{ JS }

JavaScript

လို - တို - ရှင်း

အိမောင်

Fairway

မာတိကာ

မိတ်ဆက်

Programming Language ဆိုတာဟာ လူ နဲ့ ကွန်ပျူတာရဲ့ ကြားထဲက ကြားခံဘာသာစကား ဖြစ်ပါတယ်။ လူတွေကနေ့စဉ်သုံး မြန်မာ၊ အင်္ဂလိပ် စတဲ့ ဘာသာစကားတွေကို နားလည်အသုံးပြုကြပြီး ကွန်ပျူတာကတော့ 0 နဲ့ 1 တွေ စုဖွဲ့ပါဝင်တဲ့ Binary System ခေါ် ဘာသာစကားကိုသာ နားလည်အသုံးပြုပါတယ်။ ဒါကြောင့် ကွန်ပျူတာကို ညွှန်ကြားချက်တွေ ပေးပြီး လိုသလိုစေခိုင်းလိုရင် ကွန်ပျူတာနားလည်တဲ့ ဘာသာစကားကို အသုံးပြု စေခိုင်းရမှာပါ။ လူတွေအတွက် ကွန်ပျူတာနားလည်တဲ့ Binary System ကို တိုက်ရိုက် အသုံးပြုဖို့ ခက်ခဲလွန်းပါတယ်။ အဆင်မပြေပါဘူး။ ဒါကြောင့် Programming Language ခေါ် ကြားခံဘာသာစကားတွေကို တီထွင်ပြီး ကွန်ပျူတာကို ညွှန်ကြားစေခိုင်းကြခြင်းပဲ ဖြစ်ပါတယ်။ Programming Language တွေကို အသုံးပြုပြီး ကွန်ပျူတာကို ညွှန်ကြားတဲ့ ကုဒ်တွေ ရေးသားတဲ့လုပ်ငန်းကို **ပရိုဂရမ်းမင်း** လို့ခေါ်ပြီးတော့၊ ဒီလိုရေးသားနိုင်သူကို **ပရိုဂရမ်မာ** လို့ ခေါ်ကြပါတယ်။

Programming Language တွေကို အသုံးပြုရေးသားထားတဲ့ ညွှန်ကြားချက်တွေကို ကွန်ပျူတာနားလည်တဲ့ ဘာသာစကား ဖြစ်အောင် ပြောင်းလို့ရပါတယ်။ ဒီလိုပြောင်းတဲ့အခါ ပြောင်းပုံပြောင်းနည်း (၂) မျိုးရှိတယ်လို့ အကြမ်းဖျဉ်းအားဖြင့် ပြောနိုင်ပါတယ်။

တစ်မျိုးကတော့ Compiler လို့ခေါ်တဲ့ ကြိုတင်ဘာသာပြန်တဲ့ စနစ်ဖြစ်ပါတယ်။ အင်္ဂလိပ်လိုရေးထားတဲ့ စာတစ်ပုဒ်ရှိတယ်ဆိုကြပါစို့။ ဘာသာပြန်တဲ့ လူတစ်ယောက်က အဲ့ဒီစာကို ကြိုတင်ဘာသာပြန်ပြီး ဗမာလို ရေးထားပေးလိုက်မယ်ဆိုရင်၊ ဗမာလိုဖတ်ချင်တဲ့သူက မူလအင်္ဂလိပ်လို ရေးထားတဲ့စာကို ကြည့်စရာမလိုတော့ပါဘူး။ ဗမာလို ပြန်ပြီးသားကို ဖတ်လိုက်ယုံပါပဲ။ ဒီလိုစနစ်မျိုးနဲ့ အလုပ်လုပ်တဲ့ Programming Language တွေကို Compiled Language လို့ ခေါ်ပါတယ်။ ထင်ရှားတဲ့ Compiled Language တွေကတော့ C/C++, Java, Go, Rust တို့ ဖြစ်ပါတယ်။ ဒီ Language တွေကိုအသုံးပြုပြီး ညွှန်ကြားချက်ကုဒ်တွေ ရေးပြီးရင် ကွန်ပျူတာနားလည်တဲ့ ကုဒ်ဖြစ်အောင် ကြိုတင်ဘာသာပြန်စနစ် Compiler နဲ့ Compile

လုပ်ပြီး ပြောင်းပေးရပါတယ်။ ကွန်ပျူတာက မူရင်းကုဒ်ကို မသိပါဘူး။ Compile လုပ်ပြီးပြောင်းထားတဲ့ သူနားလည်တာကိုသာ တိုက်ရိုက်ကြည့်ရှု အလုပ်လုပ်သွားမှာပဲ ဖြစ်ပါတယ်။

နောက်တစ်မျိုးကတော့ Interpreter လို့ခေါ်တဲ့ တိုက်ရိုက်ဘာသာပြန်စနစ် ဖြစ်ပါတယ်။ ဘာနဲ့တူလဲဆိုတော့၊ အင်္ဂလိပ်လိုလူတစ်ယောက်က ပြောနေတဲ့အခါ၊ ဘာသာပြန်တဲ့သူက တိုက်ရိုက် သူပြောသမျှကို ဗမာလိုဘာသာပြန် ပြောပြပေးတဲ့ သဘောမျိုးနဲ့ တူပါတယ်။ နားထောင်တဲ့သူက အင်္ဂလိပ်လို နားမလည်ရင်တောင်၊ ကြားထဲက ဘာသာပြန်က တိုက်ရိုက်အသံထွက်ပြန်ပေးနေလို့ နားလည်သွားတဲ့ သဘောမျိုးပါ။ ဒီလိုစနစ်မျိုးနဲ့ အလုပ်လုပ်တဲ့ Programming Language တွေကို Interpreted Language လို့ ခေါ်ပါတယ်။ Script Language လို့လည်း ခေါ်ကြပါသေးတယ်။ ထင်ရှားတဲ့ Interpreted Language တွေကတော့ Python, Ruby, PHP, JavaScript တို့ ဖြစ်ပါတယ်။ ဒီ Language တွေကိုသုံးပြီး ညွှန်ကြားချက်ကုဒ်တွေရေးပေးလိုက်ရင် ကွန်ပျူတာက အဲ့ဒီကုဒ်ကို နားလည်အလုပ်လုပ်နိုင်ပါတယ်။ သူ့အလိုအလျောက် နားလည်တာမျိုးမဟုတ်ဘဲ ကြားထဲက Interpreter က တိုက်ရိုက်ဘာသာပြန်ပေးနေတဲ့အတွက် နားလည်သွားတဲ့သဘောပဲ ဖြစ်ပါတယ်။

JavaScript

JavaScript ဟာ သူ့အမည်မှာကိုက Script ဆိုတဲ့အသုံးအနှုန်းပါနေပြီး ဖြစ်ပါတယ်။ Interpreted Language တစ်မျိုး၊ Script Language တစ်မျိုး ဖြစ်ပါတယ်။ ဒါကြောင့် JavaScript ကို အသုံးပြုပြီး ညွှန်ကြားချက်ကုဒ်တွေ ရေးပေးလိုက်ရင် ကွန်ပျူတာက နားလည်အလုပ်လုပ်နိုင်ပါတယ်။ အထက်မှာပြောခဲ့သလို တိုက်ရိုက်ဘာသာပြန်စနစ် Interpreter ရဲ့အကူအညီနဲ့ နားလည်နိုင်တာပါ။

JavaScript ကို အရင်က Client-side Script လို့လည်း ခေါ်ကြပါတယ်။ ဒီနေရာမှာ Client တွေ Server တွေအကြောင်းကို အကျယ်မချဲ့ချင်သေးပါဘူး။ လိုရင်းအတိုချုပ် ဒီလိုမှတ်နိုင်ပါတယ်။ JavaScript ဟာ Chrome, Edge, Firefox စတဲ့ Web Browser ထဲမှာ အလုပ်လုပ်တဲ့ Programming Language ပါ။ အဲ့ဒီ Web Browser တွေ ကိုယ်တိုင်က JavaScript ကုဒ်တွေကို တိုက်ရိုက်ဘာသာပြန်ပေးနိုင်တဲ့ Interpreter တွေ ဖြစ်နေတာပါ။ ဒီ Web Browser တွေဟာ Google, Facebook, YouTube စတဲ့ Server ကို ဆက်သွယ် အလုပ်လုပ်ပေးနိုင်တဲ့ Client Software တွေ ဖြစ်ကြပါတယ်။ ဒါကြောင့် Client Software တစ်မျိုးဖြစ်တဲ့ Web Browser ထဲမှာ အလုပ်လုပ်တဲ့ Language ဖြစ်နေလို့ JavaScript ကို Client-side Language လို့ ခေါ်ခြင်းပဲ ဖြစ်ပါတယ်။

JavaScript ကို စတင်တီထွင်စဉ်က အဲ့ဒီလို Client-side Language အနေနဲ့ တီထွင်ခဲ့ကြတာပါ။ နောက် တော့မှ Node လို့ခေါ်တဲ့ နည်းပညာတစ်မျိုး ထွက်ပေါ်လာပြီး JavaScript ကုဒ်တွေကို Browser မလို တော့ဘဲ ဘာသာပြန် အလုပ်လုပ်ပေးနိုင်လာတာပါ။

ဒါကြောင့်ကနေ့အချိန်မှာတော့ JavaScript ဟာ General Purpose ခေါ် ကဏ္ဍစုံမှာ အသုံးပြုနိုင်တဲ့ စွယ်စုံသုံး Language တစ်ခုဖြစ်နေပါပြီ။ JavaScript ကိုအသုံးပြုပြီး Web Application တွေ၊ Mobile App တွေ Desktop Software တွေ ရေးသားဖန်တီးလို့ ရပါတယ်။

စာအုပ်၏ရည်ရွယ်ချက်

ဒီစာအုပ်မှာ ကွန်ပျူတာ ပရိုဂရမ်းမင်းကို လုံးဝမသိသေးသူတွေအတွက် ရည်ရွယ်ပြီးတော့ ပရိုဂရမ်းမင်း အခြေခံ သဘော သဘာဝတွေနဲ့ JavaScript ရဲ့ အခြေခံသဘောသဘာဝတွေကို ဦးစားပေးဖော်ပြပေးသွားမှာပါ။ ဒီစာအုပ်ကို ပေါင်းကူးစာအုပ်တစ်အုပ်လို့ သဘောထားနိုင်ပါတယ်။ "HTML/CSS ပဲ သိတယ်၊ ဘာဆက်လုပ်ရမလဲ" ဆိုတဲ့သူတွေ ဖတ်ရမယ့်စာအုပ် ဖြစ်ပါတယ်။ HTML/CSS မသိသေးရင်တော့ **Bootstrap လိုတိုရှင်း** စာအုပ်ကို အရင်ဖတ်သင့်ပါတယ်။ ပြီးမှ ဒီစာအုပ်ကို ဆက်ဖတ်သင့်ပါတယ်။ HTML/CSS သိပြီးရင်တော့ ဒီစာအုပ်ကို ဆက်ဖက်ပါ။ ပြီးတဲ့အခါ ဒီစာအုပ်ကပေးတဲ့ ဗဟုသုတတွေပေါ်မှာ အခြေခံပြီး PHP လိုတိုရှင်း၊ Laravel လိုတိုရှင်း၊ React လိုတိုရှင်း၊ API လိုတိုရှင်း စတဲ့စာအုပ်တွေကို အစီအစဉ်အတိုင်း ဆက်လက်ဖတ်ရှုလေ့ကျင့်သွားမယ်ဆိုရင်တော့ နောက်ဆုံးမှာ လက်တွေ့အသုံးချ ပရောဂျက်တွေ ရေးသားနိုင်တဲ့အဆင့်ကို ရောက်ရှိသွားနိုင်မှာပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် ပေါင်းကူးစာအုပ်လို့ ပြောတာပါ။ အခြေခံအဆင့်ကနေ လက်တွေ့လုပ်ငန်းခွင်ဝင်အဆင့်ကို ကူးနိုင်ဖို့အတွက် အထောက်အကူဖြစ်စေမယ့် စာအုပ်ပဲ ဖြစ်ပါတယ်။

JavaScript သည် Java မဟုတ်

ကနေ့အချိန်မှာ ပရိုဂရမ်းမင်း ကိုစိတ်ဝင်စားလို့ လေ့လာကြသူအများစု သိကြပြီး ဖြစ်မယ်လို့ ထင်မိပေမယ့် သိပ်မသိကြသေးတာလေး တစ်ခုရှိပါသေးတယ်။ Java နဲ့ JavaScript တို့ဟာ ထင်ရှားတဲ့ Programming Language တွေ ဖြစ်ကြပါတယ်။ ဒါပေမယ့် နာမည်အားဖြင့် ဆင်တူပေမယ့် တစ်ခုနဲ့တစ်ခု ဆက်စပ်သက်ဆိုင်ခြင်းမရှိတဲ့ သီးခြားနည်းပညာတွေပါ။ အသုံးချတဲ့ နယ်ပယ်တွေလည်း မတူကြပါဘူး။ Java ကို လုပ်ငန်းသုံးဆော့ဖ်ဝဲတွေ ဖန်တီးဖို့နဲ့ Android App တွေ ဖန်တီးဖို့အပိုင်းမှာ ပိုအသုံးများကြပါတယ်။ JavaScript ကိုတော့ Web App တွေ ဖန်တီးဖို့နဲ့ Web Services တွေ ဖန်တီးဖို့ ပိုအသုံးများကြပါတယ်။

Java ဟာ Compiled Language တစ်မျိုးဖြစ်ပြီး JavaScript ဟာ Interpreted Language တစ်မျိုးပါ။ ဒီနေရာမှာလည်း မတူကြပါဘူး။ ပြီးတော့ Java ဟာ Statically Typed သဘောသဘာဝ ရှိပြီး JavaScript ကတော့ Loosely Typed သဘောသဘာဝရှိပါတယ်။ ဒီသဘောသဘာဝတွေ အကြောင်းကို Data Types တွေ အကြောင်းပြောရင်းနဲ့ နောက်ပိုင်းမှာ ဆက်ရှင်းပြသွားမှာပါ။ တစ်ခြားကွဲပြားမှုတွေလည်း ကျန်သေးပေမယ့် အခုနေပြောလိုက်ရင် အဆင့်ကျော်သလို ဖြစ်နေပါလိမ့်မယ်။ ဒါကြောင့် လိုရင်းအနှစ်ချုပ်အနေနဲ့ Java နဲ့ JavaScript တို့ဟာ အမည်အားဖြင့် ဆင်တူပေမယ့် မတူကွဲပြားတဲ့ သီးခြားနည်းပညာတွေ ဖြစ်ကြတယ် ဆိုတာလောက်ကိုပဲ မှတ်သားထားပေးပါ။

အခန်း (၁) – Variables

ပရိုဂရမ်းမင်းကို လေ့လာတဲ့အခါ ကိုယ်ရေးလိုက်တဲ့ကုဒ်ကို ကွန်ပျူတာက ဘယ်လိုနားလည် အလုပ်လုပ်သွားသလဲဆိုတာကို ပုံဖော်ကြည့်နိုင်စွမ်းရှိဖို့ အရေးကြီးပါတယ်။ ဒါအရေးအကြီးဆုံး လိုအပ်ချက်လို့ ဆိုနိုင်ပါတယ်။ ပုံဖော်ကြည့်တယ် ဆိုတဲ့နေရာမှာ ကိုယ်သုံးနေတဲ့ ကွန်ပျူတာစနစ်ရဲ့ ဟိုးအတွင်းပိုင်း အလုပ်လုပ်ပုံကို အသေးစိတ်နားလည်ပြီး ပုံဖော်ကြည့်နိုင်ရင်တော့ အကောင်းဆုံးပါပဲ။ ဒါပေမယ့် လက်တွေ့မှာ ကွန်ပျူတာစနစ်တွေက တစ်ခုနဲ့တစ်ခု မတူကြလို့ စနစ်အားလုံးကို အသေးစိတ်နားလည်ထားဖို့ လွယ်တော့မလွယ်ပါဘူး။

ဒါပေမယ့် ကိစ္စမရှိပါဘူး။ ကွန်ပျူတာစနစ်တွေမှာ Abstraction ခေါ် အလွှာလိုက် အဆင့်လိုက် ရှိနေကြလို့ ကိုယ်ထိတွေ့ရမယ့်အလွှာကို နားလည်ထားရင် ရပါပြီ။ နောက်ပိုင်းမှ ကျန်အလွှာတွေကို ထပ်ဆင့်လေ့လာသွားကြရမှာပါ။ အဝေးကြီးမကြည့်ပါနဲ့၊ လူဦးနှောက်အသိစိတ်ကိုပဲကြည့်ပါ။ လက်လေးတစ်ဖက် လှုပ်ရင် လှုပ်သွားတယ်ဆိုတာကို အသိစိတ်က သိပါတယ်။ ဒါပေမယ့် အဲ့ဒီလက်လေး လှုပ်သွားဖို့အတွက် အရိုးတွေ၊ ကြွက်သားတွေ၊ နာဗ်ကြောတွေ ဘယ်လိုပေါင်းစပ် အလုပ်လုပ်သွားသလဲ အတိအကျ မသိပါဘူး။ အဲ့ဒီအရိုး၊ ကြွက်သား၊ နာဗ်ကြောတွေ ဖြစ်ပေါ်လာဖို့ ဖွဲ့စည်းထားတဲ့ အက်တမ်တွေ၊ တစ်ရှူးတွေရဲ့ ပါဝင်မှုဖွဲ့စည်းပုံကိုလည်း မသိပါဘူး။ ဒါပေမယ့် အနည်းဆုံးအခြေခံဖြစ်တဲ့ လက်ကလေးလှုပ်သွားရင် လှုပ်တယ်ဆိုတာကို သိတဲ့ အသိစိတ်ရှိနေတာနဲ့တင် နေ့စဉ်ပုံမှန် လှုပ်ရှားသွားနိုင်နေပါပြီ။

ဒီသဘောပါပဲ။ ကွန်ပျူတာရဲ့ ဟိုးအတွင်းပိုင်း ထရန်စ္စတာလေးတွေရဲ့ အလုပ်လုပ်ပုံ၊ CPU Architecture အကြောင်း၊ Operating System တွေရဲ့ အလုပ်လုပ်ပုံ၊ Compiler/Interpreter တွေရဲ့ အလုပ်လုပ်ပုံအဆင့်ဆင့်၊ စသည်ဖြင့် သိထားရင်ကောင်းပါတယ်။ ဒါပေမယ့် အနည်းဆုံးအခြေခံဖြစ်တဲ့ ကုဒ်ရေးထုံးပိုင်းလောက် သိထားရင် လိုချင်တဲ့ရလဒ် ရရှိအောင် အလုပ်လုပ်လို့ရနေပါပြီ။ ကျန်အဆင့် ကျန်အလွှာတွေကိုတော့ ကိုယ့်ရွေးချယ်မှုပေါ်မူတည်ပြီး ဘယ်အဆင့် ဘယ်အလွှာထိ ဆက်လေ့လာသွားမလဲဆိုတာ နောင်မှာ ဆုံးဖြတ်

လေ့လာသွားကြရမှာပဲ ဖြစ်ပါတယ်။ ရှေ့ဆက်ဖော်ပြတဲ့အခါ ဒီသဘောကို အခြေခံပြီး ရှင်းလင်းဖော် ပြသွားမှာပဲဖြစ်ပါတယ်။

Variables

Variables ကို မြန်မာလို တိုက်ရိုက်ပြန်ရင်တော့ ကိန်းရှင်လို့ ပြန်ရပါမယ်။ ပြောင်းလဲနိုင်သော တန်ဖိုးပေါ့။ မျက်စိထဲမှာ ဘယ်လိုပုံဖော်ကြည့်ရမလဲဆိုရင်၊ Variable ဆိုတာ ကွန်ပျူတာ Memory ပေါ်က နှစ်သက်ရာ တန်ဖိုးတွေ သိမ်းလို့ရတဲ့ နေရာလေးတစ်ခု လို့ မြင်ကြည့်နိုင်ပါတယ်။ အခုလို ညွှန်ကြားချက်လေး တစ်ခု ပေးလိုက်မယ် ဆိုကြပါစို့။

```
var num
```

ဒါဟာ Memory ပေါ်မှာ နေရာလေးတစ်ခုယူပြီး အဲ့ဒီနေရာကို `num` လို့ အမည်ပေးလိုက်တာပါ။ ဘယ်နားမှာနေရာယူရမယ်၊ ဘယ်လိုနေရာယူရမယ်ဆိုတာ ကိုယ်တိုင်တိုက်ရိုက် စီမံစရာမလိုပါဘူး။ သတ်မှတ်ထားတဲ့ ရေးထုံးကို အသုံးပြုပြီး ညွှန်ကြားချက်ကို ရေးပေးလိုက်ယုံပါပဲ။ ကျန်တာကို Compiler/Interpreter က ကြည့်လုပ်သွားပါလိမ့်မယ်။ JavaScript မှာ Variable ကြေညာလိုရင် `var` Keyword ကို အသုံးပြုပြီး ကြေညာရတယ်ဆိုတဲ့ ရေးထုံးသတ်မှတ်ရှိလို့ သတ်မှတ်ချက်နဲ့အညီ ညွှန်ကြားလိုက်ခြင်းပဲ ဖြစ်ပါတယ်။

နေရာလေးယူပြီးပြီ ဆိုရင်တော့ အဲ့ဒီနေရာမှာ သိမ်းချင်တဲ့တန်ဖိုးတွေ သိမ်းလို့ရသွားပါပြီ။ ဒီလိုသိမ်းဖို့ အတွက် Programming Language အများစုက Equal သင်္ကေတကို သုံးကြပါတယ်။ ဒီလိုပါ –

```
var num
num = 3
```

ဒါဟာ Memory ပေါ်မှာ နေရာတစ်ခုယူ၊ `num` လို့အမည်ပေးပြီး၊ အဲ့ဒီနေရာမှာ 3 ဆိုတဲ့တန်ဖိုးကို သိမ်းလိုက်တာပါ။ တစ်ဆက်ထဲ အခုလိုရေးမယ်ဆိုရင်လည်း ရပါတယ်။

```
var num = 3
```

အများစုက Equal ဆိုရင်ညီတယ်လို့ သိထားတာမို့လို့ ပရိုဂရမ်းမင်း ကိုစလေ့လာချိန်မှာ ဒါဟာ ခေါင်းထဲမှာ ပုံဖော်ကြည်ရခက်ပြီး အခက်တွေ့ကြလေ့ရှိတဲ့ ပြဿနာတစ်ခုပါ။ ဒါကြောင့် ဒါလေးကို အထူးဂရုပြုဖို့ လိုပါလိမ့်မယ်။ Equal သင်္ကေတကို ပရိုဂရမ်းမင်းမှာ အများအားဖြင့် တန်ဖိုးသတ်မှတ်ဖို့ သုံးကြပါတယ်။ Assignment Operator လို့ခေါ်ပါတယ်။ ညီတယ်ဆိုတဲ့ အဓိပ္ပါယ်မဟုတ်ပါဘူး။ ဒါကြောင့် Equal သင်္ကေတလေးကိုတွေ့ရင် ဒီလိုမျက်စိထဲမှာ မြင်ကြည့်လိုက်ပါ။

Pseudocode

```
var num ← 3
```

နောက်ပိုင်းမှာ ကုဒ်နမူနာတွေကို ဖော်ပြတဲ့အခါ ဘာကုဒ်အမျိုးအစားလဲ ကွဲပြားသွားအောင် နမူနာရဲ့ အပေါ်နားမှာ ခေါင်းစဉ်တပ် ဖော်ပြသွားပါမယ်။ အထက်ကနမူနာဟာ လက်တွေ့အလုပ်လုပ်တဲ့ကုဒ်မဟုတ်ဘဲ သဘောသဘာဝကို ရှင်းပြဖို့အတွက် အသုံးပြုတဲ့ Pseudocode ခေါ် နမူနာကုဒ် ဖြစ်တဲ့အတွက် Pseudocode လို့ ခေါင်းစဉ်တပ် ပေးထားပါတယ်။

သိမ်းထားတဲ့ တန်ဖိုးတွေကို လိုအပ်တဲ့အခါ ပြန်ယူသုံးလို့ရသလို၊ အခြားတန်ဖိုးနဲ့ ပြောင်းသိမ်းလို့လည်း ရပါတယ်။ ဥပမာ –

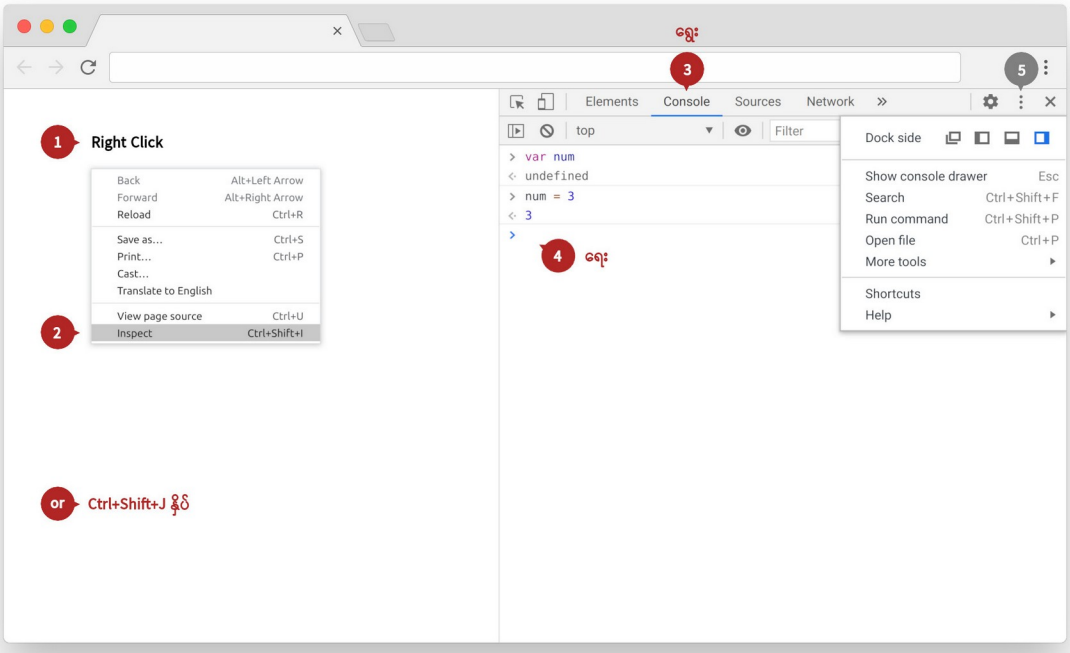
JavaScript

```
var num1 = 5
var num2 = 10
var num3 = num1 + num2
```

ရေးထားတဲ့ကုဒ်အရ Variable (၃) ခု တစ်ခုပြီးတစ်ခု ကြေညာထားတာပါ။ တန်ဖိုးတွေလည်း တစ်ခါထဲ ထည့်ထားပါတယ်။ ထူးခြားချက်အနေနဲ့ `num3` အတွက်တန်ဖိုးအတွက် `num1` နဲ့ `num2` ကို ပေါင်းပြီး ထည့်ဖို့ ညွှန်ကြားချက်ပေးထားပါတယ်။ Plus သင်္ကေတကိုတော့ ပရိုဂရမ်းမင်းမှာလည်း ကိန်းဂဏန်းတွေ ပေါင်းဖို့သုံးနိုင်ပါတယ်။

ဒီစာအုပ်မှာ ဖော်ပြထားတဲ့ နမူနာ JavaScript ကုဒ်တွေကို တစ်ခါထဲ လက်တွေ့ရေးပြီး စမ်းကြည့်နိုင်ပါတယ်။ ဒီလိုစမ်းကြည့်နိုင်ဖို့အတွက် သီးခြားနည်းပညာတွေ ထပ်ထည့်ဖို့မလိုပါဘူး။ Chrome, Edge,

Firefox စတဲ့ ပုံမှန်အသုံးပြုနေကြ Web Browser တစ်ခုရှိရင် စမ်းလို့ရပါတယ်။ Web Browser တွေမှာ DevTools လို့ခေါ်တဲ့ နည်းပညာတစ်ခု ပါဝင်ပြီး၊ အဲ့ဒီ DevTools ထဲမှာ JavaScript ကုဒ်တွေ ရေးလို့ စမ်းလို့ရတဲ့ JavaScript Console လုပ်ဆောင်ချက် ပါဝင်ပါတယ်။



နမူနာပုံမှာ လေ့လာကြည့်ပါ။ Browser ရဲ့ နေရာလွတ်မှာ Right Click နှိပ်ပြီး Inspect ကို ရွေးခြင်းအားဖြင့် DevTools ကို ဖွင့်နိုင်ပါတယ်။ Chrome Browser မှာတော့ Ctrl+Shift+I (သို့) Ctrl+Shift+J ကို Shortcut အနေနဲ့ နှိပ်ပြီးဖွင့်နိုင်ပါတယ်။ ပြီးတဲ့အခါ နမူနာပုံရဲ့ နံပါတ် (3) ပြထားတဲ့နေရာက Console ကို နှိပ်ပြီး နံပါတ် (4) ပြထားတဲ့ နေရာမှာ JavaScript ကုဒ်တွေရေးပြီး စမ်းကြည့်လို့ရပါပြီ။

နံပါတ် (5) ပြထားတဲ့ Menu ကိုနှိပ်ပြီး DevTools ဖော်ပြတဲ့နေရာ ပြောင်းလို့ရပါတယ်။ နမူနာမှာ Browser ရဲ့ ညာဘက်ခြမ်းမှာ ဖော်ပြထားပါတယ်။ အောက်ဘက်ခြမ်းမှာ ဖော်ပြချင်ရင်လည်းရပါတယ်။ ဘယ်ဘက်ခြမ်းမှာ ဖော်ပြချင်ရင်လည်း ရပါတယ်။ ဒါမှမဟုတ် DevTools ကိုသီးခြား Window တစ်ခုနဲ့ ပြစေချင်ရင်လည်းရပါတယ်။ Menu ထဲက Dock side ဘေးနားက ခလုပ်လေးတွေကို နှိပ်ပြီး ပြောင်းပေးနိုင်ပါတယ်။

Console မှာ ကုဒ်တွေရေးစမ်းတဲ့အခါ တစ်ကြောင်းရေးရင် တစ်ခါ အလုပ်လုပ်ပြီး အလုပ်လုပ်လိုက်တဲ့ အတွက် ပြန်ရလာတဲ့ ရလဒ်ကို ချက်ချင်းတွေ့မြင်ရပါတယ်။ ဥပမာ 1 + 1 ရေးပြီး Enter နှိပ်လိုက်ရင် 2 ဆိုတဲ့ရလဒ်ကို ချက်ချင်းပြန်ရပါတယ်။ အပေါ်ကနမူနာပုံမှာလည်း ကုဒ်နှစ်ကြောင်းရေးစမ်းထားပါတယ်။ `var num` ဆိုတဲ့ကုဒ်ရဲ့ ရလဒ်ကတော့ undefined ဖြစ်နေတာကို တွေ့ရပါလိမ့်မယ်။ ရလဒ်တန်ဖိုး မရှိသေးလို့ `undefined` ကိုရလဒ်အနေနဲ့ ပြန်ရတာပါ။ နောက်တစ်လိုင်းမှာ `num = 3` လို့ရေးလိုက်တဲ့အခါ ရလဒ်အနေနဲ့လည်း 3 ကိုပဲ ပြန်ရတာကို တွေ့မြင်ရမှာ ဖြစ်ပါတယ်။

ကုဒ်နမူနာတွေကို လက်တွေ့ရေးစမ်းလိုက်ခြင်းအားဖြင့်၊ ကုဒ်ရဲ့ အလုပ်လုပ်ပုံကို တစ်ခါထဲ မျက်မြင် တွေ့ရလို့ ပိုပြီးတော့ နားလည်သွားမှာဖြစ်ပါတယ်။ မှားတတ်တဲ့အမှားတွေကိုလည်း တစ်ခါထဲ တွေ့မြင်သွားမှာ ဖြစ်သလို၊ ရေးကျင့်လည်း တစ်ခါထဲ ရသွားမှာ ဖြစ်ပါတယ်။ <u>လက်တွေ့ရေးစမ်းတာထက် ပိုထိရောက်တဲ့ လေ့လာနည်း မရှိပါဘူး</u>။ ဒါကြောင့် ကုဒ်နမူနာတွေကို တစ်ခါထဲရေးစမ်းကြဖို့ တိုက်တွန်းပါတယ်။ လိုက်ရေးလို့ရအောင်လည်း နမူနာတွေကို ပေးသွားမှာပါ။

ဆက်လက်လေ့လာကြပါမယ်။ Programming Language အများစုက သတ်မှတ်ထားကြတဲ့ Variable အမည်ပေးပုံပေးနည်း အကြောင်း ပြောချင်ပါတယ်။ Language ပေါ်မူတည်ပြီး ကွဲလွဲချက်တွေရှိနိုင်ပေမယ့် အများအားဖြင့်က ဒီလိုပါ။ Variable အမည်ပေးတဲ့အခါ –

- abc စာလုံးအကြီး/အသေးတွေ ပါလို့ရပါတယ်၊
- 123 ဂဏန်းတွေ ပါလို့ရတယ် (ဒါပေမယ့် ဂဏန်းနဲ့ မစရပါဘူး)၊
- Space တွေပါလို့ မရပါဘူး၊
- လိုအပ်ရင် Underscore သင်္ကေတကို ထည့်သုံးနိုင်ပါတယ်၊
- +,-.*&#@ အပါအဝင် Special Character သင်္ကေတတွေ ပါလို့မရပါဘူး။

ဒါက Language အများစုမှာ ရှိကြတဲ့ သဘောသဘာဝပါ။ JavaScript မှာလည်း အတူတူပါပဲ။ ထူးခြားချက်အနေနဲ့။ Special Character တွေထဲက $ သင်္ကေတကို Variable အမည်မှာ ထည့်သွင်းအသုံးပြုခွင့် ပေးထားပါတယ်။ ဒါကြောင့် ဒီလိုရေးရင် ရပါတယ်။

```JavaScript
var $num = 1
```

နောက်ထပ်သတိပြုရမယ့်အချက်ကတော့ JavaScript အပါအဝင် Language အများစုဟာ Case Sensitive ဖြစ်ကြပါတယ်။ အဓိပ္ပါယ်က စာလုံးအကြီးအသေး လွဲလို့မရပါဘူး။ အတိအကျသတ်မှတ်ပြီး အတိအကျသုံးပေးရပါတယ်။ ဥပမာ –

```JavaScript
var Num1 = 3
var num2 = num1 + 5
```

ဒါဆိုရင် အဆင်မပြေပါဘူး။ ကြေညာတုံးက `Num1` လို့ စာလုံးအကြီးနဲ့ ကြေညာခဲ့ပါတယ်။ ပြန်သုံးတော့ `num1` ဆိုတဲ့စာလုံးအသေးနဲ့ ဖြစ်နေလို့ Language က `num1` ဆိုတာ မရှိဘူးဆိုတဲ့ Error ကိုပေးပါလိမ့်မယ်။ စမ်းကြည့်နိုင်ပါတယ်။

JavaScript မှာ Variable ကြေညာဖို့ နည်းလမ်းနှစ်မျိုးရှိပါတယ်။ ဒီလိုပါ။

```JavaScript
var num1 = 3
let num2 = 3
```

ရေးပုံရေးနည်းအတူတူပါပဲ။ အသုံးပြုတဲ့ Keyword ကွာသွားတာပါ။ အခုပေးထားတဲ့ နမူနာအရ `num1` နဲ့ `num2` ဆိုတဲ့ Variable နှစ်ခုမှာ တန်ဖိုးကိုယ်စီ သတ်မှတ်ပေးလိုက်တာပါ။ ထူးခြားချက်အနေနဲ့ မှတ်ရမှာက `let` Keyword ကို အသုံးပြုကြေညာတဲ့ Variable တွေကို Block Scope Variable လို့ ခေါ်ပါတယ်။ ဒီအကြောင်းကို သင့်တော်တဲ့နေရာ ရောက်ရင် နောက်တစ်ကြိမ် ထပ်ပြောပါမယ်။ အခုလောလောဆယ် Block Scope Variable ကို ဒီနမူနာလေးနဲ့ ကြည့်ပါ။

JavaScript

```
{
    var num1 = 3
    let num2 = 3
}
```

Variable နှစ်ခုလုံးကို တွန့်ကွင်းအဖွင့်အပိတ် Block ထဲမှာ ရေးထားပါတယ်။ `var` ကိုအသုံးပြုကြေညာထားတဲ့ Variable ကို ဒီ Block ရဲ့အပြင်ကနေလည်း သုံးလို့ရပါတယ်။ `let` ကိုအသုံးပြု ကြေညာထားတဲ့ Variable ကိုတော့ ဒီ Block ရဲ့ အပြင်ကနေ သုံးလို့ရမှာ မဟုတ်ပါဘူး။ ဒါကြောင့် –

JavaScript

```
num1 + 2
```

– ဆိုရင် အဖြေ 5 ရမှာပါ။ `num1` ထဲက တန်ဖိုး 3 နဲ့ 2 ကိုပေါင်းလိုက်တဲ့အတွက်ပါ။ ဒါပေမယ့် –

JavaScript

```
num2 + 2
```

ဆိုရင်တော့ Error ဖြစ်ပါတယ်။ `num2` မရှိဘူးလို့ ပြောပါလိမ့်မယ်။ ဘာကြောင့်လဲဆိုတော့ `num2` ဟာ `let` ကို အသုံးပြု ကြေညာထားတဲ့ Block Scope Variable ဖြစ်နေလို့ သူ့ Block ပြင်ပမှာ အသုံးပြုခွင့်မရှိတဲ့ အတွက်ကြောင့်ပါ။

Variable တွေကို Comma ခံပြီး အတွဲလိုက်ကြေညာလို့လည်း ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
let a, b, c = 3
```

ဒါဟာ `a, b` နဲ့ `c` ဆိုပြီး Variable (၃) ခု တစ်ခါထဲကြေညာလိုက်တာပါ။ c အတွက် တန်ဖိုးအနေနဲ့ 3 လို့လည်း တစ်ခါထဲ သတ်မှတ်ထားပါတယ်။ ဒါ c နဲ့ပဲ ဆိုင်ပါတယ်။ a တွေ b တွေနဲ့ မဆိုင်လို့ လက်ရှိကြေညာချက်အရ a နဲ့ b မှာ တန်ဖိုးတွေ မရှိကြသေးပါဘူး။

ကိန်းရှင် Variable တွေရှိသလို ကိန်းသေ Constant တွေလည်းရှိပါသေးတယ်။ Variable တွေကတော့ တန်ဖိုးကို ပြောင်းလို့ရပေမယ့် Constant ကတော့ တန်ဖိုးကို ပြောင်းလို့မရပါဘူး။ ဒီလိုရေးရပါတယ်။

JavaScript

```
const PI = 3.14
```

`const` Keyword ကိုအသုံးပြုပြီးကြေညာလိုက်တာပါ။ အထဲမှာ တန်ဖိုးတစ်ခုလည်း တစ်ခါထဲ ထည့်ပေးထားပါတယ်။ နမူနာအရ `PI` ဟာ Constant ဖြစ်သွားပါပြီ။ ဒါကြောင့် တန်ဖိုးပြောင်းလို့ မရတော့ပါဘူး။ အခုနေ ဒီလိုရေးရင် Error ဖြစ်ပါလိမ့်မယ်။

JavaScript

```
PI = 3.142
```

Constant ကိန်းသေကို ပြောင်းခွင့်မရှိဘူးဆိုတဲ့ Error ကိုရမှာပါ။ `var` နဲ့ `let` ရဲ့အားသာချက်/အားနည်းချက်၊ `const` ရဲ့ ထူးခြားချက် စသည်ဖြင့် ပြောစရာလေးတွေ ကျန်သေးပေမယ့် အဲ့ဒါတွေက ကုဒ်ရေးသားတဲ့ အတွေ့အကြုံ အနည်းအကျဉ်း ရှိပြီဆိုတော့မှ ပြောလို့ကောင်းတဲ့ အကြောင်းအရာတွေပါ။ ဒါကြောင့် အခုအစပိုင်းမှာတော့ ရေးနည်းတွေ ဒီလိုရှိတယ် ဆိုတာလောက်ကိုပဲ မှတ်ထားပေးပါ။

ရေးလိုက်တဲ့ကုဒ်တွေဟာ အမြဲတမ်း တစ်ကြောင်းပြီးမှတစ်ကြောင်း အစီအစဉ်အတိုင်း အလုပ်လုပ်တာတော့ မဟုတ်ပေါ့ဘူး။ နောက်မှရေးထားပေမယ့် Language ဒီဇိုင်းအရ အရင်အလုပ်လုပ်သွားတာမျိုးတွေရှိသလို Asynchronous Programming ခေါ် ပြုံတူအလုပ်လုပ်စေနိုင်တာမျိုးတွေလည်း ရှိကြပါတယ်။ ဒါပေမယ့် ခေါင်းထဲမှာ အလုပ်လုပ်သွားပုံကို ပုံဖော်ကြည့်တဲ့အခါ အများအားဖြင့် အစီအစဉ်အတိုင်းပဲ ကြည့်ကြရမှာပါ။ ဥပမာ –

JavaScript

```
let num1 = 1
let num2 = 2

num1 + num2
```

ဒါရှင်းပါတယ်။ `num1` ကြေညာတယ်၊ `num2` ကြေညာတယ်၊ ပြီးတော့မှ `num1` နဲ့ `num2` ကို ပေါင်းထားပါတယ်။ ဒီလိုရေးမယ်ဆိုရင် အဆင်ပြေမှာ မဟုတ်ပါဘူး။

JavaScript

```
num1 + num2

let num1 = 1
let num2 = 2
```

ပထမဆုံး `num1` နဲ့ `num2` ကို ပေါင်းခိုင်းတဲ့အခါ မရှိဘူးဆိုတဲ့ Error တက်ပါလိမ့်မယ်။ အစီအစဉ်အရ `num1` တို့ `num2` တို့ ကြေညာတဲ့အဆင့်ကို မရောက်သေးလို့ပါ။ ဒါပေမယ့် ထူးခြားချက်အနေနဲ့ `var` Keyword ကို အသုံးပြုမယ်ဆိုရင် ရလဒ်တစ်မျိုး ဖြစ်သွားပါလိမ့်မယ်။

JavaScript

```
num1 + num2

var num1 = 1
var num2 = 2
```

Console မှာ ကုဒ်တွေရေးပြီး Enter နှိပ်လိုက်ရင် ရေးလိုက်တဲ့ ကုဒ်ကို ချက်ချင်းအလုပ်လုပ်ပေးပါတယ်။ Shift + Enter နှိပ်လိုက်ရင်တော့ အလုပ်မလုပ်သေးဘဲ နောက်တစ်ကြောင်းကို ဆင်းပေးလို့ နောက်တစ်လိုင်း ထပ်ရေးနိုင်ပါတယ်။ ဒီနည်းနဲ့ နှစ်ကြောင်း သုံးကြောင်း ရေးပြီးမှ အလုပ်လုပ်စေချင်ရင်လည်း ရနိုင်တယ်ဆိုတာကို ထည့်သွင်းမှတ်သားပေးပါ။ အပေါ်ကကုဒ်နမူနာ (၃) ကြောင်းကို Shift + Enter နဲ့ (၃) ကြောင်းလုံးတစ်ခါထဲ ရေးပြီးမှ စမ်းကြည့်ပါ။

အလုပ်တော့ မလုပ်ပါဘူး။ ဒါပေမယ့် Variable မရှိဘူးဆိုတဲ့ Error မတက်တော့ပါဘူး။ Variable တွေကို အရင်မကြေညာဘဲ နောက်မှကြေညာပေမယ့် ရှိမှန်းသိပါတယ်။ ဒါပေမယ့် သူတို့ထဲက တန်းဖိုးတွေကိုတော့ မသိသေးပါဘူး။ ဒါကြောင့် မရှိဘူးဆိုတဲ့ Error မတက်တော့သလို အဖြေလည်းမှန်မှာ မဟုတ်ပါဘူး။ အဲ့ဒီကုဒ်ကို တစ်ကယ်တမ်းအလုပ်လုပ်သွားတဲ့အခါ ဒီလိုလုပ်သွားမှာပါ။

Pseudocode

```
var num1
var num2
num1 + num2
num1 = 1
num2 = 2
```

JavaScript က သူ့ဘာသာ `num1` နဲ့ `num2` ကြေညာချက်တွေကို အပေါ်ပို့လိုက်ပြီးမှ အလုပ်လုပ်သွားမှာပါ။ ဒီသဘောကို Variable Hoisting လို့ခေါ်ပါတယ်။ Variable Lifting လို့လည်းခေါ်ပါတယ်။ ကြေညာချက်တွေကို အလိုအလျောက် အပေါ်တင်ပေးလိုက်တာမို့လို့ပါ။

ဒါဟာ ထူးခြားချက်တစ်ခုမို့လို့ သိအောင်ထည့်ပြောလိုက်တာပါ။ လက်တွေ့မှာ အစီအစဉ်အတိုင်း ရှိသင့်တဲ့ ကုဒ်ကို အစီအစဉ်အတိုင်း ရေးပေးဖို့ လိုအပ်ပါတယ်။

အခန်း (၂) – Data Types

Variable တစ်ခု ကြေညာလိုက်တဲ့အခါ Memory ပေါ်မှာ နေရာယူပေးသွားတယ်ဆိုတော့၊ ဘယ်လောက် ပမာဏယူမှာလဲဆိုတာ ပြောစရာရှိလာပါတယ်။ သိမ်းမယ့်အချက်အလက် အမျိုးအစား Data Type ပေါ်မှာ မူတည်ပြီးတော့ လိုသလောက် ပမာဏကို ယူသွားတာပါ။ A, B, C ဆိုတဲ့ အင်္ဂလိပ်စာ စာလုံးလေး တစ်လုံး သိမ်းဖို့အတွက် 1 byte ပမာဏရှိတဲ့ နေရာယူလိုက်ရင် သိမ်းလို့ရသွားပါပြီ။ 1, 2, 3 ကိန်းဂဏန်းလေး တစ်လုံး သိမ်းဖို့ဆိုရင်လည်း 1 byte ပမာဏရှိတဲ့ နေရာနဲ့တင် လုံလောက်ပါတယ်။ ဒါပေမယ့် 100, 1234, 9999 စတဲ့ တန်ဖိုးတွေ သိမ်းချင်လို့တော့ 1 byte ပမာဏနဲ့ ရမှာမဟုတ်ပါဘူး။ ဒီလိုတန်ဖိုးတွေ သိမ်းဖို့ အတွက် လက်ခံသိမ်းဆည်းနိုင်တဲ့ ပမာဏကို နေရာယူဖို့ လိုပါလိမ့်မယ်။ အလားတူပဲ "Hello", "မင်္ဂလာပါ" စတဲ့စာတွေ သိမ်းချင်ရင်လည်း လက်ခံသိမ်းဆည်းနိုင်လောက်တဲ့ ပမာဏကို ယူဖို့ လိုပါလိမ့်မယ်။

Programming Language အများစု အခြေခံ Data Type တွေ ရှိကြပါတယ်။ ဒီ (၄) မျိုးပါ။

- Character
- Number
- Boolean
- String

ဒီအခြေခံ Data Type တွေကို အများအားဖြင့် Primitive Type လို့ခေါ်ကြပါတယ်။ တန်ဖိုးတစ်ခု သိမ်းလို့ရ တဲ့အမျိုးအစား ဖြစ်တဲ့အတွက် Scala Type လို့လည်း ခေါ်ကြပါတယ်။ တန်ဖိုးတွေ အတွဲလိုက်သိမ်းလို့ရတဲ့ Types တွေလည်း ရှိပါသေးတယ်။ နောက်ပိုင်းမှာ ဆက်ပြောပေးပါမယ်။

Character ဆိုတာ A, B, C စာလုံးလေး တစ်လုံးကို ပြောတာပါ။ Variable တစ်ခုကြေညာပြီး သူ့ရဲ့ Data Type ကို Character လို့ သတ်မှတ်ပေးလိုက်ရင် စာလုံးလေးတစ်လုံးကိုသာ လက်ခံသိမ်းဆည်းပေးနိုင်တဲ့ Variable တစ်ခုကို ရသွားတာပါ။ ဒီသဘောသဘာဝတွေကို ရှင်းပြဖို့အတွက် Pseudocode တွေနဲ့ နမူနာ ပြချင်ပါတယ်။ Pseudocode ဆိုတာ နမူနာကုဒ်သက်သက်ဖြစ်ပြီး တစ်ကယ် လက်တွေ့အလုပ်လုပ်မယ့် ကုဒ်မဟုတ်ကြောင်း ထပ်ပြောလိုပါတယ်။ ဒါကြောင့် ရေးစမ်းဖို့ မဟုတ်ဘဲ၊ ဖတ်ကြည့်ပြီး နားလည်အောင် ကြိုးစားဖို့ ဖြစ်ပါတယ်။

Pseudocode

```
let blood: char = 'A';
```

ဒါဟာ `let` Keyword ကိုအသုံးပြုပြီး `blood` လို့ခေါ်တဲ့ Variable တစ်ခုကြေညာလိုက်တာပါ။ ထူးခြားချက်အနေနဲ့ နောက်မှာ `char` Keyword နဲ့ Character Data Type ဖြစ်ကြောင်း ထည့်သွင်းကြေညာလိုက်တဲ့ အတွက် နေရာယူတဲ့အခါ စာလုံးတစ်လုံးစာပဲ နေရာယူသွားမှာဖြစ်လို့ သိမ်းတဲ့အခါမှာလည်း စာလုံးတစ်လုံးပဲ သိမ်းလို့ရမှာ ဖြစ်ပါတယ်။ Variable Name နဲ့ Data Type ကို Full-Colon နဲ့ ပိုင်းခြားပြီး ရေးတဲ့ နည်းကို နမူနာအနေနဲ့ အသုံးပြုထားပါတယ်။

တစ်လက်စထဲ Character တန်ဖိုးဖြစ်တဲ့ A ကို ရေးတဲ့အခါ Single Quote အဖွင့်အပိတ်ထဲမှာ ရေးပေးထားတာကိုလည်း သတိပြုသင့်ပါတယ်။ Character Data Type ရှိတဲ့ Language အများစုမှာ ဒီလိုပဲ ရေးရပါတယ်။ ကုဒ်လိုင်းရဲ့ နောက်ဆုံးက Semicolon ကိုလည်း သတိပြုသင့်ပါတယ်။ တစ်ချို့ Language တွေမှာ ကုဒ်တစ်လိုင်း ဆုံးတိုင်း Semicolon နဲ့ အဆုံးသတ်ပေးဖို့ သတ်မှတ်ထားကြလို့ မဖြစ်မနေထည့်ပေးရပါတယ်။ တစ်ချို့ Language တွေမှာတော့ မလိုအပ်ပါဘူး။

Number ဆိုတဲ့ Data Type အမျိုးအစားထဲမှာတော့ ဘယ်လို Number လဲဆိုပြီး မျိုးကွဲတွေ ထပ်ရှိနိုင်ပါသေးတယ်။ ကိန်းပြည့်တွေလား။ ဒဿမကိန်းတွေလား။ ကိန်းပြည့်တွေသိမ်းဖို့အတွက် နေရာလိုအပ်ချက်နဲ့ ဒဿမကိန်းတွေ သိမ်းဖို့အတွက် နေရာလိုအပ်ချက်က မတူပါဘူး။ ကိန်းပြည့်ဆိုတာ အပေါင်းကိန်းတွေ အနှုတ်ကိန်းတွေ အကုန်ပါတဲ့ ကိန်းပြည့်တွေ သိမ်းလို့ရတာလား။ အပေါင်းကိန်း တွေချည်းပဲ ပါတဲ့ ကိန်းပြည့်တွေပဲ ရတာလား။ ရတယ်ဆိုတာ ဘယ်လောက်ထိရတာလဲ။ 1 ကနေ 1000 ထိလား။ အဆုံးအစ မရှိ ပေးချင်သလောက် ပေးလို့ရတာလား။ စသည်ဖြင့် ရှိပါသေးတယ်။

အပေါင်း အနှုတ် အားလုံးပါတဲ့ ကိန်းပြည့်တွေကို Integer သို့မဟုတ် Signed Integer လို့ ခေါ်ကြပါတယ်။ အပေါင်းကိန်းတွေချည်းပဲ ပါတဲ့ ကိန်းပြည့်တွေကိုတော့ Unsigned Integer လို့ခေါ်ပါတယ်။ ဘယ်လောက်ထိ သိမ်းလို့ရသလဲ ဆိုတာကို Language အများစုက ပုံသေ သတ်မှတ်ထားကြပေမယ့် တစ်ချို့ Language တွေက 8-bit, 16-bit, 32-bit, 64-bit စသဖြင့် ကိုယ်လိုသလောက် နေရာယူသတ်မှတ်ခွင့်ပေးကြပါတယ်။

Pseudocode

```
let num1: i8 = 100;
let num2: i32 = 10000000;
```

နမူနာမှာ `num1` ဟာ 8-bit Integer ဖြစ်ကြောင်း `i8` Keyword နဲ့ သတ်မှတ်ပေးထားသလို `num2` ဟာ 32-bit Integer ဖြစ်ကြောင်း `i32` နဲ့ သတ်မှတ်ပေးထားတာပါ။ Signed Integer တွေမှာ ကိန်းဂဏန်းတွေ သိမ်းတဲ့အခါ $-(2^{n-1})$ ကနေ $2^{n-1} - 1$ ထိသိမ်းလို့ရပါတယ်။ ဒါကြောင့် 8-bit Integer ဆိုရင် -128 ကနေ 127 ထိသိမ်းလို့ရပါတယ်။ လက်ခံနိုင်တဲ့ ပမာဏထက် ပိုသိမ်းမိရင်တော့ Overflow Error တက်မှာပါ။ 32-bit Integer ဆိုရင်တော့ -2147483648 ကနေ 2147483647 ထိ သိမ်းလို့ရပါတယ်။

တစ်ကယ်တော့ Number ဆိုတဲ့အထဲမှာ Binary, Octal, Hexadecimal စတဲ့ စနစ်တွေ ကျန်ပါသေးတယ်။ Low Level Programming ခေါ် ကွန်ပျူတာ System အတွင်းပိုင်းထိ စီမံတဲ့ကုဒ်တွေရေးလိုရင် မဖြစ်မနေ သိထားဖို့ လိုအပ်မှာပါ။ ဒီစာအုပ်မှာ အဲ့ဒီအကြောင်းတွေ ထည့်သွင်းမဖော်ပြနိုင်ပါဘူး။ စိတ်ဝင်စားရင် Number System တွေအကြောင်း ဆက်လက်လေ့လာထားသင့်ပါတယ်။ ဒါတွေကို နားလည်ထားခြင်းအားဖြင့် ကွန်ပျူတာရဲ့ အလုပ်လုပ်ပုံကို ပိုအတွင်းကျကျ မြင်နိုင်ပါလိမ့်မယ်။

ဒဿမကိန်းတွေ သိမ်းဖို့အတွက်တော့ အနည်းဆုံး 32-bit ကနေ စလိုပါတယ်။ တစ်ချို့ Language တွေမှာ Float, Double, Long စသည်ဖြင့် ဒဿမကိန်းတွေ မူကွဲတွေ ခွဲထားကြပါတယ်။ တစ်ချို့ Language တွေမှာတော့ 32-bit Float နဲ့ 64-bit Float ဆိုပြီး နှစ်မျိုးရှိပါတယ်။ ဒီလိုကြေညာနိုင်ပါတယ်။

Pseudocode

```
let price: f32 = 99.45;
```

`price` Variable ဟာ 32-bit Float ဖြစ်ကြောင်း `f32` Keyword နဲ့ တွဲကြေညာပေးလိုက်တာပါ။

Boolean ကတော့ True သို့မဟုတ် False တန်ဖိုးနှစ်ခုထဲက တစ်ခုကိုသာ လက်ခံသိမ်းဆည်းနိုင်တဲ့ Data Type ဖြစ်ပါတယ်။ တစ်ခြားတန်ဖိုးတွေကို လက်ခံသိမ်းဆည်းနိုင်ခြင်း မရှိပါဘူး။ မှား/မှန် စစ်ပြီး လုပ်ရတဲ့ အလုပ်တွေ အများကြီးရှိလို့ အသုံးဝင်တဲ့ Data Type တစ်ခုပါ။

String ကတော့ စာတွေသိမ်းလို့ရတဲ့ Data Type အမျိုးအစားပါ။ String ဆိုတာ အခြေခံကျလွန်းတဲ့ မဖြစ်မနေလိုအပ်ချက်မို့လို့ ထည့်ပြောပေမယ့် သူ့ကို အခြေခံ Primitive Data Type လို့ ပြောဖို့ခက်ပါတယ်။ မူအားဖြင့် String ဆိုတာ Character တွေကို အတွဲလိုက် တွဲပြီးသိမ်းတာဖြစ်လို့ တန်ဖိုးတစ်ခုလို့ ပြောဖို့လည်း ခက်ပါတယ်။ ဥပမာ –

Pseudocode

```
let greet: str = "Hello";
```

ဒါဟာ String Variable တစ်ခုထဲမှာ H + e + l + l + o ဆိုတဲ့ Character (၅) ခု အတွဲလိုက် သိမ်းလိုက်တာပါ။ ဒါကြောင့် တန်ဖိုးတစ်ခုဆိုတာထက် Character (၅) ခု အတွဲလိုက်ပါတဲ့ တန်ဖိုးလို့ ပြောရပါမယ်။ ဒီနေရာမှာလည်း String တန်ဖိုးတွေကို Double Quote အဖွင့်အပိတ်ထဲမှာ ထည့်ရေးထားတာကို သတိပြုပါ။ Language အများစုမှာ ဒီလိုရေးရလေ့ ရှိပါတယ်။

JavaScript စာအုပ်မှာ နမူနာတွေကို JavaScript နဲ့ မပေးဘဲ Pseudocode နဲ့ပေးနေတာ အကြောင်းရှိပါတယ်။ Programming Language တွေကို Data Type စီမံပုံပေါ်မူတည်ပြီး Statically Typed Language နဲ့ Loosely Typed Language ဆိုပြီး အမျိုးအစား နှစ်မျိုးရှိလို့ နှိုင်းယှဉ်ဖော်ပြချင်တဲ့အတွက် ဖြစ်ပါတယ်။

Statically Typed Language တွေမှာ Data Type ဟာ သတ်မှတ်ပြီးရင် ပုံသေဖြစ်သွားပါပြီ။ အများအားဖြင့် ကိုယ်တိုင်ကြေညာပြီး သတ်မှတ်ပေးရပါတယ်။ ဒီလိုပါ –

Pseudocode

```
let num1: i32 = 3;
```

`num1` Variable အတွက် 32-bit Integer အဖြစ် Data Type ကို တစ်ခါထဲ သတ်မှတ်ပေးလိုက်တာပါ။ ဒီလိုသတ်မှတ်ပေးပြီးနောက်မှာ တစ်ခြား Data အမျိုးအစားတွေကို ဒီ Variable မှာ လက်ခံသိမ်းဆည်းနိုင်

မှာ မဟုတ်တော့ပါဘူး။ စကြည်ညာကတည်းက Integer သိမ်းဖို့ ကြေညာထားတာ ဖြစ်တဲ့အတွက် Integer ပဲ လက်ခံသိမ်းဆည်းနိုင်မှာပါ။ ဥပမာ –

Pseudocode

```
num1 = 3.14
```

ဆိုရင် Error တက်ပါပြီ။ Float တန်ဖိုးတစ်ခုကို Integer Variable ထဲမှာ သိမ်းဆည်းဖို့ ကြိုးစားနေလို့ပါ။

Pseudocode

```
num1 > 3.14
```

Greater Than သင်္ကေတကိုသုံးပြီး အကြီးအသေး နှိုင်းယှဉ်ကြည့်လိုက်တာပါ။ ဒါလည်းပဲ Error ဖြစ်ပါလိမ့်မယ်။ Integer နဲ့ Float မတူတဲ့ အမျိုးအစားနှစ်ခုကို နှိုင်းယှဉ်ဖို့ ကြိုးစားနေလို့ပါ။ ဒါဟာ Statically Typed Language တွေရဲ့ အလုပ်လုပ်ပုံ သဘောသဘာဝ ဖြစ်ပါတယ်။

JavaScript ကတော့ Loosely Typed Language ဖြစ်ပါတယ်။ Dynamic Language လို့လည်း ခေါ်ပါတယ်။ သူမှာ Primitive Data Type (၆) မျိုးရှိပြီး အခြေခံအကျဆုံး (၃) မျိုးကို မှတ်ထားရင် ဒီအဆင့်မှာ လုံလောက်ပါပြီ။ အဲ့ဒါတွေကတော့ –

- Number
- Boolean
- String

ဒီ (၃) မျိုးဖြစ်ပါတယ်။ JavaScript မှာ Character ဆိုတဲ့ Data Type သီးခြားမရှိပါဘူး။ ပြီးတော့ Integer နဲ့ Float ဆိုပြီး နှစ်မျိုးခွဲမထားဘဲ Number ဆိုပြီး တစ်မျိုးထဲသာ ရှိပါတယ်။ JavaScript ရဲ့ Number ဟာ 64-bit Float အမျိုးအစားပါ။ ဒီတစ်မျိုးထဲနဲ့ ကိန်းပြည့်၊ ဒဿမကိန်း၊ အပေါင်းကိန်း၊ အနှုတ်ကိန်း၊ အားလုံးကို သိမ်းတဲ့သဘောပဲ ဖြစ်ပါတယ်။

ဒီထက်ပိုထူးခြားတာကတော့ Dynamically Typed Language ဖြစ်လို့ Data Type ကို ကိုယ်တိုင်ကြေညာပေးစရာ မလိုပါဘူး။ ထည့်သွင်းလိုက်တဲ့ တန်ဖိုးပေါ်မူတည်ပြီး Data Type က အလိုအလျောက် ပြောင်းလဲအလုပ်လုပ်ပေးသွားမှာ ဖြစ်ပါတယ်။

JavaScript

```
let myvar
```

ပေးထားတဲ့နမူနာမှာ `myvar` အမည်နဲ့ Variable တစ်ခု ကြေညာထားပါတယ်။ Data Type ပြောမထားသလို တန်ဖိုးလည်း ထည့်သွင်းထားခြင်း မရှိသေးပါဘူး။ ဒါဆိုရင် သူရဲ့ Data Type ကို `undefined` လို့ ခေါ်ပါတယ်။ အမျိုးအစားသတ်မှတ်ထားခြင်း မရှိသေးတဲ့ Variable ပါ။

JavaScript

