The Little Go Book

by Karl Seguin

The Little Go Book

Karl Seguin

စာအုပ်နှင့်ပတ်သက်၍

လိုင်စင်

နောက်ဆုံး Version

ယခုစာအုပ်၏ နောက်ဆုံး version ကိုအောက်ဖော်ပြပါလင့်တွင် ဖတ်ရှုနိုင်ပါသည်။ https://github.com/karlseguin/the-little-go-book

အစပျိုး

Language အသစ်တစ်ခုသင်တိုင်း ကျွန်တော်အတွက် စိတ်နှစ်ခွဖြစ်ရပါတယ်။ တစ်ဖက်မှာ language တွေဟာ နေ့စဉ် လုပ်ငန်း ဆောင်တာများ၏ အခြေခံဖြစ်ပြီး အပြောင်းအလဲ အသေးလေးတစ်ခုပင် ထိရောက်မှု အများကြီးရှိနိုင်ပါတယ်။ တခုခုကို နှိပ်လိုက်ပါက သင့် program ကိုသက်ရောက်စေနိုင်သည့် ရလဒ်ကပင် *ဟာကန့်* ဖြစ်နိုင်ပြီး language များ၏ အမြင်ကို ပြန်၍ ပြောင်းလဲစေနိုင်သည်။ တဖက်ကပြန်ကြည့်ပါက language များသည် ဆင့်ကဲ design လုပ်ထားသည်ဖြစ်၍ ဘာသာစကားအသစ်တစ်ခုကို လေ့လာသည်နှင့် နှိုင်းစာလျှင် Keyword အသစ်၊ type system နှင့် coding style များမှ အစ library အသစ်များ၊ အဖွဲ့အစည်းအသစ်များနှင့် paradigms အသစ်များကို လေ့လာရခြင်းသည် အင်မတန်ပင် အလုပ်ရှုပ်သည်က မငြင်းသာ။ တခြားအရာများနှင့် နှိုင်းယှဉ်ပါက language အသစ်ကို လေ့လာခြင်းသည် အချိန်ကို အကျိုးရှိစွာ အသုံးချသည်ကော ဟုတ်ပါ့မလား ဟု သံသယဝင်မိလေသည်။

သို့ပင်သော်ညား ကျွန်တော်တို့သည် *ရှေ့ကိုဆက်တိုး* ရမည်ဖြစ်သည်။ language များသည် ကျွန်တော်တို့ လုပ်ဆောင်သည့် အရာများ၏ အခြေခံဖြစ်သဖြင့် ကျွန်တော်တို့ အနေဖြင့် *ခြေလှမ်းတစ်လှမ်းချင်း* ဖြစ်စေ လှမ်းရမည် ဖြစ်သည်။ အပြောင်းအလဲများမှာ တဆင့်ချင်းစီဖြစ်သော်လည်း

productivity ၊ Readability ၊ Performance ၊ Testability ၊ Dependency Management ၊ Error Handling ၊ Documentation ၊ Profiling ၊ Communities နှင့် Standard Libraries များမှာ ကျယ်ပြန့်ပြီး ထိရောက်မှုမှာလည်း ကွာခြားလှပေသည်။ *စားချက်တစ်ထောင်ဖြင့်* အခုတ်ခံရသည်ကို အကောင်း ဘယ်လိုပြောရမှန်းပင် မသိပေ။

ထိုနေရာတွင် မေးစရာ ပေါ်လာသည်က အဘယ်ကြောင့် *Go* ကိုရွေးချယ်ခဲ့သနည်း ဟုမေးပါက ဖြေစရာနှစ်ခုရှိသည်။ Go သည် ရိုးရိုးရှင်းရှင်း standard libraries များနှင့် ရိုးရိုးရှင်းရှင်း language တစ်ခုဖြစ်သည်။ ထပ်ဆင့် တိုးတက်သွားသော ၎င်း၏ သဘာဝကြောင့် ကျွန်တော်တို့ တခြား language များတွင် တွေ့ကြုံနေရသည့် အရှုပ်ထုပ်များ ကို ရှင်းလင်းနိုင်သည်။ နောက်တစ်ခုမှာ တခြား developer များအနေဖြင့် မိမိတို့တည်ရှိပြီးသော အရည်အချင်းများဖြင့် ပေါင်းစပ်အသုံးချနိုင်သည်။

Go မှာ system language (ဥပမာ operation system များ၊ device driver များ တည်ဆောက်ရန်) အနေဖြင့် ဖွဲ့စည်းထားပြီး C,C++ developer များအတွက် ရည်ရွယ်ထားသော်လည်း လက်တွေ့တွင် application developer များမှာ အဓိက Go ကိုအသုံးပြုသူ အများစုဖြစ်နေသည်။ အဘယ်ကြောင့်ဆိုသည်ကို system developers မဟုတ်သဖြင့် မပြောနိုင်သော်လည်း website ရေးသူများ ၊ Service နှင့် desktop application ရေးသူများ

အတွက် high-level နှင့် Low-level application များအကြား ပေါင်းကူးတံတားတစ်ခုသဖွယ် ဖြစ်နေသည်။

Messaging၊ Caching၊ နှင့် အချက်အလက်အမြောက်အများ အသုံးပြု၍ တွက်ချက်မှုများ၊ Command Line Interface များ၊ Logging၊ Monitoring နှင့် မည်သို့မည်ပုံ ခေါင်းစဉ်တပ်ရမှန်းမသိသော အချို့သော အလုပ်များ အတွက် အချိန်ကြာလာသည်နှင့်အမျှ ပို၍ရှုပ်ထွေးလာပြီး တပြိုင်နက် အမြောက်အများ ထောင်နှင့်သောင်းနှင့်ချီ၍ တိုင်းတာလာရသည့်အခါ မိမိတို့လိုအပ်သလို Infrastructure ကို ပြုပြင်ပြောင်းလဲနိုင်ရန် လိုအပ်လာသည်။ ထိုသို့သော စံနစ်များကို လူအများစု အသုံးပြုသည့် Ruby သို့မဟုတ် Python ကဲ့သို့ Language များ အသုံးပြု၍ရေးသားနိုင်သော်လည်း တင်းကျပ်သော Type စနစ်ကို အသုံးပြုသော Language များအသုံးပြုပါက အကျိုးဖြစ်ထွန်းသလို Performance လည်းပို၍ ကောင်းပေသည်။ ထိုနည်းတူ Go ကိုအသုံးပြု၍ website များ *ရေးသားနိုင်သော်လည်း* ထိုသို့သော စံနစ်များကို များသောအားဖြင့် ပို၍ ဖြန့်ကျက်ရလွယ်သည့် Node သို့မဟုတ် Ruby ကိုပို၍ သဘောကျမိသည်။

Go အားသာသည့် တခြားအပိုင်းများလည်း ရှိသေးပေသည်။ ဥပမာ Compile လုပ်ထားသော Go Program တစ်ခုအတွက် dependency များမလိုခြင်း။ သင့်အနေဖြင့် user များ Ruby သို့မဟုတ် JVM သွင်းထားခြင်း ရှိမရှိ၊ ရှိလျှင်ပင် မည်သည့် version ကိုသွင်းထားသနည်း စဉ်းစားရန်မလို။

ထိုကြောင့် Go သည် Command line အတွက်နှင့် distribute လုပ်ရန်လိုသည့် utility program များ (ဥပမာ Log Collector လိုမျိုး) အတွက် တဖြည်းဖြည်း လူကြိုက်များလာသည်။

အချုပ်အားဖြင့်ပြောရလျှင် Go ကိုသင်ယူခြင်းသည် သင့်အချိန်ကို အကျိုးရှိစွာ အသုံးချခြင်း ပင်ဖြစ်သည်။ သင့်အနေဖြင့် နာရီပေါင်းများစွာ အချိန်ကုန်ပြီး Go ကိုကျွမ်းကျင်ရန် လုပ်စရာမလိုဘဲ သိသလောက်နဲ့ပင် လက်တွေ့အသုံးချနိုင်သည်။

စာရေးသူ၏ အမှာစာ

ဒီစာအုပ်ကို ရေးရန် တွန့်ဆုတ်နေသည့် အကြောင်းအရင်းများစွာ ရှိသည့် အနက် ပထမဆုံးတစ်ခုမှာ Go ၏ Documentation ဖြစ်သော Effective Go အတော်ပင် အနှစ်ကျသည်။

နောက်တစ်ခုမှာ Language အကြောင်း စာအုပ်ထုတ်ရန် ရေးရသည့် အခါစိတ်တိုင်းမကျ။ Little MongoDB စာအုပ်ထုတ်တုံးက စာဖတ်သူများသည် Relational Database နှင့် Modeling အကြောင်း အခြေခံကို သိရှိမည်ဟု ယူဆနိုင်ပြီး The Little Redis စာအုပ်တွင်မူ သင့်အနေဖြင့် Key Value Store အကြောင်း ရင်းနှီးမည်ဟု ယူဆနိုင်သည်။

စာပိုဒ်နှင့် အခန်းခွဲများအကြောင်း စဉ်းစားရင်း သဘောပေါက်မိသည်က ယခု ကိစ္စတွင်တော့ ၎င်းကဲ့သို့ မှတ်ယူရန် မဖြစ်နိုင်ပေ။ သင့်အတွက် Interface ဆိုသည်မှာ အဘယ်နည်းကို လေ့လာရန် အချိန်မည်မျှယူမည်နည်း ဆိုသည်က အခြားသောသူများအနေဖြင့် *Go တွင် Interface ရှိသည်ဟု* သိထားရုံဖြင့် လုံလောက်သောသူများနှင့်မတူပေ။ နောက်ဆုံးတွင်တော့ တချို့အပိုင်းများ သိပ်အသေးစိတ်နေသော်လည်းကောင်း၊ လိုအပ်နေသော်လည်းကောင်း စာဖတ်သူအနေဖြင့် အသိပေးနိုင်မည် ဟုမှတ်ယူရင်း ရေးရပါသည်။ ထိုသည်ကို ထောက်ရှု၍ စာအုပ်၏ အနေအထားကို သဘောပေါက်နိုင်သည်။

စတင်ခြင်း

သင့်အနေဖြင့် Go နှင့်ရင်းနှီးလိုပါက ဘာမှ install လုပ်ရန်မလိုပဲ Go Playground တွင်စမ်းသပ်နိုင်သည်။ ထိုအပြင် Go Code မှာကို မျှဝေလေ့လာနိုင်သော Go's discussion forum တွင် အကူအညီတောင်းနိုင်ပြီး stackoverflow တွင်လည်း မေးနိုင်သည်။

Go ကို Install လုပ်ရသည်မှာ ခပ်ရိုးရိုးပင်။ Source မှဖြစ်စေ install လုပ်နိုင်သောလည်း compiled လုပ်ပြီးသား binary ကိုအသုံးပြုရန် အကြံပေးလိုသည်။ Go Download Page ကိုသွားပါက Platform မျိုးစုံအတွက် Installer များကိုတွေ့ရမည်ဖြစ်သည်။ ၎င်းတို့ကို ခဏဘေးဖယ်ပြီး Go ကို မည်သို့ setup လုပ်ရန် ကိုယ်ဖာသာကိုယ် လေ့လာကြပါစို့။ သင်တွေ့သည့် အတိုင်း သိပ်မခက်လှပါ။

တချို့သော ဥပမာများမှလွဲ၍ Go သည် Workplace တစ်ခုအတွင်း ရေးရန် ရည်ရွယ်ထားသည်။ Workplace ဆိုသည်မှာ `bin`၊ `pkg` နှင့် `src` ဟုသော folder အသေးများပါဝင်သော folder တစ်ခုဖြစ်သည်။ သင့်အနေဖြင့် ကိုယ်တိုင်စိတ်ကြိုက် folder များ မဆောက်သင့်ပါ။

ဥပမာ projects များကို `~/code` အတွင်း သိမ်းထားသည် ဆိုပါစို့။ `~/code/blog` တွင် blog နှင့်ပတ်သတ်သည့် Code များရှိမည်။ Go တွင်မူ

Workplace က `~/code/go` ဖြစ်ပါက Go နှင့်ပတ်သတ်သော blog ၏ Code များမှာ `~/code/go/src/blog` တွင်ရှိမည်ဖြစ်သည်။

ထိုကြောင့် `go` folder များတွင် source code များကို `src` ဟုသော folder အသေးတွင် အရင်ဦးဆုံးထည့်ရမည်ဖြစ်သည်။

OSX / Linux

ကိုယ့် platform အတွက် ကိုက်ညီသော `tar.gz` ကို download လုပ်ပါ။ OSX အတွက်မူ `go#.#.#.darwin-amd64-osx10.8.tar.gz` ဟုပုံစံဖြင့်ဖြစ်လိမ့်မည်။ နောက်မှ `#.#.#` Go ၏ နောက်ဆုံး version နံပါတ်များဖြစ်မည်။

folder လမ်းကြောင်း *usr*local အတွင်းသို့ `tar -C` *usr*local `-xzf go#.#.#.darwin-amd64-osx10.8.tar.gz.` ဟုရိုက်ထည့်ပြီး ဖြေချလိုက်ပါ။

ထိုနောက် environment variable နှစ်ခု သတ်မှတ်ရန်လိုပါလိမ့်မည်။

၁. သင့်၏ workplace ကို `GOPATH` ဖြင့်ညွှန်းဆိုပါ။ ကျွန်တော်အတွက်တော့ `$HOME/code/go` ဆိုရင်ရပါပြီ။ ၂. ထိုနောက် Go ၏ binary ကို `PATH` ၏ နောက်ဆုံးတွင် ထည့်ဖြည့်ပေးရန်လိုမည်။

ထိုနှစ်ခုကို shell မှ အောက်ပါအတိုင်း ရိုက်ထည့်ပြီး သတ်မှတ်နိုင်ပါသည်။

```
echo 'export GOPATH=$HOME/code/go' >> $HOME/.profile
echo 'export PATH=$PATH:usrlocal/go/bin' >> $HOME/.profile
```

ထိုသို့ သတ်မှတ်ပြီးနောက် shell ကိုပြန်ပိတ်ပြီး ဖွင့်လျှင်ဖွင့် သို့မဟုတ် `source $HOME/.profile` ဟုရိုက်ထည့်ပြီး run နိုင်သည်။

`go version` ဟုရိုက်ထည့်ပြီး မိမိတို့ အသုံးပြုနေသော version ကို `go version go1.3.3 darwin/amd64` နမူနာပုံစံအတိုင်းတွေ့ရှိနိုင်သည်။

Windows

နောက်ဆုံး zip file ကိုဒေါင်းပါ။ x64 system တွင်ဖြစ်ပါက `go#.#.#.windows-amd64.zip` ပုံစံဖြင့် file ကိုဒေါင်းပါ။ `#.#.#` သည် နောက်ဆုံး version ကိုညွှန်ပြနေပါမည်။

ထိုနောက် သင့်စိတ်ကြိုက်နေရာတစ်ခုတွင် ဥပမာ `c:\Go` လိုမျိုးနေရာတွင် unzip လိုက်ပါ။

ထိုနောက် environment variable နှစ်ခု သတ်မှတ်ရန်လိုပါလိမ့်မည်။

၁။ သင့်၏ workplace ကို `GOPATH` ဖြင့်ညွှန်းဆိုပါ။ ဥပမာ `c:\users\goku\work\go` ၂။ သင့်၏ unzip လုပ်ထားသောနေရာရှိ binary ကို `PATH` environment variable တွင်ညွှန်းဆိုဖို့လိုမည်။ ဥပမာ `c:\Go\bin`

Control Panel ထဲရှိ `System` ထဲရှိ `Advanced` Tab ကိုနှိပ်ပါက `Environment Variables` ဟု button ကိုတွေ့ရှိရမည်ဖြစ်ပြီး ထိုမှတဆင့် Environment

variables များကိုသတ်မှတ်နိုင်သည်။ Windows တစ်ချို့ version များတွင်မူ Control Panel ထဲရှိ `System` မှ `Advanced System Settings` ဟူသော option အတွင်းတွင်ရှိမည်။

ထို့နောက် command prompt ကိုဖွင့်၍ `go version` ဟုနှိပ်ပါ။ `go version go1.3.3 windows/amd64` ဟုပုံစံဖြင့် output ကိုမြင်ရပေမည်။

အခန်း (၁) - အခြေခံ

Go သည် C နှင့်ဆင်သော Syntax များနှင့် garbarge collection ပါရှိပြီး compile ပြုလုပ်ရသော type အသားပေး language တစ်ခုဖြစ်သည်။ ဆိုလိုသည်မှာ?

Compilation

Compilation ဆိုသည်မှာ မိမိတို့ရေးသားထားသော source code ကို low level language (ဥပမာ Go တွင် Assembly သို့ပြောင်းလဲပေးပြီး Java နှင့် C# တို့တွင် byte code) သို့ ပြန်၍ပြောင်းလဲသော ဖြစ်စဉ်ကို ဆိုလိုသည်။

ပုံမှန်အားဖြင့် Compile လုပ်ရသော language များသည် Compile လုပ်သည့်အချိန် ကြာလေ့ကြာထရှိသဖြင့် အလုပ်လုပ်ရသည်မှာ သာယာချမ်းမြေ့ခြင်းမရှိလှ၊ Compile လုပ်ပါက မိနစ်ပိုင်းမှစ၍ နာရီပိုင်းအထိကြာလျှင် တဆင့်ပြီးတဆင့် ရေးသားရန် ခက်ခဲလှပေသည်။ ထို့ကြောင့် Golang ၏ ရည်ရွယ် တည်ဆောက်ပုံ ကိုယ်တိုင်က Compilation ကြာချိန်ကို လျှော့ချနိုင်ရန် အသားပေးထားသည်။ ထိုအချက်သည် Project အကြီးများနှင့် အလုပ်လုပ်ရသောသူများ ၊ Feedback cycle မြန်ဆန်သဖြင့် Interpreted Language များဖြင့် အလုပ်လုပ်ရသူများ အတွက်ပါ အဆင်ပြေပါသည်။

Compiled Language များသည် ပုံမှန်အားဖြင့် ပို၍လျင်မြန်ပြီး၊ Dependencies များမလိုအပ်ဘဲ run နိုင်လေ့ရှိသည်။ (အနည်းဆုံး ထို အချက်သည် C,C++ နှင့် Go ကဲ့သို့သော Assembly သို့ တိုက်ရိုက် Compile လုပ်နိုင်သည့် language များအတွက်မှန်ကန်သည်ဟု ဆိုရမည်။)

Static Typing

Static Type ဖြစ်သည့်အတွက် variable တိုင်းသည် type တစ်ခုခု (int, string, bool, []byte စသဖြင့်) တစ်ခုခုပေါ်တွင် ကျရောက်နေမည် ဖြစ်သည်။ သို့သော် Variable တစ်ခုကို type သတ်မှတ်သည်ဖြစ်စေ မဟုတ်ပါက compiler အနေဖြင့် type ကို infer လုပ်သွားမည်ဖြစ်သည်။ (ဥပမာ အနေဖြင့် အောက်တွင် ဖော်ပြသွားပါမည်)။

Static typing နှင့်ပတ်သက်၍ အများအပြား ပြောနိုင်သော်လည်း code ကြည့်ပါက ပို၍ သဘောပေါက်မည် ဖြစ်သည်။ အကယ်၍ သင်သည် Dynamic type language ကိုအသုံးပြုသော နောက်ခံမှ လာပါက အနည်းငယ် အာရုံနောက်မည် ဖြစ်သည်။ သင်တွေးသည်က မမှား သို့သော် အချို့သော အားသာချက်များ အထူးသဖြင့် compile လုပ်ချိန်တွင် တွဲထားသော type များက အားသာချက်ရှိသည်။ ထိုနှစ်ချက်မှာ တွဲလျက်ရှိပြီး တစ်ခုရှိပါက နောက်တစ်ခု ရှိလေ့ရှိသည်။ သို့သော် golang တွင် အတင်းအကျပ် type သတ်မှတ်ရမည်ဟု စည်းကမ်းမရှိပေ။ ခိုင်မာသော

စနစ်တစ်ခုတွင် compiler တစ်ခုအနေဖြင့် စာလုံးပေါင်းများသည့် ပြဿနာများကို ကျော်လွန်၍ သိရှိနိုင်စွမ်းရှိမှသာ ပို၍ကောင်းမွန်သော optimization ကိုဆောင်ရွက်နိုင်မည် ဖြစ်သည်။

