	*True*
>	Membandingkan dua buah *operand* apakah nilai *operand* kanan lebih besar daripada nilai *operand* kiri	x > 8	*False*
<	Membandingkan dua buah *operand* apakah nilai *operand* kiri lebih kecil daripada nilai *operand* kanan	x < 3	*False*
>=	Membandingkan dua buah *operand* apakah nilai *operand* kanan lebih besar atau sama daripada nilai *operand* kiri	x >= 5	*True*

<=	Membandingkan dua buah *operand* apakah nilai *operand* kiri lebih kecil atau sama daripada nilai *operand* kanan	x <= 1	*False*

Saat kita melakukan *comparison* terhadap dua buah *operand* terdapat hasil berupa *data type boolean*. Perhatikan kode di bawah ini :

```
func main() {
  fmt.Println(4 > 1)  // Output : true
  fmt.Println(4 == 2) // Output : false
  fmt.Println(4 != 2) // Output : true
}
```

Kita juga bisa melakukan perbandingan *string* mana yang lebih besar atau biasa disebut ***lexicographical order***. Dalam bahasa sederhana string di bandingkan berdasarkan abjad per abjad. Perhatikan kode di bawah ini :

```
func main() {
  //Lexicography
  fmt.Println('Z' > 'C')       // Output : true
  fmt.Println("Glow" > "Glee") // Output : true
}
```

Logical Operator

Jika terdapat lebih dari satu perbandingan dalam satu kondisi kita bisa menggunakan *logical operator*. Bisa kita lihat pada tabel *Logical Operator* di bawah ini, sebagai contoh diketahui x = 4 dan y = 5 maka :

Tabel 7 Logical Operator

Operator	*Description*	*Example*	*Return*
&&	Bernilai *true* jika dua *expression* bernilai *true*. Bernilai *false* jika sebaliknya.	(x > 3 && y = 5)	*True*
||	Bernilai *true* jika salah satu *expression* terdapat *true*. Bernilai *false* jika sebaliknya.	(x > 3 || y = 5)	*True*
!	Bernilai *true* jika *operand* bernilai false. Bernilai *false* jika sebaliknya.	!(x!=y)	*False*

Operator OR

Operator OR (**||**) digunakan jika kondisi nilai yang diberikan benar atau dapat diterima oleh salah satu kondisi. Perhatikan contoh kode di bawah ini :

```
func main() {
  hour := 8
  if hour < 9 || hour > 17 {
    fmt.Println("The shop still closed")
  }
}
```

Operator AND

Operator AND (**&&**) digunakan jika kita ingin menguji nilai yang diberikan namun harus memenuhi dua kondisi sekaligus. Perhatikan kode di bawah ini :

```
func main() {
```

```
  hour := 4
  minute := 30

  if hour == 4 && minute == 30 {
    fmt.Println("Alarm on!")
  }
}
```

Operator NOT

Operator NOT (**!**) digunakan jika kita ingin mengubah suatu *operand* kedalam *data type boolean*, *return* berupa *true or false* secara kebalikan (*inverse value*). Perhatikan contoh kode di bawah ini :

```
func main() {
  boolean := false
  fmt.Println(!true)    // Output : false
  fmt.Println(!boolean) // Output : true
}
```

Assignment Operator

Di bawah ini adalah ***assignment operator*** dalam ***Go***. Kita dapat menggunakanya untuk menetapkan sebuah nilai dengan berbagai cara :

Tabel 8 Assignment Operator

Operator	***Description***	***Example***
=	Menetapkan sebuah ***literal*** pada sebuah variabel. Literal dapat berupa data tunggal atau sebuah ***expression***.	C = A → C = 12 C = A * B → C = 12 * 2
+=	Menetapkan sebuah literal pada ***operand*** kiri, hasil dari ***operand*** kiri adalah penjumlahan ***operand*** kiri dengan ***operand*** kanan.	C += A → C = C + A 12 += 2 → 12 = 12 + 2
-=	Menetapkan sebuah literal pada ***operand*** kiri, hasil dari ***operand*** kiri adalah selisih dari ***operand*** kiri dengan ***operand*** kanan.	C -= A → C = C - A 12 -= 2 → 12 = 12 - 2
*=	Menetapkan sebuah literal pada ***operand*** kiri, hasil dari ***operand*** kiri adalah perkalian ***operand*** kiri dengan ***operand*** kanan.	C *= A → C = C * A 12 *= 2 → 12 = 12 * 2
/=	Menetapkan sebuah literal pada ***operand*** kiri, hasil dari ***operand*** kiri adalah pembagian ***operand*** kiri dengan ***operand*** kanan.	C /= A → C = C / A 12 /= 2 → 12 = 12 / 2
%=	Menetapkan sebuah literal pada ***operand*** kiri, hasil dari ***operand*** kiri adalah modulus ***operand*** kiri dengan ***operand*** kanan.	C %= A → C = C % A 12 = 2 → 12 = 12 % 2

4. *String*

Sebuah ***string*** adalah data teks, kata ***string*** sendiri berasal dari kata "***string of character***" kata ini digunakan pada abad 19 di akhir tahun 1800 oleh para ***typesetter*** yang kemudian didukung para matematikawan untuk merepresentasikan serangkaian simbol.

Dalam bahasa pemrograman ***Go***, ***string*** diperlakukan dengan cara yang berbeda dari bahasa pemrograman lainnya seperti ***Java, C++*** atau ***Python***. Sebuah ***string*** dalam ***Go*** adalah serangkaian ***bytes*** tunggal, setiap karakter memiliki ***byte*** masing-masing.

```
func main() {
  fmt.Println(len("maudy")) //5 bytes
}
```

Pada kode di atas kita menggunakan ***function*** **len** untuk membaca jumlah ***byte*** dalam sebuah ***string***, pada ***string*** **maudy** terdapat 5 ***bytes***.

Karakter atau simbol dari bahasa china untuk satu unit nya dapat memakan 3 ***bytes***, kita dapat membuktikannya pada kode di bawah ini :

```
func main() {
  fmt.Println(len("日"))   //3
  fmt.Println(len("日hi")) //5
}
```

5. *Rune*

String digunakan untuk menampilkan serangkaian karakter dan ***runes*** digunakan untuk merepresentasikan sebuah ***character*** tunggal. Sebuah ***string literals*** dibungkus menggunakan ***double quote*** **(“”)** sementara ***rune literals*** dibungkus menggunakan ***single quote*** **(‘’)**

```go
package main

import (
  "fmt"
)

func main() {
  fmt.Println('A') //output : 65
  fmt.Println('€') //output : 8364
  fmt.Println('™') //output : 8482
}
```

Program yang ditulis menggunakan ***Go*** mendukung hampir sebagian besar bahasa yang ada diseluruh dunia, untuk mencapai hal itu ***Go*** memanfaatkan ***standard unicode*** untuk menyimpan sebuah ***runes***.

Runes disimpan dalam wujud ***numeric codes*** bukan dalam bentuk ***character***. Kita dapat membuktikanya dengan cara mengeksekusi kode di atas, hasilnya adalah ***numeric codes*** yang merepresentasikan ***character*** A.

6. *Numbers*

Bahasa pemrograman ***Go*** memiliki dua tipe *number* yaitu ***integer*** dan ***floating-point number***, pada ***floating-point number*** terdapat *decimal point* yang menjadi pembeda dari ***integer***.

Perhatikan kode di bawah ini :

```
package main

import (
  "fmt"
)

func main() {
  fmt.Println(20)   // Integer
  fmt.Println(3.14) // Floating-point number

  fmt.Println(3 + 2)     // Return Integer
  fmt.Println(3 - 2)     // Return Integer
  fmt.Println(3 * 2)     // Return Integer
  fmt.Println(3 / 2)     // Return Integer
  fmt.Println(3 / 2.0)   // Return Floating-point
  fmt.Println(3.0 / 2)   // Return Floating-point
  fmt.Println(3.0 / 2.0) // Return Floating-point
  fmt.Println(5.0 / 3.0) // Return Floating-point
}
```

Pada kode di atas kita juga melakukan operasi aritmetika dasar antara ***integer*** dengan ***integer, floating-point*** dengan ***floating-point*** dan silang antara ***integer*** dan ***floating-point.***

7. Boolean

Sebuah ***boolean*** hanya memiliki dua nilai yaitu ***literal true*** dan ***false***. Kode di bawah ini menghasilkan sebuah nilai ***boolean*** berupa ***true*** dan ***false***.

```
package main

import (
  "fmt"
)

func main() {
  fmt.Println(7 > 2)  // Return True
  fmt.Println(7 < 2)  // Return False
  fmt.Println(8 == 8) // Return True
  fmt.Println(8 == 2) // Return False
  fmt.Println(8 != 2) // Return True
  fmt.Println(3 <= 3) // Return True
  fmt.Println(4 >= 4) // Return True
}
```

8. *Import Package*

Kita akan mencoba memanipulasi *string* menggunakan **package** **strings** yang telah disediakan dalam ***Go lang***, perhatikan kode di bawah ini :

```
package main

import (
  "fmt"
  "strings"
)

func main() {
  fmt.Println(strings.Title("gun gun febrianza"))
// Output : Gun Gun Febrianza

  fmt.Println(strings.ToLower("Gun Gun Febrianza"))
// Output : gun gun febrianza

  fmt.Println(strings.ToUpper("gun gun febrianza"))
// Output :  GUN GUN FEBRIANZA
}
```

Pada kode di atas kita memanfaatkan ***function*** **Title()** untuk mengubah sebuah ***string*** agar memiliki ***title case*** (Huruf besar di awal).

Selanjutnya kita memanfaatkan ***function*** **ToLower()** untuk mengubah sebuah ***string*** menjadi huruf kecil semua dan terakhir kita memanfaatkan ***function*** **ToUpper()** untuk mengubah sebuah ***string*** menjadi huruf besar semua.

9. *Variable Declaration*

Variable

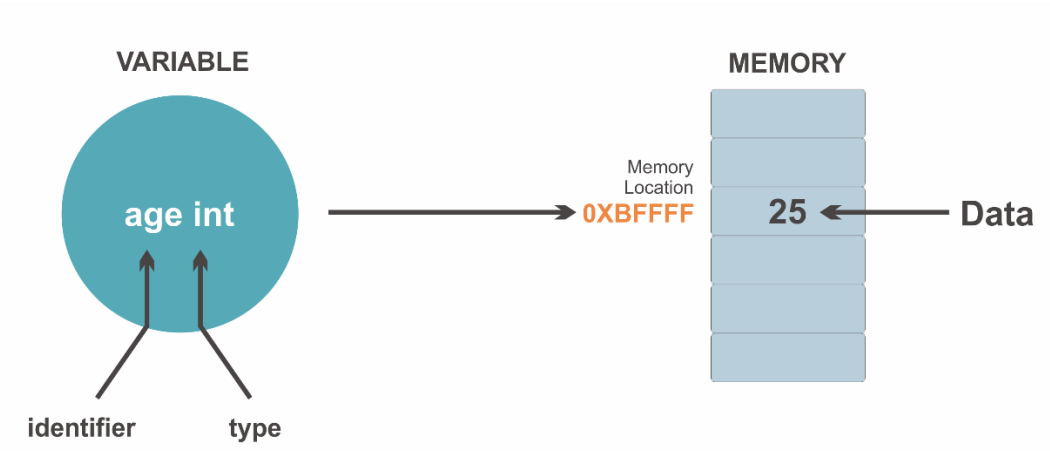
Setiap program pasti membutuhkan tempat untuk menyimpan suatu data. Data tersebut akan disimpan dilokasi memori tertentu. *RAM* (***Random Access Memory***) komputer terdiri dari jutaan sel memori. Ukuran dari setiap sel tersebut sebesar 1 *byte*.

Sebuah ***RAM*** komputer dengan ukuran 7 (***GigaBytes***) memiliki 8 x 1024MB= 8192 x 1024KB = 8.589.934.592 sel memori.

Saat anda menulis ***statement code*** di bawah ini :

```
var age int = 25;
```

Maka saat program ***running*** secara ***internal*** dalam sistem operasi di representasikan sebagai berikut :



Tabel 9 Variabel dalam memory

Variable adalah sebuah tempat yang digunakan untuk menampung dan mendapatkan kembali sebuah nilai (***value***) yang memiliki sifat dapat berubah sewaktu-waktu. ***Value***

adalah ***unit*** dari sebuah ***data***, sebuah ***value*** bisa berupa ***numeric value***, ***truth*** & ***falsity value***, ***character value*** dan ***text value*** [58].

Saat sebuah program dieksekusi dan di dalamnya terdapat aksi deklarasi variabel maka program akan mengirimkan permintaan alokasi ***memory*** pada sistem operasi. Selanjutnya sistem operasi akan mencari ***space memory*** yang cukup dan mengirimkan sebuah alamat ***memory*** kepada program.

Setiap kali program ingin membaca data yang tersimpan di dalam suatu variabel, maka yang diakses adalah alamat *memory* dari variabel tersebut. ***Value*** pada alamat memori (***memory location***) tersebut dapat diubah (***manipulate***), ***variable are the most basic form of data storage.***

Identifier

Identifier adalah nama yang digunakan oleh seorang *programmer* untuk merepresentasikan sebuah *function, variables,* dan sebagainya.

Setiap kali kita membuat *identifier* terdapat aturan dalam pembuatan namanya, tidak boleh menggunakan ***Reserved Keyword***. Pemberian *identifier* bersifat *case sensitive*.

Literal

Literal adalah representasi dari sebuah *value*.

Contoh literal adalah sebagai berikut :

1. 99 adalah *literal* untuk nilai sembilan puluh sembilan. Representasi alternatif lain untuk nilai 99 adalah 0x63 dalam bentuk *hexadecimal numeral*, 0x adalah *prefix* untuk *hexadecimal numeral*.
2. *True* adalah *literal* untuk merepresentasikan sebuah *truth* dan *False* adalah literal untuk merepresentasikan *falsity*.

3. *"Hello World"* adalah *string literal* yang merepresentasikan serangkaian karakter di dalam sebuah *quote* baik itu *single* atau *double quote*.
4. *Null* adalah *literal* untuk merepresentasikan ketidakhadiran sebuah nilai.

Di bawah ini adalah contoh deklarasi variabel lengkap dengan *reserved keyword, identifier* & *literal* :

```
var deeptech string = "DeepTech"

//var <- reserved keyword
//deeptech <- identifier
//"DeepTech" <- literal (string literal)
```

Binding

Proses pemberian *literal* pada variabel disebut dengan ***Binding*** atau ***Assignment Operation***. Pada contoh kode di atas kita melakukan ***binding string literal*** pada variabel dengan ***identifier*** **deeptech**.

Reserved Words

Dalam ***Go*** terdapat 1 grup ***reserved words*** yang tidak boleh digunakan sebagai ***identifier*** diantaranya adalah : ***Keyword.***

Reserved word adalah sebuah ***Token*** yang memiliki makna dan dikenali oleh ***Go Compiler***. ***Token*** adalah serangkaian ***character*** yang membentuk kesatuan tunggal. Misal *keyword* **var** adalah susunan dari ***character*** **v**, **a** dan **r**.

Di bawah ini adalah sekumpulan ***keyword*** dalam ***Go*** :

Keywords

Keywords adalah daftar nama yang telah disediakan oleh *compiler* sehingga tidak bisa lagi digunakan sebagai sebuah *identifier*.

Di bawah ini adalah kumpulan ***token*** yang menjadi *Go keywords* :

Tabel 10 Go Keyword

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

Notes
Jangan gunakan *Keyword* sebagai *Identifier*, *compiler* akan memproduksi *error*!

Naming Convention

Seperti yang telah kita fahami sebelumnya, ***identifier*** adalah nama yang menjadi pengenal pada suatu variabel. Penamaan ***identifier*** tidak boleh menggunakan ***reserved words*** yang telah disediakan ***Go***.

Di bawah ini adalah aturan dalam membuat ***identifier*** dalam ***Go*** :

1. Tidak dapat diawali dengan angka misal 7deeptech.

2. Dapat di awali dengan ***symbol*** _ misal _deeptech.
3. Dapat di awali dengan ***character*** dalam *unicode (*contoh π atau ö*)*. dilanjutkan dengan _ angka, atau *character* lagi misal : deep_tech atau dҽҽptech atau deeptech2024
4. Secara ***naming convention*** gunakan ***camelCase*** untuk membuat *identifier*, misal deepTech, kecilBesar atau gunGun (cirinya kata pertama tanpa huruf kapital disambung kata kedua menggunakan huruf besar).

```
var _deeptech string = "_deeptech"
fmt.Println(_deeptech)

var deep_tech = "deep_tech"
fmt.Println(deep_tech)

var dҽҽptech = "dҽҽptech"
fmt.Println(dҽҽptech)

var deeptech2019 = "deeptech2019"
fmt.Println(deeptech2019)

var namingConvention = "camelCase"
fmt.Println(namingConvention)
```

Di bawah ini adalah contoh penamaan ***identifier*** yang unik diizinkan selama dalam ruang lingkup ***unicode*** :

```
var ಠ_ಠ string = "ಠ_ಠ"
fmt.Println(ಠ_ಠ)

var ლ_ಠ益ಠ_ლ = "ლ_ಠ益ಠ_ლ"
fmt.Println(ლ_ಠ益ಠ_ლ)
```

```
var 〈〈 = "〈〈"
fmt.Println(〈〈)

// berbeda:
// 〈〈 << 〈〈;
```

Apa itu *Unicode*?

Go mendukung penamaan ***identifier*** menggunakan ***unicode***. Apakah anda tahu apa itu ***unicode***?

Unicode adalah suatu standar industri yang dirancang untuk mengizinkan teks dan simbol dari semua sistem tulisan di seluruh dunia untuk dapat ditampilkan dan dimanipulasi secara konsisten oleh komputer.

Case Sensitivity

Go adalah bahasa pemograman yang bersifat ***case sensitive***, jadi anda harus berhati-hati karena dua buah ***variable*** dengan nama pengenal (***identifier***) yang sama bisa memiliki nilai yang berbeda.

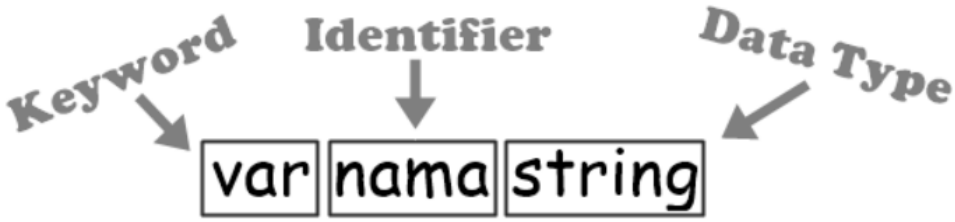
Penyebabnya adalah huruf besar dan huruf kecil yang diberikan pada sebuah ***identifier***.

```
package main

import (
  "fmt"
)

func main() {
  var A int // Case Sensitivity Test
```

```
    var a int // Case Sensitivity Test
    A = 10
    a = 15
    fmt.Println(A) // Output : 10
    fmt.Println(a) // Output : 15
}
```

Var Keyword

Dalam bahasa pemograman ***Go*** kita akan menggunakan ***keyword*** **var** untuk mendeklarasikan suatu ***variable***, perhatikan ilustrasi gambar di bawah ini :



Gambar 61 Variable Declaration Statement

Di bawah ini adalah kode sumber bagaimana melakukan deklarasi variabel dengan berbagai tipe data dalam bahasa pemrograman ***Go*** :

```
package main

import "fmt"

func main() {
  var name string            // Declare Variable
  var height, weight float64 // Declare Variable
  var age int                // Declare Variable

  name = "Gun Gun Febrianza"
  height, weight = 168, 55
  age = 27

  fmt.Println(name)
  fmt.Println("Nama : ", name, "usia", age)
  fmt.Println("Tinggi Badan :", height)
  fmt.Println("Berat Badan :", weight)
  /* Output :
```

```
  Gun Gun Febrianza
  Nama :  Gun Gun Febrianza  usia 27
  Tinggi Badan :  168
  Berat Badan :  55
  */
}
```

Constant Keyword

Deklarasi konstanta atau ***read-only variable. Constant*** bekerja seperti ***variable*** hanya saja kita tidak dapat mengubah nilainya (***value***).

Pada ***Go*** juga terdapat ***constant***, sebuah ***variable*** yang tidak bisa diubah karena hanya untuk dibaca saja (***Read only***). Sebuah ***Constant*** harus dimulai dengan abjad, ***underscore*** atau ***dollar sign*** selanjutnya boleh memiliki lagi konten ***alphabetic***, ***numeric*** dan ***underscore character***.

