အဆင့်မြင့်နည်းပညာတို့၏လမ်းစ

# လွယ်ကူသော JAVA သင်ခန်းစာများ

ဆန္ဒအောင်

**ရှင်မတောင်စာပေ**

အမှတ် (၂၉၈) ၁/အနော်ရထာ (၅)လမ်း

သာကေတ၊ ရန်ကုန်မြို့။

# လွယ်ကူသော

# JAVA

# သင်ခန်းစာများ

ဆန္ဒအောင်

# မာတိကာ


| အခန်း | အကြောင်းအရာ | စာမျက်နှာ |
|---|---|---|
| ၁ | ပရိုဂရမ်နည်းလမ်း | ၉ |
| ၂ | Variable များနှင့် Type များ | ၁၇ |
| ၃ | Method များ | ၂၇ |
| ၄ | အခြေအနေများနှင့် ထပ်တလဲလဲဖြစ်စဉ်များ | ၃၉ |
| ၅ | အသီးအပွင့်ဝေဆာသော Method များ | ၄၉ |
| ၆ | အကြိမ်ကြိမ်လည်ပတ်ခြင်း | ၆၅ |
| ၇ | စာသားများနှင့် အရာဝတ္ထုများ | ၇၉ |
| ၈ | စိတ်ဝင်စားဖွယ်ရာကောင်းသော ဝတ္ထုများ | ၈၉ |
| ၉ | ကိုယ်ပိုင်ဝတ္ထုများ တည်ဆောက်ခြင်း | ၉၉ |
| ၁၀ | Array များ | ၁၁၅ |
| ၁၁ | ဝတ္ထုများ၏ Array များ | ၁၂၇ |
| ၁၂ | Array များ၏ ဝတ္ထုများ | ၁၄၃ |
| ၁၃ | ဝတ္ထုဦးတည်သော ပရိုဂရမ်းမင်း | ၁၅၃ |

# စာရေးသူ၏အမှာ

ကမ္ဘာပေါ်မှာရှိတဲ့ ကွန်ပျူတာ ပရိုဆက်ဆာအားလုံးထက်ပိုပြီး အဆင့်မြင့်တဲ့ သတင်းအချက်အလက်ချေဖျက်စနစ်ကြီးဟာ ကျွန်တော်တို့အားလုံးရဲ့ ဦးခေါင်းခွံထဲမှာရှိပါတယ်။ လက်ရှိသိပ္ပံပညာဟာ သူ့ကို နားလည်စ အခြေအနေမှာပဲ ရှိနေသေးပြီး ကမ္ဘာပေါ်က ဘယ်ကွန်ပျူတာကမှ မစွမ်းဆောင်နိုင်တဲ့ ကိစ္စတွေကို သူက လုပ်ဆောင်နိုင်ပါတယ်။ ဥပမာ လူသားခံစားချက်တွေကို ကွန်ပျူတာစနစ်တစ်ခုထဲ လုံ့ဆော်ထည့်သွင်းလို့ မရနိုင်သလို ခံစားမှုရဲ့အခြေခံဖြစ်တဲ့ အသိစိတ်ဟာလည်း ကွန်ပျူတာတွေမှာ မရှိပါဘူး။

အီလက်ထရောနစ်အမြင်နဲ့ကြည့်ရင် လူ့ဦးနှောက်မှာ ဩချလောက်စရာ ဖြစ်စဉ်တစ်ခုရှိပါတယ်။ ကွန်ပျူတာ ချစ်ပ်ပြားတွေဟာ လျှပ်စစ်ဖြတ်စီးနိုင်တဲ့ ကြားခံနယ်တစ်ခုခုကို လျှပ်စစ်ဖြတ်မစီးနိုင်တဲ့ ပလတ်စတစ်တစ်ခုထဲမှာ လျှပ်စီးကြောင်းတွေအနေနဲ့ ပုံလောင်းထားတာ ဖြစ်ပါတယ်။ ဒီတော့ ချစ်ပ်ပြားတစ်ခုကို စထုတ်လုပ်လိုက်တဲ့အချိန်ကစပြီး သက်တမ်းကုန်မသွားခင်အထိ ဒီလျှပ်စီးကြောင်းတွေဟာ ပုံသေရှိနေပါတယ်။

သတင်းအချက်အလက် ကိုင်တွယ်ရာမှာ အသုံးပြုရတဲ့ ဒီလျှပ်စီးကြောင်းတွေဟာ လူသားဦးနှောက်ထဲမှာတော့ ကွဲပြားတဲ့အသွင်နဲ့ တည်ရှိနေပါတယ်။ ဦးနှောက်ဟာ မွေးဖွားချိန်ကစပြီး လူကြီးဖြစ်တဲ့ အချိန်အထိ ကြီးထွားနေတဲ့ ပရိုဆက်ဆာကြီးဖြစ်ပါတယ်။ ကြီးထွားနေတာအပြင် လှုပ်ရှားနိုင်တဲ့ ဦးနှောက်ဆဲလ်တွေကိုလည်း လျှပ်စီးဖြတ်စီးနိုင်တဲ့ လမ်းကြောင်းတွေနဲ့ ဆက်သွယ်ထားပါတယ်။ ဒီလမ်းကြောင်းတွေရဲ့ ထူးခြားချက်ကတော့ သူကိုင်တွယ်ဆောင်ရွက်နေတဲ့ သတင်းအချက်အလက်ရဲ့ လှုံ့ဆော်မှုအရ သူ့ရဲ့ လျှပ်စီးပတ်လမ်းကြောင်းတွေကို ပြုပြင်နိုင်ပါတယ်။ ပတ်လမ်းအသစ်တွေ သူ့ဘာသာသူ ပြန်လည်ဆက်သွယ်နိုင်ပါတယ်။

ပညာသင်တယ်ဆိုတာ ဒါကို ပြောတာပါပဲ။ အသိပညာအသစ်တစ်ခုကို ဦးနှောက်ထဲထည့်လိုက်ရင် ဦးနှောက်ဟာ ဒီအသိပညာကို ချက်ချင်းလက်ငင်း အပြည့်အဝ လက်ခံနိုင်စွမ်း မရှိပါဘူး။ သူ့ရဲ့ လျှပ်စီးပတ်လမ်းတွေကို ပြန်ပြင်ရပါတယ်။ သူ့ကိုယ်သူ ဒီအချက်အလက်ကို ကိုင်တွယ်ရာမှာ စွမ်းအားအမြင့်ဆုံးဖြစ်အောင် ပြင်ဆင်ရပါတယ်။

ဆိုလိုတာက ပညာသင်တယ်ဆိုတာ အချိန်ယူရပါတယ်။ သိသင့်သိထိုက်တဲ့ အကြောင်းအရာအားလုံးကို သိပြီးသွားပြီဆိုရင်တောင်မှ ဦးနှောက်ဟာ ဒီအချက်အလက်တွေကို ဖြစ်ဖြစ်မြောက်မြောက်ကိုင်တွယ်နိုင်ဖို့ အချိန်ယူရပါတယ်။

ဒီကိစ္စကို ပရိုဂရမ်မင်း ပညာရေးနဲ့ ရှင်းပြလိုပါတယ်။ ပညာရပ်တစ်ခုကို စပြီးသင်ယူရင် သူနဲ့ပတ်သက်ပြီး သိရှိရမယ့်အချက်တွေ ရှိပါတယ်။ သူတို့ကို မှတ်ဉာဏ်သုံးပြီး မှတ်ထားရလေ့ရှိပါတယ်။ ဒီလိုမလုပ်ဘဲ ဘာမှမတတ်သေးခင် ခြုံငုံနားလည်အောင်သွားလုပ်ရင် သင်ကြားမှုဟာ မလိုအပ်ဘဲ ရှုပ်ထွေးသွားနိုင်ပါတယ်။ ခြုံငုံနားလည်တဲ့နည်းလမ်းကို သုံးလို့မရဘူးလို့ ပြောချင်တာတော့ မဟုတ်ပါဘူး။ ဒီစာအုပ်မှာလည်း လက်တွေ့ဘဝက ဥပမာတွေနဲ့နှိုင်းယှဉ်ပြီး ပရိုဂရမ်းမင်းအယူအဆတွေကို ခြုံငုံပြီး ရှင်းလင်းထားတာ တွေ့ရပါမယ်။

အခြေခံတွေကို နားလည်တတ်ကျွမ်းသွားရင် တွေးတောတဲ့အလုပ်ကို မဖြစ်မနေ စလုပ်ရပါတယ်။ သတင်းအချက်အလက်အသစ်တွေကို ဦးနှောက်က စွမ်းအားပြည့်ကိုင်တွယ်နိုင်အောင် လမ်းကြောင်းအသစ်တွေ ဖောက်ခိုင်းရတယ် မဟုတ်လား။ ကိုယ်ကိုယ်တိုင်က ဒီအကြောင်းအရာတွေကို တက်တက်ကြွကြွ စိတ်ဝင်တစား မလေ့လာရင် သူကလည်း ဘယ်လိုလုပ်ပြီး စိတ်ပါလက်ပါ လမ်းကြောင်းသစ်တွေ ဖောက်တော့မှာလဲ။

နောက်ထပ်သတိထားရမယ့် ကိစ္စတစ်ခုက ကိုယ့်ဦးနှောက်ထဲမှာ အမြစ်တွယ်နေတဲ့ အယူအဆဟောင်းတွေဟာ အယူအဆသစ်တွေကို ဆန့်ကျင်တတ်ပါတယ်။ ပညာသင်ရခက်တယ်ဆိုတာ ဒါပါပဲ။ ဒီတော့ ပညာတတ်ချင်ရင် လောကကြီးကို ပွင့်လင်းတဲ့စိတ်ဓာတ်နဲ့ ကြည့်ရပါတယ်။ အသစ်အသစ်တွေကို နားလည်သိရှိလိုတဲ့စိတ် မွေးရပါတယ်။ အယူအဆဟောင်းတွေကို မစွဲလန်းရပါဘူး။ ဒီစာအုပ်ကိုသုံးပြီး Java ကို လေ့လာရာမှာ ဒီအလေ့အထတွေကို မွေးမြူရင်းဖတ်ရှုသွားရင် သင်ကြားမှုဖြစ်စဉ်ဟာ လွယ်ကူပြီး စိတ်ဝင်စားစရာကောင်းလာမှာပါ။

အရင်းရှင်တကာ့ထိပ်ခေါင် အမေရိကားမှာ အင်မတန်အရင်းရှင်မဆန်တဲ့ ကိစ္စတစ်ခုဖြစ်ပေါ်ခဲ့ပါတယ်။ ၁၉၈၀ ကျော်လောက်တွေဆီက MIT တက္ကသိုလ်ကျောင်းဝင်းထဲမှာ သုတေသနပညာရှင် Stallman ဟာ လွတ်လပ်တဲ့ ဆော့ဖ်ဝဲလှုပ်ရှားမှုကိုစတင်ပြီး လွတ်လပ်တဲ့ ဆော့ဖ်ဝဲဖောင်ဒေးရှင်းကို ထူထောင်ခဲ့ပါတယ်။ လွတ်လပ်တဲ့ဆော့ဖ်ဝဲတွေ ပေါ်ထွက်လာရေးအတွက် GPL လိုင်စင်ကိုလည်း ရေးသားခဲ့ပါတယ်။

အဲဒီလိုင်စင်ကိုသုံးပြီး ဆော့ဖ်ဝဲတစ်ခုကိုဖြန့်ဖြူးရင်း ရေးသားရတဲ့ မူလ code ကိုပါ ထည့်သွင်းဖြန့်ဖြူးရပါတယ်။ လက်ခံသူမှာ ဒီဆော့ဖ်ဝဲကို ပြင်ဆင်ခွင့်ရှိပါတယ်။ ပြန်လည်ဖြန့်ဖြူးခွင့်ရှိပါတယ်။ ဒီလိုဆော့ဖ်ဝဲဖြန့်ဖြူးပုံဟာ ကမ္ဘာပေါ်မှာ တစ်စတစ်စ အောင်မြင်မှု ပိုမိုရလာပြီး ဆော့ဖ်ဝဲတွေကိုသာမဟုတ်ဘဲ စာအုပ်တွေကိုပါ ဒီလိုင်စင်သုံးပြီး ဖြန့်ဖြူးလာပါတယ်။

ကျွန်တော်ရေးသားခဲ့တဲ့ ဒီစာအုပ်ဟာ နားလည်ရခက်ခဲတဲ့ ပရိုဂရမ်းမင်းသင်ရိုးတွေကို စိတ်ကုန်နေတဲ့ အမေရိကန်တက္ကသိုလ်ဆရာတစ်ယောက်က သူ့ကျောင်းသားတွေအတွက် သင်ရိုးအဖြစ် ရေးသားခဲ့တဲ့စာအုပ်ကို မှီးငြမ်းရေးသားထားတာ ဖြစ်ပါတယ်။ အရာရာကို မောင်ပိုင်စီးမထားချင်တဲ့ ပညာရှင်တွေရဲ့ဝသီအတိုင်း သူ့စာအုပ်ကို GPL လိုင်စင်သုံးပြီး အင်တာနက်ကနေ ဖြန့်ဖြူးခဲ့တာ ဝက်ဝက်ကွဲ အောင်မြင်ခဲ့ပါတယ်။

သူ့စာအုပ်ကို JavaScript စာအုပ်မှာ ကျွန်တော်ပြောခဲ့သလို ပညာရှင်လေးငါးယောက်လောက်ကပဲ ဝိုင်းစစ်ထားတာမဟုတ်ဘဲ တစ်ကမ္ဘာလုံးက ပညာရှင်တွေက ဝိုင်းစစ်ပေးပြီး အကြံပြုချက်တွေ ပေးပို့ခဲ့ပါတယ်။ ရလဒ်ကတော့ ပြည့်စုံတိကျပြီး အယူအဆတစ်ခုချင်းစီကို တစ်ဆင့်ချင်းရှင်းပြထားတဲ့ သင်ရိုးစာအုပ်ကောင်းတစ်အုပ် ဖြစ်ပါတယ်။

ဒါပေမယ့် သူဟာ ပရိုဂရမ်စမရေးခင် ပရိုဂရမ်းမင်းအယူအဆတွေကို နားလည်အောင် ကူညီပေးတဲ့စာအုပ်ဖြစ်ပြီး Core Java လို ပရိုဂရမ်ရေးနည်းစာအုပ်တော့ မဟုတ်ပါဘူး။ စာအုပ်ပြီးဆုံးသွားရင် Java မှာသာမကဘဲ ဘယ်ပရိုဂရမ်းမင်းဘာသာစကားမှာမဆို အသုံးဝင်တဲ့ အယူအဆတွေကို တတ်ကျွမ်းသွားမှာ ဖြစ်ပါတယ်။

ဒီတော့ ဒီစာအုပ်ကိုဖတ်ပြီး ပရိုဂရမ်းမင်း လေ့လာမှုလမ်းကြောင်းမှာ အောင်မြင်မှုတွေ ရပါစေလို့ ဆုမွန်ကောင်းတောင်းပြီး အမှာစာကို နိဂုံးချုပ်လိုက်ပါတယ်။

ခင်မင်မှုများစွာဖြင့်

ဆန္ဒအောင်

၁၉ နိုဝင်ဘာ၊ ၂၀၀၅။

အခန်း ၁

# ပရိုဂရမ်နည်းလမ်း

**ပရိုဂရမ်းမင်းဘာသာစကားဆိုတာဘာလဲ**

အခုသင်ယူမယ့် ဘာသာစကားဟာ Java ဖြစ်ပြီး ဘာသာစကား အသစ်တစ်ခုဖြစ်ပါတယ်။ (Sun ဟာ Java ကို ၁၉၉၅ မေမှာ စထုတ်ခဲ့ပါတယ်။) Java ဟာ အဆင့်မြင့်ဘာသာစကား (high-level language) တစ်ခုရဲ့ ဥပမာ ဖြစ်ပါတယ်။ တခြား အဆင့်မြင့်ဘာသာစကားတွေထဲမှာ Pascal, C, C++ နဲ့ FORTRAN တို့ ပါဝင်ပါတယ်။

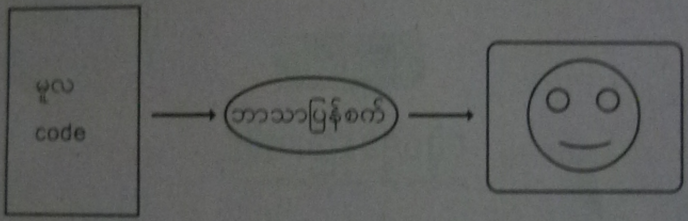
အဆင့်မြင့်ဘာသာစကားတွေရှိရင် အဆင့်နိမ့်ဘာသာစကား (low-level language)တွေလည်း ရှိပါတယ်။ သူတို့ကို စက်သုံးဘာသာစကား (machine language) ဒါမှမဟုတ် assembly ဘာသာစကားလို့လည်း ခေါ်ကြပါတယ်။ ကွန်ပျူတာဘာသာစကားတွေကို အဆင့်နိမ့်ဘာသာစကားလို့လည်း ခေါ်ကြပါတယ်။ ကွန်ပျူတာတွေဟာ အဆင့်နိမ့်ဘာသာစကားတွေနဲ့ ရေးသားတဲ့ ပရိုဂရမ်တွေကိုပဲ run နိုင်ပါတယ်။ ဒီတော့ အဆင့်မြင့် ဘာသာစကားတစ်ခုနဲ့ ရေးသားထားတဲ့ ပရိုဂရမ်တွေကို မ run ခင် ဘာသာပြန်ရပါတယ်။ ဒီလိုဘာသာပြန်ရတာ အချိန်နည်းနည်းကုန်ပြီး ဒါဟာ အဆင့်မြင့်ဘာသာစကားတွေရဲ့ အားနည်းချက်ဖြစ်ပါတယ်။

ဒါပေမယ့် အကျိုးကျေးဇူးတွေကတော့ အများကြီးပါပဲ။ အရင်ဆုံးအနေနဲ့ အဆင့်မြင့်ဘာသာစကားတစ်ခုမှာ ပရိုဂရမ်ရေးရတာ အများကြီးပိုလွယ်ပါတယ်။ ပိုလွယ်တယ်ဆိုတာ ပရိုဂ

ရမ်ရေးရတာ အချိန်ကုန်သက်သာတယ်၊ ဖတ်ရတာ ပိုတိုတယ်၊ ပိုလွယ်တယ်၊ မှန်ကန်ဖို့ ပိုအလားအလာရှိတယ်လို့ ဆိုလိုပါတယ်။ ဒုတိယအနေနဲ့ အဆင့်မြင့်ဘာသာစကားတွေဟာ သယ်ရလွယ်ပါတယ်။ အဓိပ္ပာယ်က သူတို့ဟာ ကွန်ပျူတာအမျိုးမျိုးမှာ ပြုပြင်ချက်တွေ လုံးဝမလုပ်ဘဲ ဒါမှမဟုတ် အနည်းငယ်ပြုလုပ်ပြီး run နိုင်ပါတယ်။ အဆင့်နိမ့်ပရိုဂရမ်တွေဟာ ကွန်ပျူတာတစ်မျိုးတည်းမှာပဲ run နိုင်ပြီး အခြားကွန်ပျူတာ အမျိုးအစားအတွက် ပြန်ရေးရပါတယ်။

ဒီလိုအကျိုးကျေးဇူးတွေကြောင့် ပရိုဂရမ်အားလုံးနီးပါးကို အဆင့်မြင့်ဘာသာစကားတွေနဲ့ ရေးပါတယ်။ အဆင့်နိမ့်ဘာသာစကားတွေကို လိုအပ်တဲ့ ပရိုဂရမ်အနည်းငယ်အတွက်ပဲ သုံးပါတယ်။

ပရိုဂရမ်တစ်ပုဒ်ကို ဘာသာပြန်နည်း နှစ်နည်းရှိပါတယ်။ စကားပြန်နည်း (interpreting) နဲ့ compile လုပ်နည်းဆိုပြီး ဖြစ်ပါတယ်။ စကားပြန်စက် (interpreter) ဆိုတာ အဆင့်မြင့်ပရိုဂ ရမ်တစ်ပုဒ်ကိုဖတ်ပြီး သူပြောသလိုလုပ်တဲ့ ပရိုဂရမ်တစ်ခုဖြစ်ပါတယ်။ သူဟာ ပရိုဂရမ်ကို တစ်ကြောင်းစီဖတ်ပြီး ဘာသာပြန်ပါတယ်။ စာကြောင်းတွေဖတ်လိုက် command တွေကို ဆောင်ရွက်လိုက် တစ်လှည့်စီလုပ်နေပါတယ်။




ဘာသာပြန်စက်က မူလ code ကို ဖတ်ပြီး

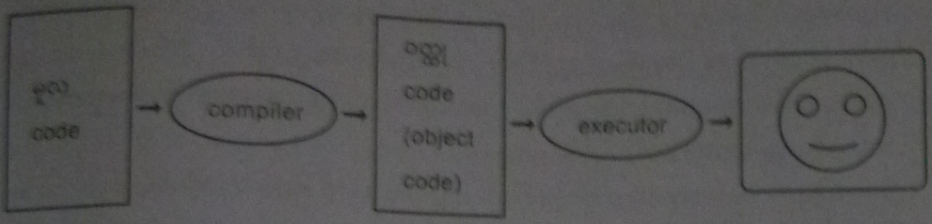
ရလဒ်ကို မြင်ကွင်းပေါ်မှာ မြင်ရပါတယ်။

Compiler ဆိုတာ အဆင့်မြင့် ပရိုဂရမ်တစ်ပုဒ်ကိုဖတ်ပြီး အားလုံးကို တစ်ပြိုင်တည်းဘာသာပြန်တဲ့ ပရိုဂရမ်ဖြစ်ပါတယ်။ ပြီးမှ command တွေကို ဆောင်ရွက်ပါတယ်။ တစ်ခါတစ်ရံမှာ ပရိုဂရမ်ကို သပ်သပ် compile လုပ်ပြီး နောက်မှ compile လုပ်ထားတဲ့ code ကို run တတ်ပါတယ်။ ဒီလိုအခြေအနေမှာ အဆင့်မြင့်ပရိုဂရမ်ကို မူလ code (source code) လို့ခေါ်ပြီး ဘာသာပြန်ထားတဲ့ ပရိုဂရမ်ကို object code ဒါမှမဟုတ် executable လို့ခေါ်ပါတယ်။

ဥပမာအနေနဲ့ C ကိုသုံးပြီး ပရိုဂရမ်တစ်ပုဒ် ရေးတယ်ဆိုပါစို့။ ပရိုဂရမ်ကိုရေးဖို့ စာသားတည်းဖြတ်ပရိုဂရမ် (text editor) တစ်ခုကို သုံးရပါတယ်။ (စာသားတည်းဖြတ် ပရိုဂရမ်ဆိုတာ ရိုးရှင်းတဲ့ စာစီစာရိုက် ပရိုဂရမ်တစ်ခုဖြစ်ပါတယ်။) ပရိုဂရမ်ရေးပြီးသွားရင် သူ့ကို program.c နာမည်နဲ့ သိမ်းပါတယ်။ ဒီနေရာမှာ program ဟာ ကိုယ်ပေးချင်လို့ လုပ်ကြံပေးထားတဲ့ နာမည်တစ်ခုဖြစ်ပြီး နောက်က .c ဟာ ဒီဖိုင်မှာ C မူလ code ပါဝင်တယ်လို့ ဆိုလိုပါတယ်။

ဒီအခါမှာ ကိုယ်သုံးတဲ့ ပရိုဂရမ်မင်း ပတ်ဝန်းကျင်ပေါ်မူတည်ပြီး command ကို run တဲ့ အလုပ်လုပ်ရပါမယ်။ Compiler ဟာ မူလ code ကို ဖတ်ပါတယ်။ ဘာသာပြန်ပါတယ်။ ပြီးရင်

program.o ဖိုင်ကိုရေးပြီး object code ကို ဖန်တီးပါတယ်။ မဟုတ်ရင် program.exe ကိုရေးပြီး executable ဖိုင်တစ်ခုကို ဖန်တီးနိုင်ပါတယ်။



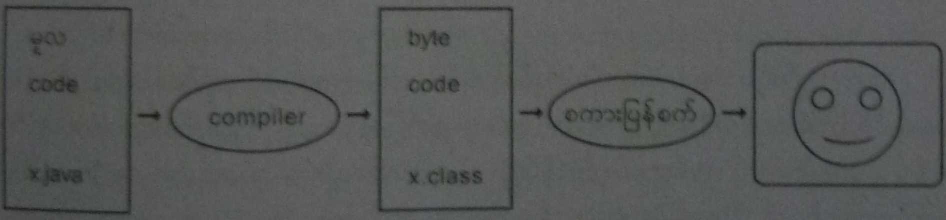

Compiler က မူလ code ကို ဖတ်ပြီး

ဝတ္ထု code ကို ထုတ်လုပ်ပါတယ်။

ပရိုဂရမ်ကို run တဲ့အခါ

ရလဒ်ဟာ မြင်ကွင်းပေါ်မှာ ပေါ်လာပါတယ်။

Java ဘာသာစကားဟာ ထူးခြားပါတယ်။ ဘာလို့လည်းဆိုတော့ သူဟာ compile လုပ်ထားတဲ့ ဘာသာစကားရော၊ စကားပြန်ဘာသာစကားရော ဖြစ်ပါတယ်။ Java ပရိုဂရမ်တွေကို စက်သုံးဘာသာစကားအဖြစ် တိုက်ရိုက်ဘာသာပြန်မယ့်အစား Java compiler ဟာ Java byte code ကို ထုတ်လုပ်ပါတယ်။ Byte code ဟာ စက်သုံးဘာသာစကားတစ်ခုလို စကားပြန်ရလွယ်ပြီး မြန်ပါတယ်။ ဒါ့အပြင် ပရိုဂရမ်တစ်ပုဒ်ကို စက်တစ်လုံးပေါ်မှာ compile လုပ်ပြီးရင် ကွန်ရက်ကနေ တခြားစက်တစ်လုံးပေါ် ပြောင်းပြီး အဲဒီစက်ပေါ်မှာ စကားပြန်လို့ရပါတယ်။ ဒီအရည်အချင်းဟာ Java ကို တခြား အဆင့်မြင့်ဘာသာစကားတွေထက် သာလွန်စေတဲ့ အကျိုးကျေးဇူးတစ်ခု ဖြစ်ပါတယ်။




Compiler က မူလ code ကို ဖတ်ပြီး

Java byte code ကို ထုတ်လုပ်ပါတယ်။

Java စကားပြန်စက်က byte code ကို ဖတ်တဲ့အခါ

ရလဒ်ကို မြင်ကွင်းပေါ်မှာ တွေ့ရပါတယ်။

ဒီဖြစ်စဉ်ဟာ ရှုပ်ထွေးတယ်လို့ထင်ရပေမယ့် ပရိုဂရမ်မင်း ပတ်ဝန်းကျင်တော်တော်များများမှာ ဒီအဆင့်တွေကို အလိုအလျောက် ဆောင်ရွက်ပေးပါတယ်။ ပုံမှန်အားဖြင့် ပရိုဂရမ်တစ်ပုဒ်ကိုရေးပြီး ခလုတ်တစ်ခု နှိပ်လိုက်ရုံပါပဲ။ မဟုတ်ရင် command တစ်ခုကို ရိုက်ထည့်ပြီး run ရုံပါပဲ။ ဒါပေမယ့် နောက်ကွယ်မှာ ဘာတွေဖြစ်နေသလဲဆိုတာ သိထားရင် ထင်ထားတာတွေဖြစ်မလာတဲ့အခါ ဘာကြောင့်လဲဆိုတာ နားလည်နိုင်တဲ့အတွက် အသုံးဝင်ပါတယ်။

**ပရိုဂရမ်တစ်ပုဒ်ဆိုတာဘာလဲ**

ပရိုဂရမ်တစ်ပုဒ်ဆိုတာ တွက်ချက်မှုတစ်ခုကို ဘယ်လိုလုပ်ရမလဲဆိုတာ ဖော်ပြထားတဲ့ ညွှန်ကြားချက် တစ်သီတစ်တန်းကြီးပဲ ဖြစ်ပါတယ်။ ဒီတွက်ချက်မှုဟာ ညီမျှခြင်းဖြေရှင်းတာတို့ ပိုလီနိုမီယယ်လ်တစ်ခုရဲ့ အမြစ်ရှာတာတို့လို သင်္ချာပြဿနာတွေ ဖြစ်နိုင်ပါတယ်။ ဒါပေမယ့် စာရွက်စာတမ်းတစ်ခုထဲမှာ စာသားရှာပြီး အစားထိုးတာတို့ (ထူးဆန်းစွာနဲ့ပဲ) ပရိုဂရမ်တစ်ပုဒ်ကို compile လုပ်တာတို့လည်း ဖြစ်နိုင်ပါတယ်။

ဖော်ပြချက် (statement) တွေလို့ခေါ်တဲ့ ဒီညွှန်ကြားချက်တွေဟာ ပရိုဂရမ်မင်းဘာသာစကားပေါ် လိုက်ပြီး ကွဲပြားပါတယ်။ ဒါပေမယ့် ဘာသာစကား တော်တော်များများက ဆောင်ရွက်နိုင်တဲ့ အခြေခံလုပ်ဆောင်ချက်အချို့ ရှိပါတယ်။

Input။ ။ Keyboard ဒါမှမဟုတ် ဖိုင်တစ်ခု ဒါမှမဟုတ် အခြားအစိတ်အပိုင်းတစ်ခုကနေ data ရယူပါတယ်။

Output။ ။ ဖိုင်တစ်ခု ဒါမှမဟုတ် တခြားကွန်ယက်အစိတ်အပိုင်းတစ်ခုကို data ပို့ပါတယ်။ မြင်ကွင်းပေါ်မှာ data ကို ပြသပါတယ်။

သင်္ချာ ။ ။ ပေါင်းခြင်းနဲ့ မြှောက်ခြင်းတို့လို အခြေခံ သင်္ချာလုပ်ဆောင်ချက်တွေကို ဆောင်ရွက်ပါတယ်။

စစ်ဆေးခြင်း။ ။ အခြေအနေတချို့အတွက် စမ်းသပ်ပြီး သင့်တော်တဲ့ ဖော်ပြချက်အစီအစဉ်တွေကို run ပါတယ်။

အကြိမ်ကြိမ်ဆောင်ရွက်ခြင်း ။ ။ လုပ်ဆောင်ချက်တစ်ခုကို ထပ်ခါထပ်ခါဆောင်ရွက်ပါတယ်။ များသောအားဖြင့် ဆောင်ရွက်ရင်းလုပ်ဆောင်ချက်မှာ ပြောင်းလဲမှုတစ်ခုကို ပြုလုပ်ပါတယ်။

ဒါဟာ လုပ်လို့ရသမျှ အားလုံးနီးပါးဖြစ်ပါတယ်။ ဘယ်ပရိုဂရမ်မဆို ဘယ်လောက်ပဲရှုပ်ထွေးနေနေ ဒီလုပ်ဆောင်ချက်တွေကို ဆောင်ရွက်တဲ့ ဖော်ပြချက်တွေနဲ့ပဲ ဖွဲ့စည်းထားပါတယ်။ ဒီတော့ ပရိုဂရမ်မင်းကို အဓိပ္ပာယ်ဖွင့်ဆိုနိုင်တဲ့ နောက်တစ်နည်းက ရှုပ်ထွေးတဲ့ အလုပ်အကိုင်အကြီးစားတစ်ခုကို ဒီအခြေခံလုပ်ဆောင်ချက်တွေနဲ့ ဆောင်ရွက်နိုင်တဲ့အထိ သေးငယ်တဲ့ အပိုင်းတွေအဖြစ် ခွဲထုတ်တဲ့ဖြစ်စဉ်ပဲ ဖြစ်ပါတယ်။

**အမှားပြင်တယ်ဆိုတာဘာလဲ**

ပရိုဂရမ်မင်းဟာ ရှုပ်ထွေးတဲ့ ဖြစ်စဉ်တစ်ခုဖြစ်ပါတယ်။ သူ့ကိုလူတွေက လုပ်တာဖြစ်တဲ့အတွက် အမှားတွေ ပေါ်ပေါက်တတ်ပါတယ်။ ကသိကအောက်နိုင်တဲ့ အကြောင်းပြချက်တွေကြောင့် ပရိုဂရမ်မင်းအမှားတွေကို bug တွေလို့ ခေါ်ပါတယ်။ အမှားရှာပြီးပြင်တဲ့ ဖြစ်စဉ်ကို debugging လို့ ခေါ်ပါတယ်။

ပရိုဂရမ်တစ်ပုဒ်မှာ ပေါ်နိုင်တဲ့အမှားအမျိုးအစား အနည်းငယ်ရှိပါတယ်။ သူတို့ကို ခွဲခြားရင် အမှားရှာရတာ ပိုလွယ်ပါတယ်။

## Compile လုပ်ချိန်မှ အမှားများ

Compiler ဟာ ပရိုဂရမ်တစ်ပုဒ်ကို သဒ္ဒါမှန်မှ ဘာသာပြန်ပါတယ်။ မဟုတ်ရင် compile လုပ်တာ မအောင်မြင်ဘဲ ပရိုဂရမ်ကို run လို့မရပါဘူး။ Syntax လို့ခေါ်တဲ့ သဒ္ဒါဟာ ပရိုဂရမ်တည်ဆောက်ပုံနဲ့ အဲဒီတည်ဆောက်ပုံနဲ့ ပတ်သက်တဲ့စည်းမျဉ်းတွေကို ဆိုလိုပါတယ်။

ဥပမာ အင်္ဂလိပ်စာမှာ စာကြောင်းကို အက္ခရာကြီးနဲ့ စရပြီး full stop တစ်ခုနဲ့ဆုံး ရပါတယ်။ စာဖတ်သူတော်တော်များများအတွက် သဒ္ဒါအမှားအနည်းလောက်ဟာ ပြောပလောက်တဲ့ ပြဿနာမဟုတ်ပါဘူး။ ဒါကြောင့်လည်း ကဗျာတွေ ဖတ်ပြီးနားလည်နိုင်တာ ဖြစ်ပါတယ်။ ဘယ်သူမှ ကဗျာဖတ်ပြီး compiler က ထုတ်သလို အမှား message တွေကို မထုတ်ပါဘူး။

Compiler တွေကတော့ ဒီလောက်ခွင့်မလွှတ်တတ်ပါဘူး။ ပရိုဂရမ်ရဲ့ ဘယ်နေရာမှာပဲ ဖြစ်ဖြစ် အမှားတစ်ခုပါလာခဲ့ရင် compiler ဟာ အမှား message တစ်ခုကို ထုတ်လုပ်ပြီး ထွက်သွားပါလိမ့်မယ်။ အဲဒီအခါ ပရိုဂရမ်ကို run လို့ မရနိုင်တော့ပါဘူး။

ပြဿနာတွေကို ပိုဆိုးစေတာက Java မှာ အင်္ဂလိပ်စာထက် သဒ္ဒါစည်းကမ်းတွေ ပိုများပါတယ်။ ပြီးတော့ compiler ကပေးတဲ့ အမှား message တွေကလည်း အကူအညီသိပ်မရပါဘူး။ ပရိုဂရမ်မာဘဝရဲ့ ပထမဆုံး ရက်သတ္တပတ်အနည်းငယ်ကို သဒ္ဒါအမှားတွေလိုက်ရှာရင်း ကုန်လွန်ရပါတယ်။ အတွေ့အကြုံရလာတာနဲ့အမျှ အမှားလုပ်တာနည်းလာပြီး အမှားရှာရတာလည်း မြန်လာပါတယ်။

## Run ချိန်မှ အမှားများ

ဒုတိယအမှားအမျိုးအစားဟာ ပရိုဂရမ် run နေချိန်မှာ ဖြစ်ပေါ်ပါတယ်။ ဒီအမှားဟာ ပရိုဂရမ်ကို မ run မချင်း မပေါ်ပါဘူး။ Java မှာ ဒီပြဿနာဟာ စကားပြန်စက်က byte code ကို run နေရင်း တစ်ခုခုမှားယွင်းသွားရင် ပေါ်လာပါတယ်။

ကောင်းသတင်းကတော့ Java ဟာ လုံခြုံတဲ့ ဘာသာစကားတစ်ခုဖြစ်ပါတယ်။ ဆိုလိုတာက run မှဖြစ်တဲ့ အမှားတွေနည်းပါတယ်။ အထူးသဖြင့် လတ်တလောရေးကြမယ့် ရိုးရှင်းတဲ့ ပရိုဂရမ်တွေအတွက် လုံခြုံပါလိမ့်မယ်။

Java မှာ run နေရင်းဖြစ်တဲ့ အမှားတွေကို ခြွင်းချက် (exception) တွေလို့ ခေါ်ပါတယ်။ ပတ်ဝန်းကျင်တော်တော်များများမှာ သူတို့ဟာ ဘာဖြစ်သွားသလဲ၊ အမှားဖြစ်တဲ့အချိန်မှာ ပရိုဂရမ်ဟာ ဘာလုပ်နေသလဲ စတဲ့အချက်အလက်တွေ ပါဝင်တဲ့ window တွေ၊ dialogue box တွေအနေနဲ့ ပေါ်လာပါတယ်။ ဒီအချက်အလက်တွေဟာ အမှားပြင်တာ (debugging) အတွက် အသုံးဝင်ပါတယ်။

### ယုတ္တိအမှားများနှင့် အဓိပ္ပာယ်လွဲမှုများ

တတိယအမှားအမျိုးအစားကတော့ ယုတ္တိဗေဒမှားတဲ့ အဓိပ္ပာယ်မှားတာ ဖြစ်ပါတယ်။ ပရိုဂရမ်ထဲမှာ ယုတ္တိအမှားတစ်ခုပါသွားရင် ကွန်ပျူတာက အမှား message တွေ မထုတ်လုပ်ပဲ compile လုပ်ပြီး ကောင်းကောင်း run ပါလိမ့်မယ်။ ဒါပေမယ့် လုပ်စေချင်တဲ့ အလုပ်တော့လုပ်မှာမဟုတ်ပါဘူး။ တခြားအလုပ်တစ်ခုကို လုပ်ပါလိမ့်မယ်။ ကိုယ်မှားပြောခဲ့တဲ့ အလုပ်ကို လုပ်ပါလိမ့်မယ်။

ပြဿနာက ကိုယ်ရေးလိုက်တဲ့ ပရိုဂရမ်ဟာ ကိုယ်ရေးချင်တဲ့ ပရိုဂရမ်မဟုတ်ပါဘူး။ ပရိုဂရမ်ရဲ့ အဓိပ္ပာယ်မှားနေပါတယ်။ ယုတ္တိအမှားတွေရှာရတာ ခက်ခဲနိုင်ပါတယ်။ ဘာလို့လဲဆိုတော့ ပရိုဂရမ်ကထုတ်ပေးတဲ့ output ကိုကြည့်ပြီး သူဘာလုပ်နေတာလဲလို့ နောက်ကြောင်းပြန်စဉ်းစားရလို့ပါ။