```
myvar = "ABC"
```

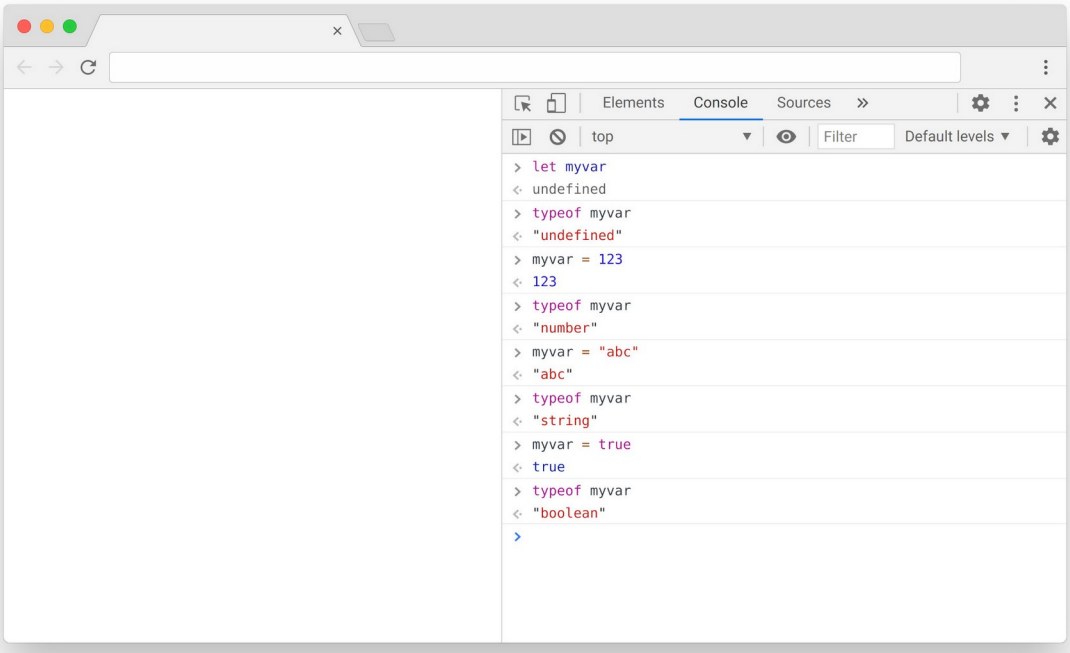
ဒီတစ်ခါ ကြိုတင်ကြေညာထားတဲ့ `myvar` ထဲကို ABC လို့ခေါ်တဲ့ တန်ဖိုးတစ်ခုထည့်သွင်းလိုက်ပါပြီ။ ဒီလို ထည့်သွင်းလိုက်တဲ့အတွက် `myvar` ရဲ့ Data Type က String ဖြစ်သွားပါပြီ။ သူ့အလိုလိုဖြစ်သွားတာပါ။ Language က ပြောင်းပေးလိုက်တာပါ။

JavaScript

```
myvar = 123
```

ဒီတစ်ခါ `myvar` ထဲမှာ 123 ဆိုတဲ့ တန်ဖိုးတစ်ခုထည့်သွင်းလိုက်ပါတယ်။ ရပါတယ်။ Error တွေဘာတွေ မဖြစ်ပါဘူး။ Language က အလိုအလျောက် `myvar` ရဲ့ Data Type ကို Number အဖြစ်ပြောင်းပြီး ဆက်အလုပ်လုပ်ပေးသွားမှာ မို့လို့ပါ။ ဒီသဘောသဘာဝကို Type Juggling လို့ခေါ်ပါတယ်။ Type တွေကို အလိုအလျောက် ပြောင်းပြောင်းပြီး သိမ်းပေးသွားတာ မို့လို့ပါ။

ပိုပြီးတော့မြင်သာချင်ရင် JavaScript မှာရှိတဲ့ `typeof` Keyword ရဲ့အကူအညီနဲ့ အခုလိုရေးပြီး စမ်းကြည့်လို့ရပါတယ်။



နမူနာပုံမှာ လေ့လာကြည့်လိုက်ရင် ဘာတန်ဖိုးမှ မသတ်မှတ်ရခင်မှာ `myvar` ရဲ့ Data Type ကို `typeof` နဲ့ ထုတ်ကြည့်လိုက်တဲ့အခါ `undefined` ဖြစ်နေတာကိုတွေ့ရမှာဖြစ်ပြီး တန်ဖိုးတွေ ထည့်သွင်းပြီးနောက် `typeof` နဲ့ ထုတ်ကြည့်လိုက်တဲ့အခါ ထည့်သွင်းလိုက်တဲ့ တန်ဖိုးပေါ်မူတည်ပြီး `myvar` ရဲ့ Data Type ပြောင်းလဲသွားတာကို တွေ့မြင်ရခြင်းပဲဖြစ်ပါတယ်။ ဒီလို ထည့်သွင်းလိုက်တဲ့ တန်ဖိုးပေါ်မူတည်ပြီး Data Type ပြောင်းလဲယုံသာမက အခုလို လုပ်ဆောင်ချက်မျိုးတွေကိုလည်း လုပ်ပေးနိုင်ပါသေးတယ်။

JavaScript

```
"ABC" > 123
```

String နဲ့ Number အမျိုးအစား မတူတဲ့ အရာနှစ်ခုကို နှိုင်းယှဉ်နေပေမယ့် Error မဖြစ်ပါဘူး။ JavaScript က သူ့ဘာသာ တူအောင်ညှိပြီး အလုပ်လုပ်ပေးသွားမှာ မို့လို့ပါ။ ABC ကို Number ပြောင်းလိုက်တဲ့အခါ 0 ရပါတယ်။ ဒါကြောင့် အဖြေအနေနဲ့ false ကိုရမှာပါ။ 0 က 123 ထက် မကြီးတဲ့အတွက်ကြောင့်ဖြစ်ပါတယ်။ ဒီသဘောသဘာဝကိုတော့ Type Coercion လို့ခေါ်ပါတယ်။ လိုအပ်တဲ့အခါ Type တွေကို အလိုအလျောက် ချိန်ညှိပြီး အလုပ်လုပ်ပေးသွားမှာမို့လို့ပါ။

ဒီလိုသဘောသဘာဝရှိတဲ့ Language အမျိုးအစားဖြစ်လို့ JavaScript ကို Dynamically Typed Language လို့ခေါ်တာပါ။ ဒီ Dynamically Typed ဖြစ်တဲ့အတွက် အားသာချက်အနေနဲ့ ရေးရတာ ပိုလွယ်သွားပါတယ်။ Type စီမံတဲ့ကိစ္စတွေ ကိုယ်တိုင်လုပ်စရာ မလိုတဲ့အတွက်ပါ။ အားနည်းချက် အနေနဲ့ မမျှော်လင့်တဲ့အမှားတွေ ကြုံရတတ်ပါတယ်။ ကိုယ်ကမခိုင်းဘဲ သူ့ဘာသာလုပ်နေလို့ ထင်မထားတဲ့ အမှားတွေ ကြုံရတာမျိုးပါ။

Data Type အကြောင်းပြောတဲ့အခါ Primitive Type တွေနဲ့တင် မပြည့်စုံပါဘူး။ Compound Type ခေါ်တန်ဖိုးတွေကို အတွဲလိုက်သိမ်းလို့ရတဲ့ အမျိုးအစားတွေရှိပါသေးတယ်။ Structure Type လို့ခေါ်တဲ့ ဖွဲ့စည်းပုံနဲ့ သိမ်းလို့ရတဲ့ အမျိုးအစားတွေလည်း ရှိပါသေးတယ်။ ဒီအကြောင်းတွေကိုတော့ Array တွေ Object တွေအကြောင်း ရောက်တော့မှ ဆက်ကြည့်ကြပါမယ်။

String

String အကြောင်း နည်းနည်းထပ်ပြောပါဦးမယ်။ JavaScript မှာ String တစ်ခုတစ်ဆောက်ဖို့အတွက် သုံးလို့ရတဲ့ သင်္ကေတ (၃) မျိုးရှိပါတယ်။ Single Quote, Double Quote နဲ့ Back tick တို့ပါ။ Character Data Type ရှိတဲ့ Language တွေမှာ Single Quote ကို Character တန်ဖိုးသတ်မှတ်ဖို့သုံးပြီး Double Quote ကို String တန်ဖိုးသတ်မှတ်ဖို့ သုံးကြပါတယ်။ ဒီလိုပါ –

Pseudocode

```
let blood: char = 'A';
let greet: str = "Hello";
```

JavaScript မှာတော့ Character Data Type သီးခြားမရှိဘဲ String တန်ဖိုးသတ်မှတ်ဖို့အတွက် Single Quote (သို့မဟုတ်) Double Quote နှစ်သက်ရာကို အသုံးပြုနိုင်ခြင်းဖြစ်ပါတယ်။

JavaScript

```
let blood = "A"
let greet = 'Hello'
```

နှစ်ခုလုံးဟာ မှန်ကန်တဲ့ String တန်ဖိုးတွေ ဖြစ်ကြပါတယ်။ အဖွင့်အပိတ် မှန်ဖို့တော့လိုပါတယ်။ ဒီနေရာမှာ ပြောစရာရှိလာတာက Escape Character အကြောင်းပါ။ အကယ်၍ 3 O' Clock ဆိုတဲ့ String တန်ဖိုးကို သတ်မှတ်လိုရင် ဘယ်လိုလုပ်ရမလဲ။ ဒါမှမဟုတ် 3" (၃ ပေ) လိုတန်ဖိုးမျိုး သတ်မှတ်လိုရင်ရော ဘယ်လို လုပ်ရမလဲ။ တန်ဖိုးထဲမှာ Quote တွေပါနေလို့ပါ။ ဒီလိုရေးလို့ရပါတယ်။

JavaScript

```
let time = "3 O' Clock"
```

Double Quote နဲ့သတ်မှတ်ထားတဲ့ String အတွင်းထဲမှာ Single Quote တန်ဖိုးပါနေတာပါ။ ရပါတယ်။ အကယ်၍ Single Quote နဲ့သတ်မှတ်ထားတဲ့ String အတွင်းထဲမှာ Single Quote တိုက်ရိုက်ရေးလို့ မရတော့ပါဘူး။ ဒီလို ရေးပေးရပါတယ်။

JavaScript

```
let time = '3 O\' Clock'
```

Backslash သင်္ကေတကိုသုံးပြီး Escape လုပ်ပေးလိုက်တာပါ။ တနည်းအားဖြင့် \' ဆိုတာ ဒီနေရာမှာ Single Quote ထည့်လို့ ညွှန်ကြားချက် ပေးလိုက်တာပါပဲ။ အလားတူပဲ၊ Double Quote နဲ့သတ်မှတ်ထားတဲ့ String အတွင်းထဲမှာ Double Quote တွေ ထည့်သွင်းဖို့ လိုရင် အလားတူ Escape လုပ်ပေးနိုင်ပါတယ်။

JavaScript

```
let height = "3\" tall"
```

တစ်ခြား Escape လုပ်ပေးဖို့လိုနိုင်တာတွေ အများကြီးရှိပါသေးတယ်။ \n ဆိုရင် ဒီနေရာမှာ နောက်တစ်လိုင်းဆင်းလို့ ညွှန်ကြားချက် ပေးလိုက်တာပါ။ \u1000 ဆိုရင် ကကြီးဆိုတဲ့စာလုံးတစ်လုံး ဒီနေရာမှာ ထည့်ပေးပါလို့ ပြောလိုက်တာပါ။ \u ကို ယူနီကုဒ်စာလုံးတွေ ထည့်သွင်းဖို့ သုံးရတာပါ။ ဒါတွေရဲ့လက်တွေ့အသုံးပြုပုံကို ပြောဖို့တော့ နည်းနည်းစောပါသေးတယ်။ ထုံးစံအတိုင်း ဒီလိုရေးထုံးမျိုးတွေ ရှိတယ်ဆိုတာကိုပဲ မှတ်သားထားပေးပါ။

နောက်ထပ်တစ်မျိုးဖြစ်တဲ့ Back Tick သင်္ကေတနဲ့ သတ်မှတ်ရတဲ့ String တွေကိုတော့ Template String လို့ခေါ်ပါတယ်။ String တစ်ခုအတွင်းမှာ တစ်ခြားတန်ဖိုးတွေကို ရောထည့်ရေးချင်ရင် အသုံးဝင်ပါတယ်။

JavaScript

```
let name = "Bob"
let greet = `Hello Alice`
```

ဒါကရိုးရိုးရေးလိုက်တာပါ။ `name` Variable အတွင်းမှာ Bob လို့ခေါ်တဲ့ String တန်ဖိုးတစ်ခုရှိပြီး `greet` Variable အတွင်းမှာတော့ Hello Alice လို့ ခေါ်တဲ့ String တန်ဖိုးတစ်ခု ရှိနေတာပါ။ Template String ရဲ့ ထူးခြားချက်ကတော့ အခုလိုရေးလို့ရခြင်းဖြစ်ပါတယ်။

```
let name = "Bob"
let greet = `Hello ${name}`
```

ဒါဆိုရင် `greet` ရဲ့ တန်ဖိုးက Hello Bob ဖြစ်သွားမှာပါ။ `name` Variable ကို String အတွင်းထဲမှာ ရောထည့်ရေးသားလိုက်ခြင်း ဖြစ်ပါတယ်။ ဒီလိုထည့်သုံးနိုင်ဖို့အတွက် `${}` သင်္ကေတကိုသုံးရပါတယ်။ ဒီသင်္ကေတအတွင်းမှာ ထည့်သွင်းလိုတဲ့ Variable ကို ပေးရခြင်း ဖြစ်ပါတယ်။ အသုံးဝင်ပါတယ်။ တစ်ခြား Programming Language တွေမှာ ဒီလိုသဘောမျိုးနဲ့ တန်ဖိုးတွေရောစပ်ပေးလို့ရတဲ့ String Format ဆိုတာ အခြေခံအကျဆုံးအနေနဲ့ ပါဝင်တဲ့ လုပ်ဆောင်ချက်ဖြစ်ပြီး JavaScript မှာ ဒီလိုလုပ်ဆောင်ချက်မျိုးပါလာတာ မကြာသေးပါဘူး။

String တန်ဖိုးတွေ သတ်မှတ်တဲ့အခါ နှစ်ကြောင်းသုံးကြောင်းခွဲပြီးတော့လည်း ရေးလို့ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
let name = "Bob"
let age = 22
let greet = `
    Hello ${name},
    you are ${age} years old.
`
```

Template String မှ မဟုတ်ပါဘူး။ Single Quote, Double Quote နဲ့ရေးတဲ့ ရိုးရိုး String တွေမှာလည်း အခုလိုပဲ လိုအပ်ရင် လိုင်းတွေခွဲရေးလို့ ရပါတယ်။

Special Data Types

JavaScript Data Type မှာ ထူးခြားပြီး မျက်စိလည်ချင်စရာကောင်းတဲ့ အမျိုးအစား (၃) မျိုးရှိပါသေးတယ်။ `undefined`, `null` နဲ့ `NaN` တို့ဖြစ်ပါတယ်။ ရေးသားပုံ စာလုံး အကြီးအသေး ပြောင်းလို့မရပါဘူး။ ဒီအတိုင်း အတိအကျရှိနေတာပါ။ `undefined` အကြောင်းတော့ အပေါ်နားမှာ ပြောခဲ့ပါတယ်။ Type အမျိုးအစား သတ်မှတ်ထားခြင်း မရှိသေးတဲ့ အခြေအနေကို `undefined` လို့ခေါ်တာပါ။ `null` ကတော့ တန်ဖိုးမရှိတဲ့ အခြေအနေကို ပြောတာပါ။ မတူပါဘူး။

JavaScript

```
let myvar
```

ဒါဟာ အမျိုးအစားသတ်မှတ်ရခြင်း မရှိသေးတဲ့ `undefined` Variable တစ်ခုပါ။

JavaScript

```
let myvar = null
```

ဒါဆိုရင်တော့ တန်ဖိုးမရှိသေးတဲ့ Variable ပါ။ သူ့ရဲ့ Data Type ကိုက `null` Type ဖြစ်နေတာပါ။

`NaN` ကတော့ Not a Number ရဲ့ အတိုကောက် ဖြစ်ပါတယ်။ Number Data Type တစ်မျိုးပါ။ ဒါပေမယ့် Number လည်း မဟုတ်ပါဘူး။ ဒါကြောင့် `NaN` ကို Number မဟုတ်တဲ့ Number Data Type လို့ ခေါ်ပါတယ်။ အဲ့ဒါကြောင့် ပြောတာပါ။ ဒီ (၃) မျိုးက မျက်စိလည်ချင်စရာ ကောင်းပါတယ်။

JavaScript

```
let num1
let num2 = 1

num1 + num2
```

ဒီကုဒ်မှာ `num1` က `undefined` ဖြစ်ပါတယ်။ `num2` ကတော့ Number ပါ။ ဒီနှစ်ခုကို ပေါင်းလိုက်တဲ့အခါ `NaN` ကိုရပါတယ်။ `undefined` ကို Number ပြောင်းလို့မရတဲ့အတွက် Number မဟုတ်တဲ့ Number ဖြစ်တဲ့ `NaN` ကိုရတာပါ။ နားရှုပ်ပြီး မျက်စိတွေလည်သွားရင် စိတ်မပျက်ပါနဲ့။ အတွေ့အကြုံရှိ ကျွမ်းကျင် ပရိုဂရမ်မာတွေကိုယ်တိုင် မျက်စိလည်ကြတဲ့ အကြောင်းအရာတစ်ခု ဖြစ်ပါတယ်။

Boolean

ဆက်လက်ပြီးတော့ Boolean အကြောင်း ပြောစရာ ရှိပါသေးတယ်။ ကွန်ပျူတာဟာ 0 နဲ့ 1 တွေ စုဖွဲ့ပါဝင်တဲ့ Binary System ကိုသာ နားလည်တဲ့စနစ် ဖြစ်ပါတယ်။ အလုပ်လုပ်တဲ့အခါမှာ 0 လား 1 လား၊ false လား true လား၊ မှားလား မှန်လား၊ ဆိုတဲ့ အခြေအနေကို ကြည့်ပြီး အလုပ်လုပ်တာပါ။ ဒါကြောင့် Boolean Type မှာ true နဲ့ false ဆိုတဲ့ တန်ဖိုးနှစ်မျိုးထဲသာ ရှိပေမယ့် အလွန်အရေးကြီးတယ်လို့ ရှေ့ပိုင်းမှာ ထည့်ပြောခဲ့တာပါ။ ကွန်ပျူတာကိုယ်တိုင်က ဒီနှစ်မျိုးပေါ်မှာ အခြေခံပြီး ဆုံးဖြတ်အလုပ်လုပ်သွားမှာ မို့လို့ပါ။

Boolean မှာ မူလတန်ဖိုးအနေနဲ့ true နဲ့ false နှစ်မျိုးပဲရှိပါတယ်။ ဒါပေမယ့် တစ်ခြားတန်ဖိုးတွေကိုလည်း လိုအပ်ရင် Boolean ပြောင်းပြီး အသုံးပြုနိုင်ပါတယ်။ ဥပမာ 0 ကို false အစား အသုံးပြုပြီး၊ 1 ကို true အစား အသုံးပြုနိုင်ပါတယ်။ ဒီလိုတန်ဖိုးတွေကို Boolean ပြောင်းတဲ့အလုပ်ကို ကိုယ်တိုင်လုပ်စရာမလိုပါဘူး။ လိုအပ်ရင် JavaScript က သူ့ဘာသာပြောင်းပြီး အလုပ်လုပ်ပေးပါတယ်။ ဒါကြောင့် ဘယ်တန်ဖိုးတွေကို Boolean ပြောင်းရင် true ရပြီး ဘယ်တန်ဖိုးတွေကို Boolean ပြောင်းရင် false ရသလဲဆိုတာ သိဖို့လိုလာပါတယ်။ အကျဉ်းချုပ်အားဖြင့် ဒီလိုမှတ်နိုင်ပါတယ်။

- `0`
- `""`
- `null`
- `undefined`
- `NaN`

ဒီ (၅) မျိုးကို Boolean ပြောင်းရင် false ဖြစ်ပါတယ်။ Zero, Empty String, `null`, `undefined` နဲ့ `NaN` တို့ပါ။ ဒီတန်ဖိုးတွေကို Falsy Value လို့ခေါ်ပါတယ်။ မလိုအပ်ဘဲ ရှုပ်မှာစိုးလို့ ထည့်မပြောတာလေးတစ်ချို့တော့ကျန်ပါသေးတယ်။ လောလောဆယ် ဒီ (၅) မျိုးကလွဲရင် ကျန်တန်ဖိုးတွေ Boolean ပြောင်းရင် true ဖြစ်တယ်လို့ အကြမ်းဖျဉ်းအားဖြင့် မှတ်နိုင်ပါတယ်။ Truthy Value လို့ခေါ်ပါတယ်။ ဥပမာတစ်ချို့ပေးရရင် ဒီလိုပါ။

- `1`
- `-2`
- `" "`
- `"false"`
- `3.14`

ဒါတွေအားလုံးက Truthy ဖြစ်တဲ့ Value တွေပါ။ အနှုတ်ကိန်းတွေ၊ ဒဿမကိန်းတွေကို Boolean ပြောင်းရင်လည်း true ပဲရမှာပါ။ နမူနာမှာ Space တစ်ခုပါတဲ့ String ကိုလည်း ထည့်ပေးထားပါတယ်။ Empty String ဟာ false ဖြစ်ပေမယ့် Space တစ်ခုပါတဲ့ String ကတော့ true ဖြစ်ပါတယ်၊ Empty မဟုတ်လို့ပါ။ ပြီးတော့ "false" ဟာလည်း true ဖြစ်ပါတယ်။ "false" လို့ရေးထားပေမယ့် String တစ်ခုဖြစ်နေလို့ပါ။

အခုလောလောဆယ် ထည့်မကြည့်ရသေးတဲ့ Array တွေ Object တွေလည်း ကျန်ပါသေးတယ်။ ဒါတွေလည်း Truthy ပါပဲ။ ဒါကြောင့် ဘာတွေက Truthy ဖြစ်သလဲ လိုက်မှတ်မယ့်အစား Falsy ဖြစ်တဲ့ (၅) မျိုးကို မှတ်ထားလိုက်ရင် ကျန်တာအားလုံး Truthy ဖြစ်တယ်လို့ ကောက်ချက်ချလို့ရသွားမှာပဲ ဖြစ်ပါတယ်။

အခန်း (၃) – Expressions, Statements & Operators

ကုဒ်တွေစရေးတော့မယ်ဆိုရင် Expression နဲ့ Statement လို့ခေါ်တဲ့ အခြေခံ အသုံးအနှုန်းလေးတွေ အကြောင်း သိထားဖို့လိုပါတယ်။ "နေကောင်းလား" ဆိုတဲ့ အမေးစကားလေးတစ်ခုဟာ Expression ဖြစ်ပြီး "ဘိုဘိုရေ နေကောင်းရင် အပြင်သွားရအောင်" ဆိုတာကတော့ ပြည့်စုံတဲ့ Statement တစ်ခုဖြစ်သွားပါပြီ။ Expression ဆိုတာ တွက်ချက်မှုလေးတွေ၊ လုပ်ဆောင်ချက်လေးတွေပါ။ ဥပမာ – `1 + 2` ဟာ Expression ဖြစ်ပါတယ်။ `num > 10` ဟာလည်း Expression တစ်ခုပါပဲ။ အဲ့ဒီတွက်ချက်မှုလေးတွေ၊ လုပ်ဆောင်ချက်လေးတွေပေါ် မူတည်ပြီး ဘာလုပ်ရမလဲ ညွှန်ကြားသတ်မှတ်လိုက်တဲ့အခါ Statement ဖြစ်သွားပါတယ်။ ဥပမာ –

JavaScript

```
let sum = 1 + 2
```

ဒါဟာ Statement တစ်ခုပါ။ 1 + 2 ကိုပေါင်းလို့ရလာတဲ့အဖြေကို `sum` ဆိုတဲ့ Variable ထဲမှာ ထည့်လိုက်ပါလို့ ညွှန်ကြားလိုက်ခြင်း ဖြစ်ပါတယ်။ ဒါကြောင့် Expression ဆိုတာ လုပ်ဆောင်ချက်တစ်ခုဖြစ်ပြီး Statement ဆိုတာ ညွှန်ကြားချက်ဖြစ်တယ် လို့ အတိုချုပ်မှတ်နိုင်ပါတယ်။

JavaScript Statement တစ်ကြောင်းရဲ့အဆုံးသတ်ကို Semicolon နဲ့ ပိတ်ပေးလို့ ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
let sum = 1 + 2;
let greet = "Hello World";
```

တစ်ချို့ Language တွေမှာ ဒီလိုနောက်ဆုံးကနေ Semicolon နဲ့ပိတ်ပေးဖို့ မဖြစ်မနေလိုအပ်ပါတယ်။

JavaScript မှာတော့ Optional ပါ။ Semicolon ပိတ်ပေးလို့ရသလို၊ မပိတ်ဘဲရေးလို့လည်းရပါတယ်။ ကုဒ် တစ်လိုင်းထဲမှာ Statement နှစ်ခုသုံးခု ရောရေးချင်ရင်တော့ Semicolon နဲ့ မဖြစ်မနေပိတ်ပေးရတော့မှာ ပါ။ ဒီလိုပါ –

JavaScript

```
let n = 5; n + 10;
```

Statement တွေအကြောင်းပြောရင်းနဲ့ ထည့်ပြောဖို့လိုလာတာက Input/Output Statement တွေပါ။ လောလောဆယ် Output Statement တစ်ခုကိုအရင်လေ့လာထားကြပါမယ်။ JavaScript မှာ ရလဒ်တွေကို ဖော်ပြဖို့အတွက် `console.log()` လို့ခေါ်တဲ့ Method တစ်မျိုးကို သုံးရပါတယ်။ ကွင်းစကွင်းပိတ်ထဲမှာ ဖော်ပြစေလိုတဲ့ တန်ဖိုးကို ပေးနိုင်ပါတယ်။ ဥပမာ –

JavaScript

```
let num = 10

console.log("Hello World")
console.log(1 + 2)
console.log(num + 5)
console.log(`The num is ${num}`)
```

`console.log()` Method မပါဘဲ Expression တစ်ခုကို Console ထဲမှာ ဒီအတိုင်းရိုက်ထည့်ရင်လည်း ရလဒ်ကို မြင်တွေ့ရလေ့ရှိပါတယ်။ ဒါက Console ထဲမှာ တိုက်ရိုက်ရေးစမ်းနေတာမို့လို့ပါ။ တစ်ကယ့်လက်တွေ့မှာ ကိုယ်က ရိုက်ထုတ်ဖော်ပြပေးပါလို့ မပြောရင် ရလဒ်ကို တွေ့မြင်ရမှာ မဟုတ်ပါဘူး။ လိုအပ်လို့ရလဒ်ကို မြင်ချင်တယ်ဆိုရင် `console.log()` ကိုသုံးပြီး ဖော်ပြဖို့ ညွှန်ကြားပေးရမှာပဲဖြစ်ပါတယ်။

Comments

Statement တွေအကြောင်းပြောရင်းနဲ့ Code Comment တွေအကြောင်းလည်း ထည့်ပြောဖို့ လိုလာပါတယ်။ ကုဒ်တွေရေးတဲ့အခါ မှတ်ချက် Comment တွေလည်း ထည့်ရေးလို့ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
let count = 10       // Number of items per page
```

နမူနာမှာ မြင်တွေ့ရတဲ့ Slash သကေ်တနှစ်ခုနောက်က ကုဒ်ကို Comment လို့ခေါ်ပါတယ်။ ရေးထားတဲ့ ကုဒ်ကို ကွန်ပျူတာက အလုပ်လုပ်တဲ့အခါ ဒီ Comment တွေကို ထည့်အလုပ်လုပ်မှာ မဟုတ်ပါဘူး။ ဒီ နည်းနဲ့ မှတ်ချက်တွေ ကုဒ်ထဲမှာ ရောရေးထားလိုရင် ရနိုင်ပါတယ်။

Comment က တစ်ကြောင်းထက်ပိုမယ်ဆိုရင် Block Comment ရေးထုံးကို အသုံးပြုနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
/*
    Number of items per page.
    Change this value to determine how many
    items to be shown on home page.
*/

let count = 10
```

ရှေ့က /* နဲ့ဖွင့်ပြီးနောက်က */ နဲ့ ပိတ်ပေးလိုက်တာပါ။ ဒီအဖွင့်အပိတ်ကြားထဲမှာ ရေးထားတာတွေက လည်း Comment တွေပါပဲ။

Comment တွေဟာ ထည့်အလုပ်လုပ်မယ့်ကုဒ် မဟုတ်ပေမယ့် အရေးကြီးပါတယ်။ ရေးထားတဲ့ကုဒ်ရဲ့ အဓိပ္ပါယ်ကို ဒီလို Comment လေးတွေနဲ့ ရှင်းပြရတာပါ။ မဟုတ်ရင် ဘာလို့ဒီလိုရေးခဲ့လဲဆိုတာ နောင်မှာ ကိုယ်တိုင် မေ့သွားတာ၊ မမှတ်မိတော့တာမျိုးတွေ ဖြစ်နိုင်ပါတယ်။ ကိုယ်ရည်ရွယ်ချက်နဲ့ကိုယ် ရေးထားလို့ ကိုယ့်အတွက် အဆင်ပြေပေမယ့်၊ သူများကလာကြည့်တဲ့အခါ ဒီကုဒ်က ဘာကိုဆိုလိုမှန်း နားမလည်ဘူးဆို တာမျိုးတွေ ဖြစ်နိုင်ပါတယ်။ လိုအပ်တဲ့နေရာမှာ Comment လေးတွေထည့်ပြီး ရှင်းပြထားခဲ့မယ်ဆိုရင် ပို ပြီးတော့ ဖတ်ရှုနားလည်ရလွယ်တဲ့ကုဒ်တွေ ဖြစ်သွားစေနိုင်ပါတယ်။

Operators

Expression တွေ Statement တွေ တည်ဆောက်ဖို့အတွက် Operator တွေလိုအပ်ပါတယ်။ အပေါင်း၊ အ နှုတ်၊ အမြှောက်၊ အစား စတဲ့တွက်ချက်မှုပိုင်း Operator တွေလိုအပ်သလို၊ တန်ဖိုးသတ်မှတ်ခြင်း၊ နှိုင်းယှဉ် ခြင်းတို့လို လုပ်ငန်းတွေအတွက်လည်း Operator တွေလိုအပ်ပါတယ်။ ဒီ Operator တွေအကြောင်း ဆက် ကြည့်ကြပါမယ်။

Arithmetic Operators

အခြေခံ ပေါင်းနှုတ်မြှောက်စား Operator တွေကတော့ ရှင်းပါတယ်။ ပေါင်းဖို့အတွက် + သင်္ကေတကိုသုံးရပါတယ်။ နှုတ်ဖို့အတွက် - သင်္ကေတကို သုံးရပါတယ်။ မြှောက်ဖို့အတွက် * သင်္ကေတကို သုံးရပြီး၊ စားဖို့အတွက် / သင်္ကေတကို သုံးရပါတယ်။

+ သင်္ကေတမှာတော့ ထူးခြားချက်နှစ်ခု ရှိပါတယ်။ သူ့ကို ကိန်းဂဏန်းတွေကို ပေါင်းဖို့အတွက် သုံးနိုင်သလို၊ String တွေ တွဲဆက်ဖို့လည်း သုံးနိုင်ပါတယ်။ String Concatenation လို့ ခေါ်ပါတယ်။ ဥပမာ -

JavaScript

```
1 + 1           // 2
'a' + 'b'       // ab
1 + 'a'         // 1a
'a' + 1         // a1
```

နောက်ပိုင်းကုဒ်နမူနာတွေမှာ လိုအပ်ရင် ရလဒ်ကို အခုလို Comment အနေနဲ့တွဲပြီးတစ်ခါထဲ ဖော်ပြပေးပါမယ်။ ကိုယ်တိုင်ရေးစမ်းတဲ့အခါ အဲ့ဒီ Comment တွေထည့်ရေးစရာမလိုပါဘူး။ စမ်းကြည့်လို့ရတဲ့ရလဒ်နဲ့ ဒီမှာပြထားတဲ့ရလဒ် တူမတူ နှိုင်းယှဉ်နိုင်ဖို့အတွက်သာ ထည့်ပေးထားတာပါ။ နောက်ထပ်ထူးခြားချက်ကတော့ + သင်္ကေတကို သုံးပြီး ကိန်းဂဏန်းမဟုတ်တဲ့ တန်ဖိုးတွေကို ကိန်းဂဏန်းဖြစ်အောင် ပြောင်းလို့ရပါတယ်။

JavaScript

```
+'5'      // 5
+'a'      // NaN
```

နမူနာ String တန်ဖိုးဖြစ်တဲ့ 5 ကို ရှေ့ကနေ + သင်္ကေတတွဲပေးလိုက်တဲ့အခါ ကိန်းဂဏန်း 5 ဖြစ်သွားပါတယ်။ String တန်ဖိုး a ကို + သင်္ကေတနဲ့ ကိန်းဂဏန်းပြောင်းဖို့ ကြိုးစားလိုက်တဲ့အခါမှာတော့ ပြောင်းလို့မရတဲ့အတွက် NaN ကိုရတာပဲ ဖြစ်ပါတယ်။

JavaScript

```
'5' + 5         // 55
+'5' + 5        // 10
```

ဒီနမူနာရဲ့ ပထမလိုင်းမှာ String 5 နဲ့ Number 5 ကို ပေါင်းခိုင်းတဲ့အခါ 55 ရပါတယ်။ ဒုတိယလိုင်းမှာတော့ + သင်္ကေတနဲ့ String 5 ကို ကိန်းဂဏန်းဖြစ်အောင် အရင်ပြောင်းလိုက်လို့ 10 ရတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

ဒီပေါင်းနှုတ်မြောက်စား သင်္ကေတတွေဟာ ကိန်းဂဏန်းလိုအပ်တဲ့နေရာမှာ ပေးလာတဲ့တန်ဖိုးကို ကိန်းဂဏန်းဖြစ်အောင် အလိုအလျောက်ပြောင်းပြီး အလုပ်လုပ်ပေးကြပါတယ်။ ဒီလိုအလိုအလျောက် ပြောင်းပြီး အလုပ်လုပ်ပေးတာဟာ + သင်္ကေတရဲ့ String Concatenation လုပ်ပေးတဲ့သဘောနဲ့ တွေ့တဲ့အခါ အခုလို ကမောက်ကမတွေကို ကြုံတွေ့ရတတ်ပါတယ်။

JavaScript

```
"11" + 1        // 111
"11" - 1        // 10
```

String 11 ကို Number 1 နဲ့ ပေါင်းတဲ့အခါ 111 ကိုရပါတယ်။ + သင်္ကေတက Number 1 ကို String ပြောင်းပြီး တွဲဆက်ပေးလိုက်လို့ပါ။ String 11 ထဲကနေ 1 နှုတ်လိုက်တဲ့အခါကျတော့ 10 ရနေပါတယ်။ String 11 ကို Number 11 ဖြစ်အောင် အလိုအလျောက် ပြောင်းပြီး အလုပ်လုပ်သွားတဲ့ အတွက်ကြောင့်ပဲ ဖြစ်ပါတယ်။

JavaScript ရဲ့ ဒီကမောက်ကမကိစ္စလေးတွေဟာ ပရိဂရမ်မာတွေရဲ့ ကြားထဲမှာ ရယ်သွမ်းသွေးစရာလေးတွေ ဖြစ်နေကြပါ။ ရယ်သွမ်းသွေးတာ သွေးလို့ရပါတယ်၊ ဒါပေမယ့် ဘာကြောင့် ဒီလိုဖြစ်တာလဲ ဆိုတဲ့ အကြောင်းရင်းကို သိထားဖို့တော့ လိုပါတယ်။ အကြောင်းရင်းကို သိထားမယ်ဆိုရင် ဖြစ်ချင်ရာတွေဖြစ်နေတာမျိုး မဟုတ်ဘဲ သူ့အဓိပ္ပါယ်နဲ့သူ အလုပ်လုပ်နေတာဆိုတာကို သတိပြုမိမှာပဲ ဖြစ်ပါတယ်။

ဖြည့်စွက်မှတ်သားရမယ့် Operator နှစ်ခုကတော့ % သင်္ကေတနဲ့ ** သင်္ကေတတို့ပဲ ဖြစ်ပါတယ်။ % သင်္ကေတကို အကြွင်းရှာဖို့ သုံးပါတယ်။ ပုံမှန်အားဖြင့် ရိုးရိုးစားလိုက်ရင် ရလဒ်ကို ဒဿမကိန်းနဲ့ ပြန်ရမှာပါ။ % သင်္ကေတကတော့ အကြွင်းကိုကိန်းပြည့်အနေနဲ့ ပြန်ပေးပါမယ်။ ဥပမာ –

JavaScript

```
5 / 3     // 1.6666666666666667
5 % 3     // 2
```

5 ကို 3 နဲ့စားတဲ့အခါ မပြတ်တဲ့အတွက် ကြွင်း 2 ကို ရလဒ်အနေနဲ့ ပြန်ရတာပါ။ ** သင်္ကေတကိုတော့ Exponent Operator ရှာဖို့သုံးရပါတယ်။ ဥပမာ –

JavaScript

```
2 ** 2     // 4
2 ** 3     // 8
```

နောက်ထပ် ဖြည့်စွက်မှတ်သားရမယ့် Operator နှစ်ခုကတော့ ++ နဲ့ -- ဖြစ်ပါတယ်။ ++ ကို တစ်တိုးဖို့သုံးပြီး -- ကို တစ်နှုတ်ဖို့သုံးပါတယ်။ ဥပမာ –

JavaScript

```
let a = 2

a++         // 2
a           // 3
```

နမူနာအရ a ရဲ့ မူလတန်ဖိုး 2 ဖြစ်ပါတယ်။ ++ နဲ့ တစ်တိုးလိုက်လို့ တစ်တိုးသွားပေမယ့် အဖြေရလဒ်အနေနဲ့ 2 ပဲ ရတာကို သတိပြုပါ။ a တန်ဖိုးကို ထပ်ထုတ်ကြည့်တဲ့အခါ တစ်တိုးပြီးတန်ဖိုး 3 ကို ရပါတယ်။ ++ သင်္ကေတကို နောက်မှာထားတဲ့အခါ အလုပ်အရင်လုပ်ပြီးမှ တစ်တိုးတယ် ဆိုတဲ့သဘော ဖြစ်ပါတယ်။

JavaScript

```
let b = 3

++b         // 4
b           // 4
```

ဒီနမူနာမှာတော့ b ရဲ့ မူလတန်းဖိုး 3 ဖြစ်ပါတယ်။ ++ နဲ့ တစ်တိုးလိုက်တဲ့အတွက် အမှန်တစ်ကယ်တစ်တိုးသွားသလို ရလဒ်အနေနဲ့လည်း တစ်တိုးပြီးတန်ဖိုးကို ရပါတယ်။ b ရဲ့ လက်ရှိတန်ဖိုးကို ကြည့်လိုက်ရင်လည်း တစ်တိုးပြီးတန်ဖိုးကိုပဲ ရပါတယ်။ ++ သင်္ကေတကို ရှေ့မှာထားတဲ့အခါ အရင်တစ်တိုးပြီးမှ အလုပ်လုပ်တယ် ဆိုတဲ့ သဘောကို သတိပြုရမှာပါ။

-- လည်း ဒီသဘောပါပဲ။ တစ်နှုတ်ပေးပါတယ်။ -- ကိုနောက်မှာထားရင် အလုပ်အရင်လုပ်ပြီးမှ တစ်နှုတ်ပြီး -- ကိုရှေ့မှာထားရင် တစ်နှုတ်ပြီးတော့မှ အလုပ်လုပ်ပေးသွားမှာပဲ ဖြစ်ပါတယ်။
ဒီ Operator တွေကို နှစ်ခုထက်ပိုတဲ့ တန်ဖိုးတွေနဲ့ တွဲသုံးလို့ရပါတယ်။ ဥပမာ –

```JavaScript
let result = 4 + 5 - 1 * 3 / 2
```

ဒီလိုတွဲသုံးတဲ့အခါ Operator တူရင် ဘယ်ကနေညာ ကို အလုပ်လုပ်ပါတယ်။ ဒါကြောင့် ဒီနှစ်ခုဟာ ရလဒ်တူမှာ မဟုတ်ပါဘူး။

```JavaScript
"$" + 4 + 5     // $45
4 + 5 + "$"     // 9$
```

ဘာကြောင့်လဲဆိုတော့ ပထမတစ်ခုမှာ $ နဲ့ 4 ကို အရင်တွဲလို့ $4 String ကိုရပြီးမှ $4 နဲ့ 5 ကိုထပ်တွဲလို့ နောက်ဆုံးရလဒ် $45 ဖြစ်သွားတာပါ။ နောက်တစ်ခုမှာတော့ 4 နဲ့ 5 ကို အရင်ပေါင်းလို့ Number 9 ရပြီးမှ 9 နဲ့ $ ကိုတွဲတဲ့အတွက် 9$ ဖြစ်သွားတာပါ။ ဒါလည်း JavaScript ရဲ့ ကမောက်ကမ သဘောသဘာဝတစ်ခုပါပဲ။ လေ့လာသူက ဘာကြောင့် ဒီလိုဖြစ်ရတာလဲ နားလည်ထားမယ်ဆိုရင်တော့ ပြဿနာမရှိတော့ပါဘူး။

လိုအပ်ရင် ဝိုက်ကွင်းအဖွင့်အပိတ်တွေ တွဲသုံးနိုင်ပါတယ်။ ဝိုက်ကွင်းအဖွင့်အပိတ်ပါရင် အထဲကအလုပ်ကို အရင်လုပ်မှာဖြစ်လို့ စောစောကနမူနာကို အခုလို ပြင်လိုက်လို့ ရနိုင်ပါတယ်။

```JavaScript
"$" + (4 + 5)   // $9
```

ပုံမှန်အားဖြင့် ဘယ်ကနေညာကို အစီအစဉ်အတိုင်း အလုပ်လုပ်ပေမယ့်၊ အခုတော့ ဝိုက်ကွင်းပါသွားလို့ အထဲကအလုပ်ကို အရင်လုပ်လိုက်တဲ့အတွက်ကြောင့်ရလဒ်က $9 ဖြစ်သွားတာပါ။ ဘယ်ကနေညာကို အလုပ်လုပ်တယ်ဆိုတာ Operator အဆင့်တူမှ လုပ်တာပါ။ Operator တွေမှာ အစီအစဉ်အဆင့် Precedent ရှိပါတယ်။ ဒီလိုပါ –

1. ဝိုက်ကွင်း
2. ++, --
3. *, /, %
4. +, -

ဒါရှိရှိသမျှ Operator အားလုံးရဲ့အစီအစဉ်တော့ မဟုတ်သေးပါဘူး။ တွဲအသုံးများမယ့် Operator တွေရဲ့ အစီအစဉ်ကိုပဲ အကျဉ်းချုပ်ပြောလိုက်တာပါ။ ဒီအစီအစဉ်အရ ကွင်းထဲက အလုပ်ကို ပထမဆုံးလုပ်ပါမယ်။ ပြီးတဲ့အခါ ++, -- ရှိအရင် ဆက်လုပ်ပါမယ်။ ပြီးတဲ့အခါ အမြှောက်၊ အစားနဲ့ အကြွင်းရှာတဲ့ အလုပ်တွေကို လုပ်ပါမယ်။ အပေါင်းနဲ့ အနှုတ်က နောက်ဆုံးမှ လုပ်မှာဖြစ်ပါတယ်။

JavaScript

```
3 - 1 + 2 * 5 / 4         // 4.5
```

ဒီနမူနာကို အလုပ်လုပ်တဲ့အခါ 3 ကနေ 1 ကို နှုတ်၊ 2 နဲ့ပေါင်း၊ 5 နဲ့မြှောက်၊ 4 နဲ့စားဆိုပြီး ဘယ်ကနေ ညာကို အစီအစဉ်အတိုင်း လုပ်သွားမှာ မဟုတ်ပါဘူး။ ထပ်ပြောတာပါ၊ Operator Precedent တူမှသာ ဘယ်ကနေညာကို လုပ်တာပါ။ ဒီနေရာမှာတော့ အဆင့်မတူလို့ Precedent မြင့်တဲ့ အမြှောက်နဲ့အစားကို အရင်ဆုံးအလုပ်လုပ်ပါတယ် (2 * 5 / 4 = 2.5) ရပါတယ်။ ပြီးတော့မှ ရလာတဲ့ရလဒ်ကိုသုံးပြီး အပေါင်းနဲ့ အနှုတ်ကို ဆက်လုပ်တာပါ။ (3 – 1 + 2.5 = 4.5) ဖြစ်တဲ့အတွက် နောက်ဆုံးရလဒ် 4.5 ကိုရတာပါ။

```
(3 - 1 + 2) * 5 / 4       // 5
```

ဒီတစ်ခါတော့ အဖြေက 5 ဖြစ်သွားပါပြီ။ 3 – 1 + 2 ကို အရင်အလုပ်လုပ်အောင် ကွင်းခတ်ပေးလိုက်တဲ့ အတွက်ကြောင့်ပါ။

Assignment Operators

Equal သင်္ကေတဟာ ညီတယ်ဆိုတဲ့ အဓိပ္ပါယ်မဟုတ်ဘူး၊ တန်ဖိုးသတ်မှတ်ဖို့သုံးရတဲ့ Assignment Operator ဖြစ်တဲ့အကြောင်း ပြောခဲ့ပြီးဖြစ်ပါတယ်။

JavaScript

```
let num = 1          // let num ← 1
```

ဒါဟာ num Variable ဟာ 1 နဲ့ ညီတယ်ဆိုတဲ့ အဓိပ္ပါယ်မဟုတ်ပါဘူး။ num Variable ထဲမှာ 1 ဆိုတဲ့ တန်ဖိုးကို ထည့်သွင်းလိုက်တာပါ။

JavaScript

```
num = 2
```

ဒါဆိုရင် num Variable ထဲကတန်ဖိုး ပြောင်းသွားပါပြီ။ 2 ဖြစ်သွားပါပြီ။ num Variable ထဲကတန်ဖိုးကို ပြောင်းချင်တာ မဟုတ်ဘဲ ပေါင်းပြီးတိုးလိုက်ချင်တာဆိုရင် ဒီလိုရေးရပါလိမ့်မယ်။

JavaScript

```
num = num + 3
```

ဒီ Assignment လုပ်ငန်းစဉ်ဟာ အထက်မှာပြောခဲ့တဲ့ Arithmetic လုပ်ငန်းစဉ်နဲ့ ပြောင်းပြန်ပါ။ ပေါင်းနှုတ်မြှောက်စား လုပ်ငန်းတွေမှာ Precedent တူရင် ဘယ်ကနေညာကို အလုပ်လုပ်ပေမယ့်၊ Assignment လုပ်ငန်းစဉ်မှာတော့ ညာကနေ ဘယ်ကိုအလုပ်လုပ်ပါတယ်။ ဒါကြောင့် ညာဘက်က num + 3 ကို အရင် အလုပ်လုပ်လိုက်တဲ့အခါ num ထဲက တန်ဖိုး 2 ကို 3 နဲ့ ပေါင်းလိုက်လို့ 5 ရမှာ ဖြစ်ပါတယ်။ ရလာတဲ့ ရလဒ်ကို num Variable ထဲ ထည့်လိုက်လို့ num Variable ထဲက လက်ရှိတန်ဖိုး 5 ဖြစ်သွားပါပြီ။ ဒါကြောင့် မူလတန်ဖိုးမှာ ထပ်တိုးပြီး ပေါင်းထပ်လိုက်တဲ့ သဘောမျိုးကို ရတာပါ

ဒီသဘောမျိုးရဖို့အတွက် အခုလိုအတိုကောက်ရေးလို့ ရနိုင်ပါသေးတယ်။

```JavaScript
let num = 2
num += 3        // num = num + 3
```

ဒါဆိုရင် မူလကြေညာချက်အရ num ရဲ့တန်ဖိုး 2 ဖြစ်ပြီး += Assignment Operator နဲ့ 3 ကိုထပ်တိုးပြီး ပေါင်းထည့်လိုက်တာပါ။ ဒါကြောင့် 5 ဖြစ်သွားပါပြီ။ ဒီလိုမျိုး တစ်ခြားအတိုကောက် Assignment Operator တွေ ရှိကြပါသေးတယ္။ -=, *=, /=, %= စသည်ဖြင့်။ အားလုံးက သဘောသဘာဝအားဖြင့် += နဲ့ အတူတူပါပဲ။ ပေါင်းထည့်ခြင်းအစား၊ နှုတ်ထည့်ခြင်း၊ မြှောက်ထည့်ခြင်း စသည်ဖြင့် ကွာသွားတာသာ ဖြစ်ပါတယ်။

ဒီလိုလည်းရေးလို့ရနိုင်ပါသေးတယ်။

```JavaScript
let a = b = c = 5
```

ဒါဆိုရင်တော့ Assignment ရဲ့ ညာကနေ ဘယ်ကိုအလုပ်လုပ်တဲ့ သဘောအရ၊ ညာဘက်အစွန်ဆုံးက 5 ကို c ထဲမှာ အရင် Assign လုပ်ပါတယ်။ ပြီးတဲ့အခါ c ရဲ့တန်ဖိုးကို b ထဲမှာ Assign လုပ်ပါတယ်။ ပြီးတော့မှ b ရဲ့ တန်ဖိုးကို a ထဲမှာ Assign လုပ်လိုက်လို့ အခုဆိုရင် a, b, c အားလုံးရဲ့ တန်ဖိုးတွေဟာ 5 တွေ ဖြစ်သွားကြပါပြီ။

Comparison Operators

ပရိုဂရမ်တစ်ခုတည်ဆောက်တဲ့အခါ အစဉ်အတိုင်း အလုပ်လုပ်သွားတဲ့ Statement တွေစုဖွဲ့ပြီး ကွန်ပျူတာကို ညွှန်ကြားချက်တွေ ပေးနေယုံနဲ့ မလုံလောက်ပါဘူး။ အခြေအနေပေါ်မူတည်ပြီး ဆုံးဖြတ်နိုင်စွမ်း ရှိအောင်လည်း ဖန်တီးပေးရပါဦးမယ်။ ဘယ်လောက်ပဲ ရှုပ်ထွေးတဲ့ပရိုဂရမ်ကြီး ဖြစ်နေပါစေ၊ ဆုံးဖြတ်ချက်တွေ ချတဲ့အခါ မှားသလား၊ မှန်သလား ဒီနှစ်မျိုးကိုပဲ ကြည့်သွားမှာပါ။ ကွန်ပျူတာကိုယ်တိုင်က 0 နဲ့ 1 တွေစုဖွဲ့ပါဝင်တဲ့ Binary ကိုပဲနားလည်တာဖြစ်သလို၊ ပရိုဂရမ်မာတွေ ဖန်တီးလိုက်တဲ့ ပရိုဂရမ်တွေကလည်း 0 လား၊ 1 လား၊ မှားသလား၊ မှန်သလား ဆိုတာကိုပဲကြည့်ပြီး ဆုံးဖြတ်အလုပ်လုပ်သွားမှာပါ။

ဒီနေရာမှာ အရေးပါလာတာကတော့ Comparison Operator ခေါ် တန်ဖိုးတွေကို နှိုင်းယှဉ်ပြီး true or false ပြန်ပေးနိုင်တဲ့ လုပ်ဆောင်ချက်တွေပါပဲ။ တန်ဖိုးတွေက ဘယ်လောက်ဆန်းပြားနေပါစေ၊ နှိုင်းယှဉ်မှုက ဘယ်လောက်ရှုပ်ထွေးနေပါစေ၊ သူပြန်ပေးမယ့် အဖြေကတော့ နှစ်ခုထဲက တစ်ခုပါပဲ။ true သို့ မဟုတ် false ဆိုတဲ့ Boolean ရလဒ်ကိုပဲ ပြန်ပေးမှာပါ။

Comparison Operator တွေထဲမှာ အရေးအကြီးဆုံးနဲ့ မကြာခဏအသုံးအများဆုံးဖြစ်မှာကတော့ Equal To Operator ပဲဖြစ်ပါတယ်။ Equal To Operator နှစ်မျိုးရှိပါတယ်။ == (Double Equal) နဲ့ === (Triple Equal) တို့ပဲဖြစ်ပါတယ်။ ရိုးရိုး = (Equal) သင်္ကေတကို Assignment အတွက်သုံးတဲ့ဆိုတာ ပြောခဲ့ပြီးပါပြီ။ တစ်ကယ်တမ်း တန်ဖိုးတွေ ညီသလားနှိုင်းယှဉ်လိုရင် Double Equal သို့မဟုတ် Triple Equal သင်္ကေတတွေကိုသုံးရတာပါ။ Double Equal ကို ရိုးရိုး Equal လို့ခေါ်ပါတယ်။ တန်ဖိုးတူရင် ရပါပြီ။ Type အတိအကျ တူစရာမလိုဘူး၊ တူသလိုလိုရှိရင်ရပြီလို့ မြင်သာအောင် ပြောချင်ပါတယ်။ Triple Equal ကိုတော့ Strict Equal လို့ခေါ်ပါတယ်။ တူသလိုလိုရှိယုံနဲ့ မရတော့ပါဘူး အတိအကျတူမှ ရတော့မှာပါ။ ကုဒ် နမူနာလေး တစ်ချို့နဲ့ ကြည့်ကြည့်ပါ။

JavaScript

```
5 == "5"        // true
5 === "5"       // false
```

နမူနာမှာ Number 5 နဲ့ String 5 ကို ရိုးရိုး Equal နဲ့ ညီသလားလို့ နှိုင်းယှဉ်တဲ့အခါ ညီတယ်ဆိုတဲ့အဖြေ true ကို ပြန်ရပါတယ်။ Type မတူပေမယ့်၊ Type Coercion သဘောသဘာဝနဲ့ အလိုအလျောက် ပြောင်းပြီး နှိုင်းယှဉ်လို့ရနေတဲ့အတွက် ညီတယ်လို့ ပြောနေတာပါ။ တန်ဖိုးတူရင်ရပြီး Type တူစရာမလိုဘူးဆိုတာ ဒါမျိုးကို ပြောတာပါ။

Strict Equal ကတော့ မရပါဘူး Number 5 နဲ့ String 5 ညီသလားနှိုင်းယှဉ်တဲ့အခါ မညီဘူးဆိုတဲ့အဖြေ false ကို ပြန်ပေးပါတယ်။ ရိုးရိုး Equal ကို မသုံးသင့်တဲ့ Operator လို့ သတ်မှတ်ကြပါတယ်။ မတိကျတဲ့အတွက် မမျှော်မှန်းနိုင်တဲ့ အမှားတွေကို ဖြစ်ပေါ်စေနိုင်တဲ့ အန္တရာယ်ရှိလို့ပါ။ လက်တွေ့ကုဒ်ထဲမှာ တန်ဖိုးတွေ ညီမညီ နှိုင်းယှဉ်ဖို့ လိုအပ်လာတဲ့အခါ Strict Equal ကိုသာ အသုံးပြုသင့်တယ်လို့ ကျွမ်းကျင်သူ ပညာရှင်များက အကြံပြုထားကြပါတယ်။

နောက်ထပ် Comparison Operator အနေနဲ့ အသုံးများမှာကတော့ Not Equal ဖြစ်ပါတယ်။ Exclamation သင်္ကေတနဲ့ Equal သင်္ကေတကိုတွဲပြီး != ရေးပေးရပါတယ်။ သူမှာလည်း ရိုးရိုးနဲ့ Strict နှစ်မျိုးရှိပါတယ်။ ဒီလိုပါ။

JavaScript

```
5 != "5"        // false
5 !== "5"       // true
```

ရိုးရိုး Not Equal က Number 5 နဲ့ String 5 မညီဘူးလားလို့ နှိုင်းယှဉ်ကြည့်တဲ့အခါ သူ့အမြင်မှာ ညီနေတဲ့အတွက် ကိုယ်မေးတာမှားနေပါတယ်။ ဒါကြောင့် false ကို ပြန်ပေးပါတယ်။ Strict Not Equal ကတော့ Number 5 နဲ့ String 5 မညီဘူးလားလို့ နှိုင်းယှဉ်ကြည့်လိုက်တဲအခါ မညီတာမှန်ကန်နေတဲ့အတွက် true ကို ပြန်ပေးပါတယ်။ ဒီနေရာမှာလည်း ရိုးရိုး Not Equal != ကို အသုံးမပြုသင့်ဘူးလို့ ဆိုရပါမယ်။ လိုအပ်လာတဲ့အခါ Strict Not Equal !== ကိုအသုံးပြုရမှာပါ။

ကျန် Comparison Operator တွေကတော့ Greater Than, Less Than, Greater Than or Equal, Less Than or Equal တို့ပဲ ဖြစ်ပါတယ်။ ကုဒ်နမူနာလေးတစ်ချို့နဲ့ ဖော်ပြပေးပါမယ်။

JavaScript

```
5 > 5     // false
5 >= 5    // true
```

ပထမတစ်ကြိမ်မှာ 5 ဟာ 5 ထက်ကြီးသလားလို့ Greater Than နဲ့ နှိုင်းယှဉ်ကြည့်တဲ့အခါ မှားနေပါတယ်။ 5 ဟာ 5 ထက်မကြီးပါဘူး။ ဒါကြောင့် false ဆိုတဲ့ရလဒ်ကို ရပါတယ်။ နောက်တစ်ကြိမ်မှာတော့ >= ကိုသုံးပြီး Greater Than or Equal ဆိုတဲ့အဓိပ္ပါယ်နဲ့နှိုင်းယှဉ်ကြည့်လိုက်ပါတယ်။ ဒီအခါမှာတော့ 5 ဟာ 5 ထက်မကြီးပေမယ့် ညီနေတဲ့အတွက် true ဆိုတဲ့အဖြေကို ရပါတယ်။

```JavaScript
let num = 3
num++
num < 5         // true
num += 2
num <= 5        // false
```

ဒီကုဒ်ကိုတော့ ကိုယ့်ဘာသာပဲ လေ့လာကြည့်လိုက်ပါ။ စာနဲ့ပြန်ရေးပြီး ရှင်းပြနေစရာ မလိုတော့ဘူးလို့ ယူဆပါတယ်။

Logical Operators

Boolean ဆိုတဲ့ အမှား နဲ့ အမှန် နှစ်မျိုးပဲရှိတဲ့စနစ်ဟာ အလွန်ရိုးရှင်းလှပါတယ်။ တန်ဖိုးတွေကို နှိုင်းယှဉ်ပြီး အမှား/အမှန် အဖြေရှာပေးနိုင်တဲ့ Comparison လုပ်ဆောင်ချက်ဟာလည်း သူချည်းဆိုရင် ရိုးရိုးလေးပါပဲ။ သူလုပ်ပေးနိုင်တာ နှစ်ခုထဲက တစ်ခုကို ရွေးပေးတာလေးပဲလေ။ အခုဆက်လေ့လာကြမယ့် Logic လို့ ခေါ် တဲ့ ကြောင်းကျိုးဆက်စပ်မှု သဘောသဘာဝဟာလည်း ဆန်းပြားလှတဲ့သဘော မဟုတ်ပါဘူး။ နှစ်ခုရှိတယ်၊ တစ်ခုမှန်ရင် ရပြီလား၊ နှစ်ခုလုံးမှန်မှ ရမှာလား၊ ဒါလေးပါပဲ။

ဒါပေမယ့် အဲ့ဒီရိုးရိုးလေးပါဆိုတဲ့ Boolean, Comparison နဲ့ Logic တို့ကို ပေါင်းစပ်လိုက်တဲ့ အခါမှာတော့၊ မူလအသိဉာဏ်မရှိတဲ့ ကွန်ပျူတာ၊ သင်မပေးရင် ဘာမှမတတ်တဲ့ ကွန်ပျူတာကို၊ အသိဉာဏ် ရှိသကဲ့သို့ ရှုပ်ထွေးဆန်းကျယ်တဲ့ ဆုံးဖြတ်ချက်တွေ ချမှတ်အလုပ်လုပ်နိုင်အောင် လမ်းညွှန် သင်ကြား ခိုင်းစေ နိုင်သွားမှာပါ။

သိပ်အံ့သြပြီး စိတ်လှုပ်ရှားစရာ ကောင်းလှပါတယ်။ ကွန်ပျူတာပရိုဂရမ်းမင်းရဲ့ အနှစ်သာရဟာ ဒီနေရာမှာ ပေါ်တာဖြစ်ပြီး၊ အဲ့ဒီအနှစ်သာရကို မြင်လိုက်ရတဲ့အချိန်မှာပဲ၊ တစ်ချို့တွေဟာ သိပ်စိတ်လှုပ်ရှားတက်ကြွသွားပြီး ပရိုဂရမ်းမင်းကို ခုံမင်နှစ်သက် မက်မောသွားကြတော့တာပါပဲ။ ဒီလို နှစ်သက်မက်မော သွားမိကြသူတွေဟာ အရည်အချင်းပြည့်ဝတဲ့ ပရိုဂရမ်မာ မလွဲမသွေ ဖြစ်လာမယ်သူတွေပါပဲ။

JavaScript မှာ AND, OR နဲ့ NOT ဆိုတဲ့ Logical Operator (၃) ခုရှိပါတယ်။ NOT ကတော့ ဆန့်ကျင်ဘက်ပါ။ Exclamation သင်္ကေတကို NOT အနေနဲ့ သုံးပါတယ်။ ဒီလိုပါ။

JavaScript

```
!true     // false
!false    // true
```

ဒါကိုနည်းနည်းပိုမြင်သာသွားအောင် အခုလိုလည်း ကြည့်နိုင်ပါတယ်။

JavaScript

```
let num = 3
num < 5         // true
!(num < 5)      // false
```

`num` ရဲ့တန်ဖိုးဟာ 3 ဖြစ်တဲ့အတွက် `num < 5` ဆိုတဲ့နှိုင်းယှဉ်ချက်ကနေ true ကိုပြန်ရပါတယ်။ သူရဲ့ဆန့်ကျင်ဘက် NOT အနေနဲ့ ! သင်္ကေတကို အသုံးပြုလိုက်တဲ့အခါ ဆန့်ကျင်ဘက်ရလဒ်ဖြစ်တဲ့ false ကို ပြန်ရတာပဲ ဖြစ်ပါတယ်။

AND ကတော့ Expression နှစ်ခုရဲ့ ရလဒ်မှာ နှစ်ခုလုံး true ဖြစ်မှ true ကို ပြန်ပေးပြီး၊ မဟုတ်ရင်တော့ false ကို ပြန်ပေးပါတယ်။ နှစ်ခုမှာ နှစ်ခုလုံး မှန်ရမယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ သင်္ကေတအနေနဲ့ Ampersand နှစ်ခုတွဲကို && အသုံးပြုပါတယ်။ ဒီလိုပါ။

JavaScript

```
let num = 3

(num > 3) && (num < 5)    // false && true → false
(num < 5) && (num > 3)    // true && false → false
(num < 5) && (num == 3)   // true && true → true
```

နမူနာမှာပေးထားတဲ့ AND Statement (၃) ခုမှာ Expression နှစ်ခုလုံး true ဖြစ်တဲ့ Statement တစ်ခုသာ နောက်ဆုံးရလဒ် true ဖြစ်တာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။ လက်တွေ့မှာ Expression တွေကို ဝိုက်ကွင်းထဲမှာ မထည့်လည်း ရပါတယ်။

JavaScript

```
let num = 3

num > 3 && num < 5          // false && true → false
num < 5 && num > 3          // true && false → false
num < 5 && num == 3         // true && true → true
```

OR ကတော့ Expression နှစ်ခုမှာ တစ်ခု မှန်တာနဲ့ true ရလဒ်ကို ပြန်ပေးပါတယ်။ သင်္ကေတအနေနဲ့ တော့ Bar Character လေးနှစ်ခုတွဲကို သုံးပါတယ်။ ဒီလိုပါ။

JavaScript

```
let num = 3

num > 3 || num < 5          // false || true → true
num < 3 || num > 3          // false || false → false
num < 5 || num == 3         // true || true → true
```

Express နှစ်ခုမှာ နှစ်ခုလုံး false ဖြစ်တဲ့ Statement လွဲရင် ကျန် Statement တွေမှာ true ရတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

လက်တွေ့မှာ Logic Gate တွေက ဒီထက်ပိုပါသေးတယ်။ AND, OR, NAND, NOR, XOR စသည်ဖြင့် ရှိကြပါတယ်။ NAND ဆိုတာ NOT AND ကို ဆိုလိုတာမို့လို့ သီးခြား Operator မလိုပါဘူး။ NOT နဲ့ AND ကို တွဲသုံးလိုက်လို့ရပါတယ်။ NOR ဆိုတာ NOT OR ကို ဆိုလိုတာမို့လို့ NOT နဲ့ OR ကို တွဲသုံးလိုက်လို့ ရနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
let num = 3

!(num > 3 || num < 5)       // !(false || true → true) → false
!(num < 3 || num > 3)       // !(false || false → false) → true
!(num < 5 || num == 3)      // !(true || true → true) → false
```

