OOP ဘယ်လဲဘာလဲ (၁)

ကျွန်တော် programming စတင် လေ့လာကာစက စာအုပ်စာတမ်း အကိုးအကား မစုံလင်ခဲ့လို့ အခက်အခဲ အများကြီး ကြုံခဲ့ရပါတယ်။ အဲဒီလိုနဲ့ ပရိုဂရမ်းမင်း ဘာသာရပ် တွေကို လေ့လာခဲ့ရာမှာ OOP ကို ချန်လှပ်ထားခဲ့ပါတယ်။ အဲဒီ အချိန်တုန်းက ပရိုဂရမ် ရေးခဲ့တဲ့ အကြောင်းအရာတွေကလဲ ရှုပ်ထွေးမှု သိပ်မရှိခဲ့လို့ အစပိုင်းမှာတော့ အဆင်ပြေ ပါတယ်။ ဒါပေမယ့် တဖြည်းဖြည်း နဲ့ ခက်ခဲတဲ့ အပိုင်းတွေကို ရေးလာရင်း OOP ကို မသုံးမိတဲ့ အတွက် အားနည်းချက်တွေ ဖြစ်လာခဲ့ပါတယ်။ ဒါကြောင့် OOP ကို အစကနေ ပြန်ပြီး လေ့လာခဲ့ရတဲ့ ကာလတွေ ရှိခဲ့ဘူးပါတယ်။ အခု Android programming အကြောင်းကို စတင်လေ့လာပြီး ရေးသားမယ် ဆိုတဲ့ အခါမှာ အဲဒီ ဆောင်းပါးလေးတွေ မစသေးဘဲ OOP အကြောင်း အရင် စတင် မိတ်ဆက်ပေးဖို့ စိတ်ကူး ရလာပါတယ်။ ဒီဆောင်းပါးမှာ java ပရိုဂရမ်းမင်း ဘာသာစကားကို အသုံးပြုပြီး OOP ရဲ့ အယူအဆ သဘောတရား concept လေးတွေကို ကျွန်တော် အတတ်နိုင်ဆုံး နားလည်လာအောင် ကြိုးစား ရေးသားထားပါတယ်။ အကိုးအကားအနေနဲ့လဲ YCC တက္ကသိုလ်က သင်ရိုးထဲမှာ ပါတဲ့ စာအုပ်အချို့ကို ကိုးကားထားပါတယ်။ တကယ်လို့ နားလည်ရ ခက်ခဲနေမယ်ဆိုရင် ကျွန်တော့် အားနည်းချက်မျှသာ ဖြစ်ကြောင်း ကြိုတင် မေတ္တာရပ်ခံလိုပါတယ် ခင်ဗျာ။

Structured Programming နဲ့ OOP

ကျွန်တော်တို့အနေနဲ့ အချက်အလက်တွေကို ပရိုဂရမ် တစ်ခုမှာ သုံးစွဲနိုင်ဖို့အတွက် variable တွေကို build in data type (int, float, double, char,...) တွေ အသုံးပြုပြီး RAM ထဲမှာ ဖန်တီးကြပါတယ်။ ဥပမာ - int Area; float Length; char c;

အဲဒီလို variable တွေကို ပြန့်ကျဲပြီး အများကြီး မသုံးချင်တဲ့ အခါ၊ ဒါမှ မဟုတ် looping, iteation လုပ်ပြီး ထိထိရောက်ရောက် သုံးချင်တဲ့ အခါမျိုးမှာ data type တစ်ခုကို နေရာတွေ အတွဲလိုက် ဖန်တီးလို့ရတဲ့ array အနေနဲ့ ပြုလုပ်ပြီး သုံးခဲ့ကြပါတယ်။ array ဖြစ်သွားတဲ့ အခါမှာ array ရဲ့ index တွေကို looping counter နဲ့ တွဲဖက်ပြီး အစဉ်လိုက် ထုတ်သုံးလို့ ရလာပါတယ်။

ဒါပေမယ့် array ရဲ့ အားနည်းချက် တစ်ခုက မတူညီတဲ့ data type တွေကိုတော့ မစုစည်း ပေးနိုင်ပါဘူး။ ဒါကြောင့် မိမိတို့ ကိုယ်တိုင် data type စိတ်ကြိုက် စုစည်း ဖန်တီးနိုင်ဖို့ structure တွေကို သုံးလာကြပါတယ်။ ဒီနည်းနဲ့ user profile လိုမျိုး ဒေတာ အစုဝေးတွေကို variable တစ်ခုတည်းမှာ စုစည်းထားနိုင်လာပါတယ်။ ဒါကြောင့် structure တွေကို user define datatype