C-Like Syntax

C,C++, Java, Javascript နှင့် C# ကဲ့သို့သော C နှင့် ဆင်သည့် language များကိုအကျွမ်းဝင်ပါက Go ကို C နှင့်ဆင်သည်ကို သတိထားမိမည်ဖြစ်ပြီး ယေဘူယျအားဖြင့် လေ့လာရာတွင်လည်း ပို၍ရင်းနှီးပါလိမ့်မည်။ ဥပမာ `&&` က boolean AND ကဲ့သို့ အသုံးပြုပြီး နှစ်ခုကို နှိုင်းယှဉ်ရန် `==` ကိုအသုံးပြုပြီး တွန့်ကွင်းဖြစ်သည့် `{` နှင့် `}` ကို scope တစ်ခု၏ အစနှင့်အဆုံးကို ပိုင်းခြားရာတွင် အသုံးပြုပြီး Array သည် 0 မှစသည်။

C နှင့်တူသော syntax ဖြစ်သော်လည်း အခြေအနေများ ဖော်ပြသည့် လက်သည်းကွင်း `(` နှင့် `)` တို့နှင့် statement တစ်ခုပြီးတိုင်း semi-colon များမှာ Go တွင် ထည့်သည်ဖြစ်စေ မထည့်သည် ဖြစ်စေ အလုပ်လုပ်သည်။ ဥပမာ `if` statment တစ်ခုသည် အောက်ပါအတိုင်းဖြစ်မည် ဖြစ်သည်။

```
if name == "Leto" {
  print("the spice must flow")
}
```

ပို၍ ရှုပ်ထွေးလာသော အခြေအနေများတွင်မူ ပိုင်းခြားရန် လက်သည်းကွင်းများသည် အရေးပါဆဲဖြစ်သည်။

```
if (name == "Goku" && power > 9000) || (name == "gohan" && power
  print("super Saiyan")
}
```

၎င်းတို့မှအပ Go သည် C# နှင့် Java တို့ဖြင့်နှိုင်းစာလျှင် C နှင့်ပို၍ နီးစပ်သည်မှာ syntax တွင်မက ရည်ရွယ်ချက်က အစပင်။ လေ့လာရင်းဖြင့် language ၏ ပြတ်သားမှုနှင့် ရိုးရှင်းမှု တဖြည်းဖြည်း ထင်ဟပ်လာမည်။

Garbage Collected

တချို့ variable များ စတင်တည်ဆောက်ကတည်းက လွယ်ကူသည်။ ဥပမာ function ပြီးဆုံးသွားပါက ပျောက်ကွယ်သွားသော အတွင်းရှိ local variable တစ်ခုကဲ့သို့။ သို့သော် တချို့သော ကိစ္စများတွင်မူ အထူးသဖြင့် Compiler အတွက် မရိုးရှင်းပေ။ function တစ်ခုမှ ပြန်လာသော variable တစ်ခု၏ သက်တမ်းသည် တခြားသော variable များနှင့် object များ၏ reference လုပ်ထားပုံပေါ်မူတည်၍ ရှုပ်ထွေးလှသည်။ Garabage Collection မရှိပါက developer မှ မည်သည့် variable သည်မည်သည့် နေရာက သုံးထားသည်ကို သိရန်လိုပြီး memory ကို free လုပ်ရန်က ၎င်း တာဝန်ဖြစ်သည်။ C ကဲ့သို့ language တွင်မူ variable ကို `free(str);` ဟု free လုပ်ပေးရန်လိုသည်။

Ruby ၊ Python ၊ Java ၊ Javascript ၊ C# နှင့် GO ကဲ့သို့သော language များတွင် ၎င်း variable ၏ အခြေအနေများကို စောင့်ကြည့်ပြီး အသုံးမပြု

ပါက ဖျက်ပစ်သော Garbage Collector များပါရှိပါသည်။ ၎င်းအလုပ်ကို လုပ်ရသဖြင့် အလုပ်ပိုသော်လည်း ဆိုးရွားလှသည့် bugs များဖြစ်နိုင်ချေကို ရှင်းလင်းနိုင်သည်။

Go Code များ runခြင်း

ကျွန်တော်တို့ ခရီးစဉ်ကို ရိုးရှင်းသည့် program တစ်ခုစတင်ရေးသားကာ compile လုပ်ရင်း run ကြည့်ခြင်းဖြင့် စတင်လိုက်ရအောင်။ သင်ကြိုက်သည့် editor ကိုဖွင့်ပြီး အောက်ပါအတိုင်းရေးသားလိုက်ပါ။

```
package main

func main() {
  println("it's over 9000!")
}
```

`main.go` ဟု save လိုက်ပါ။ လတ်တလောတွင်တော့ သင့်အနေဖြင့် save ချင်သည့်နေရာတွင် save နိုင်သည်။ Go workplace အတွင်းဖြစ်ရန်မလိုပေ။ ထိုနောက် shell/command prompt ကိုဖွင့်၍ save ထားသည့် နေရာကို သွားလိုက်ပါ။ ကျွန်တော်အတွက်ကတော့ `cd ~/code` ဟုရိုက်ရုံဖြင့် ရောက်သွားသည်။

နောက်ဆုံးတွင် အောက်က အတိုင်း ရိုက်ထည့်ပြီး program ကို run နိုင်သည်။

```
go run main.go
```

အားလုံးအဆင်ပြေပါက *it's over 9000!* ဟုစာကိုတွေ့ရမည်ဖြစ်သည်။

ဒါဖြင့်နေပါဦး compile လုပ်တာ ဘယ်ရောက်သွားသလဲ? `go run` ဆိုသည်က compile လုပ်ပြီး run ပေးသော command ဖြစ်သည်။ ၎င်းသည် ယာယီ directory ကိုအသုံးပြု၍ program ကို build လုပ်ပြီး execute လုပ်ပြီးနောက် ခြေရာလက်ရာများပါ ဖျောက်သွားခြင်းဖြစ်သည်။ ယာယီ file ၏နေရာကို အောက်ပါအတိုင်း ရိုက်ထည့်၍ တွေ့ရှိနိုင်သည်။

```
go run --work main.go
```

Compile သက်သက်ပြုလုပ်လိုပါက `go build` ကိုအသုံးပြုနိုင်သည်။

```
go build main.go
```

ထိုသို့ဖြင့် run နိုင်မည့် `main` ဟု executable တစ်ခုထုတ်ပေးမည်ဖြစ်သည်။ Linux နှင့် OSX စနစ်များတွင်မူ ရှေ့မှ dot-slash ခံပြီးမှ ခေါ်ရန် မမေ့သင့်ပေ။ ထိုကြောင့် `./main` ဟု ခေါ်ရမည်ဖြစ်သည်။

Develop လုပ်နေစဉ်တွင် `go run` သို့မဟုတ် `go build` ကိုကြိုက်နှစ်သက်ရာအသုံးပြုနိုင်ပြီး မိမိတို့ရေးသားပြီးသော code ကို deploy လုပ်လိုပါက `go build` ဟု binary ထုတ်ပြီး execute ပြုလုပ်နိုင်သည်။

Main

အပေါ်မှာ ရေးထားသော Code ကိုနားလည်မည်ဟု ထင်ပါသည်။ function တစ်ခုတည်ဆောက်ပြီး `println` ဟူသော မူလပထမ ပါလာသည် function ကိုအသုံးချ၍ ရေးသားဖော်ပြလိုက်ခြင်းဖြစ်သည်။ ထို Go function ကို အလိုအလျောက်သိသည်ဟု သင်ထင်ပါသလား? မဟုတ်ပါ တကယ်တော့ Go အတွက် program တစ်ခုတိုင်း၏ စစချင်း run ရန် entry point မှာ `main` package အတွင်းရှိ `main` function ဖြစ်သည်။

Package များအကြောင်းနှင့်ပတ်သက်၍ နောက်ပိုင်း အခန်းများတွင် အသေးစိတ် ထပ်၍ပြောသွားမည် ဖြစ်ပြီး ယခုတွင်မူ `main` package အတွင်းတွင်ရေးသားသည့် အခြေခံကိုနားလည်ရန် အဓိကထားရှင်းပြသွားမည်ဖြစ်သည်။

သင့်အနေဖြင့် စိတ်ကြိုက် code နှင့် package name ကိုပြောင်းလဲပြီး `go run` ဟု run လိုက်ပါက error တက်မည်ဖြစ်သည်။ ထိုနောက် `main` ဟုပြန်ပြောင်းပြီး တခြား function အမည်ကို အသုံးပြုပါကလည်း တခြား error တက်ဦးမည် ဖြစ်သည်။ အပေါ်ကဲ့သို့ပြောင်းလဲပြီး `go build` ဟုရိုက်ထည့်လိုက်ပါ။ ထိုအချိန်တွင် compile လုပ်မည်ဖြစ်သော်လည်း entry point မရှိသဖြင့် ဘာမှ run မည်မဟုတ်ပေ။ library တစ်ခုကိုတည်ဆောက်ပါက ထိုကဲ့သို့ အခြေအနေမျိုးသည် ပုံမှန်ပင်ဖြစ်သည်။

Imports

Go တွင် `println` ကဲ့သို့သော built-in functions များစွာပါရှိပြီး အသုံးပြုနိုင်ရန် reference လုပ်စရာမလိုပေ။ သို့သော် Go ၏ standard libtary ကိုသာအသုံးပြုပြီး thirdparty library များကိုရှောင်ရှားပါက ခပ်ဝေးဝေးရောက်နိုင်မည် မဟုတ်ပေ။ ထို့အတွက် Go တွင် အပြင်မှ Package များမှ Code များကို အသုံးပြုလိုပါက `import` keyword ကိုအသုံးပြုနိုင်သည်။ program ကိုအောက်ပါအတိုင်း ပြောင်းလဲကြည့်လိုက်ပါ။

```
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) != 2 {
    os.Exit(1)
  }
  fmt.Println("It's over", os.Args[1])
}
```

အောက်ကအတိုင်း run နိုင်ပါသည်။

```
go run main.go 9000
```

ယခုအသုံးပြုနေသည်မှာ Go ၏ standard package များဖြစ်သော `fmt` နှင့် `os` တို့ဖြစ်သည်။ ထိုအပြင် မူလပါရှိပြီး ဖြစ်သော `len` ဟူသော function ကိုပါ မိတ်ဆက်ပေးလိုက်သည်။ `len` သည် string တစ်ခု၏ size ကိုသော်လည်းကောင်း dictionary ဖြစ်ပါက ပါဝင်သော value အရေအတွက်

ကိုသော်လည်းကောင်း ယခုကဲ့သို့သော array ဖြစ်ပါက ပါဝင်သော element အရေအတွက်ကို ဖော်ပြပေးသည်။ အဘယ့်ကြောင့် သင့်အနေဖြင့် argument (၂)ခုကို မျှော်လင့်ထားသနည်းဟု ဆိုပါက index 0 တွင်ရှိသည့် ပထမ argument သည် လက်ရှိ run နေသော executable ၏ path အနေဖြင့် ပါရှိမည်ဖြစ်သည်။ (program ကို print ထုတ်မည့် value ပြောင်းလဲခြင်းဖြင့် ကိုယ်တိုင် စမ်းသပ်နိုင်သည်။) package ၏အမည်များကို function အမည်များ၏ ရှေ့တွင် prefix အနေဖြင့် သုံးနေခြင်း (ဥပမာ `fmt.Println`) ကို သတိထားမိမည် ဖြစ်သည်။ ၎င်းသည် တခြား language များနှင့် အဓိက ကွာခြားချက်ဖြစ်ကာ အသေးစိတ်ကို နောက်ပိုင်း အခန်းများတွင် ရှင်းပြသွားပါမည်။ အခုတွင်မူ package တစ်ခုကို import လုပ်တတ်ပြီး အသုံးပြုတတ်ပါက အစကောင်းဟု ဆိုရမည်။

Package များ import လုပ်ရာတွင် Go သည် အလွန်တင်းကျပ်ပါသည်။ package တစ်ခုကို import ပြီး အသုံးမပြုပါက compile လုပ်နိုင်မည် မဟုတ်။ အောက်က အတိုင်း run ကြည့်ပါ။

```
package main

import (
  "fmt"
  "os"
)

func main() {
}
```

`fmt` နှင့် `os` နှစ်ခုကို import လုပ်ထားပြီး အသုံးမပြုသောကြောင့် error နှစ်ခု တက်လိမ့်မည်ဖြစ်သည်။ အာရုံမနောက်ချေဘူးလား။ မှန်ပါသည်။ သို့သော် အချိန်ကြာလာသည်နှင့်အမျှ အာရုံနောက်သော်လည်း ရေးသားကျလာမည် ဖြစ်သည်။ တော်တော်များများ ကိုယ်တိုင်ကိုယ်ကျ ကြံုချင်မှကြံုရသော်လည်း အသုံးမပြုသော import များသည် program ကိုနှေးစေသောကြောင့် Go တွင် တင်းကျပ်ထားခြင်းဖြစ်သည်။

မှတ်သားရန်နောက်တစ်ခုမှာ Go ၏ standard library သည်ရှာဖွေရန် အချက်အလက်စုံလင်လှသည်။ https://golang.org/pkg/fmt/#Println ကို သွား၍ ကျွန်တော်တို့ အသုံးပြုသည့် `Println` အကြောင်းကိုသာမက ၎င်းကို click နှိပ်၍ မည်သို့မည်ပုံရေးသားထားသည်ကိုပါ ဖတ်ရှုနိုင်သည်။ ထိုနောက် အပေါ်ဘက်သို့ scroll ပြုလုပ်၍ Go ၏ format လုပ်နိုင်စွမ်းကိုပါ လေ့လာနိုင်မည်ဖြစ်သည်။

အင်တာနက်မရပါက Documentation ကို offline အနေဖြင့် အောက်ပါ အတိုင်း run နိုင်မည်ဖြစ်သည်။

```
godoc -http=:6060
```

ထိုနောက် browser ပေါ်တွင် `http://localhost:6060` ဟု url ကိုခေါ်ကြည့်ပါ။

Variables နှင့်ကြေညာခြင်းများ

ဒီစာအုပ်မှာ *ဒီလိုမျိုး x=4 ဟု variable ကြေညာပြီး assign လုပ်လိုက်* ဟု များသာလွယ်လျှင် အလွန်ကောင်းမည်။ သို့သော် Go တွင်ထိုကိစ္စမှာ အနည်းငယ်ရှုပ်ထွေးသည်ဟု ဆိုရမည်။ မည်သို့မည်ပုံဆိုသည်ကို အောက်မှ ဥပမာများကို ဖတ်ရှုပြီးသိနိုင်ပါသည်။ နောက်အခန်းတွင်မူ Structure များ တည်ဆောက်အသုံးပြုမှု အပိုင်းကိုပါ ရေးသားသွားမည်ဖြစ်သည်။ သို့ပင် သော်ညား မိမိနှင့်ရင်းနှီးသွားရန် အချိန်တစ်ခုလိုအပ်ပါလိမ့်မည်။

သင့်အနေဖြင့် *ဘာလို့ ဒီလောက်တောင် ရှုပ်ထွေးတာလဲ* ဟု ထင်ကောင်း ထင်လိမ့်မည်။ ယခုတော့ ဥပမာ အနေဖြင့်ကြည့်ကြည့်ပါ။ Go တွင် အရိုး ရှင်းဆုံး variable ကြေညာခြင်းနှင့်သတ်မှတ်ခြင်းသည် အောက်ပါအတိုင်း ဖြစ်သည်။

```
package main

import (
  "fmt"
)

func main() {
  var power int
  power = 9000
  fmt.Printf("It's over %d\n", power)
}
```

၎င်းတွင် `power` ဟုသော variable ကို `int` အမျိုးအစားအဖြစ်သတ်မှတ် လိုက်သည်။ မူလအတိုင်းဆိုပါက go တွင် variable များကို နိတ္ထိ တန်ဖိုး များ သတ်မှတ်လေ့ရှိသည်။ Integer ဆိုပါက `0` ၊ boolean ဆိုပါက `false` ၊

strings ဆိုပါက "" စသဖြင့်စသဖြင့်။ ထိုကြောင့် နောက်တစ်ကြောင်းတွင် power ဟုသော variable တွင် တန်ဖိုး 9000 ကိုသတ်မှတ်လိုက်ခြင်း ဖြစ်သည်။ ထိုနှစ်ကြောင်းကို တစ်ကြောင်းထဲ အဖြစ် အောက်ပါ အတိုင်း ရေးသားနိုင်သည်။

```
var power int = 9000
```

သို့ပင်သော်ညား စာအများကြီးရိုက်နေရသေးသည်။ Go အပေါ်ကဲ့သို့သော ကြေညာသတ်မှတ်ချက်အတွက် type ကို infer ပြုလုပ်ပေးသော အတိုကောက် :=operator ရှိပါသည်။

```
power := 9000
```

အတော်ပင် အသုံးဝင်ပြီး function များဖြင့်လည်း အသုံးပြုနိုင်သည်။

```
func main() {
  power := getPower()
}

func getPower() int {
  return 9001
}
```

သတိပြုရမည်မှာ := သည် assign ပြုလုပ်ရုံသာမက declare ပြုလုပ် ရာတွင်လည်းပါဝင်သည်။ ထိုကြောင့် variable တစ်ခုကို နှစ်ခါမကြေညာ နိုင်သဖြင့် အောက်ပါအတိုင်း ရေးပါက error တက်မည်ဖြစ်သည်။

```
func main() {
  power := 9000
  fmt.Printf("It's over %d\n", power)
```

```
  // COMPILER ERROR:
  // no new variables on left side of :=
  power := 9001
  fmt.Printf("It's also over %d\n", power)
}
```

no new variables on left side of := ဟု compiler က အချက်ပေးမည် ဖြစ်သည်။ ဆိုလိုသည်မှာ ပထမတစ်ခါ variable ကြေညာပါက := ကိုသုံး နိုင်ပြီး ဒုတိယတစ်ခေါက်တွင်မူ = ကိုသာသုံးရမည်ဖြစ်သည်။ မှန်သည်က မှန်သော်လည်း အကျင့်ပါနေသဖြင့် မည်သည်ကို သုံးရမည် စဉ်းစားရ သည်မှာ နည်းနည်းတော့ တိုင်ပတ်သည်။

အပေါ်က error ကိုသေချာဖတ်ကြည့်ပါက *variables* ဟူသော အများကိန်း ကို ညွှန်းဆိုထားသည်ကိုတွေ့ရမည်။ အဘယ်ကြောင့်ဆိုသော ဌo တွင် variable များစွာကို = နှင့် := ကိုအသုံးပြုနိုင်သည်။

```
func main() {
  name, power := "Goku", 9000
  fmt.Printf("%s's power is over %d\n", name, power)
}
```

variable များအနက် တစ်ခုက အသစ်ဖြစ်ပါက := ကိုအသုံးပြုနိုင်သည်။ ဥပမာ

```
func main() {
  power := 1000
  fmt.Printf("default power is %d\n", power)

  name, power := "Goku", 9000
  fmt.Printf("%s's power is over %d\n", name, power)
}
```

```
}
```

`power` သည်နှစ်ခါသုံးထားသော်လည်း ဒုတိယတစ်ခါတွက် compiler မှ error မတက်ချေ။ အဘယ်ကြောင့်ဆိုသော `name` variable မှာမူ အသစ်ဖြစ်သောကြောင့် ခွင့်လွှတ်ထားခြင်းဖြစ်သည်။ သို့သော် `power` ၏ type ကိုမူ ပြောင်းလဲခွင့်မပေးပေ။ integer ဟု ကြေညာထားသောကြောင့် integer တန်ဖိုးသာ သတ်မှတ်ခွင့်ပေးမည်ဖြစ်သည်။

လောလောဆယ် နောက်ဆုံးတစ်ခုအနေဖြင့် သိထားသင့်သည်က import ကဲ့သို့ပင် Go အတွင်း အသုံးမပြုသော variable များကိုကြေညာခွင့်မပေးချေ။ ဥပမာ

```
func main() {
  name, power := "Goku", 1000
  fmt.Printf("default power is %d\n", power)
}
```