Sebuah ***Constant*** tidak dapat lagi diberi nilai atau di deklarasi ulang lagi. Di bawah ini contoh deklarasi ***constant*** dalam ***Go*** :

```
package main

import (
  "fmt"
)

func main() {
  const x = 10
  const y = 2
  fmt.Println(x * y)
}
```

Zero Value

Dalam bahasa pemrograman ***Go***, setiap variabel memiliki nilai ***default*** yang disebut dengan **zero value**. Jika kita mendeklarasikan variabel pada sebuah ***numeric data types*** tanpa menetapkan sebuah nilai maka variabel tersebut akan memiliki nilai nol.

Pada ***string data types*** nilainya adalah `""` dan pada ***boolean data types*** tipe datanya adalah ***false***.

```
package main

import (
  "fmt"
)

func main() {
  var x int
  var y string
  var z bool
  var pointer *int
  fmt.Println(x) // Output : 0
  fmt.Println(y) // Output : ""
  fmt.Println(z) // Output : false
  fmt.Println(pointer) // Output : <nil>
}
```

Short-Variable Declaration

Untuk mempercepat penulisan kode ***Go*** mendukung cara cepat untuk mendeklarasikan variabel. Konsep ini juga disebut dengan ***Type Inference***.

Perhatikan kode di bawah ini :

```go
package main

import (
  "fmt"
  "reflect"
)

func main() {
  nama := "Gun Gun Febrianza"
  fmt.Println(reflect.TypeOf(nama)) // Output : string
  umur := 27
  fmt.Println(reflect.TypeOf(umur)) // Output : int
  tagihan := 11.33
  fmt.Println(reflect.TypeOf(tagihan)) // Output : float64
  nganggur := true
  fmt.Println(reflect.TypeOf(nganggur)) // Output : bool
}
```

Multiple-variable Declaration

Penggunaan ***short-variable declaration*** lebih nyaman dan ringkas, kita akan lebih sering menggunakanya. Kita juga dapat melakukan ***multiple declaration*** sekaligus menggunakan ***short-variable declaration*** :

```go
tinggi, berat := 167, 56
fmt.Println(reflect.TypeOf(tinggi)) // Output : int
fmt.Println(reflect.TypeOf(berat))  // Output : int
```

Subchapter 4 – Data Types ✓

Without data, you're just another person with an opinion.
— W. Edward Deming

Subchapter 4 – Objectives

- Mempelajari Apa itu ***Data***?
- Mempelajari Apa itu ***Types***?
- Mempelajari Apa itu ***Data Types***?
- Mempelajari Apa itu ***Strongly & Dynamically Typed Language***?
- Mempelajari Apa itu ***Numeric Data Types***?
- Mempelajari Apa itu ***32 bit & 64 bit processor***?
- Mempelajari Apa itu ***Signed & Unsigned Integer***?
- Mempelajari Apa itu ***Real Number & Number Theory***?
- Mempelajari Apa itu ***Floating-point?***
- Mempelajari Apa itu ***Pointer***?
- Mempelajari Apa itu ***Stack*** & ***Heap***?

Untuk memahami konsep ***Data Types*** dalam *Go* ada beberapa konsep fundamental yang harus kita pelajari secara tersusun dan sistematis.

1. Apa itu *Data*?

Data dalam komputer secara *digital electronics* direpresentasikan dalam wujud ***Binary Digits (bits)*****,** sebuah unit informasi (*unit of information*) terkecil dalam mesin komputer. Setiap *bit* dapat menyimpan satu nilai dari ***binary number*** yaitu **0 atau 1**, sekumpulan *bit* membentuk konstruksi ***Digital Data***. Jika terdapat ***8 bits*** yang dihimpun maka akan membentuk ***Binary Term*** atau ***Byte***.

Pada *level byte* sudah membentuk unit penyimpanan (*unit of storage*) yang dapat menyimpan *single character*. Satu **data *byte*** dapat menyimpan 1 *character* contoh : 'A' atau 'x' atau '$'.

Serangkaian ***byte*** dapat digunakan untuk membuat ***Binary Files***, pada ***binary files*** terdapat serangkaian ***bytes*** yang dibuat untuk diinterpretasikan lebih dari sekedar ***character*** atau ***text***.

Pada ***level*** yang lebih tinggi (***kilobyte***, ***megabyte***, ***gigabyte*** & ***terabyte***) kumpulan *bits* tersebut dapat digunakan untuk merepresentasikan teks, gambar, suara dan video.

Data dalam konteks pemrograman adalah sekumpulan ***bit*** yang merepresentasikan suatu informasi.

2. Apa itu *Types*?

Types adalah sekumpulan nilai yang dikenali oleh ***compiler*** atau ***interpreter*** untuk mengatur bagaimana data harus digunakan. Sebuah ***types*** menentukan :

1. Data seperti apa saja yang dapat disimpan?
2. Seberapa besar memori yang dibutuhkan untuk menyimpan data?
3. Operasi apa saja yang dilakukan pada data tersebut?

3. Apa itu *Data Types*?

Data Types adalah sebuah ***set*** yang memiliki :

1. ***Characteristic***

   Karakteristik yang dimilikinya adalah nama ***keyword*** dan ukuran memori yang digunakannya untuk menyimpan data.
2. ***Range Value***

   Range Value adalah jangkauan variasi nilai (***value***) yang dapat disimpan.
3. ***Type***

   Mengacu pada ***numeric*** atau ***symbolic***.

Seluruh ***data types*** dibuat dari sekumpulan ***unit of memory*** yang disebut dengan ***byte***. Kita akan mengambil ***case study*** contoh ***data types*** dalam bahasa pemrograman ***Golang***, pada kasus ini yaitu **uint8** dan **int8** :

uint8 Case Study

Charateristic

Sebagai contoh terdapat sebuah ***data types*** dengan karakteristik :

1. Bernama ***uint8*** (*uint = unsigned integer*)
2. Ukuran memorinya sebesar 1 ***byte*** atau 8 ***bits***.

Terminologi ***unsigned integer*** artinya kita hanya dapat menyimpan data ***integer*** positif saja, kita tidak bisa menyimpan data ***integer*** negatif.

Range Value

Maka pada tipe data ***uint8*** kita bisa menampung ***set of integer*** dengan ***range value*** dari :

Tabel 11 uint Range Value

uint Range Value	
Minimum	***Maximum***
0	255

Kenapa bisa tipe data *uint8* dapat menyimpan nilai dengan jarak antara *minimum* 0 dan *maximum* 255?

Berdasarkan karakteristik ukuran memori yang dimiliki ***uint8*** kita mengetahui bahwa ukuranya adalah 1 ***byte***, pada ***chapter*** sebelumnya tentang Data Hierarchy kita sudah mengetahui bahwa ***byte*** adalah representasi dari 8-*bit*. Berarti jarak nilai yang mampu dijangkaunya sebesar 2^8 yaitu 256 dan hanya bisa diisi dengan nilai *non negative*.

Tabel 12 uint memory

uint memory	
Bytes	***bits***
1	8

Kenapa nilai maksimum *integer* yang dapat disimpan adalah 255 tidak 256?

Menjadi 255 karena perhitungan (***numeration***) dalam komputer dimulai dari angka nol tidak dari angka 1, sehingga nilai maksimumnya menjadi 255. Dengan begitu ***equation*** untuk menghitung maksimum nilai ***integer*** yang dapat disimpan adalah 2^8-1 = 255.

Types

Pada tipe data ***uint8,*** *type* yang digunakan adalah ***numeric*** sehingga kita dapat melakukan *numerical computation*.

Example Code

```
package main

import "fmt"

func main() {
  var maxUint8 uint8 = 255
  fmt.Println(maxUint8) // Output : 255

  /*  var anothermaxUint8 uint8 = 255 + 1
  fmt.Println(anothermaxUint8)  */
  // Output : constant 255 overflows uint32
}
```

int8 Case Study

Characteristic

Sebagai contoh terdapat sebuah *data types* dengan karakteristik :

3. Bernama ***int8*** dan
4. ukuran memorinya sebesar 1 *bytes*.

Sebelumnya kita mengetahui bahwa ***unsigned integer*** hanya dapat menyimpan data *integer* positif saja, pada *signed integer* kita bisa menyimpan data *integer* positif dan *integer* negatif.

Range Value

Maka pada tipe data ***int8*** kita bisa menampung *set of integer* dengan *range value* dari :

Tabel 13 int Range Value

int Range Value	
Minimum	***Maximum***
-128	127

Kenapa bisa tipe data *int8* dapat menyimpan nilai dengan jarak antara *minimum* -128 (negatif) dan *maximum* 127 (positif)?

Di karenakan ***int8*** mendukung nilai negatif dan positif maka 1 *bit* digunakan sebagai penanda negatif atau positif pada suatu *integer*.

Sehingga dari 8 bit tersisa 7 *bit*, artinya *pattern* data biner yang dapat diciptakan adalah 2^7 yaitu 128. Inilah yang menjadi faktor kenapa minimum *range value* dari ***int8*** adalah

-128 dan nilai maksimum yang dapat dicapai adalah 127 dikarenakan 0 juga dihitung dalam *numeration*.

Tabel 14 int memory

int memory		
Bytes	***Storage***	***Sign***
1	7 bit	1 bit

Types

Pada tipe data ***int8,*** *type* yang digunakan adalah *numeric* sehingga kita dapat melakukan *numerical computation*.

Example Code

```
package main

import "fmt"

func main() {
  var maxInt8 int8 = 127
  fmt.Println(maxInt8) // Output : 127

  /*  var anothermaxInt8 int8 = 127 + 1
  fmt.Println(anothermaxInt8)  */
  // Output : constant 128 overflows int8
}
```

4. Apa itu *Strongly & Dynamically Typed*?

Go adalah bahasa pemrograman yang memiliki karakteristik ***Statically Typed Language*** atau ***Static Typing***, artinya nilai dalam sebuah variabel harus memiliki tipe data.

Jika sebelumnya anda sudah mempelajari bahasa pemrograman seperti ***python*** dan ***javascript***, maka anda akan memahami ***Loosely Typed Languag***e atau ***Dynamic Typing***. Anda dapat membuat variabel tanpa harus menetapkan tipe data terlebih dahulu, baik itu berupa ***literal number***, ***string*** ataupun ***boolean logic***.

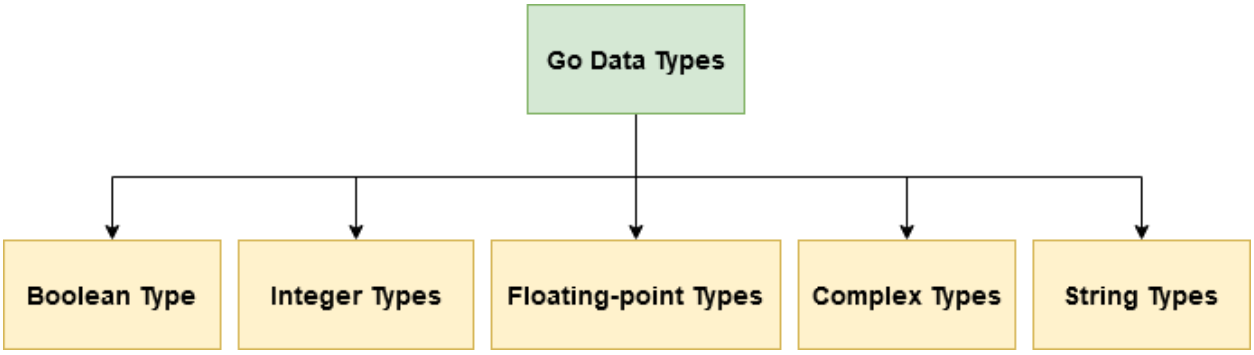
Sebuah bahasa pemograman dikatakan ***Strongly Typed*** jika sebuah *compiler* akan memproduksi ***error*** jika sebuah ***type*** digunakan dengan cara yang salah. Sebuah bahasa dikatakan ***Dynamically Typed*** karena saat ***runtime,*** tipe data dapat diubah ke tipe data lainya agar program bisa berjalan sesuai instruksi.

Pada ***Strongly Typed*** terdapat nilai ***correctness*** dan ***safety*** sementara pada ***Dynamically Typed*** terdapat ***simplicity*** dan ***speed*** untuk pengembangan ***software***.

Nilai ***correctness*** memberikan keunggulan dari segi ***performance***. Nilai ***safety*** memberikan manajemen integritas data yang baik, pendeteksian ***bug*** dan kemungkinan ***error*** saat ***runtime*** yang lebih baik.

Pada *Dynamically Typed* pengembangan *software* bisa menjadi lebih cepat, bersifat *less rigid* lebih mudah untuk membuat sebuah perubahan.

5. Go Data Types



Gambar 62 Go Data Types

Dalam bahasa pemograman ***Go*** terdapat 17 ***Built-in Types*** yang telah disediakan, diantaranya adalah :

1. ***Boolean Type*** seperti **bool.**
2. ***Integer Numeric Types*** seperti **int8**, **uint8**, **int16**, **uint16**, **int32, uint32**, **int64**, **uint64**, **int**, **uint**, dan **uintptr**.
3. ***Floating-point Numeric Types*** seperti **float32** dan **float64**.
4. ***Complex Numeric Types*** seperti **complex64** dan **complex128**.
5. ***String Type*** seperti **string**.

Jika kita perhatikan 15 dari 17 ***built-in types*** adalah ***numeric types***, seperti ***integer types, floating-point types*** dan ***complex types***. Selain itu bahasa pemograman ***Go*** juga mendukung alias pada ***built-in*** diantaranya adalah :

1. **byte** adalah alias dari **uint8**.
2. **rune** adalah alias dari **int32**.

Numeric Data Types

Maksimum angka yang dapat digunakan di dalam sebuah komputer tergantung dari arsitektur ***processor*** yang kita gunakan.

32 Bit & 64 Bit Processor

Kata **32** dan **64** ***Bit*** mengacu pada arsitektur ***processor***, **32** dan **64** ***bit*** menjadi pembeda kemampuan ***processor*** dalam melakukan akses pada ***memory*** melalui ***CPU Register***. Pada **32** ***bit*** kemampuan akses pada ***memory address*** mencapai 2^{32}.

Pada **64** ***bit*** kemampuan akses pada ***memory address*** mencapai 2^{64}.

Pada **32** ***bit processor*** kemampuan pengolahan untuk berinteraksi dengan *RAM* terbatas sampai dengan 4GB, sedangkan pada **64** ***bit processor*** kemampuan pengolahan untuk berinteraksi dengan ***RAM*** bisa lebih dari 4 GB.

Pada **64** ***bit processor*** dapat mengolah data lebih dari **32** ***bit processor***. Pada **64** ***bit processor*** kemampuan komputasi nilai lebih unggul termasuk ***memory address*** yang dimilikinya.

Selain itu, faktor signifikan pembeda antara ***32-bit processor*** dan ***64-bit processor*** adalah kemampuan ***calculation*** perdetiknya yang mempengaruhi kecepatan untuk menyelesaikan suatu pekerjaan.

Signed Integer

Pada ***Signed Integer*** kita dapat menyimpan nilai ***integer*** negatif dan ***integer*** positif. ***Signed Integer*** menggunakan 1 ***bit*** untuk merepresentasikan sebuah ***sign*** (***positive*** atau ***negative***).

Tabel 15 Signed Integer Types

No	***Types***	***Description***	***Range***
1	int8	*Signed 8-bit integers*	(-128 sampai 127)
2	int16	*Signed 16-bit integers*	(-32768 sampai 32767)

3	int32	*Signed 32-bit integers*	(-2147483648 sampai 2147483647)
4	int64	*Signed 64-bit integers*	*Signed 64-bit integers :* (-9223372036854775808 sampai 9223372036854775807)
5	int	*Platform Dependent*	*Platform Dependent*

Istilah ***Platform Dependent*** artinya dapat berupa ***Signed* 32 *bit integer*** dalam arsitektur mesin komputer **32 *bit*** dan ***Signed* 64 *bit integer*** dalam arsitektur mesin komputer **64 *bit***.

Anda akan memahami lebih dalam makna ***platform dependent*** di halaman selanjutnya tentang ***Architecture-independent Type*** & ***Implementation Specific Type***.

Di bawah ini adalah contoh ***Signed Integer*** dalam ***Golang*** :

```
func main() {
    var SignedInteger8 int8 = 127
    fmt.Println(SignedInteger8) // Output : 127

    var SignedInteger16 int16 = 32767
    fmt.Println(SignedInteger16) // Ouput : 32767

    var SignedInteger32 int32 = 2147483647
    fmt.Println(SignedInteger32) // Ouput : 2147483647

    var SignedInteger64 int64 = 9223372036854775807
    fmt.Println(SignedInteger64) // Ouput : 9223372036854775807

    // fmt.Println(SignedInteger32 + SignedInteger64)
    // invalid operation: SignedInteger32 + SignedInteger64 (mismatched types int32 and int64)
```

```
}
```

Unsigned Integer

Unsigned Integer hanya dapat menyimpan ***integer*** yang positif saja.

Tabel 16 Unsigned Integer Types

No	Types	Description	Range
1	uInt8	*Unsigned 8-bit integers*	(0 sampai 255)
2	uInt16	*Unsigned 16-bit integers*	(0 sampai 65535)
3	uInt32	*Unsigned 32-bit integers*	(0 sampai 4294967295)
4	uInt64	*Unsigned 64-bit integers*	(0 sampai 18446744073709551615)
5	uint	*Platform Dependent*	*Platform Dependent*

Di bawah ini adalah contoh *Unsigned Integer* dalam *Golang* :

```
func main() {
  var unsignedInteger8 uint8 = 255
  fmt.Println(unsignedInteger8) // Output : 255

  var unsignedInteger16 uint16 = 65535
  fmt.Println(unsignedInteger16) // Ouput : 65535

  var unsignedInteger32 uint32 = 4294967295
  fmt.Println(unsignedInteger32) // Ouput : 4294967295

  var unsignedInteger64 uint64 = 18446744073709551615
```

```
    fmt.Println(unsignedInteger64) // Ouput : 18446744073709551
615

    // fmt.Println(unsignedInteger16 + unsignedInteger8)
    // invalid operation: unsignedInteger16 + unsignedInteger8
(mismatched types uint16 and uint8)
  }
```

Operasi aritmetika antar ***number*** dengan tipe data yang berbeda tidak dapat dilakukan.

Terdapat 2 ***numeric types*** dalam bahasa pemrograman ***Golang*** :

Implementation Specific Type

Penggunaan `int` dan `uint` sangat disarankan karena terdapat arsitektur mesin komputer yang berbeda-beda (32 & 64 ***bit***).

Penggunaan ***keyword*** `int` dan `uint` akan membantu kompiler untuk memproduksi kode sesuai dengan target arsitektur mesin komputer yang dituju. Proses ini disebut dengan ***Implementation Specific Type***.

Jika kita membuat sebuah ***Go*** program di dalamnya terdapat tipe data `int` maka ketika dikompilasi dengan target arsitektur 32 ***bit*** ukuran tipe data juga 32 ***bit***, begitu juga jika kita kompilasi pada arsitektur 64 ***bit*** maka ukuran ***variable*** tetap 64 ***bit***.

Architecture-Independent Type

Pada ***Architecture-independent type***, ukuran data ***bit*** tidak berubah meskipun program dieksekusi di mesin komputer manapun.

Sebagian besar arsitektur mesin komputer hari ini sudah mencapai 32 ***bit*** dan 64 ***bit***. Jika kita membuat program dan di dalamnya memiliki tipe data `int32`, ukuranya akan tetap konstan ketika kita mengeksekusinya di arsitektur mesin komputer 64 ***bit***.

Formatting

Display Sign

Untuk menampilkan simbol positif dari suatu ***integer*** kita dapat menggunakan ***formatting*** **`+d`** seperti di bawah ini :

```
func main() {
  signedInteger := +12
  fmt.Printf("%v \n" , signedInteger) //12
  fmt.Printf("%+d" , signedInteger) //+12
}
```

Display Padding

Untuk menambahkan ***padding*** berupa ***digit*** nol kita dapat menggunakan ***formatting*** **`04d`** dan ***formatting*** untuk ***space*** menggunakan **`4d`**, angka 4 adalah jumlah ***padding***.