### သုတေသနသဘောဆန်သော အမှားပြင်ဆင်ခြင်း

ဒီစာအုပ်ဖတ်ပြီး ရရှိမယ့် အရေးအပါဆုံး အရည်အချင်းတစ်ခုကတော့ အမှားပြင်တာပဲ ဖြစ်ပါတယ်။ အမှားလိုက်ရှာပြီး ပြင်ရတာဟာ စိတ်ညစ်ညူးစရာကောင်းနိုင်ပေမယ့် ဉာဏ်ပညာအရ အကြွယ်ဝဆုံး၊ စိတ်ခေါ်မှုအရှိဆုံးနဲ့ စိတ်ဝင်စားစရာအကောင်းဆုံး ပရိုဂရမ်မင်းအလုပ်တစ်ခု ဖြစ်ပါတယ်။

တစ်မျိုးပြောရရင် အမှားပြင်ရတာဟာ စုံထောက်လုပ်ရတာနဲ့ တူပါတယ်။ ကိုယ့်ကို သဲလွန်စတွေပေးထားပြီး ကိုယ်မြင်နေရတဲ့ ရလဒ်တွေကို ဖြစ်ပေါ်စေတဲ့ ဖြစ်စဉ်တွေ၊ ဖြစ်ရပ်တွေကို စဉ်းစားယူရပါတယ်။

အမှားပြင်ရတာဟာ လက်တွေ့လုပ်ရတဲ့ သိပ္ပံပညာနဲ့လည်းတူပါတယ်။ ဘာတွေမှားသွားသလဲလို့ စဉ်းစားမိတာနဲ့ ပရိုဂရမ်ကိုပြုပြင်ပြီး ထပ်စမ်းသပ်ကြည့်ရပါတယ်။ ကိုယ့်အယူအဆကမှန်ရင် ပြုပြင်မှုရဲ့ရလဒ်ကို ကြိုတင်ခန့်မှန်းနိုင်ပါတယ်။ စုံထောက်ကြီး ရှားလော့ဟုမ်း ပြောခဲ့သလိုပဲ "မဖြစ်နိုင်တာတွေကို ဖယ်ရှားပြီးပြီဆိုရင် ကျန်တာတွေဟာ ဘယ်လောက်ပဲ ယုတ္တိမတန်ဖြစ်နေပါစေ အမှန်တရားဖြစ်ရမယ်။"

လူတချို့အတွက်တော့ ပရိုဂရမ်ရေးတာနဲ့ အမှားပြင်တာဟာ အတူတူဖြစ်ပါတယ်။ ဆိုလိုတာက ပရိုဂရမ်ရေးတယ်ဆိုတာ ပရိုဂရမ်က ကိုယ်ရေးတာမလုပ်မချင်း တဖြည်းဖြည်းအမှားပြင်ယူသွားတာ ဖြစ်ပါတယ်။ သဘောက ပရိုဂရမ်ရေးရာမှာ အလုပ်တစ်ခုခုကို လုပ်ပေးတဲ့ ပရိုဂရမ်တစ်ပုဒ်နဲ့ အမြဲစတင်ရပြီး သေးငယ်တဲ့ပြုပြင်မှုတွေကို ပြုလုပ်ရပါမယ်။ လုပ်ရင်းကိုင်ရင်း အမှားပြင်သွားရင် နောက်ဆုံးမှာ အလုပ်ဖြစ်တဲ့ ပရိုဂရမ်တစ်ပုဒ်ကို အမြဲရပါတယ်။

ဥပမာ Linux ဟာ code တွေ စာကြောင်းသောင်းချီပြီးပါတဲ့ operating system တစ်ခုဖြစ်

ပါတယ်။ ဒါပေမယ့် ဒီ operating system ဟာ Intel ရဲ့ 80386 ချစ်ပ်ပြားကိုလေ့လာဖို့ Linus Torvalds က သုံးခဲ့တဲ့ ပရိုဂရမ်ရိုးရိုးရှင်းရှင်းလေးအနေနဲ့ စတင်ခဲ့ပါတယ်။ Larry Greenfield ရဲ့ The Linux Users' Guide မှာပြောသလိုဆိုရင် Linus ရဲ့ အစောပိုင်း စီမံကိန်းတစ်ခုဟာ AAAA နဲ့ BBBB တို့ print လုပ်တာကို ပြောင်းပေးတဲ့ ပရိုဂရမ်တစ်ပုဒ် ဖြစ်ပါတယ်။ ဒီအရာဟာ နောက်ပိုင်းမှာ Linux ဖြစ်လာပါတယ်။

**ပထမဆုံးပရိုဂရမ်**

ထုံးလမ်းစဉ်လာအရ ဘာသာစကားအသစ်တစ်ခုမှာ စရေးတဲ့ ပရိုဂရမ်ကို Hello, World လို့ နာမည်ပေးတတ်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူလုပ်တဲ့ အလုပ်ဟာ Hello, World ဆိုတဲ့ စကားလုံးတွေကိုပဲ ပြသလို့ပါ။ Java မှာ ပရိုဂရမ်ဟာ ဒီလိုပုံရှိပါတယ်။

```
class Hello {
   //main: generate some simple output
   public static void main (String[] args) {
      System.out.println ("Hello, World");
   }
}
```

တချို့က ပရိုဂရမ်မင်းဘာသာစကားတစ်ခုရဲ့ အရည်အသွေးကို Hello, World ပရိုဂရမ်ရဲ့ ရိုးရှင်းမှုနဲ့ တိုင်းတာပါတယ်။ ဒီစံနှုန်းအရ Java ဟာ သိပ်အကောင်းကြီးမဟုတ်ပါဘူး။ အရိုးရှင်းဆုံးပရိုဂရမ်မှာတောင်မှ အခုမှစပြီးသင်ယူတဲ့ ပရိုဂရမ်မာတွေ နားလည်အောင် ရှင်းပြရခက်တဲ့ လက္ခဏာတချို့ပါဝင်ပါတယ်။ အခုတော့ ဒီကိစ္စတွေကို ခဏမေ့ထားရအောင်။ တချို့တစ်ဝက်ကိုနားလည်ရင် အလုပ်စလုပ်နိုင်ပါပြီ။

ပရိုဂရမ်အားလုံးကို class defination လို့ခေါ်တဲ့ class သတ်မှတ်ချက်တွေနဲ့ ဖွဲ့စည်းထားပါတယ်။ သူ့ပုံစံက ဒီလိုရှိပါတယ်။

```
class CLASSNAME {
   public static void main (String[] args) {
      STATEMENTS
   }
}
```

ဒီနေရာ CLASSNAME ဟာ ကိုယ်ဖန်တီးထားတဲ့ နာမည်တစ်ခုကို ဆိုလိုပါတယ်။ ဥပမာမှာ

class နာမည်ဟာ Hello ဖြစ်ပါတယ်။

ဒုတိယစာကြောင်းမှာတွေ့ရတဲ့ public static void ကို အခုအချိန်မှာ မေ့ထားလိုက်ပါ။ ဒါပေမယ့် main စာလုံးကို သတိပြုပါ။ main ဟာ ပရိုဂရမ်ကို စ run တဲ့ နေရာကိုဖော်ပြတဲ့ နာမည်ဖြစ်ပါတယ်။ ပရိုဂရမ် run တဲ့အခါ main ထဲက ပထမဆုံးဖော်ပြချက်ကို စ run ပြီး အစီအစဉ်အတိုင်း ဆက် run ပါတယ်။ နောက်ဆုံးဖော်ပြချက်အပြီးမှာ ထွက်သွားပါတယ်။

main ထဲမှာရှိနိုင်တဲ့ ဖော်ပြချက်အရေအတွက် ကန့်သတ်ထားတာ မရှိပါဘူး။ ဒါပေမယ့် ဥပမာမှာ တစ်ခုပဲပါပါတယ်။ ဒါဟာ print ဖော်ပြချက် ဖြစ်ပါတယ်။ အဓိပ္ပာယ်က သူဟာမြင်ကွင်းပေါ်မှာ message တစ်ခုကို print လုပ်ပါတယ်။ Print ဆိုတဲ့ စကားလုံးဟာ တစ်ခါတစ်ရံမှာ မြင်ကွင်းပေါ် တစ်ခုခုပြသတယ်လို့ အဓိပ္ပာယ်ရပြီး တစ်ခါတစ်ရံမှာ ပရင်တာဆီ တစ်ခုခုပို့လိုက်တယ်လို့ အဓိပ္ပာယ်ရတာ နားရှုပ်စရာဖြစ်ပါတယ်။ ဒီစာအုပ်ထဲမှာ ပရင်တာဆီကို တစ်ခုခုပို့တဲ့အကြောင်းမပါပါဘူး။ မြင်ကွင်းပေါ်မှာပဲ print လုပ်ပါမယ်။

မြင်ကွင်းပေါ်မှာ print လုပ်ပေးတဲ့ ဖော်ပြချက်ဟာ System.out.println ဖြစ်ပြီး လက်သည်းကွင်းတွေထဲက အရာကတော့ print လုပ်လိုတဲ့အရာ ဖြစ်ပါတယ်။ ဖော်ပြချက်အဆုံးမှာ semicolon (;) တစ်ခုရှိပြီး သူ့ကို ဖော်ပြချက်ဆုံးတိုင်း လိုအပ်ပါတယ်။

ဒီပရိုဂရမ်ရဲ့ သဒ္ဒါမှာ သတိပြုစရာ အချက်အနည်းငယ်ရှိပါသေးတယ်။ ပထမဆုံးအနေနဲ့ Java ဟာ ဖော်ပြချက်တွေကို စုစည်းဖို့ တွန့်ကွင်း ( { နဲ့ } ) တွေကို သုံးပါတယ်။ စာကြောင်း 1 နဲ့ 6 မှာရှိတဲ့ အပြင်ဘက်တွန့်ကွင်းတွေမှာ class အဓိပ္ပာယ်သတ်မှတ်ချက်ပါဝင်ပြီး အတွင်းတွန့်ကွင်းတွေမှာ main ရဲ့ အဓိပ္ပာယ်ဖွင့်ဆိုချက် ပါဝင်ပါတယ်။

ဒုတိယစာကြောင်းကို // နဲ့ စတယ်ဆိုတာလည်း သတိပြုပါ။ အဲဒါရဲ့ ဆိုလိုရင်းက ဒီစာကြောင်းမှာ မှတ်ချက် (comment) တစ်ခုပါတယ်ဆိုတာ ရှင်းပြတာပါပဲ။ မှတ်ချက်ဆိုတာ များသောအားဖြင့် ပရိုဂရမ်ဘာလုပ်တယ်ဆိုတာရှင်းပြတဲ့ ပရိုဂရမ်အလယ်က အင်္ဂလိပ်စာဖြစ်ပါတယ်။ Compiler က // တစ်ခုကိုမြင်ရင် အဲဒီနေရာကစပြီး စာကြောင်းဆုံးတဲ့အထိ ရှိရှိသမျှကို ဥပေက္ခာပြုပါတယ်။

အခန်း ၂

# Variable များနှင့် Type များ

## ပုံနှိပ် Print လုပ်ခြင်း

ပြီးခဲ့တဲ့အခန်းမှာ ပြောခဲ့သလိုပဲ main မှာ ဖော်ပြချက် ကြိုက်သလောက် ထည့်နိုင်ပါတယ်။ ဥပမာ စာကြောင်းတစ်ကြောင်းထက်ပိုပြီး print လုပ်ချင်ရင် အောက်မှာ ပြထားသလို ဆောင်ရွက်ရပါတယ်။

```
class Hello {
  // main: generate some simple output
  public static void main (String[] args) {
    System.out.println ("Hello, world.");  //print one line
    Syetem.out.println ("How are you?"); //print another
  }
}
```

အခုမြင်ရသလိုပဲ မှတ်ချက်တွေကို သူတို့ဘာသာ စာကြောင်းတစ်ကြောင်းရဲ့ အဆုံးမှာ

ထည့်သလို တခြားစာကြောင်းတစ်ကြောင်းရဲ့ အဆုံးမှာလည်း ထားနိုင်ပါတယ်။

Inverted comma တွေကြားမှာ မြင်ရတဲ့ဝါကျတွေဟာ string လို့ခေါ်တဲ့ စာသားတွေ ဖြစ်ပါတယ်။ ဘာလို့ ဒီလိုခေါ်လဲလို့ဆိုတော့ သူတို့ကို အက္ခရာအစဉ် ဒါမှမဟုတ် string တစ်ခုနဲ့ ဖွဲ့စည်းထားလို့ပါပဲ။ တကယ်တမ်းဆိုရင် string တွေမှာ အက္ခရာ၊ ဂဏန်း၊ punctuation mark တွေနဲ့ တခြားအထူးအက္ခရာတွေ ကြိုက်တဲ့အစီအစဉ်နဲ့ ပါဝင်နိုင်ပါတယ်။

pintln ဟာ printline ရဲ့ အတိုကောက်ဖြစ်ပါတယ်။ သူဟာ စာကြောင်းအဆုံးမှာ newline လို့ခေါ်တဲ့ cursor ကို နောက်တစ်ကြောင်းဆင်းစေတဲ့ အက္ခရာကို ပေါင်းထည့်ပါတယ်။ println ကို နောက်တစ်ကြိမ်ခေါ်ယူချိန်မှာ စာသားအသစ်ဟာ စာကြောင်းနောက်တစ်ကြောင်းအနေနဲ့ ပေါ်ပါတယ်။

တစ်ခါတစ်ရံမှာ တစ်ခုထက်ပိုတဲ့ print ဖော်ပြချက်တွေရဲ့ output တွေကို စာကြောင်းတစ်ကြောင်းတည်းပေါ်မှာ ပြသဖို့လိုပါတယ်။ ဒါကို print command နဲ့ လုပ်နိုင်ပါတယ်။

```
class Hello {
   //main: generate some simple output
   public static void main (String[] args) {
      System.out.print ("Goodbye, ");
      System.out.println ("cruel world!");
   }
}
```

ဒီနေရာမှာ output ဟာ Goodbye, cruel world! ဆိုပြီး တစ်ကြောင်းတည်း ပေါ်ပါတယ်။ Goodbye နဲ့ inverted comma အပိတ်ကြားမှာ ကွက်လပ်တစ်ခုရှိနေတာကို သတိပြုပါ။ ဒီကွက်လပ်ဟာ output မှာပေါ်တဲ့အတွက် ပရိုဂရမ်ရဲ့ အပြုအမူအပေါ် လွှမ်းမိုးမှုရှိပါတယ်။

Inverted comma တွေ မြင်ပမှာပေါ်တဲ့ ကွက်လပ်တွေကတော့ ပရိုဂရမ်အပြုအမူကို မလွှမ်းမိုးနိုင်ပါဘူး။ ဥပမာ အောက်မှာဆိုထားသလို ရေးနိုင်ပါတယ်။

```
clas Hello {
public static void main (String[] args) {
System.out.print ("Goodbye, ");
System.out.println ("cruel world!")'
}
}
```

ပရိုဂရမ်ဟာ မူလပရိုဂရမ်လိုပဲ compile လုပ်ပြီး run ပါလိမ့်မယ်။ စာကြောင်းဆုံးရင် နောက်ထပ် တစ်ကြောင်းဆင်းတာဟာလည်း ပရိုဂရမ်အပြုအမူအပေါ် သက်ရောက်မှု မရှိပါဘူး။ ဒီတော့ အခုလိုလည်း ရေးနိုင်ပါတယ်။

```
class Hello { public static void main (String[] args) {
System.out.print ("Goodbye, "); System.out.println
("curel world!"); }}
```

ဒါလည်း အလုပ်ဖြစ်ပါတယ်။ ဒါပေမယ့် ပရိုဂရမ်ဟာ ဖတ်ရပိုပြီးခက်လာတယ် ဆိုတာကို တော့ သတိမူမိမှာပါ။ စာကြောင်းခွဲတာနဲ့ ကွက်လပ်တွေဟာ ပရိုဂရမ်ကို ဖတ်ရလွယ်အောင် နဲ့ သဒ္ဒါအမှားတွေကို ရှာတွေ့ရလွယ်အောင် ရှင်းလင်းတဲ့အမြင်နဲ့ စီစဉ်ခွင့်ပေးပါတယ်။

## Variable များ

ပရိုဂရမ်မင်းဘာသာစကားတစ်ခုရဲ့ စွမ်းအားအကြီးဆုံး လက္ခဏာတစ်ခုဟာ variable တွေကို ကိုင်တွယ်နိုင်တဲ့ အရည်အချင်းပဲဖြစ်ပါတယ်။ Variable ဆိုတာ တန်ဖိုးတစ်ခုကို သိုလှောင်နိုင်တဲ့ နေရာတစ်ခု ဖြစ်ပါတယ်။ အဲဒီနေရာကို နာမည်ပေးထားတာ ဖြစ်ပါတယ်။ တန်ဖိုးတွေဟာ print လုပ်နိုင်၊ သိမ်းဆည်းနိုင်၊ လုပ်ဆောင်ချက်တွေ ပြုလုပ်နိုင်တဲ့ အရာတွေ ဖြစ်ပါတယ်။ အခု print လုပ်နေခဲ့တဲ့ စာသားတွေ ("Hello, world.", "Goodbye, "စတာတွေ) ဟာ တန်ဖိုးတွေဖြစ်ပါတယ်။

တန်ဖိုးတစ်ခုကို သိုလှောင်ဖို့ varaible တစ်ခုကို ဖန်တီးရပါတယ်။ သိုလှောင်လိုတဲ့ တန်ဖိုးတွေဟာ string တွေ ဖြစ်တဲ့အတွက် variable အသစ်ကို string တစ်ခုအနေနဲ့ ကြေညာပါမယ်။

```
String fred;
```

ဒီဖော်ပြချက်ကို ကြေညာချက် (declaration) လို့ခေါ်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူဟာ fred ဆိုတဲ့ variable ကို ကြေညာပြီး String အမျိုးအစား (type) အဖြစ် သတ်မှတ်ပါတယ်။ Variable တစ်ခုမှာ ဘယ်လိုတန်ဖိုးတွေ သိုလှောင်နိုင်သလဲဆိုတာ ဆုံးဖြတ်တဲ့ အမျိုးအစား (type) တစ်ခုရှိပါတယ်။ ဥပမာ int type ဟာ ကိန်းပြည့် (integer) တွေကို သိုလှောင်နိုင်ပြီး String အမျိုးအစားဟာ စာသား (string) တွေကို သိုလှောင်နိုင်ပါတယ်။

တချို့ type တွေဟာ အက္ခရာအကြီးနဲ့စပြီး တချို့က အက္ခရာအသေးနဲ့ စတယ်ဆိုတာ သတိပြုမိပါလိမ့်မယ်။ ဒီခွဲခြားမှုရဲ့ အရေးပါပုံကို နောက်မှ လေ့လာပါမယ်။ ဒါပေမယ့် အခုချိန်မှာ သူတို့ကို မှန်အောင်လုပ်ပါ။ Int တို့ string တို့လို type မျိုး မရှိပါဘူး။ ဒီလိုမျိုးလုပ်ကြံရင် compile က ကန့်ကွက်ပါလိမ့်မယ်။

ကိန်းပြည့် variable တစ်ခုကို ဖန်တီးတဲ့ သဒ္ဒါက int bob; ပါ။ ဒီနေရာမှာ bob ဟာ variable အတွက် လုပ်ကြံထားတဲ့ နာမည်ဖြစ်ပါတယ်။ ယေဘုယျအားဖြင့် variable နာမည်

တွေအများကြီး ဒီ variable တွေနဲ့ ဘာလုပ်မလဲဆိုတာ သိရှိစေတဲ့ နာမည်တွေကိုပဲ ရွေးသင့်ပါတယ်။ ဥပမာ အောက်မှာမြင်ရတဲ့ variable ကြေညာချက်တွေကိုမြင်ရင် သူတို့ထဲမှာ ဘာတွေသိမ်းထားမလဲဆိုတာ ခန့်မှန်းနိုင်ပါတယ်။

```
String firstName;
String lastName;
int hour, minute;
```

ဒီ ဥပမာဟာ အမျိုးအစားတူတဲ့ တန်ဖိုးတွေကိုကြေညာတဲ့ သင်္ကေတကို သရုပ်ဖော်ပါတယ်၊ hour နဲ့ second နှစ်ခုစလုံးဟာ int အမျိုးအစားရှိတဲ့ ကိန်းပြည့်တွေ ဖြစ်ပါတယ်။

### နေရာချထားခြင်း

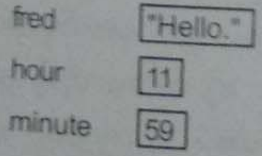
Variable အချို့ကို ဖန်တီးပြီးအခါမှာ သူတို့ထဲမှာ တန်ဖိုးတွေသိမ်းဆည်းလိုပါတယ်၊ ဒါကို နေရာချထားခြင်းဖော်ပြချက် (assignment statement) နဲ့ ပြုလုပ်ပါတယ်။

```
fred = "Hello."; // give fred the value  "Hello."
hour = 11; // assign the value 11 to hour
minute = 59; // set minute to 59
```

ဒီဥပမာဟာ နေရာချထားမှု သုံးခုကိုပြပြီး သူတို့ရဲ့ မှတ်ချက်တွေဟာ နေရာချထားမှုအကြောင်း အလိုလိုပြောနည်း သုံးပြပါတယ်။ စကားလုံးတွေဟာ ဒီနေရာမှာ နည်းနည်းရှုပ်ထွေးနိုင်ပါတယ်၊ ဒါပေမယ့် ပြောချင်တဲ့ အကြောင်းအရာကတော့ ရှင်းပါတယ်။

- Variable တစ်ခုကို ကြေညာတဲ့အခါ နာမည်ပေးထားတဲ့ သိမ်းဆည်းမှုနေရာကို ဖန်တီးပါတယ်။
- Variable တစ်ခုမှာ နေရာချထားမှု တစ်ခုလုပ်ချင်ရင် သူ့ကို တန်ဖိုးတစ်ခု ထည့်ပေးပါတယ်။

စာရွက်ပေါ်မှာ variable တွေကို သရုပ်ဖော်တဲ့နည်းက လေးထောင့်ကွက်ကလေးဆွဲပြီး အပြင်မှာ variable နာမည်ရေး၊ အထဲမှာ variable တန်ဖိုးရေးတဲ့နည်းပါပဲ။ အောက်မှာပြထားတဲ့ပုံဟာ ခုနကပြောတဲ့ နေရာချထားတာတွေကို ပြသပါတယ်။




Variable တစ်ခုစီအတွက် variable နာမည်ဟာ လေးထောင့်ကွက်အပြင်မှာရှိပြီး တန်

ဖိုးဟာ အတွင်းမှာရှိပါတယ်။

ယေဘုယျပြောရရင် variable တစ်ခုမှာ သူနဲ့ အမျိုးအစားတူတဲ့ တန်ဖိုးတစ်ခုကိုပဲ နေရာချထားသင့်ပါတယ်။ mintute ထဲမှာ String တစ်ခုကို သိုလှောင်လို့မရသလို fred ထဲမှာ ကိန်းပြည့်တစ်ခုကို ထည့်လို့မရပါဘူး။

ဒါပေမယ့် ဒီစည်းမျဉ်းဟာ နည်းနည်းရှုပ်ပါတယ်။ ဘာလို့လည်းဆိုတော့ တန်ဖိုးတွေကို အမျိုးအစားတစ်ခုကနေ နောက်တစ်ခုကို ပြောင်းလဲပစ်နိုင်ပြီး Java ဟာ တစ်ခါတစ်ရံမှာ ဒီပြောင်းလဲမှုတွေကို အလိုအလျောက် လုပ်ပေးလို့ပါပဲ။ ဒီတော့ အခုအချိန်မှာ ယေဘုယျစည်းမျဉ်းကိုမှတ်ထားပြီး နောက်ပိုင်းကျမှ အထူးကိစ္စတွေ အကြောင်းပြောပါမယ်။

နဝေတိန်တောင်ဖြစ်စရာ နောက်ကိစ္စတစ်ခုကတော့ တချို့ string တွေဟာ ကိန်းပြည့်တွေနဲ့တူပေမယ့် တကယ်တမ်းတော့ မဟုတ်ပါဘူး။ ဥပမာ fred ဟာ အက္ခရာ 1, 2 နဲ့ 3 ပါဝင်တဲ့ string "123" ကို သယ်ဆောင်နိုင်ပေမယ့် ဒါဟာ 123 ဆိုတဲ့ကိန်းနဲ့ မတူပါဘူး။

fred = "123" // legal

fred = 123 // not legal

## Variable များကို Print လုပ်ခြင်း

Variable တစ်ခုရဲ့ တန်ဖိုးကို String တွေကို print လုပ်ဖို့ သုံးခဲ့တဲ့ command တွေသုံးပြီး print လုပ်နိုင်ပါတယ်။

```
class Hello {
    public static void main (String[] args) {
    firstLine = "Hello, agian!";
    System.out.println (first Line);
    }
}
```

ဒီပရိုဂရမ်ဟာ firstLine ဆိုတဲ့ variable တစ်ခုကို ဖန်တီးပြီး သူ့မှာ "Hello, again!" ဆိုတဲ့ တန်ဖိုးကို နေရာချထားပါတယ်။ ပြီးမှ အဲဒီတန်ဖိုးကို print လုပ်ပါတယ်။ Variable တစ်ခုကို print လုပ်တယ်လို့ပြောရင် အဲဒီ variable ရဲ့ တန်ဖိုးကို print လုပ်နေတယ်လို့မှတ်ပါ။ Variable ရဲ့ နာမည်ကို print လုပ်ချင်ရင် သူ့ကို inverted comma တွေထဲမှာ ထည့်ရပါမယ်။ ဥပမာ System.out.println ("first Line");

စွန့်စားမှု နည်းနည်းလောက်လိုချင်ရင် ဒီလိုရေးနိုင်ပါတယ်။

```
String firstLing;
first Line = "Hello, again!";
```

```
System.out.print("The value of firstLine is ");
System.out.println(firstLine);
```

ဒီပရိုဂရမ်ရဲ့ output ဟာ ဒီလိုဖြစ်ပါတယ်။

```
The value of firstLine is Hello, again!
```

Variable တစ်ခုကို print လုပ်တဲ့ သဒ္ဒါဟာ variable ရဲ့ အမျိုးအစားက ဘာပဲဖြစ်နေနေ အတူတူပါပဲ။

```
int hour, minute;
hour = 11;
minute = 59;
System.out.print ("The current time is ");
System.out.print (hour);
System.out.print (":");
System.out.print (minute);
System.out.println (".");
```

ဒီပရိုဂရမ်ရဲ့ output ဟာ အောက်မှာပြထားသလို ဖြစ်ပါတယ်။

```
The current time is 11:59
```

စာကြောင်းတစ်ကြောင်းမှာ တန်ဖိုးအများကြီးကို စီစဉ်ဖို့ print command တွေအောက်မှာ println သုံးလေ့ ရှိပါတယ်။ ဒါပေမယ့် အဆုံးမှာ println ကိုထည့်ဖို့ သတိရသင့်ပါတယ်။ ပတ်ဝန်းကျင် တော်တော်များများမှာ println command ကို နှိုးမဆွမချင်း print ရဲ့ output ကို သိမ်းထားပါတယ်။ နောက်ဆုံးမှ println ကိုတွေ့တဲ့အခါ အားလုံးကို တစ်ပြိုင်နက်တည်း ပြသပါတယ်။ println ကို ချန်ထားရင် ပရိုဂရမ်ဟာ သိမ်းထားတဲ့ output ကို မပြသဘဲ ထွက်သွားပါလိမ့်မယ်။

## Keyword များ

ပြီးခဲ့တဲ့ အပိုင်းတစ်ပိုင်းမှာ variable တွေအတွက် ကြိုက်ရာနာမည်သုံးနိုင်တယ်လို့ ဆိုခဲ့ပါတယ်။ ဒါပေမယ့် ဒါဟာသိပ်မမှန်ပါဘူး။ Java မှာ compiler က ပရိုဂရမ်တည်ဆောက်ပုံကို ခွဲခြားနိုင်ဖို့ ကြိုတင်ယူထားတဲ့ စကားလုံးတွေ ရှိပါတယ်။ ဒီစကားလုံးတွေကို keyword တွေလို့ ခေါ်ပြီး public, calss, void, int စတာတွေ ပါဝင်ပါတယ်။

စာရင်းအပြည့်အစုံကို Core Java စာအုပ်မှာ ဖော်ပြထားပါပြီ။ Sun Microsystems ရဲ့ Java website (java.sun.com) မှာလည်း ရနိုင်ပါတယ်။

ဒီစာရင်းကြီးကို အလွတ်မှတ်ထားမယ့်အစား Java ရေးသားမှုပတ်ဝန်းကျင် တော်တော်များများမှာ ပံ့ပိုးပေးထားတဲ့ လက္ခဏာတစ်ခုကို အခွင့်ကောင်းယူဖို့ တိုက်တွန်းလိုပါတယ်။ အဲ

ဒါကတော့ code ကို highlight လုပ်တာဖြစ်ပါတယ်။ စာစီတဲ့အချိန်မှာ ပရိုဂရမ်ရဲ့ မတူညီတဲ့အပိုင်းတွေဟာ အရောင်အမျိုးမျိုးနဲ့ ပေါ်လာပါလိမ့်မယ်။ ဥပမာ keyword တွေက အပြာ၊ string တွေက အနီ၊ တခြား code တွေက အနက်နဲ့ ပေါ်တတ်ပါတယ်။ Variable နာမည်တစ်ခုကိုရေးရင်း အပြာရောင်ပြောင်းသွားရင် သတိထားပါ။ Compiler ဆီက ထူးဆန်းတဲ့ အပြုအမူတွေကို ရရှိပါလိမ့်မယ်။

## Operator များ

Operator တွေဟာ အပေါင်းနဲ့အမြှောက်တို့လို ရိုးရှင်းတဲ့တွက်ချက်မှုတွေကို ကိုယ်စားပြုတဲ့ အထူးသင်္ကေတတွေ ဖြစ်ပါတယ်။ Java ရဲ့ operator အများစုဟာ ကိုယ်မျှော်လင့်ထားတဲ့ အလုပ်ကို လုပ်ကိုင်ပါတယ်။ အကြောင်းက သူတို့ဟာ သင်္ချာသင်္ကေတတွေ ဖြစ်နေလို့ပါ။ ဥပမာ ကိန်းပြည့်နှစ်ခုကိုပေါင်းတဲ့ operator ဟာ + ဖြစ်ပါတယ်။

အောက်မှာပြထားတဲ့ operator အားလုံးဟာ အနည်းနဲ့အများ မှန်းဆရလွယ်တဲ့ တရားဝင် Java ဖော်ပြချက်တွေ ဖြစ်ပါတယ်။

```
1+1 hour-1 hour*60+minute minute/60
```

ဖော်ပြချက် (expression) တွေမှာ variable နာမည်တွေနဲ့ကိန်းတွေ နှစ်မျိုးလုံး ပါဝင်နိုင်ပါတယ်။ ဒီနေရာမှာ variable ရဲ့နာမည်ကို တွက်ချက်မှုမလုပ်ခင် သူ့တန်ဖိုးနဲ့ အစားထိုးလိုက်ပါတယ်။

အပေါင်း၊ အနုတ်နဲ့ အမြှောက်ဟာ ကိုယ်မျှော်လင့်ထားတဲ့ အလုပ်ကို လုပ်ပါတယ်။ ဒါပေမယ့် အစားကိုတော့ အံ့သြရပါလိမ့်မယ်။ ဥပမာ အောက်မှာပြထားတဲ့ ပရိုဂရမ်

```
int hour, minute;
hour = 11;
minute = 59;
System.out.print ("Number of minutes since midnight: ");
System.out.println (hour*60+minute);
System.out.print ("Fraction of the hour that has passed: ");
System.out.println (minute/60);
```

ဟာ အခုပြထားတဲ့ output ကို ထုတ်ပေးပါလိမ့်မယ်။

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

ပထမဆုံးစာကြောင်းဟာ ကိုယ်မှန်းထားသလို ဖြစ်နေပြီး ဒုတိယစာကြောင်းတစ်ကြောင်းကတော့ ထူးဆန်းနေပါတယ်။ minute ဆိုတဲ့ variable ရဲ့ တန်ဖိုးဟာ 59 ဖြစ်ပြီး 59 ကို 60 နဲ့စားရင် 0.98333 ရပါတယ်။ 0 မဟုတ်ပါဘူး။ ဒီကွာဟချက်ရဲ့ အကြောင်းရင်းက Java ရဲ့

ကိန်းပြည့်စားခြင်းကို ဆောင်ရွက်နေလို့ပါပဲ။

Operator တွေရဲ့ တစ်ဘက်စီမှာရှိပြီး operator က လုပ်ဆောင်ချက်တွေပြုလုပ်တဲ့ variable တွေ၊ တန်ဖိုးတွေကို operand တွေလို့ခေါ်ပါတယ်။ Operand နှစ်ခုစလုံးဟာ ကိန်းပြည့်တွေဖြစ်ကြရင် ရလဒ်ဟာလည်း ကိန်းပြည့်တစ်ခုဖြစ်ရပါမယ်။ အစဉ်အလာအရ ကိန်းပြည့်စားခြင်းမှာ အနီးဆုံးယူတဲ့အခါ ဖြတ်ချခံရပါတယ်။ အခုဖြင့်ရတဲ့ကိစ္စမှာကို ပိုကြီးတဲ့ ကိန်းပြည့်နဲ့ အင်မတန်နီးတဲ့ အချိန်မှာတောင် ဖြတ်ချခံရပါတယ်။

ဒီနေရာမှာ နောက်တစ်မျိုးလုပ်နိုင်တာက ဒဿမကိန်းရှာမယ့်အစား ရာခိုင်နှုန်းကို တွက်ဖို့ပါပဲ။

```
System.out.print ("Percentage of the how that has passed: ");
System.out.println (minute*100/60);
```

ရလဒ်က ဒီလိုဖြစ်ပါတယ်။

```
Percentage of the how that has passed: 98
```

ဒီနေရာမှာလည်း ရလဒ်ကို ဖြတ်ချခံရပါတယ်။ ဒါပေမယ့် အနည်းဆုံးတော့ အခုအချိန်မှာ အဖြေဟာ အနီးဆုံးမှန်နေပါပြီ။ ပိုပြီးတိကျတဲ့ အဖြေကို လိုချင်တယ်ဆိုရင် floating-point လို့ခေါ်တဲ့ variable နောက်တစ်မျိုးကို သုံးရပါမယ်။ Floating-point ဟာ ဒဿမကိန်းတွေကို သယ်ဆောင်နိုင်ပြီး သူ့အကြောင်းကို နောက်မှဆက်ဆွေးနွေးပါမယ်။

## ၄ လုပ်ဆောင်ချက်အစီအစဉ်

ဖော်ပြချက်တစ်ခုထဲမှာ operator တစ်ခုထက်ပိုပါရင် တွက်ထုတ်တဲ့ အစီအစဉ်ဟာ ဦးစားပေးအစီအစဉ် (precedence) ဥပဒေသတွေပေါ်မှီခိုပါတယ်။ ဦးစားပေးအစီအစဉ်ကို ပြည့်ပြည့်စုံစုံလိုက်ရှင်းရင် ရှုပ်ထွေးမှုတွေပေါ်လာနိုင်ပါတယ်။ ဒါပေမယ့် အလုပ်စဖြစ်အောင် ပြောရရင်

- အမြှောက်အစားကို အပေါင်းအနုတ်ထက် အရင်လုပ်ပါတယ်။ ဒီတော့ 2*3-1 ဟာ 5 ကို ထုတ်ပေးပြီး 4 မရပါဘူး။ 2/3-1 ဟာ -1 ကိုရစေပြီး 1 မထွက်ပါဘူး။ (ကိန်းပြည့်အစား မှာ 2/3 ဟာ 0 ဖြစ်တယ်ဆိုတာ သတိရပါ။)
- အကယ်၍ operator တွေရဲ့ ဦးစားပေးအစီအစဉ်တွေ တူနေခဲ့ရင် သူတို့ကို ဘယ်ဘက်ကနေ ညာဘက်ကို တွက်ထုတ်ပါတယ်။ ဒီတော့ minute*100/60 ဆိုတဲ့ ဖော်ပြချက်မှာ အမြှောက်ဟာ အရင်ဖြစ်ပေါ်ပြီး ပထမဆုံး 5900/60 ကို ထုတ်ပါတယ်။ နောက်မှ 98 ကို ပေးပါတယ်။ အကယ်၍ လုပ်ဆောင်ချက်တွေဟာ ညာဘက်ကနေ ဘယ်ဘက်သွားနေခဲ့တယ်ဆိုရင် ရလဒ်ဟာ 59*1 ဖြစ်ပြီးနောက်ဆုံး 59 ကို ရတာကြောင့် မှားသွားပါလိမ့်မယ်။
- ဦးစားပေးအစီအစဉ်ကို ကျော်လုပ်ချင်ရင် (ဒါမှမဟုတ် ဦးစားပေး အစီအစဉ်က ဘယ်လိုဖြစ်တယ်ဆိုတာ ဇဝေဇဝါဖြစ်နေရင်) လက်သည်းကွင်းတွေကို သုံးနိုင်ပါ

တယ်။ လက်သည်းကွင်းထဲက ဖော်ပြချက်တွေကို အရင်တွက်ထုတ်ပါတယ်။ ဒီ တော့ 3*(3-1) ဟာ 4 ဖြစ်ပါတယ်။ လက်သည်းကွင်းတွေကိုသုံးပြီး ဖော်ပြချက် တစ်ခုကို ဖတ်ရလွယ်အောင် လုပ်နိုင်ပါတယ်။ ဥပမာ (minute*100)/60 ဟာ ရလဒ်ကို မပြောင်းလဲစေပေမယ့် အမြင်ကို ပိုရှင်းစေပါတယ်။

### စာသားများအတွက် Operator များ

ယေဘုယျအားဖြင့် စာသားတွေပေါ်မှာ သင်္ချာလုပ်ဆောင်ချက်တွေ ဆောင်ရွက်လို့ မရနိုင်ပါဘူး။ စာသားတွေက ကိန်းတွေလိုဖြစ်နေရင်တောင်မှ ဒီလိုလုပ်လို့မရပါဘူး။ အောက်မှာ ပြထားတာတွေဟာ တရားမဝင်ပါဘူး။

(fred ဟာ String type ရှိတယ်ဆိုတာ သိရင်ပေါ့)

```
fred-1 "Hello/123 fred*"Hello"
```

ဒါနဲ့ fred ဟာ ကိန်းပြည့်လား စာသားလားဆိုတာ ကြည့်ရုံနဲ့ပြောနိုင်သလား။ မပြော နိုင်ပါဘူး။ Variable တစ်ခုရဲ့ အမျိုးအစားကို ပြောနိုင်တဲ့ တစ်ခုတည်းသောနည်းက သူ့ကိုကြေ ညာတဲ့နေရာမှာ သွားကြည့်ဖို့ပါပဲ။