XOR ဆိုတာကတော့ Exclusive OR ဆိုတဲ့အဓိပ္ပါယ်ပါ။ OR နဲ့ ဆင်တူပါတယ်၊ Expression နှစ်ခုမှာ တစ်ခုမှန်ရင် true ကိုရပါတယ်။ ဒါပေမယ့် ခြွင်းချက်ကတော့ Expression နှစ်ခုမှာ နှစ်ခုလုံး မမှန်ရပါဘူး။ ဒီအတွက်တော့ JavaScript မှာ အသင့်ပေးထားတဲ့ Operator မရှိပါဘူး။ Pseudocode အနေနဲ့ နမူနာ ပြရရင် ဒီလိုပြလို့ ရနိုင်ပါတယ်။

Pseudocode

```
let num = 3

num > 3 XOR num < 5        // false XOR true → true
num < 3 XOR num > 3        // false XOR false → false
num < 5 XOR num == 3       // true XOR true → false
```

Expression နှစ်ခုလုံး true ဖြစ်နေတဲ့ နောက်ဆုံး Statement ရဲ့ရလဒ် false ဖြစ်နေတာကို သတိပြုရမှာပါ။ ဒါကတော့ ဗဟုသုတအနေနဲ့ ထည့်ပြောတာပါ။ XOR ဆိုတဲ့ Operator က JavaScript မှာ မရှိပါဘူး။

ဒီ Logical Operator တွေကို Expression နှစ်ခုထက်ပိုပြီးတော့လည်း တွဲသုံးလို့ ရနိုင်ပါတယ်။ တစ်ခုနဲ့ တစ်ခုလဲ ပေါင်းစပ်အသုံးပြုလို့ ရနိုင်ပါသေးတယ်။ ဒီလိုပါ။

JavaScript

```
let x = 3
let y = 5

x < y && y > 5 && x == 3
x < y && !(y > 5 && x == 3)
x < y && y > 5 || x == 3
```

ရလဒ်တွေ ထည့်ရေးမပေးတော့ပါဘူး။ ကိုယ်တိုင်စိတ်ကူးနဲ့ အရင်အဖြေရှာကြည့်လိုက်ပါ။ ကိုယ့်စိတ်ကူးအဖြေ မှန်မမှန် သိရဖို့အတွက် အဖြေမှန်ကို Console မှချရေးပြီး နှိုင်းယှဉ်ကြည့်လိုက်ပါ။

အခန်း (၄) – Procedures & Functions

ဆက်လက်ပြီးတော့ Procedure နဲ့ Function လို့ခေါ်တဲ့ သဘောသဘာဝ ဆင်တူပြီး နောက်ထပ် အရေးပါတဲ့ အခြေခံလုပ်ဆောင်ချက်တွေအကြောင်း ဆက်ကြပါမယ်။ Procedure ဆိုတာဟာ Statement တွေကို စုစည်းထားခြင်းပဲ ဖြစ်ပါတယ်။ ဒီလိုစုစည်းဖို့အတွက် တွန့်ကွင်း အဖွင့်အပိတ်ကို အသုံးပြုကြလေ့ရှိပါတယ်။ လက်တွေ့အသုံးချကုဒ်ကို တိုက်ရိုက်ကြည့်ရင် ခေါင်းထဲမှာ ပုံဖော်ရ၊ မြင်ကြည့်ရခက်နေမှာစိုးလို့ Pseudocode လေးတစ်ချို့နဲ့ သဘောသဘာဝပိုင်းကို အရင်ရှင်းပြချင်ပါတယ်။ ဒီလိုပါ –

Pseudocode

```
{
    let a = 1;
    let b = 2;
    print a + b;
}
```

ဒါဟာ Statement တွေကို စုစည်းလိုက်တဲ့ နမူနာ ကုဒ် Block လေးတစ်ခုပါပဲ။ ဒီလို သူ့အစုနဲ့သူ စုဖွဲ့ထားတဲ့ကုဒ် Block ကို အမည်တစ်ခု သတ်မှတ်ပေးနိုင်ပါတယ်။

Pseudocode

```
add {
    let a = 1;
    let b = 2;
    print a + b;
}
```

`add` ဆိုတဲ့အမည်တစ်ခု သတ်မှတ်ပေးလိုက်တာပါ။ အမည်သတ်မှတ်တဲ့အခါ ရိုးရိုး တန်ဖိုးတစ်ခုမဟုတ်ဘဲ Procedure ဆိုတာ ပေါ်လွင်အောင် Keyword လေးထည့်ပေးလိုက်ပါမယ်။

Pseudocode

```
procedure add {
    let a = 1;
    let b = 2;
    print a + b;
}
```

Variable တွေကြေညာပြီးရင် လိုအပ်တဲ့နေရာမှာ အသုံးချနိုင်သလိုပဲ၊ အခုလို Procedure တစ်ခုကြေညာထားမယ်ဆိုရင် လိုအပ်တဲ့နေရာကနေ အသုံးချနိုင်မှာပါ။ ဒီ Procedure လေးက နမူနာအရ a နဲ့ b ကို ပေါင်းပေးတဲ့ Procedure ဖြစ်လို့ နောက်ကို a နဲ့ b ပေါင်းဖို့လိုရင် ထပ်ရေးဖို့ မလိုတော့ပါဘူး။ ဒီ Procedure ကို ခေါ်ယူအသုံးပြုလိုက်ယုံပါပဲ။ ဥပမာ အနေနဲ့ ဒီလိုခေါ်ရတယ်လို့ မြင်ကြည့်လိုက်ပါ –

Pseudocode

```
call add;     // 3
```

`call` Keyword နဲ့ `add` Procedure ကိုအသုံးပြုအလုပ်လုပ်စေလိုက်တဲ့အခါ `add` Procedure ထဲက Statement တွေ တန်းစီအလုပ်လုပ်သွားလို့ 3 ဆိုတဲ့တန်ဖိုးကို ထုတ်ဖော်ပြသတာ ဖြစ်ပါတယ်။ Procedure တွေ Function တွေဟာ သတ်မှတ်လိုက်ယုံနဲ့ အလုပ်မလုပ်သေးပါဘူး။ ခေါ်ယူအသုံးပြုတော့မှသာ အလုပ်လုပ်မှာ ဖြစ်ပါတယ်။

ပေးထားတဲ့နမူနာဟာ ကလေးကလားတော့ ဆန်းလွန်းနေပါလိမ့်မယ်။ ဒီနေရာမှာ နမူနာပြတဲ့ ကုဒ်ကလက်တွေ့ အသုံးဝင်/မဝင် အဓိကမဟုတ်ပါဘူး။ ရှင်းရှင်းလင်းလင်း မြင်သင့်တဲ့ သဘောသဘာဝကို စကတည်းက ရှင်းရှင်းလင်းလင်း မြင်ထားစေဖို့အတွက်ပါ။ အခုဆိုရင် Procedure ဆိုတာဘာလဲ ရှင်းသွားပါပြီ။ Procedure ဆိုတာဟာ Statement တွေကို စုစည်းအမည်ပေးထားလို့ လိုအပ်တဲ့နေရာမှာ ပြန်လည်အသုံးပြုနိုင်တဲ့ လုပ်ဆောင်ချက်များပဲ ဖြစ်ပါတယ်။

Function ဆိုတာလည်း Procedure တစ်မျိုးပါပဲ။ Procedure တွေက ကြိုတင်သတ်မှတ်ထားတဲ့ လုပ်ဆောင်ချက်တွေသာဖြစ်ပါတယ်။ Function တွေရဲ့ ထူးခြားချက်ကတော့ ပေးလာတဲ့ တန်ဖိုးကို လက်ခံပြီး၊ အလုပ်လုပ်လို့ရလာတဲ့ ရလဒ်ကို ပြန်ပေးခြင်းပဲ ဖြစ်ပါတယ်။

Pseudocode

```
function add {
    let a = 1;
    let b = 2;
    print a + b;
}
```

ဒီတစ်ခါ Procedure မဟုတ်တော့ပါဘူး။ Function ဖြစ်သွားပါပြီ။ ဒါပေမယ့်၊ Function ရဲ့သဘောသဘာဝ အမှန်ဖြစ်တဲ့ ပေးတဲ့တန်ဖိုးကို လက်ခံတယ်ဆိုတဲ့ သဘောသဘာဝ မပါသေးပါဘူး။ ဒါကြောင့် အခုလို ပြင်လိုက်ပါမယ်။

Pseudocode

```
function add (x, y) {
    let a = x;
    let b = y;
    print a + b;
}
```

ဒီတစ်ခါမှာ `add` ဆိုတဲ့ အမည်နောက်ကနေ ဝိုက်ကွင်းအဖွင့်အပိတ်နဲ့ လက်ခံလိုတဲ့ တန်ဖိုးနှစ်ခု သတ်မှတ်ထားပါတယ်။ x နဲ့ y ဖြစ်ပါတယ်။ ဒီသဘောသဘာဝကို Function Parameter လို့ခေါ်ပါတယ်။ `add` Function မှာ x နဲ့ y ဆိုတဲ့ Parameter နှစ်ခုရှိသွားတဲ့ သဘောပါ။ ဒါကြောင့် ဒီ Function က x, y ဆိုတဲ့ တန်ဖိုးနှစ်ခုကို လက်ခံအလုပ်လုပ်နိုင်သွားပါပြီ။ ခေါ်ယူအသုံးပြုတဲ့အခါ ဒီလိုသုံးနိုင်ပါတယ်။

Pseudocode

```
call add(2, 3);     // 5
```

စောစောက Procedure လိုကြိုတင်သတ်မှတ်ထားတဲ့ တန်ဖိုးတွေနဲ့ အလုပ်လုပ်တာ မဟုတ်တော့ဘဲ ပေးလိုက်တဲ့ x တန်ဖိုး 2 နဲ့ y တန်ဖိုး 3 တို့ကိုအခြေခံပြီး အလုပ်လုပ်တာ ဖြစ်သွားပါပြီ။ ဒီလို Function ကို ခေါ်ယူစဉ်မှာ ပေးလိုက်တဲ့ တန်ဖိုးတွေကိုတော့ Arguments လို့ ခေါ်ပါတယ်။ `add` ကို ခေါ်တဲ့အခါ Arguments အနေနဲ့ 2 နဲ့ 3 တို့ကို ပေးလိုက်တဲ့ဆိုတဲ့ အဓိပ္ပါယ်ပဲ ဖြစ်ပါတယ်။

တစ်ကယ်တော့ ခေါ်သုံးတယ်ဆိုတာ ပေါ်လွင်အောင် `call` Keyword နဲ့ နမူနာပြတာပါ။ လက်တွေ့မှာ ထည့်စရာလိုအပ်လေ့ မရှိပါဘူး။ ဒါကြောင့် နမူနာမှာ အသုံးပြုပုံနည်းနည်း ပြောင်းလိုက်ပါမယ်။ ဒီလိုပါ –

Pseudocode

```
add(2, 3);     // 5
```

နောက်ထပ် သတိပြုစရာ ကျန်ပါသေးတယ်။ ရလဒ်အနေနဲ့ 5 ကို တွေ့မြင်ရတယ်ဆိုတာ Function ထဲမှာ `print a + b` ဆိုတဲ့ Statement ရှိနေလို့ ပေါင်းခြင်းရလဒ်ကို ထုတ်ဖော် ပြသတာပါ။ Function တစ်ခုရဲ့ သဘောသဘာဝဖြစ်တဲ့ ခေါ်ယူတဲ့အခါ ရလဒ် ပြန်ပေးတယ်ဆိုတဲ့သဘော မဟုတ်သေးပါဘူး။ ဒါကြောင့် အခုလို ထပ်ပြီးတော့ ပြင်လိုက်ပါဦးမယ်။

Pseudocode

```
function add (x, y) {
    let a = x;
    let b = y;
    return a + b;
}
```

ဒီတစ်ခါအသုံးပြုတဲ့ Keyword ကို ဂရုပြုပါ။ `return` ဖြစ်သွားပါပြီ။ အဓိပ္ပါယ်ရ a + b ရလဒ်ကို ပြန်ပေးမယ်ဆိုတဲ့ အဓိပ္ပါယ်ဖြစ်သွားတာပါ။ a + b ရလဒ်ကို ထုတ်ဖော်ပြသတာ မဟုတ်တော့ပါဘူး။ ဒါကြောင့် ခေါ်ယူအသုံးပြုပုံပြုနည်းလည်း နည်းနည်း ပြောင်းသွားပါလိမ့်မယ်။ ဒီလိုပါ –

Pseudocode

```
let result = add(2, 3);
result + 2;                // 7
```

`add` Function ကို ခေါ်ယူလိုက်တဲ့အခါ Function က ပေါင်းခြင်းရလဒ်ကို ပြန်ပေးထားလို့၊ ပြန်ရလာတဲ့ ရလဒ်ကို `result` Variable ထဲမှာ ထည့်ထားလိုက်တာပါ။ ဒီနည်းရဲ့ အားသာချက်ကတော့ ဒီ Result ကို ဖော်ပြယုံတင် မဟုတ်တော့ဘဲ ဆက်လက်ပြီး လိုအပ်သလို အသုံးချသွားနိုင်ခြင်းပဲ ဖြစ်ပါတယ်။

နောက်ဆုံးတစ်ခုအနေနဲ့ Function သတ်မှတ်စဉ်မှာ ထည့်သွင်းခဲ့တဲ့ Parameter တွေဟာ Function ရဲ့ Variable တွေပဲမို့လို့ တိုက်ရိုက်အသုံးပြုနိုင်ပါတယ်။ ဒါကြောင့် x နဲ့ y ကို a နဲ့ b ထဲ တစ်ဆင့်ခံ ထည့်မနေတော့ပါဘူး။ စလက်ခံကတည်းက a နဲ့ b အနေနဲ့ တိုက်ရိုက်လက်ခံလိုက်လို့ ရပါတယ်။ ဒါကြောင့် အခုလို ပြင်လိုက်ပါဦးမယ်။

Pseudocode

```
function add (a, b) {
    return a + b;
}
```

ဒါဆိုရင်တော့ မလိုအပ်တော့တဲ့ အဆင့်တွေလျော့သွားလို့ ပိုရှင်းသွားပါပြီ။ ပေးလာတဲ့ Argument နှစ်ခုကို a နဲ့ b အဖြစ် လက်ခံပြီးပေါင်းပြီး ရလဒ်ကိုပြန်ပေးတဲ့ Function တစ်ခုကို ရသွားခြင်းပဲ ဖြစ်ပါတယ်။

ဂဏန်းလေး (၂) ခုပေါင်းတာလောက်ကတော့ Function တွေ Procedure တွေ မလိုအပ်သေးပါဘူး။ ဒီအတိုင်းတိုက်ရိုက်ပေါင်းလည်း ရတာပါပဲ။ ဒါပေမယ့် ရှုပ်ထွေးတဲ့လုပ်ဆောင်ချက်တွေကို အခုလို Function တွေ Procedure တွေအနေနဲ့ စုစည်းရေးဖွဲ့ထားမယ်ဆိုရင် ထပ်ခါထပ်ခါပြန်ရေးစရာမလိုတဲ့ အားသာချက်ကို ရမှာပဲ ဖြစ်ပါတယ်။

JavaScript အပါအဝင် Programming Language အများစုမှာ Function ရေးသားနည်းတစ်နည်း၊ Procedure ရေးသားနည်း တစ်နည်းဆိုပြီး ခွဲထားကြလေ့ မရှိပါဘူး။ တစ်နည်းထဲပဲ ထားကြပါတယ်။ နှစ်မျိုးခွဲပြီးတော့လည်း ခေါ်မနေကြတော့ပါဘူး။ နှစ်မျိုးလုံးကိုခြုံပြီး Function လို့ပဲ ခေါ်ကြပါတယ်။

JavaScript Functions

အထက်မှာ နမူနာရေးပြခဲ့တဲ့ Pseudocode Function တွေထဲက နောက်ပိုင်းနမူနာတွေဟာ JavaScript Function ရေးထုံးနဲ့ ကိုက်ညီနေပြီး ဖြစ်ပါတယ်။ Function တစ်ခုကြေညာဖို့အတွက် `function` Keyword ကိုအသုံးပြု ကြေညာရပါတယ်။ Function အမည်ကို မိမိနှစ်သက်ရာအမည်ပေးနိုင်ပြီး အမည်ရဲ့ နောက်မှာ ဝိုက်ကွင်းအဖွင့်အပိတ်နဲ့ Parameter တွေ လိုက်ရပါတယ်။ Parameter တွေမပါဘဲ Function ကြေညာလိုရင် ဝိုက်ကွင်းအဖွင့်အပိတ်ကို အလွတ်အတိုင်း ထည့်ပေးရပါတယ်။ မထည့်လို့မရပါဘူး။ Function ရဲ့ Statement တွေကိုတော့ တွန့်ကွင်း အဖွင့်အပိတ်နဲ့ စုဖွဲ့ပြီး ရေးသားပေးရမှာပဲ ဖြစ်ပါတယ်။

အကယ်၍ Function ကနေ ရလဒ်ကို ပြန်ပေးလိုရင် `return` Keyword ကို အသုံးပြုနိုင်ပါတယ်။ Return ပြန်မပေးဘဲရေးချင်ရင်လည်း ရပါတယ်။ နမူနာတစ်ချို့ ဖော်ပြပေးလိုက်ပါတယ်။

JavaScript

```
function add(a, b) {
    return a + b
}

function circle(r) {
    const PI = 3.14
    return PI * r * r
}

function greet() {
    console.log("Hello, World!")
}

function sayHello(name) {
    console.log(`Hello ${name}`)
}
```

ရေးသားထားတဲ့ Function တွေကို ခေါ်ယူအသုံးပြုတဲ့အခါ Function အမည်နဲ့အတူ Arguments စာရင်းကို ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာ ထည့်ပြီး ခေါ်ယူနိုင်ပါတယ်။ Argument မရှိရင်လည်း ဝိုက်ကွင်းအဖွင့်အပိတ် အလွတ်ကို ထည့်ပေးရပါတယ်။

JavaScript

```
add(7, 8)           // 15
circle(3)           // 28.26
greet()             // Hello, World
sayHello("Alice")   // Hello Alice
```

Function ရဲ့အမည်ပေးပုံပေးနည်းဟာ Variable ပေးပုံပေးနည်းနဲ့ အတူတူပါပဲ။ စာလုံးအကြီးအသေး နှစ်သက်ရာ အသုံးပြုနိုင်ပြီး ကိန်းဂဏန်းတွေလည်း ပါလို့ရပါတယ်။ ဒါပေမယ့် ကိန်းဂဏန်းနဲ့ စလို့မရသလို၊ Special Character တွေ Space တွေလည်း ပါလို့မရပါဘူး။ လိုအပ်ရင် Underscore ကိုအသုံးပြုနိုင်ပါတယ်။ Special Character တွေထဲက ခြွင်းချက်အနေနဲ့ $ သင်္ကေတကိုတော့ လိုအပ်ရင် ထည့်သွင်းအသုံးပြုနိုင်ပါတယ်။

`return` ကိုတော့ တစ်ချက်သတိပြုပါ။ Function တစ်ခုဟာ `return` နဲ့ရလဒ်ကို ပြန်ပေးပြီးရင် သူ့အလုပ် ပြီးသွားပါပြီ။ ဒါကြောင့် `return` ရဲ့အောက်မှာ ဆက်ရေးထားတဲ့ ကုဒ်တွေ ရှိနေရင် အလုပ်လုပ်မှာ မဟုတ်တော့ပါဘူး။

JavaScript

```
function add(a, b) {
    console.log('Start adding')
    return a + b
    console.log('Done adding')
}

let result = add(1, 2)
```

နမူနာအရ Start adding ဆိုတဲ့စာကို ဖော်ပြမှာဖြစ်ပြီး a နဲ့ b ပေါင်းခြင်းကို ပြန်ပေးလိုက်မှာပါ။ ဒါပေမယ့် Done adding ဆိုတဲ့စာကို ဖော်ပြဖို့ညွှန်ကြားထားတဲ့ Statement ကတော့ အလုပ်လုပ်မှာ မဟုတ်တော့ပါဘူး။ `return` နဲ့ တန်ဖိုးပြန်ပေးပြီးပြီမို့လို့ Function အလုပ်ပြီးဆုံးသွားခဲ့ပြီမို့လို့ပါ။

Default & Rest Parameters

Function တွေကို ခေါ်ယူအသုံးပြုတဲ့အခါ၊ ပုံမှန်အားဖြင့်၊ ကြေညာစဉ်က သတ်မှတ်ခဲ့တဲ့ Parameter အရေအတွက်အတိုင်းပဲ Argument အရေအတွက်ကို အတိအကျပေးရပါတယ်။ ပိုပေးခဲ့ရင် ပိုသွားတာတွေကို ထည့်သွင်းအလုပ်လုပ်မှာ မဟုတ်သလို၊ လိုသွားခဲ့ရင်တော့ မလိုလားအပ်တဲ့ အမှားတွေကို ကြုံတွေ့ရပါလိမ့်မယ်။

```
function add(a, b) {
    return a + b
}

add(1, 2, 3, 4)      // 3
add(1)               // NaN
```

နမူနာအရ `add` Function ကိုခေါ်ယူတဲ့အခါ Argument နှစ်ခုပေးရမှာပါ။ ပိုပေးလိုက်တဲ့အခါ ကျန်တဲ့ Argument တွေကို ထည့်သွင်းအလုပ်မလုပ်တာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ လိုသွားတဲ့အခါမှာတော့ a နဲ့ b

နှစ်ခုမှာ b အတွက် တန်ဖိုးမရှိတော့လို့ `undefined` ဖြစ်သွားပါတယ်။ `undefined` နဲ့ ကိန်းဂဏန်းကို ပေါင်းဖို့ကြိုးစားလိုက်တဲ့အခါ `NaN` ကို ပြန်ရနေလို့ မှားနေပါပြီ။

တစ်ချို့ Programming Language တွေမှာဆိုရင် Argument အရေအတွက် မပြည့်ဘဲ လိုသွားရင် အလုပ်မလုပ်ဘဲ Error ဖြစ်ပါတယ်။ JavaScript မှာ Error တော့မဖြစ်ပါဘူး။ ဒါပေမယ့် သတ်မှတ်ထားတဲ့ Argument ပြည့်စုံမှပဲ ရလဒ်အမှန်ကို ရမှာပါ။ ဒီနေရာမှာလိုအပ်ရင် Default Parameter Value ကို သတ်မှတ်ပေးထားနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
function add(a, b = 0) {
    return a + b
}

add(1, 2)       // 3
add(1)          // 1
```

Parameter သတ်မှတ်စဉ်မှာ `b = 0` လို့ပြောထားတဲ့အတွက် b မှာ Default Value ရှိသွားပါပြီ။ ဒါကြောင့် b အတွက် Argument မပါလာခဲ့ရင် ဒီ Default Value ကို အသုံးပြုပြီးတော့ အလုပ်လုပ်ပေးသွားမှာပါ။ ပါလာခဲ့ရင်တော့ ပါလာတဲ့ Argument တန်ဖိုးကို အသုံးပြုပေးသွားမှာ ဖြစ်ပါတယ်။

JavaScript မှာ Rest Parameter ဆိုတာရှိပါသေးတယ်။ ပါလာသမျှ Argument တန်ဖိုးအားလုံးကို လက်ခံပေးနိုင်တဲ့ Parameter ပါ။ ဒီလိုရေးရပါတယ်။

JavaScript

```
function add(a, b, ...c) {
    console.log(c)
}

add(1, 2, 3, 4, 5)     // [3, 4, 5]
```

နမူနာအရ a, b နဲ့ c Parameter သုံးခုရှိပြီး `c` ကို `...c` ဆိုတဲ့ Rest Parameter ရေးထုံးနဲ့ ရေးသားထားပါတယ်။ ဒါကြောင့် ပိုတဲ့ Argument အားလုံးကို `c` က လက်ခံထားပေးမှာပါ။ နမူနာမှာ `add` Function ကို

ခေါ်တဲ့အခါ Argument (၅) ခုပေးထားပါတယ်။ ပထမ Argument က a ဖြစ်သွားပြီး ဒုတိယ Argument က b ဖြစ်သွားပါတယ်။ ကျန် Argument (၃) ခုကတော့ c ထဲကို အကုန်ရောက်သွားတယ် ဆိုတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

Function Expressions

Function တစ်ခု ကြေညာလိုက်တယ်ဆိုတာဟာ Statement တစ်ခု တည်ဆောက်လိုက်တာပါပဲ။ ထူးခြားချက်ကတော့ JavaScript မှာ Function တွေကို Expression လို့လဲ သဘောထား အသုံးပြုနိုင်ပါတယ်။ ဥပမာ – 1 + 2 ဆိုတဲ့ Expression ကို Variable တစ်ခုထဲမှာ Assign လုပ်လိုက်လို့ ရသလိုပဲ Function ကိုလည်း Variable တစ်ခုထဲမှာ Assign လုပ်လိုက်လို့ ရနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
let greet = function greeting() {
    console.log("Hello, World")
}
```

ဒါကြောင့် `greet` Variable ထဲမှာ Function တစ်ခုရောက်ရှိသွားပါပြီ။ အဲ့ဒီ `greet` Variable ထဲက Function ကို အလုပ်လုပ်စေချင်ရင် အခုလို ခေါ်ယူပေးလို့ ရပါတယ်။

JavaScript

```
greet()     // Hello, World
```

ရိုးရိုး Function Statement ကိုအသုံးပြုကြေညာထားတဲ့ Function ကို ခေါ်သလိုပဲ ခေါ်ရတာပါ။ ဒီလို Function ကို Expression အနေနဲ့ အသုံးပြုတဲ့အခါ Function အမည်ကို မလိုအပ်ရင် မထည့်လို့ရပါတယ်။ ဒီသဘောသဘာဝကို Anonymous Function လို့ ခေါ်ပါတယ်။ အမည်မဲ့ Function မို့လို့ Nameless Function လို့လည်း ခေါ်ကြပါသေးတယ်။ ဒီလိုပါ –

JavaScript

```
let greet = function () {
    console.log("Hello, World")
}
```

စောစောက ကုဒ်ပါပဲ။ `function` Keyword ရဲ့နောက်မှာ Function Name မပါတော့တာပါ။ ဒီလို Function Expression တွေကို အသုံးပြုပြီးတော့ ရေးလို့ရတဲ့ IIFE ခေါ် Immediately Invoked Function Expression ရေးနည်းတစ်မျိုးလည်း ရှိပါသေးတယ်။ ဒီလိုပါ –

```
(function () {
    console.log("Hello, World")
})()

// Hello, World
```

a + b Expression ကို လိုအပ်ရင် ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာ (a + b) လို့ ထည့်ရေးနိုင်သလိုပဲ Function Expression ကိုလည်း ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာ ထည့်လိုက်တာပါ။ ပြီးတော့မှ နောက်ကနေ နောက်ထပ် ဝိုက်ကွင်းအဖွင့်အပိတ် တွဲပေးလိုက်တဲ့အခါ ဒီ Function Expression ကို ချက်ချင်းခေါ်ယူ အလုပ်လုပ်စေလိုက်တဲ့ သဘောဖြစ်သွားပါတော့တယ်။

ပုံမှန်အားဖြင့် Function ဆိုတာ ကြေညာလိုက်ယုံနဲ့ အလုပ်မလုပ်ပါဘူး။ ခေါ်ယူအသုံးပြုတော့မှသာ အလုပ်လုပ်တာပါ IIFE ရေးထုံးကို အသုံးပြုရေးသားထားတဲ့ Function ကတော့ ချက်ချင်းနေရာတင် အလုပ်လုပ်သွားတဲ့ Function တစ်ခုဖြစ်သွားတာပါ။

Function Expression တွေကို နောက်ထပ်အသုံးများတဲ့ နည်းတစ်ခုရှိပါသေးတယ်။ ဒီလိုပါ –

JavaScript

```
function twice(num, fun) {
    let result = fun(num)
    return result * 2
}
```

နမူနာ `twice` Function ကို သေချာဂရုစိုက်ကြည့်ပါ။ Parameter (၂) ခုပါဝင်ပါတယ်။ `num` နဲ့ `fun` တို့ပါ။ `num` ဟာ Number ဖြစ်ရမှာဖြစ်ပြီး၊ `fun` ကတော့ Function ဖြစ်ရမှာပါ။ ဒါကြောင့် `twice` ကိုခေါ်ယူတဲ့အခါ ပေးရမယ့် တန်ဖိုးအမျိုးအစားတွေက ဒီလိုဖြစ်ပါလိမ့်မယ်။

```
twice(Number, Function)
```

Number တွေအတွက် နှစ်သက်ရာ Number တန်ဖိုးပေးနိုင်ပြီး Function အတွက် Function Expression တစ်ခုကို ပေးနိုင်ပါတယ်။ ဒါကြောင့် အခုလို ခေါ်ယူအသုံးနိုင်ပါတယ်။

JavaScript

```
twice(5, function(x) {
    return x + 1
})
```

`num` အတွက် 5 ကိုပေးထားပါတယ်။ `fun` အတွက် x ကို 1 တိုးပေးတဲ့ Function ကိုပေးထားပါတယ်။ မူလ `twice` Function နဲ့ ပြန်တွဲပြီးကြည့်ပါ။

```
function twice(num, fun) {
    let result = fun(num)
    return result * 2
}
```

လက်ရှိ `num = 5` ဖြစ်ပါတယ်။ ဒါကြောင့် `fun(5)` ပါ။ `fun` ရဲ့ `x = 5` ဖြစ်သွားပါပြီ။ 1 တိုးပြီး ပြန်ပေးတဲ့အတွက် 6 ကိုရပါတယ်။ `result = 6` ဖြစ်သွားပါပြီ။ ပြီးတော့မှ `result` ကို 2 နဲ့ မြှောက်ပေးလိုက်တာ ဖြစ်လို့ နောက်ဆုံးရလဒ် `6 * 2 = 12` ကို ရမှာပဲ ဖြစ်ပါတယ်။

ဒီနည်းနဲ့ Function ကို ခေါ်ယူအသုံးပြုစဉ်မှာ Function Expression ကို Argument အနေနဲ့ ထည့်သွင်းပေးနိုင်ခြင်းဖြစ်ပါတယ်။ ဒီသဘောသဘာဝကို Callback လို့ခေါ်ပါတယ်။ Function တစ်ခုကို ခေါ်ယူစဉ်မှာ ဒီ Function ကို ထပ်ဆင့် ပြန်ခေါ်သုံးပေးပါလို့ ပြောလို့ရသွားတာပါ။ အစီအစဉ်အတိုင်း၊ အစဉ်အလိုက် အလုပ်လုပ်တာ မဟုတ်တော့လို့ မျက်စိတော့ လည်သွားနိုင်ပါတယ်။ နောက်တစ်ခေါက်လောက် ပြန်ကြည့်ကြည့်လိုက်ပါ။ တစ်ခါလောက် သေချာမြင်သွားပြီဆိုရင်တော့ နောင်ဒီသဘောသဘာဝကို တွေ့တိုင်း အလွယ်တကူ မြင်သွားပါလိမ့်မယ်။

ဒီနည်းကပေးတဲ့ အားသာချက်ကတော့ JavaScript Function တွေဟာ ကြိုရေးထားတဲ့အတိုင်း ပုံသေ မဟုတ်တော့ဘဲ ခေါ်ယူချိန်မှာ လိုအပ်သလို ပြောင်းလဲအလုပ်လုပ်စေနိုင်တဲ့ Function တွေ ဖြစ်သွားတော့ တာပါပဲ။ ဥပမာ – စောစောက `twice` Function ကို အခုလိုလည်း ခေါ်ယူနိုင်မှာ ဖြစ်ပါတယ်။

JavaScript

```
twice(5, function(x) {
    return x * x
});

// 50
```

ပေးလိုက်တဲ့ `num` တန်ဖိုးမပြောင်းပါဘူး။ 5 ပါပဲ။ ဒါပေမယ့် ရလဒ်တော့ လုံးဝပြောင်းသွားပါပြီ။ ဘာဖြစ်လို့လဲဆိုတော့ ခေါ်သုံးဖို့ပေးလိုက်တဲ့ Callback Function ရဲ့အလုပ်လုပ်ပုံ ပြောင်းသွားတဲ့ အတွက်ကြောင့်ပါ။

တစ်ကယ်တော့ JavaScript Function တွေရဲ့ သဘောသဘာဝဟာ ကျယ်ပြန့်လှပါတယ်။ Function တစ်ခုရဲ့ အတွင်းမှာ ထပ်ဆင့်ရှိနေတဲ့ Nested Function တွေရဲ့ သဘောသဘာဝတွေ၊ Closure လို့ခေါ်တဲ့ ထူးခြားတဲ့ Variable Scope သဘောသဘာဝတွေ၊ Recursive Function သဘောသဘာဝတွေ၊ `this` Keyword Binding လို့ခေါ်တဲ့ ရှုပ်ထွေးတဲ့ ကိစ္စတွေ ကျန်ပါသေးတယ်။ ဒါပေမယ့် ဒီအဆင့်မှာ အဲ့ဒီ အကြောင်းတွေပြောဖို့တော့ စောနေပါသေးတယ်။ ဒါကြောင့် လက်ရှိပြောထားသလောက်နဲ့ ဆက်ပြောမယ့် သဘောသဘာဝတွေကို အရင်ကျေညက်နားလည်အောင် ကြည့်ထားပြီး နောက်ပိုင်းမှာ ဒီထက်ပို ဆန်းကျယ်တာတွေ ဆက်လေ့လာဖို့ ကျန်နေသေးတယ်လို့ မှတ်သားထားပေးပါ။

Arrow Functions

ဆက်လက်ပြီး Function Expression တွေကို အတိုကောက်ရေးနည်းရှိလို့ ဆက်ကြည့်ကြပါမယ်။ Function Keyword အစား Arrow သင်္ကေတလေးနဲ့ အစားထိုးပြီး ရေးရလို့ Arrow Function လို့ ခေါ်ကြပါတယ်။ ပုံမှန် Function Expression တစ်ခုကို Variable တစ်ခုမှာ Assign လုပ်လိုတဲ့အခါ ဒီလိုရေးရပါတယ်။

```JavaScript
let add = function(a, b) {
    return a + b
}
```

ဒါကို အတိုကောက်အားဖြင့် အခုလို ရေးနိုင်ပါတယ်။

```JavaScript
let add = (a, b) => {
    return a + b
}
```

ဝိုက်ကွင်းအဖွင့်အပိတ်ရှေ့မှာ `function` မပါတော့ဘဲ ဝိုက်ကွင်းအဖွင့်အပိတ်နောက်မှာ Arrow => လေး ထည့်လိုက်တာပါ။ Arrow Function ရဲ့ ထူးခြားချက်ကတော့၊ Statement တစ်ခုထဲဆိုရင် တွန့်ကွင်းအဖွင့်အပိတ်နဲ့ `return` Keyword ကို မထည့်ဘဲ ထားလို့ရခြင်းပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် အခုလို ဖြစ်သွားမှာပါ။

```JavaScript
let add = (a, b) => a + b
```

တော်တော်လေးကို တိုတောင်းကျစ်လစ်တဲ့ ရေးဟန်တစ်ခုဖြစ်သွားတာမို့လို့ အလွန်ကြိုက်ကြပါတယ်။ နောက်တစ်ချက်အနေနဲ့ Parameter တစ်ခုထဲဆိုရင် ဝိုက်ကွင်းအဖွင့်အပိတ်ကို မထည့်ဘဲရေးလို့ရပါတယ်။ ပိုတိုသွားဦးမှာပါ။ ဒီလိုပါ –

```JavaScript
let two = n => n * 2
```

ဟိုးရှေ့မှာပြောခဲ့သလို ကုဒ်တွေရဲ့ အလုပ်လုပ်ပုံကို ခေါင်းထဲမှာ ပုံဖော်ကြည့်နိုင်ဖို့ အရေးကြီးလှပါတယ်။ တိုလွန်းလို့ ပုံဖော်ကြည့်ရ ခက်သွားမှာစိုးလို့ ရိုးရိုး Function နဲ့ နှိုင်းယှဉ်စဉ်းစားနိုင်ဖို့ ဖော်ပြခဲ့တာပါ။ လိုရင်းအချုပ် (၃) ချက်မှတ်ရင် ရပါပြီ။

1. ဝိုက်ကွင်းအဖွင့်အပိတ်ရှေ့က `function` ကိုဖယ်ပြီး ဝိုက်ကွင်းအဖွင့်အပိတ်နောက်မှာ `=>` သင်္ကေတ လေး ထည့်ပေးလိုက်ပါတယ်။
2. Function မှာ Statement တစ်ကြောင်းဘဲရှိရင် တွန့်ကွင်းအဖွင့်အပိတ်နဲ့ `return` ကို မထည့်ဘဲ ရေးလို့ရပါတယ်။ အလိုအလျောက် `return` ပြန်ပေးပါတယ်။
3. Parameter က တစ်ခုထဲဆိုရင် ဝိုက်ကွင်းအဖွင့်အပိတ်ကို မထည့်ဘဲထားလို့ ရပါတယ်။

Parameter မရှိရင်တော့ ဝိုက်ကွင်းအဖွင့်အပိတ် အလွတ်ကို ထည့်ပေးဖို့ လိုအပ်ပါတယ်။

JavaScript

```
let hello = () => "Hello, World"
```

ဒါကြောင့် တစ်ချို့ကလည်း ဝိုက်ကွင်းအဖွင့်အပိတ်အလွတ်ထည့်ရတာ ရှုပ်တယ် ထင်ကြပုံ ရပါတယ်။ Parameter မရှိရင် Underscore ကို Parameter တစ်ခုအနေနဲ့ ပေးပြီးရေးကြပါတယ်။ ဒါကြောင့် ရံဖန်ရံခါ ဒီလိုကုဒ်မျိုးကို တွေ့ရနိုင်ပါတယ်။

JavaScript

```
let hello = _ => "Hello, World"
```

Parameter မရှိရင် ဝိုက်ကွင်းအဖွင့်အပိတ် ထည့်မယ့်အစား Underscore ကို Parameter တစ်ခုသဖွယ် အစားထိုးပြီး ရေးလိုက်တဲ့သဘောပါ။ ဟိုးအပေါ်မှာ Callback အတွက် နမူနာပြခဲ့တဲ့ Function ကုဒ်ကို Arrow Function နဲ့ ပြောင်းရေးကြည့်ပါမယ်။ ဒီလိုပါ –

JavaScript

```
// Original Function
function twice(num, fun) {
    let result = fun(num)
    return result * 2
}

// Arrow Function
let twice = (n, f) => f(n) * 2
```

တော်တော်လေးကို တိုတောင်းကျစ်လစ်သွားတာကို တွေ့ရနိုင်ပါတယ်။ ဒီသဘောတွေကြောင့်ပဲ အခုနောက်ပိုင်းမှာ Arrow Function တွေကို တော်တော်ကြိုက်ကြသလို နေရာအနှံ့အပြားမှာလည်း သုံးကြပါတယ်။ ဒါကြောင့် အတိုကောက်ရေးထားတဲ့ Arrow Function ကုဒ်တွေကိုမြင်ရင် နားလည်ဖို့ လိုအပ်ပါတယ်။ ရိုးရိုး Function နဲ့ အသားကျပြီးသား အတွေ့အကြုံ အထိုက်အလျောက် ရှိသူတွေဟာ Arrow Function တွေကို မြင်တဲ့အခါ မူလအသားကျပြီးသားနဲ့ ကွဲလွဲနေလို့ နားလည်ရခက်နေတတ်ကြပါတယ်။

Function Hoisting

ဟိုးရှေ့မှာ Variable Hoisting အကြောင်းလေး ထည့်ပြောခဲ့ပါတယ်။ Variable တွေကို `var` Keyword နဲ့ ကြေညာလိုက်တဲ့အခါ JavaScript က အပေါ်ကိုပို့ပေးပြီး အလုပ်လုပ်တဲ့ သဘောမျိုးပါ။ ဒီလိုပါ –

JavaScript

```
var r = a + b
var a = 1
var b = 2
```

အရင်သုံးပြီး နောက်မှကြေညာလိုက်တာပါ။ ဒီလိုအခြေအနေမျိုးမှာ JavaScript က ကုဒ်ကို အခုလို ပုံစံပြောင်းပြီး အလုပ်လုပ်ပေးသွားမှာပါ။

Pseudocode

```
var a
var b
var r = a + b
a = 1
b = 2
```

ရလဒ်ကတော့ မှန်မှာမဟုတ်ပါဘူး။ ဒါပေမယ့် Hoisting သို့မဟုတ် Lifting လို့ခေါ်တဲ့ သဘောသဘာဝအရ ကြေညာချက်ကို အလိုအလျောက် အပေါ်ကိုပို့ပေးလိုက်တာကို သိထားဖို့ပါ။ `var` Keyword ကို သုံးမှပဲ ဒီသဘောမျိုးနဲ့ အလုပ်လုပ်ပေးပါတယ်။ `let` တို့ `const` တို့နဲ့ ကြေညာရင်တော့ ရေးထားတဲ့ အစီအစဉ်အတိုင်းသာ အလုပ်လုပ်သွားမှာပါ။

Function တွေမှာလည်း Hoisting သဘောသဘာဝ ရှိပါတယ်။ ဒါကြောင့် Function ကို အရင်သုံးပြီး နောက်မှ ကြေညာလို့ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
add(1, 2)     // 3

function add(a, b) {
    return a + b
}
```

ဒီကုဒ်ဟာ အလုပ်လုပ်ပါတယ်။ `add` Function ကို မကြေညာရသေးခင်ကတည်းက ယူသုံးထားပေမယ့် JavaScript က အလုပ်လုပ်တဲ့အခါ `add` Function ကြေညာချက်ကို အပေါ်တင်ပေးလိုက်ပြီး အလုပ်လုပ်သွားမှာ မို့လို့ပါ။ ဒါပေမယ့် ဒီလိုဆိုရင်တော့ ရမှာ မဟုတ်ပါဘူး –

JavaScript

```
add(1, 2)     // Error: add is not defined

let add = function(a, b) {
    return a + b
}
```

Function Expression နဲ့ရေးသားပြီး `add` Variable ထဲမှာ Assign လုပ်လိုက်တာပါ။ အလုပ်မလုပ်ပါဘူး။ ဘာဖြစ်လို့လဲဆိုတော့ `let` နဲ့ ကြေညာထားတဲ့ `add` ကို JavaScript က Hoist/Lift လုပ်ပြီး တင်ပေးမှာ မဟုတ်တဲ့အတွက် `add` Function ကို ခေါ်သုံးစဉ်မှာ Function မရှိသေးလို့ Error တက်သွားမှာပဲ ဖြစ်ပါတယ်။

Name Conflict

Function Name တွေဟာ အမည်တူလို့မရဘူး ဆိုတဲ့အချက်ကိုလဲ ဖြည့်စွက်မှတ်သားသင့်ပါတယ်။ ရှိပြီးသား Function ကို နောက်တစ်ခါ ထပ်ကြေညာလို့ မရဘူးလို့ ပြောတာပါ။ တစ်ချို့ Language တွေမှာ အမည်တူ ကြေညာမိရင် တစ်ခါထဲ Error ဖြစ်ပါတယ်။ JavaScript မှာ အမည်တူ ကြေညာမိလို့ Error တော့မဖြစ်ပါဘူး။ ဒါပေမယ့် နောက်မှရေးတဲ့ Function ကို အတည်ယူသွားမှာ ဖြစ်လို့ အမည်တူတဲ့ အရင်ရေးထားတဲ့ Function တွေက အလုပ်လုပ်တော့မှာ မဟုတ်ပါဘူး။

Variable Scope

Variable တွေအကြောင်းပြောတုံးက Block Scope Variable အကြောင်းပြောခဲ့ပါတယ်။ Function Scope အကြောင်းလေး ဖြည့်စွက်ပြောဖို့လိုပါသေးတယ်။ အခြေခံအားဖြင့် Function တစ်ခုအတွင်းထဲမှာ ကြေညာထားတဲ့ Variable ကို Function ရဲ့ပြင်ပနဲ့ အခြား Function များက ရယူအသုံးပြုခွင့်မရှိပါဘူး။ Function ထဲမှာ ကြေညာထားတဲ့ Variable ဟာ အဲ့ဒီ Function နဲ့သာ သက်ဆိုင်ပါတယ်။ Function ရဲ့ပြင်ပမှာ ကြေညာထားတဲ့ Variable တွေကိုတော့ Global Variable လို့ခေါ်ပါတယ်။ လိုအပ်တဲ့နေရာကနေ ရယူအသုံးပြုနိုင်ပါတယ်။

JavaScript

```
let name = "Alice"

function welcome() {
    console.log(`Welcome ${name}`)
}

function hello() {
    console.log(`Hello ${name}`)
}

welcome()       // Welcome Alice
hello()         // Hello Alice
```

နမူနာမှာကြည့်လိုက်ရင် Function တွေရဲ့ပြင်ပမှာ ကြေညာထားတဲ့ `name` ကို Function တွေထဲမှာ အသုံးပြုထားတာကို တွေ့ရနိုင်ပါတယ်။ ဒီလို Global Variable တွေဟာ အန္တရာယ်တော့ များပါတယ်။ Function တိုင်းက သုံးလို့ရနေတော့ Function တစ်ခုကြောင့် တန်ဖိုးပြောင်းသွားတဲ့အခါ သူ့ကိုသုံးထားတဲ့ အခြား Function မှာ မလိုလားအပ်တဲ့ ပြဿနာတွေ ဖြစ်စေနိုင်ပါတယ်။

JavaScript

```
function welcome() {
    let name = "Alice"
    console.log(`Welcome ${name}`)
}

function hello() {
    console.log(`Hello ${name}`)
}

welcome()       // Welcome Alice
hello()         // Hello
```

ဒီနမူနာမှာတော့ `welcome` Function ထဲမှာ `name` ကို ကြေညာထားလို့ ရလဒ်မှန်ကန်တာကို တွေ့ရနိုင်ပါတယ်။ `hello` Function ကလည်း `name` ကို အသုံးပြုထားပါတယ်။ ဒါပေမယ့် welcome ထဲမှာ ကြေညာထားတဲ့ `name` ဟာ `hello` နဲ့ သက်ဆိုင်ခြင်းမရှိတဲ့အတွက် အလုပ်မလုပ်တာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

အခန်း (၅) – Arrays & Objects

Programming Language တွေမှာပါတဲ့ Object ဆိုတဲ့သဘောသဘာဝကို နားလည်စေဖို့အတွက် ပြင်ပက ရုပ်ဝတ္ထုတွေနဲ့ နှိုင်းယှဉ်ပုံဖော်ကြည့်နိုင်ပါတယ်။ ဥပမာ – ပန်းသီး ဆိုတဲ့ ရုပ်ဝတ္ထုလေး တစ်ခုမှာ ကိုယ်ပိုင်ဂုဏ်သတ္တိတစ်ချို့ရှိပါတယ်။ အခွံအနီရောင်ရှိတယ်။ ချိုချဉ်တဲ့အရသာရှိတယ်။ ဒါကသက်မဲ့ ရုပ်ဝတ္ထုလေးတစ်ခုပါ။ သက်ရှိဖြစ်တဲ့ ကြောင်လေးတစ်ကောင်ဆိုရင်တော့ အနက်ရောင်အမွေးအမြှင်၊ မျက်လုံးလေးနှစ်လုံး၊ အမြီးလေးတစ်ချောင်းနဲ့ ခြေထောက်လေးခြောင်း စတဲ့ ပိုင်ဆိုင်မှုဂုဏ်သတ္တိတွေအပြင်၊ အသံပေးနိုင်တယ်၊ ပြေးလွှားနိုင်တယ်၊ ကြွက်ခုတ်နိုင်တယ်ဆိုတဲ့ အပြုအမူလေးတွေပါ ရှိသွားပါတယ်။

ပရိုဂရမ် Object တွေဟာလည်း ဒီသဘောပါပဲ။ တစ်ချို့ Object တွေမှာ ကိုယ်ပိုင်ဂုဏ်သတ္တိလေးတွေ ရှိကြတယ်။ Property လို့ ခေါ်ကြလေ့ ရှိပါတယ်။ တစ်ချို့ကတော့ ကိုယ်ပိုင်ဂုဏ်သတ္တိအပြင် အပြုအမူလေးတွေပါ ရှိကြတယ်။ Method လို့ ခေါ်ကြလေ့ ရှိပါတယ်။ ဒီလို Property တွေ Method တွေ စုဖွဲ့ပါဝင်တဲ့ Object တွေကို ကိုယ်တိုင် တည်ဆောက်ယူလို့ ရနိုင်သလို သက်ဆိုင်ရာ Language က တစ်ခါထဲ ထည့်ပေးထားတဲ့ အသင့်သုံး Standard Object တွေလည်း ရှိနိုင်တယ်။

Array ဟာ JavaScript ရဲ့ Standard Object တစ်ခုဖြစ်ပါတယ်။ သူ့မှာ သူ့ကိုယ်ပိုင် Property တွေနဲ့ Method တွေ အသင့်ပါဝင်တယ်။ တစ်ချို့ Language တွေမှာတော့ Array ဆိုတာ Data Type တစ်မျိုးဖြစ်ပါတယ်။ ရိုးရိုး Primitive Data Type တွေက တန်ဖိုး တစ်ခုကိုသာ လက်ခံသိမ်းဆည်းပြီး Array ကတော့ တန်ဖိုးတွေကို အတွဲလိုက်စုစည်းသိမ်းဆည်းနိုင်လို့ Compound Type သို့မဟုတ် Structure Type လို့ ခေါ်ကြပါတယ်။ JavaScript မှာ တော့ Compound Type/Structure Type ဘဘောသဘာဝမျိုး လိုချင်ရင် Object ရဲ့ Property အဖြစ် တန်ဖိုးတွေကို အတွဲလိုက် စုစည်းသိမ်းဆည်းနိုင်ပါတယ်။ တစ်ခြား Language တွေရဲ့ Array Type နဲ့ သဘောသဘာဝ ဆင်တူတဲ့လုပ်ဆောင်ချက်ကို ရရှိစေဖို့အတွက် Array လို့ခေါ်တဲ့ Standard Object တစ်ခုကို အသင့်ထည့်သွင်း ပေးထားတာပါ။

Array တစ်ခုတည်ဆောက်ဖို့အတွက် နည်းလမ်း (၂) မျိုးရှိပါတယ်။ တစ်နည်းက Array Object Constructor ကို အသုံးပြုတဲ့နည်းပါ။

JavaScript

```
let mix = new Array("Bob", 3.14, true)
```

`new` Keyword ကိုသုံးပြီး Object အသစ်တွေတည်ဆောက်ရပါတယ်။ Array() လို့ခေါ်တဲ့ Standard Object Constructor ကိုသုံးပြီး တည်ဆောက်စေလိုက်တာ ဖြစ်တဲ့အတွက် `mix` ဟာ Array Object တစ်ခုဖြစ်သွားပါပြီ။ ဒီလို တည်ဆောက်စဉ်မှာ တန်ဖိုးတွေကိုလည်း တစ်ခါထဲ Comma ခံပြီး တန်းစီပေးလိုက်တဲ့အတွက် `mix` Array ထဲမှာ Bob ဆိုတဲ့ String တန်ဖိုးတစ်ခု၊ 3.14 ဆိုတဲ့ Number တန်ဖိုးတစ်ခုနဲ့ true ဆိုတဲ့ Boolean တန်ဖိုးတစ်ခု၊ စုစုပေါင်း တန်ဖိုးသုံးခုကို အတွဲလိုက် ထည့်သွင်းပေးထားမှာပဲ ဖြစ်ပါတယ်။ ဒီလိုပုံစံမျိုးပါ –

0	1	2
Bob	3.14	true

ထည့်သွင်းလိုက်တဲ့ တန်ဖိုးတိုင်းမှာ ကိုယ်ပိုင် Index အညွှန်းကိုယ်စီလည်း ရှိသွားမှာဖြစ်ပါတယ်။ Index အညွှန်းတွေဟာ အမြဲတမ်း 0 ကစတဲ့အတွက် Index 0 မှာ Bob ဆိုတဲ့တန်ဖိုးရှိနေမှာဖြစ်ပြီး Index 1 မှာ 3.12 ဆိုတဲ့တန်ဖိုး ရှိနေမှာ ဖြစ်ပါတယ်။ Index 2 မှာတော့ true ဆိုတဲ့တန်ဖိုး ရှိနေမှာပါ။

Array တည်ဆောက်နည်း နောက်တစ်နည်းကတော့ လေးထောင်ကွင်း အဖွင့်အပိတ်ကို အသုံးပြုခြင်းဖြစ်ပါတယ်။ Single Quote / Double Quote အဖွင့်အပိတ်တွေကို String တည်ဆောက်ပေးနိုင်တဲ့ String Literal လို့ခေါ်ပြီး လေးထောင့်ကွင်း အဖွင့်အပိတ်ကိုတော့ Array တွေတည်ဆောက်ပေးနိုင်တဲ့ Array Literal လို့ ခေါ်ပါတယ်။

JavaScript

```
let mix = [ "Bob", 3.14, true ]
```

ဒီနည်းနဲ့ တန်ဖိုး (၃) ခု အတွဲလိုက်ပါဝင်တဲ့ `mix` Array တစ်ခု ရသွားပါတယ်။ အကယ်၍ Array အလွတ်ကို တည်ဆောက်လိုရင် လေးထောင့်ကွင်းအဖွင့်နဲ့အပိတ်ကို အလွတ်အတိုင်း ပေးလိုက်ယုံပါပဲ။ စောစောကပြောတဲ့ Array Object Constructor ကိုအသုံးပြုတဲ့နည်းထက် ဒီ Array Literal ကိုသုံးရတဲ့နည်းကို ပိုအသုံးများပါတယ်။ ရေးရတဲ့ကုဒ် ကျစ်လစ်တိုတောင်းလို့ပါ။

JavaScript Array တွေမှာ ထည့်သွင်းသိမ်းဆည်းနိုင်တဲ့ Data Type ကန့်သတ်ချက်မရှိပါဘူး။ နှစ်သက်ရာ အမျိုးအစားကို တွဲဖက်သိမ်းဆည်းနိုင်ပါတယ်။ ပြီးတော့ အရေအတွက်ပုံသေလည်း မရှိပါဘူး။ (၅) ခုပဲ ထည့်လို့ရမယ်၊ (၁၀) ခုပဲ ထည့်လို့ရမယ်ဆိုတာမျိုး မရှိဘဲ၊ လိုအပ်သလောက် ထပ်တိုးထည့်သွင်း သွားလို့ ရနိုင်ပါတယ်။ တစ်ချို့ Statically Typed Language တွေမှာဆိုရင်တော့ သိမ်းဆည်းလို့ရတဲ့ Data Type ကို ကြိုတင်ကြေညာပေးဖို့ လိုနိုင်ပြီး ကြိုတင်ကြေညာထားတဲ့ အမျိုးအစားကလွဲရင် တစ်ခြား အမျိုးအစားတွေကိုလက်ခံထည့်သွင်းပေးမှာ မဟုတ်ပါဘူး။ ပြီးတော့ ပါဝင်မယ့် အရေအတွက်ကို လည်းကြိုတင်ကြေညာပေးထားဖို့လိုအပ်နိုင်ပါတယ်။ ဥပမာ –

Pseudocode

```
let nums [i32, 5] = [ 1, 2, 3, 4, 5 ];
let mix [str, f32, bool] = [ "Bob", 3.14, true ];
```

ပထမတစ်ကြောင်းက `i32` အမျိုးအစား ကိန်းဂဏန်းတွေသိမ်းမယ်၊ (၅) ခုသိမ်းမယ်လို့ ကြေညာလိုက်တာပါ။ ဒါကြောင့် (၅) ခုအတိအကျသိမ်းပေးရမှာဖြစ်သလို သိမ်းလို့ရမယ့်အမျိုးအစားကလည်း 32-bit Integer တစ်မျိုးထဲပဲ သိမ်းလို့ရမှာပါ။ နောက်တစ်ကြောင်းမှာတော့ String, 32-bit Float နဲ့ Boolean စုစုပေါင်း သုံးခုသိမ်းမယ်လို့ သတ်မှတ်ထားတဲ့အတွက် သတ်မှတ်ထားတဲ့အတိုင်း အတိအကျသိမ်းပေးရမှာပါ။ တစ်ခြား အမျိုးအစားတွေ သိမ်းလို့ရမှာ မဟုတ်ပါဘူး။ ဒါက တစ်ချို့ Statically Typed Language မှာဖြစ်နိုင်တာကို ပြောတာပါ။ JavaScript မှာတော့ အဲ့ဒီလို ကန့်သတ်ချက်မျိုးတွေ မရှိပါဘူး။

Array တစ်ခုထဲမှာ ထည့်သွင်းထားတဲ့ တန်ဖိုးတွေကို လိုအပ်တဲ့အခါ Index အညွှန်းနံပါတ်ကိုသုံးပြီး ပြန်လည်ရယူ အသုံးပြုနိုင်ပါတယ်။

JavaScript

```
let mix = [ "Bob", 3.14, true ]

let name = mix[0]    // Bob
let num = mix[1]     // 3.14
let out = mix[3]     // undefined
```

နမူနာမှာ `mix` Array နောက်က လေးထောင့်ကွင်းအတွင်းမှာ Index နံပါတ်ကိုပေးပြီး Array ထဲက လိုချင်တဲ့ တန်ဖိုးကို ရယူထားပါတယ်။ မရှိတဲ့ တန်ဖိုးကို ယူဖို့ကြိုးစားတဲ့အခါမှာတော့ `undefined` ကို ရရှိတာတွေ့ရမှာပဲ ဖြစ်ပါတယ်။

တန်ဖိုးတွေပြင်လိုရင်ပဲဖြစ်ဖြစ်၊ ထပ်မံထည့်သွင်းလိုရင်ပဲဖြစ်ဖြစ် နှစ်သက်ရာ Index နံပါတ်ပေးပြီး ထည့်လို့ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
let mix = [ "Bob", 3.14, true ]

mix[0] = "Alice"
mix[4] = 5
```

မူလကြေညာစဉ်က တန်ဖိုး (၃) ခု ပါပြီး နောက်မှ Index 4 မှာ တန်ဖိုးတစ်ခုကို ထပ်ထည့်ထားတာပါ။ ရလဒ်က အခုလိုပုံစံ ဖြစ်မှာပါ။

0	1	2	3	4
Alice	3.14	true		5

မူလတန်ဖိုး (၃) ခုတို့ဟာ Index 0, 1, 2 တို့မှာ အသီးသီးရှိနေပါတယ်။ Index 0 တန်ဖိုးကတော့ နောက်မှ ပြောင်းပေးလိုက်လို့ ပြောင်းနေပါပြီ။ Index 3 က ဘာတန်ဖိုးမှ မရှိသေးတဲ့ အခန်းလွတ်တစ်ခုအနေနဲ့ ဝင်ရောက်သွားတာကို သတိပြုဖို့လိုပါလိမ့်မယ်။ 0, 1, 2 ပြီးရင် 3 လာရမှာဖြစ်ပေမယ့် တစ်ဆင့်ကျော်ပြီး 4 မှာ တန်ဖိုးသစ်တစ်ခု ထည့်လိုက်မိလို့ပါ။

Array တိုင်းမှာ သူ့ကိုယ်ပိုင်အနေနဲ့ `length` Property ရှိပါတယ်။ အဲ့ဒီ `length` Property ရဲ့တန်ဖိုးအနေနဲ့ Array ရဲ့ Size ရှိနေမှာဖြစ်ပါတယ်။ အသုံးဝင်ပါတယ်။ Array Object အပါအဝင် Object တွေရဲ့ Property တွေ Method တွေကို အသုံးပြုလိုတဲ့အခါ သက်ဆိုင်ရာ Object ပေါ်မှာ Dot Operator ကိုသုံးပြီး ရယူအသုံးပြုနိုင်ပါတယ်။

JavaScript

```
let fruits = [ "Apple", "Orange" ]

fruits.length                   // 2
fruits[2] = "Mango"
fruits.length                   // 3
fruits[fruits.length - 1]       // Mango
```

မူလ `fruits` Array မှာ Index နှစ်ခုရှိတဲ့အတွက် သူရဲ့ `length` တန်ဖိုး 2 ဖြစ်နေပြီး၊ နောက်ထပ် Index တစ်ခုထပ်တိုးလိုက်တဲ့အခါ `length` တန်ဖိုး 3 ဖြစ်သွားတာကို တွေ့မြင်ရခြင်းဖြစ်ပါတယ်။ ပြီးတဲ့အခါ `length` ရဲ့လက်ရှိတန်ဖိုး ဖြစ်နေချိန်မှာ `length - 1` နဲ့ `fruits` Array ရဲ့ Index ကို ထောက်လိုက်တဲ့အခါ `fruits[2]` ကို ထောက်လိုက်သကဲ့သို့ ဖြစ်တဲ့အတွက် `fruits` Array ရဲ့ လက်ရှိ Index 2 မှာ ရှိနေတဲ့ တန်ဖိုးကို ပြန်ရတာကိုလည်း တွေ့ရပါလိမ့်မယ်။ ဒီနည်းကို Array ရဲ့နောက်ဆုံးတန်ဖိုးယူဖို့ သုံးကြလေ့ ရှိပါတယ်။

Array တစ်ခုထဲမှာ ရိုးရိုး String, Number စတဲ့ တန်ဖိုးတွေသာမက တစ်ခြား Array တွေလည်း ရှိနိုင်ပါသေးတယ်။ ဒီလိုပါ –

JavaScript

```
let mix = [ [123, 456, 789], ['Ant', 'Cat', 'Dog'] ]
```

ပင်မ mix Array ဟာ Index နှစ်ခုရှိတဲ့ Array တစ်ခုဖြစ်ပြီး သူ့ထဲမှာ နောက်ထပ် Array တွေထပ်ရှိနေတာပါ။ ဖွဲ့စည်းပုံက အခုလို ဖြစ်ပါလိမ့်မယ်။

0		
0	1	2
123	456	789

1		
0	1	2
Ant	Cat	Dog

ဒီလိုထပ်ဆင့်ရှိနေတဲ့ Array ထဲကတန်ဖိုးတွေကိုလည်း လိုသလို ရယူပြင်ဆင်လို့ရပါတယ်။ Index အညွှန်းကိုသာ ထပ်ဆင့်မှန်အောင်ပေးဖို့ လိုတာပါ။

```
let nums = mix[0]          // [ 123, 456, 789 ]
let animals = mix[1]       // [ 'Ant', 'Cat', 'Dog' ]
let x = mix[0][1]          // 456
let rambo = mix[1][2]      // Dog
```

Array Methods

Array Data တွေကို စီမံဖို့အတွက် အသုံးဝင်တဲ့ Default Method တွေ ရှိပါတယ်။ Array တစ်ခုတည်ဆောက်လိုက်တာနဲ့ အဲဒီ Method တွေကို Array ပေါ်မှာ အသင့်သုံးလို့ရသွားမှာပါ။ အတွဲလိုက်မှတ်သားသင့်တဲ့ Method (၄) ခုကတော့ `push()`, `pop()`, `shift()` နဲ့ `unshift()` တို့ပဲ ဖြစ်ပါတယ်။