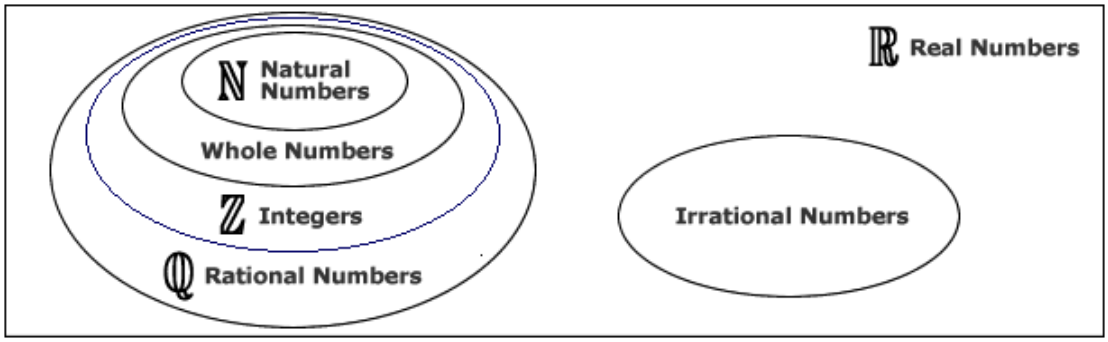
```
func main() {
  signedInteger := +12
  fmt.Printf("%04d" , signedInteger) //0012
  fmt.Print("\n")
  fmt.Printf("%4d" , signedInteger) //  12
}
```

Display Binary

Untuk menampilkan dalam bilangan biner gunakan ***formatting*** **b** seperti di bawah ini :

```
signedInteger := 12
fmt.Printf("%b" , signedInteger) //1100
```

Real Number & Number Theory



Gambar 63 Real Number Scope

Natural Number

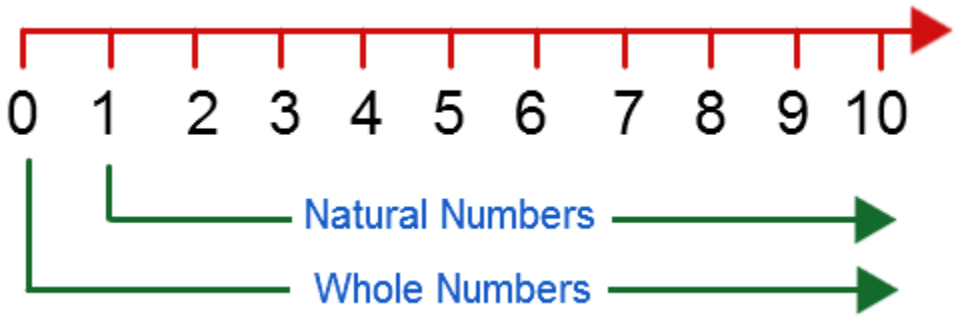
Konsep tentang ***real number system*** terus berevolusi dari waktu ke waktu, gagasan tentang konsep *number* terus berubah dari waktu ke waktu. Apa itu ***number***? ***Number*** adalah sesuatu yang bisa kita hitung sebagaimana kita bisa menghitung jumlah domba yang dimiliki seorang penggembala domba di zaman *the age of farming* saat peradaban masyarakat mayoritasnya adalah berternak. Pada kasus menghitung domba kita menyebutnya dengan ***natural number*** atau ***counting number***.

Natural number dimulai dari 1,2,3,4 … (notasi … secara simbolis bermakna dan seterusnya).

Whole Number

Ada masanya zaman dahulu konsep ***number*** pada akhirnya berkembang mengenal angka nol atau ***zero***, dengan begitu ketika seorang penggembala tidak memiliki domba maka kita bisa mengetahui bahwa penggembala memiliki nol domba atau *zero sheep*.

Penemuan angka nol membuat gagasan tentang konsep ***number*** berkembang menjadi ***whole number***, yaitu ***natural number*** yang disertai angka nol.

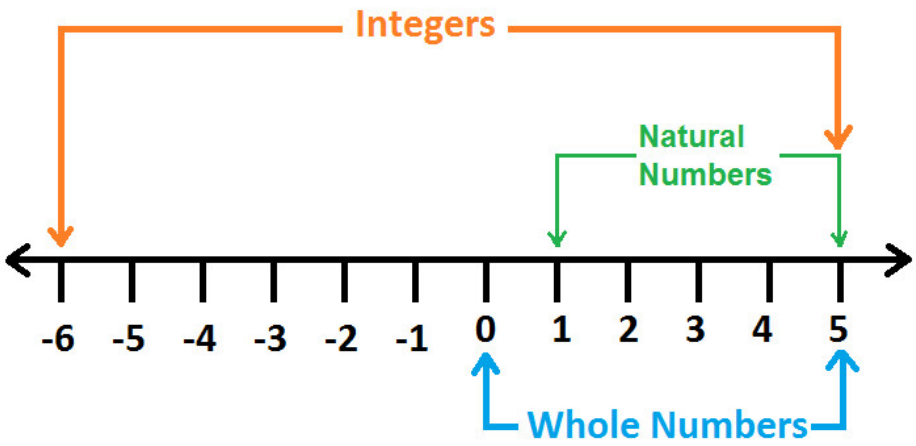


Gambar 64 Whole & Natural Number

Integer

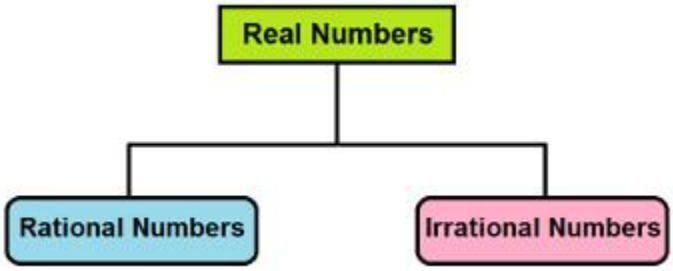
Selain nol gagasan selanjutnya adalah penemuan ***integers***, penggembala domba yang ingin memiliki domba bisa berhutang 5 buah permata agar bisa memiliki domba.

Tentu perlu ada representasi angka negatif, maka ***integer*** adalah jawabanya. Di bawah ini adalah ***scope*** sebuah ***integer*** :



Gambar 65 Integer Concept

Pada dasarnya sebuah ***real number*** tersusun oleh dua konsep ***number*** yaitu ***rational number*** & ***irrational number*** :



Gambar 66 Rational & Irrational Number

Ratio

Sebelum kita memahami ***Rational Number*** kita akan memahami dahulu apa itu ***Ratio***?

Ratio adalah jumlah satu hal yang relatif terhadap yang lain, sebuah perbandingan dua nilai. Sebuah ***ratio*** mempertimbangkan ukuran relatif dua buah angka. Sebuah ***ratio*** bisa diekspresikan sebagai berikut :

1. Sebagai sebuah ***Fraction*** misal $\frac{3}{10}$
2. Menggunakan simbol ***ratio*** (:), misal 3 : 10

3. Menggunakan kata “to” misal “3 to 10” (tiga dari sepuluh)

Rational Number

Rational Number adalah seluruh ***number*** dalam bentuk $\frac{a}{b}$ dimana *a* dan *b* adalah ***integer*** namun *b* tidak boleh bernilai nol ($b \neq 0$). ***Rational number*** biasanya disebut dengan ***fraction*** atau dalam bahasa indonesia kita menyebutnya pecahan. Susunan ***fraction*** juga terdiri dari ***numerator*** dan ***denominator.***

Dalam ***fraction*** $\frac{a}{b}$ dimana *a* adalah sebuah ***numerator*** dan *b* adalah sebuah ***denominator***.

$$\text{Fraction} = \frac{\text{Numerator}}{\text{Denominator}}$$

Gambar 67 Numerator & Denominator

Pada dasarnya seluruh ***integer*** merupakan sebuah ***rational number*** dengan dengan ***denominator*** 1, misal :

$$3 = \frac{3}{1}$$

Sebuah ***fraction*** bisa lebih kecil dari angka 1 misal :

$\frac{1}{2}$ atau $\frac{3}{4}$ sehingga diberi nama ***Proper Fraction***.

Jika sebuah ***fraction*** lebih besar dari angka 1 misal :

$\frac{4}{2}$ atau $\frac{10}{5}$ maka diberi nama ***Improper Fraction.***

Irrational Number

Irrational Number adalah ***real number*** yang tidak rasional. ***Irrational number*** tidak dapat diekspresikan dalam bentuk ***fraction***. Sebagai sebuah ***decimal*** sebuah ***irrational number*** bersifat ***never repeat*** atau ***terminate***.

Dibawah ini adalah perbandinganya dengan ***rational number*** :

$\frac{3}{4}$ = 0.75 (***Rational*** karena angka berhenti (***terminate***) di 0.75)

$\frac{2}{3}$ = 0.66666**6** (***Rational*** dengan angka yang sama 1 digit terus berulang (***repeat***))

$\frac{5}{11}$ = 0.454545**45** (***Rational*** dengan angka yang sama 2 digit terus berulang (***repeat***))

$\frac{5}{7}$ = 0.714285**714285** (***Rational*** dengan angka yang sama 6 digit terus berulang (***repeat***))

π ≈ 3.14159265... (***Irrational*** karena angka tidak berhenti dan tidak berulang (***never repeat*** & ***terminate***))

*Notasi ≈Artinya hampir mendekati atau hampir setara (***approximately equal to***)

$\sqrt{2}$ = 1.41421356... (***Irrational*** karena angka tidak berhenti dan tidak berulang (***never repeat*** & ***terminate***))

Floating-Point

Komputer menyimpan dan memanipulasi ***real number*** seperti ***constant*** **pi** 3.14 menggunakan standar ***floating-point IEEE754***. ***Go*** memiliki tipe data ***floating-point***, sehingga kita dapat berinteraksi dengan *Real Number*.

Go memiliki dua buah ***floating point*** yaitu `float32` dan `float64` yang direpresentasikan dalam **32 & 64 *bit memory***. Beberapa bahasa pemrograman memperkenalkan konsep tipe data ***double*** karena memiliki ***double precision*** untuk merepresentasikan ***64-bit floating point type***.

Nilai ***literal*** yang dapat dimiliki ***decimal integer***, ***decimal point***, ***decimal factional part*** dan ***integer exponent***.

Gambar 68 Example Float Literal

Pada gambar di atas angka setelah ***decimal point*** disebut dengan ***decimal fractional part***.

Di bawah ini adalah contoh kode ***floating-point*** dalam *Go* :

```
func main() {
  var decimalPoint1 float32 = 1.23
  fmt.Println(decimalPoint1) // Ouput : 1.23

  var decimalPoint2 float32 = 01.23
  fmt.Println(decimalPoint2) // Ouput : 1.23

  var decimalPoint3 float32 = .23
  fmt.Println(decimalPoint3) // Ouput : 0.23

  var decimalPoint4 float32 = 2.
  fmt.Println(decimalPoint4) // Ouput : 2
```

```
  var exponentPart1 float32 = 1.88e2
  fmt.Println(exponentPart1) // Output : 188

  var exponentPart2 float32 = 188e2
  fmt.Println(exponentPart2) // Output : 18800

  var exponentPart3 float32 = 188e-2
  fmt.Println(exponentPart3) // Output : 1.88
}
```

e Notation

Pada contoh kode di bawah ini untuk menghindari kesalahan penulisan ***number***, apalagi jika terdapat angka nol dan angka lainya yang sangat panjang. Untuk mengatasi hal ini kita bisa mempersingkatnya dengan menggunakan ***e notation*.**

```
func main() {
  var exponentPart1 float32 = 1.88e2
  fmt.Println(exponentPart1) // Output : 188

  var exponentPart2 float32 = 188e2
  fmt.Println(exponentPart2) // Output : 18800

  var exponentPart3 float32 = 188e-2
  fmt.Println(exponentPart3) // Output : 1.88
}
```

Precision

Pada `float32` kita menggunakan setengah ***memory*** dari `float64` sehingga menjadi kurang presisi.

Secara ***default*** jika kita membuat tipe data ***float*** dalam ***Go*** menggunakan ***short-variable declaration*** maka tipe data yang akan diberikan adalah `float64`.

Di bawah ini adalah perbedaan presisi `float32` dari dan `float64` :

```
package main

import (
  "fmt"
  "math"
)

func main() {
  var intVariable int = 100
  var float32Variable float32 = 100
  var float64Variable float64 = 100
  fmt.Println(intVariable / 3)
  // Output : 33
  fmt.Println(float32Variable / 3)
  // Output : 33.333332
  fmt.Println(float64Variable / 3)
  // Output : 33.333333333333336

  var pi64 = math.Pi
  var pi32 float32 = math.Pi
  fmt.Println(pi64)
  fmt.Println(pi32)
  /* Output :
    3.141592653589793
    3.1415927
  */
}
```

Formatting

Jika kita ingin mengatur seberapa banyak ***digit*** yang ingin ditampilkan saat menggunakan ***float***, maka kita harus menggunakan fungsi **`Printf`** menggunakan **`%f`**, perhatikan kode di bawah ini :

```
func main() {
  third := 1.0 / 3
  fmt.Println(third)
  fmt.Printf("%v\n", third)
  fmt.Printf("%f\n", third)
  fmt.Printf("%.3f\n", third)
  fmt.Printf("%.2f\n", third)
  /* Output :
  0.3333333333333333
  0.3333333333333333
  0.333333
  0.333
  0.33
  */
}
```

Untuk menentukan ***precision*** seberapa banyak digit yang kita inginkan setelah ***decimal point***, pada kode di atas kita menggunakan format **`%.2f`** artinya ***fractional part*** hanya ditampilkan 2 digita setelah ***decimal point.***

String Data Types

Sebuah ***string literal*** dapat menggunakan ***double quotation*** mark (“ … ”) atau ***backticks***.

Double Quote String

Pada ***double quote string*** kita dapat menerapkan ***escape sequence*** seperti :

1. **Newline `\n`**
2. **Tab `\t`**
3. **Double Quote \"**
4. **Backslash **

```
func main() {
  var normalstring string = "Hello \n gun"
  fmt.Println(normalstring) //
}

/*
Hello
gun
*/
```

Back Ticks

Pada ***back ticks*** kita dapat membuat ***string*** yang mengandung ***newline*** namun tidak dapat menggunakan ***escape sequence*** :

```
func main() {
  var backtick string = `Hello
  gun`
  fmt.Println(backtick) //
}

/*
Hello
        gun
*/
```

String Index

Untuk membuktikan bahwa ***string*** adalah serangkaian ***byte*** tunggal kita dapat melihat kode di bawah ini :

```
func main() {
  fmt.Println("Hello World"[1]) //101
  fmt.Println("Hello World"[2]) //108
  fmt.Println("Hello World"[3]) //108
}
```

Karakter **e** yaitu pada ***index*** pertama memiliki nilai ***byte*** 101, sementara karakter **l** pada ***index*** kedua dan ketiga memiliki nilai ***byte*** 108. ***Index*** pada ***string*** dimulai dari angka 0 yaitu karakter **H**.

String Concat

Untuk melakukan ***concatenation*** dalam ***Go***, cukup menggunakan ***operator*** **+** seperti pada kode di bawah ini :

```
func main() {
  fmt.Println("Hi" + "Maudy") //Hi Maudy
}
```

Booleans Data Types

Di dalam ***Go*** sebuah ***boolean*** hanya memiliki dua nilai yaitu ***True*** dan ***False Keywords***.

```
func main() {
  var beautiful bool = true
  ugly := false
  fmt.Println(beautiful) //true
  fmt.Println(ugly)      //false
}
```

Check Data Types

Go adalah bahasa pemograman dengan paradigma ***static typed language***, kita harus menentukan terlebih dahulu tipe data yang ingin kita gunakan sebelum membuat sebuah variabel.

Jika kita memberikan nilai yang tidak sesuai pada suatu tipe data, misal menyimpan ***string*** ke dalam ***integer*** maka ***error*** akan terjadi.

Dalam ***Go***, kita dapat mengetahui tipe data suatu nilai menggunakan **package** ***reflect***. Di dalamnya terdapat ***function*** **typeof()** yang bisa kita gunakan untuk mengetahui tipe data suatu nilai.

Perhatikan kode di bawah ini :

```
package main

import (
  "fmt"
  "reflect"
)

func main() {
  fmt.Println(reflect.TypeOf(2))            // Return : int
  fmt.Println(reflect.TypeOf(2.4))          // Return : float64
  fmt.Println(reflect.TypeOf("Hello Gun!")) // Return : string
  fmt.Println(reflect.TypeOf('g'))          // Return : int32
  fmt.Println(reflect.TypeOf(true))         // Return : bool
}
```

Apa itu *Stack* & *Heap*?

Stack digunakan untuk membuat alokasi ***Static Memory*** dan ***Heap*** digunakan untuk membuat ***Dynamic Memory***, keduanya disimpan di dalam ***RAM (Random Access Memory)***.

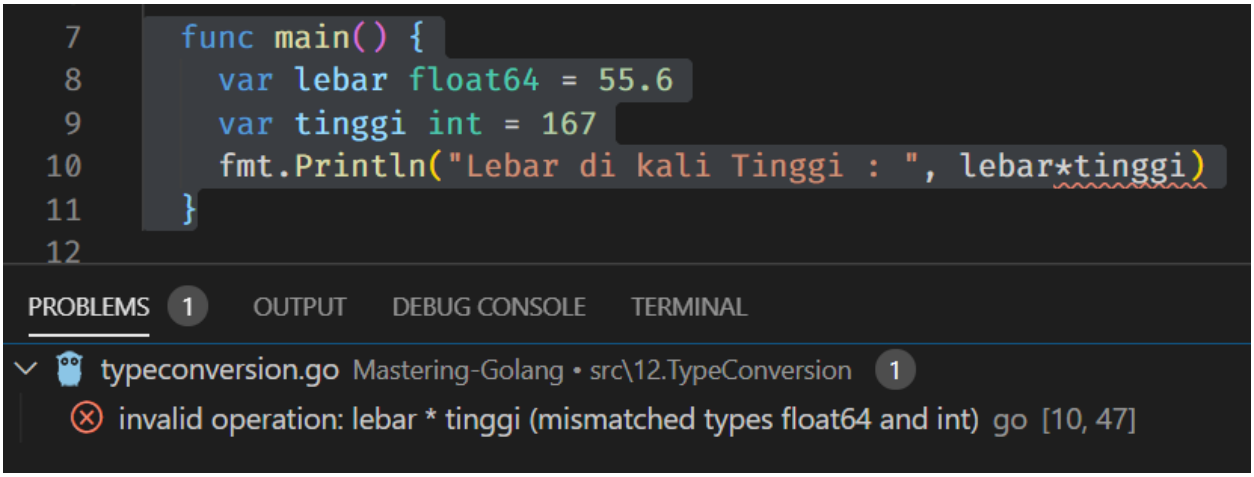
Variabel yang dialokasikan dalam ***stack*** disimpan secara langsung di dalam memori dan akses pada ***static memory*** sangat cepat. Variabel yang dialokasikan dalam ***heap*** memiliki memori yang dialokasikan saat ***run time*** dan akses pada ***dynamic memory*** cenderung lamban.

6. *Data Types Conversion*

Operasi matematika dan operasi untuk membandingkan (***comparison operation***) dalam bahasa pemrograman ***Go*** wajib dengan tipe data yang sama. Perhatikan potongan kode berikut :

```
func main() {
  var lebar float64 = 55.6
  var tinggi int = 167
  fmt.Println("Lebar di kali Tinggi : ", lebar*tinggi)
}
```

Kita melakukan operasi aritmetika berupa perkalian antara dua tipe data yang berbeda, **`float64`** dan **`int`**. Hasilnya adalah *error* :



Gambar 69 Types Error

Terdapat pesan ***error*** yang menyatakan ***mismatched types float64 and int***. Maksudnya adalah kita melakukan operasi perkalian antara dua variabel yang tidak dapat dilakukan, disebabkan tipe data yang berbeda.

```
fmt.Println("Lebar > Tinggi ", lebar>tinggi)
```

Begitupun ketika kita melakukan operasi perbandingan antara variabel yang memiliki tipe data **`float64`** dan **`int`**, hasilnya adalah ***error***.

int To float64

Kita akan belajar melakukan konversi tipe data dari **`int ke float64`**, perhatikan kode di bawah ini :

```
package main

import (
  "fmt"
  "reflect"
)

func main() {
  var usia int = 65
  fmt.Println(float64(usia)) // Output : 65

  fmt.Println(reflect.TypeOf(float64(usia)))
  // Output : float64
}
```

Pada kode di atas kita menggunakan ***function*** **float64** untuk melakukan konversi.

float64 To int

Untuk konversi tipe data dari **float64** ke **int** perhatikan potongan kode di bawah ini :

```
  var berat float64 = 4.8
  fmt.Println(int(berat)) // Output : 4 (0.8 removed)
```

```
  fmt.Println(reflect.TypeOf(int(berat))) // Output : int
```

Pada kode di atas kita menggunakan ***function*** **int** untuk melakukan konversi.

Int To String

Untuk melakukan konversi tipe data dari ***integer*** ke ***string***, kita perlu menggunakan ***package*** **strconv**. ***Import package*** tersebut ke dalam program :

```
import (
  "fmt"
  "strconv"
)
```

Setelah itu kita dapat menggunakan ***function*** **Itoa** dalam ***package package*** **strconv** :

```
func main() {
  s := strconv.Itoa(6666) // "6666"
  fmt.Println(s)
}
```

Konversi ***integer*** ke ***string*** berhasil.