စိတ်ဝင်စားစရာကောင်းတာက + operator ဟာ စာသားတွေအပေါ်လည်း သက်ရောက် မှု ရှိပါတယ်။ ဒါပေမယ့် ကိုယ်မျှော်လင့်ထားတဲ့ အလုပ်ကိုတော့ မလုပ်ပါဘူး။ စာသားတွေအ တွက် + operator ဟာ စာသားပေါင်းစည်းခြင်းလို့ ဆိုနိုင်တဲ့ concatenation ကို လုပ်ပါတယ်။ ဆိုလိုတာက operator နှစ်ခုကို မျက်နှာချင်းဆိုင် ဆက်လိုက်တာပါ။ ဒီတော့ "Hello, " + "world." ဟာ "Hello, world" ဆိုတဲ့စာသားကို ရစေပြီး fred + "ism" ဟာ fred က ဘာဖြစ်ဖြစ် သူ့နောက်မှာ ism ကို လိုက်ထည့်ပေးပါတယ်။ အတွေးအခေါ်ဝါဒသစ် နာမည်တွေ တီထွင်ချင်ရင် ဒီနည်းက အသုံးဝင်စေပါလိမ့်မယ်။

### ဖွဲ့စည်းခြင်း

အခုအချိန်အထိ variable တွေ၊ expression ဖော်ပြချက်တွေနဲ့ statement ဖော်ပြချက် တွေစတဲ့ ပရိုဂရမ်မင်းဘာသာစကားတစ်ခုရဲ့ ပါဝင်မှုတွေကို သီးသန့်လေ့လာခဲ့ပြီးပါပြီ။ သူ တို့ကို ဘယ်လိုစုပေါင်းပြီး သုံးရမယ်ဆိုတာ မလေ့လာရသေးပါဘူး။

ပရိုဂရမ်မင်းဘာသာစကားတွေရဲ့ အသုံးအဝင်ဆုံး လက္ခဏာတစ်ရပ်ကတော့ သေးငယ် တဲ့ အပိုင်းတွေကိုယူပြီး ဖွဲ့စည်းနိုင်တဲ့ အရည်အချင်းပဲ ဖြစ်ပါတယ်။ ဒါကို အင်္ဂလိပ်လို compo-sition လို့ ခေါ်ပါတယ်။ ဥပမာအနေနဲ့ အခုအချိန်မှာ သိထားတာနှစ်ခုကို စမ်းကြည့်ပါမယ်။ မြှောက်နည်းကို သိပြီးသားဖြစ်သလို print လုပ်နည်းကိုလည်း သိပါတယ်။ ဖြစ်ချင်တော့ သူတို့ နှစ်မျိုးစလုံးကို တစ်ပြိုင်နက်တည်း လုပ်နိုင်ပါတယ်။

```
System.out.println (17*3);
```

တကယ်တမ်းကျတော့ တစ်ပြိုင်တည်းဆိုတဲ့ အသုံးအနှုန်းကို မသုံးခဲ့သင့်ပါဘူး။ ဘာလို့

လဲဆိုတော့ လက်တွေ့မှာ print မလုပ်ခင် အရင်မြှောက်ရလို့ပါပဲ။ ဒါပေမယ့် ဆိုလိုချင်တာက ကိန်းတွေ၊ စာသားတွေ၊ variable တွေပါတဲ့ ဘယ်ဖော်ပြချက်မဆို print ဖော်ပြချက်ထဲမှာ ထည့်သုံးနိုင်ပါတယ်။ ဥပမာတစ်ခုကို အရင်တုန်းက မြင်ခဲ့ပြီးပါပြီ။

```
System.out.println(hour*60+minute);
```

ဒါပေမယ့် နေရာချထားတဲ့ ဖော်ပြချက်ရဲ့ ညာဖက်မှာ တန်ဖိုးပြောင်းနိုင်တဲ့ ဖော်ပြချက်တွေကိုလည်း ထည့်နိုင်ပါတယ်။

```
int percentage;
percentage = (minute*100)/60;
```

ဒီအရည်အချင်းဟာ အခုချိန်မှာ သိပ်ပြီးအထင်ကြီးလောက်စရာ မဟုတ်သလို ဖြစ်ချင်ဖြစ်နေပါမယ်။ ဒါပေမယ့် ရှုပ်ထွေးတဲ့ ဖော်ပြချက်တွေကို ဖွဲ့စည်းမှုက သေသေသပ်သပ်၊ တိုတိုတုတ်တုတ် ဘယ်လိုဖော်ပြနိုင်သလဲဆိုတာ မြင်ရဦးမှာပါ။

သတိပေးချင်တာတစ်ခုက တချို့ဖော်ပြချက်တွေကို ဘယ်နေရာမှာ သုံးရမလဲဆိုတာနဲ့ ပတ်သက်ပြီး အကန့်အသတ်တွေရှိပါတယ်။ ပြောပလောက်တာတစ်ခုက နေရာချထားတဲ့ ဖော်ပြချက်တစ်ခုရဲ့ ဘယ်ဖက်ခြမ်းဟာ variable နာမည်တစ်ခု ဖြစ်ရမှာဖြစ်ပြီး expression ဖော်ပြချက်တစ်ခုဖြစ်လို့ မရပါဘူး။ ဒါဟာ ဘာလို့လဲဆိုတော့ ဘယ်ဖက်ခြမ်းဟာ ရလဒ်ကို သိမ်းဆည်းမယ့် မှတ်ဉာဏ်နေရာကို ကိုယ်စားပြုတာကြောင့်ပါ။ ဖော်ပြချက်တွေဟာ သိမ်းဆည်းမှုနေရာတွေကို ကိုယ်စားမပြုပါဘူး။ တန်ဖိုးတွေကိုပဲ ဖော်ပြပါတယ်။ ဒီတော့ `minute + 1 = hour;` ဟာ မှားပါတယ်။

အခန်း ၃

# Method များ

## Floating-point

ပြီးခဲ့တဲ့အခန်းမှာ ကိန်းပြည့်မဟုတ်တဲ့ကိန်းတွေနဲ့ အလုပ်လုပ်ရာမှာ ပြဿနာတွေ တွေ့ခဲ့ပါတယ်။ အပိုင်းကိန်းတွေအနေနဲ့ တွက်ချက်မယ့်အစား ရာခိုင်နှုန်းတွေသုံးပြီး ဒီပြဿနာကို ရှောင်ရှားခဲ့ပါတယ်။ ပိုပြီး ယေဘုယျကျတဲ့ ဖြေရှင်းနည်းကတော့ ကိန်းပြည့်တွေအပြင် ဒသမ အပိုင်းကိန်းတွေကိုပါ ကိုယ်စားပြုနိုင်တဲ့ floating-point ကိန်းတွေကို သုံးဖို့ပါပဲ။ Java မှာ floating-point အမျိုးအစားကို double လို့ခေါ်ပါတယ်။

တခြားအမျိုးအစားတွေအတွက် သုံးခဲ့တဲ့သဒ္ဒါကို အသုံးပြုပြီး floating-point variable တွေ ကြေညာတဲ့အလုပ်နဲ့ သူတို့ဆီမှာ တန်ဖိုးတွေ နေရာချထားတဲ့အလုပ်ကို လုပ်နိုင်ပါတယ်။ ဥပမာ

```
double pi;
pi = 3.14159;
```

Variable ကို ကြေညာပြီး တစ်ချိန်တည်းမှာပဲ တန်ဖိုးတစ်ခုကို နေရာချထားနိုင်ပါတယ်။

```
int x = 1;
String empty = "";
```

```
double pi = 3.14159;
```

တကယ်ဆိုရင် ဒီသင်္ကေတကို တော်တော်တွင်တွင်ကျယ်ကျယ်သုံးပါတယ်။ ကြေညာချက်နဲ့ နေရာချထားတာကို စတင်ခြင်း (initialisation) လို့ ခေါ်ပါတယ်။

Floating-point ကိန်းတွေဟာ အသုံးဝင်ပေမယ့် တစ်ခါတစ်ရံမှာ သူတို့ဟာ ရှုပ်ထွေးမှု ရဲ့ ဇစ်မြစ်ဖြစ်တတ်ပါတယ်။ ကိန်းပြည့်တွေနဲ့ floating-point တွေကြားမှာ ထပ်နေတာတွေ ရှိနေလို့ပါ။ ဥပမာ 1 ဆိုတဲ့တန်ဖိုးရှိရင် သူဟာ ကိန်းပြည့်လား၊ Floating-point လား နှစ်မျိုး စလုံးလား။

တိတိကျကျပြောရရင် Java ဟာ ကိန်းပြည့်တန်ဖိုး 1 ကို floating-point တန်ဖိုး 1.0 နဲ့ ခွဲခြားပါတယ်။ သူတို့ဟာ ကိန်းတစ်ခုတည်းလို့ ထင်ရရင်တောင်မှ ဒီလိုခွဲခြားထားပါတယ်။ သူတို့ဟာ မတူညီတဲ့ အမျိုးအစားတွေနဲ့ သက်ဆိုင်နေပြီး အမျိုးအစားမတူတဲ့ နေရာချထားမှုတွေကို မပြု လုပ်သင့်ပါဘူး။ ဥပမာ အောက်မှာပြထားတဲ့ code ဟာ မှားနေပါတယ်။

```
int x = 1.1;
```

ဘာလို့လဲဆိုတော့ ဘယ်ဖက်က variable ဟာ int ဖြစ်ပြီး ညာဖက်ကတန်ဖိုးက double ဖြစ်နေပါတယ်။ ဒါပေမယ့် Java က အမျိုးအစားတွေ တစ်ခုကနေတစ်ခု အလိုအလျောက်ပြောင်း ပေးနိုင်တဲ့အတွက် ဒီစည်းမျဉ်းကို မေ့ရလွယ်ပါတယ်။ ဥပမာ

```
double y = 1;
```

ဒါဟာ တရားမဝင်ပါဘူး။ ဒါပေမယ့် Java က သူ့ကို int တစ်ခုကနေ double အဖြစ် အလိုအလျောက် ပြောင်းလဲပါတယ်။ ဒီလိုလျော့ပေါ့စဉ်းစားတာ အဆင်တော့ပြေစေပါရဲ့။ ဒါ ပေမယ့် ပြဿနာတွေကို ဖိတ်ခေါ်နိုင်ပါတယ်။ ဥပမာ

```
double y = 1/3;
```

Variable y ကို 0.333333 ဆိုတဲ့ တန်ဖိုးပေးမယ်လို့ ယူဆကောင်း ယူဆကြပါမယ်။ ဘာလို့လဲဆိုတော့ သူဟာ တရားဝင် floating-point တန်ဖိုးတစ်ခု ဖြစ်လို့ပါပဲ။ ဒါပေမယ့် လက်တွေ့မှာ 0.0 တန်ဖိုးကို နေရာချထားပါတယ်။ အကြောင်းကတော့ ညာဖက်က ဖော်ပြ ချက်ဟာ ကိန်းပြည့်နှစ်ခုရဲ့ အချိုးဖြစ်နေပါတယ်။ ဒီတော့ Java ဟာ ကိန်းပြည့်စားခြင်းကို ပြု လုပ်ပါတယ်။ ကိန်းပြည့်တန်ဖိုး 0 ကို ရပါတယ်။ Floating တန်ဖိုးအဖြစ် ပြောင်းတဲ့အခါ ရလဒ်ဟာ 0.0 ဖြစ်သွားပါတယ်။

ဒီပြဿနာကို ဖြေရှင်းနိုင်တဲ့ နည်းတစ်နည်းက ညာဖက်အခြမ်းကို floating-point ဖော် ပြချက်တစ်ခုအဖြစ် ပြောင်းဖို့ပါပဲ။

```
double y = 10 / 3.0;
```

ဒီလိုလုပ်လိုက်ခြင်းဖြင့် မျှော်လင့်ထားသလိုပဲ y ကို 0.333333 အဖြစ်ထားလိုက်ပါတယ်။

အပေါင်း၊ အနုတ်၊ အမြှောက်၊ အစား စတဲ့ လုပ်ဆောင်ချက်တွေအားလုံးကို floating-point တွေနဲ့လည်း ပြုလုပ်နိုင်ပါတယ်။ ဒါပေမယ့် အောက်ခြေက ယန္တရားကတော့ အင်မတန် ကွာခြားပါတယ်။ တကယ်တမ်းမှာတော့ ပရိုဆက်ဆာတော်တော်များများမှာ floating-point

လုပ်ဆောင်ချက်တွေ ဆောင်ရွက်ဖို့သက်သက် အထူးအစိတ်အပိုင်းတွေ ပါရှိတတ်ပါတယ်။

## double မှ int သို့ပြောင်းခြင်း

အရင်ကပြောခဲ့သလိုပဲ Java ဟာ int တွေကို double တွေအဖြစ် လိုအပ်လာရင် အလိုအလျောက် ပြောင်းပေးပါတယ်။ ဘာလို့လဲဆိုတော့ ဒီအပြောင်းအလဲမှာ အချက်အလက်ဆုံးရှုံးသွားတာ မရှိလို့ပါပဲ။ အခြားတစ်ဖက်မှာတော့ double တစ်ခုကနေ int တစ်ခုအဖြစ်ပြောင်းရင် အနီးဆုံးကိန်းကို ယူရပါတယ်။ Java ဟာ ဒီလုပ်ဆောင်ချက်ကို အလိုအလျောက်မလုပ်ပေးပါဘူး။ ရည်ရွယ်ချက်က ပရိုဂရမ်မာအနေနဲ့ ကိန်းရဲ့ပျောက်ဆုံးသွားတဲ့ ဒဿမအပိုင်းကိန်းကို သတိမူစေချင်လို့ပါ။

Floating-point တန်ဖိုးတစ်ခုကို ကိန်းပြည့်တစ်ခုအဖြစ် ပြောင်းနိုင်တဲ့ အလွယ်ဆုံးနည်းကတော့ typecast တစ်ခုကို သုံးဖို့ဖြစ်ပါတယ်။ ဒီလိုခေါ်ရခြင်းအကြောင်းက အမျိုးအစားတစ်ခုရှိတဲ့ တန်ဖိုးတစ်ခုကိုယူပြီး တခြားအမျိုးအစား (type) တစ်ခုအဖြစ် ပုံစံသွင်း (cast) ရလို့ပါပဲ။

ကံမကောင်းစွာနဲ့ပဲ typecast လုပ်တဲ့ သဒ္ဒါဟာ အကျည်းတန်ပါတယ်။ ပြောင်းချင်တဲ့ အမျိုးအစားကို လက်သည်းကွင်းထဲထည့်ပြီး operator တစ်ခုအနေနဲ့ သုံးပါတယ်။ ဥပမာ

```
int x = (int) Math.PI;
```

(int) operator ဟာ သူ့နောက်ကလိုက်တဲ့ကိန်းကို ကိန်းပြည့်တစ်ခုအဖြစ် ပြောင်းပေးနိုင်တဲ့ အစွမ်းရှိပါတယ်။ ဒီတော့ အောက်မှာပြထားတဲ့ ဥပမာမှာ PI ရဲ့ တန်ဖိုးကို ကိန်းပြည့်တစ်ခုအဖြစ် အရင်ပြောင်းလိုက်ပါတယ်။ ရလဒ်ဟာ 60 ဖြစ်ပြီး 62 မဟုတ်ပါဘူး။

```
int x = (int) Math.PI * 20.0;
```

ကိန်းပြည့်ပြောင်းတဲ့အခါ အမြဲတမ်းဖြတ်ချခံရပါတယ်။ ဒဿမအပိုင်းဟာ 0.99999999 ဖြစ်နေရင်တောင်မှ ဒီလိုအလုပ်ခံရပါတယ်။

ဦးစားပေးအစီအစဉ်နဲ့ အနီးစပ်ဆုံးယူတာဟာ typecast လုပ်တာကို ခက်ခဲစေနိုင်ပါတယ်။

## သင်္ချာ Method များ

သင်္ချာမှာ sin နဲ့ log လို function တွေကို တွေ့ဖူးပါလိမ့်မယ်။ sin ($\pi/2$) နဲ့ log ($1/x$) လို ဖော်ပြချက်တွေကို တွက်ထုတ်နည်းကိုလည်း လေ့လာဖူးပါလိမ့်မယ်။ အရင်ဆုံးအနေနဲ့ လက်သည်းကွင်းထဲက ဖော်ပြချက်ကို အရင်တွက်ထုတ်ရပါတယ်။ သူ့ကို function ရဲ့ argument လို့ ခေါ်ပါတယ်။ ဥပမာ - $\pi/2$ ဟာ 1.52 နဲ့ အနီးစပ်ဆုံးတူပြီး x ဟာ 10 ဖြစ်ရင် $1/x$ ဟာ 0.1 ဖြစ်ပါတယ်။

အဲဒီနောက်မှာ function ကိုယ်တိုင်ကိုပဲ တွက်နိုင်ပါတယ်။ ဒီလိုတွက်ရာမှာ ဇယားတစ်ခုကိုကြည့်ပြီး တန်ဖိုးရှာနိုင်သလို တွက်ချက်မှုတွေ အဆင့်ဆင့်လုပ်ပြီးလည်း အဖြေရနိုင်ပါတယ်။ 1.52 ရဲ့ sin ဟာ 1 ဖြစ်ပြီး 0.1 ရဲ့ log ဟာ -1 ဖြစ်ပါတယ်။ (Logarithm ရဲ့ အခြေဟာ 10 ဖြစ်

တယ်ဆိုရင်ပေါ့။)

Java ဟာ ကိုယ်တွေ့လိုရနိုင်သမျှ သင်္ချာလုပ်ဆောင်ချက် အားလုံးနီးပါးကို ထည့်သွင်းတည်ဆောက်ထားတဲ့ function တွေအနေနဲ့ ပံ့ပိုးပေးထားပါတယ်။ ဒီ function တွေကို method တွေလို့ ခေါ်ပါတယ်။ သင်္ချာ method တော်တော်များများဟာ double တွေပေါ်မှာ ဆောင်ရွက်ပါတယ်။

သင်္ချာ method တွေကို အရင်ကမြင်ခဲ့တဲ့ print method တွေကို နိူးဆွရပါတယ်၊

```
double root = Math.sqrt(1.0);
double angle = 1.5;
double height = Math.sin (angle);
```

ပထမဆုံးဥပမာဟာ root ကို 17.0 ရဲ့ နှစ်ထပ်ကိန်းရင်း (square root) နဲ့ နေရာချထားလိုက်ပါတယ်။ ဒုတိယဥပမာက angle ဆိုတဲ့ variable တန်ဖိုး 1.5 ကို sine ရှာပါတယ်။ Java က sin ကို အခြား ထရီဂို function တွေဖြစ်ကြတဲ့ cos တို့၊ tan တို့နဲ့ တွဲသုံးမယ့်တန်ဖိုးတွေဟာ radian ဖြစ်တယ်လို့ ယူဆပါတယ်။ ဒီဂရီအတိုင်းအတာကနေ radian အဖြစ်ပြောင်းဖို့ 360 နဲ့ စားပြီး 2π နဲ့ မြှောက်ရပါတယ်။ အဆင်ပြေတာက Java မှာ π တန်ဖိုးကို ထည့်သွင်းတည်ဆောက်ထားပါတယ်။

```
double degrees = 90;
double angle = degrees * 2 * Math.PI / 360.0;
```

PI ဟာ အက္ခရာအကြီးတွေနဲ့ပဲ ရှိနေတယ်ဆိုတာ သတိပြုပါ။ Java က Pi တို့၊ pi တို့၊ pie တို့ကို အသိပါဘူး။

Math ရဲ့ တခြားအသုံးဝင်တဲ့ method ကတော့ round ပဲ ဖြစ်ပါတယ်။ သူက floating-point တန်ဖိုးတစ်ခုကို အနီးဆုံးကိန်းပြည့်အဖြစ် အနီးဆုံးယူပြီး int အနေနဲ့ ထုတ်ပေးပါတယ်၊

```
int x = Math.round (Math.PI * 20.0);
```

ဒီနေရာမှာ method ကို မနိူးဆွခင် အမြှောက်ကို အရင်လုပ်ပါတယ်။ ရလဒ်ဟာ 62.8319 ကို အနီးဆုံးယူထားတဲ့ 63 ဖြစ်ပါတယ်။

## ဖွဲ့စည်းခြင်း

သင်္ချာ function တွေမှာလိုပဲ Java method တွေကို ဆင့်ကဲဖွဲ့စည်းနိုင်ပါတယ်။ ဖော်ပြချက်တစ်ခုကို နောက်ဖော်ပြချက်တစ်ခုထဲ ထည့်သုံးတယ်လို့ အဓိပ္ပာယ်ရပါတယ်။ ဥပမာ ကြိုက်ရာ ဖော်ပြချက်ကို method တစ်ခုရဲ့ argument တစ်ခုအနေနဲ့ သုံးနိုင်ပါတယ်။

```
double x = Math.cos(angle + Math.PI/2);
```

ဒီဖော်ပြချက်ဟာ Math.PI ရဲ့ တန်ဖိုးကိုယူပြီး နှစ်နဲ့စားပါတယ်။ ရလဒ်ကို angle ဆိုတဲ့ variable မှာ ပေါင်းထည့်ပါတယ်။ ပေါင်းလဒ်ကို cos method ရဲ့ argument တစ်ခုအနေနဲ့ ပေးလိုက်ပါတယ်။ (PI ဟာ variable တစ်ခုရဲ့ နာမည်ဖြစ်ပါတယ်။ Method မဟုတ်ပါဘူး

ဒီတော့ argument ဆိုလို့ argument အလွတ်တစ်ခုတောင် မပါပါဘူး။)

Method တစ်ခုရဲ့ ရလဒ်ကိုယူပြီး နောက် method တစ်ခုရဲ့ argument အနေနဲ့လည်း ပို့နိုင်ပါတယ်။

```
double x = Math.exp (Math.log(10.0);
```

Java မှာ log function ဟာ e ကို အခြေအမြဲတမ်းယူပါတယ်။ ဒီတော့ အခုဖော်ပြချက်ဟာ 10 ရဲ့ အခြေ e ရှိတဲ့ log ရှာပြီး အဲဒီတန်ဖိုးကို e မှာ ထပ်ကိန်းတင်ပါတယ်၊ ရလဒ်ကို x မှာ နေရာချထားပါတယ်။ စာဖတ်သူအနေနဲ့ အထက်တန်းသင်္ချာ ကြေညက်ခဲ့ပြီး ဒါကို နားလည်မယ်လို့ မျှော်လင့်ပါတယ်။

## Method အသစ်များ ပေါင်းထည့်ခြင်း

အခုထက်ထိတော့ Java မှာ ထည့်သွင်းတည်ဆောက်ထားတဲ့ method တွေကိုပဲ သုံးနေခဲ့ပါတယ်။ ဒါပေမယ့် method အသစ်တွေကို ပေါင်းထည့်ချင်ရင်လည်း ဖြစ်နိုင်ပါတယ်။ တကယ်တမ်းကြတော့ method အသစ်တစ်ခုသတ်မှတ်တာကို မြင်ခဲ့ပြီးပါပြီ။ သူ့နာမည်က main တဲ့။ ဒီလိုနာမည်ပေးထားတဲ့ method ဟာ ပရိုဂရမ်ကို စတင်ဆောင်ရွက်မယ့် နေရာကို ဖော်ညွှန်းပါတယ်။ ဒါပေမယ့် main ကိုရေးတဲ့ သဒ္ဒါဟာ တခြား method သတ်မှတ်ချက်တွေနဲ့ အတူတူပဲ ဖြစ်ပါတယ်။

```
public static void NAME(LIST OF PARAMETERS){
   STATEMENTS
}
```

Method အတွက် ကြိုက်တဲ့နာမည် ပေးနိုင်ပါတယ်။ ဒါပေမယ့် main တို့ တခြား key-word နာမည်တို့တော့ ခေါ်လို့မရပါဘူး။ Parameter စာရင်းဟာ function အသစ်ကိုသုံးဖို့ ဘယ်လိုသတင်းအချက်အလက်တွေ ပံ့ပိုးရမယ်ဆိုတာ ညွှန်ကြားပါတယ်။

main method အတွက် တစ်ခုတည်းသော parameter စာ String[]args ဖြစ်ပါတယ်။ ဆိုလိုတာက main ကို နှိုးဆွသူ ဘယ်သူမဆို String တွေရဲ့ array တစ်ခုကို ပံ့ပိုးပေးရပါမယ်။ အခုရေးမယ့် method နှစ်ခုမှာ parameter တွေ မပါပါဘူး။ ဒီတော့ သဒ္ဒါက ဒီလိုပုံကို ရှာပါတယ်။

```
public static void newLine() {
   System.out.println(" ");
}
```

ဒီ method ကို newLine လို့ နာမည်ပေးထားပါတယ်။ လက်သည်းကွင်းလွတ်တွေက သူတို့ဟာ parameter တွေ မယူဘူးလို့ ဆိုလိုပါတယ်။ သူ့မှာ " " နဲ့ ပြောထားတဲ့ စာသားအလွတ်တစ်ခုကို print လုပ်ပေးတဲ့ ဖော်ပြချက်တစ်ခု ပါဝင်ပါတယ်။ စာသားတစ်ခုကို ဘာမှမထည့်ဘဲ print လုပ်တာဟာ အသုံးဝင်တယ်လို့ ထင်စရာမရှိပါဘူး။ ဒါပေမယ့် println ဟာ print

လုပ်ပိုင်ရင် နောက်တစ်ကြောင်း ဆင်းသွားစေပါတယ်။ ဒါကြောင့် [illegible]
တစ်ကြောင်းကို ဆင်းသွားစေတဲ့ [illegible] ရှိပါတယ်။

main မှာ ဒီ method အသစ်ကို [illegible] Java command [illegible]
ကို [illegible] ခေါ်နိုင်ပါတယ်။

```
public static void main(String[] args) {
   System.out.println("First line.");
   newLine();
   System.out.println("Second line.");
}
```

ဒီပရိုဂရမ်ရဲ့ output ဟာ ဒီလိုဖြစ်ပါတယ်။

```
First line.
Second line.
```

စာကြောင်းနှစ်ကြောင်းကြားက ကွက်လပ်အလွတ်ကို သတိပြုပါ။ စာကြောင်းတွေကြား ကွက်လပ်ပိုလိုချင်ရင် ဘယ်လိုလုပ်ရမလဲ။ Method တစ်ခုတည်းကိုပဲ ထပ်ခါထပ်ခါ ခေါ်ယူနိုင်ပါတယ်။

```
public static void main(String[] args) {
   System.out.println("First line.");
   newLine();
   newLine();
   newLine();
   System.out.println("Second line.");
}
```

ဒါမှမဟုတ် စာကြောင်းအလွတ်သုံးကြောင်းကို print လုပ်ပေးတဲ့ threeLine method တစ်ခုကို ရေးနိုင်ပါတယ်။

```
public static void threeLine(){
   newLine(); newLine(); newLine();
}

public static void main(String[] args){
   System.out.println("First line.");
   threeLine();
   System.out.println("Second line.");
}
```

ဒီပရိုဂရမ်နဲ့ပတ်သက်ပြီး သတိထားစရာအချက်တွေ ရှိပါတယ်။

- လုပ်ဆောင်ချက်တစ်ခုကို အကြိမ်ကြိမ်နှိုးဆွနိုင်ပါတယ်။ တကယ်တမ်းဆိုရင် ဒီလိုလုပ်တာဟာ အလေ့အထလိုဖြစ်နေပြီး အသုံးလည်းဝင်ပါတယ်။
- Method တစ်ခုက နောက် method တစ်ခုကို နှိုးဆွနိုင်ပါတယ်။ ဒီနေရာမှာ main က threeLine ကို နှိုးဆွပြီး threeLine က newLine ကို နှိုးဆွပါတယ်။ ဒီတစ်ကြိမ်မှာလည်း ဒါဟာ အသုံးဝင်ပြန်ပါတယ်။
- threeLine မှာ ဖော်ပြချက်တွေအားလုံးကို စာကြောင်းတစ်ကြောင်းထဲမှာ ရေးခဲ့ပါတယ်။ ဒါဟာ သဒ္ဒါအရ မှန်ပါတယ်။ ကွက်လပ်တွေနဲ့ စာကြောင်းခွဲတာဟာ ပရိုဂရမ်ရဲ့ အဓိပ္ပာယ်ကို မပြောင်းလဲစေဘူးဆိုတာ အမှတ်ရပါ။ အခြားတစ်ဖက်မှာတော့ ဖော်ပြချက်တစ်ခုစီကို သူ့စာကြောင်းနှင့်သူထားတာ ပိုကောင်းပါတယ်။ ဒါမှ ပရိုဂရမ်ကို ဖတ်ရလွယ်မှာပါ။

အခုအချိန်ထိတော့ method အသစ်တွေကို ဖန်တီးရတာ ဘာများအကျိုးရှိသလဲလို့ သဘောပေါက်ချင်မှ ပေါက်ပါမယ်။ ဒီလိုလုပ်ရခြင်းအကြောင်းတွေ အများကြီး ရှိပါတယ်။ ဒါပေမယ့် ဒီဥပမာက အကြောင်းရင်းနှစ်ခုကို သရုပ်ဖော်ပါတယ်။

၁။ Method ရေးသားခြင်းအားဖြင့် ဖော်ပြချက်အုပ်စုတစ်စုကို နာမည်ပေးခွင့်ရသွားပါတယ်။ Method တွေဟာ command တစ်ခုရဲ့နောက်မှာ ရှုပ်ထွေးတဲ့တွက်ချက်မှုကို ကွယ်ဝှက်ထားခြင်းအားဖြင့် ပရိုဂရမ်တစ်ပုဒ်ကို ရိုးရှင်းစေပါတယ်။ နားလည်ရခက်တဲ့ code နေရာမှာ အင်္ဂလိပ်နာမည်တစ်ခုကို ရေးနိုင်တဲ့အတွက်လည်း ပိုရှင်းစေပါတယ်။ newLine နဲ့ System.out.println(" ") ဘယ်ဟာပိုရှင်းလဲ။

၂။ Method အသစ်တစ်ခုကို ရေးသားခြင်းအားဖြင့် ထပ်တလဲလဲပါနေတဲ့ code တွေ ရေးရသက်သာစေပြီး ပရိုဂရမ်ကို သေးငယ်စေပါတယ်။ ဥပမာ စာကြောင်းလွတ်ကိုးကြောင်းကို ဘယ်လို print လုပ်မလဲ။ threeLine ကို သုံးကြိမ်နှိုးဆွရင် ရနိုင်ပါတယ်။

## Class များနှင့် Method များ

ပြီးခဲ့တဲ့အပိုင်းက code အပိုင်းအစအားလုံးကို ပြန်ပြီးစုစည်းလိုက်ရင် class အဓိပ္ပာယ်သတ်မှတ်ချက်အပြည့်အစုံဟာ ဒီလိုပုံစံနဲ့ တူပါတယ်။

```
class NewLine {
    public static void newLine() {
        System.out.println(" ");
    }
    public static void threeLine() {
        newLine(); newLine(); newLine();
```

```
    }
    public static void main(String[] args) {
        System.out.println("First line.");
        threeLine();
        System.out.println("Second line.");
    }
}
```

ပထမစာကြောင်းက ဒါဟာ NewLine လို့ခေါ်တဲ့ class အသစ်တစ်ခုအတွက် class အဓိပ္ပာယ်ဖွင့်ဆိုချက် ဖြစ်တယ်လို့ ဆိုလိုပါတယ်။ Class တစ်ခုဟာ အချင်းချင်းပတ်သက်မှုရှိတဲ့ method တွေရဲ့ အစုအဝေးဖြစ်ပါတယ်။ ဒီနေရာမှာ NewLine ဆိုတဲ့ class မှာ newLine, threeLine နဲ့ main ဆိုတဲ့ method သုံးခု ပါဝင်ပါတယ်။

အရင်ကမြင်ခဲ့တဲ့ နောက်ထပ် class တစ်ခုကတော့ Math class ဖြစ်ပါတယ်။ သူ့မှာ sqrt, sin စတဲ့ method တွေ ပါဝင်ပါတယ်။ သင်္ချာ function တွေကို နှိုးဆွတဲ့အခါ Math ဆိုတဲ့ class နာမည်ကို ဖော်ပြရပါတယ်။ ပြီးရင် function နာမည်ကို ဖော်ပြရပါတယ်။ ဒါကြောင့်မို့လို့ ဒီသဒ္ဒါဟာ ထည့်သွင်းတည်ဆောက်ထားတဲ့ method တွေနဲ့ ကိုယ်တိုင်ရေးထားတဲ့ method တွေကြားမှာ ကွာဟမှုတွေ ရှိနေတာဖြစ်ပါတယ်။

```
Math.pow(2.0, 10.0);
newLine();
```

ပထမဆုံးဖော်ပြချက်ဟာ Math class ရဲ့ pow method ကို နှိုးဆွပြီး ပထမ argument ကို ဒုတိယ argument တန်ဖိုးရဲ့ ထပ်ညွှန်းနဲ့ တင်လိုက်ပါတယ်။ ဒုတိယဖော်ပြချက်ဟာ newLine method ကို နှိုးဆွပါတယ်။ Java သူ့ကို ကျွန်တော်တို့ရေးနေတဲ့ NewLine class ထဲမှာရှိတယ်လို့ ယူဆလိုက်ပါတယ်။

Method တစ်ခုကို မမှန်ကန်တဲ့ class တစ်ခုကနေ နှိုးဆွရင် compiler ဟာ အမှား တစ်ခုကို ထုတ်လုပ်ပါလိမ့်မယ်။ ဥပမာ အောက်မှာပြထားတာမျိုး လုပ်ကြတယ်ဆိုပါစို့။

```
pow(2.0, 10.0);
```

Compiler ဟာ "NewLine class ထဲမှာ pow ဆိုတဲ့ method ကို ရှာမတွေ့ဘူး" ဆိုတာမျိုးကို ပြောလာပါလိမ့်မယ်။ အရင်က ဒီသတိပေးစကားကို တွေ့ခဲ့ဖူးတယ်ဆိုရင် ဘာလို့ သူက ကိုယ့် class သတ်မှတ်ချက်ထဲမှာ pow ကို လိုက်ရှာနေရတာကိုလည်း တွေးဖူးပါလိမ့်မယ်။ အခုတော့ သိသွားပါတယ်။

## Method များစွာပါသော ပရိုဂရမ်များ

Method တွေ အများကြီးပါတဲ့ class သတ်မှတ်ချက်တစ်ခုကိုကြည့်ရင် အစကနေအဆုံး ကို အလျင်စလိုဖတ်ဖို့ တွေးမိတတ်ကြပါတယ်။ ဒါပေမယ့် ဒီလိုလုပ်ရင် ခေါင်းရှုပ်သွားနိုင်ပါ

တယ်။ ဘာလို့လဲဆိုတော့ ဒါဟာ ပရိုဂရမ် run တဲ့ အစီအစဉ် မဟုတ်လို့ပါပဲ။

Run တဲ့ ဖြစ်စဉ်ဟာ main ရဲ့ ပထမဆုံးဖော်ပြချက်ကနေ စတင်ပါတယ်။ main က ဘယ်နေရာမှရှိရှိ ဒီလိုပဲ စတင်ပါတယ်။ ဖော်ပြချက်တွေကို တစ်ချိန်မှာ တစ်ခုစီ run ပါတယ်။ Method ခေါ်ဆိုမှုတစ်ခုထိ မရောက်မချင်း ဒါကိုလုပ်ပါတယ်။ Method ခေါ်ဆိုမှုတွေဟာ ပရိုဂရမ်တစ်ပုဒ်ရဲ့ လမ်းလွှဲတွေနဲ့ တူပါတယ်။ ပရိုဂရမ်စီးဆင်းမှုကို လမ်းကြောင်းပေးပါတယ် နောက်မှာလာတဲ့ ဖော်ပြချက်ဆီသွားမယ့်အစား ခေါ်ယူခံရတဲ့ method ရဲ့ ပထမဆုံးအကြောင်းဆီကို သွားပါတယ်။ အဲဒီမှာ code အားလုံးကို run ပါတယ်။ ပြီးမှ ထွက်သွားတဲ့နေရာကို ပြန်လာပြီး ကျန်တာဆက်လုပ်ပါတယ်။

ဒီလိုကြားလိုက်ရတော့ ရှင်းနေသလိုပဲ။ ဒါပေမယ့် method တစ်ခုဟာ နောက်တစ်ခုကို နိုးဆွနိုင်တယ်ဆိုတာကိုတော့ သတိရပါ။ ဒီတော့ main ရဲ့အလယ်မှာ ရောက်နေတုန်း လုပ်နေတာကိုရပ်ပြီး threeLine ထဲက ဖော်ပြချက်တွေကို run ဖို့ လိုကောင်းလိုပါလိမ့်မယ်။ ဒါပေမယ့် threeLine ကို run နေတုန်း newLine ကို သုံးကြိမ်သွား run ဖို့ လိုကောင်းလိုပါလိမ့်မယ်။

ဒီနေရာမှာ newLine ဟာ println ဆိုတဲ့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ method ကို နိုးဆွပါတယ်။ ဒါဟာ လမ်းကြောင်းပြောင်းမှုကို ထပ်ပြီးဖြစ်ပေါ်စေပါတယ်။ ကံကောင်းတာက Java ဟာ သူ ဘယ်မှာရောက်နေသလဲလို့ လမ်းကြောင်းလိုက်တဲ့ နေရာမှာ အင်မတန်တော်ပြီး println ပြီးချိန်မှာ newLine မှာ ထားခဲ့တဲ့နေရာကနေ ပြန်စပါတယ်။ ပြီးမှ threeLine ဆီ ပြန်သွားပါတယ်။ ပြီးမှ main ဆီ ပြန်ရောက်သွားပြီး ပရိုဂရမ်ကို ပြီးဆုံးခွင့်ပေးပါတယ်။

တကယ်တမ်းဆိုရင် နည်းပညာရှုထောင့်မှာ ပရိုဂရမ်ဟာ main ရဲ့အဆုံးမှာ မရပ်တန့်ပါဘူး။ အဲဒီအစား main ကို နိုးဆွလိုက်တဲ့ ပရိုဂရမ်နေရာကို ပြန်သွားပါတယ်။ ဒီပရိုဂရမ်ဟာ Java စကားပြန်စက်ဖြစ်ပြီး၊ window ပိတ်တာတွေ၊ မှတ်ဉာဏ်ရှင်လင်းရေးတွေ လုပ်ပါတယ်။ ပြီးမှပရို ဂရမ်ပြီးဆုံးသွားပါတယ်။

ဒီပုံပြင်လေးရဲ့ သင်ခန်းစာက ဘာပါလဲ။ ပရိုဂရမ်တစ်ပုဒ်ကို ဖတ်တဲ့အခါ အစကနေ အဆုံးကို မဖတ်ပါနဲ့၊ ပရိုဂရမ်စီးဆင်းပုံကို လိုက်ပါ။

## Parameter များနှင့် Argument များ

သုံးခဲ့တဲ့ method တွေမှာ parameter တွေ ပါဝင်ပါတယ်။ သူတို့ဟာ method အနေနဲ့ အလုပ်လုပ်ဖို့ လိုအပ်တဲ့တန်ဖိုးတွေ ဖြစ်ပါတယ်။ ဥပမာ ကိန်းတစ်ခုရဲ့ sine ကို ရှာချင်ရင် ဘယ်ကိန်းကို ရှာချင်နေတာလဲဆိုတာ ပြောပြရပါတယ်။ ဒီတော့ sin ဟာ double ကို parameter အနေနဲ့ လက်ခံပါတယ်။ စာသားတစ်ခုကို print လုပ်ဖို့ စာသားကိုပံ့ပိုးပေးရပါတယ်။ ဒါကြောင့် println ဟာ String တစ်ခုကို parameter အနေနဲ့ ရယူတာဖြစ်ပါတယ်။