- **`push()`** Method ကို Array ရဲ့နောက်ဆုံးကနေ Index တစ်ခုတိုးဖို့အတွက် သုံးနိုင်ပါတယ်။
- **`pop()`** Method ကိုတော့ Array ရဲ့နောက်ဆုံး Index ကို ဖယ်ထုတ်ဖို့သုံးနိုင်ပါတယ်။
- **`shift()`** Method ကိုတော့ Array ရဲ့ရှေ့ဆုံး Index ကိုဖယ်ထုတ်ဖို့ သုံးပါတယ်။
- **`unshift()`** Method ကိုတော့ Array ရဲ့ရှေ့ဆုံးမှာ Index တစ်ခုတိုးဖို့ သုံးနိုင်ပါတယ်။

```
let animals = ["Dog", "Cat", "Bird"]

animals.push("Cow")  // animals → ["Dog", "Cat", "Bird", "Cow"]
animals.pop()              // animals → ["Dog", "Cat", "Bird"]
animals.shift()            // animals → ["Cat", "Bird"]
animals.unshift("Ant")     // animals → ["Ant", "Cat", "Bird"]
```

`push()` အတွက် နောက်ဆုံးမှာ ထပ်တိုးလိုတဲ့ တန်ဖိုးကို Argument အနေနဲ့ ပေးရပါတယ်။ `unshift()` အတွက်လည်း ရှေ့ဆုံးမှာ ထပ်တိုးလိုတဲ့ တန်ဖိုးကို ပေးရပါတယ်။ `pop()` နဲ့ `shift()` အတွက်တော့ Argument တွေ မလိုအပ်ပါဘူး။ နောက်ထပ်မှတ်သားသင့်တဲ့ အတွဲအဖက် ကတော့ `indexOf()` နဲ့ `splice()` ဖြစ်ပါတယ်။ `indexOf()` နဲ့ Index တန်ဖိုးကို ရှာနိုင်ပြီး၊ `splice()` နဲ့ မလိုချင်တဲ့ Index ကို ဖယ်ထုတ်နိုင်ပါတယ်။

JavaScript

```
let fruits = ["Apple", "Orange", "Mango", "Banana"]

fruits.indexOf("Mango")     // 2
fruits.splice(2, 1)     // fruits → ["Apple", "Orange", "Banana"]
```

Index (၄) ခုပါတဲ့ `fruits` Array ကနေ `indexOf()` နဲ့ Mango ကိုရှာလိုက်တဲ့အခါ Index 2 ဖြစ်ကြောင်းသိရပါတယ်။ `splice()` ကိုသုံးပြီး Index 2 ကိုဖျက်ခိုင်းလိုက်တဲ့အတွက် မူလ `fruits` Array မှာ Index (၃) ခုပဲကျန်တော့တာကို တွေ့ရခြင်း ဖြစ်ပါတယ်။ `splice()` အတွက် Argument နှစ်ခုပေးရပါတယ်။ ပထမ Argument က Index ဖြစ်ပြီး ဒုတိယ Argument ကတော့ အရေအတွက် ဖြစ်ပါတယ်။ အရေအတွက်မှာ 1 ကို ပေးလို့ တစ်ခုဖျက်ပေးတာပါ။ ဖျက်လိုတဲ့အရေအတွက်ကို လိုသလို ပေးနိုင်ပါတယ်။

နောက်ထပ်အသုံးဝင်တဲ့ Array Method ကတော့ `join()` ဖြစ်ပါတယ်။ Array Index တွေကို တစ်ဆက်ထဲ တစ်တွဲထဲ ဖြစ်သွားအောင် တွဲဆက်ပြီး String အနေနဲ့ ရလဒ်ကို ပြန်ပေးနိုင်တဲ့ လုပ်ဆောင်ချက်ဖြစ်ပါတယ်။

```
let fruits = ["Apple", "Orange", "Mango"]

let result = fruits.join(",")     // result → Apple,Orange,Mango
```

`join()` Method အတွက် Argument အနေနဲ့ တွဲဆက်ရာမှာ ကြားခံထားပေးစေလိုတဲ့ တန်ဖိုးကို ပေးနိုင်ပါတယ်။ နမူနာမှာ Comma တစ်ခုကို ပေးခဲ့လို့ Comma ခံပြီးတွဲဆက်ပေးသွားတာပါ။ ဘာမှမပေးရင်လည်း ရပါတယ်။ ကြားခံမထားတော့ဘဲ အကုန်လုံးကို တွဲဆက်ပေးသွားမှာပဲ ဖြစ်ပါတယ်။

ဒီနေရာမှာ တစ်ခုသတိပြုဖို့လိုပါတယ်။ တစ်ချို့ Method က မူလ Array ရဲ့ တန်ဖိုးတွေကို ပြုပြင်လိုက်တာ ဖြစ်ပြီး။ တစ်ချို့ Method တွေက မူလ Array ရဲ့တန်ဖိုးကို ပြောင်းလဲစေခြင်းမရှိဘဲ ရလဒ်ကို ပြန်ပေးတာ ဖြစ်ပါတယ်။ `push()`, `pop()`, `shift()`, `unshift()`, `splice()` စတဲ့ Method တွေက မူလ Array တန်ဖိုးတွေကို ပြုပြင် ပြောင်းလဲသွားစေတဲ့ Method တွေပါ။ `join()` ကတော့ မူလ Array တန်ဖိုးတွေကို မပြောင်းပါဘူး။ ရလာတဲ့ ရလဒ်ကိုသာ ပြန်ပေးတာပါ။ ဒါကြောင့်လည်း နမူနာမှာ ရလဒ်ကို Variable တစ်ခုမှာ Assign လုပ်ပြထားတာပါ။ ဒီလိုမတူကွဲပြားတဲ့သဘောသဘာဝလေးက အရေးကြီးပါတယ်။ ဂရုပြုမှတ်သားသင့်ပါတယ်။

ထပ်မံမှတ်သားသင့်တဲ့ အသုံးဝင်တဲ့ Array Method တွေကတော့ `map()`, `filter()` နဲ့ `reduce()` တို့ဖြစ်ပါတယ်။

- **`map`**`()` Method အတွက် Argument အနေနဲ့ Function Expression တစ်ခုပေးရပါတယ်။ ပေးလိုက်တဲ့ Function Expression ကို Array Item အားလုံးပေါ်မှာ အလုပ်လုပ်ပြီး နောက်ဆုံး ရလာတဲ့ရလဒ်ကို Array အနေနဲ့ ပြန်ပေးမှာပါ။ Array ထဲကတန်ဖိုးတွေကို တစ်ခုချင်းလိုသလို ပြုပြင်ရယူဖို့ အသုံးဝင်ပါတယ်။
- **`filter`**`()` Method အတွက်လည်း Function Expression တစ်ခုကို Argument အနေနဲ့ပေးရပါတယ်။ Array ထဲက လိုချင်တဲ့ Item တွေကို ရွေးယူချင်ရင် အသုံးဝင်ပါတယ်။
- **`reduce`**`()` Method အတွက်လည်း Function Expression တစ်ခုကို ပေးရတာပါပဲ။ သူကတော့ Array Item အားလုံးပေါ်မှာအလုပ်လုပ်ပြီး နောက်ဆုံးရလဒ်တန်ဖိုးတစ်ခုကို ပြန်ပေးပါတယ်။ ဥပမာ – Array ထဲမှရှိသမျှ တန်ဖိုးအားလုံးကို ပေါင်းလိုက်ပြီး ပေါင်းခြင်းရလဒ်ကို ပြန်ပေးတာမျိုးပါ။

ဒီ Method တွေဟာ တော်တော်လေးအသုံးဝင်တဲ့ Method တွေပါ။ လက်တွေ့ပရောဂျက်တွေမှာလည်း အသုံးများကြပါတယ်။ မူလ Array တန်ဖိုးတွေကို ထိခိုက်ပြောင်းလဲစေခြင်း မရှိဘဲ ရလဒ်ကို သီးခြားအနေနဲ့ ပြန်ပေးတဲ့ Method တွေပါ။

JavaScript

```
let nums = [1, 2, 3, 4, 5]

let result = nums.map(function(n) {
    return n + 1
})

// result → [2, 3, 4, 5, 6]
```

Array မှာ Index (၅) ခုရှိပြီး ပေးလိုက်တဲ့ Function Expression က (၅) ခုလုံးပေါ်မှာ တစ်ခုပြီးတစ်ခု အလုပ်လုပ်သွားမှာပါ။ Function Expression ရဲ့ Parameter ဖြစ်တဲ့ n ဟာ လက်ရှိအလုပ်လုပ်နေတဲ့ တန်ဖိုးဖြစ်ပါတယ်။ အဲ့ဒီတန်ဖိုးကို 1 နဲ့ပေါင်းပြီး ပြန်ပေးထားလို့ Array ထဲမှာ ရှိသမျှ တန်ဖိုးအားလုံးကို 1 တိုးလိုက်တဲ့ လုပ်ဆောင်ချက်ကို ရရှိစေပါတယ်။

JavaScript

```
let nums = [1, 2, 3, 4, 5]

let result = nums.filter(function(n) {
    return n % 2
})

// result → [1, 5, 7]
```

`filter()` ရဲ့ထူးခြားချက်ကတော့ `map()` လို Item အားလုံးကို ပြန်ပေးတာ မဟုတ်တော့ဘဲ `return true` ဖြစ်တဲ့ Item တွေကိုပဲ ပြန်ပေးတာပါ။ နမူနာအရ % သင်္ကေတကိုသုံးပြီး လက်ရှိတန်ဖိုးကို 2 နဲ့စားပြီး အကြွင်းရှာထားပါတယ်။ စာလို့ပြတ်ရင် 0 မို့လို့ false ပါ။ စားလို့မပြတ်ရင်တော့ အကြွင်းတန်ဖိုးတစ်ခုရမှာမို့လို့ true ပါ။ 2 နဲ့စားလို့ မပြတ်တဲ့ တန်ဖိုးဆိုရင် `return true` ဖြစ်မှာမို့လို့ ရလဒ်အနေနဲ့ 2 နဲ့စားလို့ မပြတ်တဲ့ မဂဏန်းတွေချည်းပဲ ပြန်ရတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ စုံကဂဏန်းတွေ လိုချင်ရင် လွယ်ပါတယ်။ NOT Operator ကိုသုံးလိုက်ယုံပါပဲ။ ဒီလိုပါ –

JavaScript

```
let nums = [1, 2, 3, 4, 5]

let result = nums.filter(function(n) {
    return !(n % 2)
})

// result → [2, 4]
```

ဒီနေရာမှာ အတိုကောက် Function Expression တွေသုံးရင် အများကြီး ပိုရှင်းပြီး တိုတောင်းကျစ်လစ်သွားမှာပါ။ ဒီလိုပါ –

JavaScript

```
let nums = [1, 2, 3, 4, 5]

let result = nums.map(n => n + 1)     // result → [2, 3, 4, 5, 6]
let odd = nums.filter(n => n % 2)     // odd → [1, 3, 5]
```

တစ်ကြောင်းထဲနဲ့ လိုချင်တဲ့ရလဒ်ကို ရသွားတာပါ။ ရိုးရိုး Function နဲ့ အတူတူပါပဲ။ `function, (), {}` နဲ့ `return` တို့ကိုဖယ်ထုတ်လိုက်ပြီး `=>` လေးထည့်ပေးလိုက်တာပါပဲ။ ဒီလုပ်ဆောင်ချက်တွေကို တွဲပြီးသုံးမယ်ဆိုရင်လည်း အခုလို သုံးနိုင်ပါတယ်။

```
let nums = [1, 2, 3, 4, 5]
let result = nums.map(n => n + 2).filter(n => n % 2)

// result → [3, 5, 7]
```

JavaScript Object Method တွေရဲ့ ထူးခြားချက်က အခုလို အတွဲလိုက် ဆက်ပြီး သုံးလို့ရခြင်း ဖြစ်ပါတယ်။ Method Chaining လို့ခေါ်ပါတယ်။ ဒီနေရာမှာ သတိပြုရမှာက `map()` ဟာ `nums` Array ပေါ်မှာ အလုပ်လုပ်သွားတာပါ။ `filter()` ကတော့ `map()` ကပြန်ပေးတဲ့ ရလဒ် Array ပေါ်မှာ အလုပ်လုပ်သွားတာပါ။ `nums` ပေါ်မှာ အလုပ်လုပ်တာ မဟုတ်ပါဘူး။

```
nums.map() → [ Array ].filter() → [ Array ]
```

ဒါလေးကို ကွဲကွဲပြားပြားမြင်မှ အလုပ်လုပ်ပုံကို ပုံဖော်ကြည့်နိုင်မှာပါ။ နောက်ဆုံးတစ်ခုကျန်နေတဲ့ `reduce()` ကိုတော့ အခုလို အသုံးပြုနိုင်ပါတယ်။

JavaScript

```
let nums = [2, 3, 4, 5, 6]

let result = nums.reduce(function(a, n) {
    return a + n
})

// result → 20
```

`reduce()` ရဲ့ Function Expression အတွက် Parameter နှစ်ခုရှိတာကို သတိပြုပါ။ နမူနာအရ a က Accumulative Value ခေါ် အလုပ်လုပ်လက်စ တန်ဖိုးဖြစ်ပြီး n က လက်ရှိအလုပ်လုပ်နေတဲ့ Item ရဲ့ တန်ဖိုးဖြစ်ပါတယ်။ ပထမတစ်ကြိမ်မှာ a က `null` ဖြစ်ပြီး n က 2 ဖြစ်ပါတယ်။ `null + 2` က 2 ပဲမို့လို့ a တန်ဖိုး 2 ဖြစ်သွားပါတယ်။ ဒုတိယတစ်ကြိမ်မှာ a တန်ဖိုး 2 ဖြစ်နေပြီး n တန်ဖိုးက 3 ဖြစ်နေမှာပါ။ a + n ဟာ 2 + 3 မို့လို့ ရလဒ် 5 ပါ။ a တန်ဖိုး 5 ဖြစ်သွားပါတယ်။ ဒီနည်းနဲ့ တစ်ခုပြီးတစ်ခု ဆက်တိုက် အလုပ်လုပ်သွားတဲ့အခါ နောက်ဆုံးမှာ Array ထဲက တန်ဖိုးအားလုံးကို စုပေါင်းထားတဲ့ ရလဒ် 20 ကို ရရှိခြင်းဖြစ်ပါတယ်။ အတိုကောက်အခုလို ရေးနိုင်ပါတယ်။

```
let result = [2, 3, 4, 5, 6].reduce((a, n) => a+ n)

// result → 20
```

Array Spread & Destructuring

Array တွေကိုပြန်ဖြည်ထုတ်လို့ရတဲ့ လုပ်ဆောင်ချက်တွေကိုလည်း ထည့်သွင်းမှတ်သားသင့်ပါတယ်။ Array Spread လို့ခေါ်တဲ့ လုပ်ဆောင်ချက်ဟာ Array တစ်ခုအတွင်းက Item တွေကို တန်ဖိုးတစ်ခုချင်းစီအဖြစ် ခွဲထုတ်ပေးနိုင်ပါတယ်။ အသုံးဝင်နိုင်မယ့် နမူနာလေးတစ်ချို့ ပေးချင်ပါတယ်။

```JavaScript
let nums = [1, 2, 3]
let alphas = ['a', 'b', 'c']
let result = [ nums, alphas ]

// [ [1, 2, 3], ['a', 'b', 'c'] ]
```

နမူနာအရ `result` Array ထဲမှာ `nums` Array နဲ့ `alphas` Array တို့ကို ထည့်ပေးလိုက်လို့ Array နှစ်ထပ် ဖြစ်သွားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ အဲ့ဒီလို နှစ်ထပ်လိုချင်တာဆိုရင် ပြဿနာမရှိပေမယ့်၊ အားလုံးကို တစ်ထပ်ထဲ တစ်ဆက်ထဲဖြစ်စေချင်တာဆိုရင် ဒီအတိုင်းမရတော့ပါဘူး။ Spread Operator ကို သုံးနိုင်ပါတယ်။ ဒီလိုပါ –

```JavaScript
let nums = [1, 2, 3]
let alphas = ['a', 'b', 'c']
let result = [ ...nums, ...alphas ]

// [ 1, 2, 3, 'a', 'b', 'c' ]
```

... Operator က Array ထဲက Item တွေ တန်ဖိုးတစ်ခုစီဖြစ်သွားအောင် ခွဲထုတ်ပေးလိုက်တဲ့အတွက် `result` Array ဟာ တစ်ဆက်ထဲ တစ်ထပ်ထဲ ဖြစ်သွားတာကို တွေ့ရမှာဖြစ်ပါတယ်။ ဒီနည်းကိုသုံးပြီး Array ရဲ့ ရှေ့ (သို့မဟုတ်) နောက်မှာ တန်ဖိုးတွေ ထပ်တိုးဖို့လည်း သုံးကြပါတယ်။ ဒီလိုပါ။

```JavaScript
let nums = [1, 2, 3]
let four = [ ...nums, 4 ]     // [1, 2, 3, 4]
let zero = [ 0, ...nums ]     // [0, 1, 2, 3]
```

သတိပြုရမှာကတော့ `push()` တို့ `unshift()` တို့လို မူလ Array တန်ဖိုးတွေကို ပြင်လိုက်တာ မဟုတ်ဘဲ Array အသစ်တစ်ခု တည်ဆောက်ပေးလိုက်တာပဲ ဖြစ်ပါတယ်။ နောက်ဥပမာ တစ်ခုပြဖို့အတွက် အခုလို Function တစ်ခုရှိတယ်လို့ သဘောထားပါ။

```JavaScript
function add(a, b) {
    return a + b
}
```

ရိုးရိုးလေးပါ။ Parameter နှစ်ခုရှိတဲ့ Function တစ်ခုဖြစ်လို့ ခေါ်ယူတဲ့အခါ Argument နှစ်ခုပေးရမှာပါ။ အခုလို Array တစ်ခုရှိတယ် ဆိုကြပါစို့ –

```JavaScript
let nums = [123, 456]
```

ဒီ Array ထဲကတန်ဖိုးတွေကို Argument အနေနဲ့ ပေးပြီး `add` Function ကို ခေါ်ချင်တယ်ဆိုရင် အခုလို ခေါ်ရမှာပါ။

```JavaScript
add(nums[0], nums[1])      // 579
```

ပထမ Argument အတွက် `nums[0]` ကို ပေးလိုက်ပြီး ဒုတိယ Argument အတွက် `nums[1]` ကို ပေးလိုက်တာပါ။ အဲ့ဒီလို လေးထောင့်ကွင်းတွေနဲ့ Index တစ်ခုချင်းထောက်ပေးနေမယ့်အစား အခုလို ပေးလိုက်လို့ ရနိုင်ပါတယ်။

```JavaScript
add(...nums)               // 579
```

ဒါဆိုရင် Spread Operator က Array ထဲကတန်ဖိုးတွေကို ခွဲဖြန့်ပေးလိုက်လို့ တူညီတဲ့ရလဒ်ကို ရရှိမှာပဲ ဖြစ်ပါတယ်။ ဒါမျိုးလေးတွေက အသေးအဖွဲ့ ဖြည့်စွက်ချက်လေးတွေလို့ ပြောလို့ရပါတယ်။ ဒါပေမယ့် သိထားဖို့လိုပါတယ်။ သိမထားရင် ဒီနည်းနဲ့ရေးထားတဲ့ ကုဒ်တွေတွေ့တဲ့အခါ ဘာကိုဆိုလိုမှန်း နားမလည်တာမျိုး ဖြစ်သွားပါလိမ့်မယ်။ ပရိုဂရမ်းမင်းကို လေ့လာတဲ့အခါ သူများရေးထားတဲ့ ကုဒ်တွေကိုဖတ်ပြီး လေ့လာတာဟာ ထိရောက်တဲ့နည်းတစ်ခုဖြစ်လို့ သူများရေးထားတဲ့ ကုဒ်ကို ဖတ်တတ်ဖို့လည်း လိုပါတယ်။

Array Destructuring လုပ်ဆောင်ချက်လည်း ရှိပါသေးတယ်။ ဒီလိုပါ –

JavaScript

```
let nums = [123, 456]

let a = nums[0]
let b = nums[1]
```

ဒါက ရိုးရိုးရှေးလိုက်တာပါ။ Array တစ်ခုရှိပြီး အဲ့ဒီ Array ထဲက တန်ဖိုးတွေကို Index တစ်ခုချင်းထောက်ပြီး သုံးလိုက်တာပါ။ အဲ့ဒီလို တစ်ခုချင်းထောက်မယ့်အစား အခုလိုလည်း ရေးလို့ရနိုင်ပါတယ်။

JavaScript

```
let nums = [123, 456]
let [a, b] = nums
```

တူညီတဲ့ရလဒ်ကို ရမှာပဲဖြစ်ပါတယ်။ Index တစ်ခုချင်းထောက်မနေဘဲ Array တစ်ခုလုံးကို ပေးလိုက်တာပါ။ လက်ခံတဲ့အခါမှာ Destructuring ရေးထုံးကိုသုံးပြီး လက်ခံထားလို့ တန်ဖိုးတွေက သူ့နေရာနဲ့သူ ရောက်ရှိသွားမှာပဲ ဖြစ်ပါတယ်။ Function နဲ့တွဲသုံးတဲ့ နမူနာလေး တစ်ခုလည်း ပေးပါဦးမယ်။

JavaScript

```
function add([a, b]) {
    return a + b
}
```

ဒီတစ်ခါ `add` Function မှာ Parameter တစ်ခုပဲ ရှိပါတယ်။ Index နှစ်ခုနဲ့ Destructure လုပ်ပြီးလက်ခံထားတဲ့ Parameter ဖြစ်လို့ ခေါ်ယူအသုံးပြုတဲ့အခါ Index နှစ်ခုရှိတဲ့ Array ကို Argument အနေနဲ့ ပေးရမှာပါ။

JavaScript

```
let nums = [123, 456]

add(nums)       // 579
```

Argument အနေနဲ့ `nums` Array ကိုပေးလိုက်ပေးမယ့်၊ လက်ခံစဉ်ကတည်းက a နဲ့ b အဖြစ် Destructure လုပ်ပြီးလက်ခံထားလို့ Function ရဲ့အတွင်းမှာ အသုံးပြုတဲ့အခါ a နဲ့ b ကို တိုက်ရိုက်အသုံးပြုနိုင်မှာပဲ ဖြစ်ပါတယ်။

String Object

JavaScript က အရာတော်တော်များများကို Object ကဲ့သို့ အသုံးပြုနိုင်အောင် စီစဉ်ပေးထားပါတယ်။ ရှေ့ပိုင်းမှာ လေ့လာခဲ့ပြီးဖြစ်တဲ့ String, Boolean, Number, Function တို့လို အခြေခံလုပ်ဆောင်ချက်တွေကအစ Object Wrapper ခေါ် အလွှာတစ်ထပ် အုပ်ထားပေးလို့ အဲ့ဒီ String, Boolean, Number, Function တွေအားလုံးကို Object ကဲ့သို့ အသုံးပြုနိုင်ပါတယ်။ သူတို့မှာလည်း Standard Property တွေ Method တွေရှိနေပါတယ်။

ဥပမာ Number တစ်ခုကို ဒဿမကိန်းအရေအတွက် ပိုင်းဖြတ်လိုရင် `toFixed()` ဆိုတဲ့ Method ကို အသုံးပြုနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
let num = 3.14159

num.toFixed(3)     // 3.142
```

`toFixed()` အတွက် Argument အနေနဲ့ 3 ကိုပေးလိုက်လို့ ဒဿမကိန်းသုံးလုံး ဖြတ်ယူပေးတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။ ဒီလို Dot Operator နဲ့ အသုံးပြုလို့ရနေတယ်ဆိုတာ Number ကိုယ်တိုင်က Object တစ်ခုဖြစ်နေလို့ပါ။ တနည်းအားဖြင့် Object ကဲ့သို့ သုံးလို့ရအောင် JavaScript ကလုပ်ထားပေးလို့ပါ။ ဒီ Object တွေထဲကမှ မှတ်သားသင့်တဲ့ အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်တွေ ပါဝင်နေတဲ့ String အကြောင်းကို ရွေးထုတ်ပြီးတော့ ပြောချင်ပါတယ်။

Array Object တစ်ခုတည်ဆောက်ဖို့ Array Object Constructor နဲ့ Array Literal ဆိုပြီး နှစ်မျိုးရှိသလိုပဲ String Object တစ်ခုတည်ဆောက်ဖို့အတွက် String Constructor နဲ့ String Literal (သို့မဟုတ်) Template Literal တို့ကို အသုံးပြုနိုင်ပါတယ်။ Single Quote နဲ့ Double Quote ကို String Literal အနေနဲ့ အသုံးပြုရပြီး Back Tick ကိုတော့ Template Literal အနေနဲ့ အသုံးပြုရတာပါ။

JavaScript

```
let name = "Bob"
let greet = `Hello ${name}`
let welcome = new String("Welcome")

name.length          // 3
welcome.length       // 7
'Hello'.length       // 5
```

နမူနာမှာ String ကြေညာပုံ အမျိုးမျိုးကိုပေးထားပါတယ်။ ကြေညာပုံထက်ပိုပြီး သတိပြုရမှာကတော့ String တွေမှာ `length` လို့ခေါ်တဲ့ Property ရှိနေခြင်းပဲဖြစ်ပါတယ်။ Object မို့လို့ Property ရှိနေတာပါ။ `length` တင်မကပါဘူး Index လည်း ရှိပါသေးတယ်။ ဒီလိုပါ –

JavaScript

```
let name = "Alice"

name[2]              // i
name.charAt(0)       // A
```

`name` String ရဲ့ Index 2 ဟာ `i` ဖြစ်တယ်ဆိုတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။ ဒါ့အပြင် `charAt()` လို့ခေါ်တဲ့ Standard String Method နဲ့ Index 0 မှာရှိတဲ့တန်ဖိုးကို ရယူနိုင်တာကိုလည်း တွေ့ရမှာပါ။ တစ်ခြားအသုံးဝင်တဲ့ Method ကတော့ `toUpperCase()`, `toLowerCase()`, `trim()`, `substr()` နဲ့ `split()` တို့ပဲ ဖြစ်ပါတယ်။

- **`toUpperCase()`** Method ကို စာလုံး အကြီးတွေပြောင်းဖို့ သုံးပါတယ်။
- **`toLowerCase()`** ကိုတော့ စာလုံး အသေးတွေပြောင်းဖို့ သုံးပါတယ်။
- **`trim()`** ကို ရှေ့ဆုံးနဲ့ နောက်ဆုံးက Space တွေရှင်းဖို့သုံးပါတယ်။
- **`substr()`** ကိုတော့ လိုချင်တဲ့အပိုင်း ဖြတ်ယူဖို့သုံးပါတယ်။
- **`split()`** ကိုတော့ အပိုင်းပိုင်း ခွဲထုတ်ဖို့ သုံးပါတယ်။

JavaScript

```
let name = "Alice"

name.toUpperCase()          // ALICE
name.toLowerCase()          // alice
name.substr(0, 3)           // Ali
```

`substr()` အတွက် Argument နှစ်ခုပေးတဲ့အခါ ပထမတစ်ခုက Index ဖြစ်ပြီး ဒုတိယတစ်ခုက စာလုံးအရေအတွက်ဖြစ်ပါတယ်။ ဒါကြောင့်နမူနာမှာ ထိပ်ဆုံးကနေစပြီး သုံးလုံးဖော်ပြနေတာပါ။

JavaScript

```
let text = "  Hello World "

text.trim()                         // Hello World
```

`trim()` Method က String ရဲ့ ရှေ့နဲ့နောက်က Space တွေကို ဖယ်ထုတ်ပေးသွားတာပါ။

```
let text = "Hello World"

text.split(" ")  // ["Hello", "World"]

text.split()

// ["H", "e", "l", "l", "o", " ", "W", "o", "r", "l", "d"]
```

`split()` Method အတွက် Space တစ်ခုကို Argument အနေနဲ့ ပေးလိုက်တဲ့အခါ Space နဲ့ ပိုင်းဖြတ်ပြီး ပိုင်းဖြတ်လို့ရလာတဲ့ ရလဒ်ကို Array တစ်ခုအနေနဲ့ ပြန်ပေးတာကို တွေ့ရမှာပါ။ Argument မပေးတဲ့အခါမှာတော့ ရှိသမျှ Character အားလုံးကို တစ်လုံးစီခွဲထုတ်လိုက်ပြီး ရလဒ်ကို Array အနေနဲ့ ပြန်ပေးမှာပဲ ဖြစ်ပါတယ်။

String ရဲ့ `split()` နဲ့ Array ရဲ့ `join()` တွေကို တွဲပြီးသုံးကြလေ့ရှိပါတယ်။ Array တွေကို String ပြောင်းချင်တဲ့အခါနဲ့ String တွေကို Array ပြောင်းချင်တဲ့အခါမျိုးမှာ အလွန်အသုံးဝင်ပါတယ်။

String Method တွေထဲမှာ `search()` နဲ့ `replace()` ဆိုတဲ့ Method တွေလည်း အသုံးဝင်ပါသေးတယ်။ ဒါပေမယ့် ဒီ Method တွေကို အသုံးပြုနိုင်ဖို့အတွက် Regular Expression ဆိုတဲ့နည်းပညာတစ်မျိုးကို သိထားဖို့လိုပါတယ်။ Regular Expression ဟာ အနည်းငယ် ခက်ခဲတဲ့ အကြောင်းအရာဖြစ်လို့ ထည့်သွင်းဖော်ပြဖို့ မသင့်တော်သေးဘူးလို့ ယူဆပါတယ်။ ဒါကြောင့် နောင်တစ်ချိန်မှာ ဆက်လက်လေ့လာရမယ့် စာရင်းထဲမှာ ထည့်ထားလိုက်စေချင်ပါတယ်။

Standard Object တွေကို လိုအပ်သလို ပြင်ဆင်ဖြည့်စွက်လို့လည်း ရပါသေးတယ်။ ဒီနည်းကိုသုံးဖို့ အားမပေးပေမယ့် ရှိမှန်းသိအောင်တော့ ထည့်ပြောပြချင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
String.prototype.greet = function() {
    return "Hello, World"
}
```

ဒါဟာ Standard String Object မှာ `greet()` Method တစ်ခု ထပ်တိုးလိုက်တာပါ။ ဒါကြောင့် String အားလုံးမှာ ဒီ Method ရှိသွားပါပြီ။ အခုလို စမ်းကြည့်နိုင်ပါတယ်။

JavaScript

```
let str = "Some String"

str.greet()     // Hello, World
```

တော်တော်လေး လန့်ဖို့ကောင်းတဲ့ လုပ်ဆောင်ချက်ပါ။ မဆင်မခြင်အသုံးပြုသူရဲ့ လက်ထဲမှာ Standard Object တွေအကုန် ကမောက်ကမ ဖြစ်ကုန်နိုင်စေလို့ပါ။ ဒါကြောင့် လက်တွေ့သုံးဖို့ထက် ဒါမျိုးရှိတယ်ဆိုတာကို သိအောင်သာ ထည့်ပြောလိုက်တာပါ။

Creating Objects

Array တစ်ခုတည်ဆောက်ဖို့အတွက် Array Constructor ကိုသုံးလို့ရသလို Array Literal ဖြစ်တဲ့ လေးထောင့်ကွင်းကိုလည်း သုံးနိုင်သလိုပါပဲ Object တစ်ခုတည်ဆောက်ဖို့အတွက်လည်း Object Constructor ကိုသုံးနိုင်သလို Object Literal Operator အဖြစ် တွန့်ကွင်းအဖွင့်အပိတ်ကို သုံးနိုင်ပါတယ်။

JavaScript

```
let cat = { }
```

ဒါဟာ `cat` ဆိုတဲ့ Object အလွတ်တစ်ခုကို တည်ဆောက်လိုက်တာပါ။ Object တစ်ခုဖြစ်တဲ့အတွက် Standard Property တွေ Method တွေရှိပေမယ့် ကိုယ်တိုင်သတ်မှတ်ပေးထားတဲ့ Property တွေ Method တွေတော့ မရှိသေးပါဘူး။

JavaScript

```
let cat = { color: "Yellow", legs: 4 }
```

ဒီတစ်ခါတော့ `cat` Object တစ်ခုတည်ဆောက်စဉ်မှာ `color` နဲ့ `legs` လို့ခေါ်တဲ့ Property (၂) ခုကို တစ်ခါထဲ ထည့်သွင်းသတ်မှတ်လိုက်တာပါ။ ရေးထုံးအရ Property နဲ့ Value ကို Full-Colon သင်္ကေတလေးနဲ့ ပိုင်းခြားပြီးတော့ ရေးပေးရပါတယ်။ Property အမည်တွေကို နမူနာမှာ ပေးထားသလို ဒီအတိုင်း ရေးလို့ရသလို String တစ်ခုကဲ့သို့ Quote အဖွင့်အပိတ်နဲ့ ရေးလို့လည်း ရပါတယ်။ Property တွေများလို့ ဖတ်ရတာ ခက်မှာစိုးရင် အခုလိုလိုင်းတွေခွဲပြီးတော့လည်း ရေးလို့ရပါတယ်။

JavaScript

```
let cat = {
    color: "Yellow",
    name: "Shwe War",
    legs: 4,
}
```

Property တစ်ခုနဲ့တစ်ခုကို Comma နဲ့ခြားပေးရတဲ့အခါ ထူးခြားချက်အနေနဲ့ ပေးထားတဲ့ နမူနာမှာ တွေ့မြင်ရသလို နောက်ဆုံးမှာ Comma တစ်ခု အပိုပါလို့ ရပါတယ်။ Trailing Comma လို့ခေါ်ပါတယ်။ အရင်က ဒီလိုနောက်ဆုံးမှာ တစ်ခုပိုနေလို့ မရပါဘူး။ အခုနောက်ပိုင်းမှ ရလာတာပါ။ ဒီလိုနောက်ဆုံးမှာ အပိုတစ်ခုပါလို့ရတဲ့အတွက် လိုအပ်လို့ Property တွေ ထပ်တိုးတဲ့အခါ၊ တစ်နေရာမှာ မတော်တဆ Comma မေ့ကျန်ခဲ့တယ် ဆိုတဲ့အမှားမျိုးတွေ လျော့နည်းသွားစေပါတယ်။

ဒီ Object ကို Property (၃) ခုပါတဲ့ Object တစ်ခုလို့ မြင်ကြည့်လို့ရသလို Index (၃) ခုပါတဲ့ Array တစ်ခုလို့လည်း မြင်ကြည့်လို့ရပါတယ်။ Index တွေက Number မဟုတ်တော့ဘဲ String ဖြစ်သွားတာပဲ ကွာသွားတာပါ။ Index ကို Key လို့လည်း ခေါ်ကြပါသေးတယ်။ ဒီလိုပါ –

color	name	legs
Yellow	Shwe War	4

တစ်ကယ်တော့ မြင်ကြည့်ယုံတင် မဟုတ်ပါဘူး။ လက်တွေ့အသုံးပြုတဲ့အခါမှာလည်း နှစ်မျိုးသုံးလို့ရပါတယ်။ Array Index ထောက်သလို လေးထောင့်ကွင်းအဖွင့်အပိတ်နဲ့ ရေးသားအသုံးပြုနိုင်သလို Object Property ကိုရယူသလို Dot Operator နဲ့လည်း ရေးသားအသုံးပြုနိုင်ပါတယ်။

JavaScript

```
cat.legs              // 4
cat["color"]          // Yellow
```

Property တန်ဖိုးတွေ ပြင်ဆင်တာ၊ ဖြည့်စွက်တာတွေကိုလည်း နှစ်မျိုးရေးလို့ရတာပါပဲ။

JavaScript

```
let bird = { color: "Green", legs: 2 }

bird.name = "Shwe Gal"
bird["color"] = "Blue"
```

Dot Operator ကိုသုံးပြီး `name` Property တစ်ခုထပ်တိုးထားသလို လေးထောင့်ကွင်းကိုသုံးပြီးတော့လည်း `color` Index ကတန်ဖိုးကို ပြင်ပြထားပါတယ်။ ဒါကြောင့် အခုနေ `bird` Object ရဲ့ဖွဲ့စည်းပုံက အခုလိုပုံစံ ဖြစ်နေမှာပါ။

color	name	legs
Blue	Shwe Gal	2

Object ဆိုတာ ရှုပ်ထွေးတဲ့ သဘောသဘာဝတစ်ခုပါ။ အဲ့ဒီလို ရှုပ်ထွေးတဲ့ သဘောသဘာဝကို ပိုပြီးရိုးရှင်းတဲ့ Array တစ်ခုကဲ့သို့ မြင်ကြည့်ရတာ ပိုလွယ်မယ်ထင်လို့ နှိုင်းယှဉ်ဖော်ပြခဲ့တာပါ။ ရေးထုံးကလည်း ရှိနေတယ်လေ။ လက်တွေ့ရေးသားတဲ့အခါ Dot Operator နဲ့ သုံးရတဲ့ Object Property ရေးထုံးက ရေးရတာ ပိုမြန်သလို ဖတ်ရတာလည်း ပိုရှင်းပါလိမ့်မယ်။

Array မှာ Spread နဲ့ Destructuring လုပ်ဆောင်ချက် ရှိသလိုပဲ Object တွေမှာလည်း ရှိပါတယ်။ ရေးနည်းက အတူတူပါပဲ။ ဒါကြောင့် အကုန်တော့ ပြန်မပြောတော့ပါဘူး။ ဥပမာလေးတစ်ခုနှစ်ခုပဲ ထည့်ပေးလိုက်ပါတော့မယ်။

JavaScript

```
let user = { name: "Bob", age: 22 }

function greet(name, age) {
    return `Hello ${name}, you are ${age} years old`
}

greet(...user)    // Hello Bob, you are 22 years old

let { name, age } = user

// name → Bob
// age → 22
```

Object မှာ Method တွေသတ်မှတ်ဖို့အတွက် Function Expression တွေကိုပဲ သုံးနိုင်ပါတယ်။ သက်ဆိုင်ရာ Index မှာ ရိုးရိုးတန်ဖိုး ပေးမယ့်အစား Function တစ်ခုပေးလိုက်ရတာပါ။ ဒီလိုပါ –

JavaScript

```
let user = {
    name: "Bob"
    hello: function() {
        return `Hello, I'm ${this.name}`
    }
}

user.name            // Bob
user.name = Alice
user.hello()         // Hello, I'm Alice
```

ဒါဟာ `name` Property နဲ့ `hello` Method တို့ပါဝင်တဲ့ Object တစ်ခုဖြစ်ပါတယ်။ `hello` Method ဟာ `name` Property ကို အသုံးပြုထားတာ သတိပြုပါ။ Object ရဲ့ ကိုယ်ပိုင် Property တွေ Method တွေကို အသုံးပြုလိုရင် `this` Keyword ကို အသုံးပြုရပါတယ်။

Object Method တွေကို အတိုကောက်ရေးတဲ့နည်း ရှိပါသေးတယ်။ အပေါ်ကနမူနာကို အခုလိုရေးရင်လည်း ရပါတယ်။ အတူတူပါပဲ –

JavaScript

```
let user = {
    name: "Bob"
    hello() {
        return `Hello, I'm ${this.name}`
    }
}
```

Object တွေဖန်တီးတဲ့အခါ ဒီလိုအခြေအနေမျိုးကိုလည်း မကြာခဏ ကြုံရနိုင်ပါတယ်။

JavaScript

```
let name = "Alice"
let age = 22

let user = {
    name: name,
    age: age
}
```

Property အမည်နဲ့ အသုံးပြုလိုတဲ့ Variable တူနေတာပါ။ အဲ့ဒီလို တူနေတဲ့အခါ နှစ်ခါရေးစရာ မလိုပါဘူး၊ အခုလို အတိုကောက်ရေးလိုက်လို့ရပါတယ်။

JavaScript

```
let name = "Alice"
let age = 22
let user = { name, age }
```

ကျစ်ကျစ်လစ်လစ် တိုတိုတုတ်တုတ် ဖြစ်သွားတာပါ။ ဒီနည်းကို Property Shorthand လို့ ခေါ်ပါတယ်။

Object တစ်ခုတည်ဆောက်မှုနဲ့ ပက်သက်ပြီး သိသင့်တာလေးတွေ ကျန်ပါသေးတယ်။ အဲ့ဒီအကြောင်းတွေကို Object-Oriented Programming အခန်းရောက်တဲ့အခါ ဆက်လက်ဖော်ပြပါမယ်။ ဒီနေရာမှာ ထည့်သွင်းပြီးတော့ ပြောချင်တာကတော့ Object Array ဖြစ်ပါတယ်။ ပရိုဂရမ်းမင်း အကြောင်းလေ့လာတဲ့အခါ Data Structure လို့ ခေါ်တဲ့ အချက်အလက် စုဖွဲ့ပုံဆိုင်ရာ သဘောသဘာဝတွေကို ထည့်သွင်းလေ့လာကြရပါတယ်။ သီအိုရီအရ Arrays အပြင် Stacks, Queues, Linked List, Trees, Graphs စသဖြင့် Data Structure အမျိုးအစား အမျိုးမျိုးရှိပေမယ့် JavaScript မှာတော့ Object တွေ Array တွေကိုသာ လိုအပ်သလို ပေါင်းစပ်ပြီးတော့ အသုံးပြုကြပါတယ်။

ဥပမာ – လူတစ်ယောက်ရဲ့ အချက်အလက်တွေကို စုဖွဲ့ထားလိုတဲ့အခါ အခုလိုထားနိုင်ပါတယ်။

JavaScript

```
let person = {
    name: "Bob",
    age: 22,
    education: [
        "B.Sc.",
        "MBA",
    ]
}
```

Object နဲ့ Array ကိုပဲ ပေါင်းစပ် စုဖွဲ့လိုက်တာပါ။ အကယ်၍ လူတွေအများကြီးရဲ့ အချက်အလက်ကို စုဖွဲ့ထားလိုရင်တော့ အခုလိုဖြစ်နိုင်ပါတယ်။

Pseudocode

```
let people = [
    { name: "Alice", age: 21, gender: "Female" },
    { name: "Bob", age: 22, gender: "Male" },
    ...
    { name: "Zack", age: 24, gender: "Male" },
]
```

Object တွေကိုပဲ Array တစ်ခုနဲ့စုဖွဲ့ထားပေးလိုက်တာပါ။ ဒီနည်းနဲ့ JavaScript မှာ အချက်အလက်တွေကို စုဖွဲ့စီမံကြလေ့ရှိတယ်ဆိုတာကို မှတ်သားထားနိုင်ပါတယ်။ အထက်မှာ ပြောခဲ့တဲ့ Array Methods တွေနဲ့ ပေါင်းစပ်လိုက်တဲ့အခါ အတော်လေးပြည့်စုံတဲ့ အချက်အလက်စီမံမှုစနစ်ကို ရရှိနိုင်ပါတယ်။ ဥပမာ - `people` ကနေ အမည်တွေချင်း ထုတ်ယူချင်ရင် အခုလို ယူလို့ရနိုင်ပါတယ်။

JavaScript

```
people.map( p => p.name )     // [ "Alice", "Bob", … , "Zack" ]
```

အကယ်၍ Male တွေချည်းပဲ ရွေးထုတ်ချင်တယ်ဆိုရင် အခုလို ရွေးယူနိုင်ပါတယ်။

JavaScript

```
people.filter( p => p.gender === "Male")

/* [
     {name: "Bob", age: 22, gender: "Male" },
     … ,
     {name: "Zack", age: 24, gender: "Female" }
] */
```

Object ကို Data အနေနဲ့ အသုံးပြုတဲ့နေရာမှာ JSON လို့ခေါ်တဲ့ သဘောသဘာဝ တစ်ခုလည်း ရှိပါသေးတယ်။ JavaScript Object Notation ရဲ့ အတိုကောက်ဖြစ်ပါတယ်။ ကနေ့ခေတ်မှာ အဓိက Data Format အနေနဲ့ အသုံးပြုကြပါတယ်။ JSON ကို အနှစ်ချုပ်အားဖြင့် အချက်အလက်တွေကို JavaScript Object ကဲ့သို့ စုဖွဲ့ထားရှိခြင်းလို့ မှတ်ယူနိုင်ပါတယ်။ အားသာချက်ကတော့ JavaScript သာမက ကနေ့ခေတ် အသုံးများတဲ့ Programming Language အားလုံးလိုလိုက JSON Format နဲ့ သိမ်းဆည်းထားတဲ့ အချက်အလက်တွေကို နားလည် အလုပ်လုပ်နိုင်ခြင်းပဲ ဖြစ်ပါတယ်။

JSON နဲ့ JavaScript Object တို့ဟာ ရေးထုံးအားဖြင့် အတူတူပါပဲ။ ကွဲလွဲချက်အနေနဲ့ (၂) ချက်မှတ်ရပါမယ်။ ပထမတစ်ချက်ကတော့ JSON မှာ Index/Key တွေကို Double Quote အဖွင့်အပိတ်နဲ့ ရေးပေးရပါတယ်။ ဒီလိုပါ -

JSON

```
{
    "name": "Bob",
    "age": 22,
    "gender": "Male"
}
```

ရေးထုံးတူပေမယ့် Index/Key တွေအားလုံး Double Quote အဖွင့်အပိတ် အတွင်းမှာ ရေးထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ပြီးတော့၊ နောက်ဆုံးက Trailing Comma ပါလို့မရတာကိုလည်း သတိပြုပါ။

ဒုတိယကွဲလွဲချက်အနေနဲ့၊ တန်ဖိုးအဖြစ် JSON က လက်ခံတဲ့ အမျိုးအစား (၆) မျိုးပဲ ရှိပါတယ်။ String, Number, Boolean, null, Array, Object တို့ဖြစ်ပါတယ်။ ဒီခြောက်မျိုးကလွဲရင် တစ်ခြားအရာတွေကို အသုံးပြုခွင့်မရှိတဲ့အတွက် JavaScript Object မှာလို Function တွေ Method တွေ ပါဝင်လို့ရမှာ မဟုတ်ပါဘူး။ Comment တွေကိုတောင် ထည့်သွင်းရေးသားလို့ ရမှာ မဟုတ်ပါဘူး။

ဒီကွဲလွဲချက် (၂) ခုကိုမှတ်ထားလိုက်ရင်တော့ ရပါပြီ။ ကျန်ရေးထုံးက အတူတူပါပဲ။ ဒါပေမယ့် လက်တွေ့မှာ ကိုယ်တိုင်အတိအကျ မှန်အောင်လိုက်ရေးနေဖို့တော့ မလိုအပ်ပါဘူး။ ရိုးရိုး JavaScript Object တွေကို JSON String ဖြစ်အောင်ပြောင်းလိုရင် `JSON.stringify()` လို့ခေါ်တဲ့ Standard Method ကို အသုံးပြုနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
let person = { name: "Alice", age: 21 }

JSON.stringify(person)     // { "name": "Alice", "age": 21 }
```

ရိုးရိုး JavaScript Object ဟာ JSON String ဖြစ်သွားပါပြီ။ JSON String လို့ပြောတာကို သတိပြုပါ။ Object မဟုတ်တော့ပါဘူး။ String ဖြစ်သွားတာပါ။ ရိုးရိုး String မဟုတ်ဘဲ JSON Format နဲ့ ဖွဲ့စည်းထားတဲ့ String တစ်ခုပါ။

အလားတူပဲ JSON String တွေကို JavaScript Object ပြောင်းလို့လည်း ရပါတယ်။ `JSON.parse()` ကို သုံးရပါတယ်။

```JavaScript
let json = '{ "name": "Alice", "age": 21 }'

JSON.parse(json)     // Object → { name: "Alice", age: 21 }
```

JSON ဆိုတာ အသုံးဝင်တဲ့ JavaScript Standard Object တွေထဲက တစ်ခုအပါအဝင်ဖြစ်ပါတယ်။ JavaScript မှာ တစ်ခြား အသုံးဝင်တဲ့ Standard Object တွေ အများကြီးကျန်ပါသေးတယ်။ ရက်စွဲ/အချိန်တွေကို စီမံနိုင်ဖို့အတွက် Date Object နဲ့ Absolute, Square Root, Power, Round, Min, Max စတဲ့ တွက်ချက်မှုတွေအတွက် Math Object တို့လို့ အခြေခံကျတဲ့ Object တွေကနေ Promise တို့ Proxy တို့လို အဆင့်မြင့်ကုဒ် Architecture အတွက် အသုံးဝင်တဲ့ Object တွေထိ ရှိနေပါတယ်။ ဒါတွေကို တစ်ခါထဲ အကုန်စုံလင်အောင် ဖော်ပြဖို့မလွယ်သလို၊ စာဖတ်သူအတွက်လည်း တစ်ခါထဲ အကုန်မှတ်သားလေ့လာဖို့ မလွယ်ပါဘူး။ တစ်ကယ်တော့ JavaScript Object တွေရဲ့ သဘောသဘာဝကို အခုလောက် သိထားပြီဆိုရင် လက်တွေ့လိုအပ်လာတော့မှသာ သက်ဆိုင်ရာ Reference တွေမှာ ကိုးကားကြည့်သွားလိုက်ရင် အဆင်ပြေသွားမှာပါ။

JavaScript Object Reference ကို ဒီနေရာမှာ အပြည့်အစုံလေ့လာနိုင်ပါတယ်။

– https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

အခန်း (၆) – Control Flows & Loops

ပုံမှန်အားဖြင့် ပရိုဂရမ်တစ်ခုမှာပါဝင်တဲ့ Statement တွေဟာ ကြိုတင်သတ်မှတ်ထားတဲ့ အစီအစဉ်အတိုင်း ပုံသေအလုပ်လုပ်သွားတာပါ။ အဲ့ဒီလိုအစီအစဉ်အတိုင်း ပုံသေမဟုတ်ဘဲ အခြေအနေပေါ် မူတည်ပြီး အလုပ်လုပ်စေချင်ရင်လည်း ရပါတယ်။ ဒီအတွက် Conditional Statement ကို အသုံးပြုနိုင်ပါတယ်။

Conditional Statement တွေကို `if` Keyword အသုံးပြုပြီး ရေးရတဲ့အတွက် If Statement လို့လည်း ခေါ်ပါတယ်။ If Statement ဟာ Expression တစ်ခုကို လက်ခံပြီးတော့ အဲ့ဒီ Expression ရဲ့ ရလဒ် Condition ပေါ်မူတည်ပြီး အလုပ်လုပ်ပေးပါတယ်။ ရလဒ်ဟာ true သို့မဟုတ် false ဆိုတဲ့ အခြေအနေပေါ်မူတည်ပြီး နှစ်မျိုးထဲက တစ်မျိုးကို လုပ်ပေးနိုင်ဆိုတဲ့ သဘောပါပဲ။ ရေးထုံးက ဒီလိုပါ –

Pseudocode

```
if(condition)
    // if true, do this statement
else
    // if false, do this statements
```

`if` Keyword နောက်မှာ ဝိုက်ကွင်းအဖွင့်အပိတ်နဲ့ Condition လိုက်ရပြီး Condition ရဲ့ရလဒ် true ဖြစ်တော့မှသာ ဆက်လိုက်လာတဲ့ Statement ကို အလုပ်လုပ်မှာဖြစ်ပြီး Condition က false ဖြစ်ခဲ့ရင်တော့ `else` Keyword နောက်ကလိုက်တဲ့ Statement ကို အလုပ်လုပ်မှာပါ။ `else` Statement မလိုအပ်ရင် မထည့်ဘဲ နေလို့ရပါတယ်။

Statement တစ်ခုထက် ပိုမယ်ဆိုရင်တော့ Statement Block နဲ့ ပေးနိုင်ပါတယ်။ ဒီလိုပါ –

Pseudocode

```
if(condition) {
    // if true
    // do these
    // statements
} else {
    // if false
    // do these
    // statements
}
```

မြင်သာတာလေး တစ်ခုလောက် ဥပမာပေးချင်ပါတယ်။ Array တစ်ခုရဲ့အတွင်းက 5 တွေကိုလိုက်ရှာပြီး 10 ပေါင်းပေးတဲ့ ကုဒ်လေးတစ်ခု ရေးကြည့်ကြပါမယ်။ ဒီလိုပါ –

JavaScript

```
let nums = [ 1, 12, 5, 4, 9, 5 ]

let result = nums.map(function(n) {
    if(n === 5) n += 10

    return n
})

// result → [ 1, 12, 15, 4, 9, 15 ]
```

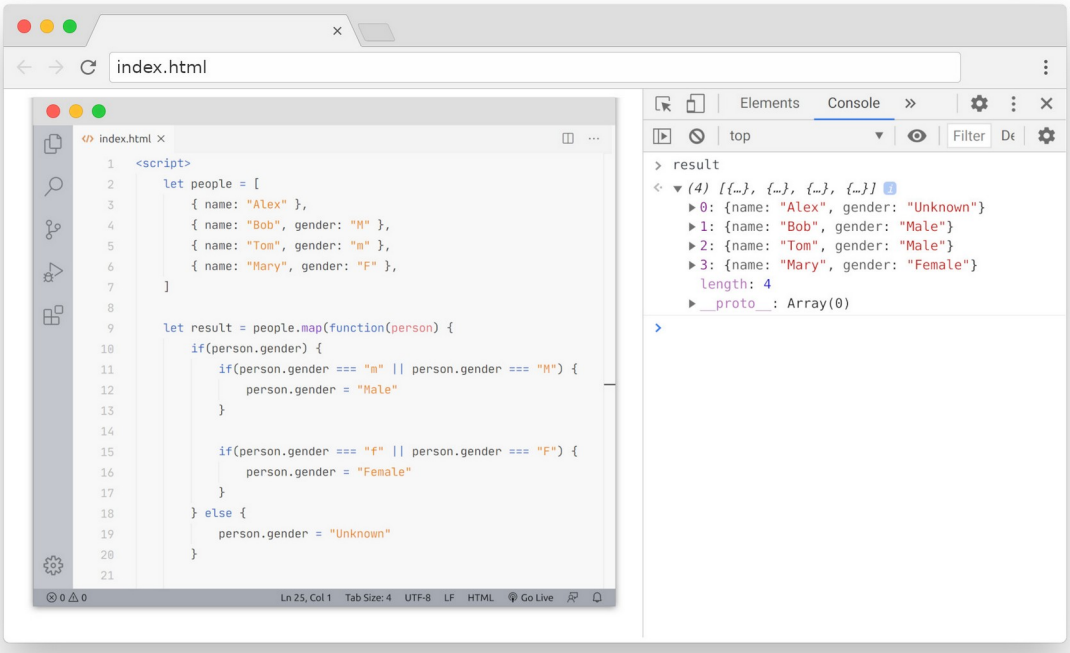
`if` Statement နဲ့ `n === 5` Condition သုံးပြီး စစ်လိုက်တာပါ။ မှန်တယ်ဆိုတော့မှ n တန်ဖိုးကို 10 ပေါင်းထည့်မှာဖြစ်ပြီး မမှန်ရင် ဒီအတိုင်းထားလိုက်မှာပါ။ `else` Statement မပါပါဘူး။ ဒါကြောင့် ရလဒ် အနေနဲ့ Array ထဲက 5 တွေကို 10 ပေါင်းပေးထားတဲ့ ရလဒ်ကို ရရှိမှာပဲ ဖြစ်ပါတယ်။

`if` Statement တွေကို Nested အထပ်ထပ်ရေးရတာမျိုးလည်း ရှိနိုင်ပါတယ်။ ဥပမာ – `people` Array ထဲမှာ `gender` မရှိရင် Unknown လို့ပေးမယ်။ `gender` က f သို့မဟုတ် F ဆိုရင် Female လို့ ပေးမယ်။ m သို့မဟုတ် M ဆိုရင် Male လို့ပေးမယ် ဆိုကြပါစို့။ ဒီလိုရေးလို့ရနိုင်ပါတယ်။

JavaScript

```
let people = [
    { name: "Alex" },
    { name: "Bob", gender: "M" },
    { name: "Tom", gender: "m" },
    { name: "Mary", gender: "F" },
]

let result = people.map(function(person) {
    if(person.gender) {
        if(person.gender === "m" || person.gender === "M") {
            person.gender = "Male"
        }

        if(person.gender === "f" || person.gender === "F") {
            person.gender = "Female"
        }
    } else {
        person.gender = "Unknown"
    }

    return person
})

/* result → [
    { name: "Alex", gender: "Unknown" },
    { name: "Bob", gender: "Male" },
    { name: "Tom", gender: "Male" },
    { name: "Mary", gender: "Female" },
] */
```

ဒီလောက်များတဲ့ ကုဒ်ကို Console ထဲမှာ ရေးစမ်းရတာ တော်တော်လက်ဝင်ပါလိမ့်မယ်။ ဒါကြောင့် HTML Document တစ်ခုတည်ဆောက်ပြီး ကုဒ်ကို `<script>` Element နဲ့ ရေးလို့ရပါတယ်။



ပုံကိုကြည့်လိုက်ရင် ကုဒ်တွေကို HTML Document တစ်ခုထဲမှာ `<script>` Element နဲ့ရေးထားပြီး အဲ့ဒီ Document ကို Browser နဲ့ဖွင့်ထားတာဖြစ်ပါတယ်။ ပြီးတော့မှ Console မှာ result ကို ထုတ်ကြည့်ထားတာပါ။ Document ထဲမှာ ရေးထားတဲ့ ကုဒ်တွေဟာ Console မှာလည်း သက်ဝင်ပါတယ်။ နောက်ပိုင်းကုဒ်တွေ ရေးရတာများလို့ Console မှာတိုက်ရိုက်ရေးရခက်ရင် ဒီနည်းဆက် ရေးစမ်းကြည့်နိုင်ပါတယ်။

Condition ဟာ Boolean true/false မဟုတ်ရင်လည်း ရပါတယ်။ Truthy ဖြစ်တဲ့တန်ဖိုးဆိုရင် true အနေနဲ့ အလုပ်လုပ်သွားမှာဖြစ်ပြီး Falsy ဖြစ်တဲ့ တန်ဖိုးဆိုရင် false အနေနဲ့ အလုပ်လုပ်ပေးသွားမှာပါ။ နမူနာမှာ `person.gender` ကို ဘာနှိုင်းယှဉ်မှုမှ မပါဘဲ Condition အနေနဲ့ ပေးထားပါတယ်။ အကယ်၍ `person.gender` မရှိရင် `undefined` ဖြစ်မှာမို့လို့ Falsy ဖြစ်ပါတယ်။ ဒါကြောင့် `else` Statement အလုပ်လုပ်သွားပြီး `person.gender` = Unknown ဖြစ်သွားမှာပါ။ `person.gender` ရှိတယ်ဆိုရင် ဘာပဲရှိရှိ Truthy မို့လို့ သူနဲ့သက်ဆိုင်တဲ့ Block ကို ဆက်အလုပ်လုပ်သွားမှာပါ။ အထဲမှာ နောက်ထပ် ထပ်ဆင့် စစ်ထားပါသေးတယ်။ Comparison Operator နဲ့ Logical Operator ကို ပေါင်းပြီးတော့ သုံးထားပါတယ်။ ကုဒ်အရ m သို့မဟုတ် M ဖြစ်ခဲ့ရင် Male လို့ သတ်မှတ်ပေးသွားမှာဖြစ်ပြီး f သို့မဟုတ် F ဖြစ်ခဲ့ရင် Female လို့ သတ်မှတ်သွားမှာဖြစ်ပါတယ်။ ဒီ ကုဒ်ကိုနောက်တစ်မျိုးလည်း ရေးလို့ရပါသေးတယ်။ ဒီလိုပါ –

JavaScript

```
let result = people.map(function(person) {
    if(person.gender === "m" || person.gender === "M") {
        person.gender = "Male"
    } else if(person.gender === "f" || person.gender === "F") {
        person.gender = "Female"
    } else {
        person.gender = "Unknown"
    }

    return person
})
```

ဒီကုဒ်ကတော့ နည်းနည်းပိုရှင်းသွားပါတယ်။ `else` နောက်မှာ ထပ်ဆင့် `if` Statement ကို ကပ်ရေးပေးလိုက်လို့ m သို့မဟုတ် M ဆိုရင် Male, f သို့မဟုတ် F ဆိုရင် Female၊ တစ်ခုမှ မဟုတ်ရင် Unknown ဆိုတဲ့ အဓိပ္ပါယ်ပေါက်သွားပါတယ်။ ဒီနေရာမှာ စကားပြောလာတာကတော့ ကိုယ့်ရဲ့ Logical Thinking Skill ပါပဲ။ ကြောင်းကျိုးဆက်စပ် တွေးခြင်တတ်ဖို့ လိုလာပါတယ်။ Logical Thinking ကောင်းသူက အလွယ်လေး မြင်သွားနိုင်ပေမယ့် Logical Thinking အားနည်းသူအတွက်တော့ နားလည်ရခက်နေနိုင်ပါတယ်။ ကြိုးစားပြီး မြင်အောင်လေ့လာကြည့်ပါ။ ပရိုဂရမ်မာကောင်းတစ်ယောက်မှာ ရှိရမယ့် အရည်အချင်းတွေထဲက အရေးအကြီးဆုံး တစ်ခုကို ရွေးထုတ်ပြပါဆိုရင် Logical Thinking လို့ ပြောရပါလိမ့်မယ်။

နောက်ထပ်အခြေနေပေါ် မူတည်အလုပ်လုပ်စေလိုတဲ့အခါ သုံးနိုင်တဲ့နည်းကတော့ `switch` Statement ဖြစ်ပါတယ်။ ရေးထုံးအရ `switch` Keyword နောက်မှာ ဝိုက်ကွင်းအဖွင့်အပိတ်နဲ့ Express ကိုပေးရပါတယ်။ သူကတော့ true/false စစ်တာမဟုတ်တော့ဘဲ ရလာတဲ့ ရလဒ်နဲ့ ကိုက်ညီတဲ့ `case` Statement ကို သွားပြီးအလုပ်လုပ်ပေးမှာ ဖြစ်ပါတယ်။ ဒီလိုပါ –

Pseudocode

```
switch(1 + 2) {
    case 1: // skip this statement
    case 2: // skip this statement
    case 3: // do this statement
    case 4: // do this statement
}
```

နမူနာအရ 1 + 2 Expression ကိုပေးလိုက်လို့ ရလဒ်က 3 ဖြစ်ပါတယ်။ ဒါကြောင့် ပေးထားတဲ့ `case` တွေထဲက 1 နဲ့ 2 ကို အလုပ်မလုပ်ဘဲ 3 ကိုအလုပ်လုပ်သွားမှာ ဖြစ်ပါတယ်။ ထူးခြားချက်ကတော့ ကိုက်ညီတဲ့ `case` ကို ရောက်ပြီးရင် ဆက်တိုက်အကုန်ဆက်လုပ်သွားမှာမို့လို့ 4 ကိုပါအလုပ်လုပ်သွားမှာ ဖြစ်ပါတယ်။ အဲ့ဒီလို မလုပ်စေချင်ရင် `break` Statement ကို သုံးပေးရပါတယ်။

Pseudocode

```
switch(1 + 2) {
    case 1:
        // skip this statement
        break
    case 2:
        // skip this statement
        break
    case 3:
        // do this statement
        break
    case 4:
        // do not reach this statement
        break
}
```

`case` တိုင်းမှာ `break` တွေလိုက်ထည့်ပေးလိုက်တာပါ။ ဒါကြောင့် `case` တစ်ခုနဲ့ကိုက်ညီလို့ အလုပ်လုပ်ပြီးတာနဲ့ `break` နဲ့ Block ထဲကနေ ပြန်ထွက်သွားမှာဖြစ်လို့ သူ့အောက်က မဆိုင်တဲ့ `case` တွေကို ဆက်လုပ်တော့မှာ မဟုတ်ပါဘူး။

အကယ်၍ `case` တွေတစ်ခုမှ Expression ရဲ့ ရလဒ်နဲ့မကိုက်ရင် ဘယ်လိုလုပ်မလဲ။ `default` Statement ကိုသုံးနိုင်ပါတယ်။ `case` တွေတစ်ခုမှ မကိုက်ရင် `switch` က `default` ကို အလုပ်လုပ်ပေးသွားမှာပါ။

JavaScript