String to Int

Untuk melakukan konversi tipe data dari ***string*** ke ***integer***, kita masih perlu menggunakan ***package*** **strconv**. ***Import package*** tersebut ke dalam program :

```
func main() {
  s := "33.3"
  if n, err := strconv.Atoi(s); err == nil {
    fmt.Println(n + 1)
```

```
  } else {
    fmt.Println(err)
  }
}
```

Pada kode di atas kita menggunakan ***function*** **Atoi** dalam ***package*** **strconv**.

String to Float

Untuk melakukan konversi tipe data dari ***string*** ke ***float***, kita masih perlu menggunakan ***package*** **strconv**. ***Import package*** tersebut ke dalam program :

```
func main() {
  f := "3.14159265"
  if s, err := strconv.ParseFloat(f, 32); err == nil {
      fmt.Println(s) // 3.1415927410125732
  }
  if s, err := strconv.ParseFloat(f, 64); err == nil {
      fmt.Println(s) // 3.14159265
  }
}
```

Pada kode di atas kita menggunakan ***function*** **ParseFloat** dalam ***package*** **strconv**. ***Parameter*** pertama tempat untuk ***input string***, kedua untuk menentukan ***precision*** yang dispesifikasi berdasarkan **bitSize**.

Arti angka 32 dan 64 pada ***parameter*** kedua adalah **float32** dan **float64**.

Int to Int64

Untuk konversi tipe data dari **int** ke **float64** perhatikan potongan kode di bawah ini :

```
func main() {
```

```
    var n int = 88
    m := int64(n)
    fmt.Println(reflect.TypeOf(m)) // int64
}
```

Pada kode di atas kita menggunakan ***function*** **int64** untuk melakukan konversi.

Subchapter 5 – Control Flow ✓

The most important property of a program is
Whether it accomplishes the intention of its user.

— C.A.R Hoare

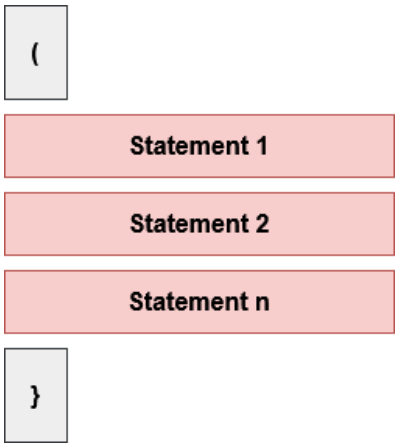
Subchapter 5 – Objectives

- Mempelajari Apa itu ***Block Statements***?
- Mempelajari Apa itu ***Conditional Statements***?
- Mempelajari Apa itu ***Multiconditional Statement***?
- Mempelajari Apa itu ***Switch Style***?

Control Flow menjelaskan mana urutan ***statements*** yang akan dieksekusi. Sehingga ***Application*** yang dibuat dengan ***Go*** menjadi lebih interaktif. Ada beberapa hal yang akan kita bahas disini di antaranya adalah :

1. *Block Statements*

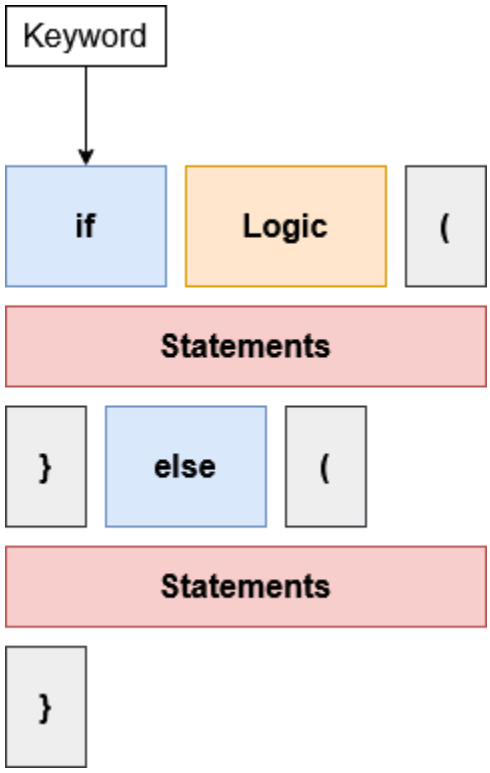
Block statements atau terkadang disebut ***compound statement*** adalah sekumpulan ***statements*** yang akan dieksekusi secara berurutan di dalam sebuah kurung kurawal buka dan kurung kurawal tutup. ({ … }). Perhatikan contoh ***syntax*** di bawah ini :



Gambar 70 Block Statement

2. *Conditional Statements*

Conditional statements adalah sekumpulan perintah yang akan dieksekusi jika kondisi ***Logic*** bernilai ***true***. Pada diagram di bawah ini terdapat dua ***block statements***, pertama pada sekup ***keyword*** **if** dan yang kedua berada pada sekup ***keyword*** **else :**



Gambar 71 Conditional Statement

Kita akan mengubah diagram di atas ke dalam contoh penggunaan ***conditional statement*** dalam bahasa pemrograman ***Go*** :

```
package main

import (
  "fmt"
)
```

```go
func main() {

  input := 7
  if input%2 == 0 {
    fmt.Println("Even")
  } else {
    fmt.Println("Odd")
  }
}
```

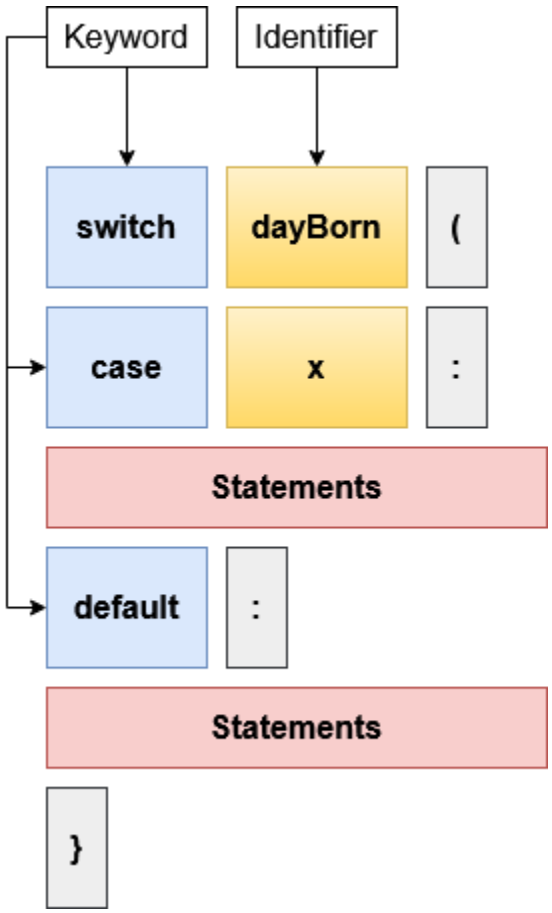
3. *Multiconditional Statement*

Untuk menghadapi ***multiple condition*** kita bisa menggunakan ***If***.. ***Else If*** dan ***Else***, ***block statement*** ke ***n*** akan dieksekusi jika kondisinya memenuhi syarat. Perhatikan contoh ***syntax*** di bawah ini :

```go
package main

import (
  "fmt"
)

func main() {

  input := 7
  if input < 0 {
    fmt.Println("Input must be positive integer")
  } else if input%2 == 0 {
    fmt.Println("Even")
  } else {
    fmt.Println("Odd")
  }
}
```

4. *Switch Style*

Selanjutnya kita akan mempelajari ***Switch Statements*** yang akan mengeksekusi ***statements*** berdasarkan ***label*** yang sama.



Gambar 72 Switch Syntax

Berdasarkan diagram di atas kita akan mencoba membuat sebuah ***Switch Statements*** dalam bahasa pemrogrmana ***Go*** :

```
package main

import (
  "fmt"
  "time"
```

```
)

func main() {
  dayBorn := time.Monday

  switch dayBorn {
  case time.Monday:
    fmt.Println("This is Monday!")
  case time.Tuesday:
    fmt.Println("This is Tuesday")
  case time.Wednesday:
    fmt.Println("This is Wednesday")
  case time.Thursday:
    fmt.Println("This is Thursday")
  case time.Friday:
    fmt.Println("This is Friday")
  default:
    fmt.Println("Day, is not valid! Error!")
  }
}
```

Subchapter 6 – Loop & Iteration ✓

Programmers are not to be measured by their ingenuity and their logic
but by the completeness of their case analysis.

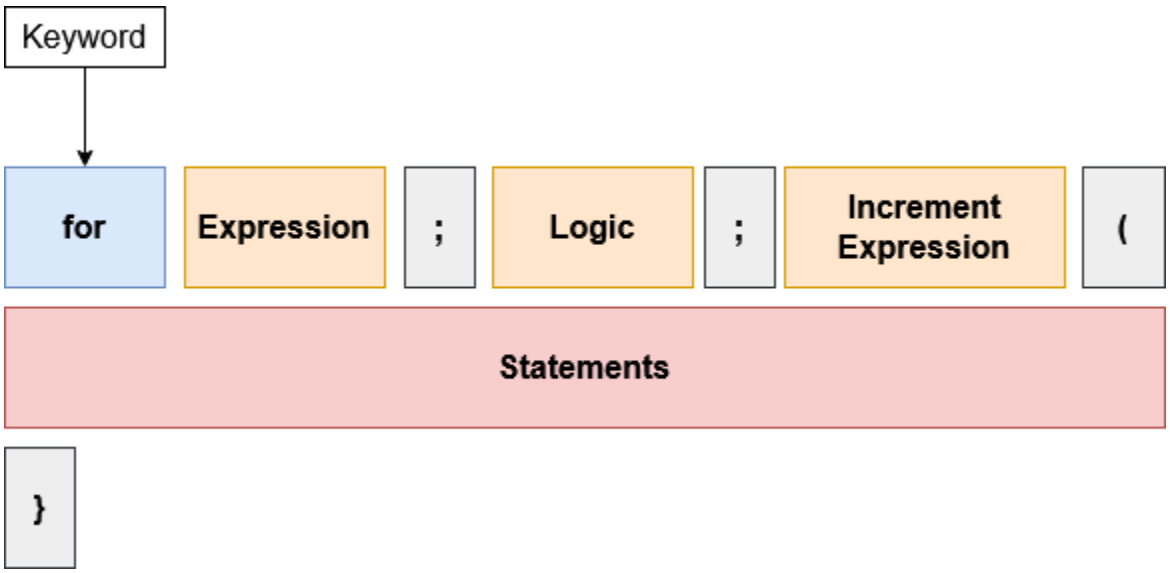
— Alan J. Perlis

Subchapter 6 – Objectives

- Mempelajari Apa itu ***For Statements***?
- Mempelajari Apa itu ***For..Range Statement?***
- Mempelajari Apa itu ***Break Statement***?
- Mempelajari Apa itu ***Continue Statement***?

Ada saatnya kita ingin mengulang kode yang sama untuk dieksekusi berkali kali, pada ***Go*** kita bisa menggunakan beberapa cara diantaranya adalah :

1. *For Statement*



Gambar 73 For Syntax

Di atas adalah contoh ***syntax for statement***.

Pada *for statement*, perulangan akan terus terjadi sampai hasil evaluasi ***condition* mendapatkan nilai *false*.** Untuk melakukan perulangan kita harus mengatur ***initialExpression*** terlebih dahulu, sebuah nilai awal untuk melakukan perulangan.

Selama ***condition*** bernilai *true* maka ***statement*** di dalam ***block for statement*** akan terus dieksekusi. Setiapkali ***statement*** di dalam ***block for statement*** dieksekusi ***incrementExpression*** akan terus meningkat.

Contoh penggunaan ***For Statements*** bisa kita lihat pada gambar di bawah ini :

```
func main() {
  for index := 0; index < 5; index++ {
    fmt.Println(index)
```

```
  }
}
```

Keluaran kode di atas :

```
/* Output :
Perulangan ke 10
Perulangan ke 9
Perulangan ke 8
Perulangan ke 7
Perulangan ke 6
*/
```

Catatan, bukan hanya ***incrementExpression*** yang akan terus bertambah satu setiap kali perulangan dilakukan bisa juga ***decrementExpression*** yang akan terus berkurang satu setiap kali perulangan dilakukan.

```
func main() {
  for index := 10; index > 5; index-- {
    fmt.Println(index)
  }
}
```

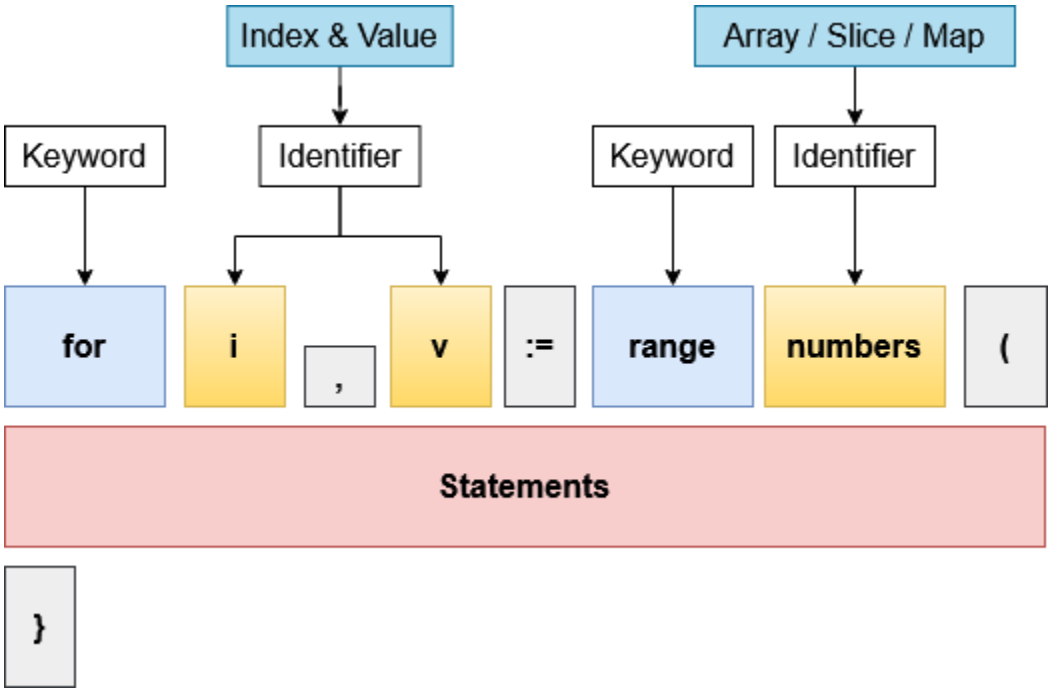
Keluaran dari kode di atas :

```
/* Output :
Perulangan ke 10
Perulangan ke 9
Perulangan ke 8
Perulangan ke 7
```

```
Perulangan ke 6
*/
```

2. *Range Statement*

Di bawah ini adalah syntax diagram dari for..range dalam bahasa pemrograman Go.



Gambar 74 For-Range Structure

Kita akan mengubah diagram di atas ke dalam bahasa pemrograman ***Go***. Contoh penggunaan ***For .. Range Statements*** bisa kita lihat pada gambar di bawah ini :

```
func main() {
 numbers := [9]int{0,1,2,3,4,5,6,7,8}

 for i, v := range numbers {
    fmt.Println("Array Element",i,"is",v)
 }
}
```

Pada kode di atas terdapat sebuah ***array integer*** yang memiliki 9 ***element***. Jika kita ingin mendapatkan nilai dari setiap ***element*** di dalamnya kita dapat menggunakan **for..range** ***statement*** seperti di atas.

Keluaran dari kode di atas :

```
/*
Array Element 0 is 0
Array Element 1 is 1
Array Element 2 is 2
Array Element 3 is 3
Array Element 4 is 4
Array Element 5 is 5
Array Element 6 is 6
Array Element 7 is 7
Array Element 8 is 8
*/
```

Anda akan mempelajari ***array*** lebih detail di ***Subchapter*** 9 tentang ***array***.

3. *Break Statement*

Pada perulangan, ***break statement*** digunakan untuk keluar dari suatu perulangan.

```
func main() {
  for index := 0; index < 5; index++ {
    if index == 3 {
      break
    }
    fmt.Println(index)
  }
}
```

Keluaran dari kode di atas :

```
/* Output :
Perulangan ke 0
Perulangan ke 1
Perulangan ke 2
*/
```

4. *Continue Statement*

Penggunaan ***continue*** dilakukan jika kita ingin menghentikan eksekusi suatu ***statement*** dalam suatu ***iteration,*** namun tetap melanjutkan kembali perulangan dengan ***iteration*** selanjutnya sampai selesai.

```
func main() {
  for index := 0; index < 5; index++ {
    if index == 3 {
      continue
    }
    fmt.Println(index)
  }
}
```

Keluaran dari kode di atas :

```
// Output
// Perulangan Ke 0
// Perulangan Ke 1
// Perulangan Ke 2
// Perulangan Ke 4
// Perulangan Ke 5
```

Subchapter 7 – Function ✓

Learning to code is learning to create and innovate.
—Enda Kenny

Subchapter 7 – Objectives

- Mempelajari Konsep ***Function*** pada ***Go***.
- Mempelajari Konsep ***First-class Function***, ***Citizen*** & ***Higher-order Function***.
- Mempelajari ***Function Structure*** pada ***Go***.
- Mempelajari ***Function Parameter*** & Argument pada ***Go***
- Mempelajari ***Function Return*** pada ***Go***.
- Mempelajari ***Variadic Function*** pada ***Go***.
- Mempelajari ***Anonymous Function*** pada ***Go***.
- Mempelajari ***Closure*** Pada ***Go***.
- Mempelajari ***Defer Statement*** pada ***Go***.

1. *Introduction to Function*

Function adalah sebuah *subprogram* yang didesain untuk menyelesaikan suatu pekerjaan. *Function* akan dieksekusi jika telah kita panggil, fenomena memanggil fungsi disebut dengan ***Invoking***.

Kita tahu bahasa sebuah ***function*** selalu menghasilkan sebuah ***return. Function*** pada ***Go*** berbeda dengan ***function*** pada bahasa pemrograman lainnya, sebab Go dapat memberikan ***return*** lebih dari satu dengan berbagai ***types***.

First Class Function

Go adalah bahasa yang mendukung ***First-class Function***, artinya ***Go*** memiliki kemampuan untuk melakukan ***passing*** sebuah ***variable*** ke dalam sebuah ***function***,

melakukan ***passing*** sebuah ***function*** sebagai sebuah ***argument*** dan bisa memiliki ***function*** yang memberikan ***return*** sebuah ***function*** lagi.

First-class Citizen

Go adalah bahasa yang dapat menetapkan ***function*** untuk sebuah ***variable***, sehingga ***function*** dalam ***Go*** mendukung ***First-class Citizen***.

Higher-order Functions

Go adalah bahasa yang mendukung ***Higher-order Functions***, artinya sebuah ***function*** dapat menerima ***function*** sebagai sebuah ***argument***.

Function of Function

Tujuan dari fungsi adalah :

Breaking Complexity

Sebuah ***function*** kita buat untuk menyelesaikan suatu permasalahan, jika terdapat permasalahan rumit yang kita hadapi maka kita perlu memecahkan kerumitan tersebut ke dalam tugas-tugas kecil.

Tugas-tugas kecil tersebut adalah sekumpulan ***function*** yang kita buat untuk memecahkan permasalahan yang rumit.

Reusing Code

Sebuah ***function*** kita buat agar kita tidak membuat kode yang repetitif. Kode yang duplikat meningkatkan kerumitan saat kita akan melakukan ***maintenance code***.

Single Responsibility

Sebuah ***function*** harus melaksanakan satu tugas saja.

Small

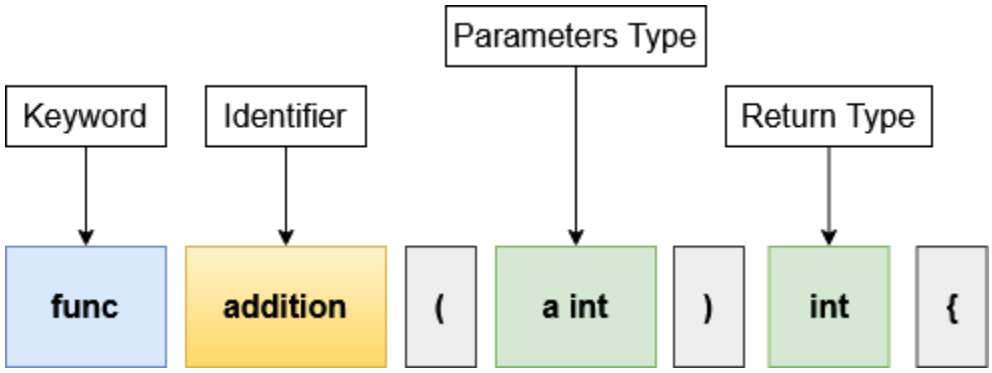
Sebuah ***function*** harus berukuran kecil, jika kita membuat ***function*** yang terdiri dari ratusan baris maka kita telah melanggar konsep ***single responsibility***. Kode yang singkat dan efektif di dalam sebuah ***function*** membantu kita saat proses ***debugging***.