တချို့ method တွေက parameter တစ်ခုထက်ပိုယူပါတယ်။ ဥပမာ pow ဟာ double နှစ်ခုကို အခြေနဲ့ထပ်ညွှန်းအဖြစ် လက်ခံပါတယ်။ ဒါကြောင့်လည်း သူတို့ရဲ့ အမျိုးအစားတွေ

[illegible] class [illegible] parameter [illegible] parameter တစ်ခုရဲ့ [illegible] ဥပမာ

```
public static void printTwice(String phil) {
        System.out.println(phil);
        System.out.println(phil);
}
```

ဒီ method ဟာ phil ဆိုတဲ့ parameter တစ်ခုကို လက်ခံပါတယ်။ phil မှာ String type ရှိပါတယ်။ အဲဒီ parameter ကိုပဲ လက်ခံရရှိ သူ့ကို နှစ်ခါ print ထုတ်ပေးပါတယ်။ parameter ကိုပေးတဲ့နာမည်ဟာ အသုံးပြုချင်ရာ ရတယ်ဆိုတာ သိစေချင်တဲ့အတွက် phil ဆိုတဲ့ နာမည်ကို ရွေးချယ်ခဲ့ပါတယ်။ ဒါပေမယ့် ယေဘုယျအားဖြင့် parameter နာမည်ကို phil ထက်ပိုပြီး သိသာလွယ်တဲ့ နာမည်မျိုးကို ရွေးသင့်ပါတယ်။

ဒီ method ကို နိုးဖို့ String တစ်ခုကို ပို့ဖို့ရပါမယ်။ ဥပမာ main method ဟာ ဒီလိုပုံစံရှိနိုင်ပါတယ်။

```
public static void main(String[] args) {
        printTwice("Don't make me say this twice!");
}
```

ပို့ပေးလိုက်တဲ့စာသားကို argument လို့ခေါ်ပါတယ်။ အဲဒီ argument ကို method ဆီ ပို့ပေးတယ် (pass) လို့ခေါ်ပါတယ်။ ဒီနေရာမှာ "Don't make me say this twice!" ဆိုတဲ့ စာပါတဲ့ စာသားတန်ဖိုးတစ်ခုကို ဖန်တီးနေတာ ဖြစ်ပါတယ်။ ပြီးတော့ သူ့ကို printTwice ဆီပို့ပြီး ကန့်ကွက်နေတဲ့ကြားက နှစ်ကြိမ် print အလုပ်ခံခိုင်းပါတယ်။

ဒီလိုမလုပ်ဘဲ String variable တစ်ခုရှိရင် သူ့ကို argument တစ်ခုအနေနဲ့ သုံးနိုင်ပါတယ်။

```
public static void main(String[] args) {
        String argument = "Never say never.";
        printTwice(argument);
}
```

ဒီအခါမှာ အင်မတန်အရေးကြီးတဲ့ ကိစ္စတစ်ခုကို သတိပြုပါ။ argument ဆိုတဲ့ ပေးပို့ခံရတဲ့ variable ရဲ့ နာမည်ဟာ phil ဆိုတဲ့ parameter နာမည်နဲ့ ဘာမှ ပတ်သက်မှုမရှိပါဘူး။ ထပ်ပြောပါဦးမယ်။ Argument တစ်ခုအဖြစ် ပေးပို့ခံရတဲ့ variable နာမည်ဟာ parameter နာမည်နဲ့ လုံးဝဥဿုံ ပတ်သက်မှုမရှိပါ။

သူတို့ဟာ အတူတူဖြစ်လို့ ရပါတယ်။ ကွဲပြားချင်လည်း ကွဲပြားနိုင်ပါတယ်။ ဒါပေမယ့်

သူတို့က တစ်ခုတည်းမဟုတ်ပါဘူး။ (ထပ်တူမညီပါဘူး။) တူညီတဲ့တန်ဖိုးကို ပိုင်ဆိုင်ထားက လွဲရင်ပေါ့။ (ဒီနေရာမှာ တူညီတဲ့တန်ဖိုးက "Never say never." ဖြစ်ပါတယ်။)

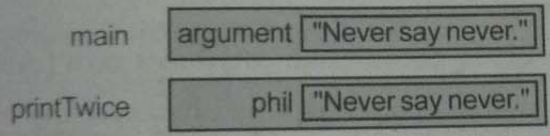
Argument အဖြစ် ပို့ပိုးပေးတဲ့ တန်ဖိုးဟာ နှိုးဆွတဲ့ method နဲ့ အမျိုးအစားတူရပါမယ်။ ဒီစည်းမျဉ်းဟာ အင်မတန်အရေးကြီးပါတယ်။ ဒါပေမယ့် ဒီကိစ္စဟာ အကြောင်းနှစ်ခုကြောင့် ရှုပ်ထွေးလာနိုင်ပါတယ်။

- အမျိုးအစားမျိုးစုံကို argument အဖြစ်လက်ခံတဲ့ method တချို့ ရှိပါတယ်။ ဥပမာ println ဆိုကို ကြိုက်တဲ့ အမျိုးအစားကို ပို့နိုင်ပါတယ်။ ဘာပဲဖြစ်ဖြစ် သူတို့ဟာမှန်တန်တဲ့ အလုပ်ကို လုပ်ပါလိမ့်မယ်။ ဒါပေမယ့် ဒီကိစ္စမျိုးဟာ ခြွင်းချက်ဖြစ်ပါတယ်။
- ဒီစည်းမျဉ်းကို ချိုးဖောက်ရင် compiler ဟာ နားရှုပ်စေတဲ့ အမှား message တစ်ခုကို ထုတ်လုပ်တတ်ပါတယ်။ "ဒီ method ဆီပို့တဲ့ argument အမျိုးအစားမှားနေတယ်" ဆိုတာမျိုး ပြောမယ့်အစား အဲဒီအမျိုးအစားရှိတဲ့ argument ကိုလက်ခံဖို့ အဲဒီနာမည်ရှိတဲ့ method တစ်ခုကို ရှာမတွေ့ဘူးလို့ ပြောလာပါမယ်။ ဒီလို အမှား meassage ကို မကြာခဏမြင်ဖူးရင် ဘယ်လိုအဓိပ္ပာယ်ကောက်ရမလဲဆိုတာ သိလာပါလိမ့်မယ်။

## Stack ပုံများ

Parameter တွေနဲ့ တခြား variable တွေဟာ သူတို့ရဲ့ method တွေထဲမှာပဲ ရှိနေနိုင်ပါတယ်။ main ရဲ့ အကန့်အသတ်တွေထဲမှာ phil ဆိုတဲ့အရာမျိုး မရှိပါဘူး။ သူ့ကိုသုံးဖို့သွားကြိုးစားရင် compiler ဟာ ငြီးတွားပါလိမ့်မယ်။ ဒီလိုပဲ printTwice ထဲမှာ argument ဆိုတဲ့ method မရှိပါဘူး။

Variable တစ်ခုစီကို ဘယ်လိုသတ်မှတ်ထားသလဲလို့ မြင်သာစေချင်ရင် stack diagram တစ်ခုကို သုံးနိုင်ပါတယ်။ ပြီးခဲ့တဲ့ ဥပမာအတွက် stack diagram ဟာ ဒီလိုပုံ ရှိပါတယ်။




Method တစ်ခုစီအတွက် frame လို့ခေါ်တဲ့ မီးခိုးရောင်အကွက်လေးတစ်ခု ရှိပါတယ်။ သူက parameter တွေ၊ ဒေသခံ variable (local variable) တွေကို သယ်ဆောင်ပါတယ်။ Method နာမည်ဟာ frame ရဲ့ ပြင်ပမှာ ရှိပါတယ်။ ထုံးစံလိုပဲ variable တန်ဖိုးတစ်ခုစီကို လေးထောင့်ကွက်ထဲမှာ variable နာမည်ကို ဘေးမှာရေးပြီး ဖော်ပြပါတယ်။

## တစ်ခုထက်ပိုသော Parameter ပါဝင်သော Method များ

Parameter တစ်ခုထက်ပိုပါတဲ့ method တွေကို ကြေညာရာမှာ/နှိုးဆွရာမှာ အသုံးပြုရတဲ့ သဒ္ဒါဟာ အမှားတွေဖြစ်ပေါ်စေတတ်တဲ့ လမ်းစကြီးတစ်ခု ဖြစ်လာပါတယ်။ ပထမဆုံး

သတိပြုရမှာက parameter တိုင်းရဲ့ အမျိုးအစားကို ကြေညာဖို့လိုပါတယ်။ ဥပမာ

```
public static void printTime(int hour, int minute) {
        System.out.print(hour);
        System.out.print(":");
        System.out.println(minute);
}
```

int hour, minute ကို ရေးကြဖို့ စဉ်းစားမိကောင်း စဉ်းစားမိကြပါမယ်။ ဒါပေမယ့် ဒါဟာ variable ကြေညာချက်တွေမှာသာ တရားဝင်ပြီး parameter တွေအတွက် တရားမဝင်ပါဘူး။

နောက်ထပ်မှားတတ်တာက argument တွေရဲ့ အမျိုးအစားကို ကြေညာစရာမလိုပါဘူး။ အောက်မှာပြောထားတာက မှားနေပါတယ်။

```
int hour = 11;
int minute = 59;
printTime(int hour, int minute); //  WRONG!
```

ဒီနေရာမှာ Java ဟာ hour နဲ့ minute ရဲ့ အမျိုးအစားကို သူတို့ရဲ့ကြေညာချက်မှာ ကြည့်ပြီး သိနိုင်ပါတယ်။ သူတို့ကို argument တွေအဖြစ် ပေးပို့တဲ့အခါ အမျိုးအစားကို ပြန်ကြေညာစရာမလိုပါဘူး။ မှန်ကန်တဲ့သဒ္ဒါဟာ printTime(hour, minute) ဖြစ်ပါမယ်။

## ရလဒ်များပါသော Method များ

အသုံးပြုခဲ့သမျှ method တွေထဲမှာ Math ရဲ့ method တွေလို တချို့ method တွေဟာ ရလဒ်တွေကို ထုတ်ပေးတယ်လို့ သတိပြုမိကောင်း ပြုမိခဲ့ကြပါမယ်။ println နဲ့ newLine ဟာ မေးခွန်းတချို့ကို ပေါ်ပေါက်စေပါတယ်။

- Method တစ်ခုကိုနှိုးဆွပြီး ရလဒ်နဲ့ ဘာမှမလုပ်ရင် ဘာဖြစ်သွားသလဲ။ (အဓိပ္ပာယ်က သူ့ကို variable တစ်ခုမှာ နေရာချမထားရင် ဒါမှမဟုတ် တခြားဖော်ပြချက်တစ်ခုမှာ ထည့်မသုံးရင် ဘာဖြစ်သွားလဲ။)
- ရလဒ်တွေထုတ်ပေးတဲ့ method တွေကို ရေးလို့ရသလား။ ဒါမှမဟုတ် newLine နဲ့ printTime လို ဘာမှ ထုတ်မပေးတဲ့ method တွေနဲ့ပဲ method ဇာတ်လမ်းဟာ ပြီးသွားမှာလား။

ပထမအခြေအနေကိုသိချင်ရင် ပရိုဂရမ်တစ်ပုဒ်မှာ ထည့်သုံးပြီး compile လုပ်ကြည့်ပါ။ Compiler ဖြေပါလိမ့်မယ်။ ဒုတိယမေးခွန်းအတွက်တော့ တန်ဖိုးတွေပြန်ထုတ်ပေးတဲ့ method တွေကို ရေးသားနိုင်တယ်လို့ ဖြေရမှာဖြစ်ပြီး သူတို့အကြောင်းကို ရှေ့အခန်းတွေမှာ ဆက်လေ့လာသွားပါမယ်။

အခန်း (၄)

# အခြေအနေများနှင့် ထပ်တလဲလဲဖြစ်စဉ်များ

## Modulus လက္ခဏာ

Modulus လက္ခဏာဟာ ကိန်းပြည့်တွေနဲ့ အလုပ်လုပ်ပြီး (ဒါမှမဟုတ် ကိန်းပြည့်ဖော်ပြချက်တွေနဲ့ အလုပ်လုပ်ပြီး) ပထမ operand ကို ဒုတိယ operand နဲ့ စားလို့ရတဲ့ စားကြွင်းကို ပြန်တွက်ပေးပါတယ်။ Java မှာ modulus လက္ခဏာဟာ ရာခိုင်နှုန်းသင်္ကေတ ဖြစ်ပါတယ်။ သဒ္ဒါကတော့ တခြားလက္ခဏာတွေနဲ့ အတူတူဖြစ်ပါတယ်။

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

ပထမ operator ဟာ ကိန်းပြည့်အစားဖြစ်ပြီး 2 ကို ထုတ်ပေးပါတယ်။ ဒုတိယ operator က 1 ထုတ်ပေးပါတယ်။ ဒီတော့ 7 ကို 3 နဲ့စားရင် 2 ရပြီး 1 ကြွင်းပါတယ်။

Modulus လက္ခဏာဟာ အံ့ဩလောက်အောင် အသုံးဝင်ပါတယ်။ ဥပမာ ကိန်းတစ်ခုကို နောက်ကိန်းနဲ့ စားလို့ပြတ်လားလို့ စစ်ဆေးနိုင်ပါတယ်။ x % y ဟာ သုညဖြစ်ရင် x ကို y စားလို့ ပြတ်ပါတယ်။

ပြီးတော့ modulus လက္ခဏာကို ကိန်းတစ်ခုရဲ့ညာစွန်က ဂဏန်းတွေခွဲထုတ်ရာမှာ သုံးနိုင်ပါတယ်။ ဥပမာ x % 10 ဟာ x ရဲ့ ညာစွန်က ဂဏန်းကို ထုတ်ပေးပြီး x % 100 ဟာ နောက်

ဆုံးတောက်နံနှစ်လုံးကို ထုတ်ပေးပါတယ်။ (အခြေ 10 သုံးမှ ဒါဟာမှန်ပါတယ်။)

ဒါကို မဟောင်ဖြစ်နေတဲ့သူတွေအတွက် 12 ကို 10 နဲ့စားရင် 2 ကြွင်းပါတယ်။ 132 ကို 100 နဲ့ စားရင် 32 ကြွင်းပါတယ်။

## အခြေအနေပေါ်မူတည်သော Run ခြင်း

အသုံးဝင်တဲ့ပရိုဂရမ်တွေ ရေးနိုင်အောင် အခြေအနေကိုစစ်ဆေးနိုင်တဲ့ အရည်အချင်းနဲ့ ပရိုဂရမ်ရဲ့အပြုအမူကို ပြုပြင်နိုင်စွမ်းကို အမြဲလိုလို အလိုရှိပါတယ်။ အခြေအနေမှီခိုတဲ့ ဖော်ပြချက်တွေဟာ ဒီစွမ်းရည်ကို ပေးစွမ်းနိုင်ပါတယ်။ အရိုးရှင်းဆုံးပုံစံကတော့ if ဖော်ပြချက် ဖြစ်ပါတယ်။

```
if(x > 0) {
    System.out.println("x is positive");
}
```

လက်သည်းကွင်းထဲက ဖော်ပြချက်ဟာ အခြေအနေဖြစ်ပါတယ်။ အကယ်၍ ဒါဟာ မှန်တယ်ဆိုရင် တွန့်ကွင်းထဲက ဖော်ပြချက်တွေကို run ပါတယ်။ အကယ်၍ အခြေအနေဟာ မမှန်ဘူးဆိုရင် ဘာမှမဖြစ်ပါဘူး။

အခြေအနေမှာ နှိုင်းရလက္ခဏာ (comparison operator) တွေ ပါဝင်နိုင်ပါတယ်။ သူတို့ကို relational operator တွေလို့လည်း ခေါ်ပါတယ်။

```
x == y          // x equals y
y != y          // x is not equal to y
x > y           // x is greater than y
x < y           // x is less than y
x >= y          // x is greater than or equal to y
x <= y          // x is less than or equal to y
```

ဒီလုပ်ဆောင်ချက်တွေဟာ ရင်းနှီးပြီးသား ဖြစ်နေသလိုလို ရှိနေပေမယ့် Java ကသုံးတဲ့ သင်္ကေတဟာ =, ≠, နဲ့ ≤ တို့လို သင်္ချာသင်္ကေတတွေနဲ့ ကွဲပြားပါတယ်။ ဖြစ်လေ့ဖြစ်ထရှိတဲ့ အမှားကတော့ == လက္ခဏာအစား = ကို သုံးတာဖြစ်ပါတယ်။ = ဟာ နေရာချထားတဲ့ လက္ခဏာဖြစ်ပြီး == ဟာ နှိုင်းရလက္ခဏာ ဖြစ်ပါတယ်။ နောက်ပြီး =< နဲ့ => လို လက္ခဏာတွေ မရှိပါဘူး။

အခြေအနေလက္ခဏာရဲ့ ဘယ်ညာက တန်ဖိုးတွေဟာ အမျိုးအစားတူရပါမယ်။ int တွေကို int တွေနဲ့ပဲ နှိုင်းယှဉ်နိုင်ပြီး double တွေကို double တွေနဲ့ပဲ နှိုင်းယှဉ်နိုင်ပါတယ်။ ကံမကောင်းစွာနဲ့ပဲ အခုချိန်မှာ String တွေကို နှိုင်းယှဉ်လို့မရနိုင်ပါဘူး။ စာသားတွေ နှိုင်းယှဉ်တဲ့နည်းတော့ ရှိပါတယ်။ ဒါပေမယ့် အခုတော့ ဒီကိစ္စကို မဆွေးနွေးသေးပါဘူး။

## ရွေးချယ်မှုပြုလုပ်သော Run ခြင်း

အခြေအနေမှီခိုတဲ့ ဒုတိယ run နည်းဟာ ရွေးချယ်မှုကိုပြုလုပ်တဲ့ နည်းပဲဖြစ်ပါတယ်။ ဖြစ်နိုင်ခြေနှစ်မျိုး ရှိပါတယ်။ အခြေအနေက ဖြစ်နိုင်ခြေနှစ်ခုရဲ့ ဘယ်တစ်ခုကို run မလဲဆိုတာ ဆုံးဖြတ်ပါတယ်။ သဒ္ဒါက ဒီလိုပုံစံရှိပါတယ်။

```
if (x % 2 == 0) {
    System.out.println("x is even");
  } else {
    System.out.println("x is odd");
  }
}
```

x ကို 2 နဲ့စားရင် ရမယ့်အဖြေဟာ သုညဖြစ်ရင် x ဟာ စုံကိန်းဖြစ်တယ်လို့ သိပါတယ်။ ဒီတော့ စုံကိန်းဖြစ်ကြောင်းကို print လုပ်ပါတယ်။ ဒီအခြေအနေဟာ မှားနေတယ်ဆိုရင် ဒုတိယ print ဖော်ပြချက်ကို run ပါတယ်။ အခြေအနေဟာ မှားရင်မှား၊ မမှားရင်မှန်ရမှာ ဖြစ်တဲ့အတွက် နှစ်ခုထဲကတစ်ခုကိုသာ run ပေးပါလိမ့်မယ်။

ကိန်းတွေစုံဖြစ်သလား၊ မဖြစ်သလားဆိုတာ မကြာခဏစစ်ချင်ရင် ဒီ code ကို method တစ်ခုထဲမှာ ထည့်ထားနိုင်ပါတယ်။

```
public static void printParity(int x) {
  if (x%2) = 0) {
        System.out.println("x is even");
  } else {
        System.out.println("x is odd");
  }
}
```

အခုအချိန်မှာ printParity ဆိုတဲ့ ကိန်းပြည့်တစ်ခုကို စုံလား မလား ပြောပြတဲ့ method တစ်ခုကို ရပါပြီ။ main မှာ ဒီ method ကို အောက်မှာပြထားသလို နှိုးဆွရပါတယ်။

```
printParity(17);
```

Method တစ်ခုကို နှိုးဆွရာမှာ ကိုယ်ပံ့ပိုးပေးတဲ့ argument တွေရဲ့ အမျိုးအစားကို ကြေညာစရာမလိုဘူးဆိုတာ မှတ်ထားပါ။ Java ဟာ သူတို့အမျိုးအစား ဘာလဲဆိုတာ အလိုအလျောက်သိပါတယ်။ အောက်မှာပြထားတဲ့ code မျိုးကို ရေးချင်တဲ့အခါတွေရှိရင် မျိုသိပ်ထားလိုက်ပါ။

```
int number = 17;
printParity(int number);        // WRONG !!!
```

### ချိတ်ဆက်ထားသော Conditional များ

တစ်ခုနဲ့တစ်ခုမှာ ဆက်စပ်မှုရှိတဲ့ အခြေအနေတွေကို စစ်ဆေးပြီး လုပ်ဆောင်ချက်ကို ရွေးချယ်လိုတဲ့အခါ တွေ ရှိလာနိုင်ပါတယ်။ ဒီလိုလုပ်နိုင်တဲ့တစ်နည်းက if တွေ else တွေကို ချိတ်ဆက်အသုံး ပြုတာ ဖြစ်ပါတယ်။

```
if (x > 0) {
  System.out.println("x is positive");
} else if (x < 0) {
  System.out.println("x is negative");
} else {
  System.out.println("x is zero");
}
```

ဒီချိတ်ဆက်မှုဟာ ကြိုက်သလောက်ရှည်နိုင်ပါတယ်။ အရမ်းရှည်လာရင်တော့ ဖတ်ရခက် လာနိုင်ပါတယ်။ သူတို့ကို ဖတ်ရလွယ်အောင် လုပ်နည်းကတော့ ကွက်လပ်ခြားတဲ့စံကို အသုံးပြု တာဖြစ်ပါတယ်။ ဒါကို ဒီဥပမာမှာ မြင်နိုင်ပါတယ်။ ဖော်ပြချက်အများကြီးနဲ့ ကွင်းကွင်းတွေကို အောက်ကိုတန်းစီထားရင် သင်္ချာအမှားတွေလုပ်မိတာ ပိုနည်းသွားပါမယ်။ မှားသွားရင်လည်း မြင်ရလွယ်ကူစေပါတယ်။

### ထပ်ဆင့်ထားသော အခြေအနေများ

အခြေအနေတွေကို ချိတ်ဆက်ထားနိုင်တာအပြင် ထပ်ဆင့်ရေးသားလို့လည်း ရပါတယ်။ ပြီးခဲ့တဲ့ ဥပမာကို ဒီလိုပြင်ရေးနိုင်ပါတယ်။

```
if (x == 0) {
  Systme.out.println("x is zero");
} else {
  if (x > 0) {
        System.out.println("x is positive");
  } else {
        System.out.println("x is negative");
  }
}
```

အခုအချိန်မှာ ပြန်ခွဲထားတဲ့ အခြေအနေလမ်းကြောင်းနှစ်ခုပါတဲ့ ပြင်ပအခြေအနေတစ် ခုကို ရရှိထားပြီဖြစ်ပါတယ်။ ပထမအခြေအနေလမ်းကြောင်းမှာ ရိုးရှင်းတဲ့ print ဖော်ပြချက် တစ်ခု ပါဝင်ပါတယ်။ ဒုတိယအခြေအနေလမ်းကြောင်းမှာတော့ နောက်ထပ် အခြေအနေဖော်

ပြချက်တစ်ခု ပါဝင်ပါတယ်။ သူမှာလည်း အခြေအနေတယ်ကြောင်းနှစ်ခု ထပ်ပါပြန်ပါတယ်။ ထိုအောက်အဆောက်များနဲ့ ဒီလယ်ကြောင်းနဲ့ နှစ်ခုစလုံးဟာ print ဖော်ပြချက်တွေ ဖြစ်နေပါတယ်။ တကယ်တမ်းမှာ သူတို့ ကိုယ်တိုင်ကပဲ အခြေအနေဖြစ်နိုင်ပါတယ်။

ကျွတ်ဝင်နေရာချထားတာဟာ တည်ဆောက်ပုံကို ရှင်းလင်းစေတယ်ဆိုတာကိုလည်း သတိပြုပါ။ ဒါပေမဲ့ ထပ်ဆင့်အခြေအနေတွေဟာ အလွယ်တကူ ဖတ်ရခက်စေပါတယ်။ ယေဘုယျအားဖြင့် သူတို့ကို တတ်နိုင်သလောက် ရှောင်သင့်ပါတယ်။

အခြားတစ်ဖက်မှာတော့ ထပ်ဆင့်တည်ဆောက်ပုံကို အသုံးများပါတယ်။ သူတို့ကိုလည်း နောက်ထပ်မြင်ရပါဦးမယ်။ ဒီတော့ သူနဲ့ယဉ်ပါးအောင် လုပ်ထားသင့်ပါတယ်။

## တန်ဖိုးပြန်ထုတ်ပေးသော ဖော်ပြချက်များ

return ဖော်ပြချက်တွေဟာ method တစ်ခု run နေတာကို အဆုံးအထိမရောက်သေးခင်မှာ ရပ်တန့်စေပါတယ်။ သူ့ကိုသုံးနိုင်တဲ့ အကြောင်းတစ်ခုက အမှားအခြေအနေတစ်ခု ဖြစ်ပါတယ်။

```
public static void printLogarithm(double x) {
   if (x <= 0.0) {
         Systme.out.println("Position numbers only, please.");
         return;
   }
   double result = Math.log(x);
   System.out.println("The log of x is " + result);
}
```

ဒီစာ printLogarithm ဆိုတဲ့ method တစ်ခုကို သတ်မှတ်ပါတယ်။ သူက x လို့ နာမည်ပေးထားတဲ့ double တစ်ခုကို parameter အဖြစ်ခံယူပါတယ်။ သူအရင်ဆုံးလုပ်တာက x ဟာ သုညနဲ့ ညီနေသလား၊ ငယ်နေသလားဆိုတာ စစ်ပါတယ်။ ဒီလိုဖြစ်နေရင် အမှား message တစ်ခုကို print လုပ်ပြီး method ထဲကထွက်ဖို့ return ဖော်ပြချက်ကို သုံးပါတယ်။ Run နေတဲ့ လမ်းကြောင်းဟာ ချက်ချင်း method ကို စခေါ်တဲ့နေရာကို ပြန်ရောက်သွားပါတယ်။ Method ကျန်တဲ့ စာကြောင်းတွေကို ဆက်ပြီး မ run တော့ပါဘူး။

ဘယ်ဖက်မှာ floating-point variable တစ်ခုရှိနေတဲ့အတွက် ညာဖက်မှာလည်း floating-point တန်ဖိုးတစ်ခုကို သုံးထားပါတယ်။

## အမျိုးအစားပြောင်းလဲခြင်း

"The log of x is " + result လို ဖော်ပြချက်မျိုးကို ဘယ်လိုလုပ်ပြီးရေးလို့ ရတာပါလိမ့်။ ဘာလို့လဲဆိုတော့ operand တစ်ခုက String ဖြစ်ပြီး ကျန်တဲ့တစ်ခုက double ဖြစ်နေ

ပါတယ်။ ဒီနေရာမှာ Java က ပရိုဂရမ်ရေးသူအစား တော်ပြနေတာဖြစ်ပါတယ်။ သူက စာသား ပေါင်းစည်းမှုမလုပ်ခင် double ကို String အဖြစ် ပြောင်းပါတယ်။

ဒီလတ္တကာဟာ ပရိုဂရမ်းမင်းဘာသာစကားတစ်ခုကို ဒီဇိုင်းထုတ်ရာမှာ ကြုံရလေ့ရှိတဲ့ ပြဿနာတစ်ခု ဖြစ်ပါတယ်။ ဒီပြဿနာကတော့ လျော့ကန်တဲ့ ဘာသာစကားတွေမှာ ခြွင်းချက် နည်းနည်းနဲ့ ရိုးရှင်းတဲ့ ဥပဒေသတွေရှိသင့်တယ်ဆိုတဲ့ လျော့ကန်မှုဝါဒနဲ့ ပရိုဂရမ်မင်းဘာသာ စကားတွေကို လက်တွေ့မှာ သုံးရလွယ်ရမယ်ဆိုတဲ့ အဆင်ပြေမှုကြားက လွန်ဆွဲပွဲဖြစ်ပါတယ်။

ဖြစ်လေ့ရှိတာကတော့ အဆင်ပြေမှုက နိုင်တတ်ပါတယ်။ ဒါဟာ တင်းကျပ်တဲ့ လျော့ကန်မှု ကနေ ကင်းလွတ်ခွင့်ရတဲ့ အတွေ့အကြုံရင့် ပရိုဂရမ်မာတွေအတွက် ကောင်းတတ်ပြီး ရှုပ်ထွေး တဲ့ဥပဒေသတွေနဲ့ ခြွင်းချက်အရေအတွက်တွေကြောင့် နဝေတိမ်တောင်ဖြစ်တတ်တဲ့ ပရိုဂရမ်း မင်း စသင်သူတွေအတွက်တော့ ဆိုးရွားတတ်ပါတယ်။ ဒီစာအုပ်မှာ ကိစ္စတွေကို ရှင်းလင်းအောင် စည်းမျဉ်းတွေမှာပဲ အာရုံစိုက်ပြီး ခြွင်းချက်တော်တော်များများကို ချန်ထားခဲ့ပါတယ်။

ဒါပေမယ့် ဖော်ပြချက်နှစ်ခုကို ပေါင်းဖို့ကြိုးစားတဲ့အခါ တစ်ခုက String ဖြစ်နေရင် Java ဟာ ကျန်တဲ့တစ်ခုကိုလည်း String အဖြစ်ပြောင်းပြီး စာသားပေါင်းစည်းမှုကို လုပ်ပါ လိမ့်မယ်။ ကိန်းပြည့်တစ်ခုနဲ့ floating-point တန်ဖိုးတစ်ခုကို ပေါင်းတဲ့အခါရော ဘာဖြစ်မယ် ထင်သလဲ။

### ထပ်တလဲလဲဖြစ်စဉ်

ပြီးခဲ့တဲ့အခန်းမှာတုန်းက method တစ်ခုဟာ နောက်ထပ် method တစ်ခုကို ပြန်ခေါ်နိုင် တယ်ဆိုတာ သိခဲ့ပါတယ်။ ဒီလို ဥပမာတွေကိုလည်း တွေ့ခဲ့ပါတယ်။ ဒါပေမယ့် method တစ်ခု ဟာ သူ့ကိုသူ ပြန်ခေါ်နိုင်တယ်ဆိုတာကိုတော့ မပြောခဲ့ပါဘူး။ ဒါဟာ ဘာကြောင့်ကောင်းသလဲ ဆိုတာကိုတော့ ချက်ချင်းမြင်မှာ မဟုတ်ပါဘူး။ ဒါပေမယ့် သူဟာ ပရိုဂရမ်မင်းရဲ့ ရင်သပ်ရှု မောဖွယ်အကောင်းဆုံးနဲ့ စိတ်ဝင်စားစရာအကောင်းဆုံး ကိစ္စတစ်ခုဖြစ်နေပါတယ်။

ဥပမာအနေနဲ့ အောက်က method ကိုကြည့်ပါ။

```
public static void countdown(int n){
    if (n == 0) {
            System.out.println("Blastoff!");
    } else {
            System.out.println(n);
            countdown(n-1);
    }
}
```

Method ရဲ့ နာမည်ဟာ countdown ဖြစ်ပြီး ကိန်းပြည့်တစ်ခုကို parameter အဖြစ်ယူ ပါတယ်။ Parameter က သုညဖြစ်ရင် Blastoff! ဆိုတဲ့စကားလုံးကို print လုပ်ပါတယ်။ မဟုတ်

ရင် အဲဒီကိန်းကို print လုပ်ပြီး သူကိုယ်တိုင်ဖြစ်တဲ့ countdown method မှာ အဲဒီကိန်းရဲ့ တစ်နုတ်ထားတဲ့တန်ဖိုးကို argument အနေနဲ့ ပို့ပေးလိုက်ပါတယ်။

ဒီ method ကို main မှာ ဒီလိုနှိုးဆွလိုက်ရင် ဘာဖြစ်သွားမလဲ။

countdown(3);

countdown ကို n=3 နဲ့ စ run ပါတယ်။ n က သုညမဟုတ်တဲ့အတွက် တန်ဖိုး 3 ကို print လုပ်ပါတယ်။ ပြီးရင် သူ့ကိုယ်သူ ပြန်ပြီးနှိုးဆွပါတယ်။

countdown ကို n=2 နဲ့ စ run ပါတယ်။ n ဟာ သုညမဟုတ်တဲ့အတွက် တန်ဖိုး 2 ကို print လုပ်ပြီး သူ့ကိုယ်သူ ပြန်နှိုးဆွပါတယ်။

countdwon ကို n=1 နဲ့ စ run ပါတယ်။ n ဟာ သုညမဟုတ်တဲ့အတွက် တန်ဖိုး 1 ကို print လုပ်ပြီး သူ့ကိုယ်သူ ပြန်နှိုးဆွပါတယ်။

countdwon ဟာ n=0 နဲ့ စ run ပါတယ်။ n ဟာ သုညဖြစ်နေတဲ့အတွက် Blastoff! ဆိုတဲ့စကားလုံးကို print လုပ်ပြီး ပြန်ပါတယ်။

n=1 ရှိတဲ့ countdown က ထွက်ပါတယ်။

n=2 ရှိတဲ့ countdown ထွက်ပါတယ်။

n=3 ရှိတဲ့ countdown ထွက်ပါတယ်။

ပြီးတဲ့အခါ main ဆီ ပြန်ရောက်ပါတယ်။ (တော်တော်သွားလိုက်ရတာပဲ။) နောက်ဆုံးရတဲ့ output ဟာ ဒီလိုပုံစံရှိပါတယ်။

```
3
2
1
Blastoff!
```

ဒုတိယဥပမာအနေနဲ့ newLine နဲ့ threeLine ဆိုတဲ့ method တွေကို ပြန်ကြည့်ပါ။

```
public static void newLine() {
    System.out.println(" ");
}

public static void threeLine() {
    newLine(); newLine(); newLine();
}
```

ဒါဟာ အလုပ်ဖြစ်ပေမယ့် စာကြောင်းနှစ်ကြောင်းကို ၁၀၆ ကြောင်းလို print လုပ်ချင်တဲ့အခါ ဒါဟာ အလုပ်မဖြစ်ပါဘူး။ ပိုကောင်းတဲ့နည်းတစ်ခုက

```
public static void nLines(int n) {
```

```
    if (n > 0) {
        System.out.println("");
        nLines(n-1);
    }
}
```

ဒီပရိုဂရမ်ဟာ အရင်က ပရိုဂရမ်နဲ့ တော်တော်ဆင်တူပါတယ်။ n ဟာ သုညထက်ကြီးသရွေ့ စာကြောင်းအသစ်တစ်ခုကို print လုပ်ပါတယ်။ သူဟာ သူ့ကိုယ်သူ n-1 အရေအတွက်ရှိတဲ့ စာကြောင်းအသစ်တွေကို print လုပ်ဖို့ နှိုးဆွပါတယ်။ အဲဒီအခါ print အလုပ်ခံရတဲ့ စာကြောင်းသစ် အရေအတွက်စုစုပေါင်းဟာ 1 + (n-1) ဖြစ်သွားပါတယ်။ အဲဒီအရေအတွက်ဟာ n ပါပဲ။ (1 + (n-1) = n)

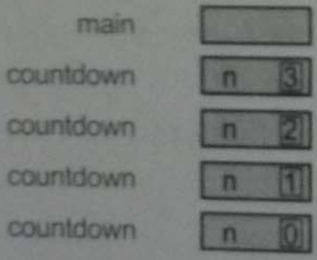
Method တစ်ခုက သူ့ကိုယ်သူ ပြန်နှိုးဆွတဲ့ဖြစ်စဉ်ကို ထပ်တလဲလဲဖြစ်ခြင်း (recursion) လို့ ခေါ်ပြီး အဲဒီ method ကို ထပ်တလဲလဲဖြစ်တဲ့ method လို့ ခေါ်ပါတယ်။

## ထပ်တလဲလဲဖြစ်သော Method များအတွက် Stack ပုံများ

ပြီးခဲ့တဲ့အခန်းတုန်းက ပရိုဂရမ်တစ်ခုရဲ့ အခြေအနေကို method ခေါ်ဆိုမှုတစ်ခု ပြုလုပ်နေချိန်မှာ ကိုယ်စားပြုဖို့ stack ပုံတစ်ခုကို သုံးခဲ့ပါတယ်။ ဒီလိုပုံမျိုးကိုသုံးပြီး ထပ်တလဲလဲဖြစ်ပေါ်တဲ့ method တစ်ခုကို နားလည်ရပိုလွယ်အောင် လုပ်နိုင်ပါတယ်။

Method တစ်ခုကို ခေါ်တဲ့အကြိမ်တိုင်း အဲဒီ method ရဲ့ instance အသစ်တစ်ခုကို ဖန်တီးပါတယ်။ သူဟာ method ရဲ့ ဒေသခံ variable နဲ့ parameter အသစ်တွေကို ပိုင်ဆိုင်ပါတယ်။

အောက်မှာပြထားတာက countdown လုပ်တဲ့ ပရိုဂရမ်ကို n=3 နဲ့ စခေါ်တဲ့ အခြေအနေအတွက် stack ပုံဖြစ်ပါတယ်။




main အတွက် instance တစ်ခုပါဝင်ပြီး countdown အတွက် instance လေးခုပါဝင်ပါတယ်။ သူတို့တစ်ခုစီအတွက် parameter n ရဲ့ တန်ဖိုးတစ်ခုစီ ပါဝင်ပါတယ်။ Stack အထပ်ရဲ့ အောက်ခြေမှာ countdown ဟာ တန်ဖိုး n=0 ရှိပြီး အောက်ခြေအနေအထားမှာ ရှိနေပါတယ်။ သူက ထပ်တလဲလဲခေါ်ဆိုမှုတစ်ခုကို မပြုလုပ်တဲ့အတွက် countdown ရဲ့ instance

တွေ ထပ်မံနေတော့ပါဘူး။

main ရဲ့ instance က လွတ်နေပါတယ်။ အကြောင်းက main မှာ parameter တွေ ဒေသခံ variable တွေ မရှိလို့ပါပဲ။

## အရှင်အလောနှင့် သဘာဝနီယာမ

ပြီးခဲ့တဲ့ အပိုင်းတွေမှာတုန်းက ဟိုလိုလုပ်မယ် ဒီလိုလုပ်မယ်ဆိုတဲ့ အကြောင်းပြချက် ကြီးကြီး ကျယ်ကျယ်မရှိဘဲ အရင်ကတည်းက ဒီလိုလုပ်ခဲ့လို့ ဒီလိုလုပ်ရဲ့ ဆုံးဖြတ်ခဲ့တဲ့ကိစ္စတွေအတွက် အရှင်အလောအရဆိုတဲ့ စေါ်ဟာရကို ရေလဲသုံးခဲ့ပါတယ်။