တွင် `name` variable သည် ကြေညာထားပြီး အသုံးမပြုသောကြောင့် compile ပြုလုပ်မည်မဟုတ်။ import ကဲ့သို့ပင် အာရုံနောက်သော်လည်း Code ၏ cleaniess နှင့် readability ကို အထောက်အကူပြုပါသည်။

ကြေညာခြင်းနှင့် သတ်မှတ်ခြင်းအတွက် သင်ယူရန် ကျန်ရှိသေးသော်လည်း ယခုတွင် သင်မှတ်ထားရန်လိုသည်မှာ နတ္ထိတန်ဖိုးအတွက် `var NAME TYPE` ဟု အသုံးပြုနိုင်ပြီး `NAME := VALUE` မှာ နှစ်ခုလုံးကို တပေါင်းတည်းပြုလုပ်

ကာ ကြေညာပြီးသော variable များအတွက်သာ `NAME = VALUE` ဟုသုံးနိုင်သည်။

Function ကြေညာခြင်းများ

Go တွင် function များသည် တခုထက်ပိုသော variable များကို return ပြန်နိုင်သည်။ အောက်က ဥပမာထဲရှိ function များဖြစ်သည့် return မပါသော ၊ return value တစ်ခုဖြင့် ၊ return value နှစ်ခုဖြင့် function ၃ခုကိုကြည့်ကြည့်ပါ။

```
func log(message string) {
}

func add(a int, b int) int {
}

func power(name string) (int, bool) {
}
```

နောက်ဆုံးတစ်ခုကို အောက်ပါအတိုင်းခေါ်နိုင်သည်။

```
value, exists := power("goku")
if exists == false {
  // handle this error case
}
```

တခါတရံ သင့်အနေဖြင့် return value တစ်ခုကိုသာလိုချင်သည့် အနေအထားမျိုးရှိပါက မလိုအပ်သော တစ်ခုကို _ ဟု assign ပြုလုပ်နိုင်သည်။

```
_, exists := power("goku")
if exists == false {
  // handle this error case
}
```

_ သည် ဗလာ identifier ဖြစ်ပြီး return value မယူရန် အထူးတည်ဆောက်ထားသဖြင့် ကြိမ်ဖန်များစွာ _ ကိုအသုံးပြုနိုင်သည်။

နောက်ဆုံးတွင် function ကြေညာခြင်းနှင့်ပတ်သက်၍ အောက်ပါအတိုင်း ပုံစံမျိုးမြင်တွေ့နိုင်သည်။ ပါဝင်သော parameter များသည် အမျိုးအစားတူညီပါက အတိုကောက်ပုံစံဖြင့် အသုံးပြုနိုင်သည်။

```
func add(a, b int) int {

}
```

value များစွာပြန်ခြင်းနှင့် _ ကိုအသုံးပြုရခြင်းသည် Go တွင်မကြာခဏ ကြုံတွေ့ရမည့် အရာများဖြစ်သည်။ return value များကို အမည်ပေးခြင်းနှင့် အနည်းငယ်လျော့ရဲသော parameter declaration များကိုမူ သိပ်ကြုံရမည်မဟုတ်။ သို့သော် အနှေးနှင့်အမြန် ကြုံရမည်ဖြစ်၍ သိထားရန်လိုသည်။

နောက်အခန်း မဖတ်ခင်

အခု ကြုံရသော အချက်များသည် သေးငယ်ပြီး အဆက်အစပ်မရှိ ဖြစ်ကောင်းဖြစ်သော်လည်း တဖြည်းဖြည်းနှင့် ပို၍ကြီးမားသော ဥပမာများကို တည်ဆောက်၍ တစ်ခုနှင့်တစ်ခု ဆက်စပ်မိမည်ဟု မျှော်လင့်မိပါသည်။

dynamic language ရေးသားသော နောက်ခံမှလာခဲ့ပါက type များအကြား ရှုပ်ထွေးမှုနှင့် ကြေညာခြင်းများသည် အနည်းငယ် အဟန့်အတားဖြစ်မည်ကို မငြင်းလို။ တချို့ system များအတွက် dynamic language များသည် အလုပ်ပိုဖြစ်လေ့ရှိသည်။

Static Type Language နောက်ခံမှ လာခဲ့သည် ဖြစ်သော သင့်အနေဖြင့် Go ကိုအသုံးပြုရသည်မှာ အဆင်ပြေပါလိမ့်မည်။ Go တစ်ခုတည်းတွင် ပါသည် မဟုတ်သော်လည်း type infer လုပ်၍ရခြင်းနှင့် return value အများပေးလို့ရခြင်းသည် မိုက်သည် ဟုဆိုရမည်။ သင်ယူနေရင်း သေသပ်လှပသည့် syntax ပုံစံကို ပို၍သဘောကျလာမိမည်ဟု မျှော်လင့်မိသည်။

အခန်း (၂) - Structure များ

Go သည် C++ ၊ Java ၊ Ruby နှင့် C# ကဲ့သို့ object-oriented (OO) language တစ်ခုမဟုတ်ပေ။ ၎င်းတွင် object ကော inheritance တို့မပါဝင်သဖြင့် OO နှင့်တွဲဖက်ပါရှိသော polymorphism နှင့် overloading concept များပါရှိမည် မဟုတ်ချေ။

Go တွင်ရှိသည်မှာ method များနှင့် ၎င်းတို့နှင့် တွဲစပ်နိုင်သော structure များသာရှိသည်။ ထိုအပြင် ရိုးရှင်းသော်လည်း အလုပ်ဖြစ်သည့် composition လည်းပါရှိသည်။ ရလာဒ်အနေဖြင့် ပို၍ ရိုးရှင်းသော code များ ရေးသားနိုင်မည် ဖြစ်သော်လည်း OO တွင်ပါသည်များကို တမ်းတကောင်း တမ်းတမိမည် ဖြစ်သည်။ အရင်ကတည်းက အကြီးအကျယ် အငြင်းပွားကြသည့် *composition over inheritance* အကြောင်းရှိသော်လည်း Go သည် ကျွန်တော်အသုံးပြုသော language များအနက် ပထမတစ်ခုကို အသားပေးသော langauage ဖြစ်သည်။

Go သည် သင်ကျွမ်းဝင်နေသော OO ကဲ့သို့ မဟုတ်သော်လည်း structure နှင့် class ကြေညာမှုများသည် ဆင်တူနေကို သတိထားမိမည် ဖြစ်သည်။ ၎င်းကို အောက်ပါ `Saiyan` structure တွင်တွေ့ရှိနိုင်ပါသည်။

```
type Saiyan struct {
  Name string
```

```
  Power int
}
```

ထိုနောက် ၎င်း structure ကို Class ကဲ့သို့ အသုံးပြုနိုင်အောင် method ဖြင့် မည်သို့ ချိတ်ဆက်မည်ကို များမကြာမီတွေ့ရမည် ဖြစ်ပြီး ယခုတွင် declaration ဘက်ကို ပြန်လှည့်ကြပါစို့။

Declarations and Initializations

variable နှင့် declaration များကို ပထမလေ့လာမိသလောက် တွေ့ရမည်မှာ မူလသတ်မှတ်ထားသော type များဖြစ်သည့် integers နှင့် string ကိုတွေ့ရ မည်ဖြစ်ပြီး ယခု structure များအကြောင်းလေ့လာရာတွင် pointer များ အကြောင်းပါ ဆက်စပ်ရှင်းပြရပါမည်။

structure တစ်ခုကို အလွယ်ဆုံးတည်ဆောက်နိုင်ရန်နည်းလမ်းမှာ

```
goku := Saiyan{
  Name: "Goku",
  Power: 9000,
}
```

သတိပြုရန် အထက်က ဥပမာတွင် , သည်မပါမဖြစ်ဖြစ်ပြီး မပါက compiler မှ error ပြမည်ဖြစ်သည်။ သင့်အနေဖြင့် ပါလိုက်မပါလိုက်ဖြစ်ပါ က ရသော language များနှင့်နှိုင်းစာရင် သဘောကျလိမ့်မည်မဟုတ်ပေ။ field အတွင်းရှိ value တစ်ခုမှမသတ်မှတ်သော်လည်း ရသည်။ ထို့ကြောင့် အောက်မှ ဥပမာ နှစ်ခုလုံး အလုပ်လုပ်မည် ဖြစ်သည်။

```
goku := Saiyan{}

// or

goku := Saiyan{Name: "Goku"}
goku.Power = 9000
```

field များသည်လည်း variable များကဲ့သို့ပင် assign မလုပ်ထားပါက နတ္ထိတိတန်ဖိုး သတ်မှတ်ထားသည်။ ထို့အပြင် field name များကို ကျော်၍ declaration order အတိုင်း assgin လုပ်နိုင်သော်လည်း ရှင်းရှင်းလင်းလင်း ဖြစ်စေရန် field အနည်းငယ်ဖြစ်မှသာ အောက်ပါအတိုင်း ပြုလုပ်သင့်သည်။

```
goku := Saiyan{"Goku", 9000}
```

အပေါ်မှ ဥပမာတွင် `goku` ဟုသော variable ကိုတည်ဆောက်ကာ assign ပြုလုပ်လိုက်ခြင်း ဖြစ်သည်။ အချို့န်တော်တော်များများတွင် ကျွန်တော်တို့၏ variable ကို value ဖြင့်တိုက်ရိုက် ချိတ်ဆက်ခြင်းပြုလုပ်သည်ထက် pointer တစ်ခုအနေဖြင့် ညွှန်းဆိုသည်က များသည်။ pointer သည် value တည်ရှိသော memory address ကိုညွှန်ပြပေးသည့် location ဖြစ်သည်။ အိမ်နှင့် အိမ်၏တည်နေရာပြသော မြေပုံ မတူညီသကဲ့သို့ပင်။

အဘယ်ကြောင့် တကယ့် value ထက်စာလျှင် value ကိုညွှန်ပြပေးသော point ကိုလိုအပ်သနည်း? Go သည် function သို့ argument အနေဖြင့် ပို့

လွှတ်သောအခါ copy အနေဖြင့်ပို့လွှတ်သောကြောင့်ဖြစ်သည်။ ထိုကြောင့် အောက်က code တွင်မည်သို့ ပေါ်မည်နည်း။

```
func main() {
  goku := Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s Saiyan) {
  s.Power += 10000
}
```

အဖြေမှာ 19000 မဟုတ်ဘဲ 9000 ဖြစ်မည်။ `Super` ဟူသော function သည် original `goku` variable မှတန်ဖိုးကိုယူသည်မဟုတ်ဘဲ copy ပွားယူသောကြောင့် `Super` မှ လုပ်ဆောင်သည် ၎င်း function ကိုခေါ်သူအတွက် အသက်ဝင်မည်မဟုတ်ပေ။ သင်လိုချင်သည့်အတိုင်း အလုပ်လုပ်စေလိုပါ pointer value ကိုသာ ပို့ပေးရန်လိုသည်။

```
func main() {
  goku := &Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s *Saiyan) {
  s.Power += 10000
}
```

အပေါ်နှင့်မတူသည်က ပြောင်းလဲမှု နှစ်ခုပြုလုပ်ထားသည်။ ပထမတစ်ခုက `&` ဟု operator ကိုအသုံးပြုကာ value ၏ address ကို လှမ်းယူလိုက်ခြင်း

ဖြစ်သည်။ ထိုနောက် `Super` မှ လက်ခံသော paramater ကို ပြောင်းလဲလိုက်သည်။ မူလ အစတွင် `Saiyan` အမျိုးအစား၏ value ကို မျှော်လင့်မည်ဖြစ်သော်လည်း ယခုတွင် `*Saiyan` ၏ address တစ်နည်းအားဖြင့် `*X` ဟု ဆိုလိုသည်မှာ *X အမျိုးအစား၏ pointer* ကိုဆိုလိုခြင်းဖြစ်သည်။ `Saiyan` နှင့် `*Saiyan` မှာ သေချာပေါက် ဆက်နွယ်သော်လည်း ကွဲပြားသော type ဖြစ်သည်။

သို့သော် `goku` value ကို copy ပြုလုပ်၍ ပို့ခြင်းပင်ဖြစ်သော်လည်း `Super` မှ လက်ခံရရှိသော value သည် address ဖြစ်သွားသောကြောင့်ဖြစ်သည်။ copy နှင့် မူရင်းသည် address အတူတူပင်ဖြစ်သောကြောင့် မူလတန်ဖိုးတွင် ပြောင်းလဲသွားခြင်းဖြစ်သည်။ ဥပမာ အနေဖြင့် စားသောက်ဆိုင် တစ်ဆိုင် လမ်းကြောင်း ဟု မှတ်ယူနိုင်သည်။ လက်ထဲတွင်ရှိသော value သည် မတူညီသော်လည်း စာသောက်ဆိုင်၏ နေရာကို ညွှန်ပြနေသည်ကတော့ အတူတူပင်ဖြစ်သည်။

၎င်းကို pointer နေရာကိုပြောင်းလဲကြည့်ခြင်းဖြင့် copy ပြုလုပ်ကြောင်း သိသာထင်ရှားစေနိုင်သည်။ (လက်တွေ့တွင်တော့ သင်လုပ်လိုသော ပုံစံမျိုး မဟုတ်ပါ)

```
func main() {
  goku := &Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}
```

```
func Super(s *Saiyan) {
  s = &Saiyan{"Gohan", 1000}
}
```

အပေါ်မှ ဥပမာတွင် 9000 ကိုသာ print လုပ်မည်ဖြစ်သည်။ ၎င်းသည် Ruby ၊ Python ၊ Java ၊ C# တို့နှင့်အတူတူပင်ဖြစ်ပြီး Go နှင့် C# ၏ အချို့အပိုင်းများတွင်သာ ထိုအချက်သည် သိသာစေနိုင်သည်။

point တစ်ခုကို သယ်ယူခြင်းသည် ရှုပ်ထွေးလှသော structure တစ်ခုလုံးကို သယ်ယူခြင်းနှင့် နှိုင်းစာလျှင် ပို၍ပေါ့ပါးသည်ဖြစ်သည်ကို သတိပြုရမည်ဖြစ်သည်။ 64bit စက်များတွင် pointer ၏ size သည် 64bit ဖြစ်သည်။ အကယ်၍ fields များစွာပါဝင်သော structure ဖြစ်ပါက copy ပြုလုပ်ခြင်းသည် မသက်သာလှပေ။ pointer မှတဆင့် တကယ်လက်ရှိ value ကိုညွှန်းဆို၍ အလုပ်လုပ်နိုင်ပြီ ဖြစ်၍ `Super` function မှ `goku` တစ်ခုလုံးကို copy ပြုလုပ်၍ပို့ ပြောင်းလဲပြီး ပြန်လက်ခံစရာကော လိုသေးပါရဲ့လား?

သို့သော် အမြဲတမ်း pointer သုံးရမည်ဟု ဆိုလိုခြင်းမဟုတ်ပေ။ ယခုအခန်း၏ အဆုံးသတ်တွင် structure နှင့် ဘာတွေလုပ်နိုင်မည်ကို သိရှိပြီး နောက်တွင်မူ pointer နှင့် value တို့အကြား အားပြိုင်ချက်ကို ထပ်၍ ပြန်ရှုပါဦးမည်။

Structure ပေါ်မှ function များ

method နှင့် structure ကိုအောက်ပါအတိုင်း တွဲစပ်နိုင်သည်။

```
type Saiyan struct {
  Name string
  Power int
}

func (s *Saiyan) Super() {
  s.Power += 10000
}
```

အပေါ်က ဥပမာတွင် `*Saiyan` ကို `Super` ၏ လက်ခံရရှိမည့်သူဖြစ်ကြောင်း ကြေညာလိုက်သည်။ ထိုကြောင့် `Super` ကိုအောက်ပါ အတိုင်း ခေါ်ယူနိုင်သည်။

```
goku := &Saiyan{"Goku", 9001}
goku.Super()
fmt.Println(goku.Power) // will print 19001
```

Constructors

Structure တွင် constructors များမပါဝင်ပါ။ ၎င်းအစား မိမိတို့လိုချင်သည့် အတိုင်း instance ကို return ပြန်ပေးသော function တစ်ခု တည်ဆောက်နိုင်ပါသည်။ (Factory pattern)

```
func NewSaiyan(name string, power int) *Saiyan {
  return &Saiyan{
    Name: name,
    Power: power,
  }
}
```

ထို pattern သည် developer အတော်များများအကြား စကားပြောစရာ ဖြစ်လာသည်။ တဖက်ကကြည့်ပါက စာသားပြောင်းလဲမှု အနည်းငယ် ကို တွေးစရာရှိသော်လည်း တစ်ဖက်တွင် ပို၍ စည်းစနစ်မကျသလို မှတ်ယူကြ သည်။ Factory တွင် pointer မပါသော်လည်း ရပါသည်။

```
func NewSaiyan(name string, power int) Saiyan {
  return Saiyan{
    Name: name,
    Power: power,
  }
}
```

New

Constructor မရှိသော်လည်း type များအတိုင်း memory တွင် သတ်မှတ် နေရာယူရန် Go တွင် built-in function ဖြစ်သော `new` ပါရှိသည်။ `new(X)` သည် `&X{}` နှင့်အတူတူပင်ဖြစ်သည်။

```
goku := new(Saiyan)
// same as
goku := &Saiyan{}
```

မိမိတို့နှစ်သက်ရာကို အသုံးပြုနိုင်သော်လည်း အတော်များများမှာ fields မှာ initialize ပြုလုပ်နိုင်ပြီး ဖတ်ရလွယ်ကူသည့် ဒုတိယနည်းလမ်းကို အသုံး များကြသည်။

```
goku := new(Saiyan)
goku.name = "goku"
goku.power = 9001
```

```
//vs

goku := &Saiyan {
  name: "goku",
  power: 9000,
}
```

မည်သည့်နည်းလမ်းကို ရွေးသည်ဖြစ်စေ အပေါ်မှ factory pattern ကို အသုံးပြုပါက သင့်အနေဖြင့် allocation ပတ်သက်သော သောကများကို ကာကွယ်ပေးနိုင်သည်။

Structure တစ်ခု၏ field များ

ဖော်ပြပြီးသော ဥပမာများအရ `Saiyan` တွင် `Name` နှင့် `Power` ဟူသော `strings` အမျိုးအစားနှင့် `int` အမျိုးအစား field နှစ်ခုရှိသည်ကိုတွေ့ရမည်။ field များသည် structure များ ၊ မပြောရသေးသော array များ၊ map များ interface များ ၊ function များ စသဖြင့် type အမျိုးစုံ ဖြစ်နိုင်သည်။

ဥပမာ `Saiyan` ၏ ဥပမာကို အောက်ပါအတိုင်း ထပ်၍ ဖြန့်ကျက်၍ ရပါသေးသည်။

```
type Saiyan struct {
  Name string
  Power int
  Father *Saiyan
}
```

၎င်းကို initialize လုပ်လိုပါက

```
gohan := &Saiyan{
  Name: "Gohan",
  Power: 1000,
  Father: &Saiyan {
    Name: "Goku",
    Power: 9001,
    Father: nil,
  },
}
```

Composition

Go တွင် structure တစ်ခုမှ အခြားတစ်ခုသို့ ထည့်သွင်းနိုင်သော composition ပါဝင်သည်။ တချို့ language များတွင် trait သို့မဟုတ် mixin ဟုခေါ်လေ့ရှိသည်။ composition ပြုလုပ်နိုင်ခြင်းမရှိသော language များတွင်မူ ခပ်ဆင်ဆင်ဖြစ်အောင် ပြုလုပ်၍ရသော်လည်း လွယ်တော့ မလွယ် ကူလှချေ။ Mixin မရှိသော Java တွင် *inheritance* ကိုအသုံးပြု၍ structure များကို extend ပြုလုပ်ရန် အောက်ပါ အတိုင်းရေးသား၍ရသည်။

```
public class Person {
  private String name;

  public String getName() {
    return this.name;
  }
}

public class Saiyan {
  // Saiyan is said to have a person
  private Person person;

  // we forward the call to person
  public String getName() {
    return this.person.getName();
```

```
  }
  ...
}
```