Function Structure

Di bawah ini adalah contoh ***function*** dalam Go :

```
func addition(a, b int) int {
  return a + b
}
```

Jika kita bedah struktur sebuah ***function***, pada umumnya terdapat 4 struktur :



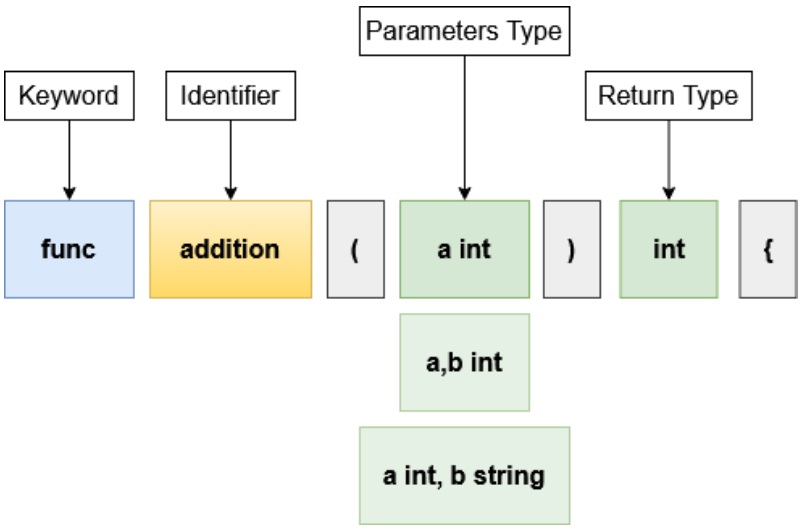
Gambar 75 Function Structure

Sebelumnya anda telah mempelajari terminologi ***keyword*** dan ***identifier*** pada bab tentang ***variable declaration***.

Parameter Type

Parameter type adalah tempat anda akan memberikan ***input*** lengkap dengan tipe datanya untuk sebuah ***function***. Anda dapat memberikan ***input*** lebih dari satu

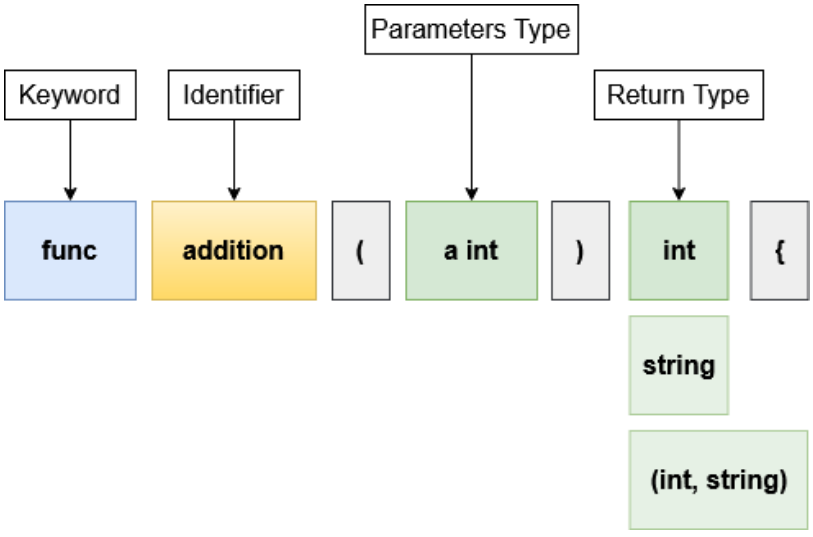
parameter seperti pada gambar di atas, terdapat ***input*** untuk ***parameter*** **a** dan ***parameter*** **b** dengan ***type*** data wajib ***integer***.



Gambar 76 Multiple Parameters

Return Type

Return type adalah ***output*** yang akan diproduksi sebuah ***function***, ***output*** dapat berupa ***integer, string, boolean, map*** atau bahkan ***function*** lainnya.



Gambar 77 Multiple Returns

2. Practice Function

Basic Function

Di bawah ini adalah bentuk paling sederhana untuk membuat fungsi :

```
// Basic Function
func sayHello() {
  fmt.Println("Hello!")
  fmt.Println("Gun Gun Febrianza!")
}
```

Untuk menggunakan fungsi tersebut kita hanya perlu memanggil nama fungsinya atau *identifier* fungsinya lengkap dengan kurung buka dan kurung tutupnya **`sayHello()`** :

```
func main() {
  sayHello()
}
/*
Output
Hello!
Gun Gun Febrianza! */
```

Kita dapat menggunakan fungsi tersebut lagi dan lagi :

```
func main() {
  sayHello()
  sayHello()
}
```

Function Parameter

Di bawah ini adalah sebuah *function* yang memiliki *parameter*, terdapat dua *parameter* yaitu **`from`** dan **`text`** :

```
func displayMessage(from string, text string) {
  fmt.Println(from + ": " + text)
}

func main() {
  displayMessage("Gun Gun Febrianza", "Hello Maudy!")
  displayMessage("Maudy", "Hello Gun!")
}
```

Pada kode di atas kita dapat memanfaatkan ***parameter*** agar bisa memberikan ***output*** yang berbeda. Pada ***function body*** kita membuat sebuah ***statement*** yang membutuhkan ***parameter*** agar dapat dieksekusi.

Function Arguments

Jika parameter ***function*** memiliki tipe data yang sama, pada kasus di bawah ini adalah **`string`** kita bisa mempersingkatnya :

```
func displayMessage(from, text string) {
  fmt.Println(from + ": " + text)
}
```

`from` dan **`text`** adalah **parameter** dan **arguments** adalah `("Maudy", "Hello Gun!")`

Function Return

Di bawah ini adalah sebuah *function* yang memiliki sebuah *return* :

```
func addition(a, b int) int {
  return a + b
}
```

Pada kode di atas kita menggunakan ***keyword*** `return` di dalam ***body function***, setiap ***function*** hanya dapat memiliki satu buah ***keyword*** `return`.

```
func main() {
  fmt.Println(addition(5, 6))
}
```

Function Multiple Return

Di bawah ini adalah sebuah *function* yang memiliki sebuah *multiple return* :

```
func addition(a, b int) (string, int) {
  return "Hasilnya adalah ", a + b
}
```

Untuk menggunakan ***function*** tersebut tulis kode di bawah ini :

```
func main() {
  fmt.Println(addition(5, 6))
}
```

Function Named Return

Di bawah ini adalah sebuah *function* yang memiliki sebuah *named return* :

```
func arithmetic(a, b int) (add, sub int) {
  add = a + b
```

```
  sub = a - b
  return
}
```

Untuk menggunakan ***function*** tersebut tulis kode di bawah ini :

```
func main() {
  fmt.Println(arithmetic(5, 6))
}
```

First-class Citizen

Di bawah ini adalah sebuah ***function*** yang diterapkan (***assign***) pada sebuah ***variable***. Inilah yang dimaksud dengan ***First-class Citizen*** dalam ***Go*** :

```
var aritmetika = func(a, b int) (add, sub int) {
  add = a + b
  sub = a - b
  return
}
```

Untuk menggunakan ***variable*** tersebut tulis kode di bawah ini :

```
func main() {
  fmt.Println(aritmetika(5, 6))
}
```

Variadic Function

Go mendukung konsep ***Variadic Function***, sebuah ***function*** yang dapat memiliki ***parameter*** dengan jumlah ***parameter*** yang tidak dispesifikasikan secara eksplisit. Pada contoh kode di bawah ini terdapat ***variadic function*** yang dapat menerima ***parameter*** dengan tipe data ***integer*** :

```
func infiniteParameter(inputs ...int) (sum int) {
  for _, n := range inputs {
    sum += n
    fmt.Println(n)
  }
  return
}
```

Untuk menggunakannya tulis kode di bawah ini :

```
func main() {
  fmt.Println(infiniteParameter(1, 2, 3, 4))
}
```

Anonymous Function

Anonymous function adalah sebuah ***function*** yang tidak memiliki ***identifier***. Sebelumnya setiap kali kita membuat sebuah ***function***, kita selalu memberikan nama pada ***function*** tersebut. ***Anonymous function*** adalah ***function*** yang tidak memiliki nama.

Di bawah ini adalah contoh kode penggunaan ***anonymous function*** :

```
func main() {
  func() {
```

```
    fmt.Println("Hi maudy") //hi maudy
  }()
}
```

Jika kita perhatikan pada kode di atas, di dalam ***entrypoint*** terdapat suatu ***function*** yang tidak memiliki ***identifier***. Selanjutnya ***function*** tersebut memiliki ***syntax*** **()** setelah kurawal tutup, ***syntax*** tersebut disebut dengan ***execution parenthesis***.

Execution Parenthesis

Fungsi dari ***execution parenthesis*** adalah tempat kita untuk memberikan parameter, perhatikan kode di bawah ini untuk memahami fungsi dari ***execution aprenthesis*** :

```
func main() {
  message := "Hi Maudy"
  func(str string) {
    fmt.Println(str)
  }(message)
}
```

Closure

Bahasa pemrograman ***Go*** mendukung ***anonymous function*** yang dapat digunakan untuk melakukan ***closure***. Perhatikan kode di bawah ini :

```
func exampleClosure() func() string {
  message := "Hello"
  return func() string {
    message += " Gun"
    return message
  }
}
```

Function **exampleClosure()**memiliki ***return*** sebuah ***function*** dengan tipe data ***string***.

Di dalam ***body function*** dari **exampleClosure()** terdapat sebuah ***anonymous function*** yang berinteraksi dengan ***variable*** **message**, ***anonymous function*** tersebut akan menjadi ***return*** untuk **exampleClosure()**.

Untuk melihat cara kerjanya eksekusi kode di bawah ini :

```
func main() {

  message := exampleClosure()

  fmt.Println(message())
  fmt.Println(message())
  fmt.Println(message())

  newMessage := exampleClosure()
  fmt.Println(newMessage())
}

/*
Hello Gun
Hello Gun Gun
Hello Gun Gun Gun
Hello Gun
*/
```

Dari hasil yang diproduksi ***closure*** anda akan memahami cara kerjanya,

Defer

Defer statement digunakan untuk menunda eksekusi sebuah ***function*** sebelum ***function*** tempat ***defer*** dieksekusi berhenti. Perhatikan kode di bawah ini :

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

/*
Hello
World
*/
```

Pada kode di atas ***keyword*** **defer** digunakan untuk membuat ***defer statement***. Terdapat ***statement*** **fmt.Println("World")** yang di ***defer***, ***statement*** tersebut akan dieksekusi saat ***main function*** berhenti.

Secara praktek ***defer*** biasanya digunakan untuk membersihkan hasil dari suatu aktivitas. Misal kita sedang berinteraksi dengan ***filesystem*** atau ***database***, sebelum kita selesai mengeksekusi suatu operasi kita dapat menghentikan operasi ***read/write*** pada ***filesystem*** atau menutup koneksi ***database***.

Defer Order of Execution

Jika kita mencoba mengeksekusi suatu ***statement*** menggunakan ***defer*** dalam perulangan, kita akan faham bagaimana cara kerja urutan eksekusi (***order of execution***) dari ***defer***.

Perhatikan kode di bawah ini :

```go
func main() {
  fmt.Println("Hello")
  for i := 1; i <= 3; i++ {
    defer fmt.Println(i)
  }
  fmt.Println("World")
}

/*
  Hello
  World
  3
  2
  1
*/
```

Angka 1 dieksekusi terakhir sementara angka 3 dieksekusi terakhir, ini dikarenakan cara kerja ***defer*** secara internal adalah ***first in last out***.

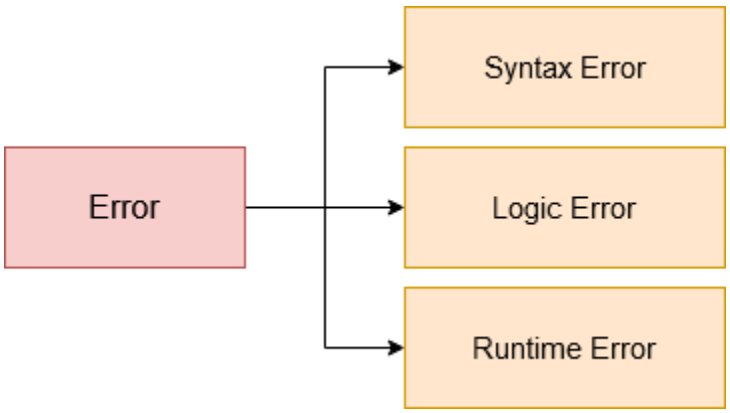
Subchapter 8 – Error Handling ✓

The best error message is the one that never shows up.

—Thomas Fuchs

Subchapter 8 – Objectives

- Mempelajari Konsep ***Syntax Error***.
- Mempelajari Konsep ***Logical Error***.
- Mempelajari Konsep ***Runtime Error***.
- Mempelajari ***log Package*** pada Go.
- Mempelajari Konsep ***Panic*** pada ***Go.***
- Mempelajari Konsep ***Recover*** pada ***Go***.



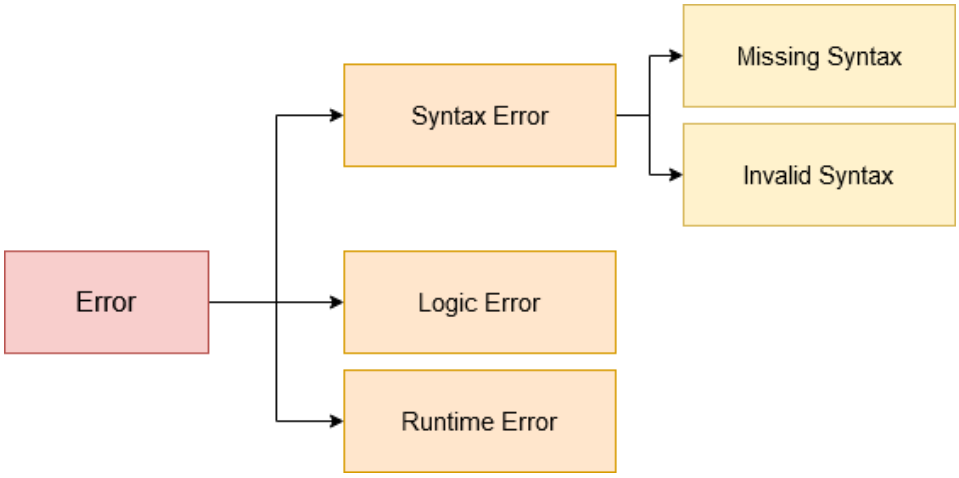
Gambar 78 Errors Type

Sebuah program memiliki tiga kemungkinan ***errors*** yaitu :

1. ***Syntax error,***
2. ***Logical error*** dan
3. ***Runtime error***.

Di halaman selanjutnya kita akan mengkaji ketiga konsep ***error*** tersebut.

1. *Syntax Error*



Gambar 79 Syntax Error

Syntax Error adalah kesalahan yang paling sering terjadi. Kesalahan ini dapat dideteksi oleh **interpreter**, karena **interpreter** memiliki sebuah program internal yang disebut dengan **syntax analyzer**.

Faktor yang paling mempengaruhinya adalah kesalahan *typing* dan kesalahan aturan penulisan bahasa pemrograman yang disebut dengan ***syntax rule***.

Missing Syntax

Sebagai contoh kode *Go* di bawah ini adalah susunan *declaration statement* yang benar :

```
var x int = 10
```

Apa yang terjadi jika kita menghapus *identifier* **x** pada kode di atas? Maka akan muncul error dengan pesan sebagai berikut :

int declared but not used

Invalid Syntax

Invalid syntax terjadi ketika kita menambahkan sebuah *syntax* yang susunanya tidak sesuai dengan *syntax rule* yang dimiliki oleh *Go*.

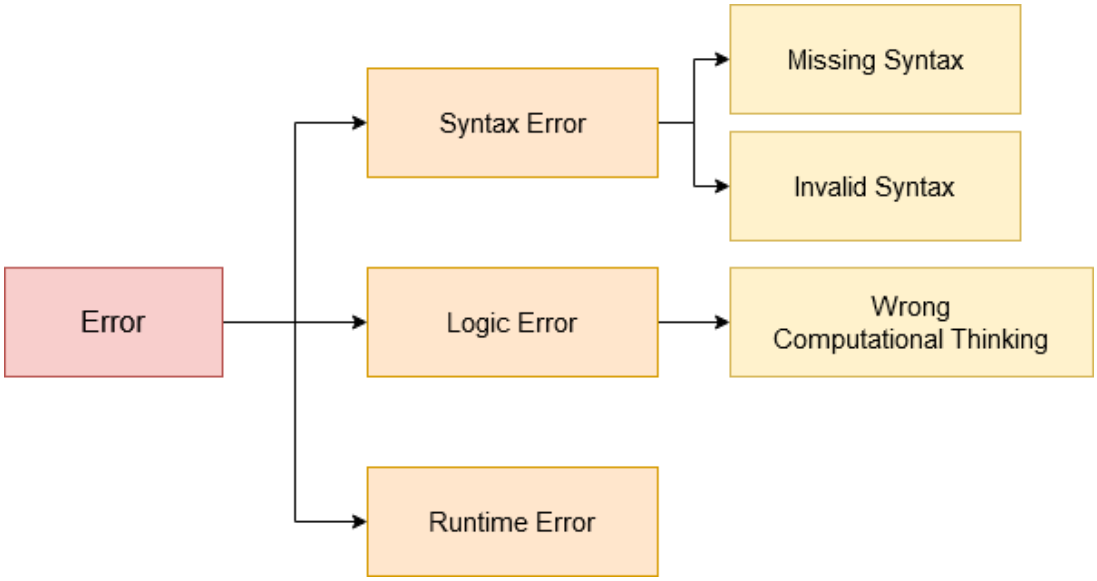
Pada kode di bawah ini kita menambahkan lagi *keyword* **`var`** setelah *literal* **`10`**, akibatnya menimbulkan *syntax error*.