ဒီနေရာတွေမှာ အစဉ်အလာအရ လုပ်ကိုင်တဲ့လုပ်ထုံးနဲ့ ရင်းနှီးပြီးအသုံးပြုရင် ကိုယ့်အ ထွက် အကျိုးရှိနိုင်ပါတယ်။ အာလို့လဲဆိုတော့ ဒီလိုလုပ်ခြင်းအားဖြင့် တခြားလူတွေက ကိုယ့် ပရိုဂရမ်တွေကို ဖတ်ရလွယ်ကူစေပါတယ်။ ဒါပေမယ့် စည်းမျဉ်းအမျိုးအစားသုံးခုကို ခွဲခြားသိ ရင် ပရိုဂရမ်ရေးလေ့လာရတာကို ပိုလွယ်ကူစေပါတယ်။

**သဘာဝနီယာမ။** ။ ဒါဟာ အောက်ခြေက ယုတ္တိ ဒါမှမဟုတ် သင်္ချာဥပဒေသတစ်ခုကြောင့် မှန်ကန်တဲ့စည်းမျဉ်းကို ဆိုလိုပါတယ်။ ဒါဟာ ပရိုဂရမ်မင်းဘာသာစကားတိုင်းသာ မဟုတ် ဘဲ လျော်ကန်မှုရှိတဲ့ စနစ်တိုင်းအတွက် မှန်ကန်ပါတယ်။ ဥပမာ အမှတ်လေးမှတ်နဲ့ ပတ် သက်တဲ့ အချက်အလက်တွေထက်လျော့ပြီး ပံ့ပိုးပေးရင် စတုဂံပုံနယ်နမိတ်တစ်ခုကို သတ် မှတ်လို့မရပါဘူး။ နောက်ပြီး ကိန်းပြည့်ပေါင်းခြင်းဟာ ဖလှယ်ရဂုဏ်သတ္တိကို လိုက်နာပါ တယ်။ ဒါဟာ အပေါင်းဥပဒေသတွေနဲ့သက်ဆိုင်ပြီး Java နဲ့ မပတ်သက်ပါဘူး။

**Java ထုံးနီယာမများ။** ။ ဒါတွေဟာ ပရိုဂရမ်မာတွေ ချိုးဖောက်လို့မရတဲ့ Java ရဲ့ သဒ္ဒါနဲ့ ဝေါဟာရဆိုင်ရာ စည်းမျဉ်းတွေဖြစ်ပါတယ်။ မဟုတ်ရင် ရလာတဲ့ပရိုဂရမ်ဟာ compile လုပ်မှာ၊ run မှာ မဟုတ်ပါဘူး။ တချို့ဥပဒေသတွေကတော့ ဟိုဘက်လိုက်သလိုလို၊ ဒီ ဘက်လိုက်လိုက်သလိုလို ပြတတ်ပါတယ်။ ဥပမာ + သင်္ကေတဟာ အပေါင်းနဲ့ variable နာမည်ကို ကိုယ်စားပြုသလို စာသားပေါင်းစည်းတာကိုလည်း ဆိုလိုပါတယ်။ တချို့က compile လုပ်တာနဲ့ run တာရဲ့ ဖြစ်စဉ်တွေရဲ့ အကန့်အသတ်ရှိတာပေါ်မှာ မူတည်ပါ တယ်။ ဥပမာ parameter တွေရဲ့ အမျိုးအစားတွေကိုသာ ကြေညာနိုင်ပြီး argument တွေမှာ ဒီလိုမလုပ်နိုင်ပါဘူး။ (Method ကြေညာရာမှာသုံးတဲ့ လက်ခံ variable တွေကို parameter လို့ခေါ်ပြီး method ခေါ်ယူရာမှာ ထည့်ပေးရတဲ့တန်ဖိုးတွေကို argument တွေလို့ ခေါ်ပါတယ်။)

**ပုံစံနှင့် အစဉ်အလာ ။** ။ Compiler က အတင်းအကျပ် မလိုက်နာခိုင်းပေမယ့်

ပရိုဂရမ်ရေးသားရာမှာ မှန်အောင် အမှားဖြစ်ရလွယ်အောင်နဲ့ အခြားပရိုဂရမ်မာတွေ ဖတ်
နိုင်အောင် လိုက်နာရတဲ့ စည်းမျဉ်းတွေရှိပါတယ်။ ဥပမာတွေမှာ ကွက်လပ်ခြားတာတွေ
တွန့်ကွင်းအသုံးပြုတာတွေနဲ့ variable, method နဲ့ class နာမည်ပေးနည်းတွေ ပါဝင်ပါတယ်။

ရှေ့ဆက်လေ့လာရာမှာ အကြံပြုချက်တွေဟာ ဘယ်အမျိုးအစားမှာ ပါတယ်ဆိုတာ တတ်
နိုင်သလောက် ပြောပြပါမယ်။ ဒါပေမယ့် တစ်ခါတစ်ရံမှာ ကိုယ့်ဘာသာတွေးရတာတွေလည်း
ရှိပါလိမ့်မယ်။

တလက်စတည်းပြောရရင် class နာမည်တွေကို အက္ခရာအကြီးနဲ့စပြီး method နာမည်
တွေကို အက္ခရာအသေးနဲ့စတယ်ဆိုတာ သတိပြုမိခဲ့ကြပါလိမ့်မယ်။ နာမည်တစ်ခုမှာ စကား
လုံးတစ်လုံးထက်ပိုပါရင် စကားလုံးတစ်ခုစီရဲ့ ပထမဆုံးအက္ခရာကို အကြီးနဲ့ရေးလေ့ရှိပါတယ်။
newLine နဲ့ printParity ဥပမာတွေကို ကြည့်ပါ။ ဒီစည်းမျဉ်းတွေကို ဘယ်အုပ်စုထဲသွင်းမလဲ

အခန်း(၅)

# အသီးအပွင့်ဝေဆာသော Method များ

**ပြန်ရသောတန်ဖိုးများ**

သင်္ချာ function တွေလို အရင်ကသုံးခဲ့ဖူးတဲ့ method အချို့ဟာ တန်ဖိုးတွေကို ထုတ်လုပ်ပါတယ်။ အဓိပ္ပာယ်က method ကို နှိုးဆွဖို့ တန်ဖိုးအသစ်တစ်ခုကို ရပါတယ်။ အဲဒီတန်ဖိုးကို variable တစ်ခုမှာ နေရာချထားတာဖြစ်ဖြစ်၊ ဖော်ပြချက်တစ်ခုမှာ ထည့်သုံးတာဖြစ်ဖြစ် လုပ်ပါတယ်။ ဥပမာ

```
double e = Math.exp(1.0);

double height = radius * Math.sin(angle);
```

ဒါပေမယ့် အခုအချိန်ထိ ရေးဖူးခဲ့တဲ့ method အားလုံးဟာ void method တွေ ဖြစ်နေပါတယ်။ ဆိုလိုရင်းက တန်ဖိုးပြန်မထုတ်ပေးတဲ့ method တွေ ဖြစ်နေခဲ့ပါတယ်။ void method တစ်ခုကို နှိုးဆွတဲ့အခါ သူ့ဘာသာသူ စာကြောင်းတစ်ကြောင်းမှာ နေပါတယ်။ နေရာချထားမှု လုပ်စရာမလိုပါဘူး။

```
nLines(3);

g.drawOval(0, 0, width, height);
```

ဒီအခန်းမှာ တန်ဖိုးတွေပြန်ထုတ်ပေးတဲ့ method တွေကို ရေးပါမယ်။ သူတို့ကို ခေါ်ရ

လွယ်အောင် အသီးအပွင့်ဝေဆာတဲ့ method တွေလို့ ခေါ်ရအောင်။ ပထမဆုံးဥပမာက area ဖြစ်ပါတယ်။ သူက double ကို parameter တစ်ခုအနေနဲ့ယူပြီး အဲဒီ အချင်းဝက်တန်ဖိုးရှိတဲ့ စက်ဝိုင်းရဲ့ဧရိယာကို ပြန်ထုတ်ပေးပါတယ်။

```
public static double area(double radius) {
        double area = Math.PI * radius * radius;
        return area;
}
```

အရင်ဆုံးသတိပြုသင့်တာက method အဓိပ္ပာယ်သတ်မှတ်ချက်ရဲ့အစဟာ အရင်မြင်ဖူးတာတွေနဲ့ ကွဲပြားနေပါတယ်။ Void method တစ်ခုကို ဖော်ညွှန်းတဲ့ public static void အစား public static double ကို တွေ့ရပါတယ်။ သူဆိုလိုတာက ဒီ method ကနေ ပြန်တဲ့တန်ဖိုးဟာ double အမျိုးအစား ရှိပါလိမ့်မယ်ဆိုပြီး ဖြစ်ပါတယ်။ public static က ဘာကိုဆိုလိုသလဲ ဆိုတာတော့ အခုထိ မရှင်းပြရသေးပါဘူး။ ဒီအတွက် စိတ်ရှည်ရှည်ထားပါ။

ပြီးတော့ return ဖော်ပြချက်ရဲ့ ပြန်တဲ့တန်ဖိုးပါတဲ့ ပုံစံကိုလည်း သတိပြုပါ။ ဒီဖော်ပြချက်ရဲ့ ဆိုလိုရင်းက "ဒီ method ချက်ချင်းတွက်ပါ။ ပြောထားတဲ့တန်ဖိုးကို ပြန်တဲ့တန်ဖိုးအနေနဲ့ သုံးပါ" ဆိုပြီးဖြစ်ပါတယ်။ ပြန်တဲ့တန်ဖိုးဟာ ရှုပ်ထွေးချင် ရှုပ်ထွေးနိုင်ပါတယ်။ ဒီတော့ ဒီ method ကို ပိုပြီး တိုတိုတုတ်တုတ်ဖြစ်အောင် ဒီလိုပြန်ပြင်ရေးနိုင်ပါတယ်။

```
public static double area(double radius) {
        return Math.PI * radius * radius;
}
```

အခြားတစ်ဖက်မှာတော့ area လို ယာယီ variable တွေဟာ အမှားပြင်ရ ပိုလွယ်စေပါလိမ့်မယ်။ ဘယ်လိုပဲလုပ်လုပ် return ဖော်ပြချက်ရဲ့ အမျိုးအစားဟာ method ကပြန်တဲ့ အမျိုးအစားနဲ့ ကိုက်ညီရပါမယ်။ နောက်တစ်မျိုးပြောရရင် ပြန်တဲ့တန်ဖိုးကို double လို့ ကြေညာခဲ့ရင် method ဟာ နောက်ဆုံးကျရင် double တန်ဖိုးတစ်ခုကို ပြန်ထုတ်ပေးမယ်လို့ ကတိပေးနေတာဖြစ်ပါတယ်။ return ဖော်ပြချက်မရေးမိရင် ဒါမှမဟုတ် အမျိုးအစား အမှားတစ်ခုကို ပြန်ပေးမိရင် compiler က ကိုယ့်ကို အလုပ်ပေးပါလိမ့်မယ်။

တစ်ခါတစ်ရံမှာ တန်ဖိုးပြန်ထုတ်ပေးတဲ့ ဖော်ပြချက်တွေကို အခြေအနေတစ်ခုရဲ့ လမ်းကြောင်းကွဲတစ်ခုစီမှာ ရေးသားရပါတယ်။

```
public static double absoluteValue(double x) {
        if (x < 0) {
                return -x;
        } else {
                return x;
        }
```

```
}
```

အခုတွေ့ခဲ့တဲ့ တန်ဖိုးပြန်ထုတ်တဲ့ ဖော်ပြချက်တွေဟာ ရွေးချယ်ပြလုပ်တဲ့ အခြေအနေထဲမှာ ရှိတဲ့အတွက် တစ်ခုတည်းကိုပဲ run ပါလိမ့်မယ်။ Method တစ်ခုထဲမှာ တရားဝင် return ဖော်ပြချက်တစ်ခုထက် ပိုရှိနိုင်ပေမယ့် တစ်ခုကို run ပြီးသွားရင် method ဟာ ကျန်တာတွေကို မ run ဘဲ ပြန်ထွက်သွားတယ်ဆိုတာကိုတော့ မှတ်ထားဖို့လိုပါတယ်။

return ဖော်ပြချက် နောက်ကလာတဲ့ code တွေကို ဘယ်တော့မှ မ run ဘဲ သူတို့ကို သေဆုံးနေတဲ့ code လို့ခေါ်ပါတယ်။ တချို့ compiler တွေဟာ ကိုယ့် code ရဲ့ အချို့အစိတ်အပိုင်းတွေဟာ သေဆုံးနေရင် သတိပေးပါလိမ့်မယ်။

တန်ဖိုးပြန်ထုတ်တဲ့ ဖော်ပြချက်တွေကို အခြေအနေတစ်ခုထဲမှာ ထားမယ်ဆိုရင် ပရိုဂရမ်ရဲ့ ဖြစ်နိုင်တဲ့လမ်းကြောင်းတိုင်းမှာ return ဖော်ပြချက်တစ်ခုပါအောင် ဂရုစိုက်ရပါမယ်။ ဥပမာ

```
public static double absoluteValue(double x) {
        if (x < 0) {
                return -x;
        } else if (x > 0) {
                return x;
        }                                   // WRONG!!
}
```

ဒီပရိုဂရမ်ဟာ တရားမဝင်ပါဘူး။ ဘာလို့လဲဆိုတော့ x ဟာ 0 ဖြစ်ခဲ့ရင် အခြေအနေနှစ်ခုစလုံး မမှန်ဘဲ method ဟာ ဒီအတိုင်းပြီးသွားပါမယ်။ တန်ဖိုးပြန်ထုတ်ပေးတဲ့ ဖော်ပြချက်တစ်ခုကိုမှ run မှာ မဟုတ်ပါဘူး။ Compile က ထုတ်ပေးမယ့် message ဟာ absoluteValue မှာ return ဖော်ပြချက် လိုအပ်တယ်လို့ပဲ ဖြစ်ပါတယ်။ ဒါဟာ မျက်စေ့လည်စရာ message တစ်ခုဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတော့ နှစ်ခုတောင် ပံ့ပိုးပေးပြီးသား ဖြစ်နေလို့ပါပဲ။

#### ပရိုဂရမ်ရေးသားမှု

ဒီနေရာမှာ ပြည့်စုံတဲ့ Java method တစ်ခုကိုကြည့်ပြီး သူ့ကို ဘာလုပ်လဲဆိုတာ ပြောနိုင်ပါပြီ။ ဒါပေမယ့် သူတို့ကို ဘယ်လိုရေးရမလဲဆိုတာ မသေမချာ ဖြစ်ချင်ဖြစ်နေပါမယ်။ အဆင့်ဆင့်ရေးသားမှုလို့ခေါ်တဲ့ နည်းစနစ်အသစ်ကို ဒီလိုဖြစ်နေသူတွေအတွက် ရှင်းပြပါမယ်။

ဥပမာတစ်ခုအနေနဲ့ အမှတ်နှစ်ခုကြားက အကွာအဝေးကို ရှာချင်တယ်လို့ ယူဆပါ။ (x1, y1) နဲ့ (x2, y2) အမှတ်တွေကြား အမှတ်တွေရဲ့ ကိုဩဒိနိတ်တွေကိုတော့ သိထားပါတယ်။ အမှတ်တွေကြားက အကွာအဝေးဟာ

$$\text{အကွာအဝေး} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

ရှိပါတယ်။

ပထမအဆင့် စဉ်းစားရမှာက Java မှာ distance ဆိုတဲ့ method ဟာ ဘယ်လိုပုံနဲ့ ရှိနေမလဲဆိုတာပဲ ဖြစ်ပါတယ်။ နောက်တစ်မျိုးဆိုရရင် ဘယ်ဟာတွေက parameter အနေနဲ့ လက်ခံတဲ့ input တွေဖြစ်ပြီး ဘယ်ဟာတွေက return တန်ဖိုးတွေအနေနဲ့ ပြန်ထုတ်ပေးတဲ့ output တွေဖြစ်မလဲဆိုတာပဲ ဖြစ်ပါတယ်။

ဒီနေရာမှာ အမှတ်နှစ်မှတ်ဟာ parameter တွေဖြစ်ပြီး သူတို့ကို double အမျိုးအစားရှိတဲ့ parameter အနေနဲ့ လက်ခံရင် သဘာဝကျပါမယ်။ နောက်ပိုင်းမှာတော့ Java မှာ Point ဆိုတဲ့ ဝတ္ထုတစ်ခုရှိတယ်ဆိုတာ မြင်ရပါမယ်။ အခုချိန်မှာတော့ အဆင်ပြေသလိုပဲ လုပ်ကြရအောင်။ ပြန်တဲ့တန်ဖိုးကတော့ double အမျိုးအစားရှိမယ့် အကွာအဝေးပဲ ဖြစ်ပါတယ်။

အခုကတည်းက method ရဲ့ အကြမ်းဖျင်းပုံစံကို ရေးနိုင်ပါပြီ။

```
public static double distance
            (double x₁, double y₁, double x₂, double y₂) {
    return 0.0;
}
```

return 0.0 ဆိုတာ ပရိုဂရမ်ကို compile လုပ်ဖို့ ခဏထားတဲ့ ဖော်ပြချက်ဖြစ်ပါတယ်။ မြင်ရသလိုပဲ ပရိုဂရမ်ဟာ အသုံးဝင်တဲ့အလုပ် တစ်ခုမှမလုပ်ပါဘူး။ ဒါပေမယ့် ဒီလိုလုပ်ကြည့်ပြီး compile လုပ်ထားရင် သဒ္ဒါတွေ ရှုပ်မလာခင် အမှားကြိုပြင်ထားနိုင်ပါတယ်။

Method အသစ်ကိုစမ်းဖို့ သူ့ကို နမူနာတန်ဖိုးတွေသုံးပြီး နှိုးဆွရပါတယ်။ main ရဲ့ တစ်နေရာမှာ အောက်မှာပြထားတာကို ပေါင်းထည့်ပါမယ်။

```
double dist = distance(1.0, 2.0, 4.0, 6.0);
```

ဒီတန်ဖိုးတွေ ရွေးရခြင်းအကြောင်းက ရေပြင်ညီအကွာအဝေးကို 3 ရှိစေပြီး ထောင်လိုက်အကွာအဝေးကို 4 ရှိအောင်ပါ။ ရလဒ်ဟာ 3-4-5 တြိဂံရဲ့ ထောင့်မှန်ခံအနား 5 ဖြစ်ပါလိမ့်မယ်။ Method တစ်ခုကို စမ်းသပ်ရာမှာ အဖြေမှန်ကိုသိထားရင် ပိုအဆင်ပြေစေပါတယ်။

Method အဓိပ္ပာယ်သတ်မှတ်ချက်ရဲ့ သဒ္ဒါကိုစစ်ပြီးရင် တစ်ကြိမ်မှာ code အနည်းငယ်ပေါင်းထည့်ပြီး စရေးနိုင်ပါပြီ။ အနည်းငယ်တိုးပြီးတာနဲ့ ပရိုဂရမ်ကို compile ပြန်လုပ်ပြီး run ပါမယ်။ ဒီလိုနည်းနဲ့ ဘယ်နေရာမှားနေသလဲဆိုတာ သိနိုင်ပါတယ်။ ထပ်ပေါင်းထည့်လိုက်တဲ့ နောက်ဆုံးစာကြောင်းမှာပါ။

နောက်တစ်ဆင့်က $x_2 - x_1$ နဲ့ $y_2 - y_1$ ခြားနားချက်တန်ဖိုးတွေကို တွက်ချက်ဖို့ဖြစ်ပါတယ်။ ရတဲ့တန်ဖိုးတွေကို dx နဲ့ dy ဆိုတဲ့ ခေတ္တ variable တွေထဲမှာ သိမ်းပါမယ်။

```
public static double distance
            (double x₁, double y₁, double x₂, double y₂) {
    double dx = x₂ - x₁;
```

```
    double dy = y2 - y1;
    System.out.println("dx is " + dx);
    System.out.println("dy is " + dy);
    return 0.0;
}
```

ရှေ့ဆက်မသွားခင် တန်ဖိုးတွေကို ချက်ချင်းစစ်ဆေးနိုင်အောင် print ဖော်ပြချက်တွေကို ပေါင်းထည့်ထားပါတယ်။ ပြောခဲ့ဖူးသလိုပဲ ရလဒ်တွေဟာ 3.0 နဲ့ 4.0 ဖြစ်ရပါမယ်။

Method ကိုရေးပြီးတဲ့အခါ print ဖော်ပြချက်တွေကို ဖျက်ပါမယ်။ ဒီလို code မျိုးကို ငြမ်းဆင်တယ်လို့ ခေါ်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူက ပရိုဂရမ်ရဲ့ အစိတ်အပိုင်းတည်ဆောက်ရာမှာ အသုံးဝင်ပေမယ့် နောက်ဆုံးရတဲ့ ပရိုဂရမ်ရဲ့ အစိတ်အပိုင်းတော့ မဟုတ်ပါဘူး။ တစ်ခါတစ်ရံမှာ ငြမ်းကို ဆက်ထားထားရင် အသုံးဝင်ပါတယ်။ မလိုရင် မှတ်ချက်အနေနဲ့ ဖျောက်လိုက်ပြီး လိုမှ ပြန်သုံးလို့ရပါတယ်။

ရေးသားမှုနောက်တစ်ဆင့်ကတော့ dx နဲ့ dy ကို နှစ်ထပ်ကိန်းတင်ဖို့ ဖြစ်ပါတယ်။ Math.pow method ကို သုံးချင်ရင် သုံးနိုင်ပါတယ်။ ဒါပေမယ့် အကွခရာကိန်းကို သူ့နဲ့ပဲ ပြန်မြှောက်ရင် ပိုရှင်းပြီး ပိုလွယ်ပါတယ်။

```
public static double distance
            (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx * dx + dy * dy;
    System.out.println("dsquared is " dsquared);
    return 0.0;
}
```

ပရိုဂရမ်ကို နောက်တစ်ခါ compile လုပ်ပြီး run ပါမယ်။ လက်ငင်းရတဲ့ တန်ဖိုးကိုလည်း စစ်ဆေးပါမယ်။ (တန်ဖိုးဟာ 25.0 ဖြစ်သင့်ပါတယ်။)

နောက်ဆုံးအနေနဲ့ Math.sqrt method ကိုသုံးပြီး ရလဒ်ကို တွက်ထုတ်ပါမယ်။ ပြီးရင် အဲဒီတန်ဖိုးကို ပြန်ပါမယ်။

```
public static double distance
            (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
```

```
double dy = y2 - y1;
double dsquared = dx * dx + dy * dy;
double result = Math.sqrt(dsquared);
return result;
}
```

ပြီးရင် main မှာ ရလဒ်ရဲ့တန်ဖိုးကို print လုပ်ပြီး စစ်ဆေးသင့်ပါတယ်။

ပရိုဂရမ်ရေးတဲ့အတွေ့အကြုံ ရလာတာနဲ့အမျှ တစ်ချိန်မှာ စာကြောင်းတစ်ကြောင်းထက်ပိုပြီး ရေးနေတာ အမှားပြင်နေတာကို သတိပြုမိပါလိမ့်မယ်။ ဒါတောင်မှ ဒီလို အဆင့်ဆင့်ရေးသားတဲ့ဖြစ်စဉ်ဟာ အမှားပြင်ရတဲ့အချိန်ကို သက်သာစေတာ တွေ့ရပါမယ်။ ဒီဖြစ်စဉ်ရဲ့ အခရာကျတဲ့ အချက်တွေကတော့

- အလုပ်ဖြစ်တဲ့ ပရိုဂရမ်တစ်ပုဒ်နဲ့စပြီး သေးငယ်တဲ့ပြုပြင်မှုတွေကို တစ်ဆင့်ချင်းလုပ်ပါ။ တစ်နေရာမှာ အမှားလုပ်မိရင် ဘယ်နေရာမှာလဲလို့ တန်းသိပါလိမ့်မယ်။
- လက်တလောရလာတဲ့ တန်ဖိုးတွေကိုသိမ်းဖို့ ယာယီ variable တွေကိုသုံးပြီး သူတို့ကို print လုပ်နိုင်၊ စစ်ဆေးနိုင်အောင်လုပ်ပါ။
- ပရိုဂရမ်ဟာ အလုပ်ဖြစ်နေပြီဆိုတာနဲ့ ငြမ်းဖျက်တာတွေ၊ ဖော်ပြချက်တွေကို တစ်ခုအနေနဲ့ ပေါင်းတာတွေကို ပြုလုပ်နိုင်ပါတယ်။ ပရိုဂရမ်ကိုတော့ ဖတ်ရလွယ်အောင် လုပ်ထားပါ။

**ဖွဲ့စည်းခြင်း**

Method အသစ်တစ်ခုကို သတ်မှတ်ပြီးနောက်မှာ သူ့ကို ဖော်ပြချက်တစ်ခုရဲ့ အစိတ်အပိုင်းအနေနဲ့ဖြစ်ဖြစ်၊ လက်ရှိ method တွေသုံးပြီး method အသစ်တွေ တည်ဆောက်ရာမှာဖြစ်ဖြစ် သုံးနိုင်တယ်ဆိုတာ သိပြီးဖြစ်ပါလိမ့်မယ်။ ဥပမာ အမှတ်နှစ်မှတ်ကို စက်ဝိုင်းတစ်ခုရဲ့ ဗဟိုနဲ့ စက်ဝန်းပေါ်က အမှတ်တစ်ခုအဖြစ် ပေးလာတယ်ဆိုရင် စက်ဝိုင်းရဲ့ဧရိယာကို ဘယ်လိုရှာမလဲ။

ဗဟိုမှတ်ကို $x_1$ နဲ့ $y_1$ variable တွေမှာ သိမ်းထားတယ်လို့ ယူဆပါ။ စက်ဝန်းမှတ်ကို $x_p$ နဲ့ $y_p$ မှာ ရှိတယ်လို့ထားပါ။ ပထမအဆင့်က စက်ဝိုင်းရဲ့အချင်းဝက်ကို ရှာဖို့ပါပဲ။ အမှတ်နှစ်ခုကြားက အကွာအဝေးကိုရှာရင် သူ့ကိုရပါတယ်။ ကံကောင်းထောက်မစွာနဲ့ပဲ အဲဒီအလုပ်ကိုလုပ်တဲ့ distance ဆိုတဲ့ method တစ်ခု ရှိနေပါတယ်။

```
double radius = distance(x1, y1, xp, yp);
```

ဒုတိယအဆင့်က အဲဒီ အချင်းဝက်ရှိတဲ့ စက်ဝိုင်းရဲ့ ဧရိယာကိုရှာပြီး သူ့ကိုတန်ဖိုးပြန်ထွက်ပေးဖို့ပါပဲ။

```
double area = area(radius);
```

```
return area;
```

ဒါတွေအားလုံးကို method တစ်ခုတည်းမှာ ထည့်ထားရင် ဒီလိုရပါတယ်။

```
public static double fred
        (double x1, double y1, double xp, double yp) {
    double radius = distance(x1, y1, xp, yp);
    double area = area(radius);
    return area;
}
```

ဒီ method ရဲ့ နာမည်ဟာ fred ဖြစ်ပြီး နည်းနည်းထူးဆန်းနေပါတယ်။ ဘာလို့လဲဆိုတာ ရှေ့အပိုင်းရောက်ရင် သိပါလိမ့်မယ်။

radius နဲ့ area ဆိုတဲ့ ခေတ္တ variable တွေဟာ ရေးသားမှုနဲ့ အမှားပြင်ရာမှာ အသုံးဝင်ပါတယ်။ ဒါပေမယ့် ပရိုဂရမ်အလုပ်လုပ်နေပြီဆိုတာနဲ့ method ခေါ်ဆိုမှုတွေကို ထပ်ဆင့်ဖွဲ့စည်းပြီး ပရိုဂရမ်ကို ကျစ်ကျစ်လျစ်လျစ်ရှိအောင် လုပ်နိုင်ပါတယ်။

```
public static double fred
        (double x1, double y1, double xp, double yp) {
    return area(distance(x1, y1, xp, yp));
}
```

## Overload လုပ်ခြင်း

ပြီးခဲ့တဲ့အပိုင်းမှာ fred နဲ့ area ဟာ ဆင်တူတဲ့ အလုပ်တွေကို လုပ်ဆောင်တယ်ဆိုတာ သတိပြုမိခဲ့ကြပါလိမ့်မယ်။ ကွဲပြားတဲ့ parameter တွေကိုလက်ခံပြီး စက်ဝိုင်းရဲ့ဧရိယာကို ရှာပါတယ်။ area အတွက် အချင်းဝက်ကို ထည့်ပေးရပြီး fred အတွက် အမှတ်နှစ်ခုကို ပံ့ပိုးပေးရပါတယ်။

Method နှစ်ခုဟာ အလုပ်တစ်ခုတည်းကို လုပ်တယ်ဆိုရင် သူတို့ကို နာမည်အတူတူပေးရင် သဘာဝကျပါလိမ့်မယ်။ နောက်တစ်မျိုးပြောရရင် fred ကို area လို့ခေါ်ရင် အဓိပ္ပာယ်ပိုရှိပါလိမ့်မယ်။

Overloading လို့ခေါ်တဲ့ နာမည်တစ်ခုတည်းရှိတဲ့ method တစ်ခုထက်ပိုရှိတာဟာ Java မှာ method တစ်ခုစီက ကွဲပြားတဲ့ parameter တွေကို လက်ခံနေသရွေ့ တရားဝင်ပါတယ်။ ဒီတော့ fred ကို နာမည်ပြင်ပေးနိုင်ပါတယ်။

```
public static double area
        (double x1, double y1, double x2, double y2) {
```

```
    return area(distance(x_1, y_1, x_p, y_p));
}
```

Overload လုပ်ထားတဲ့ method တစ်ခုကို နှိုးဆွတဲ့အခါ Java ဟာ ဘယ် method ကိုခေါ်ယူရမလဲဆိုတာ ပံ့ပိုးပေးထားတဲ့ argument တွေကိုကြည့်ပြီး သိပါတယ်။

```
double x = (3.0);
```

ဒီလိုခေါ်ဆိုမှုတစ်ခုကို ပြုလုပ်တဲ့အခါ Java ဟာ double တန်ဖိုးတစ်ခုကို argument တစ်ခုအနေနဲ့ ပံ့ပိုးတဲ့ area method ကို ရှာဖွေပါတယ်။ ဒီတော့ argument ကို အချင်းဝက်အနေနဲ့သုံးတဲ့ ပထမဆုံး method ကို သုံးပါတယ်။

```
double x = area(1.0, 2.0, 4.0, 6.0);
```

ဒီလိုခေါ်ဆိုမှုမျိုးအတွက် area ရဲ့ ဒုတိယပုံစံကို သုံးပါလိမ့်မယ်။ ပိုပြီးအံ့သြစရာကောင်းတာက ဒုတိယ area method ဟာ ပထမ method ကို နှိုးဆွပါတယ်။

Java ရဲ့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ command တော်တော်များများကို overload လုပ်ထားပါတယ်။ အဓိပ္ပာယ်က နာမည်တူပေမယ့် ကွဲပြားတဲ့ parameter အရေအတွက်နဲ့ အမျိုးအစားတွေကို လက်ခံတဲ့ method အသွင်ကွဲတွေ ရှိပါတယ်။ ဥပမာ ဘယ်အမျိုးအစားရှိတဲ့ parameter တစ်ခုကိုမဆို လက်ခံတဲ့ print နဲ့ println method အသွင်ကွဲတွေ ရှိပါတယ်။ သင်္ချာ class မှာ double တွေနဲ့ int တွေပေါ်မှာ လုပ်ကိုင်တဲ့ abs method တွေရှိပါတယ်။

Overload လုပ်တာဟာ အသုံးဝင်တဲ့ လက္ခဏာတစ်ခုဖြစ်ပေမယ့် သူ့ကို သတိနဲ့သုံးသင့်ပါတယ်။ Method တစ်ခုကို အမှားပြင်ဖို့ ကြိုးစားနေတုန်း သူနဲ့နာမည်တူတဲ့ အခြား method တစ်ခုနဲ့ ကျကျနနကြီး သွားမှားတတ်ပါတယ်။

ဒါဟာ အမှားပြင်တာရဲ့ စည်းမျဉ်းတစ်ခုကို အမှတ်ရစေပါတယ်။ ကိုယ်ကြည့်နေတဲ့ ပရိုဂရမ်ဟာ ကိုယ် run ဖို့ ကြိုးစားနေတဲ့ ပရိုဂရမ်ဖြစ်သလားဆိုတာ စစ်ဆေးပါ။ တစ်ခါတစ်ရံမှာ ကိုယ့်ပရိုဂရမ်ထဲမှာ အပြောင်းအလဲတွေ တစ်ခုပြီးတစ်ခု လုပ်နေပေမယ့် run လိုက်တဲ့အခါတိုင်း ဒါကိုပဲ ပြန်ပြန်မြင်နေရတာမျိုးတွေ ဖြစ်တတ်ပါတယ်။ ဒါဟာ ကိုယ် run နေတဲ့ ပရိုဂရမ်က ကိုယ်ထင်တဲ့ ပရိုဂရမ်မဟုတ်ဘူးဆိုတဲ့ သတိပေးချက်ပါပဲ။ သိချင်ရင် print ဖော်ပြချက်တစ်ခုကို ထည့်ကြည့်ပါ။ (ဘာကို print လုပ်လုပ် အရေးမကြီးပါဘူး။) ပရိုဂရမ်ရဲ့အပြုအမူ လိုက်ပြောင်းသလားဆိုတာ ကြည့်ပါ။

## Boolean ဖော်ပြချက်များ

မြင်ခဲ့တဲ့ လုပ်ဆောင်ချက်တော်တော်များများဟာ သူတို့ရဲ့ operand တွေနဲ့ အမျိုးအစားတူတဲ့ ရလဒ်တွေကို ထုတ်ပေးပါတယ်။ ဥပမာ + လက္ခဏာဟာ int နှစ်ခုကိုယူပြီး int တစ်ခုကို ပြန်ထုတ်ပေးပါတယ်။ ဒါမှမဟုတ် double နှစ်ခုယူပြီး double တစ်ခုကို ထုတ်ပေးပါတယ်။

တွေ့ခဲ့ရသမျှထဲမှာ ခြွင်းချက်တွေဟာ နှိုင်းရလက္ခဏာတွေ ဖြစ်ပါတယ်။ သူတို့ဟာ int တွေ၊ float တွေကို လက်ခံပြီး true ဒါမှမဟုတ် false တန်ဖိုးတွေကို ထုတ်ပေးပါတယ်။ true နဲ့ false ဟာ Java မှာ အထူးတန်ဖိုးတွေ ဖြစ်ပါတယ်။ သူတို့နှစ်ခုဟာ boolean လို့ ခေါ်တဲ့ အမျိုးအစားတစ်ခု ဖြစ်ပါတယ်။ အမျိုးအစား (type) ကို အဓိပ္ပာယ်သတ်မှတ်တုန်းက တန်ဖိုးအုပ်စုတစ်စုလို့ သတ်မှတ်ခဲ့ပါတယ်။ int တွေ၊ double တွေ၊ String တွေမှာတုန်းက ဒီအုပ်စုတွေဟာ အရွယ်နည်းနည်းကြီးပါတယ်။ boolean တွေအတွက်တော့ ဒီလောက်မကြီးပါဘူး။

boolean ဖော်ပြချက်တွေနဲ့ variable တွေဟာ တခြား ဖော်ပြချက်အမျိုးအစားတွေနဲ့ variable တွေလိုပဲ လုပ်ကိုင်ပါတယ်။

```
boolean fred;
fred = true;
boolean testResult = false;
```

ပထမဆုံးဥပမာဟာ variable ကြေညာချက်ဖြစ်ပါတယ်။ ဒုတိယဥပမာကတော့ နေရာချထားမှု ဖြစ်ပါတယ်။ တတိယဥပမာကတော့ ကြေညာတာနဲ့ နေရာချထားတာကို ပေါင်းထားတာဖြစ်ပါတယ်။ သူ့ကို စတင်တာ (intialisation) လို့လည်း ခေါ်ပါသေးတယ်။ တန်ဖိုးတွေဖြစ်ကြတဲ့ true နဲ့ false ဟာ Java မှာ keyword တွေဖြစ်ကြပြီး ပရိုဂရမ်းမင်း ပတ်ဝန်းကျင်အလိုက် အရောင်အမျိုးမျိုးနဲ့ ပေါ်ပါလိမ့်မယ်။

ခုနကပြောခဲ့သလိုပဲ အခြေအနေလက္ခဏာတစ်ခုကနေ ရတဲ့တန်ဖိုးဟာ boolean တစ်ခု ဖြစ်ပါတယ်။ ဒီတော့ နှိုင်းယှဉ်မှုတစ်ခုရဲ့ ရလဒ်ကို variable တစ်ခုမှာ သိမ်းနိုင်ပါတယ်။

```
boolean evenFlag = (n%2 == 0); //true if n is even
boolean positionflag = (x > 0); //true if x is positive
```

နောက်ပိုင်းမှာသူ့ကို အခြေအနေမှီခိုတဲ့ ဖော်ပြချက်တစ်ခုမှာ ထည့်သုံးနိုင်ပါတယ်။

```
if (evenFlag) {
    System.out.println("n was even when I checked it");
}
```

### ယုတ္တိလက္ခဏာများ

Java မှာ ယုတ္တိလက္ခဏာ သုံးခုရှိပါတယ်။ သူတို့ဟာ AND, OR နဲ့ NOT ဖြစ်ပြီး &&, || နဲ့ ! လက္ခဏာတွေနဲ့ ကိုယ်စားပြုပါတယ်။ ဒီလက္ခဏာတွေရဲ့အဓိပ္ပာယ်ကို စဉ်းစားယူလို့ ရပါတယ်။ ဥပမာ x > 0 && x < 0 ဟာ x က သုညထက်ငယ်ပြီး 10 ထက်ကြီးမှ မှန်ပါတယ်။ နှစ်ခုစလုံးမှန်မှ မှန်မယ်လို့ ဆိုလိုပါတယ်။

evenFlag || n%3 == 0 ဟာ အခြေအနေနှစ်ခုထဲက တစ်ခုမှန်ရင် မှန်ပါတယ်။ အဓိပ္ပာယ်က evenFlag က မှန်ရင်မှန် မဟုတ်ရင် အဲဒီကိန်းကို 3 နဲ့ စားလို့ပြတ်ရင် ဖော်ပြချက်အပြည့်

အစုံက မှန်ရမယ်လို့ ဆိုလိုပါတယ်။

နောက်ဆုံးအနေနဲ့ NOT လက္ခဏာဟာ boolean ဖော်ပြချက်တစ်ခုကို ပြောင်းပြန်လှန်နိုင်စွမ်း ရှိပါတယ်။ ဒီတော့ evenFlag ဟာ မှားတယ်ဆိုရင် !evenFlag က မှန်ပါတယ်။

ယုတ္တိလက္ခဏာတွေဟာ ထပ်ဆင့်ထားတဲ့ အခြေအနေဖော်ပြချက်တွေကို ရှင်းလင်းအောင် လုပ်နိုင်ပါတယ်။ ဥပမာ အောက်က code ကို အခြေအနေတစ်ခုတည်းပါအောင် ဘယ်လို ပြန်ရေးမလဲ။

```
if (x > 0) {
    if (x < 10) {
        System.out.println("x is a positive single digit.");
    }
}
```

ကျွန်တော်ကတော့ ဒီလိုပြင်ရေးလိုက်ပါတယ်။ ပိုကောင်းတာတွေ့ရင် စာရေးပြီး ပြောပါဦး။

```
if (x > 0 && x < 10) {
    System.out.println("x is a positive single digit.");
}
```

## Boolean Method များ

Method တွေဟာ boolean တန်ဖိုးတွေကိုလည်း တခြားတန်ဖိုးတွေပြန်သလို ပြန်နိုင်ပါတယ်။ ရှုပ်ထွေးတဲ့ စစ်ဆေးမှုတွေကို method တွေထဲမှာ ဝှက်ထားချင်ရင် ဒါဟာအဆင်ပြေစေပါတယ်။ ဥပမာ

```
public static boolean isSingleDigit(int x) {
    if (x >= 0 && x < 10) {
        return true;
    } else {
        return false;
    }
}
```

ဒီ method ရဲ့ နာမည်ဟာ isSingleDigit ဖြစ်ပါတယ်။ Boolean method နာမည်တွေကို ဟုတ်တယ်၊ မဟုတ်ဘူး (Yes/No) အဖြေတွေ ရရှိစေတဲ့ မေးခွန်းတွေ ပုံစံထားလေ့ ရှိပါတယ်။ ပြန်တဲ့တန်ဖိုး အမျိုးအစားဟာ boolean ဖြစ်ပါတယ်။ ဆိုလိုတာက ပြန်တဲ့ဖော်ပြချက်တိုင်းဟာ boolean ဖော်ပြချက်တစ်ခုကို ပံ့ပိုးပေးရပါတယ်။

ဒီ code ကိုယ်တိုင်ကိုက လိုရင်းရောက်ပါတယ်။ ဒါပေမယ့်သူက လိုအပ်တာထက် နည်းနည်းပိုရှည်နေပါတယ်။ x >= 0 && x < 10 ဆိုတဲ့ ဖော်ပြချက်ဟာ boolean အမျိုးအစားရှိတယ်ဆိုတာ သတိရပါ။ ဒီတော့ သူ့ကို တိုက်ရိုက်ပြန်ပြီး if ဖော်ပြချက်ကို လုံးဝရှောင်လို့ရပါတယ်။

```
public static boolean isSingleDigit(int x) {
   return (x >= 0 && x < 10);
}
```

main မှာ ဒီ method ကို အရင်လို နှိုးဆွနိုင်ပါတယ်။

```
boolean bigFlag = !isSingleDigit(17);
System.out.println(isSingleDigit(2));
```

ပထမဖော်ပြချက်ဟာ 17 က ဂဏန်းတစ်ခုပါတဲ့ ကိန်းမဟုတ်ရင် bigFlag မှာ true တန်ဖိုးကို နေရာချထားပါတယ်။ 2 ဟာ ဂဏန်းတစ်လုံးတည်းပါတဲ့ ကိန်းတစ်ခုဖြစ်တဲ့အတွက် ဒုတိယစာကြောင်းက true ကို print လုပ်ပါတယ်။ println ကို boolean တွေ ကိုင်တွယ်နိုင်အောင်လည်း overload ထားတာကို တွေ့ရပါမယ်။

Boolean method တွေ အသုံးအဝင်ဆုံး နေရာကတော့ အခြေအနေဖော်ပြချက်တွေမှာဖြစ်ပါတယ်။

```
if (isSingleDigit(x)) {
   System.out.println("x is little");
} else {
   System.out.println("x is big");
}
```

## ...ပ်တာလဲလဲဖြစ်စဉ်မှာ တစ်ဝဲဝဲလည်

အခုချိန်မှာ တန်ဖိုးတွေပြန်တဲ့ method တွေ ရေးနိုင်ပြီဖြစ်တဲ့အတွက် ပြည့်စုံတဲ့ ပရိုဂရမ်...ဘာသာစကားတစ်ခုကို ရရှိပြီလို့သိရရင် စိတ်ဝင်စားကြမယ် ထင်ပါတယ်။ ဆိုလိုတာက တွက်...ျက်လို့ရတဲ့ ဘယ်အရာကိုမဆို ဒီဘာသာစကားမှာ ဖော်ပြနိုင်ပါတယ်။ ရေးလို့ရသမျှ ပရိုဂရမ်...ိုင်းကို အခုထိသုံးခဲ့တဲ့ ဘာသာစကားလက္ခဏာတွေသုံးပြီး ပြန်ရေးနိုင်ပါတယ်။ (keyword, ...ouse, disk တွေ စတာတွေကို ထိန်းချုပ်တဲ့ command အနည်းတော့ လိုပါလိမ့်မယ်။)

ဒီလိုကြွေးကြော်မှုကို မှန်ကြောင်းသက်သေပြတာဟာ လွယ်တဲ့အလုပ်တော့ မဟုတ်ပါဘူး။ ...ကို အစောပိုင်း ကွန်ပျူတာပညာရှင်တစ်ဦးဖြစ်တဲ့ Alan Turing က အရင်ဆုံး စလုပ်ခဲ့ပါ...ယ်။ (တချို့က သူဟာ သင်္ချာပညာရှင်တစ်ဦးလို့ စောဒကတက်ကြပါလိမ့်မယ်။ ဒါပေမယ့် ...နဦး ကွန်ပျူတာပညာရှင် တော်တော်များများဟာ သင်္ချာသမားအဖြစ်နဲ့ အလုပ်စလုပ်ခဲ့ကြ...