သို့သော် တခါတရံ အာရုံနောက်လာနိုင်ပေသည်။ `Person` တွင်ရှိသော method တိုင်းသည် `Saiyan` အတွက်ပါ ရှိနေရန် လိုအပ်နေမည်ဖြစ်သည်။ Go တွင် ထိုသို့သော ကိစ္စမျိုးကို အောက်ပါအတိုင်း

```
type Person struct {
  Name string
}

func (p *Person) Introduce() {
  fmt.Printf("Hi, I'm %s\n", p.Name)
}

type Saiyan struct {
  *Person
  Power int
}

// and to use it:
goku := &Saiyan{
  Person: &Person{"Goku"},
  Power: 9001,
}
goku.Introduce()
```

`Saiyan` structure တွင် `*Person` type ၏ field တစ်ခုပါရှိပြီး field တစ်ခု အနေဖြင့် အမည်ပေးထားခြင်းမရှိသောကြောင့် compose ပြုလုပ်ထားသော type အတွင်းရှိ fields နှင့် method များကို အသုံးပြုနိုင်သည်။ သို့သော် Go compiler အနေဖြင့် field name များ *အလိုအလျောက်* သတ်မှတ်ပေးထားသည်။ ထိုကြောင့် အောက်ပါအတိုင်း လှမ်းခေါ်နိုင်သည်။

```
goku := &Saiyan{
  Person: &Person{"Goku"},
}
fmt.Println(goku.Name)
fmt.Println(goku.Person.Name)
```

အပေါ်မှ နှစ်ခုလုံး “Goku” ဟု print ပြုလုပ်မည်ဖြစ်သည်။ Composition က Inheritance ထက်ပိုကောင်းသလား။ လူအတော်များများကတော့ code ကို ပြန်လည်အသုံးပြုရာတွင် ပို၍ ခိုင်မာသည်ဟု ယူဆကြသည်။ inheritance ကိုအသုံးပြုပါက သင့်၏ class သည် parent class ဖြင့် တင်းကျပ်စွာ ချည်နှောင်ထားသလိုဖြစ်၍ behavior နှင့်နှိုင်းစာလျှင် hierarchy ဘက်ကို ပို၍ အလေးစိုက်ရဖွယ်ဖြစ်သည်။

Overloading

Overloading ကို structure များတွင်သာ အသုံးပြုနိုင်သည် မဟုတ်သော်လည်း ၎င်းကိုရှင်းပြရန်လိုအပ်သည်။ Go တွင် overloading ကို support မလုပ်ပါ။ ထို့အတွက်အကြောင့် `Load` I `LoadById` I `LoadByName` စ သဖြင့် ပုံစံများကို ရေးရ၊ မြင်ရမည် ဖြစ်သည်။

သို့သော် တိုက်ရိုက် composition မှာ compiler ၏လှည့်ကွက် တစ်ခုဖြစ်ပြီး function တစ်ခု၏ compose type ကို overwrite ပြုလုပ်၍ရနိုင်သည်။ ဥပမာ `Saiyan` structure တွင်း ၎င်းကိုယ်ပိုင် `Introduce` function ရှိပါက

```
func (s *Saiyan) Introduce() {
  fmt.Printf("Hi, I'm %s. Ya!\n", s.Name)
}
```

composed version အနေဖြင့် `s.Person.Introduce()` ဟူ၍လှမ်းခေါ်နိုင်သည်။

Pointers versus Values

Go ကိုရေးနေရင်း *value သုံးရမလား pointer သုံးရမလား* ဆိုတာ ဇဝေဇဝါဖြစ်တဲ့သူတွေအတွက် သတင်းကောင်းနှစ်ခု ရှိပါတယ်။ ပထမတစ်ခုကတော့ အောက်ပါ အခြေအနေများအတွက် ဘယ်ဟာသုံးသုံး အတူတူပါပဲ။

- local variable သတ်မှတ်ခြင်း
- structure ထဲမှ field
- function တစ်ခုမှ return ပြန်သည့်တန်ဖိုး
- function တစ်ခုသို့ parameter
- method တစ်ခုသို့ လက်ခံရယူခြင်း

ဒုတိယ တစ်ခုအနေနဲ့ကတော့ မသေချာရင် pointer သာသုံးပါ။ မြင်ခဲ့သည့်အတိုင်း value ကို pass ခြင်းဖြင့် data ကို immutable ပြုလုပ်ရာတွင် ကောင်းမွန်သလို တခါတလေကြရင် ထိုအခြေအနေမျိုး လိုအပ်မည်ဖြစ်သော်လည်း များသောအားဖြင့် မလိုလှပေ။

data ကိုပြောင်းလဲလိုခြင်းမရှိသော်လည်း large structure တစ်ခုကို copy ပြုလုပ်ပါက ကျသင့်သည့်တန်ဖိုးကို စဉ်းစားသင့်ပါသည်။ ထိုနည်းတူ အောက်ပါကဲ့သို့ သေးငယ်သော structure များအတွက်မူ

```
type Point struct {
  X int
  Y int
}
```

copy လုပ်သောအခါ ကုန်သည့်တန်ဖိုးထက်နှိုင်းစာလျှင် `x` နှင့် `y` တို့ကို တိုက်ရိုက် access ပြုလုပ်နိုင်ခြင်းက ပို၍ အားသာပါလိမ့်မည်။

ထပ်၍ ပြောရပါလျှင် ၎င်းတို့သည် သိမ်မွေ့သော ပြဿနာဖြစ်သည်။ သင့်အနေဖြင့် ထိုသို့ကဲ့သို့ ထောင်သောင်းချီသော point လုပ်စရာမလိုပါက ကွဲပြားခြားနားချက်ကို သတိထားမိမည် မဟုတ်ပေ။

နောက်အခန်း မဖတ်ခင်

ယခုအခန်းတွင် structure များအကြောင်း မိတ်ဆက်ပေးခဲ့ပြီး function တစ်ခုလက်ခံနိုင်သော structure တစ်ခု မည်သို့ တည်ဆောက်မည်နည်း ၊ Go ၏ type စနစ်တွင်ပါဝင်သည့် pointer များနှင့်ပတ်သတ်၍ ထပ်ဆင့်လေ့လာပြီးဖြစ်၍ နောက် အခန်းတွင်မူ stucture များအချင်းချင်း မည်သို့ အလုပ်လုပ်သည်ကို ဆက်၍လေ့လာသွားမည်ဖြစ်သည်။

အခန်း (၃) - Maps ၊ Arrays နှင့် Slices

အခုထက်ထိတော့ ပုံမှန် type နဲ့ structure များအကြောင်းလေ့လာပြီး နောက် ဒီအခန်းမှာတော့ array များ slice များနှင့် map များအကြောင်းကို ရှင်းပြသွားမှာဖြစ်ပါတယ်။

Arrays

Python ၊ Ruby ၊ Perl ၊ Javascript တို့ PHP ကဲ့သို့သော language များကို လေ့လာခဲ့ပါက *dynamic array* များနှင့် ရင်းနှီးနေမည်ဖြစ်သည်။ ထို array များသည် data များထည့်သွင်းသည်နှင့်အမျှ ပြန်လည် ချိန်ညှိပေးသည်။ Go တွင်မူ အပေါ်မှ အပ အခြား language များကဲ့သို့ array မှာ အသေ ဖြစ်သည်။ array ကိုကြေညာသည်နှင့် ဆိုဒ်မှာ အသေဖြစ်ပြီး တိုး၍မရ တော့ပေ။

```
var scores [10]int
scores[0] = 339
```

အထက်တွင်ဖော်ပြထားသော array တွင် `score[0]` မှ `score[9]` အထိ index ဆယ်ခုစာ ထည့်သွင်းနိုင်သည်။ ထိုမှအပ အခြား index များကိုခေါ်ပါက compiler သို့မဟုတ် runtime error တက်လိမ့်မည်။ Array ကို value များပါတပါတည်း အောက်ပါအတိုင်း ကြေညာနိုင်သည်။

```
scores := [4]int{9001, 9333, 212, 33}
```

`len` ကိုအသုံးပြု၍ array ၏ အရှည်ကိုသိနိုင်သလို `range` ကိုအသုံးပြု၍ loop ပတ်နိုင်သည်။

```
for index, value := range scores {

}
```

Array မှာ အလွန်အသုံးဝင်သောလည်း တင်းကျပ်သည်။ ပုံမှန်အားဖြင့် မိမိတို့ လိုအပ်သည့် element အရေအတွက်ကို မသိရှိနိုင်ချိန်တွင် Array အစား Slice ကိုအသုံးပြုသည်။

Slices

Go တွင် array ကိုတိုက်ရိုက် အသုံးပြုရသည်က ရှားသည်။ ထိုအစား slice ကိုအသုံးပြုကြသည်။ slice သည် array ပုံစံအတိုင်း ဖော်ပြနိုင်သည့် structure တစ်ခုဖြစ်သည်။ slice ကိုတည်ဆောက်နိုင်ရန် နည်းလမ်းမှာ တစ်ခုထက်မကရှိပြီး မည့်သည်အချိန်တွင် သုံးရမည်ကို နောက်တွင် ဆက်ပြောပါမည်။ ရှေးဦးစွာ ဖန်တီးသည့် နည်းလမ်းကွဲများ ကိုအရင် ရှင်းပြပါမည်။

```
scores := []int{1,4,293,4,9}
```

Array နှင့်မတူသည်က လေးထောင့်ကွင်းထဲ ၎င်း၏ အရှည်ကို မသတ်မှတ်ပေးရပါ။ ကွဲပြားခြားနားသည်ကို သိနိုင်ရင် slice ကိုဖန်တီးနိုင်သည့် နောက်တစ်မျိုးဖြစ်သည့် `make` ကိုအသုံးပြုကြပါစို့။

```
scores := make([]int, 10)
```

`new` အစား `make` ကိုအသုံးပြုရသည့်အကြောင်းမှာ memory တွင် နေရာယူခြင်း သက်သက်သာမဟုတ်ပဲ slice တစ်ခုကို အမှန်တကယ် ဖန်တီးလိုက်ခြင်းကြောင့်ဖြစ်သည်။ ထိုကြောင့် array အတွက် memory နေရာယူရုံသာမက slice အတွက်ပါ ကြေညာလိုက်ခြင်းဖြစ်သည်။ အပေါ်မှ ဥပမာတွင် slice ၏ အရှည်မှာ တစ်ဆယ် ဖြစ်ပြီးထည့်သွင်းနိုင်စွမ်းမှာလည်း ဆယ်ခု ဖြစ်သည်။ ထိုနှစ်ခုမှာ မတူညီပါ။ `make` ကိုအသုံးပြု၍ ၎င်းနှစ်ခုကို သီးခြားစီ ထည့်သွင်းနိုင်သည်။

```
scores := make([]int, 0, 10)
```

၎င်းတွင် slice ၏ အရှည်မှာ သုညဖြစ်ပြီး ထည့်သွင်းနိုင်စွမ်းမှာ တစ်ဆယ် ဖြစ်သည်။ သတိထားကြည့်ပါက `make` နှင့် `len` များသည် overload ပြုလုပ်ထားသည်ကို သတိထားမိမည်ဖြစ်သည်။ Go သည် developer များအသုံးပြုနိုင်အောင် ထုတ်မပေးသောလည်း language designer များက အသုံးပြုနေသည်ကိုတွေ့နေရသောကြောင့် တချို့မှာ အမြင်မကြည်ကြပေ။

အရှည်နှင့် ထည့်သွင်းနိုင်စွမ်းကို ကွဲပြားစေရန် အောက်ပါ ဥပမာကို ကြည့်ပါ။

```
func main() {
  scores := make([]int, 0, 10)
  scores[7] = 9033
  fmt.Println(scores)
}
```

Crash ဖြစ်ပါလိမ့်မည်။ အဘယ်ကြောင့်ဆိုသော် slice ၏ အရှည်မှာ 0 ဖြစ်သောကြောင့်ဖြစ်သည်။ ၎င်း၏ လက်အောက်ခံ array မှာ ဆယ်ခုရှိမည်ဖြစ်သော်လည်း ထို element များကိုခေါ်ယူအသုံးပြုနိုင်ရန် slice ကိုတိုးချဲ့မှုရမည်။ slice ကို ချဲ့နိုင်မည့် နည်းလမ်း တစ်ခုမှာ `append` ကိုအသုံးပြုခြင်းဖြစ်သည်။

```
func main() {
  scores := make([]int, 0, 10)
  scores = append(scores, 5)
  fmt.Println(scores) // prints [5]
}
```

သို့သော် ထိုသို့ပြောင်းလိုက်ခြင်းဖြင့် မူလ code ၏ရည်ရွယ်ချက် ပါပြောင်းသွားမည်ဖြစ်ပြီး အရှည် 0 ရှိသော slice ၏ နောက်မှ ထပ်ထည့်သည်ထက် အခန်း နံပါတ် (၇) တွင်ထည့်လိုသည်ဖြစ်၍ slice ကိုထပ်၍ ပိုင်းနိုင်ပါသည်။

```
func main() {
  scores := make([]int, 0, 10)
  scores = scores[0:8]
  scores[7] = 9033
  fmt.Println(scores)
}
```

slice တစ်ခုကို ဘယ်လောက်ပြန်၍ resize လုပ်နိုင်ပါသလဲ။ ယခုကိစ္စတွင် တစ်ဆယ် ဖြစ်သည်။ သင့်အနေဖြင့် ယခုအတိုင်းလုပ်ခြင်းဖြင့် *array များကဲ့သို့ အသေဖြစ်နေသည့် ပြဿနာမှာ သိပ်ပြီးမထူး* ဟု ထင်ကောင်ထင်လိမ့်မည်။ `append` တွင်ထူးခြားသည်က array ပြည့်နေပါက ပို၍ကြီးမားသော array တစ်ခုတည်ဆောက်ပြီး value များကိုပါ

သယ်ဆောင်ပေးမည်ဖြစ်သည်။ (၎င်းပုံစံ PHP ၊ Python ၊ Ruby ၊ Javascript တွင် dynamic array များ အလုပ်လုပ်ပုံနှင့် အတူတူပင် ဖြစ်သည်။) ထို့ကြောင့် `append` မှ ပြန်လာသော တန်ဖိုးကို လက်ခံနိုင်ပြီး assign ပြန်လုပ်ပေးနိုင်သည်။ `score` variable တွင်ရှိသော array မှာ original မှနေရာမရှိတော့ပါက copy ကူးယူထားသော အသစ်တန်ဖိုး ဖြစ်သည်။

အကယ်၍ Go ၏ array များသည် ဆတူတိုးသွားသည်ဆိုပါက အောက်ပါ code သည် မည်သို့ဖော်ပြမည်နည်း?

```
func main() {
  scores := make([]int, 0, 5)
  c := cap(scores)
  fmt.Println(c)

  for i := 0; i < 25; i++ {
    scores = append(scores, i)

    // if our capacity has changed,
    // Go had to grow our array to accommodate the new data
    if cap(scores) != c {
      c = cap(scores)
      fmt.Println(c)
    }
  }
}
```

ပထမဆုံး `score` ၏ သိမ်းဆည်းနိုင်သော ပမာဏသည် ငါးခုဖြစ်မည်။ နှစ်ဆယ့်ငါးခုကို သိမ်းဆည်နိုင်ရန် ဆယ်ခု ၊ အခုနှစ်ဆယ် နှင့် နောက်ဆုံး အခု

လေးဆယ် အထိ ဆတူ တိုးသွားမည်ဖြစ်သည်။ နောက်ဆုံး ဥပမာ အနေဖြင့် အောက်က code ကိုကြည့်ပါ။

```
func main() {
  scores := make([]int, 5)
  scores = append(scores, 9332)
  fmt.Println(scores)
}
```

`[0, 0, 0, 0, 0, 9332]` ဟုထွက်မည်ဖြစ်သည်။ သင့်အနေဖြင့် `[9332, 0, 0, 0, 0]` ဟုလိုချင်ကောင်းလိုချင်လိမ့်မည်။ လူအတွက်တော့ ဟုတ်ကောင်းဟုတ်မည်ဖြစ်သော်လည်း တန်ဖိုးများရှိပြီးသော slice အတွက် compile သည်ထိုသို့ ပြုမူပါလိမ့်မည်။

slice တစ်ခုကိုတည်ဆောက်ရန် နည်းလမ်းလေးမျိုးရှိသည်။

```
names := []string{"leto", "jessica", "paul"}
checks := make([]bool, 10)
var names []string
scores := make([]int, 0, 20)
```

ဘယ်အချိန်မှာ ဘယ်ဟာကိုသုံးရမလဲ? ပထမတစ်ခုအနေဖြင့် သိပ်ရှင်းရန်မလိုပါ။ value များကိုကြိုသိချိန်တွင်သုံးနိုင်သည်။

ဒုတိယနည်းလမ်း ကို index များကို တန်ဖိုးထည့်သွင်းရာအသုံးပြုနိုင်သည်။ ဥပမာ

```
func extractPowers(saiyans []*Saiyan) []int {
  powers := make([]int, len(saiyans))
  for index, saiyan := range saiyans {
```

```
    powers[index] = saiyan.Power
  }
  return powers
}
```

တတိယ အမျိုးအစားကိုမူ element မည်မျှပါဝင်သည်ကို မသိလောက်သည့် အနေအထားတွင် နတ္တိတန်ဖိုးများချည်းသာပါဝင်သည့် slice ကို တည်ဆောက်ပြီး `append` ဖြင့်ဖြည့်သွားမည့် ပုံစံဖြင့်သုံးသည်။

နောက်ဆုံး အမျိုးအစားကိုမူ အကြမ်းအားဖြင့် element မည်မျှရှိသည်ကို မှန်းဆနိုင်ပါက မူလတန်ဖိုး ထည့်သွင်းရာတွင် အသုံးဝင်သည်။

ပမာဏကို သိသည့်အခါတွင်လည်း `append` ကိုအသုံးပြုနိုင်သည်။ မိမိတို့နှစ်သက်ရာပုံစံသာ ကွဲလေ၏။

```
func extractPowers(saiyans []*Saiyan) []int {
  powers := make([]int, 0, len(saiyans))
  for _, saiyan := range saiyans {
    powers = append(powers, saiyan.Power)
  }
  return powers
}
```

Slice သည် array များ အပေါ်မှ ဖုံးအုပ်ထားသော အလွှာတစ်ခုဖြစ်ပြီး အလွန်အသုံးဝင်သည်။ Language တော်တော်များများတွင် array ကို ပိုင်းဖြတ်သည့် concept များပါရှိသည်။ Javascript နှင် Ruby တွင် array များသည် `slice` method ပါရှိသည်။ Ruby တွင် `[START..END]` ဟု သော်လည်းကောင်း Python တွင် `[START:END]` ဟုလည်းကောင်း

အသုံးပြု၍ ပိုင်းဖြတ်နိုင်သည်။ သို့သော် ထို language များတွင် slice သည် မူလတန်ဖိုးများ ကူးယူထားသော array အသစ်တစ်ခုသာဖြစ်သည်။ Ruby တွင် အောက်ပါ code ၏ ရလဒ်မှာ ဘာဖြစ်မည်နည်း။

```
scores = [1,2,3,4,5]
slice = scores[2..4]
slice[0] = 999
puts scores
```

အဖြေမှာ `[1, 2, 3, 4, 5]` ဖြစ်မည်။ အဘယ်ကြောင့်ဆိုသော် `slice` သည် array အသစ်အနေဖြင့် copy ကူးသွား၍ ဖြစ်သည်။ Go မှာထိုကဲ့သို့ပြုလုပ်ပါက

```
scores := []int{1,2,3,4,5}
slice := scores[2:4]
slice[0] = 999
fmt.Println(scores)
```

အဖြေမှာ `[1, 2, 999, 4, 5]` ဖြစ်မည်။

ထိုပြောင်းလဲခြင်းက ကုဒ်ရေးသည့်ပုံစံပါ ပြောင်းလဲသွားသည်။ ဥပမာ function များသို့ parameter ပို့သောအခါ။ javascript တွင် string တစ်ခု၏ အကွာရာ ငါးခု ကျော်ပြီးနောက် ပထမ နေရာကို လိုပါက (slice သည် string များတွင်လည်း အလုပ်လုပ်သည်) အောက်ပါအတိုင်းရေးနိုင်သည်။

```
haystack = "the spice must flow";
console.log(haystack.indexOf(" ", 5));
```

Go တွင် slice ကိုအောက်ပါ အတိုင်းသုံးနိုင်သည်။

```
strings.Index(haystack[5:], " ")
```