```
switch (2 - 3) {
    case 1: // skip this statement
    case 2: // skip this statement
    default: // do this statement
}
```

နမူနာအရ Expression ရလဒ်ဟာ -1 ဖြစ်နေလို့ case 1 နဲ့ 2 နှစ်ခုလုံးနဲ့ မကိုက်ပါဘူး။ ဒါကြောင့် `default` ကို အလုပ်လုပ်သွားမှာ ဖြစ်ပါတယ်။ စောစောက အပေါ်မှာပေးခဲ့တဲ့ နမူနာကို ဒီ `switch` Statement နဲ့လည်း ရေးလို့ရနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
let result = people.map(function(person) {
    switch(person.gender) {
        case "m":
        case "M":
            person.gender = "Male"
            break
        case "f":
        case "F":
            person.gender = "Female"
            break
        default:
            person.gender = "Unknown"
    }

    return person
})
```

အကယ်၍ `person.gender` ဟာ m ဖြစ်နေရင် `case "m"` ကို အလုပ်လုပ်သွားပါလိမ့်မယ်။ `case "m"` မှာ နမူနာအရ ဘာ Statement မှမရှိပါဘူး။ ဒါပေမယ့် `break` လည်းမပါလို့ ဆက်အလုပ်လုပ်သွားမှာ ဖြစ်လို့ `case "M"` ကိုရောက်သွားပါလိမ့်မယ်။ တန်ဖိုးကို Male လို့သတ်မှတ်ပြီးနောက်မှာတော့ break ရှိနေလို့ ရပ်သွားမှာ ဖြစ်ပါတယ်။ f နဲ့ F ကိုလည်း အလားတူပဲ ရေးထားပါတယ်။ တစ်ခုမှ ကိုက်ညီမှု မရှိဘူးဆိုတော့မှ နောက်ဆုံးမှာ `default` ကို အလုပ်လုပ်သွားမှာပဲ ဖြစ်ပါတယ်။

Condition Expression

If Statement တို့ Switch Statement တို့နဲ့ အခြေအနေပေါ်မူတည်ပြီး အလုပ်လုပ်တဲ့ Statement တွေ သတ်မှတ်နိုင်သလို Expression တွေကိုလည်း အခြေအနေပေါ်မူတည်ပြီး အလုပ်လုပ်တဲ့ Conditional Expression ဖြစ်အောင် ရေးလို့ပါတယ်။ Ternary Operator လို့ခေါ်တဲ့ Operator ကိုသုံးရပါတယ်။

ဒီလိုပါ –

Pseudocode

```
Condition ? Do-this-if-true : Do-this-if-false
```

Condition ရဲ့နောက်မှာ Question Mark သင်္ကေတလိုက်ရပြီး Condition က true ဖြစ်ရင် Question Mark နောက်က Expression ကို အလုပ်လုပ်မှာပါ။ သူ့နောက်က Full-Colon ဆက်လိုက်ရပြီး Condition က false ဖြစ်ခဲ့ရင်တော့ Full-Colon နောက်က Expression ကို လုပ်မှာဖြစ်ပါတယ်။

JavaScript

```
let user = { name: "Bob", age: 17 }
let status = user.age >= 18 ? "Authorized" : "Unauthorized"
```

နမူနာအရ `status` အတွက်တန်ဖိုး "Unauthorized" ဖြစ်မှာပါ။ Condition က `user.age >= 18` ဆိုတော့ false ဖြစ်နေပါတယ်။ `user.age` က 17 ဖြစ်နေလို့ပါ။ ဒါကြောင့် Question Mark နောက်က အလုပ်ကို မလုပ်တော့ဘဲ Full-colon နောက်ကအလုပ်ကို လုပ်သွားတဲ့အတွက် `status` ရဲ့တန်ဖိုးအဖြစ် Unauthorized ကို Assign လုပ်သွားမှာပဲ ဖြစ်ပါတယ်။

Loops

တစ်ချို့တူညီတဲ့ Statement တွေကို အကြိမ်ကြိမ်အလုပ်လုပ်ဖို့ လိုအပ်တာမျိုးတွေ ရှိနိုင်ပါတယ်။ Statement တစ်ခုကို (၁၀) ကြိမ်အလုပ်လုပ်စေချင်လို့ (၁၀) ခါရေးစရာ မလိုပါဘူး။ Loop ရေးထုံးတွေရှိပါတယ်။ ဒီ Statement ကို (၁၀) ခါလုပ်လိုက်ပါ လို့ ညွှန်ကြားလိုက်လို့ ရနိုင်ပါတယ်။ JavaScript မှာ Loop ရေးထုံးအမျိုးမျိုး ရှိပါတယ်။ ပထမဆုံးတစ်ခုအနေနဲ့ `while` Loop ကို ကြည့်ချင်ပါတယ်။

`while` Loop အတွက် ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာ Condition ကို ပေးရပါတယ်။ အဲ့ဒီ Condition က true ဖြစ်နေသရွေ့ Statement တွေကို အကြိမ်ကြိမ် အလုပ်လုပ်ပေးမှာ ဖြစ်ပါတယ်။

JavaScript

```
let count = 0

while(count < 3) {
    console.log(count)
    count++
}

// 0
// 1
// 2
```

နမူနာအရ `count` တန်ဖိုးဟာ မူလ 0 ဖြစ်ပါတယ်။ `while` Loop အတွက် Condition က `count < 3` ဖြစ်တဲ့အတွက် မှန်ပါတယ်။ ဒါကြောင့် သတ်မှတ်ထားတဲ့ Statement တွေကို အလုပ်လုပ်သွားမှာပါ။ အဲ့ဒီလို အလုပ်လုပ်တဲ့အခါ `count` တန်ဖိုးကို 1 တိုးထားလို့ တစ်ကြိမ် အလုပ်လုပ်ပြီးတိုင်း count တန်ဖိုးလိုက်တိုး သွားမှာပါ။ သုံးကြိမ်အလုပ်လုပ်ပြီးလို့ `count` တန်ဖိုး 3 ဖြစ်သွားတဲ့အခါ `count < 3` Condition က မမှန်တော့ပါဘူး `count` ဟာ 3 ထက်ငယ်လားလို့ စစ်ထားတာပါ။ မငယ်တော့ဘူးလေ။ ဒါကြောင့် ဆက်အလုပ် မလုပ်တော့ဘဲ ရပ်သွားမှာ ဖြစ်ပါတယ်။

`do-while` Loop လည်းရှိပါသေးတယ်။ `while` Loop နဲ့ သဘောသဘာဝ ဆင်တူပါပဲ။

JavaScript

```
let count = 0

do {
    console.log(count)
    count++
} while(count < 3)

// 0
// 1
// 2
```

`do` Keyword ရဲ့နောက်မှာ ကုဒ် Block လိုက်ပြီးတော့မှ `while` Condition က နောက်ဆုံးကလိုက်တာပါ။ အခုချိန်မှာ ရလဒ်ပြောင်းမှာ မဟုတ်ပါဘူး။ ရေးထုံးကွဲသွားပေမယ့် ရိုးရိုး `while` Loop နဲ့ တူညီတဲ့ရလဒ်ကိုပဲ ရမှာပါ။ သဘောသဘာဝ ထူးခြားချက်ကိုသာ သတိပြုရမှာပါ။ `while` Loop က Condition အ

ရင်စစ်ပြီးမှ အလုပ်လုပ်ပါ။ `do-while` Loop ကတော့ အလုပ်လုပ်ပြီးမှ Condition စစ်ပါတယ်။ ဒါကြောင့် `do-while` Loop မှာ Condition မှန်သည်ဖြစ်စေ မမှန်သည်ဖြစ်စေ တစ်ကြိမ်တော့ ကြိမ်းသေ အလုပ်လုပ်မှာ ဖြစ်ပါတယ်။ ပထမဆုံးအကြိမ် Condition မစစ်ခင်လုပ်လိုက်တဲ့ အလုပ်တစ်ခု ရှိနေမှာ မို့လို့ ပါ။ ဒီလိုပါ။

JavaScript

```
let count = 5

do {
    console.log(count)
    count++
} while(count < 3)

// 5
```

`count` တန်ဖိုး 5 ဖြစ်တဲ့အတွက် `count < 3` Condition ကမှားနေပါတယ်။ ဒါပေမယ့် တစ်ကြိမ် အလုပ်လုပ်သွားလို့ 5 ဆိုတဲ့ဖော်ပြချက်တစ်ခု ရှိနေမှာပဲ ဖြစ်ပါတယ်။

နောက်ထပ် Loop ရေးထုံးတစ်ခုကတော့ `for` Loop ဖြစ်ပါတယ်။ `for` Keyword နောက်မှာ ဝိုက်ကွင်း အဖွင့်အပိတ်နဲ့အတူ Expression (၃) ခုလိုက်ရပါတယ်။

- ပထမဆုံး Expression ကို Loop မစခင်တစ်ကြိမ် အလုပ်လုပ်ပါတယ်။
- ဒုတိယ Expression ကို Condition အနေနဲ့စစ်ပြီး true ဖြစ်နေသ၍ သတ်မှတ်ထားတဲ့ Statement တွေကို ထပ်ခါထပ်ခါ အလုပ်လုပ်မှာ ဖြစ်ပါတယ်။
- တတိယ Expression ကို တစ်ကြိမ်ပြီးတိုင်း တစ်ခါအလုပ်လုပ်ပါတယ်။

ဒါကြောင့် စောစောက `while` တို့ `do-while` တို့နဲ့ရေးထားတဲ့ Loop နဲ့ တူညီတဲ့ရလဒ်မျိုးရဖို့အတွက် အခုလို ရေးနိုင်ပါတယ်။

JavaScript

```
for(let count=0; count < 3; count++) {
    console.log(count)
}

// 0
// 1
// 2
```

Expression တစ်ခုနဲ့တစ်ခုကို Semicolon နဲ့ ပိုင်းခြားပေးရပါတယ်။ ရှေ့ဆုံးမှာ `let count=0` လို့ ပြောထားတဲ့အတွက် `count` Variable တစ်ခုကြေညာပြီး တန်ဖိုးကို 0 လို့သတ်မှတ်ပေးလိုက်မှာပါ။ တစ်ကြိမ်ပဲ အလုပ်လုပ်မယ့် Expression ဖြစ်ပါတယ်။ ဒုတိယ Expression ကို Condition အနေနဲ့ စစ်ကြည့်လိုက်တဲ့အခါ `count < 3` က true ဖြစ်နေတဲ့အတွက် သတ်မှတ်ထားတဲ့ Statement ကို အလုပ်လုပ်လိုက်ပါတယ်။ ပြီးတဲ့အခါ `for` ရဲ့ တတိယ Expression ကိုအလုပ်လုပ်လိုက်တဲ့အတွက် `count` တန်ဖိုး 1 တိုးသွားပါတယ်။ Condition စစ်ကြည့်တဲ့အခါ `count < 3` က true ဖြစ်နေသေးတဲ့အတွက် ဆက်အလုပ်လုပ်ပါတယ်။

`while` Loop မှာ Variable ကြေညာတဲ့ကိစ္စတွေ၊ တစ်ကြိမ်အလုပ်လုပ်ပြီးတိုင်း 1 တိုးတဲ့ကိစ္စတွေကို ကိုယ့်အစီအစဉ်နဲ့ကိုယ် ရေးပေးရပါတယ်။ `for` Loop မှာတို့ အဲ့ဒီလို ပထမဆုံးအကြိမ် ကြေညာတဲ့အလုပ်နဲ့ တစ်ကြိမ်ပြီးတိုင်း 1 တို့တဲ့အလုပ်တို့အတွက် ရေးထုံးအရ သတ်မှတ်ထားပြီးဖြစ်နေလို့ သူသတ်မှတ်ထားတဲ့အတိုင်း တစ်ခါထဲ ရေးလို့ရသွားပါတယ်။ ဆိုလိုတာက `while` Loop မှာ 1 တိုးပေးဖို့ မေ့သွားလို့ Loop က ဘယ်တော့မှ Infinite Loop ဖြစ်အောင် ရေးမိသွားတယ်ဆိုတဲ့ အမှားမျိုး အစပိုင်းမှာ ကြုံရနိုင်ပါတယ်။ `for` Loop မှာတော့ အဲ့ဒီလိုအမှားမျိုး ဖြစ်နိုင်ခြေ နည်းသွားပါတယ်။

လက်တွေ့မှာ ကိုယ့်လိုအပ်ချက်နဲ့ ကိုက်ညီတဲ့ Loop အမျိုးအစားကို သုံးနိုင်ပါတယ်။ တစ်ကြိမ်ပြီးရင် 1 တိုးတာတော့ အသုံးများတဲ့ ထုံးစံလိုဖြစ်နေပေမယ့် အမြဲတမ်း 1 ပဲတိုးရမယ်ဆိုတာမျိုး မရှိပါဘူး။ ကိုယ့်လိုအပ်ချက်နဲ့ ကိုက်ညီအောင် ရေးလို့ရပါတယ်။

Loop တွေနဲ့အတူ တွဲသုံးလေ့ရှိတဲ့ Statement နှစ်ခု ရှိပါသေးတယ်။ `break` နဲ့ `continue` ပါ။ `break` ကို အကြောင်းအမျိုးမျိုးကြောင့် Loop ကို ရပ်လိုက်စေလိုတဲ့အခါ သုံးနိုင်ပါတယ်။ `continue` ကိုတော့ Loop လုပ်ရင် တစ်ချို့အဆင့်တွေ ကျော်ပြီး ဆက်လုပ်သွားစေချင်တဲ့အခါ သုံးကြပါတယ်။

JavaScript

```
let nums = [11, 22, -1 , 44]

for(let i=0; i < nums.length; i++) {
    if(nums[i] < 0) break
    console.log(nums[i])
}

// 11
// 22
```

နမူနာကုဒ်အရ စစချင်း Variable i ရဲ့တန်ဖိုး 0 ဖြစ်ပြီး i ရဲ့တန်ဖိုးက `nums.length` ထက် ငယ်နေသ၍ အလုပ်လုပ်မှာပါ။ `nums` Array မှာ Index (၄) ခုရှိတဲ့အတွက် `nums.length` တန်ဖိုး 4 ဖြစ်မှာပါ။ ပုံမှန်အတိုင်းဆိုရင် Index 0 ကနေ Index 3 ထိ တန်ဖိုးတွေကို တစ်ခုပြီးတစ်ခု အစအဆုံး အကုန်ဖော်ပြသွားမှာ ဖြစ်ပါတယ်။

ဒါပေမယ့် `if` Condition နဲ့ အကယ်၍လက်ရှိ Index တန်ဖိုး 0 ထက်ငယ်ရင် `break` နဲ့ထွက်လိုက်ဖို့ ပြောထားတဲ့အတွက် ကျန်တဲ့အလုပ်တွေ ဆက်မလုပ်တော့ဘဲ -1 တန်ဖိုးရှိနေတဲ့ Index အရောက်မှာ ရပ်သွားတာပဲဖြစ်ပါတယ်။

JavaScript

```
let nums = [11, 22, -1 , 44]

for(let i=0; i < nums.length; i++) {
    if(nums[i] < 0) continue
    console.log(nums[i])
}

// 11
// 22
// 44
```

ဒီတစ်ခါတော့ `break` မလုပ်တော့ပါဘူး။ `continue` လုပ်ထားပါတယ်။ ဒါကြောင့် 0 ထက်ငယ်တဲ့ -1 တန်ဖိုးရှိနေတဲ့အဆင့်ကို တစ်ဆင့်ကျော်လိုက်ပြီး ကျန်အလုပ်တွေကို ပြီးဆုံးတဲ့အထိ ဆက်လုပ်သွားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

Looping Objects & Arrays

Array တွေစီမံဖို့အတွက် map, filter, reduce တို့လို Method တွေရှိပြီးဖြစ်ပါတယ်။ ဒါပေမယ့် အဲ့ဒီ Method တွေက Objects တွေအတွက်သုံးလို့ မရပါဘူး။ Object တွေကို Loop လုပ်ဖို့အတွက် `for-in` Loop ကိုသုံးနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
let user = { name: "Bob", age: 22, gender: "Male" }

for(p in user) {
    console.log(`${p} is ${user[p]}`)
}

// name is Bob
// age is 22
// gender is Male
```

Object ထဲမှာ ရှိသမျှအကုန် အစကနေအဆုံး Loop လုပ်ပေးသွားမှာပါ။ ဘယ်ကစမယ်၊ ဘယ်မှဆုံးမယ်၊ တစ်တိုးမယ်တွေ ပြောစရာမလိုပါဘူး။ p နေရာမှာ နှစ်သက်ရာ Variable အမည်ပေးနိုင်ပါတယ်။ လက်ရှိရောက်ရှိနေတဲ့ Property က p ထဲမှာ ရှိနေမှာပါ။ ဒါကြောင့် p ကို ဖော်ပြစေတဲ့အခါ Property ကို ဖော်ပြပြီး Object ရဲ့ p Index ကိုဖော်ပြစေတဲ့အခါ သက်ဆိုင်ရာတန်ဖိုးကို ရရှိခြင်းဖြစ်ပါတယ်။

`for-in` Loop ကို ရိုးရိုး Array တွေအတွက်လည်း အသုံးပြုနိုင်ပါတယ်။ Array တွေအတွက် `for-of` Loop ကိုလည်း အသုံးပြုနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
let users = ["Alice", "Bob", "Tom", "Mary"]

for(u of users) {
    console.log(u)
}

// Alice
// Bob
// Tom
// Mary
```

Array ရဲ့အစကနေ အဆုံးထိ အလိုအလျောက် အလုပ်လုပ်သွားတာပါ။ ဘယ်ကစမယ်၊ ဘယ်မှာဆုံးမယ်၊ တစ်ကြိမ်မှာတစ်တိုးရမယ် စသည်ဖြင့် ပြောနေစရာ မလိုတော့လို့ အများကြီးပို့ရှင်းပါတယ်။ `for-of` Loop ကို Iterable Object တွေအတွက်ပဲသုံးလို့ရပါတယ်။ ရိုးရိုး Object တွေအတွက် မရပါဘူး။ Object-Oriented Programming မှာ Interface လို့ခေါ်တဲ့ သဘောသဘာဝ ရှိပါတယ်။ Protocol လို့လည်းခေါ်ကြပါတယ်။ အဲ့ဒီ Interface/Protocol က Object တည်ဆောက်တဲ့အခါ လိုက်နာရမည့် စည်းမျဉ်းတွေ သတ်မှတ်နိုင်တဲ့ နည်းစနစ်တစ်မျိုးပါ။ Iterable Object ဆိုတာ Iterable Protocol က သတ်မှတ်ထားတဲ့ သတ်မှတ်ချက်နဲ့အညီ တည်ဆောက်ထားတဲ့ Object တွေကိုပြောတာပါ။ လက်ရှိလေ့လာထားတဲ့ထဲက Array နဲ့ String တို့ဟာ Iterable Protocol သတ်မှတ်ချက်နဲ့အညီ ဖန်တီးပေးထားတဲ့ Object တွေဖြစ်ကြပါတယ်။ ဒါကြောင့် Array တွေ String တွေကို `for-of` နဲ့ Loop လုပ်နိုင်ပြီး ရိုးရိုး Object တွေအတွက် လိုအပ်ရင် `for-in` ကို အသုံးပြုရမှာပဲဖြစ်ပါတယ်။

ဒီစာအုပ်မှာ Interface/Protocol အဆင့်ထိ ထည့်သွင်းဖော်ပြနိုင်မှာ မဟုတ်ပေမယ့် Object-Oriented Programming ရဲ့အခြေခံ သဘောသဘာဝတွေကိုတော့ ဆက်လက်ထည့်သွင်း ဖော်ပြသွားမှာ ဖြစ်ပါတယ်။

အခန်း (၇) – Object-Oriented Programming – OOP

Object ဆိုတာဟာ ကိုယ်ပိုင် အချက်အလက် (Property) နဲ့ ကိုယ်ပိုင် လုပ်ဆောင်ချက် (Method) တွေ စု ဖွဲ့ပါဝင်တဲ့ အရာတစ်ခုပါ။ ဒီသဘောသဘာဝကို ရှေ့ပိုင်းမှာ တွေ့မြင်ခဲ့ကြပြီး ဖြစ်ပါတယ်။ Object Oriented Programming (OOP) ဆိုတာကတော့ ကုဒ်တွေရေးတဲ့အခါ ဒီ Object တွေပေါ်မှာ အခြေခံ အလုပ်လုပ်စေဖို့ စုဖွဲ့ရေးသားတဲ့ ရေးနည်းရေးဟန်ပဲ ဖြစ်ပါတယ်။

ကုဒ်တွေကို အစီအစဉ်အတိုင်း အလုပ်လုပ်စေတဲ့ ရေးနည်းရေးဟန်ကို Imperative Programming လို့ခေါ်ကြပါတယ်။ ကွန်ပျူတာကို တစ်ဆင့်ချင်းစီ အမိန့်ပေးစေခိုင်းတဲ့ ရေးဟန်မို့လို့ပါ။ Procedure တွေ Function တွေ ပါဝင်လာပြီဆိုရင်တော့ Procedural Programming လို့ခေါ်ကြပါတယ်။ Procedure တွေ Function တွေကို လိုအပ်သလို ပေါင်းစပ်အသုံးချခြင်းအားဖြင့် အလုပ်လုပ်စေတဲ့ ရေးဟန်ဖြစ်ပါတယ်။

Functional Programming ဆိုတာလည်း ရှိပါသေးတယ်။ Pure Function တွေကို စုဖွဲ့ပေါင်းစပ်ပြီး အလုပ်လုပ်စေခြင်းအားဖြင့် ပရိုဂရမ်ကို ဖွဲ့စည်းတည်ဆောက်ရတဲ့ ရေးဟန်ပါ။ Pure Function ဆိုတာ ရိုးရိုး Function နဲ့ မတူပါဘူး။ သူ့မှာ ထူးခြားတဲ့ ဝိသေသ (၂) ရပ်ရှိပါတယ်။ Function တစ်ခုကို ခေါ်ယူ လိုက်တဲ့အခါ ပေးလိုက်တဲ့ Argument တူလို့ ပြန်ရတဲ့ရလဒ် အမြဲတမ်းတူတယ်ဆိုရင် Pure Function လို့ ခေါ်ပါတယ်။ Function တစ်ခုကို အလုပ်လုပ်စေလိုက်လို့ မူလတန်ဖိုးတွေကို ပြောင်းလဲစေတယ်ဆိုရင် Side-Effect ရှိတယ်၊ ဘေးထွက်ရလဒ်ရှိတယ်လို့ ဆိုပါတယ်။ Pure Function ဆိုတာ အဲ့ဒီလို Side-Effect မရှိတဲ့ Function တွေပါ။

ရှေ့ပိုင်းမှာ လေ့လာခဲ့တဲ့ လုပ်ဆောင်ချက်တစ်ချို့ကို ဥပမာပြန်ကြည့်နိုင်ပါတယ်။ `push()`, `pop()`, `splice()` စတဲ့ Array Function တွေဟာ Impure Function တွေပါ။ ဒီ Function တွေကြောင့် မူလ Array မှာ တန်ဖိုးတွေ ပြောင်းလဲသွားမှာမို့လို့ ဖြစ်ပါတယ်။ ဒီလိုပြောင်းလဲနေတဲ့ အတွက်ကြောင့်ပဲ တစ်

ကြိမ်နဲ့တစ်ကြိမ် ပြန်ရမယ့် ရလဒ်လည်း တူမှာ မဟုတ်ပါဘူး။ `map()`, `filter()`, `reduce()` စ တဲ့ Function တွေကတော့ Pure Function တွေပါ။ ဒီ Function တွေက ရလဒ်အသစ်ကို ပြန်ပေးပါမယ်။ မူလ Array ကို ပြောင်းလဲစေခြင်း မရှိပါဘူး။ ဒါကြောင့် ဘယ်နှစ်ကြိမ် Run သည်ဖြစ်စေ တူညီတဲ့ရလဒ်ကိုပဲ အမြဲပြန်ရမှာပါ။ ဒါဟာသိရှိထားသင့်တဲ့ ဗဟုသုတဖြစ်ပါတယ်။ ဒီသဘောသဘာဝကို သိထားမှ မတူကွဲပြား နေတဲ့ အလုပ်လုပ်ပုံကို ပုံဖော်ကြည့်နိုင်မှာမို့လို့ပါ။

JavaScript ရဲ့ထူးခြားချက်က အဲ့ဒီလို Imperative, Procedural, OOP, Function စသည်ဖြင့် ရှိနေတဲ့ ရေးဟန်တွေထဲက နှစ်သက်ရာ ရေးဟန်ကို အသုံးပြုနိုင်ခြင်းဖြစ်ပါတယ်။ ဘယ်နည်းနဲ့ပဲ ရေးရမယ်လို့ ပုံသေကန့်သတ်ထားခြင်း မရှိလို့ ရေးသားသူက မိမိနှစ်သက်ရာ နည်းလမ်းကို အသုံးပြုနိုင်မှာဖြစ်သလို၊ ဆန္ဒရှိရင်လည်း ရေးဟန်အမျိုးမျိုးကို ပေါင်းစပ်အသုံးပြု ရေးသားနိုင်မှာပဲ ဖြစ်ပါတယ်။

Classes & Objects

ရှေ့ပိုင်းမှာ Object တည်ဆောက်ပုံနည်းလမ်းနှစ်မျိုး ပြောခဲ့ပါတယ်။ Object Constructor ကို အသုံးပြု တည်ဆောက်နိုင်သလို Object Literal ဖြစ်တဲ့ တွန့်ကွင်းအဖွင့်အပိတ်နဲ့လည်း တည်ဆောက်နိုင်ပါတယ်။ အခုဆက်လက် ဖော်ပြမယ့် နည်းလမ်းကတော့ Class ရေးထုံးကို အသုံးပြုတည်ဆောက်တဲ့ နည်းလမ်းပဲ ဖြစ်ပါတယ်။

Class ဆိုတာ Object Template လို့ ဆိုနိုင်ပါတယ်။ Object တည်ဆောက်လိုက်ရင် ပါဝင်ရမယ့် အချက်အလက်နဲ့ လုပ်ဆောင်ချက်တွေကို ကြိုတင်သတ်မှတ်ပေးထားနိုင်တဲ့ နည်းလမ်းတစ်မျိုးဖြစ်ပါ တယ်။ ဥပမာ ဒီလိုပါ –

JavaScript

```
class Car {
    color = "Red"
    wheels = 4
    drive() {
        console.log("This car is driving")
    }
}
```

`class` Keyword ရဲ့နောက်မှာ Class အမည်လိုက်ရပြီး သူ့နောက်ကနေ Property တွေ Method တွေ စု ဖွဲ့ပြီး လိုက်ရတာ ဖြစ်ပါတယ်။ ဒါဟာ သေချာလေ့လာကြည့်လိုက်ရင် ရိုးရိုး Object Literal အသုံးပြု ရေးသားနည်းနဲ့ သိပ်မကွာလှပါဘူး။

JavaScript

```
let car = {
    color: "Red",
    wheels: 4,
    drive() {
        console.log("The car is driving")
    }
}
```

အသုံးပြုတဲ့ သင်္ကေတတွေနဲ့ ရေးထုံးကွဲပြားမှုပိုင်းလေးတွေ ရှိနေပေမယ့် အတော်လေး ဆင်တူတာကို တွေ့ ရပါလိမ့်မယ်။ မတူတာကတော့ Car ဟာ Class တစ်ခုဖြစ်ပြီး Object မဟုတ်သေးပါဘူး။ လိုအပ်တဲ့ Object တွေကို ဒီ Class ကိုအသုံးပြု တည်ဆောက်ပေးရမှာပါ။ အခုလို တည်ဆောက်နိုင်ပါတယ်။

JavaScript

```
let toyota = new Car
```

`toyota` ဟာ Car Class ကို အသုံးပြုတည်ဆောက်လိုက်တဲ့ Object တစ်ခုဖြစ်သွားပါပြီ။ ဒါကြောင့် Car Class မှာ သတ်မှတ်ပေးထားတဲ့ Property တွေ Method တွေ ရှိနေမှာဖြစ်ပါတယ်။

JavaScript

```
toyota.wheels        // 4
toyota.drive()       // The car is driving
```

Class ကနေတစ်ဆင့် Object တည်ဆောက်ပြီးမှ အသုံးပြုရတဲ့ သဘောပါ။ အဲ့ဒီလို Object တည်ဆောက် စရာ မလိုဘဲ Class ကနေ တိုက်ရိုက် အသုံးပြုလိုရင် သုံးလို့ရတဲ့ ရေးနည်းရှိပါတယ်။ `static` Property နဲ့ `static` Method လို့ ခေါ်ပါတယ်။ ဒီလိုပါ။

JavaScript

```
class Calculator {
    static PI = 3.14

    static add(a, b) {
        return a + b
    }
}
```

ဒီလို `static` Property/Method တွေရှိနေမယ်ဆိုရင်တော့ Class ကနေတစ်ဆင့် တိုက်ရိုက်အသုံးပြုလို့ ရနိုင်သွားပါတယ်။

JavaScript

```
Calculator.PI             // 3.14
Calculator.add(3, 4)      // 7
```

ဒီသဘောသဘာဝကို Access-Control Modifier လို့ခေါ်ပြီး Property တွေ Method တွေကို ဘယ်နည်း ဘယ်ပုံ Access လုပ် အသုံးပြုခွင့်ပြုမလဲဆိုတာကို သတ်မှတ်နိုင်တဲ့ နည်းလမ်းတွေဖြစ်ပါတယ်။ အများ အားဖြင့် တွေ့ရလေ့ရှိတဲ့ Access-Control Modifier တွေကတော့ –

- Public
- Private
- Protected
- Static

– တို့ဖြစ်ပါတယ်။ JavaScript Class တစ်ခုအတွင်းမှာ ကြေညာလိုက်တဲ့ Property တွေ Method တွေဟာ Public သဘောသဘာဝ ရှိကြပါတယ်။ Object ကနေတစ်ဆင့် အပြည့်အဝ အသုံးပြုခွင့် ပေးထားတယ်ဆို တဲ့သဘောပါ။ Static ရဲ့သဘောသဘာဝကိုတော့ အခုပဲပြောခဲ့ပြီးပါပြီ။ Class အမည်ကနေ အသုံးပြုခွင့် ပေးထားပါတယ်။ Private ဆိုရင်တော့ Object ကနေတစ်ဆင့် အသုံးပြုခွင့် မပေးတော့ပါဘူး။ ရေးထားတဲ့ Class အတွင်းထဲမှာပဲ သုံးခွင့်ရှိပါတော့တယ်။

တစ်ခြား Language အများစုမှာ `public`, `private` စတဲ့ Keyword တွေကို အသုံးပြုပြီး Access-Control ကို သတ်မှတ်ကြပါတယ်။ ဒီလိုပါ –

Pseudocode

```
class Car {
    private hp = 150;
    public color = "Red";

    static info () {
        console.log(`Horse Power: ${this.hp}`);
    }
}
```

နမူနာက JavaScript ကုဒ်မဟုတ်ပါဘူး။ သဘောသဘာဝ ရှင်းပြချင်လို့ ရေးလိုက်တဲ့ Pseudocode ပါ။ နမူနာမှာ `hp` ကို Private လို့ပြောထားပါတယ်။ ဒါကြောင့် ပြင်ပနေ တိုက်ရိုက်ပြင်လို့ သုံးလို့ မရတော့ပါဘူး။ သတ်မှတ်ထားတဲ့ နည်းလမ်းကနေသာ အသုံးပြုခွင့်ရှိတော့မှာပါ။ ဥပမာ –

Pseudocode

```
let car = new Car();

car.hp = 200;        // Error: Cannot access private property
Car.info();          // Horse Power: 150
```

ဒီသဘောသဘာဝကို OOP မှာ Encapsulation လို့ခေါ်ပါတယ်။ Object တစ်ခုရဲ့ အသေးစိတ် အချက်အလက်တွေကို မဆိုင်သူ သိစရာမလိုဘူး၊ ထိစရာမလိုဘူး ဆိုတဲ့သဘောဖြစ်ပါတယ်။

JavaScript မှာတော့ `static` Keyword တစ်ခုပဲ ရှိပါတယ်။ `public` တွေ `private` တွေမရှိပါဘူး။ Keyword တွေအစား ရေးတဲ့အခါမှာ ဒီလိုရေးပေးရပါတယ်။

JavaScript

```
class Car {
    #hp = 150
    color = "Color"

    static info() {
        console.log(`Horse Power: ${this.#hp}`)
    }
}
```

ရိုးရိုးရှေးလိုက်တဲ့ `color` Property ဟာ Public ဖြစ်ပြီး ရှေ့ကနေ Hash သင်္ကေတလေး ခံရေးထားတဲ့ `#hp` Property က Private ဖြစ်သွားပါတယ်။ ဒီရေးထုံးဟာ JavaScript မှာ အခုမှစမ်းသပ်အဆင့်ပဲ ရှိပါသေးတယ်။ ဒါကြောင့် ရေးထုံးအရမှန်ပါတယ်။ တစ်ချို့ Update မဖြစ်တဲ့ Browser တွေမှာ စမ်းကြည့်လို့တော့ ရဦးမှာ မဟုတ်ပါဘူး။ စမ်းကြည့်လို့မရရင် Browser ကို Update လုပ်ပြီး ပြန်စမ်းကြည့်နိုင်ပါတယ်။

Protected ရဲ့ သဘောသဘာဝကို တော့ခဏနေမှ ဆက်ပြောပါမယ်။ အခုဖြည့်စွက်မှတ်သားသင့်တာကတော့ Constructor လို့ခေါ်တဲ့သဘောသဘာဝ ဖြစ်ပါတယ်။ Class တစ်ခုရေးသားတဲ့အခါ Constructor ထည့်ရေးလို့ရပါတယ်။ Constructor က အဲ့ဒီ Class ကို အသုံးပြုပြီး Object တည်ဆောက်စဉ်မှာ အလုပ်လုပ်ပေးမယ့် Method တစ်မျိုးပါ။ ဒီလိုရေးရပါတယ်။

JavaScript

```
class Dog {
    constructor(name) {
        this.name = name
    }

    run() {
        console.log(`${this.name} is running...`)
    }
}
```

Constructor Method ရဲ့အမည်ကို `constructor` လို့ပေးရပါတယ်။ နမူနာအရ Constructor Method မှာ Parameter တစ်ခုရှိလို့ Object တည်ဆောက်တဲ့အခါ Argument တစ်ခုပေးပြီး တည်ဆောက်ရတော့မှာပါ။ Constructor က လက်ခံရရှိတဲ့ Argument ကို name Property ထဲမှာ Assign လုပ်ပေးလိုက်မှာ ဖြစ်ပါတယ်။ အခုလို စမ်းကြည့်နိုင်ပါတယ်။

JavaScript

```
let dog1 = new Dog("Bobby")
let dog2 = new Dog("Rambo")

dog1.run()     // Bobby is running…
dog2.run()     // Rambo is running…
```

Class တွေတည်ဆောက်တဲ့အခါ အခြား Class တစ်ခုပေါ်မှာ အခြေခံပြီး တည်ဆောက်လို့ ရပါတယ်။ ဒီသဘောသဘာဝကို Inheritance လို့ခေါ်ပါတယ်။ အမွေဆက်ခံလိုက်တဲ့သဘောဖြစ်လို့ ဒီနည်းကိုသုံးလိုက်ရင် ပင်မ Class ရဲ့လုပ်ဆောင်ချက်တွေကို ဆက်ခံတဲ့ Class က အလိုလိုရသွားတာပါ။

JavaScript

```
class Animal {
    constructor(name) {
        this.name = name
    }

    run() {
        console.log(`${this.name} is running...`)
    }
}

class Dog extends Animal {
    bark() {
        console.log(`${this.name}: Woof.. woof..`)
    }
}
```

နမူနာအရ Dog Class က Animal Class ကို Inherit လုပ်လိုက်တာပါ။ `extends` Keyword ကို သုံးရပါတယ်။ ဒါကြောင့် Dog Class မှာ Animal Class ရဲ့လုပ်ဆောင်ချက်တွေကို ရရှိသွားပါတယ်။

JavaScript

```
let milo = new Dog("Milo")

milo.bark()     // Milo: Woof.. woof..
```

Dog Class မှာ Constructor တွေ `name` Property တွေမရှိပေမယ့် Animal Class ကနေဆက်ခံပြီး ရထား

လို့ အခုလို အသုံးပြုနိုင်ခြင်းဖြစ်ပါတယ်။ Class တွေအခုလို ဆက်ခံရေးသားတဲ့အခါ တစ်ကြိမ်မှာ Class တစ်ခုကိုပဲ Inherit လုပ်လို့ရပါတယ်။ Multiple-Inheritance ခေါ် Class နှစ်ခုသုံးခုကနေ တစ်ပြိုင်တည်း ဆက်ခံရေးသားလို့တော့ မရပါဘူး။ နောက်ထပ် ဥပမာလေး တစ်ခု ထပ်ပေးပါဦးမယ်။

JavaScript

```
class Cat extends Animal {
    constructor(name, color) {
        super(name)
        this.color = color
    }

    meow() {
        console.log(`${this.name}: Meow.. meow..`)
    }
}
```

ဒီနမူနာမှာတော့ Cat Class က Animal Class ကို ဆက်ခံရေးသားထားပါတယ်။ ထူးခြားချက်အနေနဲ့ Cat Class မှာလည်း Constructor ပါဝင်ပြီး Parameter နှစ်ခုရှိနေပါတယ်။ ဒါကြောင့် Cat Class ကို အသုံးပြုပြီး Object တည်ဆောက်ရင် သူ့ရဲ့ Constructor ကိုပဲ သုံးသွားတော့မှာပါ။ ပင်မ Animal Class ရဲ့ Constructor ကို သုံးမှာ မဟုတ်တော့ပါဘူး။

ပင်မ Animal Class ရဲ့ Constructor ကိုလည်း သုံးချင်တယ်၊ ဘယ်လိုလုပ်ရမလဲ။ Cat Class Constructor ထဲမှာ ရေးသားထားတဲ့ `super()` Method က ဒီအတွက်ဖြစ်ပါတယ်။ ပင်မ Class ရဲ့ Constructor ကို လှမ်းခေါ်ပေးပါတယ်။ ဒီနည်းနဲ့ လက်တွေ့အလုပ်လုပ်တာက Cat Class Constructor ဆိုပေမယ့် ပင်မ Animal Class Constructor ကိုလည်း လိုအပ်သလို အသုံးပြုလို့ ရသွားပါတယ်။

JavaScript

```
let cat = new Cat("Shwe War", "Yellow")

cat.meow()     // Shwe War: Mewo.. meow..
```

ဒီနေရာမှာ ကျန်နေတဲ့ Protected ရဲ့ သဘောသဘာဝကို ပြောလို့ရပါတယ်။ Inheritance လုပ်ပြီး ဆက်ခံလိုက်တဲ့အခါ ဆက်ခံတဲ့ Class ရဂို့မှာက Public Property နဲ့ Method တွေကိုသာ ရရှိမှာ ဖြစ်ပါတယ်။

Private Property တွေ Method တွေကိုတော့ ရရှိမှာ မဟုတ်ပါဘူး။ Protected ရဲ့ သဘောသဘာဝကတော့ ပြင်ပကနေ Access လုပ်ခွင့်မပေးဘူး၊ ဒါပေမယ့် Inherit လုပ် ဆက်ခံတဲ့သူကို သုံးခွင့်ပေးမယ် ဆိုတဲ့ သဘောပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် Access-Control Modifier မှာ Protected လို့ ပေးထားလိုက်ရင် Private လိုမျိုး ပြင်ပကနေ အသုံးပြုခွင့်မရှိပါဘူး၊ ဒါပေမယ့် သူကိုဆက်ခံထားတဲ့ Class တွေကတော့ အသုံးပြုခွင့်ရမှာ ဖြစ်ပါတယ်။ သိထားသင့်လို့ ထည့်ပြောတာပါ၊ JavaScript မှာ Protected ရေးထုံး မရှိပါဘူး။

အသုံးနည်းပေမယ့် ဖြည့်စွက်မှတ်သားသင့်တာတစ်ခု ရှိပါသေးတယ်။ Function တွေမှာ Function Expression ရှိသလိုပဲ Class တွေမှာလည်း Class Expression ရှိပါတယ်။ ဒါကြောင့် ဒီလိုရေးလို့ ရပါတယ်။

JavaScript

```
let Car = class {
    color = "Red"
    wheels = 4
    drive() {
        console.log("The car is driving")
    }
}
```

ဒါပေမယ့် Function တွေမှာ Hoisting/Lifting သဘောသဘာဝရှိပေမယ့် Class တွေမှာ မရှိပါဘူး။ ဒါကြောင့် Function တွေလို အရင်သုံးပြီး နောက်မှကြေညာလို့မရပါဘူး။ အသုံးမပြုခင် ကြိုတင်ကြေညာသတ်မှတ်ထားပေးဖို့ လိုအပ်ပါတယ်။

Object-Oriented Programming ဆိုတာ ရေးထုံးပိုင်းအရ ခက်ခဲလှတာ မဟုတ်ပေမယ့် သဘောသဘာဝပိုင်းကတော့ လေးနက်ကျယ်ပြန့်တဲ့ ဘာသာရပ်တစ်ခုဖြစ်ပါတယ်။ Multiple Inheritance မရဘူးလို့ ပြောထားတယ်။ ရတဲ့ Language တွေရော ရှိသလား။ ရခြင်း မရခြင်းရဲ့ အားသာချက် အားနည်းချက်တွေက ဘာတွေလဲ၊ စသဖြင့် ပြောမယ်ဆို ပြောစရာတွေ ကျန်ပါသေးတယ်။ နောက်ပြီးတော့ ပြီးခဲ့တဲ့အခန်းမှာ ပြောခဲ့တဲ့ Interface တွေ Protocol တွေက ဘယ်လိုနေရာမျိုးမှာ သုံးရတာလဲ။ Dynamically Typed နဲ့ Statically Typed သဘောသဘာဝက ဒီ Object တွေပေါ်မှာ ဘယ်လိုသက်ရောက်မှုရှိသလဲ၊ စသည်ဖြင့် ကျန်ပါသေးတယ်။ ပရောဂျက်ကြီးတွေမှာ ကုဒ် Architecture အတွက် သုံးကြတဲ့ Object-Oriented Design Patterns ဆိုတဲ့ သဘောသဘာဝတွေလည်း ရှိပါသေးတယ်။

ဒီစာအုပ်ကိုတော့ အခြေခံအဆင့်အတွက် ရည်ရွယ်တာဖြစ်လို့ ဒါတွေအကုန်ထည့်ပြောဖို့ စောနေပါသေးတယ်။ အခုဖော်ပြခဲ့တဲ့ အခြေခံရေးထုံးတွေကိုပဲ ကောင်းကောင်းနားလည် အသုံးချနိုင်အောင် လေ့လာထားစေချင်ပါတယ်။ ဒီအကြောင်းတွေကို တစ်ခြားသင့်တော်တဲ့ စာအုပ်တွေမှာ ထည့်သွင်းရေးသားပေးထားမှာဖြစ်လို့ အခြေခံတွေကြေညက်ပြီး နောက်တစ်ဆင့်တက်ဖို့ အချိန်ကျလာတဲ့အခါ ဆက်လက်လေ့လာနိုင်ပါတယ်။

အခန်း (၈) – Promises & async, await

JavaScript ဟာ Single-Threaded ပုံစံအလုပ်လုပ်တဲ့ နည်းပညာဖြစ်ပါတယ်။ ဒါကိုမျက်စိထဲမှာ မြင်သာအောင် ယာဉ်ကြောတစ်ခုပဲရှိတဲ့ ကားလမ်းတစ်ခုလို မြင်ကြည့်နိုင်ပါတယ်။ ပျော်ပွဲစားထွက်ဖို့ ကားသုံးစီးထွက်လာတယ်ဆိုရင် ယာဉ်ကြောတစ်ခုပဲရှိတဲ့အတွက် ရှေ့နောက်တန်းစီပြီးသွားကြရမှာပါ။ ရှေ့ကတစ်စီးအကြောင်းတစ်ခုခုကြောင့် နှေးသွားရင် နောက်က ကားတွေလည်း လိုက်နှေးသွားမှာပါပဲ။

Multi-Threaded စနစ်တွေကိုတော့ ဒီဥပမာနဲ့ နှိုင်းယှဉ်ပြောရမယ်ဆိုရင် ယာဉ်ကြောသုံးခုပါတဲ့ ကားလမ်းနဲ့တူပါတယ်။ ကားသုံးစီးက ပြိုင်တူယှဉ်ပြီးသွားလို့ ရပါတယ်။ ဒါကြောင့် တစ်စီးနှေးနေရင် စောင့်စရာမလိုဘဲ ကျန်တဲ့ကားတွေကို ဆက်သွားလို့ ရနိုင်ပါတယ်။

Multi-Threaded စနစ်တွေဟာ ကိုင်တွယ်တတ်မယ်ဆိုရင် ပိုမြန်ပါတယ်။ ဒါပေမယ့်သူ့မှာ အခက်အခဲတစ်ခုတော့ ရှိနေပါတယ်။ ကားသုံးစီးမှာ တစ်စီးက ပန်းကန်ခွက်တွေ ယူလာတယ်။ နောက်တစ်စီးက စားစရာတွေ ယူလာတယ်။ နောက်တစ်စီးက သောက်စရာတွေ ယူလာတယ် ဆိုကြပါစို့။ Single-Threaded စနစ်တွေမှာ အားလုံးက အတူတူပဲ သွားကြတာမို့လို့ စောင့်ရတဲ့အတွက် နှေးချင်နှေးမယ်၊ နောက်ဆုံးခရီးရောက်တဲ့အခါ အတူတကွ ပါလာတဲ့ ပန်းကန်ခွက်ယောက်၊ စားသောက်ဖွယ်ရာတွေ ချပြီး ပျော်ပွဲစားကြယုံပါပဲ။

Multi-Threaded စနစ်မှာ တစ်စီးကနှေးပြီး ကျန်ခဲ့လို့ ကျန်တဲ့နှစ်စီးက ရောက်နှင့်တဲ့အခါ ဘာလုပ်မလဲ စဉ်းစားစရာ ရှိလာပါတယ်။ သောက်စရာယူလာတဲ့ကား ကျန်ခဲ့ရင် ကိစ္စမရှိဘူး။ ပန်းကန်ခွက်နဲ့ စားစရာတွေ ကြိုပြင်ထားလိုက်ရင် မြန်သွားတာပေါ့။ ပန်းကန်ခွက်တွေယူလာတဲ့ကား ကျန်ခဲ့ရင်တော့ အဆင်မပြေတော့ပါဘူး။ သူရောက်အောင် စောင့်ရပါတော့မယ်။ နောက်ပြဿနာက ဘယ်လောက်စောင့်ရမှာလဲ။ ဘီးပေါက်လို့ ကြာနေတာမျိုးဆိုရင် ကြာတော့ကြာမယ်၊ အချိန်တန်ရင် ရောက်လာပါလိမ့်မယ်။ လမ်းမှားလို့ ကြာနေတာမျိုး ဆိုရင်တော့၊ စောင့်သာစောင့်နေတာ ရောက်မလာဘူးဆိုတာမျိုးတွေ ဖြစ်နိုင်ပါတယ်။

Multi-Thread Communication စနစ်တွေနဲ့ ကျကျနန စီမံတတ်ဖို့ လိုသွားပါတယ်။

JavaScript ကတော့ Single-Thread စနစ်ပါ။ Single-Thread မို့လို့ လမ်းကြောတစ်ခုထဲမှာပဲ တစ်ခုပြီးမှ တစ်ခုလုပ်ပါတယ်။ ဒါဆိုရင် အလုပ်တစ်ခုက ကြာနေရင် တစ်ကယ်ပဲ ကျန်တဲ့အလုပ်တွေ ရှေ့ဆက်လို့မရတော့ဘဲ စောင့်ရတော့မှာလား။ ဒီလိုမဖြစ်အောင်တော့ စီစဉ်ထားပါတယ်။ ဒီအတွက် Message Queue, Frame Stack, Event Loop စတဲ့ သီအိုရီပိုင်းဆိုင်ရာ သဘောသဘာဝတွေရှိပါတယ်။ အဲ့ဒါတွေကို မြင်လွယ်အောင် ဒီလိုလေးမြင်ကြည့်ပါ။

တန်းစီပြီးသွားနေတဲ့ ကားတန်းကို အစီအစဉ်အတိုင်း တစ်ခုပြီးတစ်ခု လုပ်ရမယ့် Messages Queue လို့ မြင်နိုင်ပါတယ်။ ကားတစ်စီးတိုင်းမှာ လိုအပ်တဲ့ပစ္စည်းတွေ ထပ်ထပ်ပြီးတော့ တင်ထားပါတယ်။ အဲ့ဒါကို Frames Stack လို့ မြင်နိုင်ပါတယ်။ ကားတွေထဲက ပစ္စည်းတွေကို ချတော့မယ်ဆိုရင် ကားတွေတန်းစီ၊ တစ်စီးဝင်၊ ပါတဲ့ပစ္စည်းတွေ တစ်ခုပြီး တစ်ခုချ၊ ပစ္စည်းတစ်ခုက ညပ်နေလို့ ချလို့မရဘူးဆိုရင် နောက်မှာသွားပြန်တန်းစီ၊ နောက်တစ်စီးဝင်၊ ပါတဲ့ပစ္စည်းတွေ တန်းစီချ၊ ပြီးရင်ထွက် စသဖြင့် အစီအစဉ်အတိုင်း အလုပ်လုပ်သွားတဲ့ သဘောကို Event Loop လို့ခေါ်ပါတယ်။ Event Loop က ချစရာပစ္စည်းမရှိတဲ့အခါ ဖယ်ခိုင်းပါတယ်။ ဖယ်တဲ့အခါ လုံးဝဖယ်လိုက်တာ ဖြစ်နိုင်သလို၊ ကျန်နေသေးလို့ နောက်ကပြန်ဝင်စီတာ မျိုးလည်း ဖြစ်နိုင်ပါတယ်။ ဒီနည်းနဲ့ အရမ်းကြာမယ့်အလုပ်တွေ မပြီးမချင်း နောက်အလုပ်တွေက စောင့်နေစရာ မလိုတော့ပါဘူး။ Single-Threaded ဆိုပေမယ့် မလိုအပ်ဘဲ မကြာတော့ပါဘူး။

ဥပမာ အခုလို စမ်းကြည့်နိုင်ပါတယ်။

JavaScript

```
console.log(1)
console.log(2)
setTimeout(() => console.log(3), 1000)
console.log(4)
```

`setTimeout()` Function ကို အချိန်ခဏာစောင့်ပြီးမှ လုပ်စေချင်တဲ့ အလုပ်တွေရှိရင် သုံးနိုင်ပါတယ်။ နမူနာအရ အလုပ်တစ်ခုကို `setTimeout()` နဲ့ 1000 မီလီစက္ကန့် (တစ်စက္ကန့်) စောင့်ပြီးမှ လုပ်ဖို့ ရေးထားတာပါ။ အစီအစဉ်အတိုင်းသာဆိုရင် -

```
// 1
// 2
(တစ်စက္ကန့်စောင့်)
// 3
// 4
```

– ဖြစ်ရမှာပါ။ ဒါပေမယ့် JavaScript ရဲ့ Event Loop သဘောသဘာဝကြောင့် အဲ့ဒီလို စောင့်နေရမယ့် အလုပ်ကို ကျော်ပြီး လုပ်ပေးလိုက်မှာ ဖြစ်ပါတယ်။ ဒါကြောင့် ရလဒ်က အခုလိုဖြစ်မှာပါ။

```
// 1
// 2
// 4
// 3
```

စောင့်ရမယ့် 3 ကိုမစောင့်ဘဲ နောက်ကိုပို့လိုက်ပြီး 4 ကို အရင်လုပ်လိုက်ပါတယ်။ ပြီးမှ အချိန်ကျလာတဲ့ အခါ 3 ကို လုပ်လိုက်လို့ အခုလိုရလဒ်မျိုးကို ရရှိခြင်းပဲ ဖြစ်ပါတယ်။

Promises

ပြီးခဲ့တဲ့နမူနာကိုကြည့်ရင် `setTimeout()` အတွက် Callback အနေနဲ့ Function Expression တစ်ခုကို ပေးခဲ့တာကိုတွေ့ရနိုင်ပါတယ်။ JavaScript မှာ အဲ့ဒီလို မစောင့်ဘဲ နောက်မှလုပ်စေချင်တဲ့ အလုပ်တွေရှိတဲ့ Callback တွေကို အသုံးများကြပါတယ်။ အခုနောက်ပိုင်းမှာတော့ Callback အစား Promise လို့ခေါ်တဲ့ နည်းပညာကို အစားထိုးပြီး သုံးလာကြပါတယ်။ ထုံးစံအတိုင်း ပြောမယ်ဆိုရင် အားသာချက် အားနည်းချက်တွေ ပြောစရာရှိပေမယ့်၊ ရေးနည်းကိုပဲ အဓိကထား ကြည့်ကြရအောင်ပါ။

JavaScript

```
function add1000() {
    let result = 0

    for(let i=1; i <= 1000; i++) {
        result += i
    }

    return result
}
```

နမူနာကိုလေ့လာကြည့်ပါ။ `add1000()` Funtion ဟာ 1 ကနေ 1000 ထိ ပေါင်းပေးမယ့် Function ပါ။ ဒီအတိုင်းသာဆိုရင် သူအလုပ်လုပ်လို့ မပြီးမချင်း တစ်ခြားအလုပ်က ဆက်လုပ်လို့ရမှာ မဟုတ်ပါဘူး။ စမ်းကြည့်လို့ရပါတယ်။

JavaScript

```
console.log("some processes")
console.log(add1000())
console.log("more processes")

// some processes
// 500500
// more processes
```

အစီအစဉ်အတိုင်းပဲ အလုပ်လုပ်သွားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် အခုလို Function တစ်ခု ထပ်ရေးလိုက်ပါမယ်။

JavaScript

```
function add1000later() {
    return new Promise( done => {
        done( add1000() )
    })
}
```

`add1000later()` Function က Promise Object တစ်ခုကို ပြန်ပေးပါတယ်။ နောက်ကျကျန်ခဲ့တဲ့ကားက၊ "ငါးမိနစ်အတွင်းရောက်မယ်၊ ကတိပေးပါတယ်ကွာ၊ မင်းတို့စနှင့်ကြပါ" လို့ ပြောလိုက်သလိုပါပဲ။ ဒါကြောင့် ကျန်အလုပ်တွေက ဒီ Function အလုပ်လုပ်တာကို မစောင့်တော့ဘဲ ကိုယ့်အလုပ်ကိုယ် ဆက်လုပ်သွားလို့ ရပါတယ်။ ဒီ Function က လုပ်စရာရှိတဲ့အလုပ် ပြီးသွားရင်သာ `done()` ကို Run ပေးပါတယ်။ ဒါကြောင့် ခေါ်ယူစဉ်မှာ `done()` ကို ထည့်ပေးဖို့တော့လိုပါတယ်။ ဒီလိုပါ –

JavaScript

```
console.log("some processes")

add1000later().then( result => console.log(result) )

console.log("more processes")
```

`then()` Method ကိုသုံးပြီး လုပ်စရာရှိတာလုပ်ပြီးရင် ဘာဆက်လုပ်ရမလဲ ပြောလိုက်တာပါ။ နမူနာအရ ရလဒ်ကို ဖော်ပြဖို့ပြောထားပါတယ်။ စမ်းကြည့်လိုက်ရင် အခုလိုရမှာပါ။

```
// some processes
// more processes
// 500500
```

လုပ်စရာရှိတာတွေ သူ့ဘာသာဆက်လုပ်သွားလို့ more processes ကိုနောက်မှ Run ထားပေမယ့် အရင်ရနေတာပါ။ ဒီနည်းနဲ့ Asynchronous ခေါ် ပြုံင်တူအလုပ်လုပ်နိုင်သော၊ တစ်ခုပြီးအောင်တစ်ခု စောင့်စရာမလိုသော ကုဒ်တွေကို JavaScript မှာ ရေးလို့ရပါတယ်။

`done()` ဆိုတာ မြင်သာအောင်သာ ပြောလိုက်တာပါ။ တစ်ကယ်အသုံးအနှုန်းအမှန်က `resolve()` လို့ ခေါ်ပါတယ်။ `reject()` လည်းရှိပါသေးတယ်။ ငါးမိနစ်အတွင်း ရောက်မယ်ပြောပြီး မရောက်ရင် ဘာလုပ်မှာလဲ။ ဒါလည်း စဉ်းစားစရာ ရှိသေးတယ် မဟုတ်လား။ လုပ်စရာရှိတာလုပ်မယ် သွားနှင့်ပါ ပြောပေမယ့် အဲ့ဒီအလုပ် မအောင်မြင်ဘူးဆိုတာ ဖြစ်နိုင်ပါတယ်။ ဒီလိုဖြစ်လာခဲ့ရင် `reject()` လုပ်လို့ရနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
function add1000later() {
    return new Promise( (resolve, reject) => {
        let result = add1000()

        if(result) resolve(result)
        else reject()
    })
}
```

`add1000()` Function ကမှန်လို့ ရလဒ်ပြန်ပေးနိုင်တဲ့အခါ `resolve()` ကို အလုပ်လုပ်ပြီး မှားနေလို့ ရလဒ်ပြန်မရတဲ့အခါ `reject()` ကိုအလုပ်လုပ်ထားပါတယ်။ ခေါ်သုံးတဲ့အခါ `then()` Method နဲ့ `resolve()` ကိုပေးနိုင်ပြီး `catch()` Method နဲ့ `reject()` ကို ပေးနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
add1000later()
    .then( result => console.log(result) )
    .catch( () => console.log("Error") )
```

ဖတ်လို့ကောင်းအောင် လိုင်းခွဲပြီးရေးထားပါတယ်။ တစ်ဆက်ထဲရေးလည်း ရပါတယ်။ နမူနာအရ အကယ်၍အလုပ်လုပ်တာ မအောင်မြင်ခဲ့ရင် `catch()` နဲ့ ပေးလိုက်တဲ့ `reject()` Function အလုပ်လုပ်သွားမှာဖြစ်လို့ Error ကို ပြန်ရမှာပဲ ဖြစ်ပါတယ်။ နားလည်သလိုလို၊ နားမလည်သလိုလို ဖြစ်နေမှာ အသေအချာပါပဲ။ ခက်ခဲတဲ့ အကြောင်းအရာတစ်ခုမို့လို့ပါ။ ကူးယူရေးစမ်းပြီး နည်းနည်းပါးပါး ပြင်ကြည့်လိုက်ရင်တော့ ပိုပြီးတော့ မြင်သွားပါလိမ့်မယ်။ အဲ့ဒါမှမရသေးရင်လည်း ခဏကျော်လိုက်ပြီး ကျန်တဲ့ အခန်းတွေ ဆက်လေ့လာလို့ရပါတယ်။ နောက်တော့မှ တစ်ခေါက် ပြန်လာလေ့လာပါ။

Promise တွေကို အသုံးပြုတဲ့အခါ ဒီလိုလည်းသုံးလို့ ရနိုင်ပါသေးတယ်။

JavaScript

```
add1000later()
    .then( result => result + 1000)
    .then( result => console.log(result) )
    .catch( () => console.log("Error") )

// 501500
```

ပထမ `then()` Method က ရလဒ်ကို `return` ပြန်ပေးထားလို့ အဲ့ဒီလို ပြန်ပေးတဲ့ရလဒ်ကို နောက်ထပ် `then()` Method နဲ့ဖမ်းပြီး ဆက်အလုပ်လုပ်လို့ ရနေတာပါ။ လက်တွေ့မှာ အတော်အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်တွေပါ။

async, await

ကုဒ်ရဲ့အလုပ်လုပ်ပုံကို မြင်ကြည့်နိုင်ဖို့ အရေးကြီးကြောင်း ခဏခဏ ပြောခဲ့ပါတယ်။ Promise ရေးထုံးနဲ့ ရေးထားတဲ့ကုဒ်ကို ပုံဖော်ပြီးမြင်ကြည့်နိုင်ဖို့ သိပ်ခက်နေတယ်ထင်ရင် စိတ်မပူပါနဲ့။ ကိုယ်တစ်ယောက်ထဲ မဟုတ်ပါဘူး။ Promise ကုဒ်က နားလည်ရခက်လွန်းတယ် ဆိုသူတွေ အများကြီးရှိနေပါတယ်။ ဒါကြောင့်လည်း async & await ဆိုတဲ့ နောက်ထပ် ရေးထုံးတစ်မျိုးကို ထပ်ပြီးတော့ တီထွင်ပေးထားပါသေးတယ်။

စောစောကရေးခဲ့တဲ့ `add1000later()` Function ကို အခုလိုပြန်ရေးလိုက်ပါမယ်။

```
async function add1000later() {
    let result = await add1000()
    console.log(result)
}
```

ရှေ့ဆုံးမှာ `async` Keyword ပါသွားလို့ `add1000later()` Function ဟာရိုးရိုး Function မဟုတ်တော့ပါဘူး။ စောင့်စရာမလိုဘဲ သူ့ဘာသာလုပ်စရာရှိတာ ဆက်လုပ်မှာမို့လို့ ချန်ထားခဲ့လို့ရတဲ့ Function ဖြစ်သွားပါပြီ။ အဲ့ဒီ Function ထဲမှာ `add1000()` Function ကို ခေါ်သုံးတဲ့အခါ `await` Keyword နဲ့ ခေါ်သုံးထားတာကို သတိပြုပါ။ စောင့်ရမယ့်အလုပ်က ဒီအလုပ်မို့လို့ `await` ထည့်ပြီးလုပ်ပေးရတာပါ။ ပြီးတော့မှ ရလာတဲ့ ရလဒ်ကို ဖော်ပြစေထားပါတယ်။ အခုလိုစမ်းကြည့်နိုင်ပါတယ်။

JavaScript

```
console.log("some processes")
add1000later()
console.log("more processes")

// some processes
// more processes
// 500500
```

စောစောက Promise နဲ့ရေးခဲ့တဲ့ရလဒ်နဲ့ တူညီတဲ့ရလဒ်ကို ရရှိခြင်းပဲ ဖြစ်ပါတယ်။ တစ်ချို့အတွက်တော့ ဒီ async & await ရေးထုံးက ပိုပြီးတော့ နားလည်လွယ်နေတာမျိုး ဖြစ်နိုင်ပါတယ်။ ရလဒ်နဲ့ သဘောသဘာဝက အတူတူပါပဲ။ ရေးထုံးမှာ ကွာသွားတာပဲ ဖြစ်ပါတယ်။

