**OOP ဘယ္လဲဘာလဲ (၁)**

ကၽြန္ေတာ္ programming စတင္ ေလ႔လာကာစက စာအုပ္စာတမ္း အကိုးအကား မစံုလင္ခဲ႔လို႔ အခက္အခဲ အမ်ားႀကီး ၾကံဳခဲ႔ရပါတယ္။ အဲဒီလိုနဲ႔ ပရိုဂရမ္းမင္း ဘာသာရပ္ ေတြကို ေလ႔လာခဲ႔ရာမွာ OOP ကို ခ်န္လွပ္ထားခဲ႔ပါတယ္။ အဲဒီ အခ်ိန္တုန္းက ပရိုဂရမ္ ေရးခဲ႔တဲ႔ အေၾကာင္းအရာေတြကလဲ ရႈပ္ေထြးမႈ သိပ္မရွိခဲ႔လို႔ အစပိုင္းမွာေတာ႔ အဆင္ေျပ ပါတယ္။ ဒါေပမယ္႔ တျဖည္းျဖည္း နဲ႔ ခက္ခဲတဲ႔ အပိုင္းေတြကို ေရးလာရင္း OOP ကို မသံုးမိတဲ႔ အတြက္ အားနည္းခ်က္ေတြ ျဖစ္လာခဲ႔ပါတယ္။ ဒါေၾကာင္႔ OOP ကို အစကေန ျပန္ၿပီး ေလ႔လာခဲ႔ရတဲ႔ ကာလေတြ ရွိခဲ႔ဘူးပါတယ္။ အခု Android programming အေၾကာင္းကို စတင္ေလ႔လာၿပီး ေရးသားမယ္ ဆိုတဲ႔ အခါမွာ အဲဒီ ေဆာင္းပါးေလးေတြ မစေသးဘဲ OOP အေၾကာင္း အရင္ စတင္ မိတ္ဆက္ေပးဖို႔ စိတ္ကူး ရလာပါတယ္။ ဒီေဆာင္းပါးမွာ java ပရိုဂရမ္းမင္း ဘာသာစကားကို အသံုးျပဳၿပီး OOP ရဲ႕ အယူအဆ သေဘာတရား concept ေလးေတြကို ကၽြန္ေတာ္ အတတ္ႏိုင္ဆံုး နားလည္လာေအာင္ ႀကိဳးစား ေရးသားထားပါတယ္။ အကိုးအကားအေနနဲ႔လဲ YCC တကၠသိုလ္က သင္ရိုးထဲမွာ ပါတဲ႔ စာအုပ္အခ်ိဳ႕ကို ကိုးကားထားပါတယ္။ တကယ္လို႔ နားလည္ရ ခက္ခဲေနမယ္ဆိုရင္ ကၽြန္ေတာ္႔ အားနည္းခ်က္မွ်သာ ျဖစ္ေၾကာင္း ႀကိဳတင္ ေမတၱာရပ္ခံလိုပါတယ္ ခင္ဗ်ာ။

**Structured Programming နဲ႔ OOP**

ကၽြန္ေတာ္တို႔အေနနဲ႔ အခ်က္အလက္ေတြကို ပရိုဂရမ္ တစ္ခုမွာ သံုးစြဲႏိုင္ဖို႔အတြက္ variable ေတြကို build in data type (int, float, double, char,...) ေတြ အသံုးျပဳၿပီး RAM ထဲမွာ ဖန္တီးၾကပါတယ္။ ဥပမာ - int Area; float Length; char c;

အဲဒီလို variable ေတြကို ျပန္႔က်ဲၿပီး အမ်ားႀကီး မသံုးခ်င္တဲ႔ အခါ၊ ဒါမွ မဟုတ္ looping, iteation လုပ္ၿပီး ထိထိေရာက္ေရာက္ သံုးခ်င္တဲ႔ အခါမ်ိဳးမွာ data type တစ္ခုကို ေနရာေတြ အတြဲလိုက္ ဖန္တီးလို႔ရတဲ႔ array အေနနဲ႔ ျပဳလုပ္ၿပီး သံုးခဲ႔ၾကပါတယ္။ array ျဖစ္သြားတဲ႔ အခါမွာ array ရဲ႕ index ေတြကို looping counter နဲ႔ တြဲဖက္ၿပီး အစဥ္လိုက္ ထုတ္သံုးလို႔ ရလာပါတယ္။