အထက်ဖော်ပြပါ ဥပမာများမှ `[X:]` သည် *X မှစ၍ အဆုံးထိ* ဟု အဓိပ္ပါယ်ရပြီး `[:X]` သည် *အစမှ X အထိသို့* ဟု အဓိပ္ပါယ် သက်ရောက်သည်။ တခြား language များနှင့်ကွာခြားသည်က Go တွင် အနုတ်ကိန်းများကို support မလုပ်ချေ။ အကယ်၍ နောက်ဆုံးတစ်ခုလွဲ၍ ကျန်သော value များကို အလိုရှိပါက အောက်ပါအတိုင်းရေးရသည်။

```
scores := []int{1, 2, 3, 4, 5}
scores = scores[:len(scores)-1]
```

အပေါ်မှ ဥပမာသည် unsorted slice များ value များကို ဖယ်ထုတ်ရာတွင် အသက်သာဆုံး နည်းလမ်းဖြစ်သည်။

```
func main() {
  scores := []int{1, 2, 3, 4, 5}
  scores = removeAtIndex(scores, 2)
  fmt.Println(scores) // [1 2 5 4]
}

// won't preserve order
func removeAtIndex(source []int, index int) []int {
  lastIndex := len(source) - 1
  //swap the last value and the value we want to remove
  source[index], source[lastIndex] = source[lastIndex], source[i
  return source[:lastIndex]
}
```

နောက်ဆုံးတွင် slice များအကြောင်း သိရှိပြီး ဖြစ်၍ မူလကတည်းက ပါဝင်သော function တစ်ခုဖြစ်သည့် `copy` ကိုကြည့်ပါစို့။ `copy` သည် slice များကြောင့် ကုဒ်ရေးရပုံ အပြောင်းအလဲကို မီးမောင်းထိုးပြနိုင်သည့် ဥပမာ တစ်

ခုဖြစ်သည်။ ပုံမှန်အားဖြင့် array တစ်ခုမှ နောက်တစ်ခုသို့ copy ပြုလုပ်ရန် parameter ငါးခုလိုအပ်သည်။ ၎င်းတို့မှာ `source` ၊ `sourceStart`၊ `count` ၊ `destination` နှင့် `destinationStart` တို့ဖြစ်သည်။ Slice ကိုအသုံးပြုပါက နှစ်ခုသာလိုအပ်သည်။

```
import (
  "fmt"
  "math/rand"
  "sort"
)

func main() {
  scores := make([]int, 100)
  for i := 0; i < 100; i++ {
    scores[i] = int(rand.Int31n(1000))
  }
  sort.Ints(scores)

  worst := make([]int, 5)
  copy(worst, scores[:5])
  fmt.Println(worst)
}
```

အချိန်ခဏယူ၍ အထက်ပါ code ကိုနားလည်အောင် ကြည့်ပါ။ တစ်ချို့ အပိုင်းများကို ပြောင်းလဲကြည့်ပါ။ copy ကို `copy(worst[2:4], scores[:5])` ပြောင်းကြည့်ပါက ဘာဖြစ်မည်နည်း။ value ငါးခုထက် နည်းသော တန်ဖိုးများကို `worst` သို့ ကူးကြည့်ပါက ဘာဖြစ်မည်နည်း။

Maps

Go တွင်ပါရှိသော Map သည် အခြား language များတွင် hashtable သို့မဟုတ် dictionary ဟုခေါ်လေ့ရှိပြီး အလုပ်လုပ်ပုံမှာ အတူတူပင် ဖြစ်သည်။ key value မှာ သတ်မှတ် ၊ ခေါ်ဆို ၊ ဖျက်ပစ်ရာတွင် အသုံးပြု သည်။

Maps များသည် slice ကဲ့သို့ပင် `make` function ကိုအသုံးပြုနိုင်သည်။ အောက်က ဥပမာကို ကြည့်ကြည့်ပါ။

```
func main() {
  lookup := make(map[string]int)
  lookup["goku"] = 9001
  power, exists := lookup["vegeta"]

  // prints 0, false
  // 0 is the default value for an integer
  fmt.Println(power, exists)
}
```

key အရေအတွက်ကိုသိလိုပါက `len` ကိုအသုံးပြုနိုင်ပြီး key တစ်ခုကိုဖျက် လိုပါက `delete` ကိုအသုံးပြုနိုင်သည်။

```
// returns 1
total := len(lookup)

// has no return, can be called on a non-existing key
delete(lookup, "goku")
```

Map သည် အကန့်အသတ်မရှိသောလည်း သတ်မှတ်လိုပါက `make` ကို အသုံးပြုနိုင်သည်။

```
lookup := make(map[string]int, 100)
```

Map တွင် key မည်မျှရှိလောက်မည်ကို ခန့်မှန်းနိုင်ပါက size သတ်မှတ်ခြင်းဖြင့် performance ပိုကောင်းလာမည်ဖြစ်သည်။

structure တစ်ခု၏ field အနေဖြင့်သတ်မှတ်လိုပါက အောက်ပါအတိုင်း ကြေညာနိုင်သည်။

```
type Saiyan struct {
  Name string
  Friends map[string]*Saiyan
}
```

အပေါ်မှ ကြေညာထားသည်ကို အောက်ပါအတိုင်း ခေါ်ယူအသုံးပြုနိုင်သည်။

```
goku := &Saiyan{
  Name: "Goku",
  Friends: make(map[string]*Saiyan),
}
goku.Friends["krillin"] = ... //todo load or create Krillin
```

Array များနည်းတူ Go တွင် map များကို `make` ကိုအသုံးပြုနိုင်သလို literal အနေဖြင့်လည်း ကြေညာသတ်မှတ်နိုင်သည်။

```
lookup := map[string]int{
  "goku": 9001,
  "gohan": 2044,
}
```

ထိုနည်းတူ `for` loop များတွင်လည်း `range` ကိုအသုံးပြုနိုင်သည်။

```
for key, value := range lookup {
  ...
```

```
}
```

Map အတွင်းရှိ iteration များသည် စီထားသည် မဟုတ်ပဲ random အလိုက် key value pair များကြလာမည်ဖြစ်သည်။

Pointers versus Values

အခန်း (၂) တွင် pointer နှင့် value များကွဲပြားချက်ကို လေ့လာပြီး ဖြစ်သည်။ ယခု array နှင့် map များအပေါ်တွင် ဆက်၍လေ့လာစရာရှိပါသေးသည်။ အောက်ပါ ပုံစံနှစ်ခု အနက် မည်သည်ကို ပို၍ အသုံးပြုသင့်သနည်း။

```
a := make([]Saiyan, 10)
//or
b := make([]*Saiyan, 10)
```

developer အတော်များများသည် `b` သည်ပို၍ efficient ဖြစ်မည်ဟုထင်ကြလိမ့်မည်။ သို့သော် return ပြန်လာသော value သည် reference ဖြစ်စေကာမူ slice ၏ copy လုပ်ထားသော value ပင်ဖြစ်ဦးမည်။ ထိုကြောင့် slice ကို pass နှင့် return ပြန်ရာတွင် ကွာခြားမှုမရှိပါ။

သို့သော် slice သို့မဟုတ် map ၏ value များကို ပြင်ပါက ကွာခြားချက်များကိုတွေ့ရမည်ဖြစ်သည်။ ထိုအချိန်တွင်မူ အခန်း (၂) တွင်ပြောသကဲ့သို့ ဖြစ်သည်။ ထိုကြောင့် array နှင့် map တို့၏ pointer နှင့် value ကွာခြား

ချက်သည် ၎င်းတို့၏ value များအသုံးပြုပုံကိုသာ အဓိက မူတည်နေမည် ဖြစ်သည်။

နောက်အခန်း မဖတ်ခင်

Go ၏ Array နှင့် Map တို့သည် အခြား language များနှင့်အတူတူပင် ဖြစ်သည်။ dyanmic array များနှင့် အသားကျနေပါက အနည်းငယ် ပြောင်းလဲမည် ဖြစ်သော်လည်း `append` ကိုအသုံးပြုခြင်းဖြင့် အဆင်မပြေမှု အတော်များများ ဖြေရှင်းနိုင်သည်။ Array များကို ကျော်လွန်ပါက slice များကိုတွေ့နိုင်မည်ဖြစ်ပြီး slice များသည် အလွန် အစွမ်းထက်ပြီး သင့် code ၏ ရိုရှင်းမှုကို ထိန်းသိမ်းပေးပါသည်။

သို့သော် အစွန်းထွက်နေသော ကိစ္စတချို့ကိုမူ ဒီစာအုပ်တွင် ထည့်သွင်းထားခြင်းမရှိပါ။ သင့်အနေဖြင့်လည်း ကြုံနိုင်ရန် ခဲယဉ်းသဖြင့် ဖြစ်သည်။ အကယ်၍ ကြုံခဲ့ပါက ယခု အခြေခံများကို အသုံးပြု၍ ဖြေရှင်းနိုင်မည် ဟု ယုံကြည်ပါသည်။

အခန်း (၄) - Code အစီအစဉ်ချခြင်းနှင့် Interface များ

ယခု အခန်းတွင် Code များကို မည်သို့ နေရာချထားမည်ကို လေ့လာပါမည်။

Packages

ရှုပ်ထွေးလှသော library များ၊ system များကို အစီအစဉ်ချနိုင်ရန် Packages များအကြောင်း လေ့လာရန်လိုပါသည်။ Go တွင် package name သည် Go Workspace အတွင်းတည်ရှိသည့် folder အမည် ကိုလိုက်နာရ့ါသည်။ အကယ်၍ shopping system တစ်ခုကို တည်ဆောက်ပါက သင့်အနေဖြင့် “shopping” ဟုသော package တစ်ခုဟု အမည်ပေးလိုက်ပြီး file များကို `$GOPATH/src/shopping/` တွင်နေရာချထားရမည် ဖြစ်သည်။

အာလုံးကိုတော့ folder တစ်ခုထဲတွင် ထားချင်မည်တော့ မဟုတ်။ ဥပမာ တချို့ သော database logic များကို folder အနေဖြင့် သက်သက် ခွဲထုတ်ရန်လိုအပ်သည်။ ၎င်းအတွက် ထပ်ဆင့် folder တစ်ခုအနေဖြင့် `$GOPATH/src/shopping/db` ဟုသတ်မှတ်ပေးရန်လိုအပ်သည်။ ထို folder ၏ အမည်သည် `db` ဟုပေးလျှင်ရသော်လည်း တခြား package မှ access

လုပ်လိုပါက `shopping` package အတွင်းမှ ဖြစ်၍ `shopping/db` ဟု import လုပ်ရန်လိုသည်။

တနည်းအားဖြင့် package တစ်ခုကို `package` keyword ဖြင့် အမည်ပေးနိုင်သည်။ single value အနေဖြင့် ဥပမာ “shopping” ဝါ “db” ဟု ပေးနိုင်ပြီး import လုပ်ပါက path လမ်းကြောင်းအတိုင်း import လုပ်ရမည်ဖြစ်သည်။

စမ်းကြည့်ရအောင်။ Go workplace အတွင်းရှိ `src` folder အတွင်းမှ (ကျွန်တော်တို့ မိတ်ဆက်တုံးက စမ်းထားခဲ့သော နေရာများ) `shopping` ဟု အမည်ရှိသော folder နှင့် ၎င်းအတွင်းတွင် `db` ဟုအမည်ရှိသော folder တစ်ခုကို တည်ဆောက်လိုက်ပါ။

`shopping/db` ၏အတွင်းတွင် `db.go` ဟုသော file တစ်ခုကို တည်ဆောက်ပြီး အောက်ပါအတိုင်း ရေးလိုက်ပါ။

```
package db

type Item struct {
  Price float64
}

func LoadItem(id int) *Item {
  return &Item{
    Price: 9.001,
  }
}
```

Folder ၏ အမည်သည် package ၏ အမည်နှင့် တူနေသည်ကို သတိထား မိပါလိမ့်မည်။ ဥပမာ အနေနဲ့ဖြစ်၍ database ကို access လုပ်သော code ကိုရေးမနေတော့ပါ။ Code ကို မည်သို့ organize လုပ်ရမည်နည်းဆိုသည့် ဥပမာကိုသာ ပြလိုသောကြောင့် ဖြစ်သည်။

ထိုနောက် `shopping` ဟုသော folder အတွင်းတွင် `pricecheck.go` ဟုသော file တစ်ခုဆောက်ပြီး အောက်ပါအတိုင်းရေးလိုက်ပါ။

```
package shopping

import (
  "shopping/db"
)

func PriceCheck(itemId int) (float64, bool) {
  item := db.LoadItem(itemId)
  if item == nil {
    return 0, false
  }
  return item.Price, true
}
```

`shopping` ဟုသော package/folder အတွင်းတွင်တည်ရှိနေသဖြင့် `shopping/db` ကို import လုပ်ရမည့်ဟု ထင်ကောင်ထင်လိမ့်မည်။ လက်တွေ့တွင် သင်သည် `$GOPATH/src/shopping/db` ဟု import ပြုလုပ်လိုက်ခြင်း ဖြစ်၍ folder အမည် `src/test` ၏ အတွင်းတွင်ရှိသောcode ကို `test/db` ဟု import ဟုပြုလုပ်ရမည်။

ယခု သိထားသော အချက်အလက်များဖြင့် Package တစ်ခုကို တည်ဆောက်နိုင်မည် ဖြစ်သည်။ Executable တစ်ခုကို build ပြုလုပ်နိုင်ရန် `main` တစ်ခုလိုအပ်မည်ဖြစ်ပြီး ထိုကြောင့် `shopping` ဟုသော folder အတွင်းမှ `main` ဟု ထပ်ဆင့် folder တစ်ခုတည်ဆောက်ပြီး `main.go` ဟု file တစ်ခုတည်ဆောက်လိုက်ပြီး အောက်ပါအတိုင်းရေးသားနိုင်သည်။

```
package main

import (
  "shopping"
  "fmt"
)

func main() {
  fmt.Println(shopping.PriceCheck(4343))
}
```

ထိုအခါတွင် `shopping` project တွင်းသို့ဝင်၍ အောက်ပါ အတိုင်းရိုက်၍ run နိုင်ပါသည်။

```
go run main/main.go
```

Cyclical Imports

တဖြည်းဖြည်း ရှုပ်ထွေးလာသော စနစ်ကို ရေးသားလာသည်နှင့်အမျှ cyclical imports များနှင့်ကြုံတွေ့မည်ဖြစ်သည်။ Package A မှ Package B ကို Import လုပ်ရင်း Package B မှ Package A ကို Import လုပ်ပါက (တိုက်ရိုက်သော်လည်းကောင်း ၊ သွယ်ဝိုက်၍ နောက် package ပေါ်မှ တ

ဆင့်သို့လည်းကောင်း) ကြုံတွေ့ရမည်ဖြစ်သည်။ ထိုအခြေအနေမျိုးကို compiler မှာ ခွင့်မပြုပါ။

ကျွန်တော်တို့ရဲ့ shopping structure ကို error တက်အောင် စမ်းကြည့်ရအောင်။

`shopping/db/db.go` မှ `Item` ကြေညာထားတာကို `shopping/pricecheck.go` ကိုရွှေ့လိုက်ပါ။ `pricecheck.go` ကအောက်ပါအတိုင်းဖြစ်သွားမှာပါ။

```
package shopping

import (
  "shopping/db"
)

type Item struct {
  Price float64
}

func PriceCheck(itemId int) (float64, bool) {
  item := db.LoadItem(itemId)
  if item == nil {
    return 0, false
  }
  return item.Price, true
}
```

အဆိုပါ code ကို run ပါက `db/db.go` မှ `Item` ကို undefined ဖြစ်ကြောင်းပြပါမည်။ `Item` မှာ `db` package တွင်မရှိတော့ပဲ shopping package တွင်

ရောက်သွားသောကြောင့် နည်းလမ်းကျသည် ဟု ဆိုရမည်။ ထိုနောက် `shopping/db/db.go` ကို အောက်ပါ အတိုင်းပြင်ရန်လိုသည်။

```
package db

import (
  "shopping"
)

func LoadItem(id int) *shopping.Item {
  return &shopping.Item{
    Price: 9.001,
  }
}
```

အထက်ပါ code ကို run ပါက သင်မျှော်လင့်ထားသော *import cycle not allowed* ဟုသည့် error ရလာမည်။ ထိုပြဿနာကို structure shared လုပ်ထားသော package နောက်တစ်ခုတည်ဆောက်ခြင်း ဖြင့် ဖြေရှင်းနိုင်သည်။ သင့်၏ directory structure သည် အောက်ပါ အတိုင်းဖြစ်မည်။

```
$GOPATH/src
  - shopping
    pricecheck.go
    - db
      db.go
    - models
      item.go
    - main
      main.go
```

`pricecheck.go` သည် `shopping/db` ကိုပဲ import လုပ်မည်ဖြစ်သော်လည်း `db.go` သည် `shopping` အစား `shopping/models` ကို import လုပ်မည်ဖြစ်ပြီး တပတ်လည်ခြင်းမှ ရပ်တန့်သွားသည်။ `Item` structure ကို

`shopping/models/item.go` ကိုရွေ့လိုက်သဖြင့် `models` package မှ `Item` structure ကို reference ပြုလုပ်ရန် `shopping/db/db.go` ကိုပြောင်းရန်လိုမည်။

```
package db

import (
  "shopping/models"
)

func LoadItem(id int) *models.Item {
  return &models.Item{
    Price: 9.001,
  }
}
```

တခါတရံ `models` များသာမက `utilities` လိုမျိုး ခပ်ဆင်ဆင် folder များကို share ရန်လိုကောင်းလိုလိမ့်မည်။ share ပြုလုပ်ထားသော package များ၏ အဓိက လိုက်နာရမည့် အချက်တစ်ချက်မှာ `shopping` package နှင့်ဎင်း၏ sub-package များမှ import မလုပ်ရပါ။ နောက် အပိုင်းများတွင် ထိုသို့သော dependencies များကို ရှင်းလင်းစေမည့် interface များ အကြောင်းကို ရှင်းပြပါမည်။

Visibility

Go တွင် Package အပြင်ရှိ type နှင့် function များ၏ visible ဖြစ်မဖြစ်ကို သတ်မှတ်သည့် စံချိန်စံညွှန်းတစ်ခုရှိသည်။ function တစ်ခု၏ အမည်သည် uppercase ဖြင့်စပါက visible ဖြစ်ပြီး lowercase ဖြင့်စပါက visible မဖြစ်

ပါ။ ၎င်းသည် function များတွင်သာမက structure field များတွင်လည်း အ ကြိုးဝင်သည်။ field name သည် lower case Δဖင့်စပါက package အတွင်းတွင်သာ access လုပ်နိုင်မည်ဖြစ်သည်။ ဥပမာ `items.go` တွင် အောက်ပါ အတိုင်း function တစ်ခုရှိပါက

```
func NewItem() *Item {
  // ...
}
```

`models.NewItem()` ဟုလှမ်းခေါ်နိုင်မည်ဖြစ်သော်လည်း `newItem` ဟုအမည် ပြောင်းလိုက်ပါက တခြား package မှ လှမ်းခေါ်နိုင်မည် မဟုတ်ပေ။

`shopping` code မှ function များ၏ အမည်များ ၊ type နှင့် fields တို့ကို ပြောင်းကြည့်ပါဦး။ ဥပမာ `Item's` ၏ `Price` ကို `price` ဟုပြောင်းလိုက်ပါ က error တက်မည်ဖြစ်သည်။

Package Management

`go` command တွင် ကျွန်တော်တို့ အသုံးပြုနေသည့် `run` နှင့် `build` အပြင် `get` ဟုသော third-party library များကို ခေါ်ယူ အသုံးပြုနိုင်သည့် command ပါဝင်သည်။ `go get` သည် protocal အမြောက်အများကို support ပြုလုပ်သော်လည်း ယခု ဥပမာတွင် Github မှာ library ကို အသုံးပြုမည်ဖြစ်၍ သင့်၏ ကွန်ပျူတာတွင် `git` install ပြုလုပ်ထားရန်လို

မည်။ git install ပြုလုပ်ပြီးပါက shell/command prompt မှ အောက်ပါ အတိုင်း ရိုက်လိုက်ပါ။

```
go get github.com/mattn/go-sqlite3
```