Piratical Sample

ဒီအတိုင်းရှင်းလင်းချက်တွေချဉ်းပဲ ပြောနေတာ ပျင်းစရာကောင်းတယ်ထင်ရင် နည်းနည်းစိတ်ဝင်စားဖို့ ကောင်းသွားအောင် လက်တွေ့ကုဒ်လေးတစ်ချို့ ရေးပြီးတော့လည်း စမ်းကြည့်နိုင်ပါတယ်။ ပထမဆုံး URL လိပ်စာလေး တစ်ခုအရင်မှတ်ထားပါ။

https://api.covid19api.com/summary

ဒီ URL လိပ်စာအတိုင်း Browser မှာ ရိုက်ထည့်ကြည့်လို့ရပါတယ်။ ဒါဆိုရင် Covid-19 ရောဂါဖြစ်ပွားမှု အခြေအနေနဲ့ပက်သက်တဲ့ Data တွေကို အခုလို JSON ဖွဲ့စည်းပုံမျိုးနဲ့ တွေ့မြင်ရမှာပါ။ အင်တာနက် အဆက်အသွယ် ရှိဖို့လိုတယ်ဆိုတာကိုတော့ သတိပြုပါ။

```
{
  "Global": {
    "NewConfirmed": 100282,
    "TotalConfirmed": 1162857,
    "NewDeaths": 5658,
    "TotalDeaths": 63263,
    "NewRecovered": 15405,
    "TotalRecovered": 230845
  },
  "Countries": [
    {
      "Country": "Afghanistan",
      "CountryCode": "AF",
      "Slug": "afghanistan",
      "NewConfirmed": 18,
      "TotalConfirmed": 299,
      "NewDeaths": 1,
      "TotalDeaths": 7,
      "TotalRecovered": 10,
      "Date": "2020-04-05T06:37:00Z"
    },
    {
      "Country": "Albania",
      "CountryCode": "AL",
      "Slug": "albania",
      "NewConfirmed": 29,
      "TotalConfirmed": 333,
      "NewDeaths": 3,
      "TotalDeaths": 20,
      "TotalRecovered": 99,
      "Date": "2020-04-05T06:37:00Z"
    },
    ...
  ]
}
```

တစ်ကမ္ဘာလုံးနဲ့ သပ်ဆိုင်တဲ့ အချက်အလက်တွေက `Global` Property မှာရှိနေပြီး နိုင်ငံတစ်ခုချင်းစီ အလိုက် အချက်အလက်တွေကတော့ `Countries` Property မှာ Array တစ်ခုအနေနဲ့ ရှိနေတာကို တွေ့ ရနိုင်ပါတယ်။

JavaScript မှာ ဒီလို Data တွေကို ဆက်သွယ်ရယူပေးနိုင်တဲ့ `fetch()` လို့ခေါ်တဲ့ Standard လုပ်ဆောင်ချက်တစ်ခုရှိပါတယ်။ `fetch()` ဟာ Promise ကို အသုံးပြု အလုပ်လုပ်တဲ့ နည်းပညာတစ်ခု ပါ။ API ဘက်ပိုင်းလေ့လာတဲ့အခါ အသေးစိတ်လေ့လာရပါလိမ့်မယ်။ အခုတော့ အသေးစိတ် မဟုတ်သေး ပေမယ့် ဒီ `fetch()` ကိုအသုံးပြုပြီး အချက်အလက်ရယူပုံကို ကြည့်ကြပါမယ်။ ဒီကုဒ်ကို လေ့လာ ကြည့်ပါ။

JavaScript

```
fetch("https://api.covid19api.com/summary")

  .then(res => res.json())

  .then(data => {
    const global = data.Global
    const allCountries = data.Countries
    const myanmar = allCountries.find(c => c.Country === "Myanmar")

    console.log("Global:", global, "Myanmar:", myanmar)
  })
```

ရေးထားတဲ့ကုဒ်ကို လေ့လာကြည့်ပါ။ `fetch()` Function က URL လိပ်စာအတိုင်း အချက်အလက်တွေ သွားယူပေးပါတယ်။ ဒီလိုယူတဲ့အလုပ်ကို Promise အနေနဲ့ လုပ်သွားတာပါ။ ဒါကြောင့် တစ်ခြားအလုပ် တွေရှိရင် သွားယူတဲ့အလုပ် ပြီးအောင် စောင့်စရာမလိုပါဘူး။ ကြိုတင်ပြီး ဆက်လုပ်သွားလို့ ရနိုင်ပါတယ်။

ရယူလို့ပြီးတဲ့အခါ အချက်အလက်တွေက `then()` Method ရဲ့ `res` Parameter ထဲမှာ HTTP Response Object တစ်ခုအနေနဲ့ ရှိနေမှာပါ။ HTTP Response Object မှာ ရလဒ် JSON ကို JavaScript Object ပြောင်းတဲ့ လုပ်ဆောင်ချက် တစ်ခါထဲပါဝင်ပါတယ်။ ဒါကြောင့် အဲ့ဒီလုပ်ဆောင်ချက် အကူအညီနဲ့ `res.json()` ဆိုပြီး ရလာတဲ့ JSON အချက်အလက်ကို JavaScript Object ပြောင်းလိုက်ပါတယ်။

JavaScript Object ဟာ နောက်တစ်ဆင့် ဆက်လိုက်တဲ့ `then()` Method ရဲ့ `data` Parameter ထဲကို ရောက်သွားပါလိမ့်မယ်။ ဒါကြောင့် အဲ့ဒီ `data` ထဲကနေ တစ်ကမ္ဘာလုံးဆိုင်ရာ အချက်အလက်တွေအတွက် Global Property ကိုရယူပြီး၊ နိုင်ငံအားလုံးရဲ့ အချက်အလက်အတွက် Countries Property ထဲကနေ ရယူထားပါတယ်။ ပြီးတော့မှ မြန်မာနိုင်ငံရဲ့အချက်အလက်ကို လိုချင်တဲ့အတွက် `countries` Array မှာ `find()` နဲ့ Myanmar ကို ရှာယူထားပါတယ်။ Array Method တစ်ခုဖြစ်တဲ့ `find()` ဟာ `filter()` နဲ့ဆင်တူပါတယ်။ `filter()` က မူလ Array ကနေ လိုချင်တဲ့ Item တွေကို Array တစ်ခုအနေနဲ့ ထုတ်ယူပေးပြီး `find()` ကတော့ မူလ Array ကနေ လိုချင်တဲ့ Item တစ်ခုကို ထုတ်ယူပေးပါတယ်။

ရရှိလာတဲ့ တစ်ကမ္ဘာလုံးအချက်အလက်နဲ့ မြန်မာနိုင်ငံရဲ့ အချက်အလက်ကို ဖော်ပြစေတဲ့အတွက် စမ်းကြည့်လိုက်ရင် အခုလိုရလဒ်ကို ရမှာဖြစ်ပါတယ်။

```
Global:
{
    "NewConfirmed": 100282,
    "TotalConfirmed": 1162857,
    "NewDeaths": 5658,
    "TotalDeaths": 63263,
    "NewRecovered": 15405,
    "TotalRecovered": 230845
}

Myanmar:
{
    "Country": "Myanmar",
    "CountryCode": "MM",
    "Slug": "myanmar",
    "NewConfirmed": 1,
    "TotalConfirmed": 21,
    "NewDeaths": 0,
    "TotalDeaths": 1,
    "NewRecovered": 0,
    "TotalRecovered": 0,
    "Date": "2020-04-05T06:37:00Z"
}
```

တစ်လက်စထဲ ရလဒ်ကို ဖော်ပြစေဖို့အတွက် တန်ဖိုးနှစ်ခုသုံးခုရှိရင် `console.log()` နှစ်ကြောင်းသုံးကြောင်း မရေးတော့ဘဲ၊ တစ်ကြောင်းထဲမှာ ဖော်ပြစေလိုတဲ့ တန်ဖိုးတွေကို Comma ခံပြီး တန်းစီပေးလို့ ရတယ်ဆိုတာကိုလည်း သတိပြုပါ။ နောက်ထပ်သတိပြုစရာတစ်ခုအနေနဲ့ တန်ဖိုးတွေ ပြောင်းဖို့မရှိရင် ရိုးရိုး Variable တွေအစား Constant တွေကို အသုံးပြုသင့်ပါတယ်။ ဒါကြောင့် နမူနာမှာ Constant တွေကိုချည်းပဲ သုံးထားတာပါ။

ဒီကုဒ်ကို အင်တာနက်အဆက်အသွယ်ရှိရင် Browser Console မှာ လက်တွေ့ ရေးစမ်းလို့ရပါတယ်။ စမ်းကြည့်လိုက်ပါ။ ဒီကုဒ်ကို async, await နဲ့လည်း ပြောင်းရေးလို့ ရနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
async function covidInfo() {
    const response = await fetch("https://api.covid19api.com/summary")
    const data = await response.json()

    const global = data.Global
    const allCountries = data.Countries
    const myanmar = allCountries.find(c => c.Country === "Myanmar")

    console.log("Global:", global, "Myanmar:", myanmar)
}

covidInfo()
```

async, await နဲ့မို့လို့ ကုဒ် Block တွေ အဆင့်ဆင့်မရှိတော့ဘဲ တစ်ကြောင်းချင်း တစ်ဆင့်ချင်း ကြည့်လို့ရသွားပါတယ်။ `covidInfo()` Function ဟာ ရိုးရိုး Function မဟုတ်ဘဲ Async Function တစ်ခုဖြစ်လို့ သူ့ကို သူများတွေက စောင့်စရာမလိုဘဲ လိုအပ်ရင် ကိုယ့်အလုပ်ကိုယ် ဆက်လုပ်သွားလို့ ရနိုင်ပါတယ်။ Function ထဲမှာ `fetch()` နဲ့ Data ကို ယူတဲ့အခါ အချိန်ယူပြီး လုပ်ရမယ့်အလုပ်မို့လို့ `await` နဲ့ ယူထားပါတယ်။ ရလာတဲ့ Data ကို JavaScript Object ပြောင်းတဲ့အလုပ်ဟာလည်း အချိန်ယူပြီး လုပ်ဖို့လိုနိုင်လို့ `await` နဲ့ပဲ လုပ်ခိုင်းထားပါတယ်။ အားလုံးပြီးစီးတော့မှ စောစောကလိုပဲ Global Data ထုတ်ယူတာတွေ မြန်မာနိုင်ငံအတွက် Data ထုတ်ယူတာတွေကို ဆက်လုပ်ထားတာပါ။ ဒါကြောင့် ဒီကုဒ်ကို စမ်းကြည့်လိုက်ရင်လည်း စောစောကုဒ်နဲ့ တူညီတဲ့ရလဒ်ကိုပဲ ရမှာဖြစ်ပါတယ်။

တစ်ကယ်တော့ လေ့လာစမှာ ဒီလိုအလုပ်ရှုပ်တဲ့ကိစ္စတွေ ထည့်မပြောချင်သေးပါဘူး။ တစ်ဖြည်းဖြည်းချင်း တစ်ဆင့်ချင်း သွားတာပဲ ကောင်းပါတယ်။ အဆင့်ကျော်သလို ဖြစ်သွားရင် မကောင်းပါဘူး။ ဒါပေမယ့် တစ်ချိန်လုံး နားလည်ရလွယ်တဲ့ အသေးအဖွဲ့ နမူနာလေးတွေချည်း ကြည့်လာတော့ ပျင်းစရာဖြစ်နေမှာစိုးလို့ အခုလို လက်တွေ့ကျတဲ့ နမူနာတစ်ခုကို ထည့်ပေးလိုက်တာပါ။ သိပ်နားမလည်ဘူးဆိုရင် စိတ်မပူပါနဲ့။ အဆင့်နည်းနည်းကျော်ပြီး ပြောလိုက်မိလို့ပါ။ ဆက်လေ့လာသွားပါ။ ရှေ့ပိုင်းက လေ့လာခဲ့တာတွေ ကြေညက်အောင် ထပ်ခါထပ်ခါပြန်ကြည့် ပြန်စမ်းထားပါ။ အချိန်တန်ရင် ရသွားပါလိမ့်မယ်။

အခန်း (၉) – Code Style Guide

ကုဒ်တွေရေးတဲ့အခါ အဖွင့်အပိတ်၊ အထားအသိုတွေဟာ ဘယ်လိုပဲထားထား ရေးထုံးအရ မှန်နေသ၍ အလုပ်လုပ်နေမှာပါပဲ။ ဒါပေမယ့် အလုပ်လုပ်နေယုံနဲ့ မရပါဘူး။ ရေးသားပုံ ရှင်းလင်းသပ်ရပ်မှသာ ဖတ်ရှုနားလည်ရ လွယ်ကူတဲ့ကုဒ် ဖြစ်မှာပါ။ ဒီအတွက် Code Style Guide တွေ ရှိကြပါတယ်။ Coding Standard လို့လည်း ခေါ်ကြပါတယ်။ Language တစ်ခုနဲ့တစ်ခု မတူကြသလို၊ အဖွဲ့အစည်းတစ်ခုနဲ့တစ်ခုရဲ့ Recommendation တွေလည်း မတူကြပါဘူး။ JavaScript အတွက်ဆိုရင် Google ရဲ့ Style Guide ရှိသလို၊ Mozilla, Wordpress, Drupal, AirBnb စသဖြင့် ထိပ်တန်းအဖွဲ့အစည်း အသီးသီးမှာ သူ့သတ်မှတ်ချက်နဲ့သူရှိပါတယ်။ ပြီးတော့ ပရိုဂရမ်မာ တစ်ဦးချင်းစီလည်း အကြိုက်တွေ ကွဲကြပါသေးတယ်။ ဒါကြောင့် ဒီအကြောင်းအရာဟာလည်း သူနေရာနဲ့သူ ကျယ်ပြန့်တဲ့ အကြောင်းအရာတစ်ခုပါ။

တစ်ကယ်တော့ ပရိုဂရမ်တစ်ဦးအနေနဲ့ JavaScript တစ်မျိုးထဲ ရေးယုံနဲ့ မပြီးသေးပါဘူး။ လက်တွေ့လုပ်ငန်းခွင်မှာ တစ်ခြားကုဒ်အမျိုးအစားတွေကိုလည်း ပူးတွဲပြီး ရေးကြရဦးမှာပါ။ HTML, CSS, JavaScript, jQuery, React, PHP, Laravel စသဖြင့် Language, Library, Framework အမျိုးမျိုးနဲ့ အလုပ်လုပ်ကြရမှာပါ။ ဒါကြောင့် JavaScript အတွက် မဟုတ်ဘဲ ကုဒ်အမျိုးအစား အမျိုးမျိုးနဲ့ သင့်တော်မယ့် ယေဘုယျ ရေးဟန်လေးတွေကို စုစည်း ဖော်ပြချင်ပါတယ်။

Variables

Variable, Function, Class စသဖြင့် အမည်တွေပေးဖို့လိုတဲ့အပါ ပေးပုံပေးနည်း (၄) နည်းရှိပါတယ်။ အခေါ်အဝေါ်လေးတွေ မှတ်ထားပါ။

- All Cap: `HELLO_WORLD`
- Snake Case: `hello_world`
- Camel Case: `helloWorld`
- Capital Case: `HelloWorld`

All Cap ရေးဟန်မှာ စာလုံးအကြီးတွေချည်းပဲ သုံးပါတယ်။ Word တစ်ခုနဲ့တစ်ခု ပိုင်းခြားဖို့အတွက် Underscore ကို အသုံးပြုပါတယ်။ Snake Case ရေးဟန်မှာ စာလုံးအသေးတွေချည်းပဲ သုံးပါတယ်။ သူလည်းပဲ Word တစ်ခုနဲ့တစ်ခု ပိုင်းခြားဖို့အတွက် Underscore ကို အသုံးပြုပါတယ်။ Camel Case ကတော့ စာလုံးအကြီးအသေး ရောရေးတဲ့ ရေးဟန်ပါ။ စာလုံးသေးနဲ့စပြီး နောက်က ဆက်လိုက်တဲ့ Word တွေအားလုံးရဲ့ ရှေ့ဆုံးတစ်လုံးကို စာလုံးကြီးနဲ့ ရေးပါတယ်။ Capital Case ကတော့ Camel Case နဲ့ဆင်တူပါတယ်။ ထူးခြားချက်အနေနဲ့ ရှေ့ဆုံးစာလုံးပါ ကြီးသွားတာပါ။

Constant တွေကြေညာဖို့အတွက် All Cap ကို အသုံးပြုသင့်ပါတယ်။ ရိုးရိုး Variable တွေအတွက် Snake Case ကိုသုံးသင့်ပြီး၊ Object Property တွေအတွက် camelCase ကိုသုံးသင့်ပါတယ်။

Pseudocode

```
const PI = 3.14
const MIN = 0
const MAX = 100

let color_name = "red"
let color_code = "#112233"
```

Assignment Operator အပါဝင် Operator အားလုံးရဲ့ ရှေ့နဲ့နောက်မှာ Space တစ်ခုစီ ခြားပြီးတော့ ရေးသင့်ပါတယ်။ `x`, `y`, `a`, `b`, `n`, `i` စသည်ဖြင့် အတိုကောက်အမည်တွေဟာ နမူနာမို့လို့သာ ပေးခဲ့တာပါ။ တစ်ကယ့်လက်တွေ့မှာ ရှည်ချင်ရှည်ပါစေ၊ အဓိပ္ပါယ်ပေါ်လွင်တဲ့ Variable အမည်ကိုသာ ပေးသင့်ပါ

တယ်။ ကုဒ်တွေများလာတော့မှ `a` ဆိုတာ ဘာကိုပြောမှန်း ကိုယ့်ဘာသာမသိတော့တာ၊ `i` ဆိုတာ ဘယ်နားက `i` ကိုပြောတာလည်း ရှာရခက်ကုန်တာမျိုးတွေ ဖြစ်တတ်ပါတယ်။

Arrays & Objects

Array တွေကြေညာသတ်မှတ်တဲ့အခါ တန်ဖိုးနည်းရင် တစ်ကြောင်းထဲ၊ တစ်ဆက်ထဲ ရေးလို့ရပါတယ်။ အဲ့ဒီလိုရေးတဲ့အခါ Comma ရဲ့နောက်မှာ Space တစ်ခုထည့်သင့်ပါတယ်။

Pseudocode

```
let users = ["Alice", "Bob", "Tom", "Mary"]
```

တန်ဖိုးတွေများရင်တော့ လိုင်းခွဲပြီးရေးသင့်ပါတယ်။ အဲ့ဒီလိုခွဲရေးတဲ့အခါ အဖွင့်ကို Assignment Operator နဲ့ တစ်လိုင်းထဲမှာထားပြီး အပိတ်ကို အောက်ဆုံးမှာ သီးခြားတစ်လိုင်းနဲ့ ထားသင့်ပါတယ်။ အထဲက တန်ဖိုးတွေကို သက်ဆိုင်ရာ Array အတွင်းက တန်ဖိုးမှန်း မြင်သာအောင် Indent လေး တွန်းပြီးတစ်ညီထဲ ရေးသင့်ပါတယ်။ တစ်ကြောင်းမှာ Item တစ်ခုသာ ပါဝင်သင့်ပါတယ်။

Pseudocode

```
let users = [
    "Alice",
    "Bob",
    "Tom",
    "Mary",
]
```

နောက်ဆုံးက Comma အပို Trailing Comma ကို ထည့်လို့ရရင် ထည့်သင့်ပါတယ်။ တစ်ချို့ Language တွေက ထည့်ခွင့်မပြုလို့ နည်းနည်းတော့ သတိထားပါ။ Indent အတွက် အရှယ်အစားအနေနဲ့ တစ်ချို့က 2 Spaces သုံးကြပါတယ်။ တစ်ချို့က 4 Spaces အသုံးကြပါတယ်။ 2 Spaces က နေရာယူသက်သာလို့ လူကြိုက်များပေမယ့် စာရေးသူကတော့ 4 Spaces ကိုသာ သုံးလေ့ရှိပါတယ်။ ပိုကျဲလို့ ဖတ်ရတာ မျက်စိထဲမှာ ပိုရှင်းတဲ့အတွက်ကြောင့်ပါ။ 2 Spaces သုံးသည်ဖြစ်စေ၊ 4 Spaces သုံးသည်ဖြစ်စေ၊ ကီးဘုတ်ကနေ Space Bar ကို တစ်ချက်ချင်း နှိပ်ပြီးတော့ မရေးသင့်ပါဘူး။ Tab Key ကိုပဲသုံးသင့်ပါတယ်။ Code Editor တွေက Tab Key ကိုနှိပ်လိုက်ရင် သတ်မှတ်ထားတဲ့ Space အရေအတွက်အတိုင်း အလိုအလျောက် ထည့်ပေးသွားကြပါတယ်။ Space ကို လုံးဝ မသုံးဘဲ Indent အတွက် Tab ကိုပဲ သုံးလို့လည်းရပါတယ်။

Object တွေကိုတော့ တစ်ချို့အရမ်းတိုတဲ့ Object တွေကလွဲရင် အများအားဖြင့် လိုင်းခွဲပြီးတော့ ရေးသင့်ပါတယ်။ Property တွေ Method တွေအတွက် Camel Case ကို သုံးသင့်ပါတယ်။

Pseudocode

```
let user = {
    firstName: "James",
    sayHello () {
        // Statements
    },
}
```

Functions

Function တွေကြေညာတဲ့အခါ တွန့်ကွင်းအဖွင့်ကို `function` Keyword နဲ့ တစ်လိုင်းထဲထားလို့ ရသလို နောက်တစ်လိုင်းဆင်းပြီး ထားလို့လည်း ရပါတယ်။ JavaScript ကုဒ်တွေမှာတော့ တစ်လိုင်းထဲပဲ ထားကြလေ့ရှိပါတယ်။ တစ်ခြား Language တွေမှာတော့ ရိုးရိုး Function တွေအတွက် တစ်လိုင်းထဲ ထားပြီး Class Method တွေအတွက် နောက်တစ်လိုင်း ခွဲထားကြလေ့ ရှိပါတယ်။

Pseudocode

```
function add(a, b) {
    // Statements
}

function sum(a, b)
{
    // Statements
}
```

တစ်လိုင်းထဲထားတဲ့အခါ ဝိုက်ကွင်းအပိတ်နဲ့ တွန့်ကွင်းအဖွင့်ကြား Space တစ်ခု ထည့်ပေးသင့်ပါတယ်။ ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲက Parameter List မှာလည်း Comma နောက်မှာ Space တစ်ခု ခြားပေးရပါမယ်။

Function အမည်တွေဟာ Camel Case ဖြစ်သင့်ပါတယ်။ တစ်ချို့ Language တွေမှာ Snake Case ကို Function အမည်အတွက် သုံးကြပေမယ့် JavaScript မှာတော့ Camel Case ကို ပိုအသုံးများပါတယ်။ ရိုးရိုး Constant တန်ဖိုးတွေအတွက် စာလုံးအကြီးတွေနဲ့ ပေးသင့်တယ်လို့ ဆိုခဲ့ပေမယ့် Function

Expression တွေကို Constant နဲ့ရေးတဲ့အခါမှာတော့ Camel Case ကိုပဲ သုံးသင့်ပါတယ်။ Constant လို ကြေညာနေပေမယ့် သူ့ကို Function လို သုံးမှာမို့လို့ပါ။

Pseudocode

```
const add = function (a, b) {
    // Statements
}
```

Function Expression တွေရေးတဲ့အခါ `function` Keyword နောက်မှာ Space တစ်ခု ပါသင့်ပါတယ်။ Arrow Function တွေရေးတဲ့အခါ Arrow သင်္ကေတရဲ့ ရှေ့နောက်မှာ Space တစ်ခုစီ ပါသင့်ပါတယ်။

Pseudocode

```
const add = (a, b) => a + b
```

ပေါင်းနှုတ်မြှောက်စား၊ Comparison စသည်ဖြင့် Operator အားလုံးရဲ့ ရှေ့နောက်မှာ Space တစ်ခုစီ ပါဝင်သင့်တယ်ဆိုတာကို နောက်တစ်ကြိမ် ထပ်ပြောချင်ပါတယ်။ Operator တွေနဲ့ တန်ဖိုးတွေကို ကပ်ရေးရင် ရောထွေး ပူးကပ်ပြီး ဖတ်ရခက်တတ်ပါတယ်။

Code Blocks

If Statement တွေရေးတဲ့အခါ တွန့်ကွင်း အဖွင့်ကို `if` နဲ့တစ်လိုင်းထဲ ထားသင့်ပါတယ်။ `else` Statement ပါမယ်ဆိုရင် `if` အတွက် တွန့်ကွင်းအပိတ်နဲ့ `else` အတွက် တွန့်ကွင်းအဖွင့်တို့ကို တစ်လိုင်းထဲ ထားပြီး ရေးသင့်ပါတယ်။

Pseudocode

```
if(true) {
    // Statements
} else {
    // Statements
}
```

Switch Statement တွေရေးတဲ့အခါ `case` တစ်ခုစီနဲ့သက်ဆိုင်တဲ့ Statement တွေကို တစ်ဆက်ထဲ ရေးလို့လည်း ရပေမယ့် နောက်တစ်လိုင်းဆင်းပြီး ရေးသင့်ပါတယ်။

Pseudocode

```
switch(expression) {
    case 1:
        // Statements
        break
    case 2:
        // Statements
        break
    default:
        // Statements
}
```

ကျန်တဲ့ `while, for, for-of` စတဲ့ Loop တွေရဲ့ရေးဟန်ကို သီးခြားထပ်ထည့် မပြောတော့ပါဘူး။ ဒီသဘောပေါ်မှာပဲ အခြေခံပြီး ရေးသားရမှာ ဖြစ်ပါတယ်။ Indent တွေကိုတော့ မှန်အောင်ပေးဖို့ လိုပါမယ်။ Code Block တွေ အထပ်ထပ်အဆင့်ဆင့် ဖြစ်လာတဲ့အခါ ဘယ်ကုဒ်က ဘယ် Block အတွက်လည်းဆိုတာ Indent မှန်မှပဲ သိသာမြင်သာမှာပါ။ Indent မမှန်ရင်တော့ ကိုယ့်ကုဒ်ကို ကိုယ်တိုင်ပြန်ဖတ်လို့တောင် နားမလည်ဘူးဆိုတာမျိုးတွေ ဖြစ်လာနိုင်ပါတယ်။

Pseudocode

```
function doSomething() {
    while(condition) {
        if(condition) {
            // Statements
        }

        // Statements
    }

    // Statements
}
```

If Statement တွေမှာ Statement တစ်ကြောင်းထဲရှိလို့ တွန့်ကွင်းမပါဘဲ ရေးချင်ရင် နောက်တစ်လိုင်း မဆင်းသင့်ပါဘူး၊ တစ်ကြောင်းထဲပဲ ဖြစ်သင့်ပါတယ်။ နောက်တစ်လိုင်း ဆင်းဖို့လိုရင် တွန့်ကွင်းအဖွင့်အပိတ် ထည့်လိုက်တာ ပိုကောင်းပါတယ်။

Pseudocode

```
if(true) doSomething()
else doElse()
```

Method တစ်ခုနဲ့တစ်ခု Function တစ်ခုနဲ့တစ်ခု Block တစ်ခုနဲ့တစ်ခုကြားထဲမှာ လိုင်းအလွတ်တစ်လိုင်း ခြားပြီးတော့ ရေးသင့်ပါတယ်။ Variable ကြေညာချက်တွေ၊ Return Statement တွေနဲ့ တစ်ခြားကုဒ်တွေ ကိုလည်း လိုင်းအလွတ် တစ်လိုင်းခြားပြီး ရေးသင့်ပါတယ်။

Pseudocode

```
function sub(a, b) {
    let result = 0

    if(a > b) result = a - b
    else result = b - a

    return result
}
```

Comments

// Operator ကိုသုံးပြီးရေးတဲ့ Comment တွေမှာ Operator နဲ့ စာကြားထဲမှာ Space တစ်ခု ပါသင့်ပါတယ်။ /* */ Operator ကိုသုံးပြီး ရေးတဲ့အခါ တစ်ကြောင်းထဲဆိုရင် အဖွင့်အပိတ်ရဲ့ ရှေ့နောက်မှာ Space တစ်ခုစီပါသင့်ပါတယ်။ တစ်ကြောင်းထက် ပိုမယ်ဆိုရင် အဖွင့်ကို အပေါ်ဆုံးမှာ သပ်သပ်ရေးပြီး အပိတ်ကို အောက်ဆုံးမှာ သပ်သပ်ရေးသင့်ပါတယ်။

Pseudocode

```
// Line Comment

/* Single Line Comment */

/*
Some comments with
more than single lines
*/
```

Classes

Class Name တွေဟာ Capital Case ဖြစ်သင့်ပါတယ်။ Property တွေ Method တွေကတော့ Camel Case ကို အသုံးပြုသင့်ပါတယ်။ Class အတွက် တွန့်ကွင်းအဖွင့်ကို တစ်ခြား Language တွေမှာ အောက် တစ်လိုင်း ဆင်းရေးကြလေ့ရှိပြီး JavaScript မှာ class Keyword နဲ့ တစ်လိုင်းထဲပဲ ထားရေးကြလေ့ ရှိပါ တယ်။

Pseudocode

```
class Animal {
    constructor(name) {
        this.name = name
    }

    makeSound () {
        // Statements
    }
}
```

တစ်ချို့ Language တွေမှာ Private Property တွေ Private Method တွေကို ရှေ့ကနေ Underscore နဲ့စ သင့်တယ်ဆိုတဲ့ သတ်မှတ်ချက်ရှိပါတယ်။ အမည်ကိုကြည့်လိုက်ယုံနဲ့ ကွဲပြားစေချင်တဲ့အတွက် ဖြစ်ပါ တယ်။ JavaScript မှာတော့ Private တွေကို ရှေ့ကနေ # သင်္ကေတ ခံရေးရလို့ Underscore မထည့် လည်းပဲ ကွဲပြားပြီးဖြစ်ပါတယ်။

Callback Function & Method Chaining

Function တစ်ခုကို ခေါ်ယူစဉ်မှာ Callback Function တွေပေးရတဲ့အခါ အများအားဖြင့် Callback Function တွေက နောက်ဆုံးက နေလေ့ရှိပါတယ်။ အဲ့ဒီအခါမှာ မူလ Function ကိုခေါ်တဲ့ ဝိုက်ကွင်း အပိတ်နဲ့ Callback Function ရဲ့ တွန့်ကွင်းအပိတ်ကို တစ်လိုင်းထဲ ရေးပေးသင့်ပါတယ်။

Pseudocode

```
add(1, 2, function(3, 4) {
    // Callback Function Statements
})
```

တစ်ချို့လည်း Callback Function က ရှေ့ကနေတာမျိုး ဖြစ်နိုင်ပါတယ်။ အဲ့ဒီအခါမှာ Callback Function ရဲ့အပိတ်နောက်မှာ Comma ခံပြီး ကျန် Argument တွေကို တစ်ဆက်ထဲ ပေးသင့်ပါတယ်။

Pseudocode

```
setTimeout(function() {
    // Callback Function Statements
}, 2000)
```

တစ်ခါတစ်ရံ Callback Function နှစ်ခုသုံးခုလည်း ဖြစ်တတ်ပါတယ်။ အဲ့ဒီအခါမှာ ပထမ Callback Function ရဲ့အပိတ်တွန့်ကွင်းနောက်မှာ Comma နဲ့အတူ ဒုတိယ Callback Function လိုက်သင့်ပါတယ်။

Pseudocode

```
add(function(1, 2) {
    // Callback Function Statements
}, function(3, 4) {
    // Callback Function Statements
})
```

တစ်ချို့ Object Methods တွေကို တွဲဆက်ပြီး အတွဲလိုက် ခေါ်ယူအသုံးပြုလိုတဲ့အခါ တိုရင်တစ်ဆက်ထဲ ရေးလို့ရပါတယ်။ လိုအပ်ရင်တော့ Method တစ်ခုကိုတစ်လိုင်းခွဲပြီး သုံးသင့်ပါတယ်။

Pseudocode

```
users.map(u => u.name).filter(u => u.age > 18)

users
    .map(u => u.name)
    .filter(u => u.age > 18)
```

Semicolon

ဒီကိစ္စကတော့ ရှင်းမလိုလိုနဲ့ ရှုပ်နေတဲ့ကိစ္စလေးပါ။ Language အတော်များများမှာ Statement တစ်ခုဆုံးတိုင်း Semicolon နဲ့ ပိတ်ပေးရပါတယ်။ JavaScript မှာလည်း အဲ့ဒီလို ပိတ်ပေးဖို့ လိုအပ်ပါတယ်။ ဒါပေမယ့် Automatic Semicolon Insertion (ASI) လို့ခေါ်တဲ့ နည်းစနစ်တစ်မျိုး ရှိနေလို့ ကိုယ့်ဘာသာ ထည့်

ပေးစရာမလိုပါဘူး။ Language က လိုတဲ့နေရာမှာ သူ့ဘာသာထည့်ပြီး အလုပ်လုပ်သွားပါတယ်။ ဒါကြောင့် ရေးတဲ့သူက Semicolon တွေကို ထည့်ရေးလည်းရတယ်။ မထည့်ဘဲရေးလည်း ရနေပါတယ်။

လက်တွေ့ကုဒ်တွေ ရေးတဲ့အခါ JavaScript တစ်ခုထဲနဲ့ မပြီးပါဘူး၊ တစ်ခြား Language တွေ နည်းပညာတွေနဲ့ လုပ်ငန်းလိုအပ်ချက်ပေါ်မူတည်ပြီး ပူးတွဲအသုံးပြုရတာတွေ ရှိပါလိမ့်မယ်။ အဲ့ဒီမှာ C/C++ တို့ Java တို့လို Language မျိုးတွေမှာ Statement ဆုံးတိုင်း မဖြစ်မနေ Semicolon ထည့်ရေးပေးရပါတယ်။ Ruby တို့ Python တို့လို Language မျိုးတွေမှာကျတော့ ထည့်ဖို့မလိုတော့ပြန်ပါဘူး။ JavaScript နဲ့ တွဲသုံးဖြစ်ဖို့များတဲ့ CSS တို့ PHP တို့မှာ ထည့်ဖို့ လိုသွားပြန်ပါတယ်။ Semicolon တွေထည့်လိုက် မထည့်လိုက်ဆိုရင် အလေ့အကျင့်ပျက်ပြီး မကျန်သင့်တဲ့နေရာမှာ ကျန်ခဲ့လို့ မလိုအပ်ဘဲ Error တွေတက်တယ်ဆိုမျိုး ဖြစ်နိုင်ပါတယ်။ ဒါကြောင့် တစ်ချို့က Statement ဆုံးတိုင်း Semicolon နဲ့ အဆုံးသတ်ပေးတဲ့ အလေ့အကျင့် ရှိထားသင့်တယ်လို့ ဆိုကြပါတယ်။

ပြီးတော့ Automatic Semicolon Insertion စနစ်က တစ်ချို့ဆန်းပြားတဲ့ ကုဒ်တွေမှာ မထည့်သင့်တဲ့နေရာ Semicolon ထည့်ပြီး အလုပ်လုပ်နေလို့ မှားနေတာမျိုးတွေ ရှိတတ်ပါတယ်။ ဖြစ်ခဲ့ပေမယ့် အဲ့ဒီလို မှားတဲ့အခါ အမှားရှာက အရမ်းခက်ပါတယ်။ ဒီအချက်ကြောင့်လည်း Statement ဆုံးတိုင်း Semicolon နဲ့ အဆုံးသတ်ပေးသင့်တယ်လို့ ဆိုကြပါတယ်။ စာရေးသူကိုယ်တိုင်လည်း အရင်က အမြဲထည့်သင့်တယ်လို့ ခံယူပေမယ့်၊ အခုနောက်ပိုင်းမှာ တစ်ချို့ Library ကုဒ်တွေ Framework ကုဒ်တွေက မထည့်ဘဲ ရေးကြတော့ ကိုယ်ကလည်း တူညီသွားအောင် မထည့်ဘဲ လိုက်ရေးရတာတွေ ရှိပါတယ်။

ဒီစာအုပ်မှာ ဖော်ပြခဲ့တဲ့ နမူနာတွေမှာ Semicolon ထည့်ပြီး မဖော်ပြခဲ့ပါဘူး။ စလေ့လာမယ့်သူတွေအတွက် မျက်စိရှုပ်စရာ တစ်ခုသက်သာလည်း မနည်းဘူးလို့သဘောထားတဲ့အတွက် မထည့်ဘဲ ရေးပြခဲ့တာပါ။ မတော်တစ်ဆ မထည့်သင့်တဲ့နေရာ ထည့်မိပြီး Error တွေ ဖြစ်နေမှာ စိုးလို့လည်း ပါပါတယ်။ နောက်ဆုံး ဥပမာပေးခဲ့တဲ့ ကုဒ်မှာတောင် အဲ့ဒီလိုအမှားမျိုး ဖြစ်နိုင်ခြေရှိပါတယ်။

Pseudocode

```
users.map(u => u.name).filter(u => u.age > 18);

users
    .map(u => u.name);
    .filter(u => u.age > 18);
```

ပထမကုဒ်ရဲ့နောက်ဆုံးမှာ Semicolon ပိတ်ပေးထားတာ မှန်ပါတယ်။ ဒုတိယကုဒ်မှာတော့ မှားသွားပါပြီ။ Line အဆုံးမို့လို့ ယောင်ပြီး ထည့်လိုက်မိတာမျိုးပါ။ တစ်ကယ်တော့ `map()` နောက်မှာ Semicolon မထည့်ရပါဘူး။ Statement မပြီးသေးပါဘူး။ ဒီလိုအမှားမျိုးတွေကြောင့် မလိုအပ်ဘဲ Error တွေတက်ပြီး ကြန့်ကြာမှာစိုးလို့ Semicolon မထည့်ဘဲ ရေးပြခဲ့တာပါ။ လက်တွေ့မှာ ထည့်ရေးတာက အလေ့အကျင့်ကောင်းဖြစ်ပြီး မထည့်ဘဲရေးရင်လည်း ပြဿနာတော့ မရှိပါဘူးလို့ ဆိုချင်ပါတယ်။

အခန်း (၁၀) – Modules

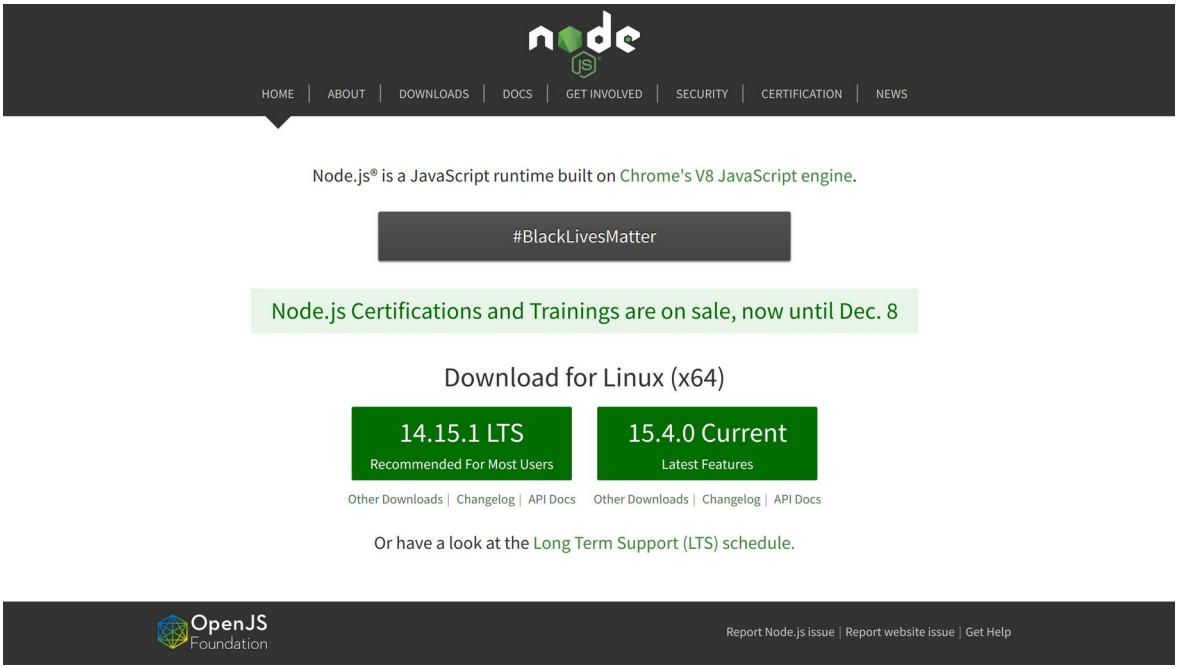
JavaScript မှာ Environment နှစ်ခုရှိတယ်လို့ ဆိုနိုင်ပါတယ်။ Browser နဲ့ Node တို့ပါ။ နှစ်မျိုးလုံးက JavaScript ကုဒ်တွေကို Run ပေးနိုင်တဲ့ နည်းပညာတွေပါ။ ဒါကြောင့် လက်ရှိလေ့လာထားတဲ့ JavaScript ကုဒ်တွေကို Browser မှာ Run လို့ ရသလို့ Node နဲ့လည်း Run လို့ရနိုင်ပါတယ်။ Browser နဲ့ပက်သက်ရင် ရိုးရိုး JavaScript ထက်ပိုပြီး ထူးခြားသွားတာက Document Object Model (DOM) လို့ခေါ်တဲ့ သဘောသဘာဝ ဖြစ်ပါတယ်။ နောက်တစ်ခန်းမှာ သီးခြားဖော်ပြပါမယ်။

Node နဲ့ပက်သက်ရင် ရိုးရိုး JavaScript ထက်ပိုပြီး ထူးခြားသွားတာကတော့ Module နဲ့ Package Management System ဖြစ်ပါတယ်။ ဒီအကြောင်းတွေကို နည်းနည်းပိုအဆင့်မြင့်သွားတဲ့ JavaScript နည်းပညာတွေအကြောင်း ဖော်ပြထားတဲ့ **React လိုတိုရှင်း၊ API လိုတိုရှင်း** စသည်ဖြင့် သက်ဆိုင်ရာစာအုပ်တွေမှာ ဖော်ပြထားလို့ အချိန်ကျလာတဲ့အခါ၊ လိုအပ်လာတဲ့အခါ ဆက်လက်လေ့လာနိုင်ပါတယ်။ ဒီစာအုပ်မှာ JavaScript အခြေခံသဘောသဘာဝအနေနဲ့ ထည့်သွင်းလေ့လာသင့်တဲ့အတွက် ရွေးထုတ်ဖော်ပြချင်တာက Module ရေးထုံးအကြောင်း ဖြစ်ပါတယ်။

Node ဆိုတဲ့နည်းပညာ စပေါ်ကတည်းက သူ့မှာ Module ရေးထုံးပါဝင်ပြီး ဖြစ်ပါတယ်။ (၂၀၀၈) ခုနှစ်ဝန်းကျင်လောက်ကပါ။ အဲ့ဒီအချိန် ရိုးရိုး JavaScript မှာ Module ရေးထုံးမရှိသေးလို့ သူက ဖြည့်စွက်ထည့်သွင်းပေးထားတာလို့ ဆိုနိုင်ပါတယ်။ အခုတော့ ရိုးရိုး JavaScript မှာလည်း သူ့ကိုယ်ပိုင် Module ရေးထုံးရှိနေပါပြီ။ Node Module ရေးထုံးက အရင်ပေါ်ပြီး ရိုးရိုး JavaScript Module ရေးထုံးက နောက်မှပေါ်တာလို့ ဆိုနိုင်ပါတယ်။ ရေးထုံးမတူကြပါဘူး။ နှစ်မျိုးလုံးကို ဖော်ပြပေးသွားမှာ ဖြစ်ပါတယ်။

ပထမဆုံးအနေနဲ့ Node ကို ဒီဝဘ်ဆိုက်ကနေ Download လုပ်ပြီး Install လုပ်ထားဖို့လိုပါတယ်။

– https://nodejs.org

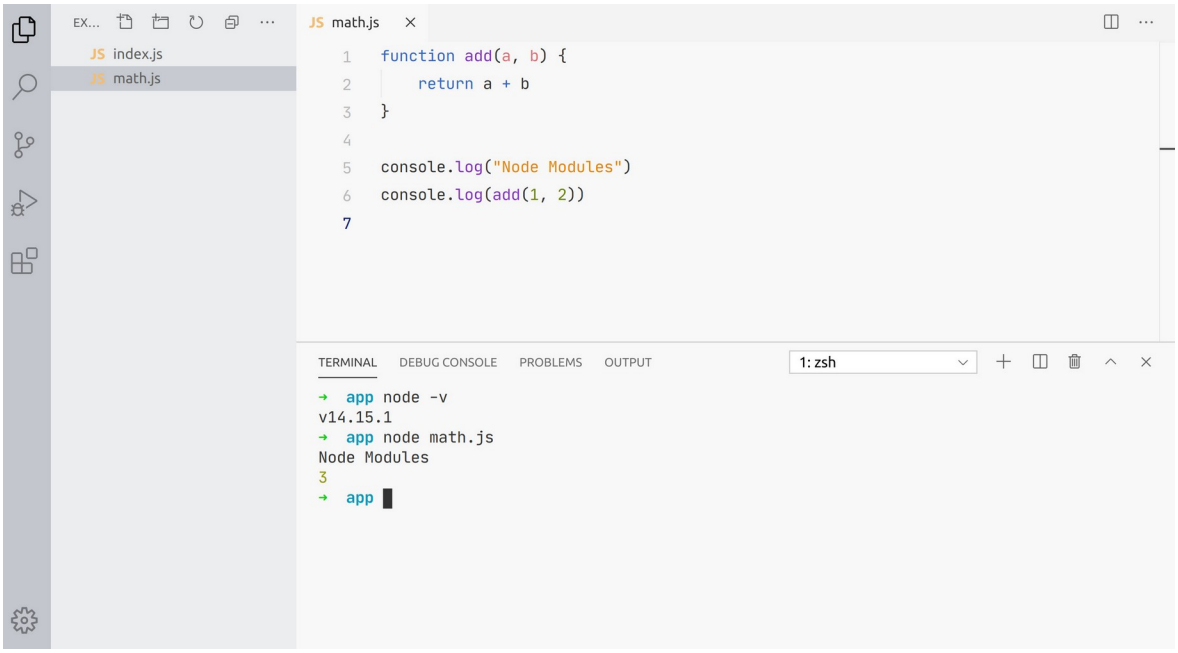


Download လုပ်ပုံလုပ်နည်းနဲ့ Install လုပ်ပုံလုပ်နည်းကိုတော့ ထည့်မပြောတော့ပါဘူး။ ခက်ခဲရှုပ်ထွေးတဲ့ အဆင့်တွေမရှိပါဘူး။ အပေါ်ကပုံမှာ ကြည့်လိုက်ရင် သူဝဘ်ဆိုက်မှာ 14.15.1 LTS နဲ့ 15.4.0 Current ဆိုပြီး နှစ်မျိုးပေးထားတာကို တွေ့ရနိုင်ပါတယ်။ Current ကနောက်ဆုံး Version ဖြစ်ပြီး LTS ကတော့ Long Term Support Version ခေါ် အများသုံးသင့်တဲ့ Version ဖြစ်ပါတယ်။ စမ်းသပ်ရေးသားယုံပဲ မို့လို့ နှစ်မျိုးလုံး ဘာသုံးသုံး အဆင်ပြေပါတယ်။ လက်တွေ့သုံးမှာဆိုရင်တော့ LTS Version ကို ရွေးသင့်ပါတယ်။

Download လုပ်ပြီးရင် Install လုပ်လိုက်ပါ။ ပေးထားတဲ့ Options တွေအတိုင်း သွားယုံပါပဲ။ ဘာမှ ပြင်စရာ ရွေးစရာမလိုပါဘူး။ Install လုပ်ပြီးနောက် Command Prompt (သို့) Terminal ကို ဖွင့်ပြီး အခုလို စမ်းကြည့်နိုင်ပါတယ်။

```
node -v
v14.15.1
```

`node` Command အတွက် `-v` Option ပေးလိုက်တာပါ။ ဒါကြောင့် Install လုပ်ထားတဲ့ Node Version ကို ပြန်ရမှာ ဖြစ်ပါတယ်။ ရေးထားတဲ့ JavaScript ကုဒ်တွေကို Run ဖို့အတွက် `node` Command နောက်မှာ Run လိုတဲ့ File Name ကို ပေးလိုက်ရင် ရပါပြီ။ VS Code ကို အသုံးပြုပြီး ကုဒ်တွေရေးသားတာဆိုရင် VS Code မှာပါတဲ့ Integrated Terminal ကိုသုံးပြီးတော့ Run လို့ရပါတယ်။ Ctrl + ` ကိုနှိပ်လိုက်ရင် အောက်နားမှာ Integrated Terminal ပေါ်လာပါလိမ့်မယ်။ အဲ့ဒီထဲမှာ Run လိုတဲ့ Command တွေကို Run ယုံပါပဲ။



နမူနာပုံကိုကြည့်လိုက်ရင် `math.js` ဆိုတဲ့ဖိုင်ထဲမှာ Function တစ်ခုနဲ့ `console.log()` Statement နှစ်ခုရှိနေပါတယ်။ အဲ့ဒီဖိုင်ကို Terminal ထဲမှာ node နဲ့ Run ထားပါတယ်။

```
node math.js
```

အကယ်၍ ရိုးရိုး Command Prompt (သို့) Terminal ကို သုံးတာဆိုရင် `node` Command အတွက် File Name ပေးတဲ့အခါ ဖိုင်ရဲ့တည်နေရာ Path လမ်းကြောင်းကို မှန်အောင်ပေးဖို့လိုတယ်ဆိုတာကို သတိပြုပါ။ Command Prompt ကို သုံးတဲ့အလေ့အကျင့် မရှိသေးဘူးဆိုရင် VS Code ရဲ့ Integrated Terminal ထဲမှာပဲ စမ်းလိုက်ပါ။ အဲ့ဒါပိုပြီးတော့ စမ်းရလွယ်ပါတယ်။

Module ဆိုတာ သီးခြားဖိုင်တစ်ခုနဲ့ ခွဲခြားရေးသားထားတဲ့ ကုဒ်တွေကို ခေါ်ယူအသုံးပြုနည်း လို့ အလွယ်မှတ်နိုင်ပါတယ်။ ဒီဖွင့်ဆိုချက်က မပြည့်စုံသေးပေမယ့် လိုရင်းက ဒါပါပဲ။

Node Module

ဥပမာ စမ်းကြည့်နိုင်ဖို့ `math.js` ဆိုတဲ့ ဖိုင်တစ်ခုနဲ့ `index.js` ဆိုတဲ့ ဖိုင်တစ်ခု၊ စုစုပေါင်း (၂) ခု တည်ဆောက်လိုက်ပါ။ `math.js` ထဲမှာ ရေးထားတဲ့ လုပ်ဆောင်ချက်ကို `index.js` ကနေ ရယူအသုံးပြုပြီး စမ်းကြည့်ကြပါမယ်။ ဒါကြောင့် `math.js` ထဲမှာ အခုလိုရေးလိုက်ပါ။

JavaScript

```
function add(a, b) {
    return a + b
}

function div(a, b) {
    if(b === 0) return "Cannot divided by zero"
    else return a / b
}

module.exports = div
```

`add()` နဲ့ `div()` ဆိုတဲ့ Function နှစ်ခုရှိပါတယ်။ အောက်ဆုံးမှာရေးထားတဲ့ `module.exports` ရဲ့ အဓိပ္ပါယ်ကတော့ `div()` Function ကို Module အနေနဲ့ ထုတ်ပေးမယ်လို့ ပြောတာပါ။ ဒါကြောင့် ဒီဖိုင်ထဲမှာ ရေးထားတဲ့ ကုဒ်တွေထဲကမှ `div()` ကို တစ်ခြားလိုအပ်တဲ့ဖိုင်ကနေ ချိတ်ဆက်အသုံးပြုလို့ရသွားပါပြီ။ ဒါကိုစမ်းကြည့်နိုင်ဖို့အတွက် `index.js` မှာ အခုလိုရေးနိုင်ပါတယ်။

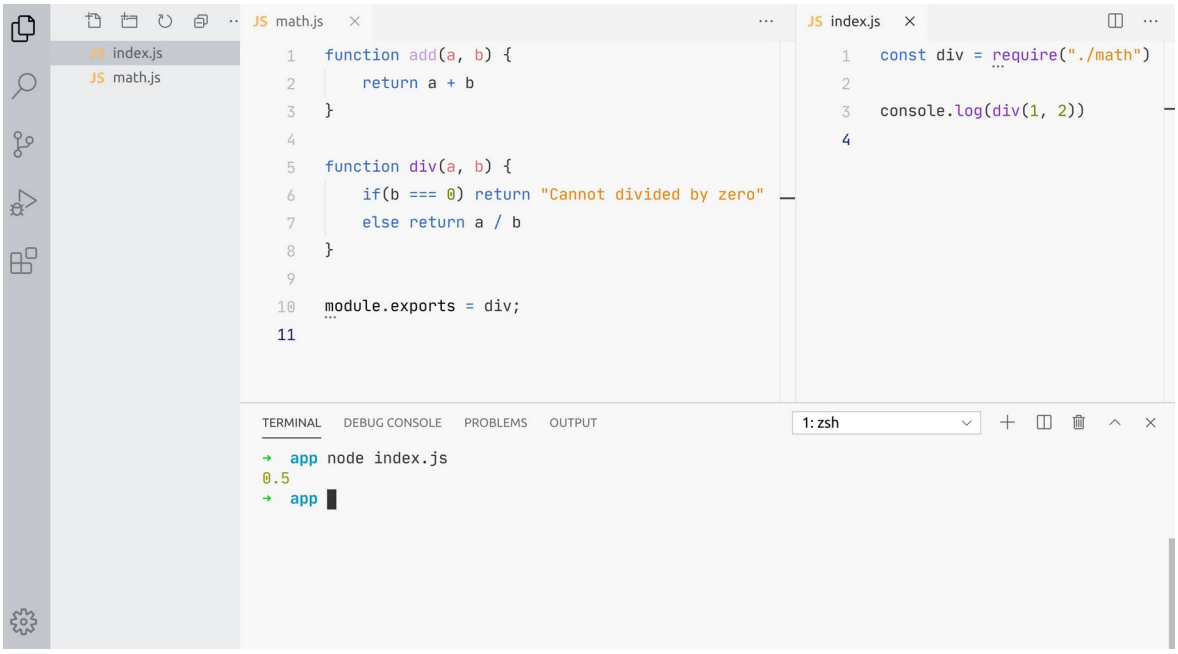
JavaScript

```
const div = require("./math")

console.log(div(1, 2))
```

`require()` Statement ကိုသုံးပြီး Module ကိုချိတ်ဆက်ရတာပါ။ Argument အနေနဲ့ Standard Module တွေ Package Module တွေဆိုရင် Module အမည်ကို ပေးရပြီး၊ ကိုယ့်ဘာသာရေးထားတဲ့ Module ဆိုရင်တော့ Module ဖိုင်ရဲ့ တည်နေရာနဲ့အမည်ကို ပေးရပါတယ်။ နမူနာမှာ ကိုယ့်ဘာသာ

ရေးထားတဲ့ `math.js` ကို အသုံးပြုလိုတာဖြစ်တဲ့အတွက် `./math` လို့ပေးထားပါတယ်။ `./` ရဲ့အဓိပ္ပါယ်က လက်ရှိဖိုဒါလို့ ပြောလိုက်တာပါ။ ဒါကြောင့် Module ဖိုင်ကို လက်ရှိ `index.js` ရှိနေတဲ့ ဖိုဒါထဲမှာပဲ သွားရှာမှာပါ။ ဖိုင်အမည် အပြည့်အစုံက `math.js` ဖြစ်ပေမယ့် နောက်က `.js` Extension ကို ပေးစရာမလိုလို့ ထည့်ပေးမထားတာပါ။

`math.js` ကိုချိတ်ဆက်လိုက်လို့ရလာတဲ့ လုပ်ဆောင်ချက်ကို `div()` ထဲမှာ ထည့်ထားပါတယ်။ Module လုပ်ဆောင်ချက်ကို ကိုယ့်ဘက်က ပြင်စရာအကြောင်းသိပ်မရှိလို့ Constant အနေနဲ့ ကြေညာထားပါတယ်။ အခုလိုပြည့်စုံအောင် ချိတ်ဆက်ပြီးရင်တဲ့ ရရှိထားတဲ့ လုပ်ဆောင်ချက်ကို အသုံးပြုလို့ ရနေပြီမို့လို့ `console.log()` နဲ့ `div()` ရဲ့ရလဒ်ကို ဖော်ပြထားခြင်းပဲဖြစ်ပါတယ်။



ဒီနေရာမှာ သတိပြုရမှာက `math.js` မှာ လုပ်ဆောင်ချက် နှစ်ခုရှိပေမယ့်၊ Module အနေနဲ့ ထုတ်ပေးတော့ တစ်ခုပဲ ပေးထားတာကို သတိပြုပါ။ ဒါကြောင့် ချိတ်ဆက်ရယူသူက ပေးထားတဲ့ တစ်ခုကိုပဲ ယူလို့ရတာပါ။ အကယ်၍ နှစ်ခုသုံးခုပေးချင်ရင်တော့ Object အနေနဲ့ ပြန်ပေးကြလေ့ရှိပါတယ်။ `meth.js` ကို အခုလို ပြင်ကြည့်လိုက်ပါ။

JavaScript

```
const PI = 3.14

function add(a, b) {
    return a + b
}

function div(a, b) {
    if(b === 0) return "Cannot divided by zero"
    else return a / b
}

module.exports = { PI, add, div }
```

Constant တစ်ခုနဲ့ Function နှစ်ခု သုံးခုရှိရာမှာ သုံးခုလုံးကို Module Export အနေနဲ့ ပေးထားတာပါ။ Property Shorthand ရေးထုံးကို သုံးထားတာ သတိပြုပါ။ အပြည့်အစုံရေးရင် ဒီလိုဖြစ်ပါလိမ့်မယ်။

```
module.exports = { PI: PI, add: add, div: div }
```

ဒီလိုပေးထားတဲ့အတွက် ရယူအသုံးပြုတဲ့ `index.js` ကလည်း ပေးထားသမျှ အကုန်ယူလို့ ရသွားပါတယ်။ `index.js` မှာ အခုလိုစမ်းကြည့်နိုင်ပါတယ်။

JavaScript

```
const math = require("./math")

console.log(math.PI)     // 3.14
```

`math.js` က ပြန်ပေးထားတဲ့ Object ကို math Constant နဲ့လက်ခံထားပြီးတော့မှ လိုချင်တဲ့ လိုဆောင်ချက်ကို `math` ရဲ့ Property/Method ကနေတစ်ဆင့် အသုံးပြုလိုက်တာပါ။ Destructuring ရေးထုံးကို သုံးပြီး အခုလိုရေးလို့လည်း ရနိုင်ပါသေးတယ်။

JavaScript

```
const { PI, add } = require("./math")

console.log( add(1, 2) )     // 3
```

လက်ခံရရှိမယ့် Module Object ကို Destructure လုပ်ပြီး သက်ဆိုင်ရာ Constant Variable အနေနဲ့ တစ်ခါထဲ လက်ခံထားလိုက်တာပါ။

ဒီ ရေးထုံးကို CommonJS Module ရေးထုံးလို့ ခေါ်ပါတယ်။ CommonJS ဆိုတဲ့အမည်နဲ့ JavaScript Library တစ်ခုရှိခဲ့ဖူးပြီး အဲ့ဒီ Library က JavaScript မှာမရှိသေးတဲ့ Module ရေးထုံးကို တီထွင်ပေးထားတာပါ။ Node က CommonJS ကို နမူနာယူ အသုံးပြုထားလို့ Node ရဲ့ Module ရေးထုံးကို CommonJS Module ရေးထုံးလို့ခေါ်တာပါ။

ES Module

JavaScript ဟာ ECMA လို့ခေါ်တဲ့ အဖွဲ့အစည်းက စံချိန်စံညွှန်းတစ်ခုအဖြစ် သတ်မှတ်ထားတဲ့ ECMAScript ကို အခြေခံ တီထွင်ထားတာပါ။ မူအရ ECMAScript ဆိုတာ စံသတ်မှတ်ချက်ဖြစ်ပြီး ဒီစံသတ်မှတ်ချက်နဲ့အညီ Programming Language တွေ တီထွင်လို့ရတယ်ဆိုတဲ့ သဘောမျိုးပါ။ ActionScript, JScript စသည်ဖြင့် ECMAScript ကို အခြေခံထားတဲ့ တစ်ခြား Language တွေ ရှိသေးပေမယ့် အသုံးတွင်ကျယ်ခြင်းတော့ မရှိတော့ပါဘူး။

ECMAScript ကို အဆက်မပြတ် Version အဆင့်ဆင့် မြှင့်တင်နေပါတယ်။ ဒါကြောင့် Version နံပါတ်နဲ့တွဲပြီး ES5, ES6, ES7 စသည်ဖြင့် အတိုကောက် ခေါ်ကြလေ့ ရှိပါတယ်။ သက်ဆိုင်ရာ Version ကို ထုတ်လုပ်တဲ့ ခုနှစ်နဲ့တွဲပြီးတော့ ECMAScript 2015, ECMAScript 2019 စသည်ဖြင့်လည်း ခေါ်ကြပါသေးတယ်။ အဲ့ဒီထဲမှာ အရေးအပါဆုံးလို့ ဆိုနိုင်တာကတော့ ES6 (ECMAScript 2015) ဖြစ်ပါတယ်။ `let, const` တို့လို့ ရေးထုံးမျိုးတွေ၊ Rest Parameters, Destructuring, Arrow Function စတဲ့ ထူးခြားအသုံးဝင်တဲ့ လုပ်ဆောင်ချက်တွေဟာ ES6 မှာ စတင်ပြီး ပါဝင်လာတဲ့အတွက်ကြောင့်ပါ။ သူနောက်ပိုင်း ES7, ES8 စသည်ဖြင့်ထွက်ပေါ်လာခဲ့တာ အခုဒီစာကို ရေးနေတဲ့အချိန်မှာ ES11 ကို ရောက်ရှိနေပါပြီ။ နောက်ပိုင်း Version တွေမှာလည်း အပြောင်းအလဲနဲ့ အဆင့်မြှင့်တင်မှုတွေ ဆက်တိုက်ပါဝင်နေပေမယ့် ES6 မှာပါဝင်ခဲ့တဲ့ အပြောင်းအလဲတွေက JavaScript Community တစ်ခုလုံးကို ကိုင်လှုပ်ပြောင်းလဲသွားစေခဲ့လို့ ES6 က အထင်ရှားဆုံးနဲ့ လူပြောအများဆုံး ဖြစ်ခဲ့ပါတယ်။

ES6 မှာပါဝင်ခဲ့တဲ့ လုပ်ဆောင်ချက်တွေထဲမှာ Module လည်း အပါအဝင်ဖြစ်ပါတယ်။ မူလက JavaScript မှာ Module လုပ်ဆောင်ချက်မရှိပေမယ့် ES6 ထွက်ပေါ်လာပြီးနောက်မှာတော့ Module လုပ်ဆောင်ချက်

Official ပါဝင်သွားခဲ့ပါပြီ။ ဒီလိုပါဝင်သွားတဲ့ Module စနစ်ဟာ Node မှာသုံးနေတဲ့ Module စနစ်နဲ့ မတူတဲ့အတွက် ကွဲပြားသွားအောင် ES Module လို့ သုံးနှုန်းပြောဆိုကြလေ့ ရှိပါတယ်။

အခုတော့ Node ကလည်း သူ့မူလ CommonJS Module ရေးထုံးကိုသာမက ES Module ရေးထုံးကိုပါ ပံ့ပိုးပေးလာပါပြီ။ နှစ်မျိုးရေးလို့ရတယ်ဆိုတဲ့ သဘောပါ။ ဒါပေမယ့် စမ်းသပ်အဆင့်မှာပဲ ရှိသေးလို့ ကန့်သတ်ချက်နဲ့ အခက်အခဲလေးတစ်ချို့တော့ ရှိပါတယ်။ အခက်အခဲကတော့ မူလ CommonJS Module တွေနဲ့ ES Module တွေကို တွဲသုံးရ ခက်ခဲခြင်း ဖြစ်ပါတယ်။ အကန့်အသတ်အနေနဲ့ကတော့ လက်ရှိ ဒီစာရေးနေချိန်ထိ CommonJS Module က Default ဖြစ်ပြီး ES Module ကိုသုံးလိုရင် File Extension ကို `mjs` လို့ပေးပြီး သုံးရမှာဖြစ်ပါတယ်။

ဒါကြောင့် စောစောက စမ်းခဲ့တဲ့ Module လုပ်ဆောင်ချက်ကိုပဲ ES Module အနေနဲ့ ပြန်လည်စမ်းသပ်နိုင်ဖို့အတွက် `math.mjs` နဲ့ `index.mjs` ဆိုတဲ့ ဖိုင်နှစ်ခုတည်ဆောက်လိုက်ပါ။ ပြီးတဲ့အခါ `math.mjs` ဖိုင်ထဲမှာ ဒီကုဒ်ကို ရေးပြီး လေ့လာကြည့်လိုက်ပါ။

JavaScript