ဒါေပမယ္႔ array ရဲ႕ အားနည္းခ်က္ တစ္ခုက မတူညီတဲ႔ data type ေတြကိုေတာ႔ မစုစည္း ေပးႏိုင္ပါဘူး။ ဒါေၾကာင္႔ မိမိတို႔ ကိုယ္တိုင္ data type စိတ္ႀကိဳက္ စုစည္း ဖန္တီးႏိုင္ဖို႔ structure ေတြကို သံုးလာၾကပါတယ္။ ဒီနည္းနဲ႔ user profile လိုမ်ိဳး ေဒတာ အစုေဝးေတြကို variable တစ္ခုတည္းမွာ စုစည္းထားႏိုင္လာပါတယ္။ ဒါေၾကာင္႔ structure ေတြကို user define datatype ေတြလို႔ ေခၚဆိုႏိုင္ပါတယ္။ structure တစ္ခုကို define ျပဳလုပ္ျခင္းဟာ data ေတြကို ပံုစံခြက္ သေဘာမ်ိဳး ျပဳလုပ္ထားျခင္းမွ်သာ ျဖစ္ၿပီး variable ေတြကို ေၾကျငာသလိုမ်ိဳး ေၾကျငာေပးမွသာ RAM ထဲမွာ အဲဒီ structure က define လုပ္ သတ္မွတ္ေပးထားတဲ႔ အတိုင္း ေဒတာ ေတြထည္႔ဖို႔ ေနရာေလးေတြ အစုလိုက္ ျဖစ္ေပၚလာတာ ျဖစ္ပါတယ္။

ေနာက္ build in data type (int, float, double, char,...) ေတြ သံုးတုန္းက ႀကံဳေတြရတဲ႔ ျပႆနာ အခ်ိဳ႕ရွိခဲ႔ပါတယ္။ function ေတြ ေရးသားတဲ႔ အခါမွာ ကၽြန္ေတာ္တို႔ တြက္ခ်က္ ရယူထားတဲ႔ အခ်က္အလက္ေတြက တစ္ခုထဲ မဟုတ္ဘဲ အမ်ားအျပား ျဖစ္လာခဲ႔ရင္ return ဘယ္လို ျပန္ေပးရမလဲ ဆိုတဲ႔ ျပႆနာပါ။ external variables (global variables) ေတြကို သံုးလို႔ ရေပမယ္႔ အဲဒါေတြ သံုးလို႔ ျဖစ္လာမယ္႔ ျပႆနာက ပိုမ်ားႏိုင္ပါတယ္။ ဒါေၾကာင္႔ ဒီလိုေနရာမ်ိဳးမွာ structure ေတြကို အသံုးျပဳျခင္းအားျဖင္႔ ေဒတာမ်ားကို စုစည္းၿပီး return ျပန္ပို႔ေပးႏိုင္ပါတယ္။

ကၽြန္ေတာ္တို႔ အေနနဲ႔ မၾကာခဏ ျပဳလုပ္တတ္နဲ႔ လုပ္ေဆာင္ခ်က္ ေတြကိုေတာ႔ ကုဒ္ေတြ စုစည္းၿပီး နာမည္ တစ္ခုေပးကာ function အေနနဲ႔ define ျပဳလုပ္ထားႏိုင္ပါတယ္။ လိုအပ္တဲ႔ အခ်ိန္မွာ အႀကိမ္ႀကိမ္ ျပန္လည္အသံုးခ်ဖို႔ အတြက္ အဲဒီ နာမည္ေလးကို ေခၚယူ (invoke or call) ၿပီး အသံုးျပဳ ႏိုင္ပါတယ္။ အဲဒီလို ေခၚယူ အသံုးျပဳတဲ႔ အခ်ိန္မွာ input parameter ေတြကို ေျပာင္းလဲေပးျခင္းအားျဖင္႔ လုပ္ေဆာင္ခ်က္ေတြကို လိုအပ္သလို အမ်ိဳးမ်ိဳးေျပာင္းလဲ ႏိုင္ၿပီး flexible ျဖစ္တဲ႔ function တစ္ခုကို ဖန္တီးႏိုင္မွာ ျဖစ္ပါတယ္။