တွေလို့ ခေါ်ဆိုနိုင်ပါတယ်။ structure တစ်ခုကို define ပြုလုပ်ခြင်းဟာ data တွေကို ပုံစံခွက် သဘောမျိုး ပြုလုပ်ထားခြင်းမျှသာ ဖြစ်ပြီး variable တွေကို ကြေငြာသလိုမျိုး ကြေငြာပေးမှသာ RAM ထဲမှာ အဲဒီ structure က define လုပ် သတ်မှတ်ပေးထားတဲ့ အတိုင်း ဒေတာ တွေထည့်ဖို့ နေရာလေးတွေ အစုလိုက် ဖြစ်ပေါ်လာတာ ဖြစ်ပါတယ်။

နောက် build in data type (int, float, double, char,...) တွေ သုံးတုန်းက ကြုံတွေ့ရတဲ့ ပြဿနာ အချို့ရှိခဲ့ပါတယ်။ function တွေ ရေးသားတဲ့ အခါမှာ ကျွန်တော်တို့ တွက်ချက် ရယူထားတဲ့ အချက်အလက်တွေက တစ်ခုထဲ မဟုတ်ဘဲ အများအပြား ဖြစ်လာခဲ့ရင် return ဘယ်လို ပြန်ပေးရမလဲ ဆိုတဲ့ ပြဿနာပါ။ external variables (global variables) တွေကို သုံးလို့ ရပေမယ့် အဲဒါတွေ သုံးလို့ ဖြစ်လာမယ့် ပြဿနာက ပိုများနိုင်ပါတယ်။ ဒါကြောင့် ဒီလိုနေရာမျိုးမှာ structure တွေကို အသုံးပြုခြင်းအားဖြင့် ဒေတာများကို စုစည်းပြီး return ပြန်ပို့ပေးနိုင်ပါတယ်။

ကျွန်တော်တို့ အနေနဲ့ မကြာခဏ ပြုလုပ်တတ်နဲ့ လုပ်ဆောင်ချက် တွေကိုတော့ ကုဒ်တွေ စုစည်းပြီး နာမည် တစ်ခုပေးကာ function အနေနဲ့ define ပြုလုပ်ထားနိုင်ပါတယ်။ လိုအပ်တဲ့ အချိန်မှာ အကြိမ်ကြိမ် ပြန်လည်အသုံးချဖို့ အတွက် အဲဒီ နာမည်လေးကို ခေါ်ယူ (invoke or call) ပြီး အသုံးပြု နိုင်ပါတယ်။ အဲဒီလို ခေါ်ယူ အသုံးပြုတဲ့ အချိန်မှာ input parameter တွေကို ပြောင်းလဲပေးခြင်းအားဖြင့် လုပ်ဆောင်ချက်တွေကို လိုအပ်သလို အမျိုးမျိုးပြောင်းလဲ နိုင်ပြီး flexible ဖြစ်တဲ့ function တစ်ခုကို ဖန်တီးနိုင်မှာ ဖြစ်ပါတယ်။

အဲဒီလို functions တွေကို တူရာတူရာ စုစည်းထားဖို့အတွက် Modules တွေ Library တွေကို ပြုလုပ်လာကြပါတယ်။ များသောအားဖြင့် ဖိုင်တစ်ခုစီမှာ သီးခြား စုစည်း ရေးသားထားလေ့ ရှိပြီး အဲဒီ function တွေကို ခေါ်ယူ အသုံးချဖို့ လိုအပ်တဲ့ အချက်အလက်တွေကိုတော့ သက်ဆိုင်ရာ header files တွေထဲမှာ ရေးသားထားကြပါတယ်။ အဲဒီနည်းဟာ programming သမိုင်းမှာ မှတ်တိုင်တစ်ခု စိုက်ထားသလို ထင်ရှားခဲ့ပြီး ပရိုဂရမ်မာတွေက နှစ်အတော်ကြာကြာ အသုံးပြုခဲ့ကြပါတယ်။ အဲဒီလိုရေးသားနည်းကို Structured Programming လို့ ခေါ်ဆိုခဲ့ကြပါတယ်။