ပါတယ်။) ဒီသက်သေပြချက်ကို Turing စာတမ်းလို့ ခေါ်ပါတယ်။

အခုထက်ထိ လေ့လာခဲ့တဲ့ ကိရိယာတန်ဆာပလာတွေနဲ့ ဘာလုပ်နိုင်သလဲဆိုတာ မြင်သာအောင် ထပ်တလဲလဲသတ်မှတ်ထားတဲ့ သင်္ချာ function တချို့ကို တွက်ထုတ်ပေးတဲ့ method တွေကို ကြည့်ရအောင်။ ထပ်တလဲလဲသတ်မှတ်ချက်ဟာ ပတ်လည်သတ်မှတ်ချက်တစ်ခုနဲ့ တူပါတယ်။ သတ်မှတ်ချက်မှာ သတ်မှတ်ခံရတဲ့အရာဆီကို ပြန်ညွှန်းပါတယ်။ အပြည့်အဝ ပတ်လည်ရိုက်နေတဲ့ အဓိပ္ပာယ်သတ်မှတ်ချက်တစ်ခုဟာ သိပ်တော့အသုံးမဝင်တတ်ပါဘူး။ ဥပမာ အီးရောင်ဝါးဆိုတာ အီးရောင်ဝါးဖြစ်တဲ့အရာတစ်ခုကို ဖော်ပြတဲ့ နာမဝိသေသနတစ်ခု ဖြစ်ပါတယ်။

ဒီလို အဓိပ္ပာယ်ဖွင့်ဆိုချက်မျိုးကို အဘိဓာန်တစ်ခုထဲမှာ သွားတွေ့မိရင် တော်တော်စိတ်ညစ်သွားပါလိမ့်မယ်။ အခြားတစ်ဖက်မှာတော့ factorial ဆိုတဲ့ သင်္ချာ function ရဲ့ အဓိပ္ပာယ်ဖွင့်ဆိုချက်ကိုကြည့်ရင် ဒီလိုတွေ့ရပါမယ်။

```
0! = 1
n! = n·(n-1)!
```

(Factorial ကို ! သင်္ကေတနဲ့ ပြသတတ်ပါတယ်။ Java ရဲ့ ယုတ္တိလက္ခဏာ ! နဲ့ မမှားပါနဲ့။) ဒီအဓိပ္ပာယ်ဖွင့်ဆိုချက်က ပြောတာက 0 ရဲ့ factorial ဟာ 1 ဖြစ်ပြီး n နဲ့ကိုယ်စားပြုတဲ့ အခြားတန်ဖိုးတစ်ခုရဲ့ factorial ဟာ n ဟာ n ထက် 1 လျော့တဲ့ n-1 ရဲ့ factorial နဲ့ မြှောက်ထားတာ ဖြစ်ပါတယ်။ ဒီတော့ 3! ဟာ 3 x 2! ဖြစ်ပြီး၊ 2! ဟာ 2 x 1! ဖြစ်ပါတယ်။ ဒါဟာ 1 x 0! ဖြစ်ပါတယ်။ အားလုံးကို တွဲရေးလိုက်ရင် 3! ဟာ 3 x 2 x 1 x 1 နဲ့ညီပြီး 6 ရပါတယ်။

ပုစ္ဆာတစ်ပုဒ်ရဲ့ ထပ်တလဲလဲလည်တဲ့ အဓိပ္ပာယ်ဖွင့်ဆိုချက်တစ်ခုကို ရေးနိုင်ရင် သူ့ကိုတွက်ဖို့ Java ပရိုဂရမ်တစ်ပုဒ်ကို ရေးသားနိုင်ပြီဖြစ်တယ်။ ပထမအဆင့်က ဒီ function အတွက် parameter တွေက ဘာလဲဆိုတာနဲ့ ပြန်တဲ့အမျိုးအစားက ဘာလဲဆိုတာ ဖြစ်ပါတယ်။ နည်းနည်းတွေးလိုက်ရင် factorial ဟာ ကိန်းပြည့်တစ်ခုကို parameter တစ်ခုအနေနဲ့ယူပြီး ကိန်းပြည့်တစ်ခုကို ပြန်ထုတ်ပေးတယ်ဆိုတာ သိနိုင်ပါတယ်။

```
public static int factorial(int n) {
}
```

Argument ဟာ သုညဖြစ်နေရင် 1 ပြန်ပေးရပါမယ်။

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
}
```

မဟုတ်ရင် ဒါဟာ စိတ်ဝင်စားစရာကောင်းတဲ့ အပိုင်းဖြစ်ပြီး n-1 ရဲ့ factorial ကိုရှာဖို့ ခေါ်ဆိုမှုကို အထပ်ထပ်လုပ်ရပါမယ်။ ပြီးရင် သူ့ကို n နဲ့ မြှောက်ရပါမယ်။

```
public static int factorial(int n) {
    if (n == 0) {
            return 1;
    } else {
int recurse = factorial(n-1);
        return result;
    }
}
```

ဒီပရိုဂရမ်ရဲ့ run တဲ့လမ်းကြောင်းကိုကြည့်ရင် ပြီးခဲ့တဲ့အခန်းက nLines နဲ့ တူပါတယ်။ factorial ကို တန်ဖိုး 3 နဲ့စပြီး နှိုးဆွပါတယ်။

3 ဟာ သုညမဟုတ်တဲ့အတွက် ဒုတိယလမ်းကြောင်းကိုလိုက်ပြီး n-1 ရဲ့ factorial ကို တွက်ပါတယ်။

2 ကို သုညမဟုတ်တဲ့အတွက် ဒုတိယလမ်းကြောင်းကိုလိုက်ပြီး n-1 ရဲ့ factorial ကို တွက်ပါတယ်။

1 ဟာ သုညမဟုတ်လို့ ဒုတိယလမ်းကြောင်းကိုလိုက်ပြီး n-1ရဲ့ factorial ကို တွက်ပါတယ်။

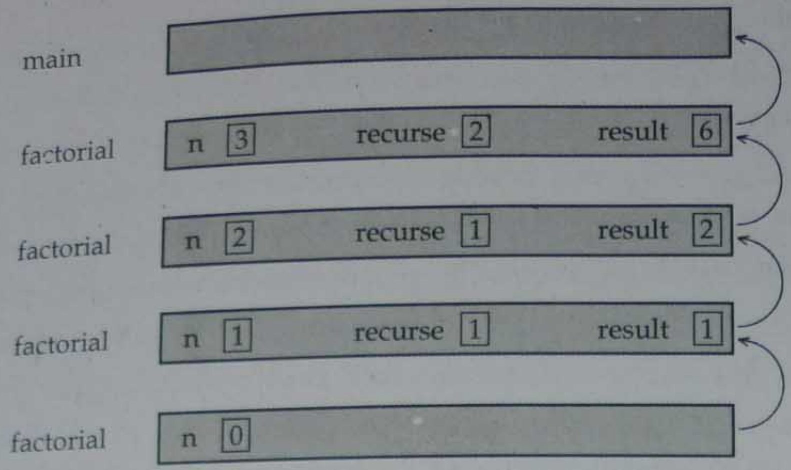
e ဟာ သုညဖြစ်တဲ့အတွက် ပထမလမ်းကြောင်းကိုလိုက်ပြီး တန်ဖိုး 1 ကို နောက်ထပ်ခေါ်ဆိုမှုတွေ ဆက်မလုပ်တော့ဘဲ ထုတ်ပေးလိုက်ပါတယ်။

ပြန်တဲ့တန်ဖိုး 1 ကို တန်ဖိုး 1 ရှိနေတဲ့ n နဲ့ မြှောက်ပြီး ရလဒ်ကို ပြန်ထုတ်ပေးပါတယ်။

ပြန်တဲ့တန်ဖိုး 1 ကို တန်ဖိုး 2 ရှိတဲ့ n နဲ့ မြှောက်ပြီး ရလဒ်ကို ပြန်ထုတ်ပေးပါတယ်။

ပြန်တဲ့တန်ဖိုး 2 ကို တန်ဖိုး 3 ရှိတဲ့ n နဲ့ မြှောက်ပါတယ်။ ရလဒ်ဟာ 6 ဖြစ်ပြီး သူ့ကို နှိုးဆွတဲ့နေရာကို ပြန်ပေးပါတယ်။

အောက်မှာပြထားတာက ဒီ function ခေါ်ဆိုမှု အစီအစဉ်အတွက် stack ပုံဖြစ်ပါတယ်။




ပြန်တဲ့တန်ဖိုးတွေကို stack အဆင့်ဆင့် ပို့ပေးတဲ့အနေနဲ့ ပြထားပါတယ်။

`factorial` ရဲ့ နောက်ဆုံးအဆင့်မှာ `recurse` နဲ့ `result` ဆိုတဲ့ ဒေသခံ variable တွေ မရှိပါဘူး။ ဘာလို့လဲဆိုတော့ n=0 ဖြစ်တဲ့အခါ သူတို့ကိုဖန်တီးတဲ့ လမ်းကြောင်းကို မရွေးချယ်လို့ဘဲ ဖြစ်ပါတယ်။

### ယုံကြည်မှုဖြင့် ခုန်ကျော်ခြင်း

ပရိုဂရမ် run တဲ့ လမ်းကြောင်းကို လိုက်တာဟာ ပရိုဂရမ်တွေဖတ်တဲ့ နည်းတစ်နည်းဖြစ်ပါတယ်။ ဒါပေမယ့် ပြီးခဲ့တဲ့အပိုင်းမှာ မြင်ခဲ့သလိုပဲ ဒါဟာ အလွယ်တကူ ဝက်ဘာထဲရောက်သလို ဖြစ်သွားနိုင်ပါတယ်။ ဖတ်နည်းနောက်တစ်နည်းက ယုံကြည်မှုရှိရှိနဲ့ ခုန်ကျော်သွားတာပဲ ဖြစ်ပါတယ်။ Method နှိုးဆွမှုတစ်ခုကို တွေ့တဲ့အခါ ပရိုဂရမ် run တဲ့လမ်းကြောင်းကို လိုက်မယ့်အစား အဲဒီ method ဟာ မှန်မှန်ကန်ကန်အလုပ်လုပ်ပြီး သင့်တော်တဲ့ တန်ဖိုးကို ပြန်ပေးတယ်လို့ ယူဆဖို့ပါပဲ။

တကယ်တမ်းကျတော့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ method တွေသုံးတဲ့အခါ ဒီလို ယုံကြည်တဲ့ ခုန်ကျော်မှုမျိုးကို လုပ်ဖူးပြီးသား ဖြစ်နေပါတယ်။ `Math.cos` တို့၊ `drawOval` တို့ကို နှိုးဆွတဲ့အခါ ဒီ method တွေကို ဘယ်လိုအကောင်အထည်ဖော်ထားသလဲဆိုတာ မစစ်ဆေးပါဘူး။ သူတို့ အလုပ်ကောင်းကောင်းလုပ်တယ်လို့ပဲ ယူဆလိုက်ပါတယ်။ ဘာလို့လဲဆိုတော့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ class တွေကို ရေးတဲ့သူတွေက ပရိုဂရမ်မာကောင်းတွေ ဖြစ်ကြလို့ပါပဲ။

ပြောရရင် ကိုယ့် method တွေကို နှိုးဆွတဲ့အခါမှာလည်း ဒါဟာ မှန်ပါတယ်။ ဥပမာ အရင်က အပိုင်းတစ်ပိုင်းမှာ ကိန်းတစ်ခုဟာ 0 နဲ့ 4 ကြားရှိမရှိ စစ်ဆေးတဲ့ `isSingleDigit` method ကို ရေးခဲ့ပါတယ်။ စမ်းသပ်ချက်နဲ့ code ကို စစ်ဆေးမှုအပြီးမှာ method ဟာ မှန်တယ်လို့ ယုံကြည်ပြီဆိုတာနဲ့ ဒီ code ကို ထပ်ကြည့်စရာမလိုဘဲ method ကို အသုံးပြုနိုင်ပါတယ်။

ထပ်တလဲလဲ အလုပ်တွေလုပ်ကိုင်တဲ့ ပရိုဂရမ်တွေအတွက်လည်း ဒါဟာ မှန်ပါတယ်။ ထပ်တလဲလဲ ခေါ်ဆိုမှုနေရာကိုရောက်ရင် ပရိုဂရမ် run တဲ့လမ်းကြောင်းကို လိုက်မယ့်အစား ထပ်တလဲလဲခေါ်ဆိုမှုဟာ အလုပ်ဖြစ်တယ်၊ မှန်ကန်တဲ့ရလဒ်ကို ထုတ်ပေးတယ်လို့ ယူဆပြီး "n-1 ရဲ့ factorial ကို ရှာနိုင်ရင် n ရဲ့ factorial ကိုလည်း ရှာနိုင်မလား"လို့ ကိုယ့်ကိုကိုယ် ပြန်မေးသင့်ပါတယ်။

ရေးလို့မပြီးသေးခင် method ကို မှန်တယ်လို့ ယူဆရတာ နည်းနည်းတော့ အူကြောင်ကြောင်နိုင်ပါတယ်။ ဒါပေမယ့် အဲဒါကြောင့်ပဲ ယုံကြည်မှုနဲ့ ခုန်ကျော်တယ်လို့ ပြောတာမဟုတ်လား။

### နောက်ထပ်ဥပမာတစ်ခု

ပြီးခဲ့တဲ့ဥပမာမှာတုန်းက အဆင့်တွေမြင်အောင်နဲ့ အမှားပြင်ရလွယ်အောင် ကြားခံ variable တွေ သုံးခဲ့ပါတယ်။ ဒါပေမယ့် စာကြောင်းနည်းနည်းလောက် အကုန်သက်သာအောင် လုပ်နိုင်ပါတယ်။

```
public static int factorial(int n) {
    if (n = 0) {
    } else {
        return n * factorial(n-1);
    }
}
```

အခုအချိန်ကစပြီး ပိုပြီးတိုတဲ့ပုံစံကို တတ်နိုင်သလောက် သုံးပါမယ်။ ဒါပေမယ့် ကိုယ်တိုင် code ရေးသားတဲ့အခါ ပိုပြီးတိုက်ရိုက်ဆန်တဲ့ ရေးသားနည်းကို သုံးပါလို့ အကြံပြုချင်ပါတယ်။ အလုပ်ပြီးသွားလို့ အာရုံရသေးရင် ကျစ်ကျစ်လျစ်လျစ်ရှိအောင် လုပ်နိုင်ပါသေးတယ်။

factorial အပြီးမှာ fibonacci လို့ခေါ်တဲ့ ထပ်တလဲလဲ သတ်မှတ်ထားတဲ့ သင်္ချာ function တစ်ခုကို ပရိုဂရမ်ရေးကြည့်ကြပါမယ်။ သူ့ရဲ့ အဓိပ္ပာယ်ဖွင့်ဆိုချက်ကတော့ ဒီလိုပါ။

```
fibonacci(0) = 1
fibonacci (1) = 1
fibonacci (n) = fibonacci(n-1) + fibonacci(n-2);
```

Java ကို ပြန်ရေးရင် ဒီလိုရပါတယ်။

```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fibonacci(n-1) + fabonacci(n-2);
```

```
    }
}
```

ဒီနေရာမှာ ပရိုဂရမ် run တဲ့ လမ်းကြောင်းကိုလိုက်ရင် သေးငယ်တဲ့ n တန်ဖိုးတွေအတွက် တောင် ဦးနှောက်ပျက်သွားနိုင်ပါတယ်။ ဒါပေမယ့် ယုံကြည်မှုရှိရှိ ခုန်ကျော်ခြင်းအရ ထပ်တလဲလဲခေါ်ဆိုမှုနှစ်ခု လုပ်ကြည့်လို့မှန်နေရင် သူတို့နှစ်ခုကိုပေါင်းပြီး မှန်ကန်တဲ့ရလဒ်ကို ရနိုင်တာ သေချာပါတယ်။

**အခန်း (၆)**

# အကြိမ်ကြိမ်လည်ပတ်ခြင်း

## အကြိမ်ကြိမ်နေရာချထားခြင်း

ဒီအကြောင်းတော့ သိပ်မပြောခဲ့ဖူးပါဘူး။ ဒါပေမယ့် variable တစ်ခုမှာ နေရာချထားမှု တစ်ကြိမ်ထက်ပိုပြီး ပြုလုပ်နိုင်ပါတယ်။ ဒုတိယနေရာချထားတာရဲ့ သက်ရောက်မှုဟာ variable ရဲ့ တန်ဖိုးဟောင်းကို တန်ဖိုးသစ်နဲ့ လဲဖို့ပါပဲ။

```
int fred = 5;
System.out.print(fred);
fred = 7;
System.out.println(fred);
```

ဒီပရိုဂရမ်ရဲ့ output ဟာ 57 ဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတော့ fred ကို ပထမဆုံးအကြိမ် print လုပ်တဲ့အခါ 5 ရပြီး ဒုတိယအကြိမ် print လုပ်တဲ့အခါ 7 ရလို့ပါပဲ။

ဒီလို အကြိမ်ကြိမ်နေရာချထားမှုဟာ variable တွေကို တန်ဖိုးသယ်ဆောင်သူအနေနဲ့ ဖော်ပြရတဲ့ အကြောင်းရင်းဖြစ်ပါတယ်။ Variable တစ်ခုမှာ တန်ဖိုးတစ်ခုကို နေရာချထားလိုက်ရင် ပုံမှာပြထားသလို သယ်ဆောင်သူရဲ့ ပါဝင်မှုကို ပြောင်းလဲလိုက်ပါတယ်။

```
int fred = 5;     fred 5
fred = 7;         fred 7
```

Variable တစ်ခုမှာ နေရာချထားမှု တစ်ခုထက်ပိုလုပ်ရင် နေရာချထားတဲ့ဖော်ပြချက်နဲ့ ညီမျှချက်ဖော်ပြချက်ကြားမှာ ခွဲခြားသိဖို့လိုပါတယ်။ Java က = လက္ခဏာကို နေရာချထားမှုအတွက် သုံးတဲ့အတွက် a = b လိုဖော်ပြချက်မျိုးကို ညီမျှခြင်းဖော်ပြချက်လို့ အဓိပ္ပာယ်ကောက်ချင်စရာလေး ဖြစ်နေပါတယ်။ တကယ်တော့ သူတို့နှစ်ခု ဘာမှမဆိုင်ပါဘူး။

ပထမဆုံးအနေနဲ့ ညီမျှချက်ဟာ ဖလှယ်ရဂုဏ်သတ္တိကို ပိုင်ဆိုင်ပါတယ်။ နေရာချထားမှုမှာ ဒီလိုမရှိပါဘူး။ ဥပမာ သင်္ချာမှာ a = 7 ဖြစ်ရင် 7 = a ဖြစ်ပါတယ်။ ဒါပေမယ့် Java မှာ a = 7 ဟာ တရားဝင်ဖော်ပြချက်တစ်ခုဖြစ်ပြီး 7 = a ကတော့ တရားမဝင်ပါဘူး။

နောက်ပြီးတော့ သင်္ချာမှာ ညီမျှချက်ဖော်ပြချက်တစ်ခုဟာ အမြဲမှန်ပါတယ်။ အခုအချိန်မှာ a = b ဖြစ်နေရင် a ဟာ b နဲ့ အမြဲတမ်းညီနေလိမ့်မယ်။ Java မှာ နေရာချထားတဲ့ဖော်ပြချက်တစ်ခုဟာ variable နှစ်ခုကို ညီစေပါတယ်။ ဒါပေမယ့် သူတို့ဟာ ဒီလိုအမြဲတမ်းကြီးရှိသွားမယ်လို့ မဆိုလိုပါဘူး။

```
int a = 5;
int b = a;    // a and b are now equal
a = 3         // a and b are no loger equal
```

တတိယစာကြောင်းက a ရဲ့တန်ဖိုးကို ပြောင်းလဲစေပေမယ့် b ရဲ့ တန်ဖိုးကိုတော့ အပြောင်းအလဲ မဖြစ်စေပါဘူး။ ဒီတော့ သူတို့ဟာ ဆက်ပြီးမညီတော့ပါဘူး။ ပရိုဂရမ်မင်းဘာသာစကား တော်တော်များများမှာ ဒီရောထွေးမှုကိုရှောင်ရှားဖို့ နေရာချထားတဲ့နေရာမှာ <> နဲ့ != လို သင်္ကေတတွေကို သုံးကြပါတယ်။

အကြိမ်ကြိမ်နေရာချထားတာဟာ အသုံးဝင်တတ်ပေမယ့် သူ့ကို သတိနဲ့သုံးသင့်ပါတယ်။ Variable တွေရဲ့ တန်ဖိုးတွေဟာ ပရိုဂရမ်တစ်လျှောက်လုံး လိုက်ပြောင်းနေရင် code ကို ဖတ်ရခက်စေပြီး အမှားပြင်ရတာ အခက်အခဲရှိစေနိုင်ပါတယ်။

### အကြိမ်ကြိမ်လည်ပတ်ခြင်း

ကွန်ပျူတာတွေကို သုံးကြတဲ့အကြောင်းရင်းတစ်ရပ်က အထပ်ထပ်အခါခါလုပ်ရတဲ့ အလုပ်တွေကို အလိုအလျောက် ခိုင်းစေဖို့ဖြစ်ပါတယ်။ ထပ်တူညီတဲ့အလုပ်တွေနဲ့ ခပ်ဆင်ဆင်တူတဲ့ အလုပ်တွေကို အမှားမရှိဘဲ ထပ်ထပ်လုပ်တာဟာ ကွန်ပျူတာတွေ ကောင်းကောင်းလုပ်တတ်ပြီး လူတွေဆိုးဆိုးဝါးဝါးလုပ်တတ်တဲ့ အရာဖြစ်ပါတယ်။

`nLines` နဲ့ `countdown` တို့လို ထပ်တလဲလဲလုပ်ဆောင်ချက်တွေ ပြုလုပ်တဲ့ ပရိုဂရမ်တွေကို မြင်ခဲ့ဖူးပါပြီ။ ဒီလို ထပ်တလဲလဲလုပ်တာမျိုးကို အကြိမ်ကြိမ်လည်ပတ်ခြင်း (iteration) လို့ခေါ်ပြီး Java ဟာ အကြိမ်ကြိမ်လည်ပတ်တဲ့ ပရိုဂရမ်တွေရေးရလွယ်အောင် လုပ်ပေးတဲ့ ပရို

ဂရမ်လက္ခဏာ တော်တော်များများကို ပံ့ပိုးပေးထားပါတယ်။

အခုလေ့လာမယ့် လက္ခဏာနှစ်ခုကတော့ while ဖော်ပြချက်နဲ့ for ဖော်ပြချက် ဖြစ်ပါတယ်။

## while ဖော်ပြချက်

while ဖော်ပြချက်ကိုသုံးပြီး countdown ကို ပြန်ပြင်ရေးနိုင်ပါတယ်။

```
public static void countdown(int n) {
   while (n > 0){
         System.out.println(n);
         n = n-1;
   }
   System.out.println("Blastoff!");
}
```

while ဖော်ပြချက်ကို စာဖတ်သလို ဖတ်နိုင်ပါတယ်။ အဓိပ္ပာယ်ပြန်ရရင် n က သုညထက်ကြီးနေသ၍ n ရဲ့ တန်ဖိုးကို ဆက်ပြီး print လုပ်ပါ။ လုပ်နေရင်း n ရဲ့တန်ဖိုးကို 1 လျှော့ပါ။ သုညဆီရောက်ရင် Blastoff! ဆိုတဲ့ စကားလုံးကို print လုပ်ပါ။

ပိုပြီးကျကျနနပြောရရင် while ဖော်ပြချက် run တဲ့ လမ်းကြောင်းဟာ ဒီလိုပုံစံရှိပါတယ်။

၁။ လက်သည်းကွင်းတွေထဲက အခြေအနေကို တွက်ထုတ်ပြီး true ဒါမှမဟုတ် false ကို ရပါတယ်။

၂။ အခြေအနေကမှားနေရင် while ဖော်ပြချက်ကနေထွက်ပြီး နောက်လာတဲ့ ဖော်ပြချက်ကို run ပါတယ်။

၃။ အခြေအနေမှန်ရင် တွန့်ကွင်းတွေထဲက ဖော်ပြချက်တစ်ခုချင်းစီကို run ပြီး အဆင့် ၁ ဆီ ပြန်သွားပါတယ်။

ဒီစာကြောင်းမျိုးကို သံသရာ (loop) လို့ ခေါ်ပါတယ်။ ဘာလို့လဲဆိုတော့ တတိယအဆင့်ဟာ ထိပ်ဆုံးဆီ ပြန်ပတ်လို့ပါပဲ။ သံသရာကို ပထမဆုံး ဖြတ်တဲ့အခါ မှားရင် သံသရာထဲက ဖော်ပြချက်တွေကို ဘယ်တော့မှ run မှာ မဟုတ်ပါဘူး။ သံသရာထဲက ဖော်ပြချက်တွေကို သံသရာရဲ့ ကိုယ်ထည်လို့ ခေါ်ပါတယ်။

သံသရာရဲ့ ကိုယ်ထည်ဟာ variable တန်ဖိုးတစ်ခုခုကို ပြောင်းလဲပြီး နောက်ဆုံးမှာ အခြေအနေမှားအောင် လုပ်ပါတယ်။ အဲဒီအခါမှာ သံသရာရပ်သွားပါလိမ့်မယ်။ မဟုတ်ရင် သံသရာဟာ မဆုံးနိုင်အောင်လည်ပြီး သူ့ကို အဆုံးအစမရှိတဲ့ သံသရာ (infinite loop) လို့ ခေါ်ပါတယ်။

ကွန်ပျူတာပညာရှင်တွေ ရယ်စရာတွေ့တာတစ်ခုက ခေါင်းလျှော်ရည်ဘူးပေါ်မှာ "ဆပ်ပြာ ထည့်ပါ၊ ရေဆေးပါ၊ ထပ်လုပ်ပါ" လို့ ပြောထားတဲ့ အဆုံးအစမရှိ သံသရာပါ။

countdown မှာ သံသရာဟာ အဆုံးသတ်မယ်လို့ သက်သေပြနိုင်ပါတယ်။ ဘာလို့လဲဆို တော့ n ဟာ အဆုံးအစရှိပါတယ်။ n နဲ့ တန်ဖိုးဟာ သံသရာကို တစ်ကြိမ်ဖြတ်တိုင်း ငယ်သွား တယ်ဆိုတာ မြင်နိုင်ပါတယ်။

```
public static void sequence(int n) {
    while (n ! = 1) {
            System.out.println(n);

            if (n % 2 == 0) {           // n is even
            n = n / 2;
            } else {
                    n = n = n= 3 + 1;
            }
    }
}
```

ဒီသံသရာအတွက် အခြေအနေဟာ n != 1 ဖြစ်ပါတယ်။ ဒီတော့ သံသရာဟာ n = 1 မဖြစ်သ၍ ဆက်နေပါလိမ့်မယ်။ n - 1 ဖြစ်ရင်တော့ အခြေအနေကို မှားစေပါလိမ့်မယ်။

တစ်ကြိမ်ဖတ်တိုင်း ပရိုဂရမ်ဟာ n ရဲ့တန်ဖိုးကို print လုပ်ပြီး စုံလား၊ မလား စစ်ပါလိမ့် မယ်။ စုံဆိုရင် n ကို နှစ်နဲ့စားပါတယ်။ မဆိုရင် တန်ဖိုးကို သုံးနဲ့မြှောက်ပြီး တစ်ပေါင်းပါတယ်။ sequence ဆီကို ပို့ပေးလိုက်တဲ့ argument ဟာ 3 ဆိုရင် ကနဦးတန်ဖိုး 3 နဲ့ ရရှိတဲ့ကိန်းစဉ် ဟာ 3, 10, 5, 16, 8, 4, 2, 1 ဖြစ်ပါလိမ့်မယ်။

n ရဲ့တန်ဖိုးဟာ တိုးလိုက်လျော့လိုက် ဖြစ်နေတဲ့အတွက် n ဟာ 1 ဖြစ်ပါ့မလားလို့ ကံသေ ကံမ မပြောနိုင်သလို ပရိုဂရမ်ရပ်တန့်လိမ့်မယ်လို့လည်း ယုံထားလို့မရပါဘူး။ n ရဲ့ တချို့တန် ဖိုးတွေအတွက် ရပ်တန့်မယ့်အခြေအနေရှိကြောင်းကို သက်သေပြနိုင်ပါတယ်။ ဥပမာ စတဲ့တန် ဖိုးတွေဟာ နှစ်ရဲ့ထပ်ကိန်းတွေဖြစ်မယ်ဆိုရင် n ရဲ့တန်ဖိုးဟာ သံသရာက တစ်လျှောက်လုံး စုံဖြစ်ပြီး 1 မရမချင်း နှစ်နဲ့ အစားခံရပါလိမ့်မယ်။ ပြီးခဲ့တဲ့ဥပမာမှာ 16 ကနေစပြီး ဒီလိုကိန်း စဉ်မျိုးနဲ့ အဆုံးသတ်သွားပါတယ်။

တန်ဖိုးတချို့အတွက် ရပ်တန့်တာကလွဲလို့ ဒီပရိုဂရမ်ဟာ n ရဲ့ တန်ဖိုးအားလုံးအတွက် ရပ်တန့်မှုရှိမရှိကို သက်သေပြနိုင်သလားဆိုတာ စိတ်ဝင်စားစရာကောင်းတဲ့ မေးခွန်းတစ်ခုဖြစ် ပါတယ်။ အခုထက်ထိတော့ ဘယ်သူမှ ဟုတ်လား မဟုတ်လားဆိုတာ သက်သေပြလို့ မရသေး ပါဘူး။

**ဇယားများ**

သံသရာတွေအသုံးဝင်တဲ့ နေရာတစ်ခုကတော့ ဇယားအချက်အလက်ကို ထုတ်လုပ်ပြီး print လုပ်ဖို့ ဖြစ်ပါတယ်။ ဥပမာ ကွန်ပျူတာတွေ အလွယ်တကူမရှိခင်က လူတွေဟာ logarithm တွေ၊ sine တွေ၊ cosine တွေနဲ့ တခြားသင်္ချာ function တွေကို လက်နဲ့တွက်ခဲ့ရပါတယ်။

ဒါတွေကို ပိုလွယ်အောင်လုပ်ဖို့ function ပေါင်းစုံရဲ့တန်ဖိုးတွေကို ရှာနိုင်တဲ့ ဇယားရှည်ရှည်ကြီးတွေကို စာအုပ်လုပ်ထားကြပါတယ်။ ဒီဇယားတွေ ရေးရတာ နှေးကွေးပြီး ပျင်းစရာကောင်းပါတယ်။ ရလဒ်ကလည်း အမှားတွေ ပါတတ်ပါတယ်။

ကွန်ပျူတာတွေလည်းပေါ်လာရော အစောပိုင်းတုန့်ပြန်မှုတစ်ခုက "ကောင်းလိုက်တာ၊ ကွန်ပျူတာတွေသုံးပြီး ဇယားတွေထုတ်နိုင်ပြီ။ အမှားတွေရှိတော့မှာ မဟုတ်ဘူး" ဆိုပြီး ဖြစ်ပါတယ်။ ဒါဟာ မှန်တော့မှန်လာပါရဲ့။ ဒါပေမယ့် ရှေ့ရေးကို ဝေးဝေးမြင်တဲ့ အယူအဆတော့ မဟုတ်ပါဘူး။ နောက်ပိုင်းကျတော့ ကွန်ပျူတာတွေရဲ့ ဂဏန်းတွက်တဲ့ အရည်အချင်းက ကောင်းလွန်းလို့ ဇယားတွေတောင် ခေတ်နောက်ကျကုန်ပါတယ်။

အဲဒီလောက်နီးပါးကို ဖြစ်ကုန်ပါတယ်။ ဒါပေမယ့် တချို့တွက်ချက်မှုတွေအတွက် ကွန်ပျူတာတွေဟာ တန်ဖိုးဇယားတွေကို အနီးဆုံးအဖြေရအောင် သုံးပါတယ်။ ပြီးမှ အနီးဆုံးယူတာကို ပိုကောင်းအောင် တွက်ချက်မှုတွေ လုပ်ပါတယ်။ တချို့နေရာတွေမှာ အောက်ခြေကဇယားတွေရဲ့ အမှားတွေရှိနေခဲ့ပါတယ်။ အကျော်ကြားဆုံးကတော့ မူလ Intel Pentium ရဲ့ floating-point အစားအတွက်သုံးခဲ့တဲ့ ဇယားပဲဖြစ်ပါတယ်။

log ဇယားတစ်ခုဟာ အရင်ကလောက်တော့ အသုံးမဝင်ပေမယ့် အကြိမ်ကြိမ်လည်ပတ်တာနဲ့ ပတ်သက်ပြီးတော့ ဥပမာကောင်းတစ်ခု ဖြစ်ပါတယ်။ အောက်ကပရိုဂရမ်ဟာ ဘယ်ဘက်ကော်လံမှာ တန်ဖိုးတွေ အစီအစဉ်နဲ့ရေးပြီ ညာဖက်မှာ သူတို့ရဲ့ logarithm တွေကို ရေးပါတယ်။

```
double x = 1.0;
while (x < 10.0) {
    System.out.println(x + "    " + Math.log(x));
    x = x + 1. 0;
}
```

ပရိုဂရမ်ရဲ့ output ကတော့ အောက်မှာပြထားတဲ့အတိုင်း ဖြစ်ပါတယ်။

```
1.0    0.0
2.0    0.6931471805599453
3.0    1.0986122886681098
4.0    1.3862943611198906
```