အဲဒီလို functions ေတြကို တူရာတူရာ စုစည္းထားဖို႔အတြက္ Modules ေတြ Library ေတြကို ျပဳလုပ္လာၾကပါတယ္။ မ်ားေသာအားျဖင္႔ ဖိုင္တစ္ခုစီမွာ သီးျခား စုစည္း ေရးသားထားေလ႔ ရွိၿပီး အဲဒီ function ေတြကို ေခၚယူ အသံုးခ်ဖို႔ လိုအပ္တဲ႔ အခ်က္အလက္ေတြကိုေတာ႔ သက္ဆိုင္ရာ header files ေတြထဲမွာ ေရးသားထားၾကပါတယ္။ အဲဒီနည္းဟာ programming သမိုင္းမွာ မွတ္တိုင္တစ္ခု စိုက္ထားသလို ထင္ရွားခဲ႔ၿပီး ပရိုဂရမ္မာေတြက ႏွစ္အေတာ္ၾကာၾကာ အသံုးျပဳခဲ႔ၾကပါတယ္။ အဲဒီလိုေရးသားနည္းကို Structured Programming လို႔ ေခၚဆိုခဲ႔ၾကပါတယ္။

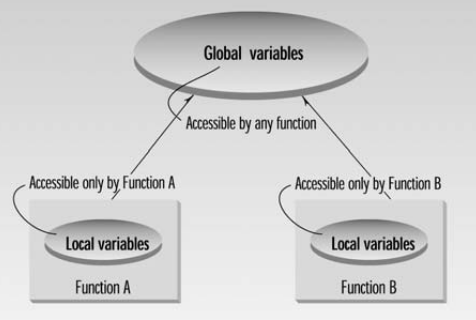
အဲဒီ Structured Programming မွာလဲ ျပႆနာေလးေတြ ရွိပါတယ္။ Program တစ္ပုဒ္ခ်င္းစီဟာ ႀကီးမား ရႈပ္ေထြးလာတာနဲ႔ အမွ် Structured Programming နည္းလမ္းရဲ႕ အားနည္းခ်က္ေတြ စတင္ေပၚလြင္ လာပါတယ္။ Programming အိပ္မက္ဆိုးေတြ ရွိခဲ႔ဘူးပါတယ္။ ပရိုဂရမ္ တစ္ပုဒ္က ကိုင္တြယ္ရ ခက္လာေအာင္ကို ရႈပ္ေထြးလာပါတယ္။ သတ္မွတ္ေပးထားတဲ႔ အခ်ိန္မီ မၿပီးစီးႏိုင္ဘဲ ေက်ာ္လြန္သြားရတယ္။ ပရိုဂရမ္ထဲကို feature အသစ္ေတြ ခက္ခက္ခဲခဲ ထပ္ထည္႔ေနရတယ္။ ေရးသားတဲ႔ ပရိုဂရမ္မာေတြ မ်ားလာတယ္။ လူေတြ ေျပာင္းလဲ ေရးသားၾကၿပီး ကုဒ္ေတြကို နားလည္ရ ပိုခက္လာပါတယ္။ ကုန္က်စားရိတ္လဲ အလြန္ ျမင္႔တက္လာပါတယ္။ ဒါေၾကာင္႔လဲ အခ်ိန္မီအပ္ဖို႔ ပိုၿပီး ခက္ခဲလာပါတယ္။ ဒီလို ျပႆနာေတြနဲ႔ အဆံုးသတ္ မလွ ျဖစ္ခဲ႔ၾကရပါတယ္။

အထက္ပါ အခ်က္ေတြကို ေလ႔လာၾကည္႔ရာကေန procedural paradigm ကိုယ္တိုင္ရဲ႕ အားနည္းခ်က္ေတြကို ေတြ႔ရွိလာခဲ႔တယ္။ ဘယ္ေလာက္ပဲ structure က်ေအာင္ ႀကိဳးစား ေရးသားထားေပမယ္႔ ပရိုဂရမ္ အႀကီးစားေတြ အတြက္ကေတာ႔ ရႈပ္ေထြးလာတတ္စျမဲ ျဖစ္ပါတယ္။

ဘာေၾကာင္႔ ဒီျပႆနာေတြ ျဖစ္ေပၚလာရသလဲ? အခ်က္ ႏွစ္ခ်က္ကို ေတြ႔ရွိလာၾကပါတယ္ -

၁။ function ေတြက global data ေတြကို အကန္႔အသတ္မရွိ ရယူေျပာင္းလဲ ျပင္ဆင္ႏိုင္တယ္။

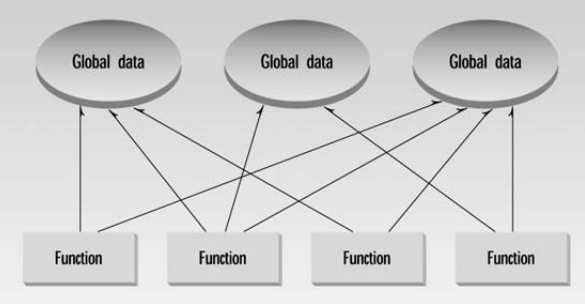
၂။ function နဲ႔ သူသံုးတဲ႔ data ေတြက တစ္ကြဲတစ္ျပားစီ သီးသန္႔ ျဖစ္ေနၾကတယ္။