အဲဒီ Structured Programming မှာလဲ ပြဿနာလေးတွေ ရှိပါတယ်။ Program တစ်ပုဒ်ချင်းစီဟာ ကြီးမား ရှုပ်ထွေးလာတာနဲ့ အမျှ Structured Programming နည်းလမ်းရဲ့ အားနည်းချက်တွေ စတင်ပေါ်လွင် လာပါတယ်။ Programming အိပ်မက်ဆိုးတွေ ရှိခဲ့ဘူးပါတယ်။ ပရိုဂရမ် တစ်ပုဒ်က ကိုင်တွယ်ရ ခက်လာအောင်ကို ရှုပ်ထွေးလာပါတယ်။ သတ်မှတ်ပေးထားတဲ့ အချိန်မီ မပြီးစီးနိုင်ဘဲ ကျော်လွန်သွားရတယ်။ ပရိုဂရမ်ထဲကို feature အသစ်တွေ ခက်ခက်ခဲခဲ

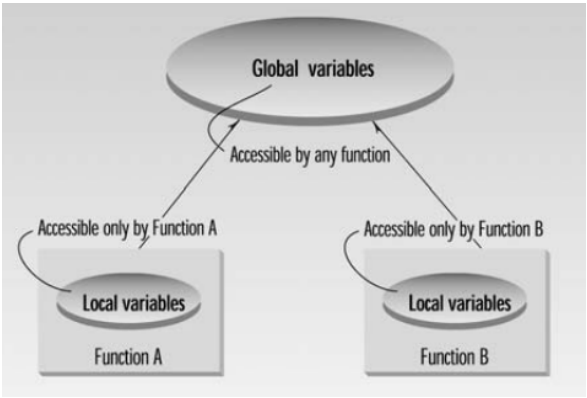
ထပ်ထည့်နေရတယ်။ ရေးသားတဲ့ ပရိုဂရမ်မာတွေ များလာတယ်။ လူတွေ ပြောင်းလဲ ရေးသားကြပြီး ကုဒ်တွေကို နားလည်ရ ပိုခက်လာပါတယ်။ ကုန်ကျစားရိတ်လဲ အလွန် မြင့်တက်လာပါတယ်။ ဒါကြောင့်လဲ အချိန်မီအပ်ဖို့ ပိုပြီး ခက်ခဲလာပါတယ်။ ဒီလို ပြဿနာတွေနဲ့ အဆုံးသတ် မလှ ဖြစ်ခဲ့ကြရပါတယ်။

အထက်ပါ အချက်တွေကို လေ့လာကြည့်ရာကနေ procedural paradigm ကိုယ်တိုင်ရဲ့ အားနည်းချက်တွေကို တွေ့ရှိလာခဲ့တယ်။ ဘယ်လောက်ပဲ structure ကျအောင် ကြိုးစား ရေးသားထားပေမယ့် ပရိုဂရမ် အကြီးစားတွေ အတွက်ကတော့ ရှုပ်ထွေးလာတတ်စမြဲ ဖြစ်ပါတယ်။

ဘာကြောင့် ဒီပြဿနာတွေ ဖြစ်ပေါ်လာရသလဲ? အချက် နှစ်ချက်ကို တွေ့ရှိလာကြပါတယ် -

၁။ function တွေက global data တွေကို အကန့်အသတ်မရှိ ရယူပြောင်းလဲ ပြင်ဆင်နိုင်တယ်။

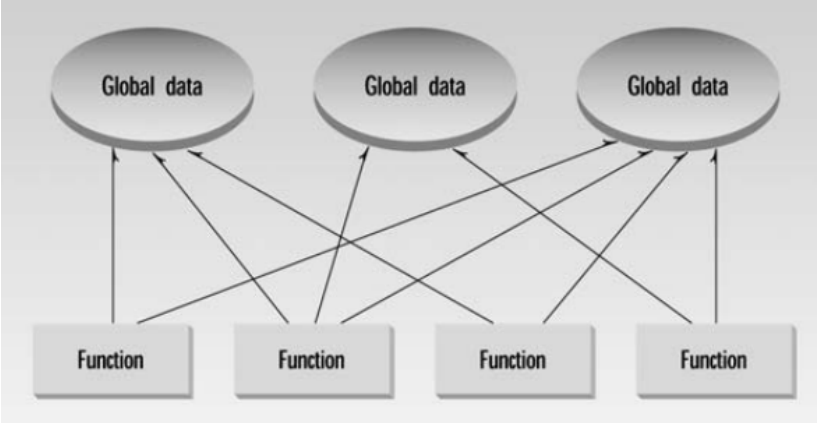
၂။ function နဲ့ သူသုံးတဲ့ data တွေက တစ်ကွဲတစ်ပြားစီ သီးသန့် ဖြစ်နေကြတယ်။