```
5.0    1.6094379124341003
6.0    1.791759469228055
7.0    1.9459101490553132
8.0    2.0794415416798357
9.0    2.1972245773362196
```

ဒီတန်ဖိုးတွေကိုကြည့်ပြီး log function ရဲ့ ပုံမှန်အခြေဟာ ဘာဖြစ်မယ်လို့ ထင်လဲ။

နှစ်ရဲ့ထပ်ကိန်းတွေဟာ ကွန်ပျူတာဘာသာရပ်မှာ အရေးပါတဲ့အတွက် အခြေ 2 ရှိတဲ့ logrithm တွေကို မကြာခဏ ရှာဖွေရပါတယ်။ သူ့ကိုရှာလို့ ဒီဖော်မြူလာကို သုံးရပါတယ်။

$$\log_2 x = \log_e x / \log_e 2$$

print ဖော်ပြချက်ကို ဒီလိုပြောင်းလိုက်ရင်

```
System.out.println(x + "    " + Math.log(x) / Math. log(2.0);
```

ကိုရပြီး

```
1.0    0.0
2.0    1.0
3.0    1.5849625007211563
4.0    2.0
5.0    2.321928094887362
6.0    2.584962500721156
7.0    2.807354922057604
8.0    3.0
9.0    3.1699250014423126
```

ကို ထုတ်ပေးပါတယ်။

1, 2, 4 နဲ့ 8 ဟာ နှစ်ရဲ့ ထပ်ကိန်းတွေဖြစ်ကြပါတယ်။ ဘာလို့လဲဆိုတော့ သူတို့ရဲ့ အခြေ 2 ရှိတဲ့ logrithm တွေဟာ ဒသမအပိုင်း မပါလို့ပါပဲ။ 2 ရဲ့တခြားထပ်ကိန်းတွေရဲ့ logarithm တွေကို ရှာချင်ရင် ပရိုဂရမ်ကို ဒီလိုပြန်ပြင်ရေးနိုင်ပါတယ်။

```
double x = 1.0;
while (x < 100.0) {
```

```
    System.out.println(x + "    " + Math.log(x) / Math.log(2.0));
    x = x * 2.0;
}
```

အခုတော့ သံသရာဖြတ်သန်းတိုင်း x မှာ တစ်ခုခုပေါင်းပြီး arithmetic ကိန်းစဉ်တစ်ခု ရယူမယ့်အစား x ကို တစ်ခုခုနဲ့မြှောက်ပြီး geometric ကိန်းစဉ်တစ်ခုကို ရယူပါတယ်။ ရလဒ်တော့

```
1.0    0.0
2.0    1.0
4.0    2.0
8.0    3.0
16.0   4.0
32.0   5.0
64.0   6.0
```

log ဇယားတွေဟာ ဆက်ပြီးအသုံးမဝင်တော့ပါဘူး။ ဒါပေမယ့် ကွန်ပျူတာပညာရှင်တွေအတွက်တော့ နှစ်ရဲ့ထပ်ကိန်းတွေကို သိထားတာဟာ အသုံးဝင်ပါသေးတယ်။ အလုပ်မရှိတဲ့အချိန်တွေမှာ နှစ်ရဲ့ထပ်ကိန်းတွေကို 65536 ($2^{16}$) အထိ မှတ်နေသင့်ပါတယ်။

### နှစ်ဖက်မြင်ဇယားများ

နှစ်ဖက်မြင်ဇယားဆိုတာ အတန်းတစ်ခု အတိုင်တစ်ခုရွေးပြီး သူတို့ဆုံမှတ်ရဲ့ တန်ဖိုးကို ဖတ်ရှုတဲ့ ဇယားဖြစ်ပါတယ်။ အမြှောက်ဇယားဟာ ဥပမာတစ်ခုဖြစ်ပါတယ်။ 1 ကနေ 6 အထိ တန်ဖိုးတွေရဲ့ အမြှောက်ဇယားတစ်ခုကို ပြုလုပ်ချင်တယ်လို့ ယူဆပါ။

အကောင်းဆုံးစနည်းကတော့ 2 ရဲ့ ဆတိုးကိန်းတွေကို print လုပ်တဲ့ သံသရာရိုးရိုးရှင်းရှင်းတစ်ခုကို စာကြောင်းတစ်ကြောင်းပေါ်မှာ ရေးဖို့ပါပဲ။

```
int i = 1;
while (i <= 6) {
    System.out.print(2 * 1 + "  ");
    i = i + 1;
}
System.out.println(" ");
```

ပထမဆုံးစာကြောင်းဟာ 1 ဆိုတဲ့ variable တစ်ခုကို စတင်ပါတယ်။ သူက ရေတွက်ကိန်း

ဂဏန်း (counter) အဖြစ်ဆောင်ရွက်မှာ ဖြစ်ပါတယ်။ သူ့ကို သံသရာ variable (loop variable) လို့လည်း ခေါ်ပါသေးတယ်။ သံသရာ run တဲ့အခါ i ရဲ့တန်ဖိုးဟာ 1 ကနေ 6 အထိ တိုးလာပြီး i က 7 ဖြစ်တဲ့အခါ သံသရာပြီးဆုံးသွားပါတယ်။ သံသရာကို တစ်ပတ်ဖြတ်တိုင်း 2 * i ရဲ့တန်ဖိုးကို ကွက်လပ်သုံးခုနောက်မှာထားပြီး print လုပ်ပါတယ်။ println command အစား print command ကို သုံးနေတာဖြစ်တဲ့အတွက် output ကို စာကြောင်းတစ်ကြောင်းတည်းနဲ့ ပေါ်ပါတယ်။

ပြီးခဲ့တဲ့ အပိုင်းတစ်ပိုင်းမှာ ပြောခဲ့သလိုပဲ တချို့ပတ်ဝန်းကျင်တွေဟာ print ရဲ့ output ကို println နှိုးဆွမှုမပြုမချင်း သိမ်းဆည်းထားတတ်ပါတယ်။ ပရိုဂရမ်ပြီးသွားတဲ့အထိ println ကို နှိုးဆွဖို့ မေ့နေတယ်ဆိုရင် သိမ်းထားတဲ့ output ကို မမြင်လိုက်ရတဲ့အခါတွေ ရှိလာနိုင်တယ်။

ဒီပရိုဂရမ်ရဲ့ output ဟာ အောက်မှာပြောထားတဲ့အတိုင်း ဖြစ်ပါတယ်။

```
2 4 6 8 10 12
```

အခုထိတော့ ဟုတ်နေတာပါပဲ။ နောက်တစ်ဆင့်ကတော့ အုပ်ယူခြင်းနဲ့ ယေဘုယျပြုလုပ်ခြင်းပဲ ဖြစ်ပါတယ်။

### အုပ်ယူခြင်းနှင့် ယေဘုယျပြုလုပ်ခြင်း

အုပ်ယူခြင်း (encapsulation) ဆိုတာ code တချို့ကိုယူပြီး method တစ်ခုထဲမှာ ထုပ်ထည့်တာကို ဆိုလိုပါတယ်။ ဒီလိုလုပ်ပြီး method တွေရဲ့ ကောင်းကျိုးတွေကို အသုံးချခွင့်ပေးပါတယ်။ အုပ်ယူတဲ့ ဥပမာနှစ်ခုကို တွေ့ခဲ့ပါတယ်။ printParity နဲ့ isSingleDigit ကို ရေးခဲ့တုန်းက အချက်အလက်နဲ့ လုပ်ဆောင်ချက်ကို method တစ်ခုထဲမှာ အုပ်ယူခဲ့ပါတယ်။

ယေဘုယျပြုလုပ်တယ် (generalisation) ဆိုတာ 2 ရဲ့ ဆတိုးကိန်းတွေ print လုပ်တာလို တိကျတဲ့အလုပ်တစ်ခုကို ပိုပြီးယေဘုယျကျအောင် ပြင်တာပဲဖြစ်ပါတယ်။ ဥပမာ ကိန်းပြည့်တိုင်းရဲ့ ဆတိုးကိန်းတွေကို print လုပ်ရင် ယေဘုယျလုပ်တာ ဖြစ်ပါတယ်။

အောက်မှာပြထားတာကတော့ ပြီးခဲ့တဲ့အပိုင်းကို အုပ်ယူထားပြီး n ရဲ့ ဆတိုးကိန်းတွေကို print လုပ်ပေးအောင် ယေဘုယျလုပ်ထားတာ ဖြစ်ပါတယ်။

```
public static void printMultiples(int n) {
    int i = 1;
    while (i <= 6) {
            System.out.print(n * i + "   ");
            i = i + 1;
    }
```

```
    System.out.println(" ");
  }
```

အုပ်ယူဖို့ လိုအပ်သမျှကတော့ ပထမစာကြောင်းကို ပေါင်းထည့်ပြီး နာမည်၊ parameter နဲ့ ပြန်တဲ့တန်ဖိုးကို ကြေညာရပါတယ်။ ယေဘုယျလုပ်ဖို့ တန်ဖိုး 2 နေရာမှာ n ဆိုတဲ့ parameter နဲ့ လဲထည့်ရပါတယ်။

ဒီ method ကို 2 ဆိုတဲ့ argument နဲ့ နှိုးဆွရင် output ကို အရင်ကလိုပဲရပြီး argument က 3 ဖြစ်ရင် output က ဒီလိုရပါတယ်။

```
3 6 9 12 15 18
```

Argument က 4 ဖြစ်ရင် output က ဒီလိုဖြစ်သွားပါတယ်။

```
4 8 12 16 20 24
```

အခုလောက်ဆိုရင် အမြှောက်ဇယားတစ်ခုကို ဘယ်လို print လုပ်မယ်ဆိုတာ သိနေလောက်ပါပြီ။ printMultiples ကို argument အမျိုးမျိုးနဲ့ နှိုးဆွပါမယ်။ ပိုပြီးတိတိကျကျပြောရရင် နောက်ထပ် သံသရာတစ်ခုကိုသုံးပြီး အတန်းတွေရအောင် လုပ်ပါတယ်။

```
int i = 1
while (i <= 6) {
   printMultiples(i);
   i = i + 1;
}
```

ပထမဆုံးအနေနဲ့ ဒီသံသရာဟာ printMultiples ထဲက သံသရာနဲ့ ဘယ်လောက်တူသလဲဆိုတာကို သတိပြုပါ။ အခုလုပ်လိုက်တာကတော့ print ဖော်ပြချက်ကို method နှိုးဆွမှုတစ်ခုနဲ့ အစားထိုးလိုက်တာပဲ ဖြစ်ပါတယ်။

ဒီပရိုဂရမ်ရဲ့ output ကတော့ ဒီလိုဖြစ်ပါတယ်။

```
1   2   3   4   5   6
2   4   6   8   10  12
3   6   9   12  15  18
4   8   12  16  20  24
5   10  15  20  25  30
6   12  18  24  30  36
```

ဒီအမြှောက်ဇယားကတော့ နည်းနည်းစောင်းနေပါတယ်။ ဒီလိုစောင်းနေတာကို မကျေနပ်ဖြစ်နေရင် Java ဟာ output ရဲ့ ခင်းကျင်းပုံကို ထိန်းချုပ်ခွင့်ပေးတဲ့ method တွေကို ပံ့ပိုးပေးထားပါတယ်။ ဒါပေမယ့် အခုအချိန်မှာတော့ ဒါကို မဆွေးနွေးပါဘူး။

## Method များ

ပြီးခဲ့တဲ့အပိုင်းမှာတုန်းက method တွေရဲ့ ကောင်းကျိုးတွေဆိုတဲ့ စကားလုံးကို သုံးနှုန်းခဲ့ပါတယ်။ အခုအချိန်မှာတော့ အဲဒီကောင်းကျိုးတွေက ဘာလဲဆိုတာ သိချင်နေပါလိမ့်မယ်။ အောက်မှာပြောထားတာက method တွေ အသုံးဝင်ရခြင်း အကြောင်းရင်းတချို့ ဖြစ်ပါတယ်။

- ဖော်ပြချက် အစုအဝေးတစ်ခုကို နာမည်တစ်ခုပေးခြင်းအားဖြင့် ပရိုဂရမ်ကို ဖတ်ရ၊ အမှားပြင်ရ လွယ်စေပါတယ်။
- ရှည်လျားတဲ့ ပရိုဂရမ်တစ်ပုဒ်ကို method တွေအဖြစ် ပိုင်းခြင်းအားဖြင့် ပရိုဂရမ်ရဲ့ အစိတ်အပိုင်းတွေကို ခွဲထုတ်နိုင်ပါတယ်။ သူတို့ကို သီးသန့်အမှားရှာနိုင်ပါတယ်။ ပြီးရင် တစ်ခုလုံးအနေနဲ့ ပြန်ဖွဲ့စည်းနိုင်ပါတယ်။
- Method တွေဟာ ထပ်တလဲလဲလည်ပတ်တာနဲ့ အကြိမ်ကြိမ်လည်ပတ်တာ နှစ်ခုစလုံးကို အထောက်အပံ့ပြုပါတယ်။
- ဒီဇိုင်းကောင်းကောင်းထုတ်ထားတဲ့ method တွေဟာ ပရိုဂရမ်အများကြီးအတွက် အသုံးဝင်တတ်ပါတယ်။ Method တစ်ခုကိုရေးပြီး အမှားပြင်ပြီးတာနဲ့ သူ့ကို အကြိမ်ကြိမ် ပြန်သုံးနိုင်ပါတယ်။

### အုပ်ယူခြင်းကို ဆက်လက်လေ့လာခြင်း

အုပ်ယူခြင်းအကြောင်းကို ထပ်ပြီးရှင်းပြဖို့ ပြီးခဲ့တဲ့အပိုင်းက code ကိုယူပြီး method တစ်ခုထဲ ထုပ်ထည့်လိုက်ပါမယ်။

```
public static void printMultTable() {
  int i = 1;
  while (i <= 6) {
        printMultiples(i);
        i = i + 1;
  }
}
```

ဒီမှာသရုပ်ဖော်နေတဲ့ ဖြစ်စဉ်ကတော့ အသုံးများတဲ့ ရေးသားမှုအစီအစဉ်တစ်ခု ဖြစ်ပါတယ်။ Code ရေးသားရာမှာ main မှာ ထပ်ထပ်ရေးပြီး ဒါမှမဟုတ် တခြားနေရာမှာရေးပြီး code ကို ဖွံ့ဖြိုးလာအောင် လုပ်ရပါတယ်။ ပြီးမှ အလုပ်ဖြစ်နေတဲ့အခါ သူ့ကို method တစ်ခုထဲမှာ ပါဆယ်ထုပ်သလို သပ်သပ်ထုပ်ထည့်ရပါတယ်။

ဒီလိုလုပ်တာ အသုံးဝင်ရခြင်းအကြောင်းက တစ်ခါတစ်ရံမှာ ပရိုဂရမ်သာ စရေးလိုက်ရတယ်။ ကိုယ်ပရိုဂရမ်ကို ဘယ်လိုပိုင်းရမလဲဆိုတာ စဉ်းစားမရတဲ့အချိန်တွေ ရှိတတ်ပါတယ်။ ဒီနည်းတစ်နည်းဟာ လုပ်ရင်းကိုင်ရင်းနဲ့ ဒီဇိုင်းထုတ်ခွင့်ရသွားပါတယ်။

## ဒေသခံ Variable များ

ဒီအချိန်မှာ i ဆိုတဲ့ variable တစ်ခုတည်းကို printMultiples မှာရော၊ printMultTable မှာရော သုံးနိုင်ရတာလဲဆိုတာ တွေးကောင်းတွေးနေပါမယ်။ Variable တစ်ခုကို တစ်ခါပဲကြေညာနိုင်တယ်လို့ မပြောခဲ့ဘူးလား။ ပြီးတော့ method တစ်ခုက variable ရဲ့ တန်ဖိုးကို သွားပြောင်းရင် ပြဿနာတွေ မတက်ကုန်ဘူးလား။

ဒီမေးခွန်းနှစ်ခုစလုံးအတွက် အဖြေကတော့ မဟုတ်ဘူးလို့ပဲ ဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတော့ printMultiples ထဲက i နဲ့ printMultTable ထဲက i ဟာ တူညီတဲ့ variable တွေ မဟုတ်ကြလို့ပါပဲ။ သူတို့ဟာ နာမည်အတူတူရှိပေမယ့် သိမ်းဆည်းတဲ့နေရာတစ်ခုတည်းကို မညွှန်းပါဘူး။ Variable တစ်ခုရဲ့ တန်ဖိုးကိုပြောင်းတာဟာ နောက်တစ်ခုအပေါ် သက်ရောက်မှုမရှိပါဘူး။

Method သတ်မှတ်ချက်ထဲမှာ ကြေညာထားတဲ့ variable တွေကို ဒေသခံ variable တွေလို့ ခေါ်ပါတယ်။ ဒေသခံ variable တစ်ခုကို သူ့ရဲ့မိခင် method ရဲ့အဖြစ်ကနေ မရောက်ရှိနိုင်ပါဘူး။ ပြီးတော့ နာမည်တူနေတဲ့ variable တွေကို ဆန္ဒရှိသလောက် လွတ်လွတ်လပ်လပ် ပိုင်ဆိုင်နိုင်ပါတယ်။ Method မတူသရွေ့ပေါ့။

တစ်ခါတရံမှာ မျက်စေ့ရှုပ်သက်သာအောင် methdo အမျိုးမျိုးကွဲပြားတဲ့ variable နာမည်အမျိုးမျိုးကို သုံးရတာ အဆင်ပြေပါတယ်။ ဒါပေမယ့် နာမည်တွေကို ပြန်သုံးသင့်တဲ့အကြောင်းရင်းတွေ ကောင်းကောင်းရှိတတ်ပါတယ်။ ဥပမာ i, j နဲ့ k ကို သံသရာ variable အဖြစ် အသုံးများပါတယ်။ Method တစ်ခုထဲမှာ သုံးပြီးသားဖြစ်နေလို့ နောက်တစ်ခုမှာ ထပ်မသုံးရင် သံသရာ variable တွေ အများကြီးဖြစ်နေပြီး ပရိုဂရမ်ကို ဖတ်ရပိုခက်စေပါတယ်။

## ယေဘုယျပြုလုပ်ခြင်းကို ပိုမိုလေ့လာခြင်း

ယေဘုယျလုပ်တဲ့ ဥပမာနောက်တစ်ခုအနေနဲ့ 6x6 ဇယားလောက်ပဲ မဟုတ်ဘဲ အရွယ်အစား အမျိုးမျိုးဖြစ်နိုင်တဲ့ အမြှောက်ဇယားထုတ်ပေးတဲ့ ပရိုဂရမ်တစ်ပုဒ်ကို ရေးချင်တယ်လို့ ယူဆပါ။ printMultTable မှာ parameter တစ်ခုကို ပေါင်းထည့်နိုင်ပါတယ်။

```
public static void printMultTable(int high) {
    int i = 1;
    while (i <= high) {
            printMultiples(i);
            i = i + 1;
```

```
    }
}
```

6 ဆိုတဲ့တန်ဖိုးကို high နဲ့ အစားထိုးလိုက်ပါတယ်။ printMultTable ကို 7 ဆိုတဲ့ argument နဲ့ အစားထိုးလိုက်ရင် အောက်မှာပြထားသလို ရပါတယ်။

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 4 | 6 | 8 | 10 | 12 |
| 3 | 6 | 9 | 12 | 15 | 18 |
| 4 | 8 | 12 | 16 | 20 | 24 |
| 5 | 10 | 15 | 20 | 25 | 30 |
| 6 | 12 | 18 | 24 | 30 | 36 |
| 7 | 14 | 21 | 28 | 35 | 42 |

ဒါဟာ အဆင်တော့ ပြေပါတယ်။ ဒါပေမယ့် ဇယားကိုတော့ စတုရန်းကျစေချင်ပါတယ်။ (ကော်လံနဲ့ အတန်းအရေအတွက်တွေကို တူစေချင်ပါတယ်။) ဒီတော့ printMultiples မှာ parameter နောက်တစ်ခု ထည့်ပေါင်းပြီး ဇယားမှာရှိရမယ့် ကော်လံအရေအတွက်ကို သတ်မှတ်ခိုင်းပါတယ်။

နည်းနည်းလေး စိတ်ကသိကအောက်ဖြစ်အောင် ဒီ parameter ကို high လို့ပဲ နာမည်ပေးပါတယ်။ မတူညီတဲ့ method တွေမှာ နာမည်အတူတူရှိတဲ့ parameter တွေ ထားလို့ရတယ် ဆိုတာကို သရုပ်ဖော်ရင်းပေါ့။

```
public static void printMultiples (int n, int high) {
    int i = 1;
    while (i <= high) {
            System.out.print(n i + "    ");
            i = i + 1;
    }
    newLine();
}

public static void printMultTable (int high) {
    int i = 1;
    while (i <= high) {
            printMultiples (i, high);
            i = i + 1;
```

```
    }
}
```

Parameter အသစ်တစ်ခုကို ထည့်တဲ့အခါ method ရဲ့ ပထမဦးဆုံးစာကြောင်းကိုပါ ပြောင်းပေးရတယ်ဆိုတာ သတိပြုပါ။ `printMultTable` မှာ method ကို နှိုးဆွတဲ့နေရာလည်း ပြောင်းပေးရပါတယ်။ မျှော်လင့်ထားသလိုပဲ ဒီပရိုဂရမ်ဟာ 7 x 7 ဇယားကို ထုတ်လုပ်ပါတယ်။

```
1  2   3   4   5   6   7
2  4   6   8   10  12  14
3  6   9   12  15  18  21
4  8   12  16  20  24  28
5  10  15  20  25  30  35
6  12  18  24  30  36  42
7  14  21  28  35  42  49
```

Method တစ်ခုကို ယေဘုယျလုပ်တဲ့အခါ ကိုယ်မရည်ရွယ်တဲ့ အရည်အချင်းတွေကို တွေ့ရပါလိမ့်မယ်။ ဥပမာ ab = ba ဖြစ်တဲ့အတွက် အမြှောက်ဇယားဟာ ခေါက်ချိုးညီပါတယ်။ ဒီတော့ ဇယားအဖွဲ့ဝင်အားလုံး နှစ်ခါပါနေပါတယ်။ ဇယားတစ်ဝက်ကိုပဲ print လုပ်ပြီး မင်ကုန်သက်သာအောင် လုပ်နိုင်ပါတယ်။ ဒီလိုလုပ်ဖို့ `printMultTable` မှာ စာကြောင်းတစ်ကြောင်းပဲ ပြောင်းစရာလိုပါတယ်။

`printMultiples (i, high);` ကို

`printMultiples (i, i);` လို့ ပြောင်းလိုက်ပါ။

```
1
2  4
3  6   9
4  8   12  16
5  10  15  20  25
6  12  18  24  30  36
7  14  21  28  35  42  49
```

ကို ရပါတယ်။ ဘယ်လိုရသွားသလဲဆိုတာကို ပရိုဂရမ်မှာ ပြန်ကြည့်ပြီး စဉ်းစားပါ။

အခန်း ၇

# စာသားများနှင့် အရာဝတ္ထုများ

## ဝတ္ထုများပေါ်တွင် method များကို နှိုးဆွခြင်း

Java နဲ့ တခြား ပရိုဂရမ်းမင်းဘာသာစကားတွေမှာ ဝတ္ထုတွေဟာ method တစ်စုံနဲ့ ပါရှိတဲ့ ဆင်တူရာ data အစုအဝေးတွေ ဖြစ်ပါတယ်။ ဒီ method တွေဟာ တွက်ချက်မှုတွေ ဆောင်ရွက်ပြီး၊ တစ်ခါတစ်ရံမှာ ဝတ္ထုရဲ့ data တွေကို ပြုပြင်ပြီး ဝတ္ထုတွေအပေါ်မှာ လုပ်ဆောင်ချက်တွေ လုပ်ကိုင်ပါတယ်။ မြင်ဖူးသမျှ အမျိုးအစားတွေထဲမှာ String တွေပဲ ဝတ္ထုတွေဖြစ်ကြပါတယ်။ ဝတ္ထုရဲ့ အဓိပ္ပာယ်ဖွင့်ဆိုချက်ကိုမြင်ပြီး String ဝတ္ထုထဲမှာ ဘယ်လို data တွေကို သိမ်းထားသလဲ၊ String ဝတ္ထုတွေပေါ်မှာ ဘယ်လို method မျိုးတွေကို နှိုးဆွနိုင်သလဲဆိုတာ မေးစရာရှိလာပါတယ်။

String ဝတ္ထုထဲမှာ ထည့်ထားတဲ့ data ဟာ စာသားရဲ့ အက္ခရာတွေ ဖြစ်ပါတယ်။ စာသားတွေအပေါ်မှာ လုပ်ကိုင်နိုင်တဲ့ method တော်တော်များများရှိပါတယ်။ ဒါပေမယ့် ဒီစာအုပ်ထဲမှာတော့ နည်းနည်းပဲဆွေးနွေးပါမယ်။ စာရင်းအပြည့်အစုံကို လိုချင်ရင် http://java.sun.com/j2se/1.4.1/docs/api/java/lang/String.html မှာ သွားကြည့်နိုင်ပါတယ်။

ပထမဆုံးကြည့်ရှုကြမယ့် method ဟာ charAt ဖြစ်ပါတယ်။ သူ့ကိုသုံးပြီး String တစ်ခုကနေ အက္ခရာတွေကို နှိုက်ထုတ်နိုင်ပါတယ်။ ရလဒ်ကိုသိမ်းဆည်းဖို့ အက္ခရာတစ်လုံးစီကို သိမ်းပေးနိုင်တဲ့ variable အမျိုးအစားတစ်ခုကို လိုအပ်ပါတယ်။ အက္ခရာတစ်ခုစီကို character တွေလို့ခေါ်ပြီး သူတို့ကို သိမ်းဆည်းတဲ့အမျိုးအစားကို char လို့ခေါ်ပါတယ်။

char တွေဟာ အရင်ကမြင်ဖူးခဲ့တဲ့ အမျိုးအစားတွေလိုပဲ ပြုမူပါတယ်။

```
char fred = 'c';
if (fred == 'c') {
      System.out.println(fred);
}
```

အက္ခရာတန်ဖိုးတွေကို inverted comma တစ်ထပ်အထဲမှာ ရေးသားရပါတယ်။ စာသားတန်ဖိုးတွေနဲ့ မတူတာက အက္ခရာတန်ဖိုးတွေမှာ အက္ခရာတစ်လုံး ဒါမှမဟုတ် သင်္ကေတတစ်ခုပဲ ပါဝင်နိုင်ပါတယ်။

အောက်မှာပြထားတာကတော့ charAt method ကို အသုံးပြုပုံပါ။

```
String fruit = "banana";
char letter = fruit.charAt(1);
System.out.println(letter);
```

fruit.charAt ဆိုတဲ့သဒ္ဒါဟာ fruit လို့နာမည်ပေးထားတဲ့ ဝတ္ထုအပေါ်မှာ charAt ဆိုတဲ့ method ကို နှိုးဆွနေတယ်လို့ ဆိုလိုပါတယ်။ ဒီ method ဆီကို 1 ဆိုတဲ့ argument ပို့ပေးနေပါတယ်။ ဆိုလိုတာက စာသားရဲ့ ပထမအက္ခရာကို သိချင်တယ်လို့ပဲ ဖြစ်ပါတယ်။ ရလဒ်ဟာ character တစ်ခုဖြစ်ပြီး သူ့ကို letter လို့ နာမည်ပေးထားတဲ့ char တစ်ခုထဲမှာ သိမ်းထားပါတယ်။ letter ရဲ့ တန်ဖိုးကို print လုပ်တော့ အံ့ဩစရာတစ်ခုကို တွေ့ရပါတယ်။ ထွက်လာတာကတော့ a ပဲ ဖြစ်ပါတယ်။

a ဟာ "banana" ရဲ့ ပထမဆုံးအက္ခရာ မဟုတ်ပါဘူး။ လျှို့ဝှက်နက်နဲတဲ့ အကြောင်းပြချက်တွေအတွက် ကွန်ပျူတာပညာရှင်တွေဟာ ရေတွက်မှုကို သုညကနေ စတင်ပါတယ်။ "banana" ရဲ့ သုညခုမြောက်အက္ခရာဟာ b ဖြစ်ပါတယ်။ ပထမမြောက်အက္ခရာဟာ a ဖြစ်ပြီး ဒုတိယမြောက်အက္ခရာကတော့ n ဖြစ်ပါတယ်။

စာသားရဲ့ သုညခုမြောက်အက္ခရာကို လိုချင်ရင် သုညကို argument အနေနဲ့ ပေးပို့ရပါတယ်။

```
char letter = fruit.charAt(0);
```

**အရှည်**

အခုကြည့်ရှုကြမယ့် String ရဲ့ ဒုတိယ method ကတော့ length ပဲ ဖြစ်ပါတယ်။ သူက စာသားတစ်ခုထဲက အက္ခရာအရေအတွက်ကို ပြန်ထုတ်ပေးပါတယ်။ ဥပမာ

```
int length = fruit.length();
```

length ဟာ () မှာ မြင်ရသလိုပဲ argument တွေ မယူပါဘူး။ ကိန်းပြည့်တစ်ခုကို ပြန်ထုတ်ပေးပါတယ်။ ဒီနေရာမှာ ပြန်ထုတ်ပေးတဲ့ တန်ဖိုးဟာ 6 ဖြစ်ပါတယ်။ Variable ကို method နာမည်နဲ့တူအောင်လည်း ပေးလို့ရပါတယ်။ ဒါပေမယ့် ဒါဟာ ဖတ်ရှုတဲ့ လူသားတွေအတွက်တော့ မျက်စေ့ရှုပ် နားရှုပ်စရာ ဖြစ်နေပါလိမ့်မယ်။

စာသားတစ်ခုရဲ့ နောက်ဆုံးအက္ခရာကိုရှာဖို့ အောက်မှာပြထားတဲ့ code မျိုးကို ရေးချင်စိတ်တွေ ပေါက်လာတတ်ပါတယ်။

```
int length = fruit.length();
char last = fruit.charAt(length);    // WRONG!!
```

ဒါဟာ တရားမဝင်ပါဘူး။ ဘာလို့လဲဆိုတော့ "banana" မှာ ဆဋ္ဌမမြောက်အက္ခရာ မရှိပါဘူး။ ရေတွက်မှုကို 0 ကနေ စတင်ခဲ့တဲ့အတွက် အက္ခရာခြောက်ခုကို 0 ကနေ 5 အထိ နံပါတ်တပ်ခဲ့ပါတယ်။ နောက်ဆုံးအက္ခရာကိုရဖို့ length ကနေ 1 နုတ်ရပါတယ်။

```
int length = fruit.length();
char last = fruit.charAt(length - 1);
```

**ဖြတ်သန်းခြင်း**

စာသားတွေနဲ့ ပြုလုပ်လေ့ရှိတဲ့ အရာတစ်ခုကတော့ အစကနေ အက္ခရာတစ်ခုချင်းစီကို ရွေးပြီး မဆုံးမချင်း တစ်ခုခုလုပ်တာ ဖြစ်ပါတယ်။ ဒီလို ချေဖျက်မှုပုံစံကို ဖြတ်သန်းတယ်လို့ ခေါ်ပါတယ်။ ဖြတ်သန်းတဲ့ဖြစ်စဉ်ကို သဘာဝကျကျ ရေးသားနိုင်တဲ့နည်းကတော့ while ဖော်ပြချက်ပဲ ဖြစ်ပါတယ်။

```
int index = 0;
while (index < fruit.length()) {
   char letter = fruit.charAt(index);
   System.out.println(letter);
   index = index + 1;
}
```

ဒီသံသရာဟာ စာသားကိုဖြတ်သန်းပြီး အက္ခရာတစ်ခုစီကို စာကြောင်းတစ်ကြောင်းမှာ print လုပ်ပါတယ်။ အခြေအနေဟာ index < fruit.length() ဖြစ်တယ်ဆိုတာကို သတိပြုပါ။ ဆိုလိုတာက index ဟာ စာသားရဲ့ အလျားနဲ့တူညီနေရင် အခြေအနေဟာ မှားသွားပြီး သံသရာရဲ့ကိုယ်ထည်ကို run တော့မှာ မဟုတ်ပါဘူး။ ရောက်ရှိခဲ့သမျှ နောက်ဆုံးအက္ခရာဟာ index တန်ဖိုး fruit.length()-1 ရှိတဲ့ အက္ခရာပဲဖြစ်ပါတယ်။

သံသရာ variable ရဲ့ နာမည်က index ဖြစ်ပါတယ်။ Index ဟာ အစီအစဉ်တကျရှိနေတဲ့ အစုအဝေးတစ်ခုရဲ့ အဖွဲ့ဝင်တစ်ခုကို ဖော်ပြတဲ့ variable တစ်ခု ဒါမှမဟုတ် တန်ဖိုးတစ်

ခုဖြစ်ပါတယ်။ ဒီနေရာမှာ အစုအဝေးဟာ စာသားထဲက အက္ခရာတွဲဖြစ်ပါတယ်။ Index က ကိုယ်လိုချင်တဲ့ အဖွဲ့ဝင်ကို ဖော်ပြပါတယ်။ အက္ခရာတစ်ခုစီဟာ index တစ်ခုကို ပိုင်ဆိုင်ပြီး index တိုင်းဟာ အက္ခရာတစ်ခုတည်းကို ညွှန်းဆိုအောင် အစုအဝေးကို အစီအစဉ်ချထားပါတယ်။

လေ့ကျင့်ခန်းအနေနဲ့ စာသားကို argument တစ်ခုအနေနဲ့ လက်ခံပြီး အက္ခရာအားလုံးကို စာကြောင်းတစ်ကြောင်းတည်းပေါ်မှာ print လုပ်တဲ့ method တစ်ခုကို ရေးကြည့်ပါ။

## Run နေစဉ် ဖြစ်ပေါ်သည့်အမှားများ

အစောပိုင်းမှာတုန်းက run နေတုန်းဖြစ်တဲ့ အမှားတွေအကြောင်းကို ပြောခဲ့ပါတယ်။ သူတို့ဟာ ပရိုဂရမ်စ မ run မချင်း ပေါ်မလာတဲ့အမှားတွေ ဖြစ်ပါတယ်။ Java မှာ run တုန်းဖြစ်တဲ့ အမှားတွေကို ခြွင်းချက် (exception) တွေလို့ ခေါ်ပါတယ်။

အခုအချိန်ထိတော့ run နေတုန်းဖြစ်တဲ့ အမှားတွေကို တွေ့ခဲ့မှာမဟုတ်ပါဘူး။ ဘာလို့လဲဆိုတော့ ဒီလိုဖြစ်ပေါ်စေတဲ့ကိစ္စတွေ သိပ်မလုပ်ဘူးလို့ပါပဲ။ အခုတော့ ဒီလိုစလုပ်ရပါတော့မယ်။ charAt command ကိုသုံးပြီး အနုတ်တန်ဖိုးရှိတဲ့ ဒါမှမဟုတ် length()-1 ထက်ပိုကြီးတဲ့ index တန်ဖိုးတစ်ခုကို ပံ့ပိုးမိရင် ခြွင်းချက်တစ်ခုကို ရပါလိမ့်မယ်။ ပိုပြီး တိတိကျကျပြောရရင် StringIndexOutOfBoundException ကို ရပါလိမ့်မယ်။ စမ်းကြည့်ပြီး မျက်မြင်ကြည့်ရှုပါ။

ခြွင်းချက်တစ်ခုဖြစ်ပေါ်ရင် ပရိုဂရမ်ဟာ ခြွင်းချက်အမျိုးအစားနဲ့ ပရိုဂရမ်ရဲ့ ဘယ်နေရာမှာ ဒီခြွင်းချက်ဖြစ်ပေါ်သလဲဆိုတာ ဖော်ပြတဲ့ အမှား message တစ်ခုကို print လုပ်ပြီး ပရိုဂရမ်ထွက်သွားပါလိမ့်မယ်။

## စာရွက်စာတမ်းကို ဖတ်ရှုခြင်း

http://java.sun.com/j2se/1.4/docs/api/java/lang/String.html ကို သွားရောက်လည်ပတ်ပြီး charAt ကို click လုပ်ရင် အောက်မှာပြထားသလို စာရွက်စာတမ်းကို တွေ့ရပါမယ်။

```
public char charAt(int index)

Returns the character at the specified index.
An index ranges from 0 to length() - 1.

Parameters: index - the index of the character.

Returns: the character at the specified index of this string.
```

```
        The first character is at index 0.