ဒီျပႆနာ ႏွစ္ခုကို ပစၥည္းထားသိုတဲ႔ ပရိုဂရမ္ ဥပမာေလးနဲ႔ ေလ႔လာၾကည္႔ရေအာင္။ procedural program ေတြမွာ data အမ်ိဳးအစား ႏွစ္မ်ိဳးရွိတယ္။ function ထဲမွာ hide လုပ္ၿပီး ဖြက္ထားတဲ႔ local variable နဲ႔ function အားလံုးကေန တိုက္ရိုက္ ရယူ ျပင္ဆင္ႏိုင္တဲ႔ global data အမ်ိဳးအစားပဲ ျဖစ္ပါတယ္။ local data ေတြ အေနနဲ႔ function ထဲမွာပဲ လံုျခံဳစြာ ရွိေနၿပီး အျခား function ေတြက ဝင္ေရာက္ျပင္ဆင္ ခံရျခင္းက လြတ္ကင္းေနပါတယ္။

ဒါေပမယ္႔ function ႏွစ္ခု သို႔မဟုတ္ ႏွစ္ခုထက္ ပိုၿပီး ေဒတာ တစ္ခုတည္းကို ရယူဖို႔ လိုေနမယ္ ဆိုရင္၊ ေနာက္တစ္ခုက အဲဒီ ေဒတာကလည္း အေရးႀကီးတဲ႔ ေဒတာ ျဖစ္ေနေလ႔ရွိတယ္ဆိုေတာ႔ global data အေနနဲ႔ ရယူ သံုးစြဲၾကရပါတယ္။

ပရိုဂရမ္ ႀကီးလာရင္ functin ေတြ မ်ားလာသလို global data ေတြလည္း မ်ားလာပါတယ္။ ဒါေၾကာင္႔ ၄င္းတို႔ အခ်င္းခ်င္းၾကား ဆက္သြယ္မႈေတြလဲ မ်ားျပားလာပါတယ္။ အဲဒီ မ်ားျပားေဖာင္းပြလာတဲ႔ ဆက္သြယ္မႈေတြက အထက္က ေျပာခဲ႔တဲ႔ ျပႆနာေတြရဲ႕ အစပဲ ျဖစ္ပါတယ္။ ဒါေၾကာင္႔ ပထမ အခ်က္အေနနဲ႔ ပရိုဂရမ္ကို ျခံဳငံု နားလည္ႏိုင္စြမ္း နဲပါးလာပါတယ္။ ဒုတိယ အခ်က္အေနနဲ႔ကေတာ႔ ပရိုဂရမ္ကို ျပန္လည္ ျပင္ဆင္ေရးသားႏိုင္ဖို႔ အတြက္လည္း ခက္ခဲလာေစပါတယ္။



တကယ္လို႔ ေျပာင္းလဲရမယ္႔ ေဒတာေတြထဲမွာ global data ေတြပါေနၿပီး ၄င္းတို႔ကို ေျပာင္းလဲ ပစ္ခ်င္တယ္ဆိုရင္ အဲဒီ ေဒတာေတြကို ယူသံုးထားတဲ႔ function အားလံုးကိုလဲ ေျပာင္းလဲပစ္ဖို႔ လိုအပ္လာပါတယ္။ ပရိုဂရမ္ အႀကီးစား တစ္ခုမွာ ေျပာင္းလဲဖို႔ဆိုတာကေတာ႔ တကယ္႔ကို ခက္ခဲတဲ႔ အလုပ္တစ္ခုပဲ ျဖစ္ပါတယ္။ အရာအားလံုးဟာ တစ္ခုနဲ႔ တစ္ခု ဆက္စပ္ေနၾကၿပီး တစ္ခုခုကို ေျပာင္းလဲျခင္းအားျဖင္႔ မရည္ရြယ္ထားတဲ႔ မလိုလားအပ္တဲ႔ အက်ိဳးဆက္ေတြ ျဖစ္ေပၚလာတတ္ၿပီး function ေတြ ဒါမွမဟုတ္ program တစ္ခုလံုး မွားယြင္းသြားႏိုင္ပါတယ္။