```
const PI = 3.14

export default function add(a, b) {
    return a + b
}
```

`add()` Function ကို ကြေညာစဉ်မှာ `export default` ရေးထုံးကိုသုံးပြီး Module လုပ်ဆောင်ချက်အနေနဲ့ ပေးထားတာ ဖြစ်ပါတယ်။ အဲ့ဒီလို Function ကြေညာစဉ်မှာ မပေးဘဲ နောက်မှ Export လုပ်ပေးလို့လည်း ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
const PI = 3.14
function add(a, b) {
    return a + b
}

export default add
```

ရလဒ်အတူတူပါပဲ။ Function ကြေညာစဉ်မှာ တစ်ခါထဲ Export မလုပ်တော့ဘဲ နောက်ဆုံးကျတော့မှ Export လုပ်ပေးလိုက်တာပါ။ ဒီလို Export လုပ်ပေးထားတဲ့ Module ကို ပြန်လည်အသုံးပြု စမ်းသပ်နိုင်ဖို့ အတွက် `index.mjs` ဖိုင်မှာ အခုလို ရေးပြီး စမ်းကြည့်နိုင်ပါတယ်။

JavaScript

```
import add from "./math.mjs"

console.log( add(1, 2) )     // 3
```

`require()` Statement ကို မသုံးတော့ပါဘူး။ `import` Statement ကိုသုံးပါတယ်။ `require()` မှာလို ဝိုက်ကွင်းအဖွင့်အပိတ် မသုံးဘဲ `from` Keyword နောက်ကနေ File Name လိုက်ရပါတယ်။ ရိုးရိုး `js` Extension ရှိတဲ့ဖိုင်အတွက် Extension ထည့်မပေးရင် ရပေမယ့် `mjs` Extension ဖိုင်အတွက် Extension ကိုပါ အပြည့်အစုံ ထည့်ပေးရပါတယ်။ ဒီဖိုင်ကို `node` နဲ့ Run ကြည့်ရင် လိုချင်တဲ့ရလဒ် ရတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

```
node index.mjs
```

ပေးရတဲ့ `index` ဖိုင်ရဲ့ Extension က `mjs` ဆိုတာကို မမေ့ပါနဲ့။ မေ့တတ်ပါတယ်။ မှားတတ်ပါတယ်။ လုပ်ဆောင်ချက်တစ်ခုထက်ပိုပြီး Export ပေးချင်ရင်လည်း အခုလို ပေးနိုင်ပါတယ်။

JavaScript

```
export const PI = 3.14

export function add(a, b) {
    return a + b
}

export function div(a, b) {
    if(b === 0) return "Cannot divided by zero"
    else return a / b
}
```

ဒါမှမဟုတ် ကြေညာစဉ်မှာ တစ်ခါထဲ Export မလုပ်ဘဲ နောက်မှလုပ်လို့လည်း ရပါတယ်။

```
const PI = 3.14

function add(a, b) {
    return a + b
}

function div(a, b) {
    if(b === 0) return "Cannot divided by zero"
    else return a / b
}

export { PI, add, div }
```

ဒီလို လုပ်ဆောင်ချက်တစ်ခုထက်ပိုပြီး ပေးထားတဲ့ လုပ်ဆောင်ချက်တွေကို Destructuring ပုံစံရေးဟန်မျိုးနဲ့ Import ပြန်လုပ်ပြီး သုံးလို့ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
import { PI, div } from "./math.mjs"

console.log( div(1, 2) )     // 0.5
```

နောက်ပိုင်းမှာတော့ နေရာတိုင်းမှာ ES Module ရေးထုံးကို အသုံးပြုကြပါလိမ့်မယ်။ လက်ရှိအချိန်ထိတော့ Node ပရောဂျက်တွေမှာ CommonJS Module ရေးထုံးကို သူနေရာနဲ့သူ ဆက်သုံးကြရဦးမှာပါ။

ဒီ Module လုပ်ဆောင်ချက်ဟာ အတော်လေး အရေးပါတဲ့ လုပ်ဆောင်ချက်တစ်ခုပါ။ အရင်ကဆိုရင် JavaScript ကုဒ်ဖိုင်တွေ ပူးတွဲအသုံးပြုချင်ရင် HTML Document ထဲမှာ `<script>` Element တွေနဲ့ တန်းစီချိတ်ဆက်ပြီး သုံးခဲ့ကြရပါတယ်။ JavaScript ကုဒ်ကနေ တစ်ခြား JavaScript ကုဒ်ဖိုင်ကို ချိတ်ဆက် အသုံးပြုရေးသားလို့ လုံးဝမရခဲ့တာပါ။

အခုတော့ Module ရေးထုံးတွေရဲ့ အကူအညီနဲ့ ကိုယ်တိုင်ရေးသားတဲ့ ကုဒ်တွေကို စနစ်ကျအောင် ခွဲခြားချိတ်ဆက်ရေးသားနိုင်သွားမှာ ဖြစ်သလို၊ အများရေးသား ဖြန့်ဝေထားတဲ့ Library တွေ Module တွေ Framework တွေကို ရယူပေါင်းစပ် အသုံးပြုရတာ အများကြီးပိုအဆင်ပြေသွားပြီဖြစ်ပါတယ်။

Practical Sample

ဒီနေရာမှာလည်း လက်တွေ့နမူနာလေးတစ်ခုလောက် ထည့်ပေးချင်ပါတယ်။ ပြီးခဲ့တဲ့ Promise အခန်းမှာ Covid-19 နဲ့ပက်သက်တဲ့ အချက်အလက်တွေ ရယူပုံကို ဖော်ပြခဲ့ပါတယ်။ Browser ထဲမှာ အလုပ်လုပ်တဲ့ ကုဒ်နဲ့ ဖော်ပြခဲ့တာပါ။ Node နဲ့လည်း အဲ့ဒီအချက်အလက်တွေကို ရယူကြည့်ချင်ပါတယ်။

Node မှာ `fetch()` လုပ်ဆောင်ချက် ပါဝင်ခြင်းမရှိပါဘူး။ အသုံးပြုလိုရင် Third-party Module အနေနဲ့ ထပ်ထည့်ပေးရပါတယ်။ ဒါပေမယ့် Node မှာ အင်တာနက်ပေါ်က အချက်အလက်တွေကို ရယူအလုပ်လုပ်ပေးနိုင်တဲ့ HTTP နဲ့ HTTPS Module တို့ ပါဝင်ပါတယ်။ ဒါကြောင့် `fetch()` Module ကို ထပ်ထည့်မနေတော့ဘဲ သူ့မှာ ပါပြီးသား HTTPS Module ကို အသုံးပြုသွားပါမယ်။ ပြီးတဲ့အခါ Node ရဲ့ File System Module ကိုသုံးပြီး ရလာတဲ့အချက်အလက်တွေကို ဖိုင်တစ်ခုအနေနဲ့ သိမ်းပေးလိုက်မှာပါ ဖြစ်ပါတယ်။ `get-covid-info.js` ဆိုတဲ့ဖိုင်တစ်ခုနဲ့ ဒီကုဒ်ကို ရေးပြီးစမ်းကြည့်ပါ။

JavaScript

```
const fs = require("fs")
const https = require("https")

https.get("https://api.covid19api.com/summary", res => {
    let data = ''

    res.on('data', chunk => {
        data += chunk
    })

    res.on('end', () => {
        fs.writeFile('covid-info.json', data, () => {
            console.log("Save to file: Completed")
        })
    })
})
```

ပထမဆုံး `https` Module နဲ့ `fs` Module တို့ကို Import လုပ်ပေးထားပါတယ်။ Standard Module တွေမို့လို့ Path လမ်းကြောင်းတွေဘာတွေ ပေးစရာမလိုဘဲ Module အမည်နဲ့တင် ချိတ်လို့ရတာကို သတိပြုပါ။ `https` ရဲ့ `get()` Method အတွက် Argument နှစ်ခုပေးရပါတယ်။ ပထမတစ်ခုက Data ရယူလိုတဲ့ URL ဖြစ်ပြီး ဒုတိယ Argument ကတော့ Callback Function ပါ။ Data ရယူလို့ ပြီးတဲ့အခါ Callback Function အလုပ်လုပ်သွားမှာပါ။

Callback Function ထဲမှာ `res.on('data')` နဲ့ Data တွေလက်ခံရရှိရင် ဘာလုပ်ရမလဲ သတ်မှတ်ပေးရပါတယ်။ သတိပြုရမှာက၊ `get()` Method က Data တွေကို တစ်ခါထဲ ရှိသမျှအကုန်ယူပေးတာမဟုတ်ပါဘူး။ Data တွေကို အပိုင်းလိုက်၊ အပိုင်းလိုက်၊ ပိုင်းပြီး ရယူပေးမှာဖြစ်လို့ တစ်ပိုင်းရရင်တစ်ခါ += နဲ့ ကြိုကြေညာထားတဲ့ Variable ထဲကို ထပ်တိုးပြီး ထည့်ထည့်ပေးသွားရတာပါ။ ပြီးတဲ့ခါ `res.on('end')` နဲ့ Data တွေယူလို့ ကုန်ပြီဆိုတော့မှ နောက်ဆုံးရလဒ်ကို `covid-info.json` ဆိုတဲ့ ဖိုင်ထဲမှာ သိမ်းပေးလိုက်တာ ဖြစ်ပါတယ်။

ဖိုင်အနေနဲ့ သိမ်းဖို့အတွက် `fs.writeFile()` Method ကိုသုံးထားပြီး သိမ်းလိုတဲ့ File Name နဲ့ အထဲမှာ ထည့်ပေးရမယ့် Data တို့ကို Argument အနေနဲ့ပေးရပါတယ်။ နောက်ဆုံးကနေ သိမ်းပြီးတဲ့အခါ လုပ်ရမယ့် Callback Function ကိုပေးရပါတယ်။ ဒီနည်းနဲ့ HTTPS ကိုသုံးပြီး Data တွေရယူတယ်၊ ရလာတဲ့ Data တွေကို ဖိုင်တစ်ခုနဲ့ သိမ်းလိုက်တယ်ဆိုတဲ့ လုပ်ဆောင်ချက် ရသွားပါတယ်။ စမ်းကြည့်လို့ရပါတယ်။

```
node get-covid-info.js
```

လက်တွေ့စမ်းကြည့်တဲ့အခါ Server ဘက်က Down နေတာတို့၊ Data ရယူတာ အကြိမ်ရေပြည့်သွားလို့ ထပ်မရတော့တာတို့၊ File Permission ပြဿနာရှိနေတာတို့လို ပြဿနာလေးတွေ ရှိနိုင်ပါတယ်။ ဒါတွေကို အဖြေရှာပုံ ရှာနည်းတွေထည့်ပြောမထားလို့ စမ်းကြည့်ရတာ အဆင်မပြေရင် စိတ်မပျက်ပါနဲ့။ ရိုးရိုးနမူနာတွေချည်းပဲ စမ်းနေရတာ ပျင်းစရာဖြစ်နေမှာစိုးလို့သာ လက်တွေ့ကျတဲ့ နမူနာတစ်ခုကို ထည့်ဖော်ပြလိုက်တာပါ။ တစ်ကယ်တော့ ဒီကုဒ်ကလည်း နည်းနည်းအဆင့်ကျော်ပြီး ဖော်ပြလိုက်တဲ့ကုဒ်လို့ ဆိုနိုင်ပါတယ်။

လက်စနဲ့ သိမ်းထားတဲ့ဖိုင်ကနေ ပြန်ယူပုံကိုလည်း တစ်ခါထဲပြောလိုက်ချင်ပါတယ်။ `show-covid-info.js` အမည်နဲ့ အခုလိုရေးပြီး စမ်းကြည့်နိုင်ပါတယ်။

JavaScript

```
const fs = require("fs")

fs.readFile('covid-info.json', (err, data) => {
    if(err) {
        console.log(err)
    } else {
        const result = JSON.parse(data)
        const global = result.Global
        const allCountries = result.Countries
        const myanmar = allCountries.find(c => c.Country==="Myanmar")

        console.log("Global:", global, "Myanmar:", myanmar)
    }
})
```

`fs.readFile()` Method ကိုသုံးပြီး စောစောကသိမ်းထားတဲ့ဖိုင်ကို ဖတ်ယူလိုက်တာပါ။ Argument အနေနဲ့ File Name နဲ့ Callback တို့ကိုပေးရပါတယ်။ Callback Function က `err` နဲ့ `data` ဆိုတဲ့ Argument နှစ်ခုကို လက်ခံပါတယ်။ အကြောင်းအမျိုးမျိုးကြောင့် ဖိုင်ကိုဖတ်လို့မရရင် `err` ထဲမှာ Error Message ရှိနေမှာဖြစ်ပြီး ဖတ်လို့ရရင် `data` ထဲမှာ File Content တွေ ရှိနေမှာပါ။

ဒါကြောင့် `err` ရှိမရှိစစ်ပြီး `err` မရှိတော့မှ `data` ကို `JSON.parse()` နဲ့ JavaScript Object ပြောင်းပေးလိုက်တာပါ။ ကျန်အဆင့်တွေရဲ့ အဓိပ္ပါယ်ကိုတော့ ရှေ့မှာတစ်ခါ ရေးခဲ့ဖူးပြီးသားမို့လို့ ထပ်ပြောစရာ မလိုတော့ဘူးလို့ထင်ပါတယ်။ လက်တွေ့စမ်းကြည့်လိုက်ပါ။

```
node show-covid-info.js
```

Node မှာအခုလို File System နဲ့ Network ပိုင်းစီမံတဲ့ Module တွေအပါအဝင် တစ်ခြား အသုံးဝင်တဲ့ Standard Module တွေ တစ်ခါထဲ ပါဝင်ပါသေးတယ်။ စိတ်ဝင်စားရင် ဒီမှာ လေ့လာကြည့်နိုင်ပါတယ်။

https://nodejs.org/api

အခန်း (၁၁) – Document Object Model – DOM

Document Object Model (DOM) ဆိုတာ JavaScript ကို အသုံးပြုပြီး HTML Document တွေကို စီမံလို့ရအောင် တီထွင်ထားတဲ့ နည်းပညာ ဖြစ်ပါတယ်။ JavaScript ကို မူလအစကတည်းက Browser ထဲမှာ အလုပ်လုပ်တဲ့ Client-side Language တစ်ခုအနေနဲ့ တီထွင်ခဲ့ကြတာမို့လို့ JavaScript နဲ့ DOM ကို ခွဲလို့မရပါဘူး။ DOM ဆိုတာ JavaScript ရဲ့ အရေးအကြီးဆုံး အခန်းကဏ္ဍတစ်ခု ဖြစ်ပါတယ်။

ကနေ့ခေတ်မှာ DOM Manipulation ခေါ် HTML Element တွေစီမံတဲ့အလုပ်တွေ လုပ်ဖို့အတွက် ရိုးရိုး JavaScript ကုဒ်တွေ အစအဆုံး ကိုယ်တိုင်ရေးဖို့ လိုချင်မှ လိုပါလိမ့်မယ်။ jQuery တို့ React တို့ Vuejs တို့ လို နည်းပညာတွေရဲ့ အကူအညီနဲ့ တစ်ဆင့်ခံ ဆောင်ရွက်ကြဖို့ များပါတယ်။

ဒီနေရာမှာ အရေးကြီးတာလေးတစ်ခုကို သတိပြုပါ။ လေ့လာနည်းမမှန်ရင် အမြဲမပြတ်ပြောင်းလဲထွက်ပေါ်နေတဲ့ နည်းပညာသစ်တွေ နောက်ကို အမြဲမပြတ် လိုက်နေရတာ ရင်မောစရာကြီး ဖြစ်နေပါလိမ့်မယ်။ jQuery ပေါ်လာတယ်၊ ကောင်းတယ်ဆိုလို့ jQuery လေ့လာလိုက်တယ်။ နောက်တော့မှ React က ပိုကောင်းတယ် ဆိုလို့ React ကို အစအဆုံး ပြန်ပြောင်း လေ့လာရပြန်တယ်။ စသည်ဖြင့် နည်းပညာသစ်တစ်ခုပေါ်တိုင်း တစ်ခါ အစအဆုံး ကြက်တူရွေးနှုတ်တိုက် ဆိုသလို ၏-သည် မရွေး လေ့လာနေရတာ မလွယ်ပါဘူး။ ဒါပေမယ့် သိသင့်သိထိုက်တဲ့ အခြေခံတွေ ကြည်ကြည်ထားသူအတွက် ထပ်ဆင့်နည်းပညာတွေ ဘယ်လောက်ပြောင်းပြောင်း ရင်းမြစ်ကိုနားလည်ပြီး ဖြစ်လို့ တစ်ခုကနေ နောက်တစ်ခုကို အလွယ်တစ်ကူ ကူးပြောင်း လေ့လာ အသုံးချနိုင်ပါလိမ့်မယ်။ နည်းပညာသစ်မို့လို့ အသစ်ထပ် လေ့လာရတာကို ရှောင်လို့မရနိုင်ပေမယ့် အများကြီး လွယ်ကူမြန်ဆန်သွားမှာဖြစ်သလို အဲ့ဒီလို လေ့လာရလို့လည်း ရင်မောစရာ မဖြစ်တော့တဲ့အပြင် စိတ်ဝင်စား ပျော်ရွှင်စရာတောင် ဖြစ်နေနိုင်ပါသေးတယ်။

ဒါကြောင့် DOM နဲ့ပက်သက်တဲ့ အကြောင်းအရာတွေကို ဖော်ပြတဲ့အခါ၊ Browser ကိုအခြေခံဖန်တီးတဲ့ ပရောဂျက်အားလုံးရဲ့ အခြေခံရင်းမြစ်တစ်ခုအနေနဲ့ မဖြစ်မနေသိထားသင့်တဲ့အတွက် ဖော်ပြခြင်းဖြစ်ပါတယ်။ တစ်ချို့ပရောဂျက်တွေမှာ ဒီနည်းတွေကို တိုက်ရိုက် သုံးဖြစ်ပါလိမ့်မယ်။ တစ်ချို့ပရောဂျက်တွေမှာ တစ်ဆင့်ခံ Framework တစ်ခုခုနဲ့ သုံးဖြစ်ပါလိမ့်မယ်။ သုံးခြင်း/မသုံးခြင်း က အဓိကမဟုတ်ပါဘူး။ အဓိကကတော့ အခြေခံရင်းမြစ် နည်းပညာတစ်ခုအနေနဲ့ သိထားခြင်းအားဖြင့် ရေရှည်မှာ အကျိုးများမှာမို့လို့ လေ့လာခြင်းဖြစ်ပါတယ်။

ရှေ့ဆက်ဖော်ပြမယ့် အကြောင်းအရာတွေဟာ၊ စာဖတ်သူက HTML/CSS အခြေခံတွေကို သိရှိတတ်ကျွမ်းပြီး ဖြစ်တယ်လို့ သဘောထားပြီး ဆက်လက်ဖော်ပြ သွားတော့မှာပါ။ ဒါကြောင့် HTML/CSS မသိရင် ဒီအတိုင်းရှေ့ဆက်ဖတ်လို့ အဆင်ပြေမှာ မဟုတ်ပါဘူး။ အရင်လေ့လာပြီးမှ ပြန်လာဖတ်တာက ပိုထိရောက်ပါလိမ့်မယ်။ လိုအပ်ရင် **Bootstrap လိုတို့ရှင်း** စာအုပ်မှာ HTML/CSS အခြေခံတွေကို လေ့လာနိုင်ပါတယ်။ HTML/CSS သိပြီးသားသူတွေအတွက်တော့ ဆက်လေ့လာလိုက်ရင် အဆင်ပြေသွားမှာပါ။

Document Object Model

Node မှာ Standard Module တွေရှိသလိုပဲ Browser မှာလည်း Standard Object တွေရှိပါတယ်။ အဓိက အကျဆုံး သုံးခုကတော့ `navigator`, `window` နဲ့ `document` ဆိုတဲ့ Object သုံးခုပါ။

`navigator` Object မှာ Browser ကြီးတစ်ခုလုံးနဲ့ သက်ဆိုင်တဲ့ Property တွေ Method တွေ၊ System နဲ့ပက်သက်တဲ့ Property တွေ Method တွေ ပါဝင်ပါတယ်။ ဥပမာ Browser Version ကိုသိချင်ရင် User Agent Property ကို သုံးနိုင်တယ်။ `navigator.userAgent` ဆိုရင်ရပါတယ်။ လက်ရှိ အင်တာနက်ဆက်သွယ်ထားခြင်း ရှိမရှိ၊ ဘက်ထရီ ရှိမရှိ၊ အခဲ့ဒါမျိုးတွေ သိချင်ရင်လည်း `navigator` Object ကနေ တစ်ဆင့် ရယူအသုံးပြုနိုင်ပါတယ်။ ဒီနေရာမှာ အဓိကထားပြောမယ့်အကြောင်းအရာတွေတော့ မဟုတ်သေးပါဘူး။ နောင်လိုအပ်လာရင် `navigator` Object နဲ့ပက်သက်တဲ့ လုပ်ဆောင်ချက်တွေကို ဒီလိပ်စာမှာ လေ့လာလို့ ရနိုင်ပါတယ်။

https://developer.mozilla.org/en-US/docs/Web/API/Navigator

`window` Object မှာတော့ လက်ရှိဖွင့်ထားတဲ့ Tab နဲ့ သက်ဆိုင်တဲ့ Property တွေ Method တွေ ပါဝင်ပါ

တယ်။ အရင်က Tab Browsing ဆိုတာ မရှိပါဘူး။ Web Page အသစ်တစ်ခုကို ထပ်ဖွင့်ချင်ရင် Browser Window အသစ်တစ်ခုနဲ့ပဲ ဖွင့်ရတာပါ။ Window ပဲ ရှိခဲ့လို့ Object အမည်က `tab` Object မဟုတ်ဘဲ `window` Object ဖြစ်နေတာပါ။ ဒါကြောင့် `window` Object ဆိုတာ လက်ရှိဖွင့်ထားတဲ့ Tab နဲ့ သက်ဆိုင်တဲ့ Object ဖြစ်တယ်လို့ မှတ်ထားရမှာပါ။ Window ရဲ့ Size တို့၊ Cursor Position တို့ လက်ရှိ URL တို့လို တန်ဖိုးတွေဟာ Property အနေနဲ့ ရှိနေပါတယ်။ ဥပမာ `window.location.href` ဆိုရင် လက်ရှိဖွင့်ထားတဲ့ URL ကို ရပါတယ်။ ဒါလည်းပဲ ဒီနေရာမှာ အဓိကထည့်သွင်းလေ့လာမယ့် လုပ်ဆောင်ချက် မဟုတ်သေးပါဘူး။ နောက်လိုအပ်ရင် ဒီလိပ်စာမှာ လေ့လာနိုင်ပါတယ်။

https://developer.mozilla.org/en-US/docs/Web/API/Window

ထူးခြားချက်တစ်ခုကိုတော့ ထည့်ပြောပြချင်ပါတယ်။ Browser တွေက `window` Object ရဲ့ Property တွေ Method တွေကို အသုံးပြုလိုရင် `window` Object တစ်ဆင့်မခံဘဲ တိုက်ရိုက်အသုံးပြုလို့ရအောင် စီစဉ်ပေးထားပါတယ်။ စောစောက နမူနာပြောခဲ့တဲ့ `window.location.href` ကို တစ်ကယ်သုံးရင် `location.href` လို့ သုံးနိုင်ပါတယ်။ ရှေ့က `window` မထည့်လည်းရပါတယ်။ ရလဒ်အတူတူပဲ ဖြစ်မှာပါ။ နောက်ဥပမာ တစ်ခုအနေနဲ့ `window.alert` လို့ခေါ်တဲ့ Message Box တစ်ခုဖော်ပြပေးနိုင်တဲ့ လုပ်ဆောင်ချက် ရှိပါတယ်။ လက်တွေ့သုံးတဲ့အခါ နှစ်မျိုးသုံးလို့ရပါတယ်။ ဒီလိုပါ –

JavaScript

```
window.alert("Hello DOM")
alert("Hello DOM")
```

ဒီထူးခြားချက်လေးကို မှတ်ထားပါ။ တစ်ကယ်တော့ Browser JavaScript မှာ Variable တွေ Function တွေကြေညာလိုက်ရင် ကြေညာလိုက်တဲ့ Variable တွေ Function တွေဟာ `window` Object ရဲ့ Property တွေ Method တွေဖြစ်သွားတာပါ။

JavaScript

```
var name = "Alice"
console.log(name)     // Alice
console.log(window.name)     // Alice

function add(a, b) {
    return a + b
}

console.log(add(1, 2))                // 3
console.log(window.add(1, 2))         // 3
```

ဒါကြောင့် မူလရှိနေတဲ့ Global Function တွေရော၊ ကိုယ်ကြေညာလိုက်တဲ့ Global Variable တွေ Function တွေရာ အားလုံးက `window` ရဲ့ Property တွေ Method တွေ ဖြစ်ကြတယ်လို့ မှတ်ရမှာပါ။

`document` Object ကတော့ အရေးအကြီးဆုံးပါပဲ။ အခုဒီနေရာမှာ `document` Object နဲ့ပက်သက်တဲ့ အကြောင်းအရာတွေကို အဓိကထား လေ့လာချင်တာပါ။ `document` Object ဟာ လက်ရှိ HTML Document ကို ရည်ညွှန်းပြီး Document ထဲမှာပါတဲ့ Element နဲ့ Content တွေကို စီမံနိုင်တဲ့ Property တွေ Method တွေ စုစည်း ပါဝင်ပါတယ်။ နမူနာအနေနဲ့ ဒီ HTML Structure ကိုကြည့်ပါ။

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>Heading</h1>
    <p>
        Some paragraph content
        <a href="#">A Link</a>
    </p>
</body>
</html>
```

ပါဝင်တဲ့ Element တွေ Content တွေကို DOM နည်းပညာက Object Tree အဖြစ်ပြောင်းပြီး စီမံသွားမှာ ဖြစ်ပါတယ်။ ဖွဲ့စည်းပုံက ဒီလိုဖြစ်မှာပါ –

```
doctype
html
├── head
│   └── title
└── body
    ├── h1
    └── p
        ├── Text
        └── a
```

ဒါဟာ Document Object Model (DOM) ပါပဲ။ HTML Element တွေကို အပြန်အလှန် ဆက်စပ်နေတဲ့ Object Tree တစ်ခုကဲ့သို့ မှတ်ယူပြီး စီမံပေးနိုင်တဲ့ စနစ်ပါ။ HTML Element တွေဟာ Object Tree ကြီး တစ်ခုထဲက Node Object တွေဖြစ်သွားကြတာပါ။ အခေါ်အဝေါ် (၆) မျိုးမှတ်ထားဖို့ လိုပါလိမ့်မယ်။

- Root Node
- Element Node
- Text Node
- Child Node
- Parent Node
- Decedent Node

– နမူနာအရ `doctype` နဲ့ `html` တို့ဟာ ပင်မ **Root Node** တွေပါ။ သူတို့ကရင်းမြစ်ပါပဲ။ သူတို့အပေါ်မှာ တစ်ခြား Node တွေမရှိတော့ပါဘူး။

– ဒီ Object Tree မှာ ပါဝင်တဲ့ Node တစ်ခုချင်းစီကို **Element Node** လို့ခေါ်ပါတယ်။

– `html` Node အောက်မှာ `head` နဲ့ `body` ဆိုတဲ့ **Child Node** တွေ ရှိကြပါတယ်။ အပြန်အလှန်အားဖြင့် `head` Node ရဲ့ **Parent Node** က `html` Node ဖြစ်တယ်လို့ ပြောရပါတယ်။

– လက်ရှိ `body` Node မှာ `h1` နဲ့ `p` ဆိုတဲ့ Child Node နှစ်ခု ထပ်ဆင့်ရှိပါသေးတယ်။ `h1` နဲ့ `p` တို့ဟာ `html` ရဲ့ Child Node တွေ မဟုတ်ကြပါဘူး။ ဒါပေမယ့် သူတို့ကို `html` ရဲ့ **Decedent Node** လို့ခေါ်ပါတယ်။ `html` ရဲ့ Child မဟုတ်ပေမယ့် သူအောက်မှာပဲ ထပ်ဆင့်ရှိနေတဲ့ Node တွေမို့လို့ပါ။

– အပေါ်က နမူနာ HTML Structure ကို ပြန်ကြည့်ပါ။ `<p>` Element အတွင်းမှာ စာတစ်ချို့ပါဝင်ပါတယ်။ အဲ့ဒီလိုစာတွေကို **Text Node** လို့ခေါ်တာပါ။ ဒါကြောင့် `p` ရဲ့ Child Node အနေနဲ့ Text Node တစ်ခုနဲ့ `a` Element Node တို့ ရှိနေကြတာပါ။

Node တစ်ခုချင်းစီကို Object တွေလို့ မှတ်ယူနိုင်ပါတယ်။ ပြည့်ပြည့်စုံစုံ ပြောမယ်ဆိုရင်တော့ HTML Element Object နဲ့ DOM Node Object နှစ်မျိုးကနေ ပေါင်းစပ် ဆင်းသက်လာတဲ့ Object တွေပါ။ ဒါကြောင့် အဲ့ဒီ Node တွေမှာ Property တွေ Method တွေရှိကြပါတယ်။ ဒီ Property တွေ Method တွေထဲက ရွေးချယ်မှတ်သားသင့်တာတွေကို ခဏနေတော့ စုစည်းဖော်ပြပေးပါမယ်။

အရင်လေ့လာကြရမှာကတော့ လိုချင်တဲ့ Node ကို DOM Tree ကနေ ရွေးယူတဲ့နည်း ဖြစ်ပါတယ်။ ဒီလို ရွေးယူနိုင်ဖို့အတွက် Selector Method တွေကို သုံးရပါတယ်။ အရင်ကတော့ အခုလို Method တွေကို သုံးကြပါတယ်။

- `document.`**`getElementById`**`()`
- `document.`**`getElementsByTagName`**`()`
- `document.`**`getElementsByClassName`**`()`

Element တွေကို `id` နဲ့ရွေးယူနိုင်သလို Element Name နဲ့လည်း ရွေးယူနိုင်မှာပါ။ `class` နဲ့လည်း ရွေးယူနိုင်ပါတယ်။ အခုနောက်ပိုင်းမှာတော့ ဒီ Method တွေထက်ပိုကောင်းတဲ့ Method တွေ ရှိနေပါပြီ။

- `document.`**`querySelector`**`()`
- `document.`**`querySelectorAll`**`()`

ဒီ Method တွေကတော့ CSS Selector ရေးထုံးအတိုင်း အသုံးပြုပြီး Element တွေကို ရွေးယူပေးနိုင်တဲ့ Method တွေပါ။ CSS မှာ Element Selector, Class Selector, ID Selector, Decedent Selector, Child Selector, Attribute Selector စသည်ဖြင့် စုံနေအောင်ရှိသလို၊ အဲ့ဒီ Selector တွေအတိုင်း JavaScript မှာလည်း အကုန်အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်။ `querySelector()` Method က Element တစ်ခုကို ရွေးယူဖို့ သုံးနိုင်ပြီး `querySelectorAll()` ကိုတော့ ပေးလိုက်တဲ့ Selector နဲ့ကိုက်ညီတဲ့ Element List ကို ရယူဖို့ သုံးနိုင်ပါတယ်။

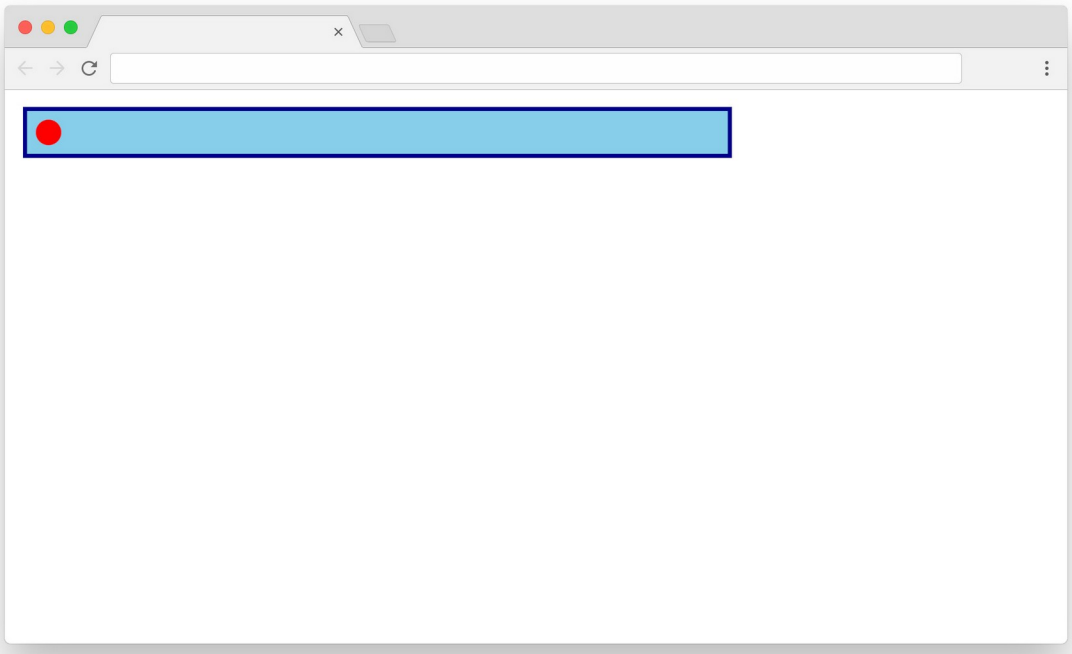
Element တစ်ခုကို ရွေးယူပြီးနောက်မှာ DOM Property တွေ Method တွေကိုသုံးပြီး အဲ့ဒီ Element ကို စီမံလို့ရပြီ ဖြစ်ပါတယ်။ အသုံးများမယ့် Property တွေ Method တွေအကြောင်း မပြောခင်၊ ပိုပြီးတော့ စိတ်ဝင်စားဖို့ ကောင်းသွားအောင် လက်တွေ့နမူနာလေးတစ်ခုလောက် အရင်လုပ်ကြည့်လိုက်ချင်ပါတယ်။

HTML Document တစ်ခုတည်ဆောက်ပြီး ဒီကုဒ်ကို ကူးရေးလိုက်ပါ။

HTML & CSS

```
<!DOCTYPE html>
<html>
<head>
    <title>Title</title>
    <style>
        #box {
            position: relative;
            width: 800px;
            height: 30px;
            padding: 10px;
            background: skyblue;
            border: 5px solid darkblue;
        }

        #ball {
            position: absolute;
            left: 0;
            top: 0;
            width: 30px;
            height: 30px;
            border-radius: 30px;
            background-color: red;
        }
    </style>
</head>
<body>
    <div id="box">
        <div id="ball"></div>
    </div>
</body>
</html>
```

HTML/CSS တွေအကြောင်းကို သိရှိပြီးသားဖြစ်တယ်လို့ သဘောထားတဲ့အတွက် ဒီကုဒ်ရဲ့ အကြောင်းကို ရှင်းပြမနေတော့ပါဘူး။ ရှင်းနေမှ ရှုပ်နေပါသေးတယ်။ ရေးထားတဲ့ ကုဒ်ကိုသာ လေ့လာကြည့်လိုက်ပါ။ ရလဒ်ကတော့ အခုလိုရရှိမှာဖြစ်ပါတယ်။



လုပ်ကြည့်ချင်တာက `id` မှာ `ball` လို့ပေးထားတဲ့ Element လေး ရွှေ့လျားနေတဲ့ Animation လုပ်ဆောင်ချက်လေး ရအောင် JavaScript နဲ့ ရေးကြည့်ချင်တာပါ။ သိပ်မခက်ပါဘူး။ လက်ရှိရေးထားတဲ့ CSS ကုဒ်ကိုလေ့လာကြည့်လိုက်ရင် `#ball` Element အတွက် `position: absolute` နဲ့ `top: 0; left: 0` လို့ သတ်မှတ်ပေးထားတာကို သတိပြုပါ။ JavaScript ကုဒ်လေးတစ်ချို့ စရေးကြပါမယ်။ ဒီကုဒ်ကို ကို HTML Structure ရဲ့အောက်နား `</body>` မပိတ်ခင်လေးမှာ ရေးထည့်ပေးလိုက်ပါ။

HTML & JavaScript

```
<script>
    let ball = document.querySelector("#ball")
    ball.style.left = "100px"
</script>
```

`querySelector()` ကိုသုံးပြီး ID Selector နဲ့ `#ball` Element ကို ရွေးယူလိုက်ပါတယ်။ ရလဒ်ကို `ball` Variable ထဲမှာ ထည့်လိုက်ပါတယ်။ Variable ထဲကိုရောက်ရှိသွားမှာ Element Reference ခေါ်

အညွှန်းတန်ဖိုးတစ်ခုဖြစ်ပါတယ်။ ဒီသဘောကို Variable က ရွေးထားတဲ့ Element ကို ထောက်ထားလိုက်တာ လို့ မြင်ကြည့်နိုင်ပါတယ်။ ဒါကြောင့် Variable ပေါ်မှာပြုလိုက်လိုက်တဲ့ အပြောင်းအလဲတွေဟာ Element ပေါ်မှာလည်း သက်ရောက်သွားမှာဖြစ်ပါတယ်။

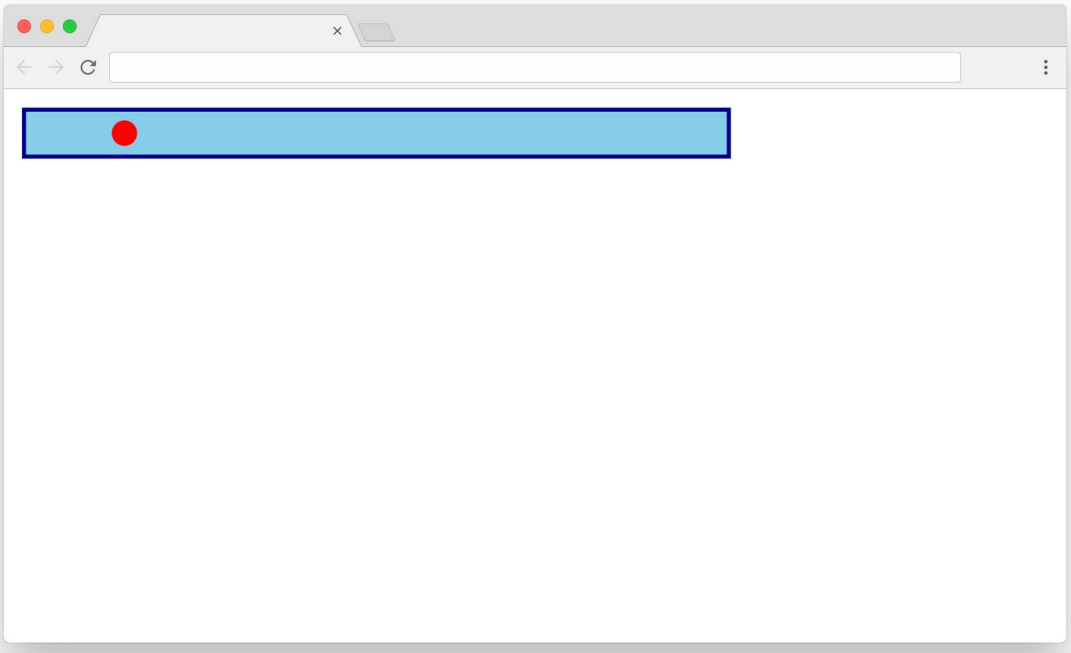
ပထမဦးဆုံး DOM Property အနေနဲ့ `style` Property ကို အသုံးပြုထားတာကို တွေ့ရနိုင်ပါတယ်။ `style` Property နဲ့အတူ CSS Property တွေထဲက နှစ်သက်ရာကို ပူးတွဲအသုံးပြုနိုင်ပါတယ်။ `background, border, color, textAlign, margin, paddingTop` စသည်ဖြင့်ပါ။ ဒီနေရာမှာ ရေးဟန်ကို သတိပြုပါ။ CSS တွေရေးတဲ့အခါ `text-align, padding-top` စသည်ဖြင့် Dash လေးတွေသုံးပြီး ရေးကြပါတယ်။ JavaScript ကုဒ်ထဲမှာ Dash တွေကိုသုံးခွင့်မရှိ Camel Case နဲ့ ပြောင်းရေးပေးရပါတယ်။ `text-align` ဆိုရင် `textAlign`၊ `padding-top` ဆိုရင် `paddingTop` စသည်ဖြင့် ပြောင်းရေးပေးရပါတယ်။

အကယ်၍ CSS ရေးထုံးအတိုင်းပဲ ရေးချင်တယ်ဆိုရင်လည်း ရတော့ရပါတယ်။ Object Property ရေးထုံးအစား Array Index ရေးထုံးကိုသုံးနိုင်ပါတယ်။ ဥပမာ ဒီလိုပါ –

Pseudocode

```
element.style["padding-top"] = "10px"
element.style["text-align"] = "center"
```

နမူနာမှာတော့ Object Property ရေးထုံးကိုပဲသုံးပြီးတော့ `ball.style.left = 100px` လို့ ပြောထားတဲ့အတွက် `#ball` Element ရဲ့ CSS Style `left` ရဲ့တန်ဖိုး `100px` ဖြစ်သွားပါပြီ။ မူလ CSS ကုဒ်ထဲမှာ 0 လို့ပြောထားပေမယ့် ဒီ JavaScript ကုဒ်အလုပ်လုပ်သွားတဲ့အခါ တန်ဖိုးပြောင်းသွားပါပြီ။ ဒါကြောင့် စမ်းကြည့်လိုက်ရင် ရလဒ်ကို အခုလိုတွေ့မြင်ရမှာပါ။



`#ball` Element ဟာ `left: 100px` မှာနေရာယူဖော်ပြနေခြင်းပဲဖြစ်ပါတယ်။ ဒီ Element လေး တစ်ဖြည်းဖြည်းနဲ့ ရွှေ့သွားတဲ့ လုပ်ဆောင်ချက်လေး ရဖို့အတွက် JavaScript ကုဒ်ကို ဒီလိုပြင်ရေးပေးလိုက်ပါမယ်။

HTML & JavaScript

```
<script>
    let ball = document.querySelector("#ball")

    let left = 0

    setInterval(() => {
        left += 1
        if(left > 790) left = 0
        ball.style.left = `${left}px`
    }, 10)
</script>
```

`left` Variable တစ်ခုကြေညာထားပြီး တန်ဖိုးကို 0 လို့သတ်မှတ်ထားပါတယ်။ ပြီးခဲ့တဲ့အခန်းတွေမှာ `setTimeout()` Function အကြောင်းကို လေ့လာဖူးခဲ့ကြပါတယ်။ သတ်မှတ်ထားတဲ့အချိန် ရောက်တော့မှ Callback Function ကို အလုပ်လုပ်ပေးတဲ့ Function ပါ။ အခုတစ်ခါ `setInterval()`

Function ကို သုံးပြထားပါတယ်။ `setInterval()` ကလည်း `setTimeout()` လိုပါပဲ။ သတ်မှတ်အချိန် ရောက်တော့မှ Callback Function ကို အလုပ်လုပ်ပေးတာပါ။ ဒါပေမယ့် `setTimeout()` က တစ်ကြိမ်ပဲ လုပ်ပြီး `setInterval()` ကတော့ ထပ်ခါထပ်ခါ လုပ်သွားမှာပဲ ဖြစ်ပါတယ်။ နမူနာမှာ `setInterval()` ရဲ့ ဒုတိယ Argument ကို 10 လို့ပေးထားတဲ့အတွက် 10 မီလီစက္ကန့်ပြည့်တိုင်း တစ်ကြိမ် ထပ်ခါထပ်ခါ အလုပ်လုပ်သွားမှာ ဖြစ်ပါတယ်။

ဘာတွေ လုပ်သွားတာလဲဆိုတော့ ရေးထားတဲ့ကုဒ်ကို လေ့လာကြည့်ပါ။ `left` ကို 1 တိုးပေးထားတာပါ။ ဒါကြောင့် 10 မီလီစက္ကန့်တိုင်းမှာ `1px` စီဘယ်ကနေညာကို ရွှေ့သွားမှာ ဖြစ်ပါတယ်။ အဲ့ဒီလိုရွှေ့သွားတဲ့ အခါပင်မ `box` Element ထက်တော့ ကျော်မသွားစေချင်ပါဘူး။ ဒါကြောင့် `box` ရဲ့ `width` ဖြစ်တဲ့ 800 ကို ကျော်သွားခဲ့ရင် `left` တန်ဖိုးကို 0 ပြန်လုပ်ပေးရမှာပါ။ နမူနာမှာ 800 လို့ မသတ်မှတ်ဘဲ 790 ကို ကျော်သွားရင် 0 ဖြစ်ရမယ်လို့ ပြန်သတ်မှတ် ထားတာကတော့ `box width + (padding * 2) – ball width` ကိုတွက်ယူထားတာ မို့လို့ပါ။ CSS Box Model ကိုလေ့လာဖူးရင် Padding ကြောင့် Box အရွယ်အစား ပြောင်းပုံကို သိကြပြီးဖြစ်မှာပါ။ မိမိနှစ်သက်ရာတန်ဖိုးတွေ အမျိုးမျိုးအတိုးအလျှော့ပြောင်းပြီး စမ်းကြည့်သင့်ပါတယ်။

တစ်ကယ်တော့ ဒီလုပ်ဆောင်ချက် ရဖို့အတွက် JavaScript ကုဒ်တွေ မလိုသေးပါဘူး၊ CSS Animation နဲ့တင် လုပ်လို့ရနိုင်ပါတယ်။ ဒါပေမယ့် HTML Element တွေကို JavaScript နဲ့ စီမံနိုင်တယ်ဆိုတဲ့ သဘောသဘာဝ နမူနာကို ရစေဖို့အတွက် ဖော်ပြလိုက်တာပါ။ ရေးလက်စနဲ့ စဉ်းစားစရာ Condition Logic လေးတစ်ချို့ ထပ်ဖြည့်ပါဦးမယ်။

လက်ရှိနမူနာမှာ `ball` လေးက အဆုံးကိုရောက်ရင် အစကို ချက်ချင်းပြန်ရောက်သွားပါတယ်။ အဲ့ဒီလို ချက်ချင်းပြန်ရောက်မယ့်အစား တစ်ရွှေ့ရွှေ့နဲ့ အဆုံးကိုရောက်သွားရင် တစ်ရွှေ့ရွှေ့နဲ့ပဲ အစကို ပြန်သွားအောင် လုပ်ကြည့်ချင်ပါတယ်။ ရေးထားတဲ့ကုဒ်ကို ဒီလိုပြင်ပေးရမှာပါ။

HTML & JavaScript

```
<script>
    let ball = document.querySelector("#ball")

    let left = 0
    let direction = "right"

    setInterval(() => {
        if(direction === "right") {
            left += 1
            if(left > 790) direction = "left"
        } else {
            left -= 1
            if(left < 1) direction = "right"
        }

        ball.style.left = `${left}px`
    }, 10)
</script>
```

နမူနာမှာ `direction` Variable တစ်ခု ပါသွားပါပြီ။ `setInterval()` ထဲမှာ `direction` က `right` ဖြစ်မှ `left` တန်ဖိုးကို 1 တိုးထားပါတယ်။ မဟုတ်ဘူးဆိုရင် 1 နှုတ်ထားပါတယ်။ ဒါကြောင့် တစ်ရွေ့ရွေ့ 1 တိုးပြီးသွားရာကနေ အဆုံးကိုရောက်တဲ့အခါ တစ်ရွေ့ရွေ့ 1 နှုတ်ပြီး ပြန်လာမှာ ဖြစ်ပါတယ်။ ရေးထားတဲ့ကုဒ်ကို လေ့လာပြီး သေချာစဉ်းစားကြည့်ပါ။ ပရိုဂရမ်းမင်းဆိုတာ ဒီလိုပဲ Logic လေးတွေကို စဉ်းစားတွေးခေါ် အသုံးချရတာမျိုးပါ။

ဆက်လက်ပြီးတော့ ရွေ့နေတဲ့ `ball` လေးကို ကြိုက်တဲ့အချိန် ရပ်လို့/ပြန်စလို့ ရတဲ့ လုပ်ဆောင်ချက် ထပ်ထည့်ပါဦးမယ်။ ဒီအတွက် `onClick` လို့ခေါ်တဲ့ HTML Attribute ကို သုံးနိုင်ပါတယ်။ ရေးလက်စ HTML Structure ထဲမှာ အခုလို `<button>` Element နှစ်ခု ထည့်ပေးပါ။

HTML

```
<button onClick="start()">Start</button>
<button onClick="stop()">Stop</button>
```

Button တွေမှာ `onClick` Attribute ကိုယ်စီပါပြီး တန်ဖိုးအနေနဲ့ JavaScript ကုဒ်ကို ပေးရပါတယ်။ နမူနာအရ Start Button ကိုနှိပ်ရင် `start()` Function ကိုခေါ်ထားပြီး Stop Button ကိုနှိပ်ရင် `stop()` Function ကို ခေါ်ထားပါတယ်။ ရေးလက်စ JavaScript ကုဒ်ကို အခုလိုပြင်ပေးပါ။

HTML & JavaScript

```
<script>
    let ball = document.querySelector("#ball")
    let left = 0
    let direction = "right"

    let interval = setInterval(move, 10)

    function move() {
        if(direction === "right") {
            left += 1
            if(left > 790) direction = "left"
        } else {
            left -= 1
            if(left < 1) direction = "right"
        }

        ball.style.left = `${left}px`
    }

    function start() {
        interval = setInterval(move, 10)
    }

    function stop() {
        clearInterval(interval)
    }
</script>
```

စောစောက Arrow Function နဲ့ ရေးထားတဲ့ `setInterval()` အတွက် Callback Function ကို move ဆိုတဲ့အမည်နဲ့ သီးခြား Function အဖြစ် ခွဲထုတ်လိုက်တာပါ။ ပြီးတော့မှ `setInterval()` ရဲ့ Callback နေရာမှာ `move()` ကို ပြန်ပေးလိုက်ပါတယ်။ ထူးခြားချက်အနေနဲ့ `setInterval()` ကပြန်ပေးတဲ့ ရလဒ်ကို `interval` Variable ထဲမှာ ထည့်ထားတာကို သတိပြုပါ။

ဆက်လက်ပြီး `start()` Function နဲ့ `stop()` Function တို့ကို ရေးထားပါတယ်။ `start()` Function က `setInterval()` ကိုပဲ နောက်တစ်ခါ ပြန် Run ထားတာပါ။ ဒါကြောင့်နှိပ်လိုက်ရင် ရွေ့တဲ့အလုပ် လုပ်ပါလိမ့်မယ်။ `stop()` Function ကတော့ `clearInterval()` လို့ခေါ်တဲ့ Standard Function ကိုသုံးပြီး Run လက်စ `interval()` ကို ရပ်လိုက်တာပါ။ ဒါကြောင့် နှိပ်လိုက်ရင် ရပ်သွားပါလိမ့်မယ်။ စမ်းကြည့်လို့ရပါပြီ။

ကုဒ်တွေကို အဆင့်လိုက် အပိုင်းအစလေးတွေ ခွဲရေးပြထားလို့ အဆင်မပြေရင် ကုဒ်အပြည့်အစုံကို ဒီမှာ ဒေါင်းပြီး စမ်းကြည့်လို့ရပါတယ်။

– https://github.com/eimg/javascript-book

DOM Properties & Methods

Element အတွင်းထဲက Content ကို စီမံလိုရင် `textContent` (သို့မဟုတ်) `innerHTML` Property ကိုသုံးနိုင်ပါတယ်။ `textContent` က Content ထဲမှာ HTML Element တွေထည့်ပေးရင် Element အနေနဲ့ အလုပ်မလုပ်ဘဲ စာအနေနဲ့ပဲ ပြပေးမှာပါ။ `innderHTML` ကတော့ Content ထဲမှာ HTML Element တွေပါရင် ထည့်အလုပ်လုပ် ပေးပါလိမ့်မယ်။

HTML

```
<div id="box1">
   Box One Content
</div>

<div id="box2">
    Box Two Content
</div>
```

JavaScript

```
document.querySelector("#box1").textContent = "Hello <b>DOM</b>"
document.querySelector("#box2").innerHTML = "Hello <b>DOM</b>"
```

နမူနာမှာ `#box1` နဲ့ `#box2` နှစ်ခုရှိပြီး တစ်ခုကို `textContent` နဲ့ ပြင်ထားပါတယ်။ နောက်တစ်ခုကို `innerHTML` နဲ့ ပြင်ထားပါတယ်။ ပေးလိုက်တဲ့ Content ကအတူတူပါပဲ။ ရလဒ်ကတော့ တူမှာမဟုတ်ပါဘူး။ ဒီလိုပုံစံဖြစ်မှာပါ –

```
Hello <b>DOM</b>
Hello DOM
```

လိုအပ်ချက်ပေါ်မူတည်ပြီး နှစ်သက်ရာကိုသုံးနိုင်ပါတယ်။ အများအားဖြင့် `textContent` ကိုသုံးသင့်တယ်လို့ အကြံပြုကြပါတယ်။ လုံခြုံရေးအရပဲကြည့်ကြည့် Performance အရပဲကြည့်ကြည့် ပိုကောင်းလို့ပါ။ မဖြစ်မနေ လိုအပ်တာသေချာတယ်ဆိုမှသာ `innerHTML` ကို သုံးသင့်ပါတယ်။

Element ထဲက Content ကို အကုန်ပြင်တာမျိုး မဟုတ်ဘဲ၊ ထပ်တိုးချင်ရင် `appendChild()` Method နဲ့ တိုးတိုင်ပါတယ်။

HTML

```
<div id="box">
    An Element.
</div>
```

JavaScript

```
let hello = document.createTextNode("Hello DOM.")
document.querySelector("#box").appendChild(hello)
```

ဒါဆိုရင် `#box` Element ထဲမှာ Hello DOM. ဆိုတဲ့စာကို ထပ်တိုးပေးသွားမှာပါ။ ဒီလိုထပ်တိုးလို့ရဖို့အတွက် `createTextNode()` Method ကိုသုံးပြီး အရင်ဆုံး Text Node တစ်ခုတည်ဆောက်ပေးရတယ် ဆိုတာကိုတော့ သတိပြုပါ။ အကယ်၍ Element Node တည်ဆောက်ပြီး တိုးချင်ရင်လည်း `createElement()` Method ကိုသုံးနိုင်ပါတယ်။

JavaScript

```
let hello = document.createElement("b")
hello.textContent = "Hello DOM."
document.querySelector("div").appendChild(hello)
```

ဒီတစ်ခါတော့ <b> Element တစ်ခုအရင်ဆောက်လိုက်ပြီး၊ အဲ့ဒီ <b> Element ထဲမှာ ထည့်ချင်တဲ့စာကို textContent နဲ့ထည့်လိုက်ပါတယ်။ ပြီးတော့မှ appendChild() နဲ့ ထပ်တိုးပေးလိုက်တာပါ။

appendChild() Method က ထပ်တိုးတဲ့ Element တွေကို နောက်ကနေတိုးပေးတာပါ။ ရှေ့ကနေတိုးချင်ရင် insertBefore() Method ကိုသုံးနိုင်ပါတယ်။

JavaScript

```
let hello = document.createElement("b")

hello.textContent = "Hello DOM."

let div = document.querySelector("div")

div.insertBefore(hello, div.firstChild)
```

insertBefore() အတွက် Argument နှစ်ခုပေးရပါတယ်။ ပထမ Argument ကထပ်တိုးချင်တဲ့ Element ဖြစ်ပြီး ဒုတိယ Argument က ဘယ်သူ့ရှေ့မှာ ထပ်ထိုးရမှာလဲဆိုတဲ့ Element ဖြစ်ပါတယ်။ နမူနာမှာ div ကို အရင်သီးခြား Select လုပ်လိုက်ပြီး div ထဲကို insertBefore() နဲ့ ထပ်တိုးထားပါတယ်။ div.firstChild နဲ့ div ထဲမှာ မူလရှိနေတဲ့ ပထမဆုံး Child Node ရဲ့ရှေ့ကနေ ထပ်တိုးခိုင်းလိုက်တာပါ။ တစ်လက်စထဲ Element တစ်ခုအတွင်းက ပထမဆုံး Node ကိုလိုချင်ရင် firstChild Property ကိုသုံးရတယ်ဆိုတာကိုပါ ထည့်သွင်းမှတ်သားနိုင်ပါတယ်။ နောက်ဆုံး Node ကို လိုချင်ရင်တော့ lastChild Property ကို သုံးနိုင်ပါတယ်။

Element တစ်ခုကို ပယ်ဖျက်လိုရင်တော့ remove() Method ကိုသုံးနိုင်ပါတယ်။

JavaScript

```
document.querySelector("div").remove()
```

ဒါဆိုရင် Element ကို Document ပေါ်ကနေ ဖယ်ဖျက်လိုက်မှာပါ။

Element မှာ Attribute တွေသတ်မှတ်ဖို့အတွက် `setAttribute()` Method ကိုသုံးနိုင်ပါတယ်။

JavaScript

```
let hello = document.createElement("b")

hello.textContent = "Hello DOM."
hello.setAttribute("title", "A new element")
document.querySelector("div").appendChild(hello)
```

အသစ်တည်ဆောက်လိုက်တဲ့ Element မှာ `title` Attribute ထည့်ပေးလိုက်တာပါ။ Attribute တန်ဖိုးကို လိုချင်ရင် `getAttribute()` Method ကိုသုံးနိုင်ပြီး Attribute တန်ဖိုးကို ပယ်ဖျက်လိုရင် `removeAttribute()` ကို သုံးနိုင်ပါတယ်။

`class` တွေကိုစီမံလိုရင်တော့ `setAttribute()` တို့ `getAttribute()` တို့ကိုပဲ သုံးရင်ရသလို၊ `classList` Property ကို `add(), remove(), toggle()` Method တွေနဲ့လည်း တွဲသုံးနိုင်ပါတယ်။

JavaScript

```
let hello = document.createElement("b")

hello.textContent = "Hello DOM."
hello.classList.add("alert", "info")

document.querySelector("div").appendChild(hello)
```

နမူနာအရ အသစ်တည်ဆောက်လိုက်တဲ့ Element မှာ `alert` နဲ့ `info` ဆိုပြီး `class` နှစ်ခုထည့်ပေးလိုက်တာပါ။ ထပ်ပေးချင်တဲ့ `class` တွေရှိရင် Argument စာရင်းထဲမှာ ထပ်တိုးပေးလိုက်ယုံပါပဲ။ ပြန်ဖြုတ်လိုရင်တော့ `classList.remove()` နဲ့ဖြုတ်နိုင်ပါတယ်။ `classList.toggle()` ကတော့

`class` မရှိရင် ထည့်ပေးပြီး ရှိနေရင် ဖြုတ်ပေးပါတယ်။ ထည့်လိုက်/ထုတ်လိုက် လုပ်ဖို့လိုတဲ့နေရာတွေမှာ အသုံးဝင်ပါတယ်။

Element တစ်ခုကို မိတ္တူပွားယူချင်ရင် `cloneNode()` Method ကိုသုံးနိုင်ပါတယ်။

JavaScript

```
let hi = hello.cloneNode(true)
```

နမူနာအရ စောစောက အသစ်တည်ဆောက်ထားတဲ့ `hello` Element ကို `hi` ဆိုတဲ့အမည်နဲ့ ပွားယူလိုက်တာပါ။ Argument မှာ true လို့ပေးလိုက်တာကတော့ အကုန်ရှိသမျှပွားယူမယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ အကယ်၍ false ဆိုရင် Element နဲ့ Attribute တွေပဲ ပွားယူပြီး အထဲက Child တွေ ပါမှာ မဟုတ်ပါဘူး။

လက်ရှိ Element ရဲ့ Parent Node ကို ရယူလိုရင် `parentNode` Property ကို အသုံးပြုနိုင်ပါတယ်။ ဒါလည်း အသုံးဝင်ပါတယ်။ Select လုပ်တဲ့အခါ CSS Selector ရေးထုံးအရ Child Selector သာ ရှိပါတယ်။ Parent Selector ဆိုတာ မရှိပါဘူး။ ဒါကြောင့် Parent Element လိုချင်ရင် ဒီနည်းကိုပဲ အသုံးပြုရမှာပါ။

HTML

```
<p>
    Some content <b id="badge">10</b>
</p>
```

JavaScript

```
let p = document.querySelector("#badge").parentNode
```

ဒါဟာ `#badge` Element ရှိနေတဲ့ Parent ဖြစ်တဲ့ `<p>` ကို Select လုပ်လိုက်တာပါပဲ။

တစ်ခြားအသုံးဝင်တဲ့ DOM Manipulation ကုဒ်နမူနာလေးတွေကို ဒီလိပ်စာမှာလေ့လာနိုင်ပါတယ်။

https://htmldom.dev/

Events

တစ်ချို့အလုပ်တွေကို Click နှိပ်လိုက်တော့မှ အလုပ်လုပ်စေချင်တယ်၊ Keyboard က Key တစ်ခုခုကို နှိပ်လိုက်တော့မှ အလုပ်လုပ်စေချင်တာမျိုးတွေရှိရင် Event Handler ကို သုံးနိုင်ပါတယ်။ Event ဆိုတာ အခြေအနေတစ်ခုခု ဖြစ်ပေါ်တော့မှ လုပ်ဖို့ သတ်မှတ်ထားတဲ့ အလုပ်တွေပါ။

ရှိတဲ့ Event တွေကတော့ အများကြီးပါပဲ။ တစ်ချို့ Standard Event တွေဖြစ်ပြီး တစ်ချို့ နောက်မှ ထပ်တိုးထားတဲ့ Event တွေလည်း ရှိနိုင်ပါတယ်။ Online ဖြစ်သွားရင် ဘာလုပ်ရမယ်၊ Offline ဖြစ်သွားရင် ဘာလုပ်ရမယ်ဆိုတာမျိုးတွေ သတ်မှတ်ထားလို့ ရနိုင်ပါတယ်။ Animation တစ်ခု စသွားရင် ဘာလုပ်ရမယ်၊ ဆုံးသွားရင် ဘာလုပ်ရမယ် ဆိုတာမျိုးတွေ သတ်မှတ်ထားလို့ ရနိုင်ပါတယ်။ Copy ကူးလိုက်ရင် ဘာလုပ်ရမယ်၊ Paste လုပ်လိုက်ရင် ဘာလုပ်ရမယ် ဆိုတာမျိုးတွေ သတ်မှတ်ထားလို့ ရနိုင်ပါတယ်။ ဗွီဒီယို စသွားရင် ဘာလုပ်ရမယ်၊ ဆုံးသွားရင် ဘာလုပ်ရမယ်ဆိုတာမျိုးတွေ သတ်မှတ်ထားလို့ ရနိုင်ပါတယ်။ Node အကြောင်းပြောတုံးက နမူနာပေးခဲ့တဲ့ `on('data')` တို့ `on('end')` တို့ဆိုတာလည်း Event တွေပါပဲ။ Data ရောက်လာရင် ဘာလုပ်ရမယ်၊ ပြီးသွားရင် ဘာလုပ်ရမယ်ဆိုတာမျိုးတွေ သတ်မှတ်ထားတာပါ။