ဒီပြဿနာ နှစ်ခုကို ပစ္စည်းထားသိုတဲ့ ပရိုဂရမ် ဥပမာလေးနဲ့ လေ့လာကြည့်ရအောင်။ procedural program တွေမှာ data အမျိုးအစား နှစ်မျိုးရှိတယ်။ function ထဲမှာ hide လုပ်ပြီး ဖွက်ထားတဲ့ local variable နဲ့ function အားလုံးကနေ တိုက်ရိုက် ရယူ ပြင်ဆင်နိုင်တဲ့ global data အမျိုးအစားပဲ ဖြစ်ပါတယ်။ local data တွေ အနေနဲ့ function ထဲမှာပဲ လုံခြုံစွာ ရှိနေပြီး အခြား function တွေက ဝင်ရောက်ပြင်ဆင် ခံရခြင်းက လွတ်ကင်းနေပါတယ်။

ဒါပေမယ့် function နှစ်ခု သို့မဟုတ် နှစ်ခုထက် ပိုပြီး ဒေတာ တစ်ခုတည်းကို ရယူဖို့ လိုနေမယ် ဆိုရင်၊ နောက်တစ်ခုက အဲဒီ ဒေတာကလည်း အရေးကြီးတဲ့ ဒေတာ ဖြစ်နေလေ့ရှိတယ်ဆိုတော့ global data အနေနဲ့ ရယူ သုံးစွဲကြရပါတယ်။

ပရိုဂရမ် ကြီးလာရင် functin တွေ များလာသလို global data တွေလည်း များလာပါတယ်။ ဒါကြောင့် ၄င်းတို့ အချင်းချင်းကြား ဆက်သွယ်မှုတွေလဲ များပြားလာပါတယ်။ အဲဒီ များပြားဖောင်းပွလာတဲ့ ဆက်သွယ်မှုတွေက အထက်က ပြောခဲ့တဲ့ ပြဿနာတွေရဲ့ အစပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် ပထမ အချက်အနေနဲ့ ပရိုဂရမ်ကို ခြုံငုံ နားလည်နိုင်စွမ်း နဲပါးလာပါတယ်။ ဒုတိယ အချက်အနေနဲ့ကတော့ ပရိုဂရမ်ကို ပြန်လည် ပြင်ဆင်ရေးသားနိုင်ဖို့ အတွက်လည်း ခက်ခဲလာစေပါတယ်။



တကယ်လို့ ပြောင်းလဲရမယ့် ဒေတာတွေထဲမှာ global data တွေပါနေပြီး ၄င်းတို့ကို ပြောင်းလဲ ပစ်ချင်တယ်ဆိုရင် အဲဒီ ဒေတာတွေကို ယူသုံးထားတဲ့ function အားလုံးကိုလဲ ပြောင်းလဲပစ်ဖို့ လိုအပ်လာပါတယ်။ ပရိုဂရမ် အကြီးစား တစ်ခုမှာ ပြောင်းလဲဖို့ဆိုတာကတော့ တကယ့်ကို ခက်ခဲတဲ့ အလုပ်တစ်ခုပဲ ဖြစ်ပါတယ်။ အရာအားလုံးဟာ တစ်ခုနဲ့ တစ်ခု ဆက်စပ်နေကြပြီး တစ်ခုခုကို ပြောင်းလဲခြင်းအားဖြင့် မရည်ရွယ်ထားတဲ့ မလိုလားအပ်တဲ့ အကျိုးဆက်တွေ ဖြစ်ပေါ်လာတတ်ပြီး function တွေ ဒါမှမဟုတ် program တစ်ခုလုံး မှားယွင်းသွားနိုင်ပါတယ်။