**Real-World Modeling**

Procedural programming ရဲ႕ ေဒတာနဲ႔ function ေတြကို ခြဲျခားထားတဲ႔ နည္းလမ္းက ျပင္ပေလာက က Real object ေတြရဲ႕ တကယ္႔ျဖစ္စဥ္ေတြနဲ႔ ေဝးကြာေနပါတယ္။ လူ၊ ကား၊ အိမ္ စတဲ႔ ျပင္ပေလာကမွာ ရွိေနတဲ႔ အရာဝတၳဳေတြ အားလံုးမွာ function နဲ႔ data ေတြ စုစည္း တည္ရွိေနပါတယ္။ အဲဒါေတြကို attribute နဲ႔ behavior ေတြလို႔လဲ ေခၚဆိုၾကပါတယ္။

**Attributes**

characteristics ေတြလို႔လဲ ေခၚဆိုၾကပါတယ္။ ျပင္ပ သြင္ျပင္ yလကၡဏာေတြ၊ အရည္အေသြးေတြရဲ႕ တန္ဖိုးေတြ၊ အမ်ိဳးအစားေတြ စသျဖင္႔ ခြဲျခားဖို႔ သံုးပါတယ္။ လူေတြ အတြက္ဆို နာမည္၊ အသက္၊ ေမြးသကၠရာဇ္၊ ကိုယ္အေလးခ်ိန္၊ အရပ္အျမင္႔၊ အလုပ္အကိုင္ ေတြျဖစ္ပါတယ္။ ကားတစ္စီး အတြက္ဆိုရင္ေတာ႔ ကားအမ်ိဳးအစား၊ အေရာင္၊ ေစ်းႏႈန္း၊ ျမင္းေကာင္ေရ၊ ထုတ္လုပ္တဲ႔ ကုမၸဏီ စတာေတြပဲ ျဖစ္ပါတယ္။ real world အျပင္ကမာၻမွာ ရွိေနတဲ႔ အဲဒီ အခ်က္အလက္ေတြကို ပရိုဂရမ္ထဲမွာေတာ႔ attribute ေတြ အေနနဲ႔ ကိုယ္စားျပဳ ေဖာ္ျပၿပီး ေရးသားၾကရတာပဲ ျဖစ္ပါတယ္။

**Behavior**

behavior ဆိုတာကေတာ႔ ျပင္ပေလာကမွာ လႈံ႕ေဆာ္မႈေတြကို တံု႔ျပန္တဲ႔ ပံုစံ အျပဳအမူေတြကို ေျပာတာ ျဖစ္ပါတယ္။ လူတစ္ေယာက္ကို ျပံဳးျပရင္ သူက ျပန္ျပံဳးျပမယ္၊ မျပံဳးရင္လဲ ေနမယ္။ ဒါက သူ႔ရဲ႕ အျပဳအမူပဲ ျဖစ္ပါတယ္။ ကားတစ္စီးရဲ႕ ဘရိတ္ကို နင္းလိုက္မယ္ဆိုရင္ ပံုမွန္အားျဖင္႔ အဲဒီကားက ေႏွးၿပီး ရပ္သြားမွာပဲ ျဖစ္ပါတယ္။ ျပံဳးျပတာ၊ မဲ႔တာ၊ ရပ္သြားတာ၊ ျမန္လာတာ စတာေတြဟာ behavior ေတြပဲ ျဖစ္ပါတယ္။ programming မွာေတာ႔ function ေတြ (methods) အေနနဲ႔ ေရးသားေလ႔ ရွိပါတယ္။ အလုပ္တစ္ခုကို လုပ္ခ်င္တဲ႔ အခ်ိန္မွာ အဲဒီ လုပ္ငန္းစဥ္ေတြ ပါဝင္တဲ႔ ကုဒ္ေတြကို စုစည္းေရးသားထားတဲ႔ function တစ္ခုကို ေခၚယူ အသံုးျပဳေလ႔ ရွိပါတယ္။ ဥပမာ-

တကယ္႔ အျပင္ေလာက က အရာဝတၳဳေတြမွာ attribute နဲ႔ behavior ေတြကို ေပါင္းစပ္ထားတဲ႔ သေဘာပဲ ျဖစ္ပါတယ္။ အဲဒါကို အတုယူၿပီး object model တည္ေဆာက္ကာ OOP approach နဲ႔ု "Noun" မွန္သမွ်ကို object အျဖစ္ attributes (datas) ေတြ function ေတြကို ေပါင္းစပ္ျခင္းအားျဖင္႔ တီထြင္ ဖန္တီးခဲ႔တာပဲ ျဖစ္ပါတယ္။