```
var x = 10var
```

Di bawah ini adalah pesan ***error*** yang muncul saat kompilasi :

syntax error: unexpected var at end of statement

2. *Logical Error*



Gambar 80 Logic Error

Logical Error adalah kesalahan yang terjadi pada logika kode pemrograman yang ditulis oleh seorang *programmer*. Kesalahan ini tidak bisa dideteksi oleh kompiler, kesalahan ini hanya bisa diketahui ketika seorang *programmer* mulai melakukan evaluasi lagi pada kode pemrograman yang dibuatnya.

Kita coba dengan *study case* yang sederhana :

```
var x int = 10;
var xx int = 20
result := 10 * x;
fmt.Println(result);
```

Kita asumsikan anda telah menulis kode hingga ratusan atau ribuan baris, kemudian anda berpikir bahwa seharusnya nilai dari variabel **`result`** adalah **`200`**.

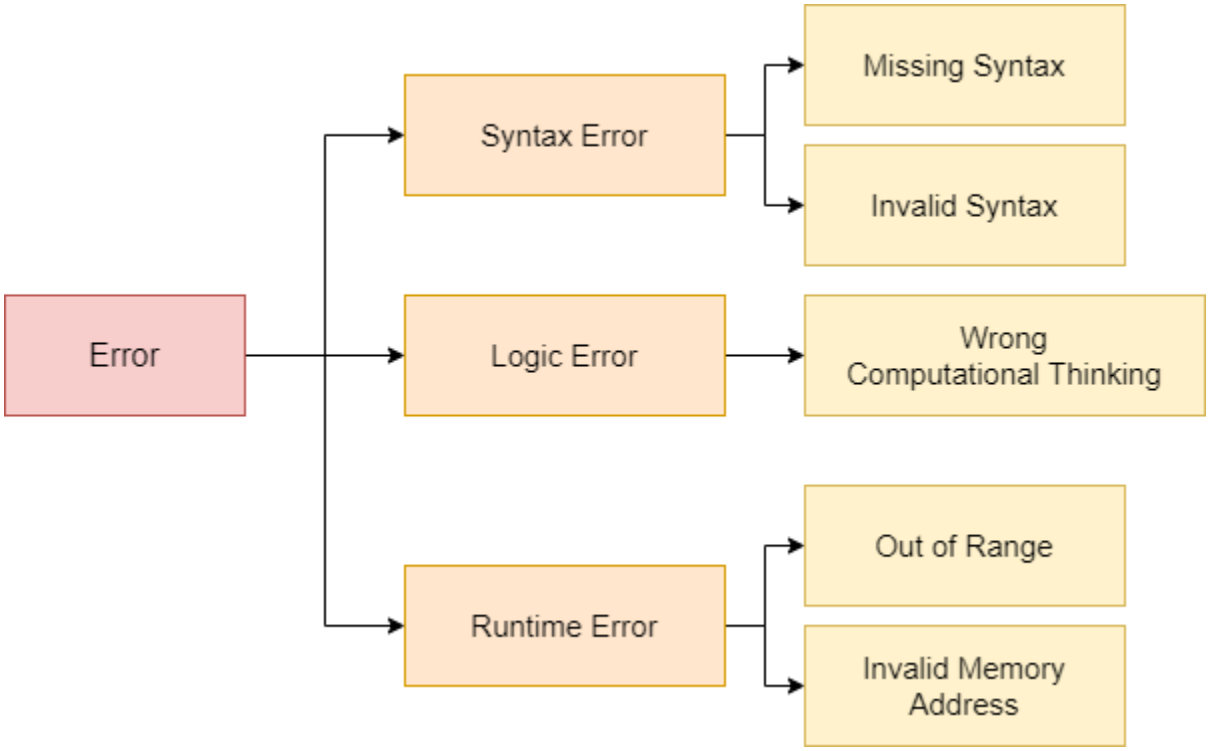
Karena anda yakin nilai **`x`** adalah **`20`**, padahal nilai **`x`** bukanlah **`20`** melainkan **`10`**, namun anda tidak menyadarinya sama sekali.

Pada kasus di atas terdapat dua kemungkinan *logic error* :

1. Kita salah memberikan **variable**, seharusnya **variable `xx`** agar hasilnya adalah **`200`**.
2. Jika nilai **`x`** adalah hasil dari suatu proses komputasi tertentu dan hasilnya tidak sesuai dengan perhitungan anda, maka terdapat kemungkinan kesalahan rumus komputasi. Terdapat kemungkinan kesalahan dalam cara berpikir kita untuk menghitung (***human error***).

Dalam pemrograman kesalahan seperti ini bisa menjadi sangat sulit untuk di deteksi, *perhaps you will blamming your computer*. Lol

3. *Runtime Error*



Gambar 81 Runtime Error

Runtime Error adalah kesalahan yang dapat terjadi saat sebuah program sedang dalam keadaan dieksekusi. Ada ***runtime error*** yang sering terjadi saat kita menulis program menggunakan ***Go*** yaitu ***out of range*** dan ***invalid memory address***.

Error ini hanya dapat terjadi saat ***runtime***, ***Go*** tidak seperti bahasa pemograman lainnya yang memiliki ***exception handling***. Sebuah mekanisme yang dapat membantu kita untuk melakukan ***throw exception***.

4. *Error Package*

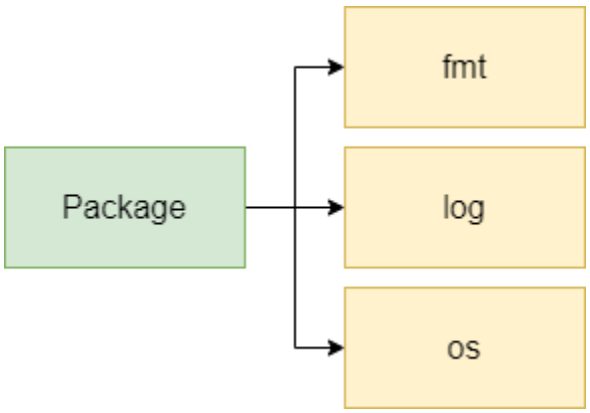
Go tidak memiliki mekanisme ***try & catch*** untuk ***exception handling*** seperti di dalam ***Java***, C***#,*** dan sebagainya. Namun ***Go*** memiliki mekanisme **defer-panic-recover** mekanisme yang akan kita pelajari dihalaman berikutnya.

Untuk melakukan ***handling error*** dalam ***Go***, sebuah ***function*** dapat memberikan ***return*** berupa ***nil value*** (tidak terjadi ***error***) atau ***error*** (terjadi ***error***). Kenapa ***Go*** tidak mendukung ***error handling*** berbasis ***try, catch & finnaly***?

Alasannya adalah ***simplicity***, ***error handling*** berbasis ***try, catch & finnaly idiom*** dianggap rumit di dalam ***Go***. Bahasa pemrograman ***Go*** membawa pendekatan baru dalam menangani ***error*** agar produktivitas ***programmer*** bisa lebih baik.

Log Package

Kita akan membuat sebuah ***program*** yang berinteraksi dengan ***filesystem***. Untuk itu kita harus membuat program yang akan melakukan import 3 ***library*** di bawah ini :



Gambar 82 Log Package

Untuk melakukannya tulis kode di bawah ini :

```
import (
  "fmt"
```

```
  "log"
  "os"
)
```

log adalah ***package*** yang menyediakan manajemen ***logging*** untuk program yang kita buat, kita dapat menulis standar pesan ***error*** yang dapat menampilkan tanggal dan waktu terjadi ***error***.

os adalah ***package*** yang menyediakan layanan agar kita bisa berinteraksi dengan fungsionalitas dalam sistem operasi yang kita gunakan. Perhatikan kode di bawah ini :

```
var fileName = "file.txt"

func openFile() {
  _, err := os.Open(fileName)
  if err != nil {
    log.Println("Error: ", err)
  }
}

func main() {
  openFile()
  fmt.Println("Back to main function after error")
}
```

Pada kode di atas kita membuat ***function*** **openFile()** yang akan kita gunakan untuk membaca suatu ***file*** yaitu **fileName**, ***file*** yang sebenarnya tidak ada. Selanjutnya kita mencoba mengeksekusinya di dalam main ***function*** agar terjadi ***error***.

```
2020/04/14 09:15:58 Error:  open file.txt: The system cannot find the file specified.
Back to main function after error
```

Pada log di atas terdapat informasi tanggal dan waktu terjadinya ***error*** dan pesan ***error***, selanjutnya terdapat pesan ***back to main function after error***. Ini adalah konsep bagaimana kita melakukan ***handling error*** di dalam ***Go***.

Fatal & Exit

Pada kode di bawah ini kita mengubah **log.Println** menjadi **log.Fatalln** agar jika terjadi ***error*** program harus berhenti saat itu juga :

```
func openFile() {
  _, err := os.Open(fileName)
  if err != nil {
    log.Fatalln("Error: ", err)
  }
}
```

Keluaran dari hasil eksekusi kode di atas :

```
2020/04/14 09:15:58 Error:  open file.txt: The system cannot find
the file specified.

exit status 1
```

Pesan ***error*** di atas menyatakan bahwa ***program*** mengalami exit atau berhenti.

5. *Panic & Recover*

Saat kita menulis kode menggunakan ***Go***, seluruh ***error*** akan terdeteksi saat kita melakukan kompilasi (***compile time***). Namun ***error*** juga dapat terjadi saat ***runtime*** yaitu saat program yang telah kita kompilasi di eksekusi.

Saat sebuah program yang telah dikompilasi dieksekusi, jika terdapat ***error*** maka program akan mengalami ***panic*** dan menghentikan seluruh eksekusi. ***Panic*** adalah ***built-in function*** yang telah disediakan di dalam ***Go*** untuk menghentikan program.

Ketika suatu ***function*** memanggil **panic** ***function***, maka eksekusi ***function*** tersebut akan dipaksa berhenti. Pada kode di bawah ini kita mengeksekusi sebuah **panic** ***function***, dan memanggilnya di dalam main ***function*** :

```
function testPanic() {
  panic("Panic Happened!")
}

func main() {
  fmt.Println("Maudy");
  testPanic()
  fmt.Println("Ayunda");
}
```

Jika kode di atas kita eksekusi maka akan menghasilkan keluaran seperti di bawah ini :

```
Maudy
panic: Panic Happened!
goroutine 1 [running]:
main.testPanic(...)
        E:/~/Panic.go:8
```

```
main.main()
        E:/~/ Panic.go:13 +0x9d
exit status 2
```

Untuk mengatasi hal ini kita dapat menggunakan **defer** ***function*** untuk melakukan **recover**, perhatikan kode di bawah ini :

```
func testPanic() {
  defer func() {
    if err := recover(); err != nil {
      fmt.Println(err); //Panic Happened!
      fmt.Println("Recover")
    }
  }()
  panic("Panic Happened!")
}
```

Pada kode di atas di dalam percabangan kita mendeklarasikan **recover**, sehingga ***function*** **main** terus dieksekusi sampai selesai. Di bawah ini adalah keluaran dari program di atas setelah kita menambahkan **defer** & **recover** :

```
/*
Maudy
Panic Happened!
Recover
Ayunda
*/
```

Subchapter 9 – Composite Types ✓

Good code is its own best documentation.

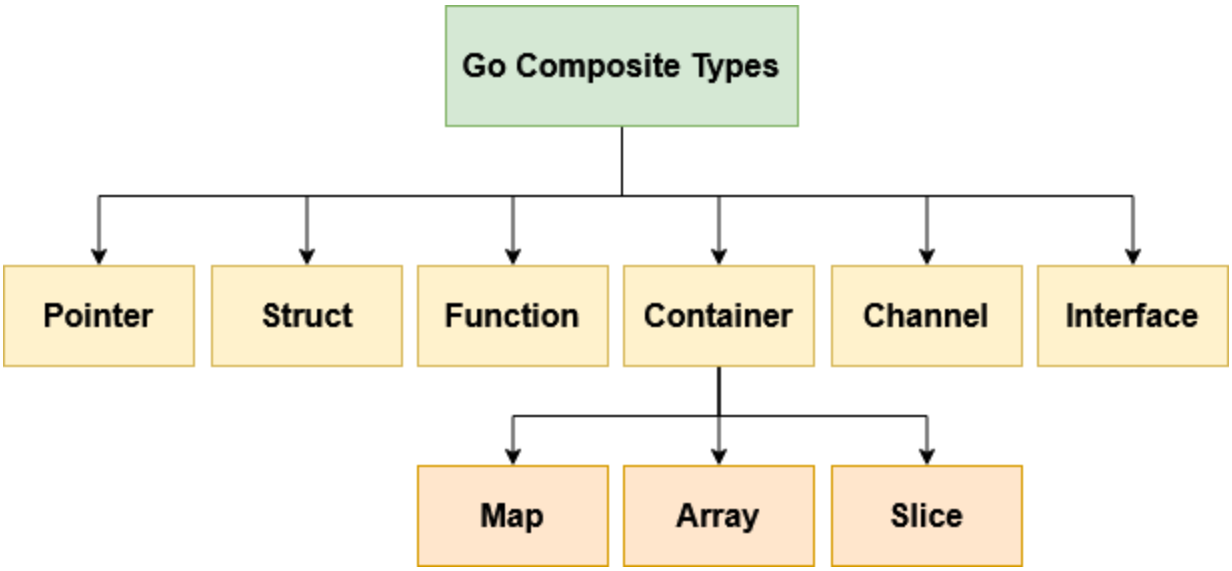
—Steve Mconnel

Subchapter 9 – Objectives

- Mempelajari ***Pointer*** pada ***Go***.
- Mempelajari Konsep ***Passing By Value*** pada ***Go***.
- Mempelajari Konsep ***Passing By Pointer*** pada ***Go***.
- Mempelajari Konsep ***Nil Value*** pada ***Go***.
- Mempelajari ***Struct*** pada ***Go***.
- Mempelajari Konsep ***Struct As Parameter*** pada ***Go.***
- Mempelajari Konsep ***Struct As Pointer*** pada ***Go.***
- Mempelajari Konsep ***Nested Struct*** pada ***Go.***
- Mempelajari Konsep ***Struct Method*** pada ***Go.***

Sebelumnya pada ***Subchapter*** 3 kita mempelajari ***literal*** yang ada pada ***Go*** dan pada ***Subchapter*** 4 – ***Data Types*** kita mempelajari tipe data dasar dalam ***Go***. Sekarang kita akan mengenal bahwa ***Go*** juga memiliki ***Composite Types***.

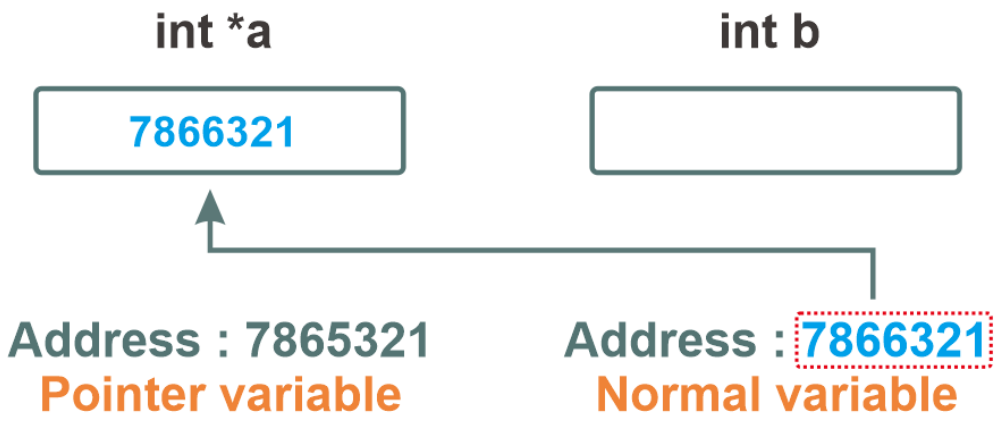
Pada ***composite types*** kita dapat menyimpan data yang lebih rumit dengan karakteristik tertentu, tidak seperti pada ***primitive types*** yang hanya bisa menyimpan satu ***literal*** saja. Sebelumnya kita telah belajar menggunakan ***composite*** saat membuat sebuah ***function***.



Gambar 83 Composite Types

1. Apa itu *Pointer*?

Seperti yang telah kita pelajari sebelumnya variabel adalah tempat untuk menyimpan *data*. Setiap variabel memiliki representasi berupa alamat memori (***memory address***).



Gambar 84 Pointer Example

De-referencing

Untuk mendapatkan ***value*** dari suatu ***memory address***, sebuah program harus membaca ***memory address*** tersebut. Konsep ini dikenal dengan istilah "***De-referencing***"

Read Memory Address

Di bawah ini adalah contoh kode untuk mengetahui ***memory address*** dari suatu ***variable***, kita perlu menggunakan simbol & sebagai ***prefix*** dari ***variable*** untuk mendapatkannya :

```
func main() {
  var person string = "Maudy Ayunda Faza"
  fmt.Printf("Address of a variable: %x\n", &person)
}
```

Hasil dari kode di atas adalah ***memori address*** dari ***variable*** **person** :

```
```

```
// Address of a variable: c00003a1f0
```

Pointer Variable

Pointer adalah sebuah variabel yang nilainya adalah alamat memori (***memory address***) suatu variabel. Untuk membuat ***pointer variable*** kita hanya perlu menambahkan simbol * sebagai ***prefix*** pada tipe data dari ***variable*** :

```
var student *string
```

Student adalah ***pointer variable*** yang dapat digunakan untuk menyimpan ***memory address*** suatu *variable* dengan tipe data ***string***.

Store Memory Address

Sebelumnya kita menggunakan simbol & untuk mendapatkan ***memory address*** dari suatu ***variable***. Untuk menyimpan ***memory address variable*** **person** ke dalam ***pointer*** **student** tulis kode di bawah ini :

```
student = &person
```

Access Pointer Variable

Untuk membaca ***memory address*** yang tersimpan di dalam suatu ***pointer variable***, tulis kode di bawah ini :

```
fmt.Printf("Stored Memory Address: %x\n", student)
```

Untuk membaca ***value*** menggunakan ***pointer*** tulis kode di bawah ini :

```

```

```
fmt.Printf("Value of *student variable: %d\n", *student)
```

Pointer As Parameter

Okay, kita sudah memahami ***pointer*** secara sederhana, sekarang kita akan mempelajari ***pointer*** di kasus yang sedikit lebih kompleks. Yaitu penggunaan ***pointer*** sebagai sebuah ***parameter*** di dalam ***function*** :

```
package main

import (
  "fmt"
)

func passingByValue(count int) {
  count += 10
  fmt.Println("Passed Value : ", count)
}

func passingByPointer(count *int) {
  //Dereference the value and add 10 to it
  *count += 10
  fmt.Println("Passed Pointer : ", *count)
}

func main() {
  var count int
  passingByValue(count) // passing value
  fmt.Println("Function passingByValue Result:", count)
```

```
  fmt.Println()

  // use & to pass a pointer to the variable
  passingByPointer(&count) // passing value by pointer
  fmt.Println("Function passingByPointer Result:", count)
}

/*
Passed Value :  10
Function passingByValue Result: 0

Passed Pointer :  10
Function passingByPointer Result: 10
*/
```

Pada kode di atas jika kita melakukan ***passing value*** menggunakan ***pointer*** maka efeknya akan mengubah ***value*** original dari ***variable*** yang diberikan kepada ***function***.

Saat melakukan ***passing value*** tanpa menggunakan ***pointer*** maka efeknya perubahan ***value*** dari ***variable*** hanya berlaku di dalam sekup ***function*** itu saja. ***Value*** original yang berada diluar sekup ***function*** tidak akan berubah.

Dalam pemrograman kita dapat melakukan ***passing*** sebuah ***value*** dari suatu ***variable*** ke dalam sebuah ***function***, agar sebuah ***function*** dapat mengolah data dari ***value*** tersebut.

Data hanya berubah dalam sekup ***function*** itu saja, tidak mengubah ***value*** dari ***variable*** aslinya yang berada diluar sekup ***function*** tersebut.

Passing By Value

Go akan menyalin ***value*** dan menggunakannya di dalam sekup terisolasi yang dimiliki oleh suatu ***function***, karena ***Go*** hanya menyalin ***value*** maka setiap perubahan pada ***value*** tersebut hanya akan berlaku di dalam ***function*** saja.

Tidak mengubah ***value*** aslinya yang berada diluar sekup ***function***.

Passing By Pointer

Go memiliki ***memory management system*** yang di sebut dengan ***Stack***, jika kita terus menyalin suatu ***value*** maka ukuran ***memory*** akan membesar apalagi jika ***passing value*** terjadi dari suatu ***function*** ke ***function*** lainya.

Untuk mengatasi permasalahan tersebut, kita dapat menggunakan ***passing by pointer***. Dengan begitu kita tetap dapat menyalin dengan ***memory*** yang lebih ringan (***less memory***).

Pointer membantu kita untuk tidak perlu lagi menyalin ***value***, cukup menggunakan ***value*** dengan *pointer* tunggal saja untuk digunakan di dalam ***function***.

Pointer bekerja seperti penanda lokasi agar ***Go*** tidak menyalin ***value*** saat melakukan ***passing pointer*** pada ***function***, ***Go*** akan mengubah langsung ***value*** dari ***variable*** aslinya melalui ***pointer***.

Nil Value

Selain itu kita juga dapat menggunakan ***pointer*** untuk membedakan ***zero value*** dan ***unset value***. ***Go*** menyediakan ***type*** `nil` sebagai ***special type*** untuk merepresentasikan sesuatu yang belum memiliki ***value*** (***unset value***).

Sebelumnya pada pembahasan *variable declaration* kita telah mempelajari konsep ***zero value***. Pada ***non-pointer variable*** ketidakhadiran nilai akan bernilai ***zero*** atau nol (berdasarkan spesifikasi ***Go***).

Pointer memiliki kemampuan untuk bisa menyediakan *special type* `nil`, agar kita tetap bisa mendapatkan ***value*** dari suatu ***pointer*** meskipun ***pointer*** tersebut belum terasosiasi dengan suatu ***value***.

Untuk membuat ***special type*** **nil** tulis kode di bawah ini :

```
var pointerInteger *int //Output : nil
fmt.Printf("-pointerInteger: %#v\n", pointerInteger)
// Output : pointerInteger: (*int)(nil)
```

Namun ada catatan yang perlu diketahui, jika kita mencoba mengakses ***pointer*** tersebut dengan menggunakan simbol ***** secara langsung maka dapat menimbulkan ***runtime error***.

Kita dapat membuktikannya, silahkan tulis kode di bawah ini :

```
fmt.Printf("-pointerInteger: %#v\n", *pointerInteger)
```

Informasi *error* yang tampil :

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xc0000005 code=0x0 addr=0x0 pc=0x49f62a]
```

Gambar 85 Runtime Error

Untuk mencegahnya kita harus melakukan operasi perbandingan terlebih dahulu. Memastikan apakah ***pointer*** bernilai **nil** atau tidak seperti yang telah kita ketahui sebelumnya.

```
if pointerInteger != nil {
  fmt.Printf("pointerInteger-: %#v\n", *pointerInteger)
  // Output : no output
}
```

Pointer Template String

Sebelum kita memeriksa *value* sebuah *pointer*, kita harus memeriksa *pointer* tersebut tidak bernilai `nil` dengan melakukan operasi perbandingan terlebih dahulu. Seperti pada kode di bawah ini.

```
package main

import (
  "fmt"
)

func main() {
  variableInteger := 5
  // Membuat pointer dengan symbol &
  var dataPointer = &variableInteger
  fmt.Printf("variableInteger: %#v\n", variableInteger)
  // Output : variableInteger: 5
  fmt.Printf("dataPointer: %#v\n", dataPointer)
  // Output : dataPointer: (*int)(0xc0000140b0)

  var pointerValue = dataPointer
  fmt.Println(pointerValue)
  // Output : 0xc0000140b0

  if pointerValue != nil {
    // baca pointer dengan symbol *
    fmt.Printf("pointerValue: %#v\n", *pointerValue)
    // Output : pointerValue: 5
  }
}
```

Kita harus menambahkan *character* `*` pada *variable* untuk membaca nilai dari suatu *pointer*. Untuk menampilkan ***output*** kita bisa menggunakan ***function*** `fmt.Printf` agar bisa menggunakan ***template language***.