Real-World Modeling

Procedural programming ရဲ့ ဒေတာနဲ့ function တွေကို ခွဲခြားထားတဲ့ နည်းလမ်းက ပြင်ပလောက က Real object တွေရဲ့ တကယ့်ဖြစ်စဉ်တွေနဲ့ ဝေးကွာနေပါတယ်။ လူ၊ ကား၊ အိမ် စတဲ့

ပြင်ပလောကမှာ ရှိနေတဲ့ အရာဝတ္ထုတွေ အားလုံးမှာ function နဲ့ data တွေ စုစည်းတည်ရှိနေပါတယ်။ အဲဒါတွေကို attribute နဲ့ behavior တွေလို့လဲ ခေါ်ဆိုကြပါတယ်။

Attributes

characteristics တွေလို့လဲ ခေါ်ဆိုကြပါတယ်။ ပြင်ပ သွင်ပြင် yလက္ခဏာတွေ၊ အရည်အသွေးတွေရဲ့ တန်ဖိုးတွေ၊ အမျိုးအစားတွေ စသဖြင့် ခွဲခြားဖို့ သုံးပါတယ်။ လူတွေ အတွက်ဆို နာမည်၊ အသက်၊ မွေးသက္ကရာဇ်၊ ကိုယ်အလေးချိန်၊ အရပ်အမြင့်၊ အလုပ်အကိုင် တွေဖြစ်ပါတယ်။ ကားတစ်စီး အတွက်ဆိုရင်တော့ ကားအမျိုးအစား၊ အရောင်၊ ဈေးနှုန်း၊ မြင်းကောင်ရေ၊ ထုတ်လုပ်တဲ့ ကုမ္ပဏီ စတာတွေပဲ ဖြစ်ပါတယ်။ real world အပြင်ကမ္ဘာမှာ ရှိနေတဲ့ အဲဒီ အချက်အလက်တွေကို ပရိုဂရမ်ထဲမှာတော့ attribute တွေ အနေနဲ့ ကိုယ်စားပြု ဖော်ပြပြီး ရေးသားကြရတာပဲ ဖြစ်ပါတယ်။

Behavior

behavior ဆိုတာကတော့ ပြင်ပလောကမှာ လှုံ့ဆော်မှုတွေကို တုံ့ပြန်တဲ့ ပုံစံ အပြုအမူတွေကို ပြောတာ ဖြစ်ပါတယ်။ လူတစ်ယောက်ကို ပြုံးပြရင် သူက ပြန်ပြုံးပြမယ်၊ မပြုံးရင်လဲ နေမယ်။ ဒါက သူ့ရဲ့ အပြုအမူပဲ ဖြစ်ပါတယ်။ ကားတစ်စီးရဲ့ ဘရိတ်ကို နင်းလိုက်မယ်ဆိုရင် ပုံမှန်အားဖြင့် အဲဒီကားက နှေးပြီး ရပ်သွားမှာပဲ ဖြစ်ပါတယ်။ ပြုံးပြတာ၊ မဲ့တာ၊ ရပ်သွားတာ၊ မြန်လာတာ စတာတွေဟာ behavior တွေပဲ ဖြစ်ပါတယ်။ programming မှာတော့ function တွေ (methods) အနေနဲ့ ရေးသားလေ့ ရှိပါတယ်။ အလုပ်တစ်ခုကို လုပ်ချင်တဲ့ အချိန်မှာ အဲဒီ လုပ်ငန်းစဉ်တွေ ပါဝင်တဲ့ ကုဒ်တွေကို စုစည်းရေးသားထားတဲ့ function တစ်ခုကို ခေါ်ယူ အသုံးပြုလေ့ ရှိပါတယ်။ ဥပမာ-

တကယ့် အပြင်လောက က အရာဝတ္ထုတွေမှာ attribute နဲ့ behavior တွေကို ပေါင်းစပ်ထားတဲ့ သဘောပဲ ဖြစ်ပါတယ်။ အဲဒါကို အတုယူပြီး object model တည်ဆောက်ကာ OOP approach နဲ့ "Noun" မှန်သမျှကို object အဖြစ် attributes (datas) တွေ function တွေကို ပေါင်းစပ်ခြင်းအားဖြင့် တီထွင် ဖန်တီးခဲ့တာပဲ ဖြစ်ပါတယ်။