**The Automobile as an Object**

Object model ကို နားလည္ေအာင္ ကားတစ္စီးနဲ႔ ဥပမာ ေပးခ်င္ပါတယ္။ ကၽြန္ေတာ္တို႔ အေနနဲ႔ လီဗာ နင္းလိုက္ရင္ ပိုျမန္သြားမယ္႔ ကားတစ္စီး လိုခ်င္တယ္ ဆိုၾကပါစို႔။ ပထမဦးဆံုး အဲဒီကားျဖစ္လာဖို႔ တစ္ေယာက္ေယာက္က ဒီဇိုင္းျပဳလုပ္ေပးရမွာ ျဖစ္ပါတယ္။ အဲဒီ ဒီဇိုင္းဟာ စာရြက္ေပၚမွာ ဆြဲထားတဲ႔ ပံုစံျဖစ္ၿပီး ၄င္းထဲမွာ လီဗာ အလုပ္လုပ္ပံု အေသးစိတ္ကို ထည္႔သြင္း ေရးဆြဲထားရမွာပဲ ျဖစ္ပါတယ္။ အဲဒီ လီဗာဟာ ကားေမာင္းတဲ႔ သူကို ကးကို ဘယ္လိုျမန္ေအာင္ လုပ္ထားတယ္ဆိုတဲ႔ စက္အစိတ္အပိုင္း အလုပ္လုပ္ပံု အေသးစိတ္ကို ဖံုးကြယ္ထားၿပီး သူ႔ကို နင္းရင္ ပိုျမန္လာေစတယ္ ဆိုတာကိုပဲ သိေစပါတယ္။ စတီယာရင္နဲ႔ ဘရိတ္ေတြရဲ႕ သေဘာတရားကလည္း အတူတူပဲ ျဖစ္ပါတယ္။

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

}

};

မီးဖိုေခ်ာင္ပံု ဆြဲထားတဲ႔ blueprint (drawing) ထဲမွာ ဟင္းခ်က္လို႔ မရသလို စကၠဴေပၚမွာ ဆြဲထားတဲ႔ ကားကို တက္ေမာင္းလို႔ မရႏိုင္ပါဘူး။ ဒီအတြက္ အဲဒီ drawing ေတြက ဒီဇိုင္းထုတ္ထားတဲ႔ အတိုင္းပဲ တစ္ေယာက္ေယာက္က တည္ေဆာက္ေပးဖို႔ လိုပါတယ္။ တည္ေဆာက္ၿပီးတဲ႔ ကားမွာ တကယ္႔ လီဗာ ပါလာၿပီး ကားကို ျမန္ေအာင္ လုပ္ႏိုင္လာပါၿပီ။ ဒါေပမယ္႔ လီဗာက သူ႔အလိုလိုေတာ႔ ကားကို ပိုျမန္လာေအာင္ လုပ္မေပးႏိုင္ပါဘူး။ တစ္ေယာက္ေယာက္က နင္းေပးဖို႔ လိုပါတယ္။

OOP model မွာလဲ အဲဒီလို ကိုယ္လိုခ်င္တဲ႔ object ကို ဒီဇိုင္းျပဳလုပ္ဖို႔ class တစ္ခု တည္ေဆာက္ေပးရပါတယ္။ ဒါေပမယ္႔ ဒါဟာ object မဟုတ္ေသးပါဘူး။ object ျဖစ္လာဖို႔ အတြက္ instantiate လုပ္ေပးရပါတယ္။ အဲဒီလို လုပ္ေပးလို႔ ျဖစ္လာတဲ႔ object ေတြကို instant ေတြလို႔လဲ ေခၚပါတယ္။ ကား ဒီဇိုင္း drawing တစ္ခုကေန ကားအမ်ားႀကီး ထုတ္လုပ္လို႔ ရသလိုပဲ class တစ္ခုကေန object အေျမာက္အမ်ား ဖန္တီးႏိုင္ပါတယ္။ အဲဒီ object တစ္ခုခ်င္းစီမွာ ကိုယ္ပိုင္ ေဒတာေတြနဲ႔ function ေတြ အသီးသီး ပါဝင္ၾကပါတယ္။

ဆက္ရန္...

**အကိုးအကားမ်ား**

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

ေဒါက္တာ ေအာင္ဝင္းထြဋ္

(bluephoenix)

http://tech4mm.com