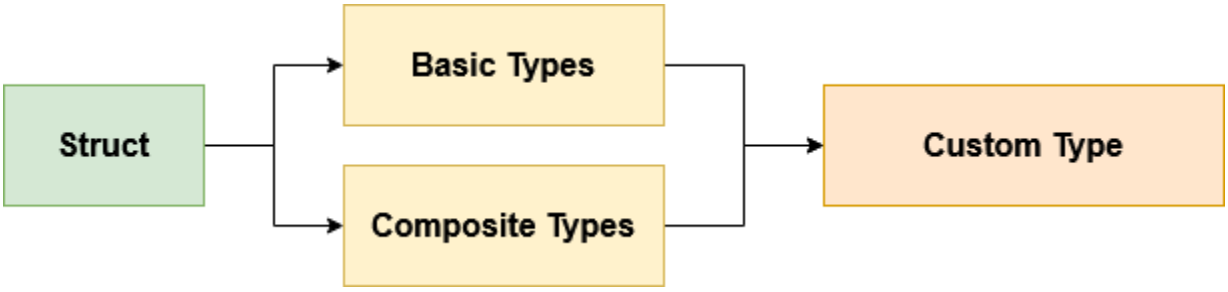
Template language membantu kita untuk bisa mentransformasikan sebuah nilai menggunakan ***Substitution***. Penggunaan ***substitution*** untuk melakukan transformation sangat membantu untuk melihat ***value*** dan ***type*** dari suatu ***variable***.

Di bawah ini beberapa ***substitution*** yang sering digunakan :

Tabel 17 Substitution

Substitution	***Description***
%v	Mencetak nilai saja tanpa menampilkan informasi tipe data.
%#v	Mencetak nilai dari suatu variabel.
%T	Mencetak tipe data dari suatu variabel.
%d	Decimal
%s	String

2. *Struct*

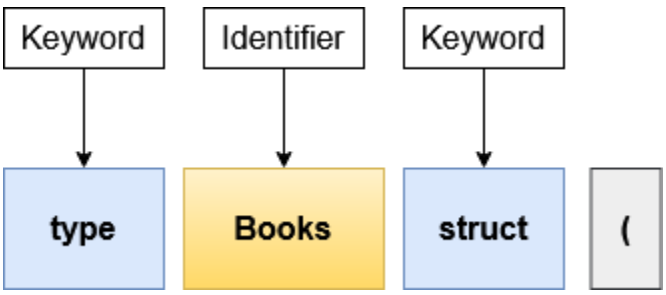


Gambar 86 Struct

Struct atau ***structure*** dalam ***Go*** sering kali disebut dengan ***user-defined type***, artinya kita dapat menggabung beberapa ***basic type*** & ***composite type*** untuk membuat sebuah ***custom type***.

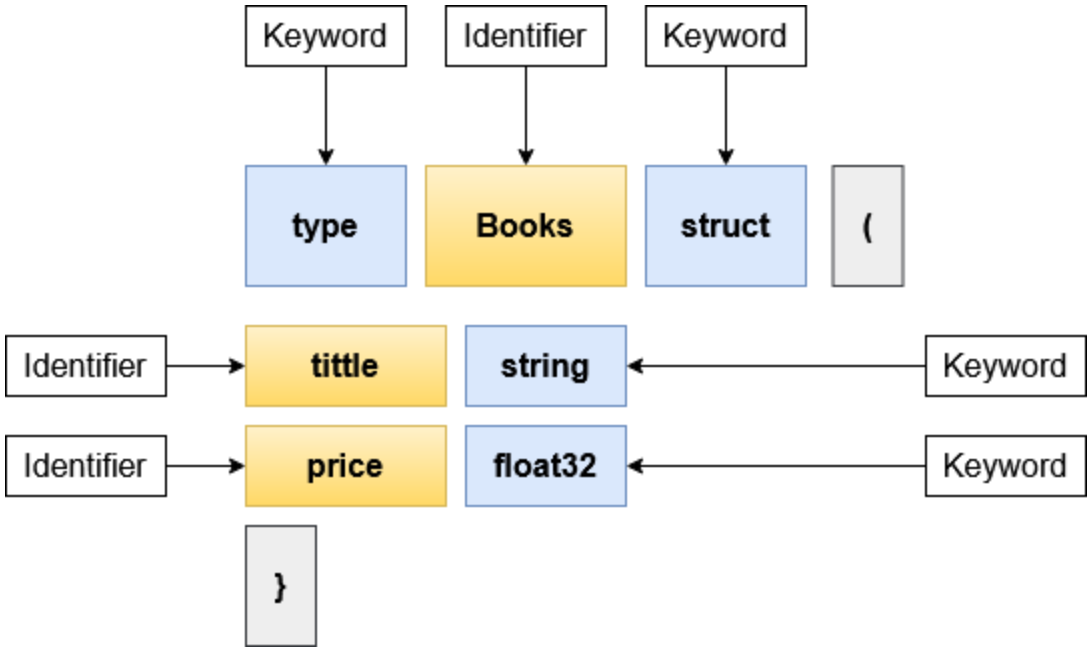
Entitas di dunia nyata dapat di interpretasikan ke dalam ***struct*** yang dapat mengekspresikan ***properties*** atau ***field***, sama seperti pada ***class*** dalam paradigma pemrograman ***object-oriented***. Hanya saja ***struct*** dalam ***Go*** tidak mendukung ***inheritance***.

Jika kita bedah struktur sebuah ***interface***, terdapat 3 struktur :



Gambar 87 Struct Syntax

Sebuah ***struct*** dapat memiliki ***properties*** atau ***field*** :



Gambar 88 Fields

Pada diagram gambar di atas **tittle** dan **price** adalah ***properties*** atau ***fields*** dalam ***struct*** **Books**. Sekarang kita akan mengubah diagram di atas kedalam ***struct*** dalam bahasa pemrograman ***Go***.

Create Struct

Di bawah ini adalah contoh ***struct*** yang merepresentasikan entitas buku :

```
type Books struct {
  bookID int32
  title   string
  author  string
  price float32
}

func main() {

}
```

Declare Custom Type

Setelah ***struct*** di buat kita dapat menggunakan ***custom type*** **books** dengan menulis kode di bawah ini di dalam **main** ***function*** :

```
var ComputerScience Books
```

Selanjutnya kita dapat mengisi ***variable*** **ComputerScience** dengan ***properties*** sesuai dengan **Books** ***struct*** :

```
ComputerScience.bookID = 1
ComputerScience.title = "Belajar Dengan Jenius Javascript"
ComputerScience.author = "Gun Gun Febrianza"
ComputerScience.price = 88000.100
```

Selain menggunakan cara di atas kita juga dapat mendeklarasikan variable dengan cara sebagai berikut :

```
var exampleBook = Books{1, "Go Language", "Gun", 90000}
fmt.Println(exampleBook) // {1 Go Language Gun 90000}
```

Read Struct Field

Untuk membaca ***field*** pada **Books** ***struct***, tulis kode di bawah ini :

```
fmt.Printf("Book ID : %#v\n", ComputerScience.bookID)
fmt.Printf("Title : %#v\n", ComputerScience.title)
fmt.Printf("Author : %#v\n", ComputerScience.author)
fmt.Printf("Price : %#v\n", ComputerScience.price)
```

Struct As Parameter

Kita dapat menggunakan ***struct*** sebagai ***parameter*** sebuah ***function***. Buatlah sebuah ***function*** di luar **main** ***function*** seperti di bawah ini :

```
func getDiscount(book Books) {
  fmt.Printf("Diskon : %.2f" , book.price * 0.2)
}
```

Dan untuk menggunakannya tulis kode di bawah ini :

```
getDisccount(ComputerScience)
```

Keluarannya :

```
// Diskon : 17600.02
```

Struct As Pointer

Untuk membuat ***struct*** sebagai ***pointer***, kita mulai dengan menyimpan ***memory address*** ke dalam ***variable*** **book1** :

```
func main() {
  var ComputerScience Books
  ComputerScience.bookID = 1
  ComputerScience.title = "Belajar Dengan Jenius Javascript"
  ComputerScience.author = "Gun Gun Febrianza"
  ComputerScience.price = 88000.100
  book1 := &ComputerScience
}
```

Untuk membaca salah satu ***field*** pada ***struct pointer***, perhatikan kode di bawah ini :

```
   (*book1).price = 100000
   fmt.Println(book1)
```

Keluaran :

```
//&{1 Belajar Dengan Jenius Javascript Gun Gun Febrianza 100000}
```

Nested Struct

Kita juga dapat menggunakan ***struct*** sebagai suatu ***types*** di dalam ***struct*** lainnya. Sebagai contoh kita dapat membuat ***struct*** **Publisher** untuk digabung dengan ***struct*** **Books**. Perhatikan kode di bawah ini :

```
type Publisher struct {
  company string
  location string
}

type Books struct {
  bookID int32
  title   string
  author  string
  price float32
  publisher Publisher
}
```

Untuk mengisi ***struct*** **Publisher** perhatikan kode di bawah ini :

```
   var ComputerScience Books
```

```
  ComputerScience.publisher.company = "Open Library Indonesia"
  ComputerScience.publisher.location = "Indonesia"
```

Kita dapat membaca nilai dalam ***struct*** **Publisher** :

```
  fmt.Println(ComputerScience.publisher)
  fmt.Println(ComputerScience.publisher.company)
  fmt.Println(ComputerScience.publisher.location)
```

Add Method to Struct

Kita dapat memberikan sebuah ***method*** untuk sebuah ***struct*** menggunakan ***method receiver***. Pada contoh kode di bawah ini kita menambahkan ***method*** **BooksInfo** untuk ***struct*** **Books** :

```
type Books struct {
  bookID int32
  title   string
  author  string
  price float32
  publisher Publisher
}

func (e Books) BooksInfo() {
  fmt.Println(e.bookID)
  fmt.Println(e.title, e.author)
  fmt.Println(e.price)
}
```

Access Struct Method

Untuk memanggil ***method*** pada ***variable struct*** akses langsung ***function*** **BooksInfo** seperti pada kode di bawah ini :

```
func main() {
  var ComputerScience Books
  ComputerScience.bookID = 1
  ComputerScience.title = "Belajar Dengan Jenius Javascript"
  ComputerScience.author = "Gun Gun Febrianza"
  ComputerScience.price = 88000.100
  ComputerScience.BooksInfo()
}
```

Keluaran :

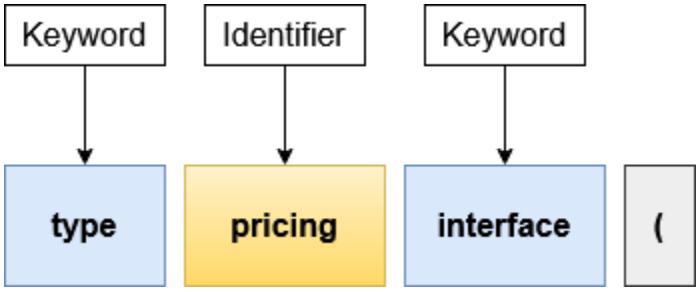
```
/*
1
Belajar Dengan Jenius Javascript Gun Gun Febrianza
88000.1
*/
```

3. *Interface*

Interface dalam bahasa pemrograman ***Go*** berbeda dengan konsep ***interface*** dalam bahasa pemrograman lainnya. ***Interface*** adalah sebuah ***custom type*** yang digunakan untuk membuat spesifikasi ***method*** tunggal atau sekumpulan ***method signatures***.

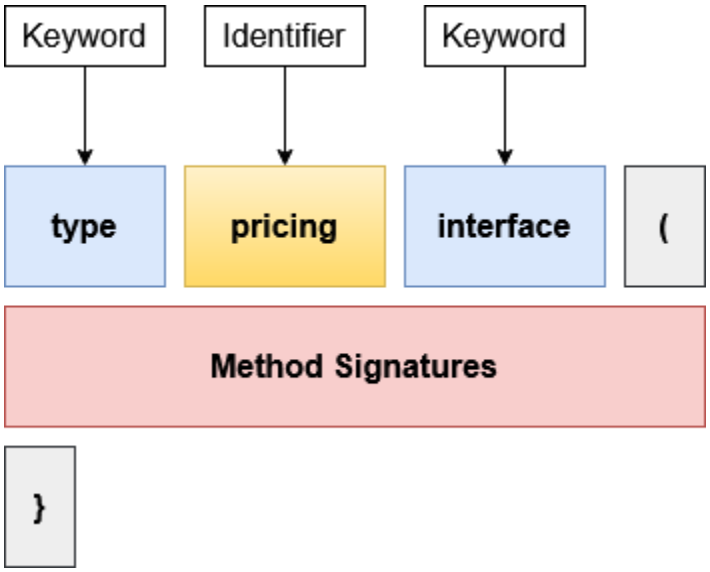
Interface dikatakan ***abstract*** karena kita tidak akan membuat sebuah ***instance*** dari sebuah ***interface***.

Jika kita bedah struktur sebuah ***interface***, terdapat 3 struktur :



Gambar 89 Interface Syntax

Di dalam sebuah ***interface*** terdapat ***method signature*** :



Gambar 90 Method Signature

Sebelumnya kita telah mempelajari cara untuk membuat ***struct*** yaitu membuat ***struct*** **Books**, dengan ***fields*** sebagai berikut :

```
type Books struct {
  bookID int32
  title   string
  author  string
  price float32
}
```

Selanjutnya kita hendak membuat sebuah ***interface*** yang memiliki dua buah ***method signature*** yaitu ***method*** **getDiscount()** dan **getHalfPrice()** :

```
type pricing interface {
  getDiscount() float32
  getHalfPrice() float32
}
```

Selanjutnya kita membuat ***method*** yang telah kita tulis di dalam ***interface*** :

```
func (m Books) getDiscount() float32 {
  return  m.price * 0.2
}

func (m Books) getHalfPrice() float32 {
  return  m.price / 2
}
```

Jika kita perhatikan ada yang berbeda dari cara menyusun ***function*** untuk ***interface***, setelah ***keyword*** **func** terdapat ***parameter*** untuk ***struct***. Selanjutnya di dalam **main** ***function*** buat lagi ***variable struct*** :

```
func main() {
  var ComputerScience Books
  ComputerScience.bookID = 1
  ComputerScience.title = "Belajar Dengan Jenius Javascript"
  ComputerScience.author = "Gun Gun Febrianza"
  ComputerScience.price = 88000.100
}
```

Selanjutnya masukan kode di bawah ini di dalam **main** ***function***, kita mencoba mengakses ***method*** yang dimiliki oleh ***interface*** :

```
  fmt.Println(ComputerScience.getDiscount()) //17600.021
  fmt.Println(ComputerScience.getHalfPrice()) //44000.05
```

Keluaran dari program di atas adalah :

```
  //17600.021
  //44000.05
```

Evaluation – Learning Metrics

Composite Types - Pointer		
[ ]	Memahami Konsep **De-referencing** pada Go.	
[ ]	Memahami **Read Memory Address** pada Go.	
[ ]	Memahami **Pointer Variable** pada Go.	
[ ]	Memahami **Store Memory Address** pada Go.	
[ ]	Memahami **Access Pointer Variable** pada Go.	
[ ]	Memahami **Pointer As Parameter** pada Go.	
[ ]	Memahami Konsep **Passing By Value** pada Go.	
[ ]	Memahami Konsep **Passing By Pointer** pada Go.	
[ ]	Memahami Konsep **Nil Value** pada Go.	
[ ]	Memahami Konsep **Pointer Template String** pada Go.	

Composite Types - Pointer	
Score Faham	**Score** Belum Faham
(/ 10)	(/ 10)

Composite Types - Struct		
[ ]	Memahami Konsep **Struct** pada Go.	
[ ]	Memahami **Create Struct** pada Go.	
[ ]	Memahami **Declare Custom Type** pada Go.	
[ ]	Memahami **Read Struct Field** pada Go.	
[ ]	Memahami **Struct As Parameter** pada Go.	
[ ]	Memahami **Struct As Pointer** pada Go.	
[ ]	Memahami **Nested Struct** pada Go.	
[ ]	Memahami **Struct As Pointer** pada Go.	
[ ]	Memahami **Struct Method** pada Go.	
[ ]	Memahami **Access Struct Method** pada Go.	

Composite Types - Struct	
Score Faham	**Score** Belum Faham
(/ 10)	(/ 10)

Subchapter 10 – *Data Structure* ✓

Learning to write programs stretches your mind,
and helps you think better,
creates a way of thinking about things
that I think is helpful in all domains.

—Bill Gates

Subchapter* 10 – *Objectives

- Mempelajari Konsep ***Data Structure*** pada ***Go***.
- Mempelajari ***Array*** pada ***Go***.
- Mempelajari ***Slice*** pada ***Go***.
- Mempelajari ***Map*** pada ***Go***.

Data structure adalah cara untuk mengorganisir sekumpulan data agar bisa digunakan secara efisien dan mampu memecahkan masalah dalam pemrograman.

Setiap ***data structure*** memiliki kelebihan dan kekurangan. Dalam dunia pemrograman terdapat 2 data struktur yaitu :

1. ***Linear data structure***
2. ***Non linear data structure***.

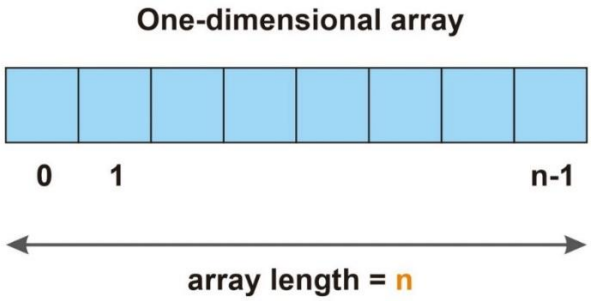
Pada ***Linear Data Structure*** sebuah nilai disusun secara ***linear fashion*** atau disimpan secara berurutan. Diantaranya adalah ***array***, ***list***, ***stack*** dan ***queue***.

Sementara ***Non Linear data structure*** nilai tidak disusun secara berurutan diantaranya adalah ***Graph***, ***Map*** dan ***Tree***.

1. *Array*

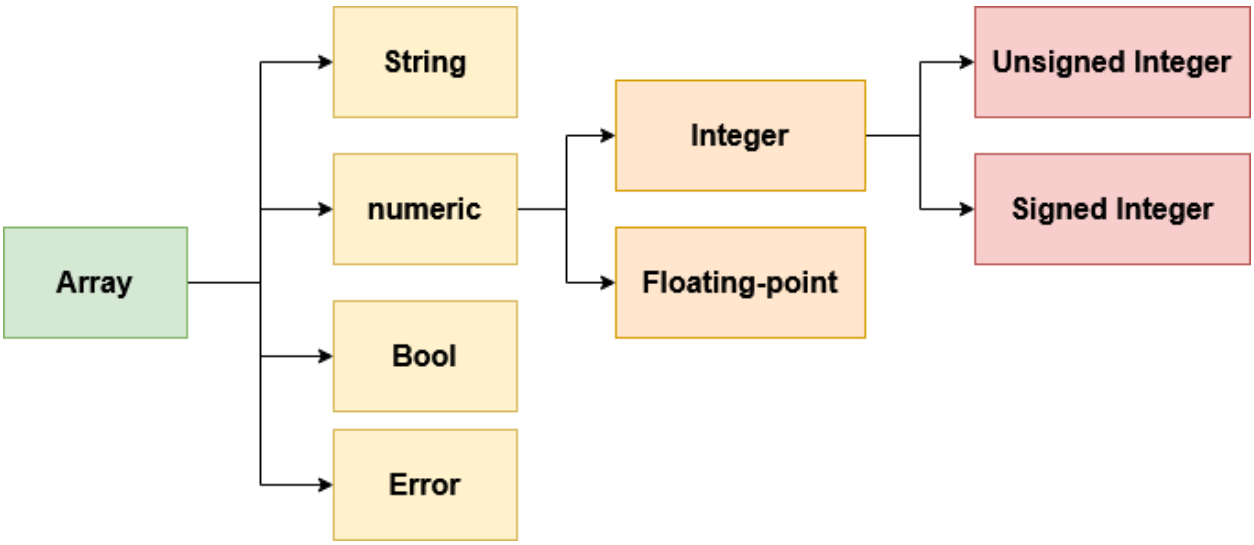
Array adalah ***data structure*** yang paling sering digunakan di dalam pemrograman. Permasalahan matematika seperti ***matrix*** dan sebagian besar dalam dunia ***algebra*** bisa diselesaikan menggunakan ***array.***

Array adalah sebuah ***data structure*** dengan tipe ***linear data structure***, data disimpan secara berurutan dimulai dari ***index*** ke 0. Setiap ***index*** di dalam ***array*** dapat menyimpan sebuah nilai yang disebut dengan ***element***.