`go get` သည် remote file များကို fetch လုပ်ပြီး သင့်၏ workplace တွင် သိမ်းထားပေးသည်။ `$GOPATH/src` တွင်ကြည့်လိုက်ပါ။ ကျွန်တော်တို့ တည်ဆောက်ထားသော `shopping` project အပြင် `github.com` ဟု folder ကိုတွေ့ရမည်ဖြစ်ပြီး အတွင်းတွင် `mattn` ဟု folder တစ်ခု ၊ ၎င်းအထဲတွင် `go-sqlite3` ဟု folder တွေ့ရမည်ဖြစ်သည်။

package များကို မည်သို့ import ပြုလုပ်ရမည်နည်းဆိုသည်ကို ပြောပြပြီး ဖြစ်သည်။ လက်ရှိရထားသော `go-sqlite3` package ကို အသုံးပြုနိုင်ရန် အောက်ပါ အတိုင်း import ပြုလုပ်နိုင်သည်။

```
import (
  "github.com/mattn/go-sqlite3"
)
```

URL နှင့်ဆင်တူသော်လည်း လက်တွေ့တွင် `$GOPATH/src/github.com/mattn/go-sqlite3` တွင်တည်ရှိသော `go-sqlite3` ကို import ပြုလုပ်ခြင်းဖြစ်သည်။

Dependency Management

`go get` တွင် အခြား အသုံးပြုနိုင်သည့်နည်းလမ်းများလည်းရှိသေးသည်။ project အတွင်းတွင် `go get` ဟုလှမ်းခေါ်ပါက files များအားလုံးကို scane ပြုလုပ်ပြီး `import` များကိုရှာဖွေပါ third-party library များကိုရှာဖွေပြီး download လုပ်ပေးသည်။ ထိုကြောင့်ဖြင့် `Gemfile` နှင့် `package.json` တို့၏ သဘောနှင့် ဆင်တူသည်ဟု ပြောနိုင်သည်။

`go get -u` ဟုခေါ်ပါက packages များကို update ပြုလုပ်သွားနိုင်မည် ဖြစ်သည်။ (သို့မဟုတ်ပါက လိုချင်သည့် package တစ်ခုချင်းစီကို `go get -u FULL_PACKAGE_NAME` ဟု update ပြုလုပ်နိုင်သည်။) တဖြည်းဖြည်းဖြင့် `go get` ကို မလုံလောက်ဟု သိလာမည်ဖြစ်သည်။ တနည်းအားဖြင့် revision အနေဖြင့် သတ်မှတ်နိုင်ခြင်း မရှိပါ။ master/head/trunk/default ကိုသာ အမြဲတမ်း point လုပ်နေမည်ဖြစ်သည်။ အကယ်၍ သင့်တွင် library တစ်ခု၏ မတူညီသော version များကို project နှစ်ခုတွင် အသုံးပြုလိုပါက ပြဿနာရှိပါသည်။

ထိုပြဿနာကိုဖြေရှင်းနိုင်ရန် third-party dependency management tool တစ်ခုခုကို အသုံးပြုနိုင်သည်။ ၎င်းတို့သည် ထုတ်ထားသည်က မကြာသေးသောလည်း အဆင်ပြေလောက်သည့်နှစ်ခုမှာ goop နှင့် godep တို့ ဖြစ်သည်။ list အပြည့်အစုံကို go-wiki တွင်တွေ့နိုင်သည်။

Interfaces

Interface များသည် contract များကို သတ်မှတ်သော type ဖြစ်ပြီး implementation မပါဝင်ပေ။ ဥပမာ

```
type Logger interface {
  Log(message string)
}
```

သင့်အနေဖြင့် ဘယ်လိုနေရာမှာ အသုံးပြုမလဲ စဉ်းစားနေမည်ဖြစ်သည်။ Interface များသည် implementation များမှ code များကို decouple ပြုလုပ်ရာတွင် အထောက်အကူပြုသည်။ ဥပမာ ကျွန်တော်တို့တွင် အမျိုး အစားစုံလင်သော logger များရှိသည်ဆိုပါစို့။

```
type SqlLogger struct { ... }
type ConsoleLogger struct { ... }
type FileLogger struct { ... }
```

မူလပုံစံအတိုင်း implementation အကျအနရေးခြင်းထက် interface များကိုအသုံးပြုခြင်းဖြင့် Code ကို impact မဖြစ်စေပဲ အလွယ်တကူ ပြင်ဆင် (နှင့် test ပြုလုပ်နိုင်) သည်။

ဘယ်လိုအသုံးပြုရမည်နည်း။ အခြားသို့ type များကဲ့သို့ structure တစ်ခု၏ field အနေနဲ့ဖြင့်လည်း ဖြစ်နိုင်သည်။

```
type Server struct {
  logger Logger
}
```

သို့မဟုတ် function parameter တစ်ခု (ဝါ) return value လည်းဖြစ်နိုင်ပါသေးသည်။

```
func process(logger Logger) {
  logger.Log("hello!")
}
```

C# နှင့် Java ကဲ့သို့သော language များတွင် interface တစ်ခုကို အသုံးပြုလိုပါက Class တစ်ခုအနေဖြင့် မပါမဖြစ်ပါရသည်။

```
public class ConsoleLogger : Logger {
  public void Logger(message string) {
    Console.WriteLine(message)
  }
}
```

Go တွင်ထိုသို့ မလိုအပ်ပါ။ `string` parameter တစ်ခုလက်ခံပြီး ဘာမှ return မပြန်သော `Log` ဟုသော function တစ်ခုရှိပြီး ၎င်းကို `Logger` အနေဖြင့် အသုံးပြုနိုင်သည်။ ထိုကြောင့် inteface များလည်း အလွန်အသုံးဝင်သည်။

```
type ConsoleLogger struct {}
func (l ConsoleLogger) Log(message string) {
  fmt.Println(message)
}
```

၎င်းတွင် သေးငယ်ပြီး တစ်ခုဆိုတစ်ခု အာရုံစိုက်ထားသော interface များတည်ဆောက်ရန်အားပေးသည်။ standard library တစ်ခုလုံး interface များဖြင့်တည်ဆောက်ထားသည်။ `io` package တွင် နာမည်ကြီး `io.Reader`

I `io.Writer` နှင့် `io.Closer` များသည် interface များဖြစ်သည်။ အကယ်၍ `Close()` တစ်ခုတည်းသာ ခေါ်သော function တစ်ခုကိုရေးပါက `io.Closer` ကိုလက်ခံသင့်သည်။

Interface များသည် composition များအဖြစ် ပါဝင်ဆင်နွဲနိုင်သည်။ interface တစ်ခုသည် အခြား interface များပေါင်းစပ်ဖန်တီး ထားသည် လည်းဖြစ်နိုင်သည်။ ဥပမာ `io.ReadCloser` သည် `io.Reader` နှင့် `io.Closer` ပေါင်းစပ်ထားသည့် interface တစ်ခုဖြစ်သည်။

နောက်ဆုံးအနေဖြင့် interface များသည် cyclical imports ကိုရှောင်ရှား ရာတွင် အသုံးပြုသည်။ implementation မရှိသဖြင့် depenedency အများ အစားလိုအပ်ခြင်းမရှိပါ။

နောက်အခန်း မဖတ်ခင်

အထူးသဖြင့် Go workplace တွင် code ကိုဘယ်လို structure ချရသည့် နည်းလမ်းများသည် project များစွာရေးပြီးမှ ခံစားသိရှိရနိုင်သည်။ package name များနှင့် directory name များအကြား တင်းကျပ်သည့် relationship များသည် အရေးကြီးသည် ဆိုသည့်အချက် အပါအဝင် ဖြစ်သည်။ (project အတွင်းသာမက workplace တစ်ခုလုံး) Go ၏ type visiiblity သည် ရိုးရှင်းပြီး အလုပ်ဖြစ်သည့် အပြင် တသမတ်တည်း ဖြစ်သည်။ အချို့သောအရာများဖြစ်သည့် constant နှင့် global variable

များအကြောင်း မပြောထားသောလည်း ၎င်းတို့၏ visiblity သည် naming rule အတူတူပင်ဖြစ်သည်။

နောက်ဆုံးတွင် interface များသည် အသစ်ဖြစ်နေပါက ရင်းနှီးရန် အတိုင်းအတာတစ်ခု လိုအပ်သော်လည်း သင့်အနေဖြင့် `io.Reader` ကို လိုအပ်သည့် function တစ်ခုကိုတွေ့ပါက သင့်အနေဖြင့် ၎င်းကိုရေးသားသူ သည် တကယ်လို၍ ထည့်ရေးသည်လား ပိုနေလားကို ခွဲခြားသိရှိနိုင်ပါသည်။

အခန်း (၅) - Tidbits

အခန်းတွင်မူ တခြား မည့်သည့် language နှင့်မတူညီပဲ တမူထူးခြားနေသော Go ၏ feature များအကြောင်းကို ပြောပါမည်။

Error Handling

Go တွင် အထူးပြုလိုသည် error handling မှာ exception များမဟုတ်ပဲ return value များဖြစ်သည်။ string တစ်ခုကို interger သို့ကူးပြောင်းပေးသည့် `strconv.Atoi` ဆိုပါစို့။

```
package main

import (
  "fmt"
  "os"
  "strconv"
)

func main() {
  if len(os.Args) != 2 {
    os.Exit(1)
  }

  n, err := strconv.Atoi(os.Args[1])
  if err != nil {
    fmt.Println("not a valid number")
  } else {
    fmt.Println(n)
  }
}
```

ကိုယ့် error type တစ်ခု ဖန်တီးနိုင်ပြီး: built-in `error` interface ကိုအခြေခံရန်သာလိုအပ်သည်။

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

ပုံမှန်အားဖြင့် `errors` package ကို import ပြုလုပ်ပြီး function အသစ်ရေးသားခြင်းဖြင့် မိမိတို့ဖာသာ error များပြုလုပ်နိုင်သည်။

```
import (
  "errors"
)


func process(count int) error {
  if count < 1 {
    return errors.New("Invalid count")
  }
  ...
  return nil
}
```

Go ၏ standard library တွင် error variable များအသုံးပြုခြင်း သုံးလေသုံးထရှိသော pattern တစ်ခုဖြစ်သည်။ ဥပမာ `io` package တွင်ရှိသည့် `EOF` variable ကိုအောက်ပါ အတိုင်းဖန်တီးနိုင်သည်။

```
var EOF = errors.New("EOF")
```

၎င်းသည် function ၏ပြင်ပတွင် ဖန်တီးထားသော package variable ဖြစ်ပြီး ပထမစာလုံးသည် အကြီးဖြစ်သောကြောင့် အခြား package များမှ

လည်း access ပြုလုပ်နိုင်သည်။ အခြားသော function များမှ ၎င်း error ကို return ပြန်နိုင်သည်။ ဥပမာ ကျွန်တော်တို့ file တစ်ခုကို read သောအခါ ဖြစ်စေ STDIN ဖြစ်စေ ယခု error ကိုအသုံးပြုနိုင်သည်။ အသုံးပြုသူ အနေဖြင့် ထို singleton ပုံစံကို အသုံးချနိုင်သည်။

```
package main

import (
  "fmt"
  "io"
)

func main() {
  var input int
  _, err := fmt.Scan(&input)
  if err == io.EOF {
    fmt.Println("no more input!")
  }
}
```

နောက်ဆုံးအနေဖြင့် Go တွင် `panic` နှင့် `recover` ကဲ့သို့သော function များ လည်းရှိသည်။ `panic` သည် exception throw ပြုလုပ်ခြင်းဖြင့် ဆင်တူပြီး `recover` မှာ `catch` နှင့်ဆင်တူသည်။ သို့သော်လည်း အသုံးနည်းပါသည်။

Defer

Go တွင် garbage collector ပါဝင်သော်လည်း အချို့သော resource များမှ ကိုယ်တိုင် release လုပ်ပေးရန်လိုသည်။ ဥပမာ files များကို process လုပ်ပြီးပါက `Close()` ပြုလုပ်ရန်လိုသည်။ ထိုကဲ့သို့ Code များ အမြဲအားဖြင့်

အန္တရာယ်များလေ့ရှိသည်။ Code ဆယ်ကြောင်းခန့်ရေးပြီးပါက `Close` ပြုလုပ်ရန် မေ့သည်က ဖြစ်ကောင်းဖြစ်နိုင်သည်။ ထို့အပြင် function များအတွက် return point ပေါင်းများစွာ လိုအပ်သည်လည်း ဖြစ်နိုင်သည်။ ထိုအတွက်ကြောင့် go တွင် `defer` keyword ကိုအသုံးပြုသည်။

```
package main

import (
  "fmt"
  "os"
)

func main() {
  file, err := os.Open("a_file_to_read")
  if err != nil {
    fmt.Println(err)
    return
  }
  defer file.Close()
  // read the file
}
```

အပေါ်မှ code ကို run ပါက error တက်ပါလိမ့်မည်။ (file မရှိသဖြင့်) ၎င်းသည် `defer` ၏ အလုပ်လုပ်သည့် ဥပမာ ပုံစံဖြစ်သည်။ function ၏ နောက်ဆုံး လိုင်းကို execute လုပ်ပြီးပါက `defer` သည်စတင်အလုပ်လုပ်ပါလိမ့်မည်။ ဆိုးဆိုးရွားရွား error တက်သည်ကို ကြုံတွေ့သည်တိုင် defer သည် resource များကို release လုပ်ရာတွင် အသုံးပြုနိုင်သည်။ ထိုကြောင့် return points များကို handle လုပ်ရာတွင် အာရုံစိုက်နိုင်သည်။

go fmt

Go တွင် program အများစုသည် တူညီသော formatting rule ကိုသာ လိုက်နာသည်။ ဥပမာ tab ကို indent အနေဖြင့် အသုံးပြုပြီး brace များသည် statment နှင့်တကြောင်းတည်း တည်ရှိသည်။

သင့်တွင် သင် ရေးနေကြပုံစံ အတိုင်းသာ ရေးလိုမည်။ နားလည်ပါသည်။ ကျွန်တော်လည်း ထိုကဲ့သို ရေးနေသည် အချိန်အတော်ကြာပြီး နောက်ဆုံးတွင် လက်လျော့လိုက်သည်။ အဓိက အကြောင်းအရင်းမှာ `go fmt` ကြောင့် ဖြစ်သည်။ ၎င်းသည် အလွန်အသုံးပြုရလွယ်ကူပြီး စိတ်ချရသည်။ (ဆို တော့ ဘယ် preference ကပိုကောင်းသလဲ ဆိုတာ ငြင်းရန်မလိုတော့) Project အတွင်းမှ အခြားသော sub project အားလုံးကို formatting rule ကို apply ပြုလုပ်နိုင်သည်။

```
go fmt ./...
```

စမ်းကြည့်ပါ။ ၎င်းသည် code ကို indent လုပ်ရုံတင်မက field delcaration များကိုပါ align လုပ်ပေးပြီး import များကို ပါ အကွာရာအလိုက် စီပေးသွားသည်။

Initialized If

Go တွင် condition မစစ်ခင် value များ assign လုပ်နိုင်သော if-statement များကိုအထောက်အပံပေးသည်။

```
if x := 10; count > x {
  ...
```

```
}
```

အပေါ်ပုံစံ ဥပမာ ဖြစ်ပြီး လက်တွေ့တွင်မူ အောက်ပါအတိုင်း သုံးလေ့ရှိသည်။

```
if err := process(); err != nil {
  return err
}
```

သတိပြုရန်မှာ ထို value များသည် if statement ၏ အပြင်ဘက်တွင် အသုံးပြု၍မရနိုင်ပါ။ `else if` နှင့် `else` တို့ရှိပါက ၎င်းအတွင်းတွင် သုံးနိုင်သည်။

Empty Interface and Conversions

Object-oriented language အတော်များများတွင် `object` ဟုခေါ်လေ့ရှိသည့် class များအားလုံး၏ class ဖြစ်သော superclass တစ်ခုရှိသည်။ Go တွင် inheritance မပါရှိသဖြင့် ထိုသို့ superclass လည်းမရှိပါ။ သူမှာရှိသည်က method များတစ်ခုမှ မပါသည့် interface အလွတ်ကို `interface{}` ဟုခေါ်နိုင်သည်။ type တိုင်းသည် interface အလွတ်မှ ဆင်းသက် ခြင်းဖြစ်ပြီး type တိုင်းသည် empty interface ၏ contract ကို လက်ခံနိုင်သည်။

အကယ်၍ `add` function ကိုအောက်ပါအတိုင်းရေးသားနိုင်သည်။

```
func add(a interface{}, b interface{}) interface{} {
  ...
```

```
}
```

Interface variable မှ type တစ်ခုခုကိုပြောင်းလဲလိုပါက `.(TYPE)` ကို အသုံးပြုနိုင်သည်။

```
return a.(int) + b.(int)
```

မှတ်သားရန်မှာ ၎င်း၏ type သည် `int` မဟုတ်ပါက အပေါ်မှ code သည် error တက်ပါလိမ့်မည်။ switch ကို အသုံးပြု၍လည်း အောက်ပါအတိုင်း ရေးသားနိုင်ပါသေးသည်။

```
switch a.(type) {
  case int:
    fmt.Printf("a is now an int and equals %d\n", a)
  case bool, string:
    // ...
  default:
    // ...
}
```

နောက်ပိုင်းတွင် သင်ထင်ထားသည်ထက် empty interface ကို ပို၍ အသုံးပြုဖြစ်လိမ့်မည်။ သို့သော် clean code ဖြစ်လိမ့်မည် မဟုတ်ပေ။ Convert ခဏခဏ ပြန်လုပ်နေရခြင်းသည် အရုပ်ဆိုးပြီး အန္တရာယ်များသောလည်း static language ဖြစ်၍ တခါတရံ လုပ်ရသည့် အနေအထားရှိသည်။

Strings and Byte Arrays

String နှင့် byte array များသည် ဆက်စပ်နေပြီး တစ်ခုနှင့်တစ်ခု convert ပြုလုပ်နိုင်သည်။

```
stra := "the spice must flow"
byts := []byte(stra)
strb := string(byts)
```

တကယ်တော့ အဆိုပါနည်းလမ်းဖြင့် convert ပြုလုပ်ခြင်းသည် အခြား type များအတွက်လည်း အသုံးပြုနိုင်သည်။ အချို့သော functions များသည် `int32` သို့မဟုတ် `int64` နှင့် signed မပြုလုပ်ထားသော အမျိုးအစားများသာ ထည့်သွင်း၍ရမည်ဖြစ်သည်။ ထိုကြောင့် အောက်ကကဲ့သို့ မျိုး ပြုလုပ်ရနိုင်သည်။

```
int64(count)
```

သို့သော်လည်း byte နှင့် string တွင်မူ ထိုကဲ့သိုကိုသာ အတော်များများ လုပ်မိပါလိမ့်မည်။ သတိပြုရန်မှာ `[]byte(X)` ဖြစ်ဖြစ် `string(X)` အသုံးပြုပါက data ကို copy လုပ်ယူခြင်းဖြစ်သည်။ string များမှာ immutable ဖြစ်သောကြောင့်ဖြစ်သည်။

string များကို `runes` ဟုခေါ်သည့် unicode code points များဖြင့် တည်ဆောက်ထားပြီး string တစ်ခု၏ အရှည်ကိုလှမ်းယူပါက သင်ထင်ထားသည့်အတိုင်းဖြစ်မည် မဟုတ်၊ အောက်ပါ code ကို run ပါက 3 ဟု ထွက်ပါလိမ့်မည်။

```
fmt.Println(len("椒"))
```

strings ကို `range` ကိုအသုံးပြုပြီး iterate ပြုလုပ်ပါက rune ကိုရမည်ဖြစ်ပြီး bye ကိုရမည်မဟုတ်ပါ။ အကယ်၍ `[]byte` ကိုအသုံးပြုပါက လိုချင်သည့် အတိုင်းရမည်ဖြစ်သည်။