The Automobile as an Object

Object model ကို နားလည်အောင် ကားတစ်စီးနဲ့ ဥပမာ ပေးချင်ပါတယ်။ ကျွန်တော်တို့ အနေနဲ့ လီဗာ နင်းလိုက်ရင် ပိုမြန်သွားမယ့် ကားတစ်စီး လိုချင်တယ် ဆိုကြပါစို့။ ပထမဦးဆုံး

အဲဒီကားဖြစ်လာဖို့ တစ်ယောက်ယောက်က ဒီဇိုင်းပြုလုပ်ပေးရမှာ ဖြစ်ပါတယ်။ အဲဒီ ဒီဇိုင်းဟာ စာရွက်ပေါ်မှာ ဆွဲထားတဲ့ ပုံစံဖြစ်ပြီး ၄င်းထဲမှာ လီဗာ အလုပ်လုပ်ပုံ အသေးစိတ်ကို ထည့်သွင်းရေးဆွဲထားရမှာပဲ ဖြစ်ပါတယ်။ အဲဒီ လီဗာဟာ ကားမောင်းတဲ့ သူကို ကားကို ဘယ်လိုမြန်အောင်လုပ်ထားတယ်ဆိုတဲ့ စက်အစိတ်အပိုင်း အလုပ်လုပ်ပုံ အသေးစိတ်ကို ဖုံးကွယ်ထားပြီး သူ့ကို နင်းရင် ပိုမြန်လာစေတယ် ဆိုတာကိုပဲ သိစေပါတယ်။ စတီယာရင်နဲ့ ဘရိတ်တွေရဲ့ သဘောတရားကလည်း အတူတူပဲ ဖြစ်ပါတယ်။

```
public class Automobile

{

        private String Model;

        private String LicenseNo;

        private int YearOfProduction;

        private int Price;

        private float km_per_hour;

        public void SpeedUp()

        {

                //Write code to Speed Up the car

        }

        public void Break()

        {

                //Write code to Stop the car

        }
```

};

မီးဖိုချောင်ပုံ ဆွဲထားတဲ့ blueprint (drawing) ထဲမှာ ဟင်းချက်လို့ မရသလို စက္ကူပေါ်မှာ ဆွဲထားတဲ့ ကားကို တက်မောင်းလို့ မရနိုင်ပါဘူး။ ဒီအတွက် အဲဒီ drawing တွေက ဒီဇိုင်းထုတ်ထားတဲ့ အတိုင်းပဲ တစ်ယောက်ယောက်က တည်ဆောက်ပေးဖို့ လိုပါတယ်။ တည်ဆောက်ပြီးတဲ့ ကားမှာ တကယ့် လီဗာ ပါလာပြီး ကားကို မြန်အောင် လုပ်နိုင်လာပါပြီ။ ဒါပေမယ့် လီဗာက သူ့အလိုလိုတော့ ကားကို ပိုမြန်လာအောင် လုပ်မပေးနိုင်ပါဘူး။ တစ်ယောက်ယောက်က နင်းပေးဖို့ လိုပါတယ်။

OOP model မှာလဲ အဲဒီလို ကိုယ်လိုချင်တဲ့ object ကို ဒီဇိုင်းပြုလုပ်ဖို့ class တစ်ခု တည်ဆောက်ပေးရပါတယ်။ ဒါပေမယ့် ဒါဟာ object မဟုတ်သေးပါဘူး။ object ဖြစ်လာဖို့ အတွက် instantiate လုပ်ပေးရပါတယ်။ အဲဒီလို လုပ်ပေးလို့ ဖြစ်လာတဲ့ object တွေကို instant တွေလို့လဲ ခေါ်ပါတယ်။ ကား ဒီဇိုင်း drawing တစ်ခုကနေ ကားအများကြီး ထုတ်လုပ်လို့ ရသလိုပဲ class တစ်ခုကနေ object အမြောက်အများ ဖန်တီးနိုင်ပါတယ်။ အဲဒီ object တစ်ခုချင်းစီမှာ ကိုယ်ပိုင် ဒေတာတွေနဲ့ function တွေ အသီးသီး ပါဝင်ကြပါတယ်။

ဆက်ရန်...

အကိုးအကားများ

၁။ Java : how to program / P.J.Deitel, H.M. Deitel. --9th ed. 2012

၂။ Object-Oriented Programmming in C++ / Robert Lafore. --4th ed.2002

ဒေါက်တာ အောင်ဝင်းထွဋ်

(bluephoenix)

http://tech4mm.com