Gambar 91 One-dimensional Array

Kita bisa membuat ***array*** dengan berbagai ***primitive types*** seperti :



Gambar 92 Array Types

Create Fixed-length Array

Create Array with Ellipses

Access Array Element

Modify Array Element

Read Array Length

Looping Array

Multidimensional Array

Looping Multidimensional Array

2. *Slice*

Keterbatasan sebuah ***array*** adalah sifatnya yang memiliki lebar ***fixed*** atau ***finite***. Untuk mengatasi hal ini diciptakanlah ***slice*** yang lebarnya bersifat dinamis. ***Slice*** adalah sebuah potongan ***array*** dan secara ***internal slice*** adalah sebuah ***array***.

Create Slice

Kita dapat membuat sebuah ***slice*** tanpa perlu menentukan lebarnya :

```
  var slice []int //nill value
```

Kita dapat membuat ***slice*** sebuah ***array integer*** :

```
func main() {
  var slice []int
  slice = []int{1, 2, 3, 4, 5}
  fmt.Println(mySlice) //[1 2 3 4 5]
}
```

Jika kita ingin membuat sebuah ***slice*** tanpa perlu memberikan nilai awal kita dapat menggunakan ***function*** **make** :

```
func main() {
  slice := make([]float64, 4)
  fmt.Println(slice) //[0 0 0 0]
}
```

Pada kode di atas kita membuat sebuah ***slice*** yang secara ***internal*** berupa ***array*** **float64** dengan lebar untuk 5 ***element***. Jika kita periksa nilai setiap ***elements*** dalam ***array*** tersebut akan memiliki nilai ***default*** 0.

Create Sub-slice

Untuk membuat ***sub-slice***, kita akan membuat sebuah ***slice*** terlebih dahulu. Sebuah ***slice*** dengan ***length*** 4 dan ***capacity*** 4 :

```
slice := make([]float64, 4)
slice = []float64{1.1, 2.2, 3.3, 4.4}
```

Untuk membuat ***sub-slice*** dari ***slice*** di atas tulis kode di bawah ini ;

```
var subSlice = slice[1:4]
fmt.Println(subSlice) //[2.2 3.3 4.4]
```

Capacity

Untuk mengetahui lebar maksimum suatu ***slice*** jika kita ingin melakukan ***re-slice***, pada kode di atas ***sub-slice*** hanya memiliki 3 ***elements*** sehingga nilai **cap** dari *sub-slice* adalah 3 :

```
   fmt.Println(cap(subSlice)) //3
```

Low & High Expression

Kita dapat membuat sebuah ***slice*** dengan ***low & high expression*** menggunakan notasi [low : high]. Di bawah ini kita membuat ***fixed-length array*** dengan lebar 5, lalu memotong ***array*** untuk mendapatkan ***element*** pada ***index*** ke 1 dan ke 4 saja :

```
func main() {
  arr := [5]int64{1, 2, 3, 4, 5}
  slice := arr[1:4] //[2 3 4]
  fmt.Println(slice)
```

```
}
```

Reference Type

Jika pada kode sebelumnya kita mengubah nilai ***slice*** pada ***index*** ke 1, maka nilai ***slice*** pada ***index*** ke 1 akan berubah :

```
slice[1] = 99
fmt.Println(slice) //99
```

Namun karena **slice** tersebut menggunakan ***memory*** miliki ***variable*** **arr** maka jika kita memeriksa nilai ***array*** yang ada pada ***variable*** **arr**, ***element*** yang dimilikinya juga ikut berubah :

```
func main() {
  arr := [5]int64{1, 2, 3, 4, 5}
  slice := arr[1:4] //[2 3 4]
  fmt.Println(slice)
  slice[1] = 99
  fmt.Println(slice)
  fmt.Println(arr) // [1 2 99 4 5]
}
```

Pada ***variable*** **slice** ***element*** 99 ada pada ***index*** ke 1 dan pada ***variable*** **arr** terdapat pada ***index*** ke 2. Sifat ini membuat ***slice*** seperti ***pointer*** untuk potongan ***array***.

Append to Slice

Jika kita ingin menambahkan suatu ***element*** pada suatu ***slice***, kita dapat menggunakan ***function*** **append** seperti pada kode di bawah ini :

```
```

```
func main() {
  slice := make([]float64, 4)
  slice = []float64{1.1, 2.2, 3.3, 4.4}
  var subSlice = slice[1:4]
  fmt.Println(subSlice) //[2.2 3.3 4.4]
  subSlice = append(subSlice, 5.5)
  fmt.Println(subSlice) //[2.2 3.3 4.4 5.5]
}
```

Pada kode di atas kita menambahkan ***element*** **5.5** kepada **subslice**.

Copy Slice

Kita juga dapat menyalin sebuah slice, terdapat function copy untuk melakukannya :

```
  subSlice = append(subSlice, 5.5)
  fmt.Println(subSlice) //[2.2 3.3 4.4 5.5]

  var clone = make([]float64, 4)
  copy(clone, subSlice)
  fmt.Println(clone) //[2.2 3.3 4.4 5.5]
```

Pada kode di atas kita menyalin ***slice*** yang ada pada variable **subSlice** untuk ***variable*** **clone**.

Looping Slice

Untuk melakukan perulangan pada sebuah ***slice*** perhatikan kode di bawah ini :

```
func main() {
  slice := make([]float64, 4)
  slice = []float64{1.1, 2.2, 3.3, 4.4}
  var subSlice = slice[1:4]
```

```
    fmt.Println(subSlice) //[2.2 3.3 4.4]
    for i, v := range subSlice {
      fmt.Println(i, v)
    }
}
```

Pada kode di atas kita melakukan perulangan pada ***variable*** **subSlice** menggunakan perulangan ***for – range***.

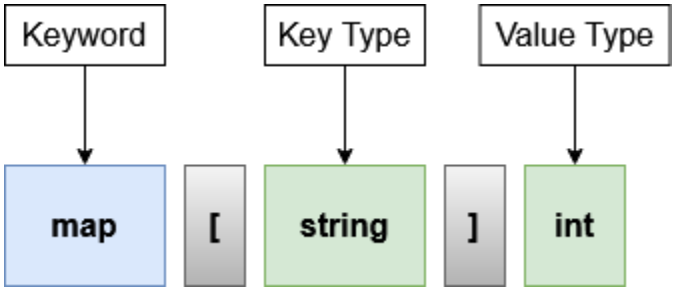
```
    for i, v := range subSlice {
      fmt.Println(i, v)
    }
```

Keluaran atau ***output*** yang dihasilkan :

```
/*
0 2.2
1 3.3
2 4.4
*/
```

3. *Map*

Map adalah sebuah ***data structure non-linear*** yang terdiri dari koleksi ***key/value pair***. Kita dapat menyimpan ***value*** pada sebuah ***map*** menggunakan sebuah ***key*** dan ***key*** harus unik.



Gambar 95 Example Map Syntax

Kelebihan dari ***map*** adalah kecepatan untuk mendapatkan data berdasarkan ***key***, karena ***key*** bekerja seperti ***index*** yang menjadi penunjuk untuk mendapatkan nilainya.

Create Map

Untuk membuat sebuah ***map*** perhatikan kode di bawah ini :

```
func main() {
 var couple = map[string]int{"Maudy Ayunda": 28, "Gun Gun Febrianza": 28}
 fmt.Println(couple)
   // map[Gun Gun Febrianza:28 Maudy Ayunda:28]
}
```

Selain menggunakan cara di atas kita juga dapat menggunakan ***function*** **make** untuk membuat sebuah ***map***, perhatikan kode di bawah ini :

```
func main() {
```

```
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  fmt.Println(couple)
  // map[Gun Gun Febrianza:28 Maudy Ayunda:28]
}
```

Check Map Types

Untuk memeriksa tipe data pada ***key & value*** yang digunakan pada ***map,*** tulis kode di bawah ini :

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  fmt.Printf("%T\n", couple) //map[string]int
  fmt.Println(reflect.TypeOf(couple)) //map[string]int
}
```

Pada kode di atas kita menggunakan ***template string*** **atau** ***reflect package*** untuk membaca tipe data untuk ***key & value*** yang digunakan pada ***map.***

Read Map Length

Untuk membaca jumlah ***element*** yang terdapat pada suatu ***map*** tulis kode di bawah ini:

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
```

```
  fmt.Println(len(couple)) //2
}
```

Pada kode di atas kita menggunakan ***function*** **len** untuk menghitung jumlah ***element*** pada ***map*** **couple**.

Add Element

Untuk menambahkan sebuah ***element*** pada map **couple**, buat satu ***assignment statement*** lengkap dengan ***key & value*** yang dimilikinya. Seperti pada contoh kode di bawah ini :

```
  couple["Afghan"] = 33
```

Pada kode di atas kita menambahkan 1 buah ***element*** pada ***map*** **couple** dengan ***key*** **Afghan** dan ***value*** **33**.

Read Element

Untuk membaca sebuah ***element*** di dalam ***map***, gunakan ***key*** yang ingin di cari :

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  fmt.Println(couple["Gun Gun Febrianza"]) //28
}
```

Pada kode di atas kita menggunakan ***key*** **Gun Gun Febrianza** untuk mencari ***element*** di dalam ***map*** **couple**.

Modify Element

Untuk memodifikasi sebuah ***element*** di dalam ***map***, gunakan ***key*** dari ***element*** yang ingin di modifikasi :

```
func main() {
  couple := make(map[string]int)
  couple["Gun Gun Febrianza"] = 28
  fmt.Println(couple["Gun Gun Febrianza"]) //28
  couple["Gun Gun Febrianza"] = 21
  fmt.Println(couple["Gun Gun Febrianza"]) //21
}
```

Pada kode di atas kita menggunakan ***key*** **Gun Gun Febrianza** untuk memodifikasi nilai ***element*** di dalam ***map*** **couple**.

```
  //sebelum di modifikasi
  couple["Gun Gun Febrianza"] = 28
  //sesudah di modifikasi
  couple["Gun Gun Febrianza"] = 21
```

Delete Element

Untuk menghapus sebuah ***element*** di dalam ***map***, gunakan ***key*** dari ***element*** yang ingin di hapus :

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  delete(couple, "Gun Gun Febrianza")
```

```
  fmt.Println(len(couple)) //1
}
```

Pada kode di atas kita menggunakan ***function*** **delete** untuk menghapus sebuah ***element*** di dalam ***map***. Pada ***parameter*** pertama ***function*** **delete** terdapat nama ***map*** dan para ***parameter*** kedua ***key*** dari ***element*** yang ingin di hapus di dalam ***map***.

```
  delete(couple, "Gun Gun Febrianza")
```

Pada ***parameter*** pertama terdapat ***map*** **couple** dan pada ***parameter*** kedua terdapat ***key*** **Gun Gun Febrianza** untuk menghapus nilai ***element*** di dalam ***map***.

```
  fmt.Println(len(couple)) //1
```

Selanjutnya untuk membuktikan bahwa ***element*** telah terhapus kita membaca kembali jumlah ***element*** menggunakan ***function*** **len**. Keluaran yang dihasilkan adalah 1.

Looping Map

Untuk melakukan perulangan pada sebuah ***map*** perhatikan kode di bawah ini :

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  couple["Afghan"] = 33
  for key, element := range couple {
    fmt.Println("Key:", key, "=>", "Value:", element)
  }
}
```

Kita menggunakan perulangan **for..range** untuk melakukan iterasi pada ***map*** :

```
  for key, element := range couple {
    fmt.Println("Key:", key, "=>", "Value:", element)
  }
```

Keluaran atau ***output*** yang dihasilkan :

```
/*
Key: Maudy Ayunda => Value: 28
Key: Gun Gun Febrianza => Value: 28
Key: Afghan => Value: 33
*/
```

Truncate Map

Untuk menghapus seluruh ***element*** yang ada pada ***map***, timpa ***map*** dengan ***map*** yang kosong seperti pada kode di bawah ini :

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  couple["Afghan"] = 33
  couple = make(map[string]int)
  fmt.Println(len(couple)) //0
}
```

Sorting Map By Key

Untuk melakukan ***sorting map by key*** kita perlu menggunakan ***package*** **sort** :

```
import (
  "fmt"
  "sort"
)
```

Jika kita ingin melakukan ***sorting elements*** berdasarkan abjad secara ***ascending***, tambahkan terlebih dahulu ***statements*** di bawah ini pada **main** ***function*** :

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  couple["Afghan"] = 33
}
```

Selanjutnya kita harus membuat ***slice*** untuk menyimpan ***key*** yang lebarnya disesuaikan dengan jumlah ***elements*** di dalam ***map*** :

```
  keys := make([]string, 0, len(couple))
```

Simpan setiap ***keys*** pada ***map*** ke dalam ***slice*** :

```
  for k := range couple {
    keys = append(keys, k)
  }
```

Sorting keys dalam ***map*** secara ***ascending*** menggunakan **Strings** ***function*** yang telah disediakan dalam ***package*** **sort**,

```
  sort.Strings(keys)
```

Tampilkan setiap ***elements*** sesuai dengan ***keys*** yang telah di ***sorting*** di dalam ***slice*** :

```
  for _, k := range keys {
    fmt.Println(k, couple[k])
  }
```

Sorting Map By Values

Untuk melakukan ***sorting map by value*** kita masih menggunakan ***package*** **sort** :

```
import (
  "fmt"
  "sort"
)
```

Jika kita ingin melakukan ***sorting elements*** berdasarkan angka secara ***ascending***, tambahkan terlebih dahulu ***statements*** di bawah ini pada **main** ***function*** :

```
func main() {
  couple := make(map[string]int)
  couple["Maudy Ayunda"] = 28
  couple["Gun Gun Febrianza"] = 28
  couple["Afghan"] = 33
}
```

Selanjutnya kita harus membuat ***slice*** untuk menyimpan ***value*** yang lebarnya disesuaikan dengan jumlah ***elements*** di dalam ***map*** :

```
  values := make([]int, 0, len(couple))
```

Simpan setiap ***values*** pada ***map*** ke dalam ***slice*** :

```
  for _, v := range couple {
    values = append(values, v)
  }
```

Sorting values dalam ***map*** secara ***ascending*** menggunakan **Int** ***function*** yang telah disediakan dalam ***package*** **sort**,

```
  sort.Ints(values)
```

Tampilkan setiap ***elements*** sesuai dengan ***values*** yang telah di ***sorting*** di dalam ***slice*** :

```
  for _, v := range values {
    fmt.Println(v)
  }
```

Merging Map

Untuk menggabungkan dua buah ***map***, buatlah dua buah ***map*** seperti di bawah ini :

```
func main() {
  first := map[string]int{"C#": 1, "C++": 2, "Go": 3}
   second := map[string]int{"Java": 1, "Rust": 5, "C": 3, "JS": 4}
}
```

Setelah itu gunakan perulangan **for..range** untuk mendapatkan ***keys*** & ***value*** dari ***map*** **second**, untuk disimpan ke dalam ***map*** **first** :

```
  for k, v := range second {
```

```
    first[k] = v
  }
```

Tampilkan nilai ***map*** :

```
  fmt.Println(first)
```

Keluaran yang dihasilkan :

```
  // map[C:3 C#:1 C++:2 Go:3 JS:4 Java:1 Rust:5]
```

References

[1] "Statista," [Online]. Available: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. [Accessed 11 August 2019].

[2] Turbak, Franklyn and Gifford, David, "Syntax, Semantic and Pragmatic," in *Design Concept In Programming Language*, USA: MIT Press, p. 4.

[3] O'riordan, "Introduction to C," in *Learning GNU C*, GNU, 2007, p. 3.

[4] London, Keith, "4, Programming," in *Introduction to Computers*, London, Faber and Faber Limited, 1968, p. 184.

[5] Giloi, Wolfgang, K., "Konrad Zuse's Plankalkül: The First High-Level "non von Neumann," *Programming Language,* vol. 19, pp. 17-24, 1997.

[6] Irvine, "Basic Concepts," in *Assembly Language for x86 Processor, 7th edition*, Pearson, 2014, p. 4.

[7] Jorgensen & Brian, "Ecological Informatics – Computer Language," in *Encyclopedia of Ecology*, Amsterdam, Elsevier, 2004, p. 728.

[8] Casey & Ben, "Appendix F : Programming Language," in *Processing – A Programming Handbook for Visual Designer and Artist, Second Edition*, London , MIT Press, 2014, p. 620.

[9] Frank, Lisa Trendich, "Hopper, Grace Murray (1906-1992)," in *An Encyclopedia of American Women at War – From the Home Front to the Battlefields*, ABC-CLIO, 2013, p. 308.

[10] Barone Luciano Maria, Organtini Ezzo, Ricci Frederico, "Programming Language," in *Scientific Programming : C Language, Algorithm and Model in Science*, World Scientific Publishing, 2013, p. 47 .

[11] Kakde, O.G., "What is Cross-Compiler," in *Algorithm for Compiler Design*, Massachussets , Charles River Media, 2002, p. 2.

[12] Dick, Kees, Henri, Ceriel and Koen, "Assemblers, Dissasemblers, Linkers & Loaders," in *Modern Compiler Design*, springer, 2012, p. 367.

[13] Tanenbaum & Herbert, "Requirement for Virtualization," in *Modern Operating System*, United State America, Pearson Prentice Hall, 2004 , p. 475.

[14] Farr, Terrence, "The Big Picture," in *The Definitive ANTLR 4 Reference, 2nd Edition*, USA, Pragmatic, 2013, p. 10.

[15] Puntambekar, "Overview of Compilation," in *Compiler Design*, India , Technical Publication Pune, 2009, p. 4.

[16] Gosselin, Don, "Introduction to Programming and Visual C++," in *Microsoft Visual C++ .NET*, Cengage, 2010, p. 10.

[17] Malik, D.S, "Processing C++ Program," in *C++ Programming: Program Design Including Data Structures*, Cengage, 2009, p. 13.

Tentang Penulis

Penulis adalah Mahasiswa lulusan Universitas Komputer Indonesia (UNIKOM Bandung). Semenjak masuk ke bangku kuliah sudah memiliki *habit* membuat karya tulis di bidang pemrograman, *habit* ini mengantarkan penulis untuk membangun skripsi di sektor ***Compiler Construction*** dengan judul "Kompiler untuk pemrograman dalam Bahasa Indonesia".

Skripsi yang fokus membuat bahasa pemrograman berbahasa Indonesia, dengan paradigma *object-oriented programming*.

Penulis adalah *Founder* sekaligus *(CTO) Market Koin Indonesia*, sebuah *platform Trading Engine* tempat masyarakat dapat membeli dan menjual *bitcoins, ethereum* dan *alternative cryptocurrency* lainnya.

Dari tahun 2017 penulis sudah mendapatkan investasi dan pembiayaan *equity financing* dari pengusaha-pengusaha di Eropa untuk mengembangkan *platform* Market Koin & Blockchain.

Riset-riset yang sedang penulis kembangkan adalah teknologi *Cross-border Payment, Cryptocurrency Arbitrage System, High-Frequency Trading (HFT) Engine,* dan *platform* terbaru yang sedang dikembangkan adalah ***Lightning Bank***.

Sebuah teknologi yang penulis kembangkan untuk membantu perbankan di *Denmark* dan Indonesia agar bisa bertransaksi secara *instant* dan biaya transaksi di bawah 3% sesuai target *Sustainable Development Goals (SDG) United Nation*.

Penulis juga aktif dalam kegiatan literasi finansial untuk masyarakat dan pengembangan Industri Maritim Indonesia, tempat penulis membangun usaha di sektor Industri Udang (*vannamei*).