Function Type

function များသည် ပထမအဆင့် type များဖြစ်သည်။

```
type Add func(a int, b int) int
```

field type အနေဖြင့်သော်လည်းကောင်း၊ parameter အဖြစ်နေသော်လည်းကောင်း return value အနေဖြင့်သော်လည်းကောင်း အသုံးပြုနိုင်သည်။

```
package main

import (
  "fmt"
)

type Add func(a int, b int) int

func main() {
  fmt.Println(process(func(a int, b int) int{
      return a + b
  }))
}

func process(adder Add) int {
  return adder(1, 2)
}
```

function များကို အသုံးပြုခြင်းဖြင့် interface များကိုအသုံးပြုပါက ရရှိသည့် အကျိုးကျေးဇူးများဖြစ်သည့် implementation မှ decouple ပြုလုပ်နိုင်သည် အချက်ကိုလည်းရရှိနိုင်ပါသည်။

နောက်အခန်း မဖတ်ခင်

Go နှင့် programming အသုံးပြုခြင်း၏ ရှုထောင့်ပေါင်းစုံကို ကြည့့်ခဲ့ပြီးဖြစ်သည်။ error handling မည့်သို့ဆောင်ရွက်ပုံ ၊ resource များ၏ connection ကို မည့်သို့ release လုပ်သနည်း၊ နှင့် file များကို open ပြုလုပ်ပုံ။ လူအတော်များများ Go ၏ error handling ကိုမကြိုက်ကြချေ။ ခေတ်နောက်ကျနေသလိုလို။ သဘောတူပါသော်လည်း တခါတရံ follow လုပ်ရန်လွယ်သော code များကို ရေးသားနိုင်အောင် လှုံ့ဆော်ပေးသလို။ `defer` လို မျိုးကတော့ လက်တွေ့ကျသော်လည်း ထူးထူးဆန်ဆန်း ရေးရတဲ့ resource management ပုံစံမျိုး။ တကယ့်တော့ ၎င်းကို resource management အတွက်သာမဟုတ် ကျန်သော နေရာများတွင်လည်း သုံးသည်။ ဥပမာ function မှ အထွက် logging လုပ်သည့်အခါမျိုးကဲ့သို့။

သို့သော်လည်း Go မှ ပါဝင်သော feature အကုန်လုံးတော့ မကြည့်ခဲ့ပါ။ သို့သော် ဘာလာလာ လေ့လာရန် အဆင်ပြေသည့် အဆင့်တစ်ခု ရောက်သွားပြီ ဟု ယုံကြည့်သင့်သည်။

Chapter 6 - Concurrency

Go သည် concurrent-friendly ဖြစ်သည့် language ဟု မကြာခဏ သတ်မှတ်ခြင်းခံရသည်။ အကြောင်းမှာ goroutine နဲ့ channels တို့ကဲ့သို့သော် အလွန်ရိုးရှင်းသော်လည်း အစွမ်းထက်သည့် mechanism ကို support ပေးသောကြောင့်ဖြစ်သည်။

Goroutines

Goroutine သည် thread နှင့်ဆင်တူသော်လည်း ကွာခြားသည်က OS မှ schedule လုပ်သည် မဟုတ်ပဲ Go ကပြုလုပ်ခြင်းဖြစ်သည်။ goroutine အတွင်းရှိ code အခြား code များနှင့် အပြိုင်အလုပ်လုပ်သည်။ အောက်က ဥပမာကို ကြည့်ပါ။

```
package main

import (
  "fmt"
  "time"
)

func main() {
  fmt.Println("start")
  go process()
  time.Sleep(time.Millisecond * 10) // this is bad, don't do thi
  fmt.Println("done")
}

func process() {
```

```
  fmt.Println("processing")
}
```

အချို့စိတ်ဝင်စားစရာ အချက်များပါရှိသော်လည်း အရေးအကြီးဆုံးမှာ goroutine တစ်ခုကို ဘယ်လိုစရမလည်း ဆိုက ကနဦးပါ။ ကိုယ်အသုံးပြုချင်သည့် function ၏ အရှေ့တွင် go keyword ကိုထည့်လိုက်သည်နှင့် စ၍သုံးနိုင်သည်။ function တစ်ခုကို အမည်ပေးမကြေညာလိုပါက anonymous function အနေဖြင့် wrap လုပ်၍ သုံးနိုင်သည်။ မှတ်ထားရန် တစ်ခုမှာ anonymous function များသည် goroutine နှင့်သာ တွဲသုံးနိုင်သည်တော့ မဟုတ် ၊ သို့သော်

```
go func() {
  fmt.Println("processing")
}()
```

Goroutine များမှာ တည်ဆောက်ရန်လွယ်ကူပြီး overhead အနည်းငယ်မျှသာရှိသည်။ goroutine အမြောက်အများမှ OS thread တစ်ခုအတွင်းတွင် run နိုင်သည်။ ၎င်းကို အရေအတွက် M မျှရှိသော application thread (goroutine) များသည် အရေအတွက် N မျှရှိသော OS thread ပေါ်တွင် run သောကြောင့် M:N threading model ဟုခေါ်သည်။ ရလဒ်အနေဖြင့် gorotuine များမှာ overhead အနေဖြင့် OS thread များနှင့်နှိုင်းစာလျှင် အနည်းငယ်မျှသာ (KB အနည်းငယ်) ရှိသည်။ ယနေ့ခေတ် စက်များအနေဖြင့် goroutine သန်းနဲ့ချီ run နိုင်သည်။

ထိုအပြင် mapping နှင့် scheduling ၏ ရှုပ်ထွေးမှုနှင့် တို့ကို ဖုံးကွယ်ထားသည်။ ထိုကြောင့် *ထို code သည် concurrently run သည်* ဆိုသည်ကိုသာ သိရန်လိုပြီး ကျန်သည်ကို Go ကဆောင်ရွက်ပေးသည်။

အထက်က ဥပမာတွင် မီလီစက္ကန့် အနည်းငယ်မျှ `Sleep` ပြုလုပ်လိုက်သည်ကို တွေ့ရမည်ဖြစ်သည်။ ၎င်းမှာ goroutine လုပ်၍မပြီးခင် main process က ပြီးသွားပါက စောင့်မည်မဟုတ်ပဲ ပြီးသွားဖြစ်သည်။ ထိုပြဿနာကို ဖြေရှင်းနိုင်ရန် Code ကိုပြင်ရန်လိုသည်။

Synchronization

Goroutine များတည်ဆောက်ခြင်းသည် လွယ်ကူသည့်အပြင် ၎င်း၏ တန်ဖိုးမှာ ပေါလှသဖြင့် အမြောက်အများ တည်ဆောက်နိုင်သည်။ သို့သော် concurrent code များသည် coordinate လုပ်ရန်လိုသည်။ ထိုပြဿနာကို ဖြေရှင်းနိုင်ရန် Go တွင် `channel` များကို ထည့်သွင်းထားသည်။ `channel` များမလေ့လာမီ concurrent programming ၏ အခြေခံကို နားလည်ရန်လိုသည်။

Concurrent Code များကိုရေးခြင်းဖြင့် value များကို မည်သို့ read & write လုပ်ရမည်ကို သတိထားရန်လိုသည်။ တနည်းအားဖြင့် garbage collector မပါပဲ program ရေးသကဲ့သို့ မိမိတို့၏ data ကို ရှုထောင့်အသစ်မှ မြင်တက်ရန်လိုသလို အန္တရာယ်များကိုလည်း ရှောင်ရှားရန်လိုသည်။

```go
package main

import (
  "fmt"
  "time"
)

var counter = 0

func main() {
  for i := 0; i < 20; i++ {
    go incr()
  }
  time.Sleep(time.Millisecond * 10)
}

func incr() {
  counter++
  fmt.Println(counter)
}
```

ဘာထွက်လာမည်ဟု ထင်သနည်း။

အဖြေမှာ `1, 2, ... 20` က မှန်လည်းမှန် မှားလည်းမှားမည်ဟု ဆိုရမည်။ အထက်ပါ code ကို run ပါက တခါတရံ အဆိုပါ output ကိုရလျင်ရ လိမ့်မည်။ သို့သော် ထိုအခြေအနေသည် မကျိန်းသေ။ အဘယ်ကြောင့်ဆို သော် သင့်အတွက် goroutine ပေါင်းများစွာကို တချိန်တည်းတွင် variable တစ်ခုတည်းဖြစ်သည့် `counter` သို့ access ပြုလုပ်နေခြင်း ကြောင့် ဖြစ်သည်။ goroutine တစ်ခုမှ write ပြုလုပ်ချိန်တွင် နောက်တစ်ခုက read လုပ်နေသည်က ဖြစ်နိုင်သည်။

ဒါကပြဿနာရှိနိုင်လား? လုံးဝအတိအကျပါပဲ။ `counter++` ဟာ ရိုးရိုး code ဖြစ်ပေမယ့် assembly statement အနေနဲ့ကြရင် သင် run တဲ့ platform ပေါ်မူတည်ပြီး အများကြီးဖြစ်နိုင်ပါတယ်။ ထို့ပြဿနာကို run ပါက number တွေအများကြီးက ထူးဆန်းတဲ့ order အတိုင်း print လုပ်တာ တချို့ number တွေက ထပ်တာ ၊ ပျောက်တာတွေ ဖြစ်နိုင်ပါတယ်။ ဒီထက်ပိုဆိုးတာက system crash ဖြစ်တာတွေ၊ တချို့ data တွေ increment ဖြစ်တာတွေ ဖြစ်နိုင်ပါသေးတယ်။

variable တစ်ခုကို concurrent လုပ်နိုင်တာက read တာတစ်ခုပါပဲ။ reader များကို ကြိုက်သလောက် ထားနိုင်သော်လည်း write များကတော့ အစီအစဉ်တကျဖြစ်မှရမှာပါ။ ၎င်းအတွက် နည်းလမ်းမျိုးစုံရှိသည်။ ဥပမာ အထူး CPU instruction များကိုမှီခိုသည့် atomic operation များကို အသုံးပြုနိုင်သည်။ သို့သော် များသောအားဖြင့် သုံးသည်က mutex ပါ။

```
package main

import (
  "fmt"
  "time"
  "sync"
)

var (
  counter = 0
  lock sync.Mutex
)

func main() {
```

```
  for i := 0; i < 20; i++ {
    go incr()
  }
  time.Sleep(time.Millisecond * 10)
}

func incr() {
  lock.Lock()
  defer lock.Unlock()
  counter++
  fmt.Println(counter)
}
```

mutex သည် lock ကိုအသုံးပြု၍ serialize လုပ်ပေးသည်။ `lock` `sync.Mutex` ဟု၍အလွယ်တကူ lock ကိုသတ်မှတ်၍ရခြင်းသည် `sync.Mutex` ၏ default value မှာ unlock ဖြစ်သောကြောင့်ဖြစ်သည်။

ရိုးရှင်းတယ်လို့ ထင်ရလား။ အပေါ်က ဥပမာ မှာ တကယ်သုံး၍ မရပါ။ concurrent programming ကိုအသုံးပြုရင်း ကြုံလာနိုင်သည့် bug များ ပြဿနာ မြောက်များစွာရှိသည်။ ပထမဆုံး အနေဖြင့် ဘယ် code ကို protect လုပ်ရမည်ကို မခွဲခြားနိုင်ပါ။ တခါတရံ coarse lock များ (Code အမြောက်အများကို lock ပြုလုပ်ထားခြင်း) ကိုသုံးဖို့ ကြိုရွယ်ကြလေ့ရှိပြီး နောက်ဆုံးတွင် concurrent programming သုံးရသည့် ရည်ရွယ်ချက်ပါ ပျောက်သွားလေတော့သည်။ ပုံမှန်အားဖြင့် ကောင်းမွန်သော lock များကို အလိုရှိသည်။ သို့မဟုတ်ပါက ဆယ်လမ်းသွားလမ်းမကြီးမှ ရုတ်တရက် လမ်းသွယ်လေးဆီသို့ ရောက်သွားသည့် အတိုင်းဖြစ်လိမ့်မည်။

နောက်ပြဿနာတစ်ခုကတော့ deadlock တွေပါ။ lock တစ်ခုတည်းဆိုရင်တော့ ပြဿနာမဟုတ်သော်လည်း ဒီ code အတွက်ကို နှစ်ခုထက်ပိုသော lock များကိုဖြေရှင်းရပါက goroutineA မှ lockA ကို ကိုင်ထားပြီး lockB မှ access လိုသော်လည်း goroutineB မှ lockB ကိုကိုင်ထားပြီး lockA မှ access လိုသော ပြဿနာများဖြစ်နိုင်သည်။

ထိုကဲ့သို့ ပြဿနာများသည် release မလုပ်မိပါက lock တစ်ခုတည်းနဲ့တင် ဖြစ်နိုင်သော်လည်း multi-lock deadlock လောက် (သိရှိနိုင်ရန်ခက်ခဲလှသဖြင့်) အန္တရာယ်မများပေ။ သို့သော် ဘာဖြစ်လဲသိနိုင်အောင် အောက်ပါအတိုင်း run နိုင်ပါသည်။

```
package main

import (
  "time"
  "sync"
)

var (
  lock sync.Mutex
)

func main() {
  go func() { lock.Lock() }()
  time.Sleep(time.Millisecond * 10)
  lock.Lock()
}
```

Concurrent Programming အပေါ်မှ ဥပမာထက် တခြားဟာတွေ အများကြီးရှိပါသေးသည်။ နှစ်ခုစလုံးပြုလုပ်နိုင်သော read-write mutex ဆိုသည်

လည်းရှိသေးသည်။ ၎င်းသည် reading အတွက်ရော writing အတွက်မှာ function နှစ်ခုကို expose ပြုလုပ်ပေးသည်။ ၎င်းသည် reader ပေါင်းများစွာ အလုပ်လုပ်နေချိန်တွင် writing သည်မထိခိုက်ပါ ဟု သေချာစေနိုင်သည်။ GO တွင် `sync.RWMutex` ထိုသို့သော lock ဖြစ်သည်။ `sync.Mutex` တွင်ပါဝင်သည့် `Lock` နှင့် `Unlock` method တို့အပြင် *Read* ပြုလုပ်ရန် အတွက် `RLock` နှင့် `RUnlock` ဟုသော method များလည်း ပါဝင်ပါသေးသည်။ read-write mutexes များသည် အများအားဖြင့် အသုံးပြုသော်လည်း ၎င်းသည် အသုံးပြုသော developer အတွက် ဘယ်အချိန် data ကို access ပြုလုပ်သည်သာမက ဘယ်လို data ကို access ပြုလုပ်သည်ကို တာဝန်ပိုလာသည်။

ထိုထက်ပို၍ concurrent programming ဆိုသည်မှာလည်း code ၏ အကျဉ်းအကျပ်အပိုင်းများကို serialize ပြုလုပ်ခြင်းမဟုတ်ပဲ goroutine ပေါင်းများစွာကို coodinate ပြုလုပ်ခြင်းဖြစ်သည်။ ဥပမာ ၁၀ မီလီစက္ကန့်ခန့် sleep ပြုလုပ်ခြင်းသည် လှပသော ဖြေရှင်းနည်း မဟုတ်ပေ။ အကယ်၍ goroutine မှာ ၁၀ မီလီစက္ကန့်ထက် ပိုကြာနေပါက မည်သို့ ဆောင်ရွက်မည်နည်း။ သို့မဟုတ် ထိုထက်ပိုမြန်နေပြီး ကျန်တဲ့ ကာလများကို အချိန်ဖြုန်းနေသလို ဖြစ်နေပါက မည်သို့ဆောင်ရွက်မည်နည်း။ ထိုအပြင် ထို့သို့စောင့်ဆိုင်းမည့်အစား *မင်းမှာ data အသစ်ကို လုပ်စရာကျန်သေးသည်* ဟု မည်သို့ အသိပေးမည်နည်း။

ထိုအချက်များကို `channels` များမပါပဲ ဆောင်ရွက်နိုင်သည်။ ရိုးရှင်းသော အခြေအနေများတွင် `sync.Mutex` နှင့်`syncRWMutex` တို့ကို *အသုံးပြု*သင့် သင်ဟုထင်သော်လည်း နောက်အပိုင်းတွင် `channel` များကိုအသုံးပြုပြီး concurrent prgoramming ကို ပို၍ error ကင်းစင် ပြီး သန့်ရှင်းသော Code များ မည်သို့ရေးရမည်နည်းကို အဓိကထား၍ လေ့လာသွားပါမည်။

Channels

Concurrent Programming ၏အဓိက အခက်အခဲမှာ data sharing ဖြစ်သည်။ သင့်၏ goroutine သည် share ရန် data မလိုပါက synchronize ပြုလုပ်ရန်အတွက် သိပ်စိတ်ပူစရာမလိုပေ။ သို့သော် စနစ်အားလုံးအတွက် ၎င်းသည် မဖြစ်နိုင်ပေ။ ထိုအပြင် ထို စနစ်တော်တော်များများ ပြောင်းပြန်ပုံစံဖြစ်သည့် Request များစွာကပင် တူညီသော data ကို ကိုင်တွယ်ရန် တည်ဆောက်ထားလေသည်။ In memory cache သို့မဟုတ် database တို့၏ ၎င်းတို့ကို ညွှန်းဆိုနိုင်သည့် ဥပမာများဖြစ်သည်။ ၎င်းတို့သည် လက်တွေ့အခြေအနေအတွက် အသုံးပြုသည့် ပုံစံများဖြစ်သည်။

Channel များသည် data များကို မျှဝေခြင်းဖြင့် concurrent programming အတွက် အထောက်အကူပြုပေးသည်။ Channel တစ်ခုသည် goroutine များအကြား data ပို့ပေးသည့် pipeline တစ်ခုဖြစ်လာသည်။ တနည်းအားဖြင့် goroutine တစ်ခုမှ နောက်တစ်ခုသို့ channel ကိုအသုံးပြု၍ ပို့ပေး

နိုင်သည်။ ရလဒ်အနေဖြင့် အချိန်ကာလတစ်ခုအတွင်း goroutine တစ်ခုမှ သာ ထို data ကို access ပြုလုပ်နိုင်သည်။

Channel တစ်ခုသည် အခြားသော အရာများကဲ့သို့ type တည်ရှိသည်။ ၎င်း type အတိုင်း data သည် channel မှ တဆင့် စီးဆင်းသွားသည်။ ဥပမာ integer များကို pass နိုင်ရန် channel တစ်ခုကို အောက်ပါအတိုင်း တည်ဆောက်နိုင်သည်။

```
c := make(chan int)
```

အဆိုပါ channel အမျိုးအစားကို `chan int` ဟုခေါ်သည်။ ထို channel ကို function တစ်ခုသို့ pass ရန် အောက်ပါပုံစံကို အသုံးပြုရမည်။

```
func worker(c chan int) { ... }
```

Channel များသည် သယ်ယူခြင်းနှင့် လက်ခံခြင်းဟုသော တာဝန်နှစ်ခုကို ဆောင်ရွက်ပေးသည်။ channel တစ်ခုသို့ ပိုဆောင်ရန် အောက်ပါအတိုင်း

```
CHANNEL <- DATA
```

ဆောင်ရွက်နိုင်ပြီး လက်ခံပါက အောက်ပါအတိုင်း သုံးနိုင်သည်။

```
VAR := <-CHANNEL
```

မြှားပြသည့်ဘက်ကို data ကိုသယ်ယူမည်ဖြစ်သည်။ ပို့ဆောင်ပါက data သည် channel ထဲသို့ရောက်ရှိမည်ဖြစ်ပြီး လက်ခံပါက channel မှ data ထွက်လာမည်။

နောက်ဆုံးအနေဖြင့် သိရန်မှာ data လက်ခံသည့်အချိန်နှင့် ပို့သည့်အချိန်တွင် blocking ဖြစ်မည်ဖြစ်သည်။ channel တစ်ခုမှ လက်ခံသည့်အချိန်တွင် data မရမချင်း goroutine သည် ဆက်လုပ်မည်မဟုတ်။ ထိုနည်းတူ ပို့ဆောင်သည့်အချိန်တွင်လည်း data လက်ခံမရရှိခြင်း အလုပ်လုပ်မည် ဖြစ်သည်။

goroutine များစွာမှ data များကို ရယူပြီး လုပ်ဆောင်သည့် စနစ်တစ်ခုကို မှန်းကြည့်ပါစို့။ ၎င်းသည် ပုံမှန်လိုအပ်ချက်တစ်ခုဖြစ်သည်။ အကယ်၍ goroutine များစွာကိုအသုံးပြု၍ data-intensive ဖြစ်သည့် အလုပ်များဆောင်ရွက်ပါက client များကို timing out ဖြစ်နိုင်သည့် risk တစ်ခုရှိသည်။ ထိုအတွက် ပထမဦးစွာ worker များကိုတည်ဆောင်ရန်လိုသည်။ ၎င်းသည် ရိုးရှင်းသော function တစ်ခုဖြစ်နိုင်သော်လည်း goroutine ဖြင့် မသုံးဖူးသည့် structure တစ်ခုအနေဖြင့် ရေးသားပြပါမည်။

```
type Worker struct {
  id int
}

func (w Worker) process(c chan int) {
  for {
    data := <-c
    fmt.Printf("worker %d got %d\n", w.id, data)
  }
}
```