အခဲ့ဒီလို Event တွေအများကြီး ရှိတဲ့ ထဲကမှ အသုံးများမှာတွေကတော့ Input Event တွေဖြစ်ပါတယ်။ Click, Keydown, Keyup, Keypress, Focus, Blur, Change, Submit တို့လို Event တွေပါ။ Click Event က Mouse Click မှာ ဖြစ်ပေါ်ပါတယ်။ Keydown, Keyup, Keypress Event တွေကတော့ Keyboard က Key တစ်ခုခုကိုနှိပ်လိုက်ရင် ဖြစ်ပေါ်ပါတယ်။ နှိပ်လိုက်ချိန်၊ လွှတ်လိုက်ချိန်၊ နှိပ်ပြီးသွားချိန် ဆိုပြီး သုံးခု ခွဲထားတာပါ။ Focus Event ကတော့ Focus ဖြစ်သွားချိန် ဖြစ်ပေါ်ပြီး Blur ကတော့ Focus လွတ်သွားချိန်မှာ ဖြစ်ပေါ်ပါတယ်။ Change ကတော့ Input Value ပြောင်းသွားချိန်မှာ ဖြစ်ပေါ်ပါတယ်။ Submit ကတော့ HTML Form တစ်ခုကို Submit လုပ်လိုက်ချိန်မှာ ဖြစ်ပေါ်ပါတယ်။

နောက်ထပ်အသုံးများမယ့် Event တွေကတော့ Load နဲ့ Unload တို့ပါ။ Document ကိုဖွင့်လိုက်ချိန်မှာ Load Event ဖြစ်ပေါ်ပြီး ပိတ်လိုက်ချိန်မှာ Unload Event ဖြစ်ပေါ်ပါတယ်။ ရှိရှိသမျှ Event စာရင်းကို သိချင်ရင် ဒီမှာကြည့်လို့ရပါတယ်။

https://developer.mozilla.org/en-US/docs/Web/Events

ဒီ Event တွေကို အသုံးပြုပုံပြုနည်း (၃) နည်းရှိပါတယ်။ တစ်နည်းကတော့ HTML Event Attribute တွေကို အသုံးပြုနိုင်ပါတယ်။ ပြီးခဲ့တဲ့ နမူနာမှာလည်း တွေ့ခဲ့ပြီးသားပါ။

HTML

```
<p id="note">
    Hello World
</p>
<button onClick="hello()">Button</button>
```

နမူနာအရ `<button>` Element မှာ `onClick` Attribute ကိုသုံးပြီး Click Event ဖြစ်ပေါ်တဲ့အခါ လုပ်ရမယ့် ကုဒ်ကို သတ်မှတ်ပေးထားပါတယ်။ `hello()` Function ကို ခေါ်လိုက်မှာပါ။ ဒါကြောင့် `hello()` Function ရှိနေဖို့တော့လိုပါတယ်။

HTML & JavaScript

```
<script>
    function hello() {
        document.querySelector("#note").textContent = "Hello Event"
    }
</script>
```

နောက်တစ်နည်းက `addEventListener()` Method ကို အသုံးပြုခြင်းဖြစ်ပါတယ်။ ဒီနည်းကို သုံးမယ်ဆိုရင် HTML Attribute မလိုတော့ပါဘူး။

HTML & JavaScript

```
<script>
    document.querySelector("button").addEventListener("click", e => {
        document.querySelector("#note").textContent = "Hello Event"
    })
</script>
```

`addEventListener()` Method အတွက် Argument နှစ်ခုပေးရပါတယ်။ Event အမျိုးအစားနဲ့ Event ဖြစ်ပေါ်တဲ့အခါ လုပ်ရမယ့် Callback Function တို့ပါ။ ဒါကြောင့် နမူနာအရ `<button>` မှာ `click` Event ဖြစ်ပေါ်တဲ့အခါ `#note` Element ရဲ့ `textContent` ပြောင်းသွားမှာပါ။ Event

Callback ရဲ့ `e` Parameter ထဲမှာ Click Event ဆိုရင် Left Click လား၊ Right Click လား။ Keypress Event ဆိုရင် ဘယ် Key ကိုနှိပ်တာလဲ၊ စသည်ဖြင့် Event နဲ့သက်ဆိုင်တဲ့ အချက်အလက်တွေ ရှိနေမှာဖြစ်ပါတယ်။

နောက်တစ်နည်းကတော့ `on` နဲ့စတဲ့ Property တွေကို အသုံးပြုရပါတယ်။ `onclick, onkeydown, onfocus` စသည်ဖြင့် ရေးလို့ရပါတယ်။

HTML & JavaScript

```
<script>
    document.querySelector("button").onclick = function(e) {
        document.querySelector("#note").textContent = "Hello Event"
    }
</script>
```

အပြောင်းအလဲဖြစ်သွားအောင် ရိုးရိုး Function Expression ပြောင်းရေးထားပေမယ့် Arrow Function နဲ့ ရေးမယ်ဆိုရင်လည်း ရပါတယ်။ အတူတူပါပဲ။ ကျန် Event တစ်ချို့ကို နမူနာတွေနဲ့အတူ ဆက်ကြည့်ကြပါမယ်။

Input Sample

နမူနာကုဒ်လို့တစ်ချို့ ဆက်ရေးကြည့်ချင်ပါတယ်။ သိပ်ထူးထူးဆန်းဆန်းကြီး မဟုတ်ပါဘူး။ `<input>` နှစ်ခုပေးထားပြီး၊ `<button>` ကိုနှိပ်လိုက်ရင် `<input>` တွေမှာ ရေးဖြည့်ထားတဲ့ တန်ဖိုးတွေရဲ့ ပေါင်းခြင်းရလဒ်ကို ပြပေးတဲ့ ကုဒ်လေးပါ။ အခုလိုရေးစမ်းကြည့်နိုင်ပါတယ်။

HTML, CSS & JavaScript

```
<!DOCTYPE html>
<html>
<head>
    <title>Title</title>
    <style>
        #result {
            font-size: 2em;
            font-weight: bold;
        }
    </style>
</head>
```

```
<body>
    <h1>Add Form</h1>
    <div class="form">
        <div>
            <input type="text" id="a">
        </div>
        <div>
            <input type="text" id="b">
        </div>
        <div id="result"></div>
        <button id="add">Add</button>
    </div>

    <script>
        document.querySelector("#add").onclick = function() {
            let a = document.querySelector("#a").value
            let b = document.querySelector("#b").value

            document.querySelector("#result").textContent = a + b
        }
    </script>
</body>
</html>
```

ရေးထားတဲ့ကုဒ်ကိုလေ့လာကြည့်လိုက်ရင် `#add` Button အတွက် `onclick` Property နဲ့ Click နှိပ်လိုက်ရင် လုပ်ရမယ့်အလုပ်ကို သတ်မှတ်ပေးထားပါတယ်။ `value` Property ကိုသုံးပြီး Input တွေထဲမှာ ရေးဖြည့်ထားတဲ့တန်ဖိုးကို ယူပြီးတော့ ပေါင်းခြင်းရလဒ်ကို `#result` Element ထဲမှာ ပြပေးလိုက်တာပါ။

မှန်တော့ မှန်ဦးမှာ မဟုတ်ပါဘူး။ 1 နဲ့ 2 ရိုက်ထည့်ပြီး ခလုပ်ကိုနှိပ်လိုက်ရင် ရလဒ်က 3 ဖြစ်ရမယ့်အစား 12 ဖြစ်နေမှာပါ။ a နဲ့ b ရိုက်ထည့်ပြီး ခလုပ်နှိပ်လိုက်ရင် ab ကိုရလဒ်အနေနဲ့ ရမှာပါ။ ဘာကြောင့်လဲဆိုတော့ Input တွေမှာ ရေးဖြည့်ထားတဲ့ တန်ဖိုးတွေက String တွေဖြစ်နေလို့ပါ။ ဒါကြောင့် JavaScript ကုဒ်ကို အခုလိုပြင်ပေးလိုက်ပါ။

JavaScript

```
document.querySelector("#add").onclick = function() {
    let a = document.querySelector("#a").value
    let b = document.querySelector("#b").value
    let result = parseInt(a) + parseInt(b)

    document.querySelector("#result").textContent = result
}
```

ဒီတစ်ခါတော့ `parseInt()` Function ရဲ့အကူအညီနဲ့ String ကို Number ပြောင်းထားပါတယ်။ ပြီးတော့မှ ပေါင်းတဲ့အတွက် ဂဏန်းတွေပေါင်းတဲ့အခါ မှန်သွားပါလိမ့်မယ်။

အကယ်၍ ဂဏန်းမဟုတ်တာတွေ ရိုက်ထည့်ခဲ့မယ်ဆိုရင်တော့ NaN ဖြစ်နေမှာပါ။ ပြီးတော့ ဘာမှရိုက်ထည့်ရင်လည်း NaN ဖြစ်နေမှာပါ။ ဒါကြောင့် ဂဏန်းမဟုတ်တဲ့ တန်ဖိုးတွေဖြစ်နေရင် ဂဏန်းတွေသာ ရိုက်ထည့်ပေးပါလို့ Message Box လေးနဲ့ ပြပေးကြပါမယ်။

တန်ဖိုးက Number ဟုတ်မဟုတ်စစ်ဖို့အတွက် နည်းလမ်းနှစ်ခုရှိပါတယ်။ တစ်ခုကတော့ Integer တန်ဖိုးဟုတ်သလားလို့ `Number.isInteger()` Method ကိုအသုံးပြု စစ်ဆေးခြင်းဖြစ်ပြီး နောက်တစ်မျိုးကတော့ `isNaN()` ကိုအသုံးပြုပြီး Number မဟုတ်ဘူးလားလို့ ပြောင်းပြန် စစ်ဆေးခြင်းဖြစ်ပါတယ်။ အခုလို ပြင်ရေးပေးလိုက်ပါ။

JavaScript

```
document.querySelector("#add").onclick = function() {
    let a = document.querySelector("#a").value
    let b = document.querySelector("#b").value

    let result = parseInt(a) + parseInt(b)

    if( isNaN(result) ) {
        alert("Please enter correct numbers")
    } else {
        document.querySelector("#result").textContent = result
    }
}
```

`isNaN()` ကိုသုံးပြီး အကယ်၍ `result` က NaN ဖြစ်နေခဲ့ရင် Message လေးတစ်ခုပြပေးဖို့ `alert()` Method နဲ့ ရေးထားလိုက်ပါတယ်။ NaN မဟုတ်တော့မှသာ ရလဒ်ကိုပြခိုင်းလိုက်တာဖြစ်လို့ အဆင်ပြေသွားပါပြီ။

ဒါကိုနည်းနည်းပိုကောင်းသွားအောင် ထပ်ပြင်ပါဦးမယ်။ ရိုက်ထည့်တဲ့တန်ဖိုး မမှန်တဲ့အခါ ပြတဲ့ Message ကို သက်ဆိုင်ရာ Input နဲ့အတူ တွဲပြလိုက်ချင်ပါတယ်။ ဒါကြောင့် အခုလို Message Label လေးတွေ HTML ထဲမှာ ထပ်တိုးလိုက်ပါ။

JavaScript

```
<div class="form">
    <div>
        <input type="text" id="a">
        <label for="a">Please enter correct number</label>
    </div>
    <div>
        <input type="text" id="b">
        <label for="b">Please enter correct number</label>
    </div>
    <div id="result"></div>
    <button id="add">Add</button>
</div>
```

ပြီးတဲ့အခါ အဲ့ဒီ Message လေးတွေကို CSS နဲ့အခုလို ခဏဖျောက်ထားလိုက်ပါ။

CSS

```
label {
    display: none;
    color: red;
}
```

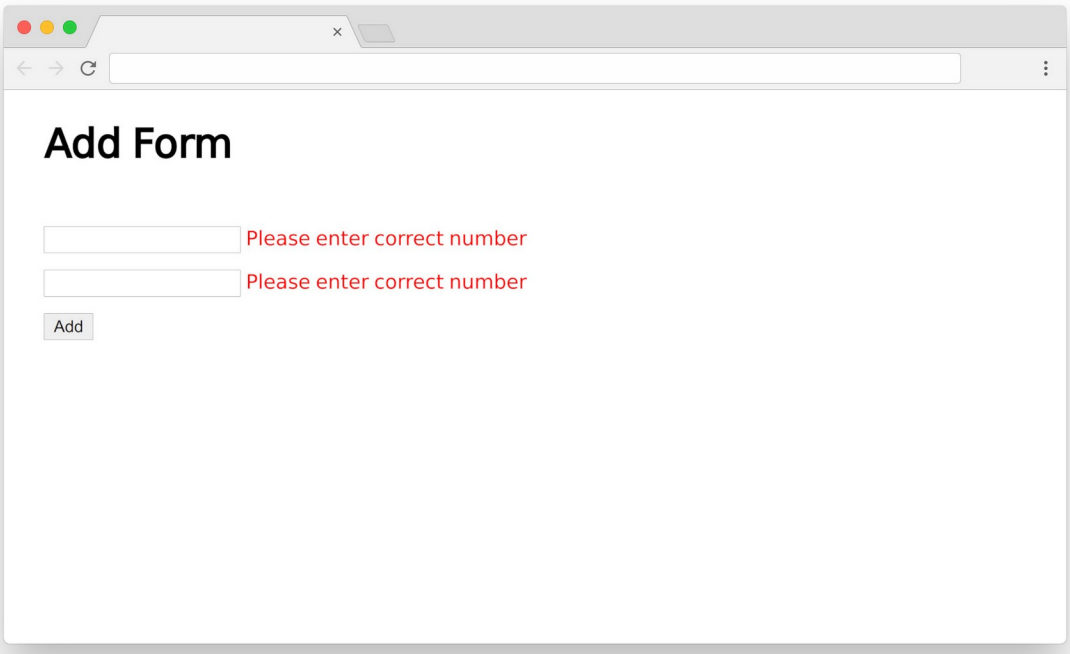
ပြီးတဲ့အခါ စောစောက JavaScript ကုဒ်ကို အခုလို ပြင်ပေးလိုက်ပါ။

JavaScript

```
document.querySelector("#add").onclick = function() {

    let a = parseInt(document.querySelector("#a").value)
    let b = parseInt(document.querySelector("#b").value)

    if( isNaN(a) ) {
        document.querySelector("[for=a]")
            .style.display = "inline"
    }

    if( isNaN(b) ) {
        document.querySelector("[for=b]")
            .style.display = "inline"
    }

    let result = a + b
```

```
    if( !isNaN(result) ) {
        document.querySelector("#result").textContent = result
    }
}
```

ဒီတစ်ခါ Input တွေစယူကတည်းက တန်ဖိုးတွေကို `parseInt()` လုပ်ထားတာကို သတိပြုပါ။ ပြီးတော့မှ `isNaN()` နဲ့ပဲ a ကိုအရင် စစ်လိုက်ပါတယ်။ a တန်ဖိုးမမှန်ရင် `querySelector()` နဲ့ Attribute Selector ရေးထုံးကိုသုံးပြီး `for=a` Attribute ရှိတဲ့ Label ကို Select လုပ်ယူပါတယ်။ CSS `display` ကို `none` လို့ကြိုရေးပြီး ဖျောက်ထားတာဖြစ်လို့ `display` တန်ဖိုးကို `inline` လို့ပြန်ပေးလိုက်တဲ့တွက်ပေါ်လာမှာ ဖြစ်ပါတယ်။ b အတွက်လည်း အလားတူပဲ စစ်ပေးထားပါတယ်။ ဒါကြောင့် ရလဒ်က အခုလို ဖြစ်မှာပါ –



လိုချင်တဲ့ပုံစံရသွားပေမယ့် သိပ်တော့အဆင်မပြေသေးပါဘူး။ Message က တစ်ကြိမ်ပေါ်ပြီးရင် မှန်ကန်တဲ့တန်ဖိုး ရိုက်ထည့်လိုက်ရင်တောင်မှ ဆက်ပြနေဦးမှာပါ။ ဒါကြောင့် မှန်ကန်တဲ့တန်ဖိုး ရိုက်ထည့်လိုက်ရင် ပြန်ဖျောက်ပေးအောင် ရေးပါဦးမယ်။ ဒီအတွက် `<input>` ပေါ်မှာ Blur, Change, Keyup ဆိုတဲ့ Event (၃) မျိုးထဲက နှစ်သက်ရာတစ်ခုကို ရွေးသုံးနိုင်ပါတယ်။

Blur ကိုသုံးခဲ့ရင် လက်ရှိ `<input>` ရဲ့ Focus လွတ်သွားချိန်မှာ အလုပ်လုပ်ပေးမှာပါ။ Change ကတော့ ရိုက်ထည့်လိုက်တဲ့တန်ဖိုး ပြောင်းသွားချိန် အလုပ်လုပ်ပေးမှာပါ။ Keyup ကတော့ Keyboard ကနေ တစ်ခုခုနှိပ်လိုက်တာနဲ့ အလုပ်လုပ်ပေးမှာပါ။ Keyup နဲ့ နမူနာပေးပါမယ်။ ပိုအဆင်ပြေလို့ပါ။ Blur တွေ Change တွေကိုတော့ စမ်းကြည့်ချင်ရင် ကိုယ့်ဘာသာ ပြောင်းပြီးစမ်းကြည့်နိုင်ပါတယ်။ ဒီလိုရေးရမှာပါ –

JavaScript

```
document.querySelector("#a").onkeyup = function() {

    let a = parseInt(document.querySelector("#a").value)

    if( !isNaN(a) ) {
        document.querySelector("[for=a]")
            .style.display = "none"
    }
}

document.querySelector("#b").onkeyup = function() {

    let a = parseInt(document.querySelector("#b").value)

    if( !isNaN(a) ) {
        document.querySelector("[for=b]")
            .style.display = "none"
    }
}
```

`<input>` နှစ်ခုအတွက် နှစ်ခါရေးထားပါတယ်။ ကုဒ်ရဲ့သဘောက အတူတူပါပဲ။ Input တွေပေါ်မှာ Key တစ်ခုခုနှိပ်တာနဲ့ Number ဟုတ်မဟုတ်စစ်ပြီး ဟုတ်တာနဲ့ Label တွေရဲ့ `display` ကို `none` ပြောင်းပြီး ပြန်ဖျောက်ပေးလိုက်တာမို့လို့ အခုဆိုရင်တော့ အားလုံးပြည့်စုံသွားပြီဖြစ်ပါတယ်။ တစ်ဆင့်ချင်းစီပြခဲ့တာကို ကူးရေးရတာ အဆင်မပြေခဲ့ရင် ဒီမှာကုဒ်အပြည့်အစုံကို ဒေါင်းပြီးစမ်းကြည့်လို့ရပါတယ်။

– https://github.com/eimg/javascript-book

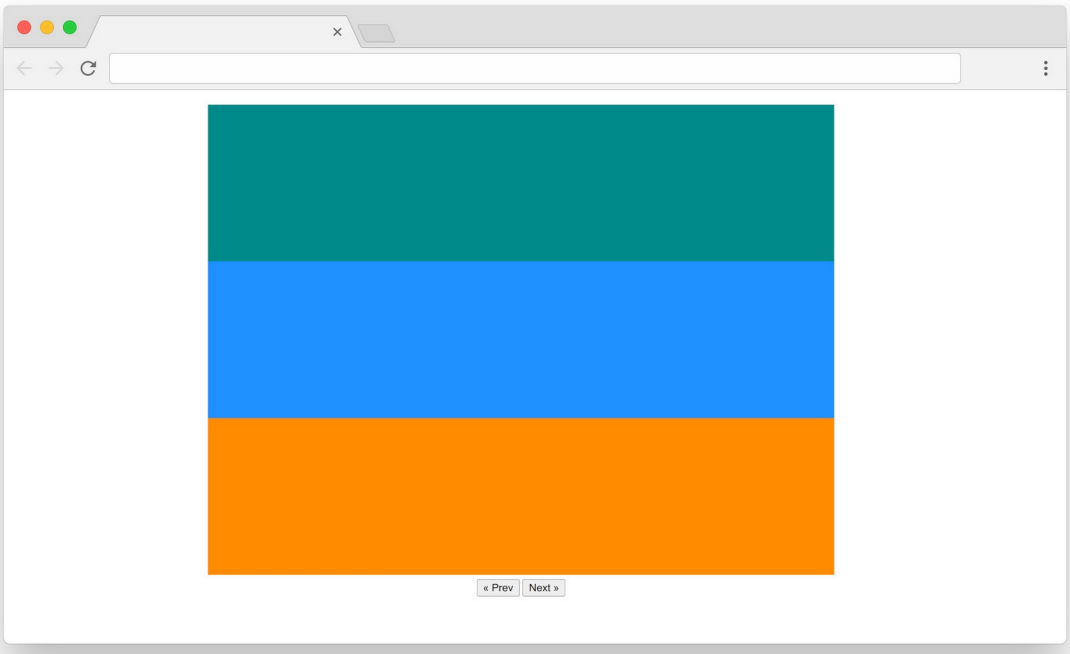
Carousel Sample

နောက်ထပ် စမ်းကြည့်စရာနမူနာလေးတစ်ခုအနေနဲ့ Carousel လေးတစ်ခုကို ဖန်တီးကြည့်ကြပါမယ်။ Carousel ဆိုတာ Slide Show ပုံစံ Box လေးတွေတစ်ခု တစ်ခုပြီးတစ်ခု ပြောင်းနေတဲ့ လုပ်ဆောင်ချက်လေးပါ။ သတ်မှတ်အချိန်နဲ့ သူ့အလိုအလျောက်ပြောင်းသလို ခလုပ်လေးနှိပ်နှိပ်ပြီးတော့လည်း ပြောင်းလို့ရနိုင်ပါတယ်။ ပထမတစ်ဆင့်အနေနဲ့ ဒီလိုလေး ရေးပေးလိုက်ပါ –

HTML & CSS

```
<!DOCTYPE html>
<html>
<head>
    <title>Title</title>
    <style>
        .boxes {
            width: 800px;
            margin: 20px auto;
            text-align: center;
        }

        .box {
            height: 200px;
        }

        #box1 {
            background: darkcyan;
        }

        #box2 {
            background:dodgerblue;
        }

        #box3 {
            background: darkorange;
        }
    </style>
</head>
<body>
    <div class="boxes">
        <div class="box" id="box1"></div>
        <div class="box" id="box2"></div>
        <div class="box" id="box3"></div>
        <div class="controls">
            <button id="prev">&laquo; Prev</button>
            <button id="next">Next &raquo;</button>
        </div>
    </div>
</body>
</html>
```

ဒါ `.box` ကလေး (၃) ခုနဲ့ `<button>` လေး (၂) ခုပါဝင်တဲ့ HTML Structure ဖြစ်ပြီး သင့်တော်တဲ့ CSS Style လေးတွေ ထည့်ရေးထားလို့ ရလဒ်က အခုလိုပုံစံ ဖြစ်မှာပါ။



ပြီးတော့မှာ `#box2` နဲ့ `#box3` ကို `display: none` နဲ့ ခဏဖျောက်ထားလိုက်ပါ။ CSS မှာ ဒီလို ပြောင်းပေးရမှာပါ။

CSS

```
#box1 {
    background: darkcyan;
    display: none;
}

#box2 {
    background: dodgerblue;
    display: none;
}
```

ဒါကြောင့် Box (၃) ခုမှာ (၁) ခုပဲ ကျန်မှာဖြစ်ပါတယ်။ Next ခလုပ်ကိုနှိပ်ရင် နောက် Box တစ်ခုကို ပြောင်းပြပြီး Prev ခလုပ်ကိုနှိပ်ရင် ရှေ့ Box တစ်ခုကို ပြောင်းပြအောင် အခုလိုလေး ရေးပေးလိုက်ပါ။

HTML & JavaScript

```
<script>
    let box = 1

    function next() {
        document.querySelector(`#box${box}`).style.display = "none"
        box++; if(box > 3) box = 1
        document.querySelector(`#box${box}`).style.display = "block"
    }

    function prev() {
        document.querySelector(`#box${box}`).style.display = "none"
        box--; if(box < 1) box = 3
        document.querySelector(`#box${box}`).style.display = "block"
    }

    document.querySelector("#next").onclick = next
    document.querySelector("#prev").onclick = prev

    setInterval(next, 5000)
</script>
```

ပထမဆုံး box အမည်နဲ့ Variable တစ်ခုကို အားလုံးရဲ့အပေါ်မှာ ကြေညာထားပါတယ်။ next() Function က အဲ့ဒီ box တန်ဖိုးနဲ့ Element ကို Select လုပ်ပြီးဖျောက်လိုက်လို့ နှိပ်လိုက်ရင် #box1 ပျောက်သွားမှာပါ။ ပြီးတော့မှ box++ နဲ့ တစ်တိုးပြီး Select ပြန်လုပ်ထားလို့ ဒီတစ်ခါ Select လုပ်မှာ #box2 ဖြစ်သွားပါပြီ။ ဒါကြောင့် #box2 ပေါ်လာမှာပါ။ Next ကိုနောက်တစ်ကြိမ်နှိပ်တဲ့အခါ box Variable တန်ဖိုးက 2 ဖြစ်နေပါပြီ။ ဒါကြောင့် #box2 ပျောက်သွားပြီး #box3 ပေါ်လာမှာပါ။ ဒီနည်းနဲ့ တစ်ချက်နှစ်တိုင်း နောက် Box တစ်ခုကို ပြောင်းပြတဲ့ လုပ်ဆောင်ချက်ကို ရသွားပါတယ်။ ဒါပေမယ့် Box က (၃) ခုပဲရှိလို့ box Variable တန်ဖိုး 3 ထက်ကျော်မှာစိုးလို့ 3 ထက်ကျော်ရင် 1 ကိုပြန်ပြောင်းပေးထားလိုက်လို့ နောက်ဆုံး Box ကို ရောက်ပြီးရင် ရှေ့ဆုံး Box ကို ပြန်ရောက်သွားမှာ ဖြစ်ပါတယ်။ ကုဒ်နမူနာမှာ Statement နှစ်ကြောင်းကို တစ်ကြောင်းထဲ ရောရေးထားလို့ Semicolon လေးခံထားတာကိုလည်း သတိပြုပါ။

prev() Function ရဲ့လုပ်ဆောင်ချက်က next() နဲ့အတူတူပါပဲ။ box Variable ကို 1 တိုးမယ့်အစား 1 နှုတ်ပေးလိုက်တာပါ။ အဲ့ဒီလိုနှုတ်တဲ့အခါ 1 ထက်ငယ်ရင် 3 လို့ပေးထားတဲ့အတွက် ရှေ့ဆုံးရောက်နေချိန် prev() ကို သွားလိုက်ရင် နောက်ဆုံးကိုရောက်သွားမှာပါ။ ဒီနည်းနဲ့ ရှေ့နောက် သွားလို့ရတဲ့ Function နှစ်ခုရသွားပါတယ်။

ပြီးတော့မှ အဲ့ဒီ Function တွေကို သက်ဆိုင်ရာ Button မှာတွဲပေးလိုက်ပါတယ်။ ဒါတင်မက `setIntervel()` နဲ့လည်း `next` ကို Run ပေးထားလို့ နမူနာအရ ၅ စက္ကန့်တစ်ခါ အလိုအလျောက် ပြောင်းနေတဲ့ Carousel လုပ်ဆောင်ချက်လေးကို ရရှိသွားပြီပဲ ဖြစ်ပါတယ်။

ဒီလိုမျိုး နားလည်ရသိပ်မခက်ဘဲ လက်တွေ့အသုံးဝင်တဲ့ နမူနာလေးတွေကိုကြိုရေးပြီး တင်ထားပေးပါ မယ်။ ဒီလိပ်စာမှာ Download ရယူပြီး၊ တစ်ခုပြီးတစ်ခု လေ့လာစမ်းသပ် ကြည့်စေချင်ပါတယ်။

– https://github.com/eimg/javascript-book

Element List

အခုနမူနာတွေရေးတဲ့အခါ သုံးရပိုလွယ်တဲ့ `querySelector()` ကိုပဲ တောက်လျှောက်သုံးခဲ့ပါတယ်။ လိုချင်တဲ့ Element ကိုတန်းရလို့ပါ။ တစ်ကယ့်လက်တွေ့မှာ `querySelectorAll()` ကိုပိုအသုံးများ နိုင်ပါတယ်။ Selector တစ်ခုကပြန်ပေးတဲ့ Element တွေအများကြီး ဖြစ်နိုင်ပါတယ်။ ဥပမာ `ul > li` ဆိုရင် `<ul>` ထဲမှာ ရှိသမျှ `<li>` တွေအကုန်ရမှာပါ။ ဒါပေမယ့် `querySelector()` နဲ့ဆိုရင် တစ်ခုပဲ ရပါလိမ့်မယ်။ တစ်ကယ် အကုန်လိုချင်ရင်တော့ `querySelectorAll()` ကိုသုံးမှ ရပါလိမ့်မယ်။

`querySelectorAll()` နဲ့ Select လုပ်ယူတဲ့အခါ ပြန်ရမယ့် ရလဒ်က Array နဲ့ဆင်တူတဲ့ Node List ကိုပြန်ရမှာဖြစ်ပါတယ်။ Array နဲ့ဆင်တူပေမယ့် တစ်ကယ် Array တော့မဟုတ်ပါဘူး။ ဒါကြောင့် သုံးတဲ့ အခါ Array ကဲ့သို့ သုံးနိုင်ပေမယ့် `map()` တို့ `filter()` တို့လို Array Method တွေ အလုပ်မလုပ်ပါဘူး။ ဒါပေမယ့် Iterable အမျိုးအစားဖြစ်လို့ `for-of` Loop နဲ့ သူ့ကို Loop လုပ်လို့ရနိုင်ပါတယ်။

HTML & JavaScript

```
<ul>
    <li>Item One</li>
    <li>Item Two</li>
    <li>Item Three</li>
    <li>Item Four</li>
</ul>
```

```
<script>
    let items = document.querySelectorAll("ul > li")

    for(li of items) {
        console.log(li.textContent)
    }
</script>
```

ဒါဟာ `<li>` Element တွေပါဝင်တဲ့ Node List ကို `for-of` နဲ့ Loop လုပ်ပြီး သူ့ရဲ့ Content ကို Console မှာ တန်းစီရိုက်ထုတ်လိုက်တာပါ။ ဒါကြောင့် Browser Console ကိုဖွင့်ကြည့်ရင် အခုလိုရလဒ်ကို တွေ့မြင်ရမှာပါ။

```
Item One
Item Two
Item Three
Item Four
```

အကယ်၍ Node List ပေါ်မှာ `map()` လိုမျိုး Loop လုပ်လို့ရတဲ့ Method သုံးချင်ရင် `forEach()` ကိုသုံးလို့လည်း ရနိုင်ပါသေးတယ်။ ဒီလိုပါ –

HTML & JavaScript

```
<ul>
    <li>Item One</li>
    <li>Item Two</li>
    <li>Item Three</li>
    <li>Item Four</li>
</ul>

<script>
    let items = document.querySelectorAll("ul > li")
    items.forEach(li => li.textContent)
</script>
```

တူညီတဲ့ရလဒ်ကို ရမှာဖြစ်ပါတယ်။ ဒီနေရာမှာ တစ်ခုသတိပြုရမှာက `forEach()` က Item တစ်ခုချင်းစီပေါ်မှာ ပေးလိုက်တဲ့ Callback Function ကို အလုပ်လုပ်သွားပေမယ့် ရလဒ်ကို `map()` လို Array အနေနဲ့ ပြန်မပေးပါဘူး။ ဒါကြောင့် လိုချင်တဲ့အလုပ်ခိုင်းလို့ ရပါတယ်၊ Return Value တော့ ရှိမှာ မဟုတ်ပါဘူး။

လက်စနဲ့ Array တစ်ခုကို သုံးပြီး Element List တစ်ခု တည်ဆောက်ဖော်ပြစေတဲ့ ကုဒ်လေးလည်း နမူနာ ထည့်ပြလိုက်ချင်ပါတယ်။ ဒီလိုပါ –

HTML & JavaScript

```
<ul></ul>

<script>
    let fruits = ["Apple", "Orange", "Mango"]
    let ul = document.querySelector("ul")

    fruits.map(fruit => {
        let li = document.createElement("li")
        li.textContent = fruit
        ul.appendChild(li)
    })
</script>
```

ဒါဟာ Array တစ်ခုထဲမှာပါတဲ့ Item တွေကိုသုံးပြီး `<ul>` Element ထဲမှာ `<li>` Element တွေ တစ်ခုပြီး တစ်ခုတည်ဆောက်ပြီး ထည့်ပေးလိုက်တာပါ။ ဒီတစ်ခါတော့ Array ပေါ်မှာ အလုပ်လုပ်တာမို့လို့ `map()` ကိုသုံးထားပါတယ်။

Wrapping Up

အခုလို DOM Manipulation လုပ်ငန်းတွေနဲ့ပက်သက်ရင် အရမ်းကောင်းတဲ့ JavaScript Library နည်း ပညာတစ်ခုရှိပါတယ်။ jQuery လို့ခေါ်ပါတယ်။ လူသုံးတော်တော်များတဲ့ နည်းပညာတစ်ခုပါ။ ဘယ်လောက်တောင် လူသုံးများသလဲဆိုရင် အင်တာနက်ပေါ်မှာရှိသမျှ ဝဘ်ဆိုက်တွေရဲ့ ထက်ဝက်နီးပါး က jQuery ကို အသုံးပြုထားကြပါတယ်။

jQuery က JavaScript မှာ `querySelector()` တို့ `querySelectorAll()` တို့ မရှိခင်ကတည်းက CSS Selector အတိုင်း Element တွေကို Select လုပ်လို့ရအောင် စီစဉ်ပေးထားတာပါ။ JavaScript မှာ `forEach()` မရှိခင်ကတည်းက အလားတူ လုပ်ဆောင်ချက်မျိုးတွေ သူမှာ ရှိနေတာပါ။ ဒါက အခုပြောလက်စရှိတဲ့ နမူနာတွေလောက်ကိုသာ ရွေးထုတ်ပြောတာပါ။ တစ်ကယ်တော့ အသုံးဝင်ပြီး မူလ JavaScript မှာ မပါတဲ့၊ မရှိတဲ့၊ လုပ်ဆောင်ချက်တွေ တော်တော်များများကို jQuery က ဖန်တီးပေး ထားပါတယ်။ ပြီးတော့၊ အရင်က Browser တွေတစ်ခုနဲ့တစ်ခု အလုပ်လုပ်ပုံ မတူကွဲပြားမှုတွေ ရှိနေစဉ်

ကာလမှာ၊ ရေးလိုက်တဲ့ကုဒ်တွေ Browser အားလုံးမှာ တူညီစွာ အလုပ်လုပ်အောင်လည်း jQuery က စီစဉ်ပေးထားပါသေးတယ်။ ဒါကြောင့် တော်တော်အသုံးဝင်ပြီး မရှိမဖြစ်နည်းပညာဖြစ်ခဲ့ပါတယ်။

အခုနောက်ပိုင်းမှာတော့ Internet Explorer လို အလွန်နှေးပြီး ပြဿနာတွေများတဲ့ Browser မျိုးကို လူသုံးနည်းသွားသလို JavaScript ကိုယ်တိုင်မှာလည်း jQuery က လုပ်ပေးနိုင်တဲ့ လုပ်ဆောင်ချက်တွေ တစ်ခါထဲ ပါဝင်လာပါပြီ။ ဒါကြောင့်အခုချိန်မှာ jQuery ကို ထည့်သွင်းလေ့လာထားရင် ကောင်းပေမယ့် မဖြစ်မနေ လိုအပ်တာမျိုး မဟုတ်တော့ပါဘူး။

jQuery နဲ့ ပြိုင်တူလိုလို ထွက်ပေါ်ခဲ့တဲ့ အခြား JavaScript နည်းပညာ အမြောက်အများ ရှိခဲ့သလို jQuery ရဲ့နောက်မှာ ထပ်ဆင့်ထွက်ပေါ်လာတဲ့ JavaScript နည်းပညာတွေလည်း အမြောက်အမြား ရှိပါသေးတယ်။ BackboneJS, AngularJS, EmberJS စသည်ဖြင့် ထင်ရှားခဲ့ကြပါတယ်။ လက်ရှိ အထင်ရှားဆုံးနည်းပညာတွေကတော့ React နဲ့ Vuejs တို့ပဲ ဖြစ်ပါတယ်။ React အကြောင်းကို **React လိုတိုရှင်း** စာအုပ်မှာ ဆက်လက်လေ့လာနိုင်ပါတယ်။

ဒါက Development ပိုင်းကိုပဲ ပြောတာပါ။ Tooling ခေါ် ဆက်စပ်နည်းပညာတွေလည်း ရှိပါသေးတယ်။ ESLint လို JavaScript ကုဒ်တွေထဲမှာ အမှားရှာပေးနိုင်တဲ့ နည်းပညာတွေ၊ Webpack လို Node Module တွေကို Web မှာသုံးလို့ရအောင် စီမံပေးတဲ့ နည်းပညာတွေ၊ Babel လို အသစ်တီထွင်ထားတဲ့ ES Feature သစ်တွေကို လောလောဆယ် Support မလုပ်သေးတဲ့ Browser တွေမှာ အလုပ်လုပ်အောင် စီမံပေးနိုင်တဲ့ နည်းပညာတွေ၊ NPM လို JavaScript Module တွေ Package တွေကို စီမံပေးနိုင်တဲ့ နည်းပညာတွေ၊ စသည်ဖြင့် ရှိနေပါတယ်။

JavaScript ဟာ တော်တော် လေးကျယ်ပြန့်တဲ့ ဘာသာရပ်တစ်ခုပါ။ အခုဒီစာအုပ်ပေးတဲ့ ဗဟုသုတပေါ်မှာ အခြေခံပြီး နောက်အဆင့် နောက်အဆင့်တွေကို တစ်ဆင့်ခြင်း ဆက်လက် တက်လှမ်းသွားရမှာပဲ ဖြစ်ပါတယ်။

အခန်း (၁၂) – Debugging

ကုဒ်ထဲမှာ မသိလိုက်ဘဲ ပါသွားတဲ့ အမှားလေးတွေကို Bug လို့ခေါ်ပါတယ်။ ဟိုးရှေးရှေးက ကွန်ပျူတာဆိုတာ အခုလို စားပွဲတင်ကွန်ပျူတာ မဟုတ်သေးဘဲ အခန်းတစ်ခန်းလုံးစာ အပြည့်ယူတဲ့ စက်ကြီးတွေပါ။ အဲ့ဒီခေတ်က ကွန်ပျူတာ အလုပ်မလုပ်လို့ အဖြေရှာကြည့်လိုက်တာ ပိုးဟပ်တစ်ကောင် စက်ထဲမှာဝင်နေလို့ အလုပ်မလုပ်တာကို သွားတွေ့ရာကနေ နောက်ပိုင်းမှာ အလုပ်မလုပ်အောင် ပြဿနာပေးနေတဲ့ အမှားလေးတွေကို Bug လို့ခေါ်တဲ့ အလေ့အထ ဖြစ်သွားခဲ့တာလို့ ဆိုပါတယ်။ ဒါကြောင့် Debug လုပ်တယ်ဆိုတာ Bug ကို ဖယ်ထုတ်လိုက်တာ၊ ရှင်းလိုက်တာပါ။ အမှားကို မှန်အောင်ပြင်လိုက်တာပါ။

JavaScript ကုဒ်တွေရေးတဲ့အခါ Console မှာ တိုက်ရိုက်ရေးပြီး စမ်းကြည့်တဲ့ ကုဒ်တွေက သိပ်ပြဿနာမရှိပါဘူး။ မှားရင် Error တက်လို့ မှားမှန်း ချက်ချင်းသိရပါတယ်။ လုံးဝအတိအကျကြီး မဟုတ်ပေမယ့် ဘာကြောင့်မှားလဲဆိုတာကို Error Message တွေပြပေးလို့ အဲ့ဒီ Error Message တွေကို လေ့လာပြီး ကိုယ့်ကုဒ်ကို လိုအပ်သလို မှန်အောင် ပြင်ပေးလိုက်လို့ရနိုင်ပါတယ်။

Browser မှာ Run တဲ့အခါမှာတော့ ရေးထားတဲ့ JavaScript ကုဒ်မှားနေရင် Browser က Error မပြပါဘူး။ ဒါကြောင့် ရေးပြီးစမ်းကြည့်လိုက်လို့ အလုပ်မလုပ်တဲ့အခါ ဘာကြောင့်အလုပ်မလုပ်မှန်း မသိတော့တာမျိုးတွေ ဖြစ်ကြပါတယ်။ တစ်ကယ်တော့ Error တွေကို လုံးဝမပြတာ မဟုတ်ပါဘူး။ Browser မြင်ကွင်းမှာ မပြပေမယ့် Console ထဲမှာတော့ ပြပေးပါတယ်။ ဒါကြောင့် Debug လုပ်ပြီး အမှားရှာနိုင်ဖို့အတွက် ရေးတဲ့ကုဒ် အလုပ်မလုပ်ရင် Browser Console ဖွင့်ကြည့်ပြီး အဖြေရှာရတယ် ဆိုတာကို ဦးဆုံးမှတ်ထားဖို့ လိုပါတယ်။

အခုလို စာအုပ်တွေရေးသလို သင်တန်းတွေဖွင့်ပြီး စာတွေလည်း သင်နေတော့ အမြဲကြုံရပါတယ်။ အလုပ်မလုပ်ရင် Console ကို ဖွင့်ကြည့်ပါ လို့ ဘယ်လောက်ပဲ ထပ်ခါထပ်ခါ ပြောထားထား၊ အဆင်မပြေတာနဲ့ ဘာ

လုပ်လို့ ဘာကိုင်ရမှန်းမသိ စိတ်ညစ်ပြီး၊ "ဆရာ မရဘူး" တို့ "ဆရာ အလုပ်မလုပ်ဘူး" တို့ဆိုတာမျိုးတွေ အမြဲတမ်း မေးကြပါတယ်။ မေးတာက ပြဿနာမဟုတ်ပါဘူး။ "Console ကိုဖွင့်ကြည့်ပြီးပြီလား" လိုပြန်မေးလိုက်ရင် အများစုက မကြည့်ရသေးပါဘူး။ တစ်ကယ်တမ်း Console ဖွင့်ကြည့်လိုက်တဲ့အခါ မှားရတဲ့အကြောင်းရင်းကို သိသွားပြီး ကိုယ်ဘာသာမှန်အောင် ပြင်လိုက်နိုင်ကြတာ များပါတယ်။

အစပိုင်းလေ့လာစရေးတဲ့ ကုဒ်တွေဆိုတာ ဘာမှရှုပ်ထွေးတဲ့ကုဒ်တွေ မဟုတ်သေးပါဘူး။ ဒီအဆင့်မှာ တော်ယုံတန်ယုံ အမှားလောက်က Console မှာပြတဲ့ Error တွေနဲ့တင် အဖြေထွက်ပါတယ်။ တစ်ခါတစ်လေတော့ ဘာ Error မှ မပြဘဲ အလုပ်မလုပ်တဲ့ ပြဿနာမျိုးတွေတော့ ရှိတတ်ပါတယ်။ အဲ့ဒီလို ရှိလာတဲ့အခါ မမှားဘူး ထင်ရပေမယ့် မှားကြတဲ့အမှားလေးတွေ ရှိပါတယ်။ ဘာလဲဆိုတော့ ကုဒ်တွေရေးနေတာက ဖိုင်တစ်ခု၊ တစ်ကယ်တမ်း ဖွင့်စမ်းနေတာက ဖိုင်တစ်ခု ဖြစ်ကြတာကိုလည်း မကြာမကြာ ကြုံရပါတယ်။ မဖြစ်လောက်ပါဘူး မထင်နဲ့၊ တစ်ခါတစ်လေ မဟုတ်ပါဘူး၊ ခဏခဏကို ဖြစ်ကြတာပါ။ ဒီတော့ ရေးချင်ရာရေး၊ ပြင်ချင်သလိုပြင်၊ အလုပ်ကို မလုပ်ဘူးဖြစ်နေတာပေါ့။ ဘယ်လုပ်ပါ့မလဲ၊ ဖွင့်စမ်းနေတာက လက်ရှိရေးနေတဲ့ ဖိုင်မှ မဟုတ်တာ။ ဒါကြောင့် Error လည်းမပြဘူး၊ အလုပ်လည်းမလုပ်ဘူးဆိုရင် ရေးတဲ့ဖိုင်နဲ့ စမ်းတဲ့ဖိုင်တူရဲ့လားဆိုတာကို အရင်ဆုံး သေချာပြန်စစ်ပါ။

နောက်ပြဿနာတစ်ခုကတော့၊ ရေးတဲ့ဖိုင်နဲ့ စမ်းတဲ့ဖိုင် တူတယ်။ Browser မှာ Refresh လုပ်ဖို့ မေ့ပြီး ဒီတိုင်းစမ်းနေမိတာ ဆိုတာမျိုးလည်း ဖြစ်တတ်ပါသေးတယ်။ လေ့လာသူတွေမပြောနဲ့ အတွေ့အကြုံရှိသူတွေတောင် မကြာခဏ ဖြစ်တတ်ကြပါသေးတယ်။ Browser မှာ Refresh မလုပ်ရင်တော့ ရေးလိုက်တဲ့ကုဒ်က အသက်ဝင်မှာ မဟုတ်ပါဘူး။ နဂိုအဟောင်းကြီးကိုပဲ အလုပ်လုပ်နေမှာပါ။ ဒါကြောင့် ကိုယ်ကတော့ ပြင်လိုက်တယ် ရလဒ်မှာ ပြောင်းမသွားဘူးဆိုတာမျိုး ဖြစ်တတ်ပါတယ်။ ဒါကြောင့် Browser ကို Refresh လုပ်ပြီးမှ စမ်းကြည့်ဖို့ကိုလည်း မမေ့ဖို့သတိပြုရပါမယ်။

ရေးတဲ့ဖိုင်လည်းမှန်တယ်၊ စမ်းတဲ့ဖိုင်လည်း မှန်တယ်၊ Refresh တွေဘာတွေလည်း သေချာလုပ်တယ်။အဲ့ဒါလည်း အလုပ်မလုပ်သေးဘူး၊ Error လည်းမပြဘူး ဆိုတာမျိုး ဖြစ်လာရင်တော့ `console.log()` Method ကို အားကိုးရမှာပဲ ဖြစ်ပါတယ်။ ဥပမာအနေနဲ့ မှားနေတဲ့ ကုဒ်လေးတစ်ခု ပေးချင်ပါတယ်။

HTML & JavaScript

```
<input type="text">
<label></label>

<script>
    document.querySelector("input").onekeyup = function() {
        let count = document.querySelector("input").value.lenght
        if(count) {
            document.querySelector("label").testContent = count
        }
    }
</script>
```

<input> မှာ တစ်ခုခုရိုက်ထည့်လိုက်တာနဲ့ စာလုံးအရေအတွက်ကို <label> ထဲမှာ ပြအောင် ရေးထားတဲ့ ကုဒ်ပါ။ ဒီကုဒ်ဟာ မှားနေတဲ့အတွက် အလုပ်လုပ်မှာ မဟုတ်ပါဘူး။ ပြဿနာက Error Message လည်း ပြမှာ မဟုတ်ပါဘူး။ ကိုယ်ဘာသာဖတ်ကြည့်ပြီး အမှားကိုတွေ့အောင်ရှာရပါတော့မယ်။ ဒီကုဒ်လောက်မှာ ရှာကောင်းရှာနိုင်ပေမယ့် ကုဒ်တွေများလာရင် တစ်လုံးချင်းဖတ်ပြီး ရှာဖို့ဆိုတာ မလွယ်တော့ပါဘူး။ ဒါကြောင့် ဘယ်အပိုင်း အလုပ်လုပ်တယ်၊ ဘယ်အပိုင်း အလုပ်မလုပ်မဘူးဆိုတာ ပေါ်လွင်အောင် လက်ရှိကုဒ်ရဲ့ ကြားထဲမှာ console.log() လေးတွေ လိုက်ထုတ်ကြည့်လို့ ရနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript

```
console.log("JavaScript working")

document.querySelector("input").onekeyup = function() {

    console.log("Keyup working")

    let count = document.querySelector("input").value.lenght

    if(count) {

        console.log("Inside if block")

        document.querySelector("label").testContent = count
    }
}
```

console.log() Statement တွေပါသွားလို့ **JavaScript working** ဆိုတဲ့ဖော်ပြချက် Console မှာပေါ်ရင် ဒီအဆင့်ထိ အလုပ်လုပ်တယ်ဆိုတာ သေချာသွားပါပြီ။ keyup လုပ်ဆောင်ချက်သာ အလုပ်လုပ်မယ်ဆိုရင် **Keyup working** ဆိုတဲ့ဖော်ပြချက် Console မှာထွက်ရပါမယ်။ ပေးထားတဲ့ နမူနာမှာ ထွက်မှာ မဟုတ်ပါဘူး။ ဒါဆိုရင် သူက Error မပြပေမယ့် ကိုယ့်အစီအစဉ်နဲ့ကိုယ် Keyup လုပ်ဆောင်ချက် အလုပ်မ

လုပ်တာ သိသွားပြီမို့လို့ အခဲ့ဒီအပိုင်းကို ဦးစားပေးအဖြေရှာလိုက်လို့ ရသွားပါပြီ။ နမူနာမှာ `onkeyup` ဖြစ်ရမှာကို `onekeyup` ဖြစ်နေပါတယ်။ အခဲ့ဒါလေးပြင်ပြီး နောက်တစ်ကြိမ် စမ်းကြည့်ရင် **Keyup working** ဆိုတဲ့ဖော်ပြချက်ကို တွေ့မြင်ရမှာပါ။ ဒါဆိုရင် ပြဿနာတစ်ခု ပြေလည်သွားပါပြီ။

အမှားတွေကျန်နေသေးလို့ အလုပ်တော့လုပ်ဦးမှာ မဟုတ်ပါဘူး။ အကယ်၍ **Inside if block** ဆိုတဲ့ ရလဒ်ထွက်ပေါ်မယ်ဆိုရင် `count` တန်ဖိုးမှန်လို့ပါ။ မပေါ်ရင်တော့ `count` တန်ဖိုးမမှန်လို့ပါ။ ဒါဆိုရင် `count` တန်ဖိုးယူထားတဲ့ Statement ကို စစ်လိုက်ယုံပါပဲ။ နမူနာမှာ `value.length` ဖြစ်ရမယ့်အစား `value.lenght` ဖြစ်နေပါတယ်။ ပြင်ပေးလိုက်ရင် မှန်သွားလို့ **Inside if block** ဆိုတဲ့ ဖော်ပြချက်ကို တွေ့မြင်ရပါလိမ့်မယ်။

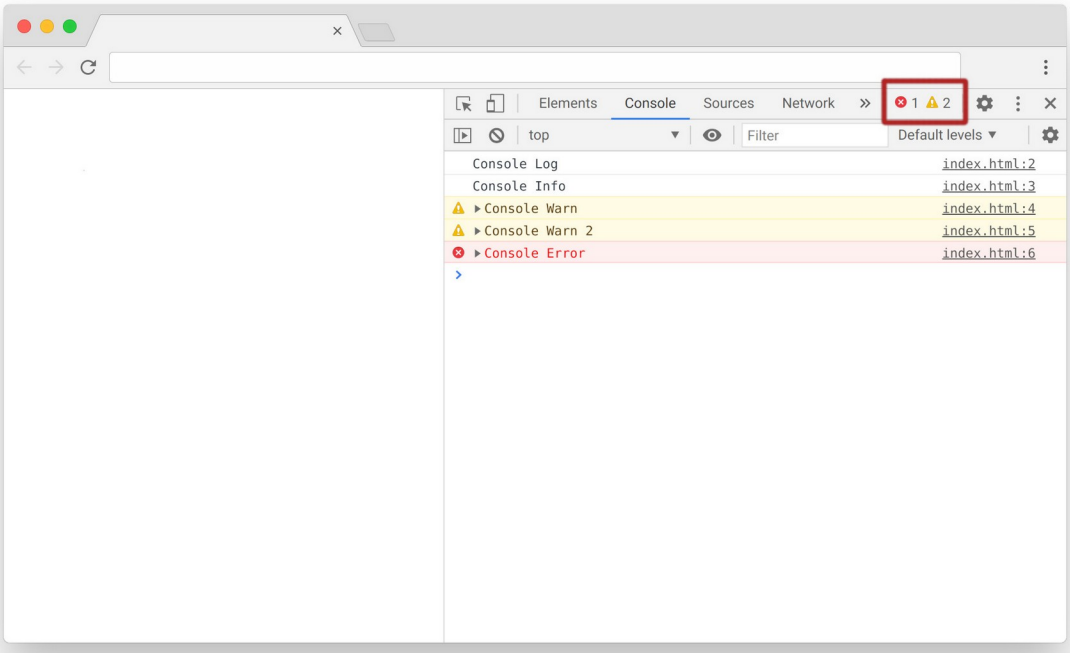
ဒါပေမယ့် အမှားကျန်နေသေးလို့ လိုချင်တဲ့ရလဒ်အမှန် မရသေးပါဘူး။ **Inside if block** ဆိုတဲ့ ဖော်ပြချက်ကြောင့် အခဲ့ဒီနားထိ ရောက်နေပြီ၊ မှန်နေပြီဆိုတာ သေချာနေပြီမို့လို့ စစ်စရာ တစ်နေရာပဲ ကျန်ပါတော့တယ်။ သတိထားကြည့်လိုက်ရင် တွေ့ရပါလိမ့်မယ်။ `textContent` ဖြစ်ရမယ့်အစား `testContent` ဖြစ်နေပါတယ်။ အခဲ့ဒါလေးပြင်ပြီး စမ်းကြည့်လိုက်ရင်တော့ အားလုံးမှန်သွားပြီမို့လို့ လိုချင်တဲ့ရလဒ်အမှန်ကို ရရှိသွားမှာပဲ ဖြစ်ပါတယ်။

`console.log()` အပြင် အသုံးဝင်တဲ့ တစ်ခြား Method လေးတွေ ရှိပါသေးတယ်။

`console.`**`info`**`()` – `console.log()` နဲ့အတူတူပါပဲ။ Firefox Browser မှာဆိုရင် ရှေ့ကနေ Info Icon လေးထည့်ပြီး ပြပေးပါတယ်။ Chrome မှာတော့ မပြပါဘူး။

`console.`**`warn`**`()` – `console.log()` နဲ့အတူတူပါပဲ။ စာလုံးအဝါရောင်နဲ့ ရှေ့ကနေ Warning Icon လေးထည့်ပြီး ပြပေးပါတယ်။

`console.`**`error`**`()` – `console.log()` နဲ့အတူတူပါပဲ။ စာလုံးအနီရောင်နဲ့ ရှေ့ကနေ Error Icon လေးထည့်ပြီး ပြပေးပါတယ်။



နမူနာပုံကိုကြည့်လိုက်ရင် `console.log()`, `console.info()`, `console.warn()` နဲ့ `console.error()` တို့ကို သုံးပြထားပါ။ Console မှာပြတဲ့ ဖော်ပြချက်လေးတွေ အရောင်ပြောင်းသွားယုံသာမက သင့်တော်တဲ့ Icon လေးတွေ တွဲပြတာကိုတွေ့ရမှာပါ။ ပြီးတော့ ဟိုးအပေါ်ညာဘက်နားမှာ Warning နဲ့ Error အရေအတွက်ကိုလည်း ပြပေးနေတာကို တွေ့ရပါလိမ့်မယ်။

`console.`**`table`**`()` – Array တွေ Object တွေမှာပါတဲ့ Data တွေကို Table လေးတစ်ခုနဲ့ ကြည့်ကောင်းအောင် ပြပေးပါတယ်။ ရိုးရိုး `console.log()` နဲ့တစ်ချို့ Structure Data တွေကို ကြည့်ရခက်တယ်ထင်ရင် `console.table()` နဲ့ ထုတ်ကြည့်နိုင်ပါတယ်။

`console.`**`trace`**`()` – Function တွေထဲမှာ ဒီ Statement ကို ရေးထားမယ်ဆိုရင် Function ကို ဘယ်ကခေါ်သလဲဆိုတဲ့ခေါ်ယူမှု အဆင့်ဆင့်ကို ပြပေးပါတယ်။

`console.`**`time`**`()`, `console.`**`timeEnd`**`()` – တစ်ချို့ကုဒ်တွေ အလုပ်လုပ်တာ ဘယ်လောက်ကြာသလဲ သိချင်ရင် ဒီနှစ်ခုကို တွဲသုံးနိုင်ပါတယ်။ ကုဒ်ရဲ့အပေါ်မှာ `console.time()` ကို ရေးပြီး ကုဒ်ရဲ့ အောက်မှာ `console.timeEnd()` ကိုရေးထားမယ်ဆိုရင် ကြာချိန်ကို ဖော်ပြပေးမှာ ဖြစ်ပါတယ်။

ဒီလို Console မှာထုတ်ကြည့်ပြီး Debug လုပ်တဲ့နည်းဟာ စနစ်ကျတဲ့နည်း မဟုတ်ပေမယ့် လူတိုင်းသုံးနေတဲ့နည်းဖြစ်ပါတယ်။ စီနီယာအဆင့် ပရိုဂရမ်မာတွေကိုယ်တိုင် ဒီနည်းသုံးတဲ့သူတွေကို ဟာသလုပ်ပြီး နောက်ကြပေမယ့် သူတို့ကိုယ်တိုင် ဒီနည်းကို လက်မလွှတ်နိုင်ကြပါဘူး။

Debug လုပ်ဖို့ ထည့်ထားတဲ့ `console.log()` အပါအဝင် `console` Method တွေကို Debug လုပ်ပြီး နောက် ပြန်ဖြုတ်ဖို့တော့ မမေ့ပါနဲ့။ စနစ်မကျဘူးဆိုတာ ဒါကိုပြောတာပါ။ ကိုယ့်ဘာသာ Manual ထည့်ပြီး စမ်းထားမိတော့၊ ပြီးတဲ့အခါမှာလည်း ကိုယ့်ဘာသာပဲ Manual ပြန်ထုတ်ပေးရမှာပါ။ မထုတ်မိရင် အဲ့ဒီလို စမ်းထားတဲ့ Log တွေက ကုဒ်ကို Run လိုက်တိုင်း Console မှာ အမြဲတမ်း လာပေါ်နေမှာပါ။

တစ်ကယ့်ပရောဂျက်ကြီးတွေမှာတော့ ပိုပြီးစနစ်ကျတဲ့ Debugging Tool တွေကို သုံးရမှာဖြစ်ပေမယ့်၊ လောလောဆယ် လေ့လာဆဲအဆင့်မှာတော့ ဒီနည်းနဲ့တင် အဆင်ပြေနေပါပြီ။

နိဂုံးချုပ်

ဒီစာအုပ်ဟာ လိုတိုရှင်းစာအုပ် စီးရီးထဲမှာ တစ်အုပ်အပါအဝင် ဖြစ်ပါတယ်။ ရှေ့ပိုင်းမှာ **React လိုတိုရှင်း**၊ **Laravel လိုတိုရှင်း**၊ **API လိုတိုရှင်း** နဲ့ **Bootstrap လိုတိုရှင်း** ဆိုတဲ့စာအုပ် (၄) အုပ်ထွက်ထားပြီး ဖြစ်ပါတယ်။ မူလက စီးရီးတစ်ခုအနေနဲ့ ရေးဖို့စတင်ခဲ့တာ မဟုတ်လို့ အစကနေအဆုံး အစီအစဉ်အတိုင်း မဟုတ်ဘဲ စာရေးသူ ဓာတ်ကျရာအပိုင်းကနေ စရေးခဲ့တာပါ။ Covid-19 ကြောင့် အလုပ်တွေနားထားချိန်၊ အချိန်အားရနေတုံးမှာ၊ ရေးလက်စနဲ့ မထူးပါဘူးဆိုပြီး စီးရီးတစ်ခုအဖြစ် လိုအပ်တဲ့အပိုင်းတွေကို နောက်မှချိတ်ဆက်ဖြည့်စွက် ရေးသားခဲ့တာပါ။ နောက်ထပ်တစ်အုပ်အနေနဲ့ **PHP လိုတိုရှင်း** ကို ဆက်လက်ရေးသား သွားဦးမှာဖြစ်ပါတယ်။

အသင့်အတင့် လေ့လာဖူးသူတွေကတော့ ကိစ္စမရှိပါဘူး။ မိမိနှစ်သစ်ရာ အကြောင်းအရာကို နှစ်သက်ရာအစီအစဉ်နဲ့ လေ့လာနိုင်ပါတယ်။ အခုမှစလေ့လာမယ့်သူတွေကတော့ ဒီအစီအစဉ်အတိုင်း ဖတ်ရှုလေ့လာသင့်ပါတယ်။

၁။ Bootstrap လိုတိုရှင်း
၂။ JavaScript လိုတိုရှင်း
၃။ PHP လိုတိုရှင်း
၄။ Laravel လိုတိုရှင်း
၅။ React လိုတိုရှင်း
၆။ API လိုတိုရှင်း

Bootstrap လိုတိုရှင်း နဲ့ ဒီစာအုပ် **JavaScript လိုတိုရှင်း** တို့ဟာ အခြေခံအဆင့် စာအုပ်တွေဖြစ်ပါတယ်။ **PHP လိုတိုရှင်း** ကတော့ အလယ်အလတ်အဆင့် ဖြစ်မှာပါ။ **Laravel လိုတိုရှင်း** နဲ့ **React လိုတိုရှင်း** တို့ ကတော့ လုပ်ငန်းခွင်သုံးနည်းပညာတွေ ဖြစ်သွားပါပြီ။ **API လိုတိုရှင်း** ကတော့ အဆင့်မြင့်ပိုင်းလို့ ပြောလို့ရ နိုင်ပါတယ်။ ဒီကြောင့် ဒီအစီအစဉ်အတိုင်းသာ စနစ်တကျ လေ့လာသွားမယ်ဆိုရင် အခြေခံအဆင့်ကနေ လုပ်ငန်းခွင်ဝင် Web Developer တစ်ဦးဖြစ်တဲ့ အဆင့်ထိ ရောက်ရှိသွားနိုင်ပါတယ်။ ဒီစီးရီးထဲမှာ Front-End ပိုင်းတွေ Back-End ပိုင်းတွေ အကုန်စုစည်းပါဝင်သွားတာပါ။

အကယ်၍စာဖတ်သူက Web Development ပိုင်းကို မသွားလိုဘဲ အခြား Software Development လမ်းကြောင်းတစ်မျိုးဘက်ကို သွားချင်တာတာဆိုရင်လည်း ဒီစာအုပ်ကနေရရှိလိုက်တဲ့ ပရိုဂရမ်းမင်း အခြေခံ သဘောသဘာဝတွေဟာ စာဖတ်သူအတွက် ဆက်လက်အသုံးဝင်သွားပါလိမ့်မယ်။

ထွက်သမျှစာအုပ်တွေကို တစ်စိုက်မတ်မတ် ဖတ်ရှုအားပေးကြသူများ အပါအဝင် စာဖတ်ပရိတ်သတ်များ အားလုံးကို ကျေးဇူးအထူးတင်ကြောင်းပြောရင် နိဂုံးချုပ်အပ်ပါတယ်။ အားလုံးပဲ ရောဂါကပ်ဘေးတွေ ကနေ ကင်းဝေးပြီး မိမိတို့မျှော်မှန်းထားတဲ့ အောင်မြင်မှုများကို ရရှိနိုင်ကြပါစေ။

အိမောင် (Fairway)

၂၀၂၀ ပြည့်နှစ်၊ ဒီဇင်ဘာလ (၁၅) ရက်နေ့တွင် ရေးသားပြီးစီးသည်။