ကျွန်တော်တို့ တည်ဆောက်လိုက်သော worker သည် ရိုးရှင်းသည်။ data available ဖြစ်ချိန်ကိုစောင့်၍ process လုပ်မည်။ မပြီးဆုံးနိုင်သော loop တစ်ခုအတွင်းတွင်တည်ရှိ၍ အမြဲတမ်း process အတွက် data လာနေသည်ကို စောင့်ဆိုင်းနေမည်။

၎င်းကိုအသုံးပြုနိုင်ရန် ဦးစွာ worker များကိုစတင်ရန်လိုသည်။

```
c := make(chan int)
for i := 0; i < 5; i++ {
  worker := &Worker{id: i}
  go worker.process(c)
}
```

ထိုနောက် အလုပ်များခိုင်းနိုင်သည်။

```
for {
  c <- rand.Int()
  time.Sleep(time.Millisecond * 50)
}
```

အောက်မှ စတင် run နိုင်ရန် အစအဆုံး code ဖြစ်သည်။

```
package main

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

func main() {
  c := make(chan int)
  for i := 0; i < 5; i++ {
    worker := &Worker{id: i}
    go worker.process(c)
```

```
  }

  for {
    c <- rand.Int()
    time.Sleep(time.Millisecond * 50)
  }
}

type Worker struct {
  id int
}

func (w *Worker) process(c chan int) {
  for {
    data := <-c
    fmt.Printf("worker %d got %d\n", w.id, data)
  }
}
```

ဘယ် worker မှ ဘယ် data ကိုရမည်နည်းက မသိနိုင်ပေ။ သိနိုင်သည်က Go သည် channel မှ data ကို တစ်ဦးတည်းကိုသာ ပေးမည်ဟု အာမခံထားသည်။

ထိုနေရာတွင် တစ်နေရာတည်းသော shard ထားသည့်နေရာမှာ channel ဖြစ်ပြီး လက်ခံသည်နှင့်ပို့ဆောင်သည်ကို concurrent လုပ်ဆောင်နိုင်သည်။ channel များသည် synchronization ပြုလုပ်ရန်လိုသည့် code များကို အထောက်အပံပေးပြီး သတ်မှတ်ထားသော အချိန်တစ်ခုတွင် goroutine တစ်ခုမှသာ data တစ်ခုကိုဆောင်ရွက်ပေးသည်။

Buffered Channels

အပေါ်မှ code တွင် ကျွန်တော်တို့ handle လုပ်နိုင်သော data ပမာဏထက် ပိုများလာလျှင် မည်သို့ဖြစ်မည်နည်း။ ထို ဥပမာကို worker တစ်ခုမှ data လက်ခံပြီးတိုင်း sleep လုပ်ကြည့်ပါက မြင်သာလာမည်ဖြစ်သည်

```
for {
  data := <-c
  fmt.Printf("worker %d got %d\n", w.id, data)
  time.Sleep(time.Millisecond * 500)
}
```

ကျွန်တော်တို့ ma

Main Code ဘက်မှကြည့်ပါက user ၏ data အဝင်ကို (ကျွန်တော်တို့ random number generator ဖြင့်အင်ထုထားသော) ကို block ပြုလုပ်ထားသလို ဖြစ်နေသည်။ အဘယ်ကြောင့်ဆိုသော် လက်ခံရရှိသည့် channel မရှိသဖြင့်။

data process ပြုလုပ်ထားသည်ကို အာမခံချက်ရှိရန်လိုသော အခါများတွင် သင့်အနေဖြင့် client ကို block ပြုလုပ်ရန်လိုသည်။ ၎င်းကို ပြုလုပ် နည်းလမ်းအများအပြားရှိသော်လည်း အများစု အသုံးပြုကြသည်မှာ data ကို buffer ပြုလုပ်ခြင်းဖြစ်သည်။ worker မအားပါက data ကို queue ထဲသို့ ခဏသိမ်းထားမည်။ Channels များသည် နဂိုကတည်းကပင် buffer ပြုလုပ်နိုင်သည်။ ကျွန်တော်တို့ `make` ကိုအသုံးပြု၍ channel တည်ဆောက်ကတည်းက channel ၏ length ကိုအောက်ပါအတိုင်းပေးထားခဲ့သည်။

```
c := make(chan int, 100)
```

ထိုသို့ပြောင်းလဲနိုင်သော်လည်း process ပြုလုပ်ခြင်းမှာ အကန့်အသတ်ရှိသလိုဖြစ်နေသည်။ Buffer ပြုလုပ်ထားသော channel မှာ ပို၍ထည့်၍မရချေ။ queue သဘောမျိုးဖြင့် work များကို ခဏစောင့်ဆိုင်းစေသဖြင့် ရုတ်တရက် ဆောင့်တက်လာသော အခါမျိုးတွင် ဖြေရှင်းနိုင်အောင် ဖြစ်သည်။ ကျွန်တော်တို့ ဥပမာတွင်မူ worker များ အလုပ်လုပ်နိုင်သည်ထက် ပို၍ data များကို အဆက်မပြတ် ပို့လွတ်နေသည်။

သို့သော်လည်း ကျွန်တော်တို့ buffer channel ၏ သဘောကို နားလည်လာမည်ဖြစ်သည်။ တနည်းအားဖြင့် buffer ကို channel ၏ အရှည်ကို `len` ဟု အသုံးပြု၍ ကြည့်နိုင်သည်။

```
for {
  c <- rand.Int()
  fmt.Println(len(c))
  time.Sleep(time.Millisecond * 50)
}
```

တဖြည်းဖြည်း တိုးလာပြီး နောက်ဆုံးတွင် channel ကိုပို့သည့် နေရာမှာ ထပ်၍ block ဖြစ်လာမည်ဖြစ်သည်။

Select

buffer ဖြင့်ပင် အခြေအနေတစ်ခုရောက်ပါက message များ မရောက်တော့သည် အနေအထားမျိုးရှိနိုင်သည်။ ကျွန်တော်တို့ အနေဖြင့် worker

များပြီးသွားလျင် memory free ပြုလုပ်မည်ဟု ယူဆပြီး memory အကန့်အသတ်မရှိသုံး၍ မရပေ။ ထိုသို့သော အခြေအနေများတွင် Go ၏ `select` ကိုအသုံးပြုသည်။

`select` သည် အသုံးပြုပုံအားဖြင့် switch နှင့်အနည်းငယ်ဆင်သည်။ ၎င်းဖြင့် Channel အနေဖြင့် တာဝန်ယူနိုင်မယူနိုင်ကို ဆန်းစစ်သည့် code အပိုင်းကို ရေးသားနိုင်သည်။ ပထမဦးစွာ `select` ၏ အလုပ်လုပ်ပုံကို သေချာစွာ သိနိုင်ရန် channel ၏ buffering ကို အရင်ဆုံး ဖယ်ရှားလိုက်ပါ။

```
c := make(chan int)
```

Next, we change our `for` loop:

```
for {
  select {
  case c <- rand.Int():
    //optional code here
  default:
    //this can be left empty to silently drop the data
    fmt.Println("dropped")
  }
  time.Sleep(time.Millisecond * 50)
}
```

ကျွန်တော်တို့သည် တစ်စက္ကန့်ကို message အစောင် နှစ်ဆယ် စာပို့နေပြီး worker များသည် တစ်စက္ကန့်လျင် ဆယ်ခုသာ အလုပ်လုပ်နိုင်သည်။ ထို့ကြောင့် တဝက်မှာ ပျောက်သွားမည်ဖြစ်သည်။

၎င်းမှာ `select` ကိုအသုံးပြုနိုင်ခြင်း၏ အစပင်ဖြစ်သည်။ select ၏ အဓိက ရည်ရွယ်ချက်မှာ channel ပေါင်းများစွာနှင့် တွဲဖက် အလုပ်လုပ်ရန်ဖြစ် သည်။ channel ပေါင်းများစွာဖြင့် `select` သည် ပထမဦဆုံး တစ်ခု avaialbe မဖြစ်မခြင်း block ဖြစ်နေမည် ဖြစ်သည်။ worker တစ်ခုမှ available မဖြစ်ပါက default ကို execute ပြုလုပ်မည်ဖြစ်သည်။ worker တစ်ခုထက်ပို၍ available ဖြစ်ပါက random အတိုင်း ရွေးချယ်ပေးပို့သွား မည်ဖြစ်သည်။

ထိုကဲ့သို့ advanced ဖြစ်သော feature များကို ရှင်းလင်းရန် ရိုးရှင်းသော ဥပမာဖြင့် ပြရန်ခက်သည်။ နောက်တစ်ပိုင်းတွင်တော့ ပို၍ ရှင်းလင်းလာ မည်ဟု မျှော်လင့်ပါသည်။

Timeout

Buffering message များကိုကြည့်ပါက drop ပြုလုပ်ရင်းဖြင့် ဖြေရှင်းသည် ကိုတွေ့ရမယ်။ နောက်ထပ် အသုံးများသည့် နည်းလမ်းတစ်ခုမှာ timeout ဖြစ်သည်။ အချိန်တစ်ခုထိ block ဖြစ်သည်ကို လက်ခံနိုင်သော်လည်း အမြဲတမ်းတော့ မဖြစ်ရပါ။ ၎င်းသည်လည် Go တွင် အလွယ်တကူ စွမ်းဆောင်နိုင်သည်။ အမှန်အတိုင်းပြောရရင် ထို syntax သည် အနည်းငယ် ခက်ခဲမည်ဖြစ်သော်လည်း အလွန်သေသပ်ပြီး အသုံးဝင်သော feature ဖြစ်သဖြင့် ချန်ထားခဲ့၍မရပါ။

block ပြုလုပ်နိုင်သည့် maximium amount ကိုသတ်မှတ်ထားနိုင်ရန် `time.After` ဟုသော function အသုံးပြုနိုင်သည်။ ၎င်းကို အသုံးပြုနိုင်ရန် ကျွန်တော်တို့ ၏ sender ကိုအောက်ပါအတိုင်း

```
for {
  select {
  case c <- rand.Int():
  case <-time.After(time.Millisecond * 100):
    fmt.Println("timed out")
  }
  time.Sleep(time.Millisecond * 50)
}
```

`time.After` မှ channel ကို return ပြန်ပြီး ၎င်းကို `select` အတွင်း အသုံးပြုနိုင်သည်။ ၎င်း channel သည် အချိန်ကာလ ကျော်လွန်သွားပါက write ပြုလုပ်ပါသည်။ ထူးထူးဆန်းဆန်းတော့ မဟုတ်ပါ။ စိတ်ဝင်စားပါက `after` ဘယ်လို implement လုပ်ထားသနည်းကို အောက်ပါအတိုင်း ဖတ်ကြည့်နိုင်သည်။

```
func after(d time.Duration) chan bool {
  c := make(chan bool)
  go func() {
    time.Sleep(d)
    c <- true
  }()
  return c
}
```

`select` ကိုပြန်သွားပါက တခြား စမ်းစရာလေးများရှိသေးသည်။ ရှေးဦးစွာ `default` ကိုပြန်ထည့်ပါက ဘာဖြစ်မည်နည်း။ မှန်းလို့ရပါသလား။ စမ်း

ကြည့်ပါက။ ဘယ်လိုဖြစ်မလဲ မှန်းဆလို့မရပါက channel များ available မဖြစ်ပါက `default` ဆီသို့သွားမည်ဖြစ်သည်။

ထိုအပြင် `time.After` သည် `chan time.Time` အမျိုးအစား channel ဖြစ်သည်။ အပေါ်မှ ဥပမာတွင် sent ထားသော value လို မသုံးသဖြင့် discard လုပ်ထားသော်လည်း အသုံးပြုခြင်းပါက လက်ခံနိုင်ပါသည်။

```
case t := <-time.After(time.Millisecond * 100):
  fmt.Println("timed out at", t)
```

`select` ကို အာရုံစိုက်ကြည့်ပါက `c` ကိုပို့သော်လည်း `time.After` မှလက်ခံရရှိသည်ကို တွေ့ရမည်။ `select` သည် လက်ခံသည်ဖြစ်စေ ပို့သည်ဖြစ်စေ ၊ နှစ်ခုစလုံးပြုလုပ်သည်ဖြစ်စေ အလုပ်လုပ်သည့် ပုံစံက အတူတူပါ။

- available ဖြစ်သော် channel ကိုရွေးချယ်သည်
- channel တစ်ခုထက်ပို၍ရှိပါက randomly ရွေးချယ်သည်
- channel မရှိပါက default case ကိုအလုပ်လုပ်သည်
- default မရှိပါက select block များကိုအလုပ်လုပ်သည်

နောက်ဆုံးတွင် အတွေ့များသည်က `for` အတွင်းတွင် `select` ကိုထည့်သုံးထားခြင်းဖြစ်သည်။ ဥပမာ

```
for {
  select {
  case data := <-c:
    fmt.Printf("worker %d got %d\n", w.id, data)
  case <-time.After(time.Millisecond * 10):
```

```
    fmt.Println("Break time")
    time.Sleep(time.Second)
  }
}
```

နောက်အခန်း မဖတ်ခင်

concurrent programming ကိုအသစ်ဖြစ်နေပါက ၎င်းကို လန့်လျင်လန့်ပါလိမ့်မည်။ ပုံမှန်ထက်ပို၍ အာရုံစိုက် ဂရုစိုက်ရ အားထုတ်ရပြီး Go တွင် ထို အချက်များကို ပိုမိုလွယ်ကူအောင်ဆောင်ရွက်ပေးသည်။

Goroutine များသည် concurrent code များအတွက် abstract ပြုလုပ်နိုင်ရန် အလွန်အသုံးဝင်သည်။ Channel များသည် မျှဝေသုံးစွဲသည့် data များနှင့်ပတ်သတ်သော bugs များကို ချေဖျက်ပေးသည်။ ထိုသို့ ချေဖျက်ပေးရုံသာမက concurrent programming နှင့်ပတ်သတ်သော approach ကိုပါ ပြောင်းလဲသွားစေနိုင်သည်။ သင့်အနေဖြင့် အန္တရာယ်များသော code များအစား message passing ကိုမျိုးကို အသုံးပြုရန်အားပေးသည်။

ထိုသို့ပြောသော်လည်း တခါတရံ `sync` နှင့် `sync/atomic` package များမှ synchronization primitives များကို ပါပြောထားသည်။ နှစ်ခုစလုံး အရေးကြီးသည်ဟု ကျွန်တော်ထင်သည်။ ရှေဦးစွာ channel များကို အာရုံစိုက်သင့်သော်လည်း တခါတရံ တိုတောင်းသောကာလအတွင်း lock တစ်ခုမှ အသုံးပြုရန်လိုသော ပုံစံများတွေ့ပါက mutext နှင့် read-write mutex များကို အသုံးပြုသင့်သည်။

နိဒါန်း

မကြာခဏကြားရသည်မှာ Go သည် *ပျင်းစရာကောင်းသည်* ဟုပြောသံများဖြစ်သည်။ သင်ရတာလွယ်၊ ရေးရတာလွယ်ပြီး အထူးသဖြင့် ဖတ်ရတာလွယ်သဖြင့် ပျင်းဖို့ကောင်းသည် ဟုထင်ကောင်းထင်နိုင်သည်။ ကျွန်တော်တို့ အခန်း ၃ ခန်းလောက် type များနှင့် variable တစ်ခုကို ဘယ်လို declare လုပ်ရမလဲ ဟု အချိန်ယူခဲ့သည်။

သင့်အနေဖြင့် static type language များနှင့်အတွေ့အကြုံရှိပါက အပေါ်မှ ကြုံတွေ့ခဲ့ရသည့် အတော်များများသည် သင့်အတွက် ပြန်နွေးစရာလိုပင်ဖြစ်နေလိမ့်မည်။ Go တွင် pointer များ အသုံးပြုခြင်း Array များအပေါ်မှ wrapper များအသုံးပြု၍ slice များကိုအသုံးပြုခြင်းသည် Java နှင့် C# developer များအတွက်မူ အထူးအဆန်းမဟုတ်ပေ။

သင့်အနေဖြင့် dynamic language များကို အသုံးပြုနေပါက အနည်းငယ်ကွဲပြားမှုကို ခံစားရမည်ဖြစ်သည်။ ၎င်းသည် အတော်လေး သင်ယူရန်လိုသည်။ declaration နှင့် initiation အတွက် အသုံးပြုရမည့် syntax ပေါင်းများစွာကို မှတ်သားရန်မှာ အနည်းငယ်တော့မဟုတ်။ Go ၏ ရိုးရှင်းမှု ကို ကြိုက်နှစ်သက်သော်လည်း ထိုအချက်များသည် ရိုးရှင်းမှုအတွက် ဦးတည်ရာတွင် သိပ်မရိုးရှင်းလှပေ။ သို့ပင်သော်ညား (ဥပမာ variable ကို တစ်ခါသာ ကြေညာ၍ရခြင်း နှင့် := ကိုအသုံးပြု၍ ကြေညာခြင်း) နှင့်

အခြေခံနားလည်မှုများ (ဥပမာ `new(X)` နှင့် `&x{}` တို့သည် memory တွင် allocate ပြုလုပ်ခြင်း ဖြစ်ပြီး slice များ map များနှင့် channel များ အတွက်မူ initialization အတွက်ပါလိုအပ်သဖြင့် `make` ပြုလုပ်ရန်ပါ လိုအပ်ခြင်း) ဟုသော အခြေခံဥပဒေသ များကို ကြေညက်ရန်လိုသည်။

ထိုမှ ကျော်လွန်ပါက go သည် Code များ organize ပြုလုပ်ရန် ရိုးရှင်းပြီး အလုပ်ဖြစ်သော နည်းလမ်းများကို အားပေးသည်။ Interface များ၊ return အခြေပြု error handling၊ resource management အတွက် `defer` နှင့် composition ကို အလွယ်တကူဆောင်ရွက်နိုင်ခြင်း စသဖြင့်။

နောက်ဆုံးတွင်မူ concurrency အတွက် built-in support ပါဝင်ခြင်းဖြစ်ပြီး ၎င်းမှာ အရေးပါလှသည်။ goroutine များသည် ရိုးရှင်း အသုံးပြုရလွယ်ကူပြီး အလုပ်ဖြစ်လှသည်။ ခွဲထုတ်ရန်လည်း လွယ်ကူသည်။ Channel မှာ အနည်းငယ်ရှုပ်ထွေးသည်။ အမြဲတမ်း အခြေခံကို နားလည်ရန်က နောက်တစ်ဆင့်ထက်ပို၍ အရေးကြီးသည့် ဟု ကျွန်တော်က ယူဆသည်။ Channel များမပါရှိသော Concurrent Programming မှာလည်း *အရေးကြီးသည်ဟု* ထင်သည်။ သို့သော်လည်း Channel များမှာ ကျွန်တော်အတွက်တော့ ရိုးရှင်းစွာ abstrct ပြုလုပ်နိုင်ရန်မလွယ်လှဟုထင်သည်။ ၎င်းတို့ သူ့ဟာသူ အရေးပါလှသည့် အခြေခံ အုတ်မြစ် သက်သက် တစ်ခုဖြစ်သည်။ ၎င်းတို့သည် သင့်၏ Concurrent Programming ၏ ရေးသားမှုနှင့် တွေးခေါ်မှုကိုပါ

ပြောင်းလဲသွားနိုင်သည်။ Concurrent Programming ကိုယ်တိုင်မှာ ခက်ခဲသည် ဖြစ်သဖြင့် ကောင်းသည်ဟု ဆိုရမည်။