Throws: StringIndexOutOfBoundException if the index is out of range.
```

ပထမဆုံးစာကြောင်းကို method ရဲ့ prototype လို့ခေါ်ပါတယ်။ Prototype ဆိုတာ method ရဲ့ အနေအထားကို ဖော်ပြပြီး method ရဲ့ နာမည်၊ parameter အမျိုးအစားနဲ့ ပြန်တဲ့တန်ဖိုးကို ပြောပြပါတယ်။

နောက်စာကြောင်းက method က လုပ်တဲ့အလုပ်ကို ပြောပြပြီး သူ့နောက်ကစာကြောင်းက parameter တွေနဲ့ ပြန်တဲ့တန်ဖိုးတွေအကြောင်းကို ရှင်းပြပါတယ်။ ဒီနေရာမှာ ရှင်းပြချက်တွေက မလိုအပ်ဘဲ နည်းနည်းတော့ ပိုနေပါတယ်။ ဒါပေမယ့် စာရွက်စာတမ်းဟာ စံပုံစံဝင်ဖို့လိုပါတယ်။ နောက်ဆုံးစာကြောင်းက ဒီ method ကနေ ဖြစ်ပေါ်စေနိုင်တဲ့ ခြွင်းချက်တွေရှိရင် သူတို့အကြောင်းကို ရှင်းပြပါတယ်။

## indexOf Method

indexOf ဟာ charAt ရဲ့ ပြောင်းပြန်နဲ့ တူပါတယ်။ charAt က index ကို လက်ခံပြီး အဲဒီ index မှာရှိတဲ့ အက္ခရာကို ထုတ်ပေးပါတယ်။ indexOf က အက္ခရာကို လက်ခံပြီး အဲဒီအက္ခရာပေါ်တဲ့ index တန်ဖိုးကို ရှာဖွေပါတယ်။ Index ဟာ စာသားအရှည်ရဲ့ ပြင်ပကိုရောက်နေရင် indexOf အလုပ်ဆက်မလုပ်နိုင်ဘဲ ခြွင်းချက်တစ်ခုကို ပစ်ပါတယ်။ စာသားထဲမှာ အက္ခရာမပါရင် indexOf ဟာ အလုပ်မပြီးမြောက်တဲ့အတွက် တန်ဖိုး -1 ကို ပြန်ထုတ်ပေးပါတယ်။

```
String fruit = "banana";
int index = fruit.indexOf('a');
```

ဒါဟာ စာသားထဲက အက္ခရာ 'a' ရဲ့ index ကို ရှာဖွေပါတယ်။ ဒီနေရာမှာ အက္ခရာဟာ သုံးကြိမ်ပေါ်ပြီး indexOf အနေနဲ့ ဘာလုပ်သင့်သလဲဆိုတာ သိပ်မသိသာပါဘူး။ စာရွက်စာတမ်းရဲ့အလိုအရ သူဟာ စပေါ်ပေါက်တဲ့ index ကို ထုတ်ပေးပါတယ်။

နောက်ထပ်ပေါ်လာတဲ့ index တွေကို ရှာချင်ရင် indexOf ရဲ့ နောက်ပုံစံတစ်မျိုး ရှိပါသေးတယ်။ ဒါကို overload လုပ်တယ်လို့ခေါ်ပြီး သူ့အကြောင်းကို သိပ်နားမလည်သေးရင် ပြီးခဲ့တဲ့ အပိုင်းတစ်ပိုင်းမှာ ပြန်ဖတ်နိုင်ပါတယ်။ သူက ဒုတိယ argument တစ်ခုကို လက်ခံပြီး စာသားရဲ့ ဘယ်နေရာမှာ စရှာရမလဲဆိုတာ တောင်းခံပါတယ်။

`int index = fruit.indexOf('a', 2);` ဆိုပြီးနှိုးဆွလိုက်ရင် index တန်ဖိုး 2 ရှိတဲ့ ပထမ 'n' နဲ့ စတင်ပြီး ဒုတိယ 'a' ကို ရှာဖွေပါလိမ့်မယ်။ သူက index 3 မှာ ရှိနေပါတယ်။ အက္ခရာဟာ အစ index မှာ ရှိနေရင် အဲဒီအစ index ဟာ အဖြေဖြစ်ပါတယ်။

ဒီတော့ အောက်မှာပြထားတဲ့ code ဟာ တန်ဖိုး 5 ကို ပြန်ထုတ်ပေးပါတယ်။

```
int index = fruit.indexOf('a', 5);
```

စာရွက်စာတမ်းအရ အစ index ဟာ စာသားအရှည်ရဲ့ ပြင်ပရောက်နေရင် ဘာဖြစ်လာမလဲဆိုတာ စဉ်းစားရတာ နည်းနည်းခက်ပါတယ်။ စာရွက်စာတမ်းမှာ ပြောထားတာက "indexOf သည် အက္ခရာ မပေါ်ပေါက်လျှင် fromIndex သို့မဟုတ် -1 ထက်ကြီးသော သို့မဟုတ် ငယ်သော ဂဏန်းဝတ္ထုမှ ကိုယ်စားပြုသည့် အက္ခရာအစီအစဉ်တွင် ပထမဆုံး ပေါ်ပေါက်သည့် index ကို ပြန်ထုတ်ပေးသည်။"

ဒါဟာ ဘာကိုဆိုလိုသလဲဆိုတာ သိနိုင်တဲ့နည်းက အခြေအနေတချို့ကို စမ်းသပ်ကြည့်ဖို့ပါပဲ။ အောက်မှာပြထားတာတွေက စမ်းသပ်ချက်တချို့ရဲ့ ရလဒ်တွေ ဖြစ်ပါတယ်။

- စတဲ့ index က length() ထက်ကြီးရင် ဒါမှမဟုတ် သူနဲ့တူရင် ရလဒ်ဟာ -1 ဖြစ်ပြီး အက္ခရာဟာ အစ index ထက်ပိုကြီးတဲ့ ဘယ်နေရာမှာမှ မပေါ်ပေါက်ဘူးလို့ ဆိုလိုပါတယ်။
- အစ index ဟာ **အနုတ်**ဖြစ်ရင် ရလဒ်ဟာ 1 ဖြစ်ပါတယ်။ ဆိုလိုတာက အက္ခရာ စ ပေါ်တဲ့နေရာဟာ **အစ** index ထက်ကြီးတဲ့ index တယ်လို့ ဆိုလိုပါတယ်။

စာရွက်စာတမ်းဆီ ပြန်သွားမယ်ဆိုရင် ဒီအပြုအမူဟာ အဓိပ္ပာယ်သတ်မှတ်ချက်နဲ့ ညီညွတ်တယ်ဆိုတာ ရုတ်တရက် မမြင်သာရင်တောင်မှ တွေ့ရပါလိမ့်မယ်။ indexOf ဘယ်လို အလုပ်လုပ်သလဲဆိုတာ သိတဲ့အခါ သူ့ကို ပရိုဂရမ်ရဲ့ တစ်စိတ်တစ်ပိုင်းအဖြစ် သုံးနိုင်ပါပြီ။

### သံသရာလည်ခြင်းနှင့် ရေတွက်ခြင်း

အောက်မှာပြထားတဲ့ ပရိုဂရမ်ဟာ စာသားတစ်ခုထဲမှာ အက္ခရာ 'a' ဘယ်နှခါပေါ်သလဲဆိုတာ ရေတွက်ပါတယ်။

```
String fruit = "banana";
int length = fruit.length();
int count = 0;

int index = 0;
while (index < length) {
   if (fruit.charAt(index) == 'a') {
         count = counnt + 1;
   }
   index = index + 1;
}
System.out.println(count);
```

ဒီပရိုဂရမ်ဟာ counter လို့ခေါ်တဲ့ အသုံးများဝေါဟာရတစ်ခုကို သရုပ်ဖော်ပါတယ်။ count

ဆိုတဲ့ variable ကို 0 အဖြစ်စတင်ပြီး 'a' ကိုတွေ့တဲ့အခါတိုင်း increment လုပ်ပါတယ်။ (Increment လုပ်တယ်ဆိုတာ တစ်တိုးတာကို ဆိုလိုပြီး decrement ရဲ့ ပြောင်းပြန်ဖြစ်ပါတယ်။) သံသရာကထွက်တဲ့အခါ count ဟာ ရလဒ်ကို ဆက်ထိန်းထားပြီး အဲဒီတန်ဖိုးဟာ a အရေအတွက် ဖြစ်ပါတယ်။

## Increment နှင့် Decremet လက္ခဏာများ

Increment လုပ်တာနဲ့ decrement လုပ်တာဟာ အသုံးများတဲ့ လုပ်ဆောင်ချက်တွေ ဖြစ်တဲ့အတွက် Java မှာ သူတို့အတွက် သီးသန့်လက္ခဏာတွေ ထည့်ပေးထားပါတယ်။ ++ လက္ခဏာဟာ int တစ်ခု ဒါမှမဟုတ် char တစ်ခုရဲ့တန်ဖိုးကို တစ်ပေါင်းပေးပါတယ်။ -- ဟာ တစ်နုတ်ပေးပါတယ်။ ဒီလက္ခဏာနှစ်ခုစလုံး double တွေ၊ boolean တွေနဲ့ String တွေပေါ်မှာ မလုပ်ကိုင်ပါဘူး။

နည်းပညာသဘောအရ variable တစ်ခုကို increment လုပ်ပြီး တစ်ပြိုင်တည်း ဖော်ပြချက်တစ်ခုမှာ ထည့်သုံးနိုင်ပါတယ်။ ဥပမာ အောက်မှာပြထားတဲ့ code ကို တရားဝင်သုံးနိုင်ပါတယ်။

```
System.out.println(i++);
```

ဒါကိုကြည့်ခြင်းအားဖြင့် increment လုပ်တာဟာ တန်ဖိုးကို print မလုပ်ခင် အရာရောက်မလား၊ လုပ်ပြီးမှ အရာရောက်မလားဆိုတာ မကွဲပြားပါဘူး။ ဒီလိုဖော်ပြချက်မျိုးတွေဟာ ရှုပ်ထွေးတတ်တဲ့အတွက် သူတို့ကို အသုံးပြုတာကို ကန့်ကွက်ပါတယ်။ ပိုပြီးကန့်ကွက်တဲ့အနေနဲ့ အဖြေဘာရမလဲဆိုတာကိုတောင် မပြောတော့ပါဘူး။ အရမ်းသိချင်လွန်းနေရင်တော့ ကိုယ့်ဘာသာကိုယ် လုပ်ကြည့်နိုင်ပါတယ်။

Increment လက္ခဏာတွေကိုသုံးပြီး အက္ခရာရေတွက်တဲ့ ပရိုဂရမ်ကို ပြန်ပြင်ရေးနိုင်ပါတယ်။

```
int index = 0;
while (index < length) {
  if (fruit.charAt(index) == 'a') {
      count++;
  }
  index++;
}
```

အောက်မှာပြောထားသလို မှားရေးတတ်ကြပါတယ်။

```
index = index++;          //WRONG!!
```

ကံမကောင်းစွာနဲ့ပဲ ဒါဟာ သဒ္ဒါသဘောတရားအရ တရားဝင်ပါတယ်။ ဒီတော့ compiler ဟာ သတိပေးတော့မှာ မဟုတ်ပါဘူး။ ဒီဖော်ပြချက်ရဲ့ အကျိုးသက်ရောက်မှုက index

ရဲ့တန်ဖိုးဟာ မပြောင်းမလဲဘဲ ရှိနေပါတယ်။ ဒါဟာ များသောအားဖြင့် လိုက်ရှာရခက်တဲ့ အမှားတစ်ခု ဖြစ်ပါတယ်။

`index = index + 1;` ကို ရေးနိုင်ပြီး `index++;` ကိုလည်း ရေးနိုင်ပါတယ်။ ဒါပေမယ့် သူတို့နှစ်မျိုးကို ရောရေးလို့မရပါဘူး။

## String များကို ပြုပြင်၍မရနိုင်ပါ

String method တွေရဲ့ စာရွက်စာတမ်းကိုပြန်ကြည့်ရင် toUpperCase နဲ့ toLowerCase ကို သတိပြုမိကြပါလိမ့်မယ်။ ဒီ method တွေဟာ တစ်ခါတစ်ရံမှာ မျက်စေ့ရှုပ်နားရှုပ်စရာ အကြောင်းအရင်းတစ်ခု ဖြစ်တတ်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူတို့ကိုကြားရတာ လက်ရှိစာသားတစ်ခုကို ပြောင်းလဲစေတဲ့ သက်ရောက်မှုရှိတယ်လို့ ထင်ရစေလို့ပါပဲ။ တကယ်တမ်းကျတော့ စာသားတစ်ခုကို ဘယ် method ကမှ မပြောင်းနိုင်ပါဘူး။ ဘာလို့လဲဆိုတော့ စာသားတွေဟာ ပြောင်းလဲလို့ မရနိုင်လို့ပါပဲ။ ဒါကို အင်္ဂလိပ်လို immutable ဖြစ်တယ်လို့ ခေါ်ပါတယ်။

String တစ်ခုပေါ်မှာ toUpperCase ကိုနှိုးဆွရင် String အသစ်တစ်ခုကို ပြန်တဲ့တန်ဖိုးအနေနဲ့ ရပါတယ်။ ဥပမာ

```
String name = "Alan Turing";
String upperName = name.toUpperCase();
```

ဒုတိယစာကြောင်းကို run ပြီးတဲ့အခါမှာ upperName မှာ "ALAN TURING" ဆိုတဲ့ တန်ဖိုးရှိနေပြီး name မှာတော့ "Alan Turing" ဆက်ရှိနေပါတယ်။

## String များကို နှိုင်းယှဉ်၍မရပါ

တစ်ခါတစ်ရံမှာ String တွေကို တူနေသလား၊ အက္ခရာအစီအစဉ်အလိုက် ဘယ်သူအရှေ့ရောက်သလဲဆိုတာ သိချင်တဲ့အတွက် နှိုင်းယှဉ်ဖို့ လိုလာပါတယ်။ နှိုင်းရလက္ခဏာတွေ ဖြစ်ကြတဲ့ == နဲ့ > တွေကို သုံးနိုင်ရင် အင်မတန်ကောင်းမှာ ဖြစ်ပါတယ်။ ဒါပေမယ့် သုံးလို့မရပါဘူး။

String တွေကို နှိုင်းယှဉ်ဖို့ equals နဲ့ compareTo method တွေကို သုံးရပါမယ်။ ဥပမာ

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";

if (name1.equals(name2)) {
   System.out.println("The names are the same.");
}
```

```
int flag = name1.compareTo(name2);
if (flag == 0) {
  System.out.println("The names are the same.");
} else if (flag < 0) {
  System.out.println("name1 comes before name2.");
} else if (flag > 0) {
  System.out.println("name2 comes before name1.");
}
```

ဒီသဒ္ဒါဟာ နည်းနည်းတော့ ထူးဆန်းနေပါတယ်။ အရာဝတ္ထုနှစ်ခုကို နှိုင်းယှဉ်ဖို့ သူတို့ထဲက တစ်ခုပေါ်မှာ method တစ်ခုကို နှိုးဆွရပြီး ကျန်တဲ့အရာဝတ္ထုကို argument တစ်ခု အနေနဲ့ ပို့ပေးရပါတယ်။

equals ကရတဲ့ ပြန်တဲ့တန်ဖိုးဟာ စာသားတွေမှာ တူညီတဲ့အက္ခရာတွေပါရင် true ဖြစ်ပြီး အဲဒီလိုမဟုတ်ရင် false ဖြစ်ပါတယ်။

compareTo ကနေ ပြန်ရတဲ့တန်ဖိုးကတော့ တစ်မျိုးဖြစ်နေပါတယ်။ ကွဲပြားတဲ့အရာကတော့ စာသားတွေရဲ့ ပထမဆုံးအက္ခရာတွေပဲ ဖြစ်ပါတယ်။ စာသားတွေတူကြရင် သူဟာ 0 ဖြစ်ပါတယ်။ အကယ်၍ method နှိုးဆွခံရတဲ့ ပထမစာသားဟာ အက္ခရာအစီအစဉ် ရှေ့ပိုင်းရောက်နေရင် ရလဒ်ဟာ အနုတ်ဖြစ်ပါတယ်။ မဟုတ်ရင် အပေါင်းဖြစ်ပါတယ်။ အခုအခြေအနေမှာ ပြန်တဲ့တန်ဖိုးဟာ အပေါင်း 8 ဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတော့ Ada ရဲ့ ဒုတိယအက္ခရာဟာ Alan ရဲ့ ဒုတိယအက္ခရာထက် အက္ခရာရှစ်လုံးရှေ့ ရောက်နေလို့ပါပဲ။

compareTo ကို သုံးတာဟာ နည်းနည်းခက်ခဲနိုင်ပါတယ်။ ပြန်မကြည့်ရဘဲ ဘယ်ဟာကဘာဆိုတာ သတိမရနိုင်ပါဘူး။ ဒါပေမယ့် ဒီလုပ်ပုံကိုင်ပုံဟာ ဝတ္ထုနှိုင်းယှဉ်မှု တော်တော်များများအတွက် စံပုံစံလိုတောင် ဖြစ်နေပါပြီ။ ဒီပုံစံကိုနားလည်တာနဲ့ ကျန်တာအားလုံးအတွက် အဆင်ပြေသွားနိုင်ပါတယ်။

ပိုပြီးပြည့်စုံသွားအောင် String တွေအတွက် == လက္ခဏာကို တရားဝင်သုံးနိုင်ပါတယ်။ ဒါပေမယ့် ဒီလိုလုပ်တာဟာ မှန်ခဲပါတယ်။ အခုပြောရခြင်းအဓိပ္ပာယ်ကို နောက်ပိုင်းကျမှ နားလည်လာပါလိမ့်မယ်။ အခုတော့ ဒီလိုမလုပ်ပါနဲ့လို့ပဲ ပြောပါရစေ။

အခန်း ၈

# စိတ်ဝင်စားဖွယ်ရာကောင်းသော ဝတ္ထုများ

**မည်သည့်နေရာတွင် စိတ်ဝင်စားဖွယ်ကောင်းသနည်း**

String တွေဟာ ဝတ္ထုတွေဖြစ်ကြပေမယ့် သူတို့ဟာ သိပ်ပြီးစိတ်ဝင်စားစရာကောင်းတဲ့ ဝတ္ထုတွေတော့ မဟုတ်ပါဘူး။ ဘာလို့လဲဆိုတော့

- သူတို့ကို ပြုပြင်လို့ မရနိုင်ပါဘူး။
- သူတို့မှာ instance variable တွေ မရှိပါဘူး။
- String ဝတ္ထုတစ်ခုကို ဖန်တီးဖို့ new command ကို သုံးစရာမလိုပါဘူး။

ဒီအခန်းမှာ Java ဘာသာစကားရဲ့ တစ်စိတ်တစ်ဒေသဖြစ်တဲ့ Point နဲ့ Rectangle ဝတ္ထုအသစ်နှစ်ခုကို အသုံးပြုကြပါတယ်။ အစကတည်းက ရှင်းလင်းအောင်ပြောချင်တာက တော့ ဒီအမှတ်တွေ၊ ထောင့်မှန်စတုဂံတွေဟာ မြင်ကွင်းပေါ်မှာပေါ်တဲ့ ဂရပ်ဖစ်ဝတ္ထုတွေ မဟုတ်ပါဘူး။ သူတို့ဟာ int တွေ၊ double တွေလို data သယ်ဆောင်တဲ့ variable တွေ ဖြစ်ကြပါတယ်။ တခြား variable တွေလိုပဲ သူတို့ကို တွက်ချက်မှုတွေလုပ်ဆောင်ဖို့ အတွင်းပိုင်းမှာ အသုံးပြုကြပါတယ်။

Point နဲ့ Ractangle class တွေရဲ့ အဓိပ္ပာယ်သတ်မှတ်ချက်တွေကို java.awt package ထဲမှာ တွေ့နိုင်ပြီး သူတို့ကို သွင်းယူရပါမယ်။

## Package များ

ထည့်သွင်းတည်ဆောက်ထားတဲ့ Java class တွေကို package တွေအဖြစ် ပိုင်းထားပါတယ်။ သူတို့ထဲမှာ အခုအချိန်အထိ မြင်ခဲ့တဲ့ class အားလုံးနီးပါးပါဝင်တဲ့ java.lang နဲ့ Java ရဲ့ Abstract Window Toolkit (AWT) ကို လိုက်နာတဲ့ class တွေပါဝင်တဲ့ java.awt တို့ ပါဝင်ပါတယ်။

Package တစ်ခုကိုသုံးဖို့ သူ့ကို သွင်းယူ (import) လုပ်ရပါမယ်။ ဒါကြောင့်မို့လို့ ဂရပ်ဖစ်ပရိုဂရမ်တွေဟာ import java.awt.* နဲ့ စတင်တာဖြစ်ပါတယ်။ * ရဲ့ အဓိပ္ပာယ်က AWT package ထဲက class တွေအားလုံးကို သွင်းယူမယ်လို့ ဆိုလိုပါတယ်။ စိတ်ဆန္ဒရှိရင် ကိုယ်သွင်းယူချင်တဲ့ class နာမည်ကို တိတိကျကျပြောပြီး သွင်းယူနိုင်ပါတယ်။ ဒါပေမယ့် ကြီးကြီးမားမား အကျိုးကျေးဇူးရယ်လို့တော့ မရှိပါဘူး။ java.lang ထဲက class တွေကို အလိုအလျောက်သွင်းယူပေးပါတယ်။ အဲဒါကြောင့်မို့လို့ လက်ရှိအချိန်အထိ ရေးဖူးခဲ့သမျှ ပရိုဂရမ်တွေမှာ import ဖော်ပြချက်ကို မလိုအပ်တာဖြစ်ပါတယ်။

Import ဖော်ပြချက်တွေအားလုံး class အဓိပ္ပာယ်သတ်မှတ်ချက်ရဲ့ ပြင်ပ ပရိုဂရမ်အစမှာ ရှိကြပါတယ်။

## Point ဝတ္ထုများ

အခြေခံအကျဆုံးအဆင့်မှာ အမှတ်တစ်ခုဟာ ကိုဩဒိနိတ်တွေကို ကိုယ်စားပြုတဲ့ ကိန်းနှစ်ခုပါဝင်ပြီး သူတို့ကို ဝတ္ထုတစ်ခုအနေနဲ့ စုစည်းဆက်ဆံပါတယ်။ သင်္ချာသင်္ကေတမှာ အမှတ်တွေကို လက်သည်းကွင်းတစ်စုံထဲမှာ comma ခြားပြီး ဖော်ပြလေ့ရှိပါတယ်။ ဥပမာ (0, 0) ဟာ မူလမှတ်ကို ကိုယ်စားပြုပြီး (x, y) ဟာ မူလမှတ်ကနေ ညာဖက်ကို x ယူနစ်ရွှေ့ပြီး အပေါ်ဘက်ကို y ယူနစ်ရွှေ့ထားတဲ့ အမှတ်ကို ဆိုလိုပါတယ်။

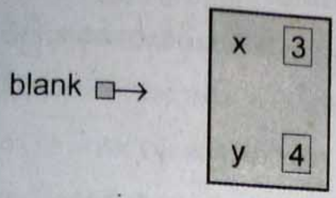
Java မှာ အမှတ်တစ်ခုကို Point ဝတ္ထုတစ်ခုနဲ့ ကိုယ်စားပြုပါတယ်။ အမှတ်အသစ်တစ်ခုကို ဖန်တီးဖို့ new command ကို အသုံးပြုရပါတယ်။

```
Point blank;
blank = new Point(3, 4);
```

ပထမစာကြောင်းဟာ ပုံမှန် variable ကြေညာချက်ဖြစ်ပါတယ်။ blank မှာ Point အမျိုးအစား ရှိပါတယ်။ ဒုတိယစာကြောင်းကတော့ နည်းနည်းရယ်စရာကောင်းပါတယ်။ သူက new command ကို နိူးဆွပြီး ဝတ္ထုအသစ်ရဲ့ အမျိုးအစားကို ဖော်ပြပါတယ်။ ပြီးတော့ argument တွေကို ပံ့ပိုးပါတယ်။ Argument တွေဟာ အမှတ်အသစ်ရဲ့ ကိုဩဒိနိတ်တွေဖြစ်ကြတဲ့ (3, 4) ဆိုတာတော့ အံ့ဩစရာမရှိပါဘူး။

new command ရဲ့ ရလဒ်ဟာ အမှတ်အသစ်ဆီ ရောက်ရှိတဲ့ အညွှန်း (reference) တစ်ခုဖြစ်ပါတယ်။ အညွှန်းတွေအကြောင်း နောက်မှရှင်းပြပါမယ်။ အခုချိန်မှာတော့ အရေးကြီးတဲ့ အချက်က blank ဆိုတဲ့ variable မှာ အသစ်ဖန်တီးလိုက်တဲ့ဝတ္ထုဆီ အညွှန်းတစ်ခု ပါဝင်ပါ

တယ်။ ဒီနေရာချထားမှုကို ပုံဆွဲတဲ့စံပုံ နည်းတစ်နည်းရှိပါတယ်။ ဒါကို ပုံမှာပြထားပါတယ်။




ပုံမှန်လိုပဲ blank ဆိုတဲ့ variable ရဲ့နာမည်ဟာ လေးထောင့်ကွက်ရဲ့ ပြင်ပမှာပေါ်ပြီး သူ့ရဲ့တန်ဖိုးတွေကို လေးထောင့်ကွက်ထဲမှာ ထည့်ထားပါတယ်။ ဒီနေရာမှာ အဲဒီတန်ဖိုးဟာ အညွှန်းတစ်ခုဖြစ်ပြီး ဒါကို နောက်ပိတ်ပါတဲ့ မြားတစ်ခုနဲ့ သရုပ်ဖော်ထားပါတယ်။ မြားဟာ blank က ညွှန်းဆိုတဲ့ဝတ္ထုကို ညွှန်ပြပါတယ်။

လေးထောင့်ကွက်အကြီးကြီးက တန်ဖိုးနှစ်ခုပါတဲ့ဝတ္ထုအသစ်ကို ပြသပါတယ်။ x နဲ့ y ဟာ instance variable တွေရဲ့ နာမည်တွေဖြစ်ပါတယ်။

ပရိုဂရမ်တစ်ပုဒ်ထဲက variable တွေ၊ တန်ဖိုးတွေနဲ့ ဝတ္ထုတွေကို ခြုံပြီး အခြေအနေ (state) လို့ ခေါ်ပါတယ်။ ပရိုဂရမ်ရဲ့အခြေအနေကိုပြတဲ့ ဒီလိုပုံမျိုးကို အခြေအနေပုံ (state diagram) လို့ ခေါ်ပါတယ်။ ပရိုဂရမ် run တဲ့အခါ အခြေအနေပြောင်းသွားပါတယ်။ ဒီတော့ အခြေအနေပုံနှစ်ခုကို run နေတဲ့ ပရိုဂရမ်တစ်ခုရဲ့ အချိန်ကာလတစ်ခုမှာ ဓာတ်ပုံဖမ်းရိုက်ထားတဲ့ ပုံအနေနဲ့ မြင်သင့်ပါတယ်။

## Instance Variable များ

ဝတ္ထုကို ဖွဲ့စည်းထားတဲ့ data အပိုင်းအစတွေကို တစ်ခါတစ်ရံမှာ component တွေ၊ record တွေ၊ field တွေလို့လည်း ခေါ်တတ်ကြပါတယ်။ Java မှာ သူတို့ကို instance variable လို့ ခေါ်ပါတယ်။ ဘာလို့လဲဆိုတော့ အမျိုးအစားတစ်ခုရဲ့ ဥပမာ (instance) တစ်ခုဖြစ်တဲ့ ဝတ္ထုတိုင်းမှာ instance variable တွေရဲ့ ကိုယ်ပွားတွေကို ပိုင်ဆိုင်လို့ပါပဲ။

ဒါဟာ ကားတစ်စီးရဲ့ ပစ္စည်းသိမ်းတဲ့အကန့်နဲ့ တူပါတယ်။ ကားတိုင်းဟာ ကားဆိုတဲ့အမျိုးအစားရဲ့ ဥပမာတွေဖြစ်ကြပါတယ်။ ကားတစ်စီးစီမှာလည်း သူ့ကိုယ်ပိုင် ပစ္စည်းသိမ်းတဲ့ အကန့်ရှိပါတယ်။ လူတစ်ယောက်က ကျွန်တော့်ကားထဲက ပစ္စည်းတစ်ခုခုကို ယူပေးပါလို့ အကူအညီတောင်းလာရင် ဘယ်ကားက သူ့ကားလဲလို့ ပြောဖို့လိုလာပါလိမ့်မယ်။

ဒီလိုပဲ instance variable က တန်ဖိုးတစ်ခုကိုဖတ်ချင်ရင် ဘယ်ဝတ္ထုကနေ အဲဒီတန်ဖိုးကို လိုချင်သလဲဆိုတာ ပြောဖို့လိုပါလိမ့်မယ်။ Java မှာဒါကို အစက်သင်္ကေတနဲ့ ပြုလုပ်ပါတယ်။

```
int x = blank.x;
```

blank.x ဖော်ပြချက်ရဲ့ ဆိုလိုရင်းက blank ကညွှန်ပြနေတဲ့ ဝတ္ထုဆီသွားပြီး x ရဲ့ တန်ဖိုးကိုရယူပါလို့ပဲ ဖြစ်ပါတယ်။ ဒီနေရာမှာ အဲဒီတန်ဖိုးကို ဒေသခံ variable x မှာ နေရာချထားလိုက်ပါတယ်။ ဒီနေရာမှာ x ဆိုတဲ့ ဒေသခံ varaible နဲ့ x လို့နာမည်ပေးထားတဲ့ instance

variable ကြားမှာ ပဋိပက္ခမရှိဘူးဆိုတာ သတိပြုပါ။ အစက်သင်္ကေတရဲ့ ရည်ရွယ်ချက်က ဘယ် variable ကို ညွှန်းဆိုနေသလဲဆိုတာ ဒွိဟမဖြစ်စေဘဲ ခွဲခြားနိုင်ဖို့ဖြစ်ပါတယ်။

သင်္ကေတကို ဘယ် Java ဖော်ပြချက်မျိုးမဆိုရဲ့ အစိတ်အပိုင်းတစ်ခုအဖြစ် သုံးနိုင်ပါတယ်။ ဒီတော့ အောက်မှာပြောထားတာတွေက တရားဝင်ပါတယ်။

```
System.out.println(blank.x + ", " + blank.y);
int distance = blank.x * blank.x + blank.y * blank.y;
```

ပထမစာကြောင်းက 3, 4 ကို print လုပ်ပြီး ဒုတိယစာကြောင်းက 25 ဆိုတဲ့တန်ဖိုးကို တွက်ထုတ်ပေးပါတယ်။

## ဝတ္ထုများကို Parameter များအဖြစ်သုံးခြင်း

ဝတ္ထုတွေကို ပုံမှန်အတိုင်း parameter တွေအဖြစ် ပေးပို့နိုင်ပါတယ်။ ဥပမာ

```
public static void printPoint(Point p) {
    System.out.pirntln("(" + p.x + ", " + p.y + ")");
}
```

သူဟာ အမှတ်တစ်ခုကို argument တစ်ခုအနေနဲ့ယူပြီး စံပုံစံနဲ့ print လုပ်ပေးတဲ့ method တစ်ခုဖြစ်ပါတယ်။ printPoint(p) ကိုနှိုးဆွရင် (3, 4) ကို print လုပ်မှာဖြစ်ပါတယ်။ တကယ်တမ်းကြတော့ Java မှာ Point တွေကို print လုပ်ပေးတဲ့ method တစ်ခုကို ထည့်သွင်းတည်ဆောက်ထားပါတယ်။ System.out.println(blank) ကို နှိုးဆွရင် java.awt.Point[x=3,y=4] ကိုရပါလိမ့်မယ်။

ဒါဟာ Java က ဝတ္ထုတွေကို print လုပ်ရင်သုံးတဲ့ စံပုံစံဖြစ်ပါတယ်။ သူက အမျိုးအစားရဲ့နာမည်ကို print လုပ်ပြီး သူ့နောက်မှာ ဝတ္ထုရဲ့ပါဝင်မှုတွေ လိုက်ပါတယ်။ ပါဝင်မှုတွေမှာ instance variable တွေရဲ့ နာမည်တွေနဲ့ တန်ဖိုးတွေပါဝင်ပါတယ်။

ဒုတိယဥပမာအနေနဲ့ ပြီးခဲ့တဲ့အပိုင်းတုန်းက distance method ကို ပြန်ပြင်ရေးပြီး double လေးခုအစား Point နှစ်ခုကို parameter အဖြစ် လက်ခံအောင် လုပ်နိုင်ပါတယ်။

```
public static double distance(Point p1, Point p2) {
    double dx = (double)(p2.x - p1.x);
    double dy = (double)(p2.y - p1.y);
    return Math.sqrt(dx*dx + dy*dy);
}
```

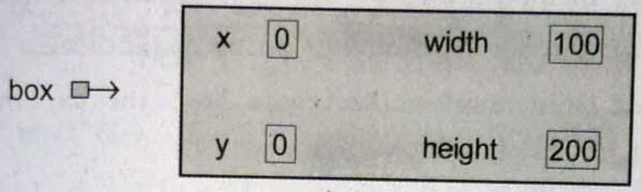
Typecast တွေကို မရှိမဖြစ်လိုတာတော့ မဟုတ်ပါဘူး။ Point တစ်ခုထဲက instance variable တွေဟာ ကိန်းပြည့်တွေဖြစ်တယ်လို့ သတိပေးတဲ့အနေနဲ့ သူတို့ကို ထည့်ထားတာဖြစ်ပါတယ်။

#### ထောင့်မှန်စတုဂံများ

Rectangle တွေဟာ အမှတ်တွေနဲ့ ဆင်တူပါတယ်။ သူ့မှာ instance variable လေးခုပါတာပဲ ပိုထူးပါတယ်။ Instance varaible လေးခုဟာ x, y, width နဲ့ height ဖြစ်ပါတယ်။ ဒါတွေကလွဲရင် ကျန်တာအားလုံးဟာ အတူတူနီးပါး ဖြစ်ပါတယ်။

```
Rectangle box = new Rectangle(0, 0, 100, 200);
```

အပေါ်မှာပြထားတဲ့ code ဟာ Rectangle ဝတ္ထုအသစ်တစ်ခုကို ဖန်တီးပြီး box ကို ဒီဝတ္ထုဆီသွားတဲ့အညွှန်း တပ်ပေးလိုက်ပါတယ်။ အောက်မှာပြထားတဲ့ပုံဟာ ဒီနေရာချထားမှုရဲ့ သက်ရောက်မှုကို သရုပ်ဖော်ပါတယ်။




box ကို print လုပ်ရင် java.awt.Rectangle[x=0,y=0,width=100,height=200] ကိုရပါတယ်။ ဒီနေရာမှာလည်း ဒါဟာ Rectangle ဝတ္ထုတွေကို ဘယ်လို pirnt လုပ်ရမလဲဆိုတာသိတဲ့ Java ရဲ့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ method ရဲ့ ရလဒ်ပဲဖြစ်ပါတယ်။

#### ဝတ္ထုများကို ပြန်သောအမျိုးအစားများအဖြစ်သုံးခြင်း

Method တွေကို ဝတ္ထုတွေပြန်ပေးအောင် ရေးသားနိုင်ပါတယ်။ ဥပမာ findCentre ဟာ Rectangle ကို argument တစ်ခုအနေနဲ့ယူပြီး Rectangle ရဲ့ဗဟိုမှတ်ရဲ့ ကိုသြဒိနိတ်တွေပါဝင်တဲ့ Point တစ်ခုကို ပြန်ထုတ်ပေးပါတယ်။

```
public static Point findCentre(Rectangle box) {
    int y = box.x + box.width/2;
    int y = box.y + box.height/2;
    return new Point(x, y);
}
```

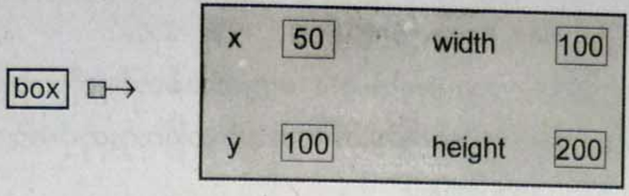
new ကိုသုံးပြီး ဝတ္ထုအသစ်တစ်ခု ဖန်တီးနေချိန်မှာပဲ ထွက်လာတဲ့ရလဒ်ကို ပြန်တဲ့တန်ဖိုးအနေနဲ့ တစ်ပြိုင်တည်းသုံးနိုင်တယ်ဆိုတာ သတိပြုပါ။

#### ဝတ္ထုတွေကို ပြုပြင်နိုင်ပါတယ်

ဝတ္ထုတစ်ခုရဲ့ပါဝင်မှုတွေကို သူ့ရဲ့ instance variable တစ်ခုမှာ နေရာချထားမှု တစ်ခုလုပ်ပြီး ပြောင်းလဲနိုင်ပါတယ်။ ဥပမာ ထောင့်မှန်စတုဂံတစ်ခုကို ပုံမပျက်ဘဲရွှေ့ချင်ရင် x, y တန်ဖိုးတွေကို ပြုပြင်နိုင်ပါတယ်။

```
box.x = box.x + 50;
box.y = box.y + 100;
```

ရလဒ်တွေကို အပေါ်မှာ ပြထားပါတယ်။




ဒီ code ကိုယူပြီး method တစ်ခုထဲမှာ အုပ်ထည့်ထားနိုင်ပါတယ်။ ပြီးတော့ ထောင့်မှန်စတုဂံကို ကြိုက်တဲ့ပမာဏသုံးပြီး ရွှေ့နိုင်အောင် ယေဘုယျပြုနိုင်ပါတယ်။

```
public static void moveRect(Rectangle box, int dx, int dy) {
    box.x = box.x + dx;
    box.y = box.y + dy;
}
```

dx နဲ့ dy ဆိုတဲ့ variable တွေဟာ ထောင့်မှန်စတုဂံကို ဦးတည်ချက်တစ်ခုစီအတွက် ဘယ်လောက်အကွာအဝေး ရွှေ့မလဲဆိုတာ ဖော်ပြပါတယ်။ ဒီ method ကို နှိုးဆွခြင်းအားဖြင့် argument အဖြစ် ပေးပို့ခံရတဲ့ Rectangle ကို ပြုပြင်ပေးတဲ့ သက်ရောက်မှုကို ရစေပါတယ်။

```
Rectangle box = new Rectangle(0, 0, 100, 200);
moveRect(box, 50, 100);
System.out.println(box);
```

ဆိုတဲ့ code ဟာ `java.awt.Rectangle[x=50,y=100,width=100,height=200]` ကို print လုပ်ပေးပါတယ်။

ဝတ္ထုတွေကို argument တွေအနေနဲ့ method တွေဆီပို့ပြီး ပြုပြင်ခိုင်းတာဟာ အသုံးဝင်နိုင်ပါတယ်။ ဒါပေမယ့် ဒီလိုလုပ်ခြင်းအားဖြင့် အမှားပြင်ရတာ ပိုခက်စေပါတယ်။ ဘာလို့လဲဆိုတော့ ဘယ် method တွေက သူတို့ရဲ့ argument တွေကို ပြုပြင်သလဲ၊ မပြုပြင်ဘဲ ထားသလဲ ဆိုတာ မကွဲပြားလို့ဘဲ ဖြစ်ပါတယ်။ ဒီပရိုဂရမ်ရေးဟန်ရဲ့ အားသာချက်၊ အားနည်းချက်တွေကို နောက်ပိုင်းကျရင် ဆွေးနွေးကြပါမယ်။

ဒီကြားထဲမှာတော့ Java ရဲ့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ method တွေရဲ့ အဆင်ပြေမှုကို ခံစားနိုင်ပါတယ်။ သူတို့ထဲမှာ နှိုးဆွရတဲ့ သဒ္ဒါနည်းနည်းကွဲပေမယ့် moveRect နဲ့ လုပ်ကိုင်ပုံတူတဲ့ translate လည်း ပါဝင်ပါတယ်။ Rectangle ကို argument တစ်ခုအနေနဲ့ ပို့ပေးမယ့်အစား dx နဲ့ dy ကိုပဲ argument အနေနဲ့ပေးပို့ပြီး Rectangle ပေါ်မှာ translate ကို နှိုးဆွနိုင်ပါတယ်။

```
box.translate(50, 100);
```

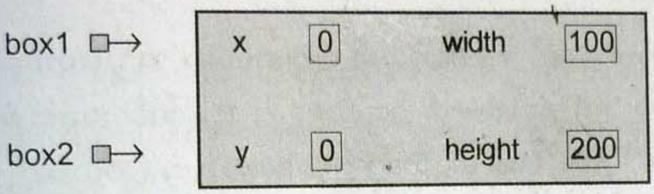
သက်ရောက်မှုကတော့ အတူတူပဲ ဖြစ်ပါတယ်။

### နာမည်နှစ်ခုပေးခြင်း

ဝတ္ထု variable တစ်ခုမှာ နေရာချထားမှုတစ်ခုကို ပြုလုပ်ရင် အဲဒီဝတ္ထုမှာ နေရာချထားနေတာ ဖြစ်ပါတယ်။ ဝတ္ထုတစ်ခုတည်းကို ညွှန်းဆိုတဲ့ variable တွေ အများကြီးထားနိုင်ပါတယ်။ ဥပမာ အောက်ကပြထားတဲ့ code ကို လေ့လာပါ။

```
Rectangle box1 = new Rectangle(0, 0, 100, 200);
Rectangle box2 = box1;
```

ဒီ code ဟာ အောက်မှာပြထားတဲ့ အခြေအနေပုံမျိုးကို ရရှိစေပါတယ်။



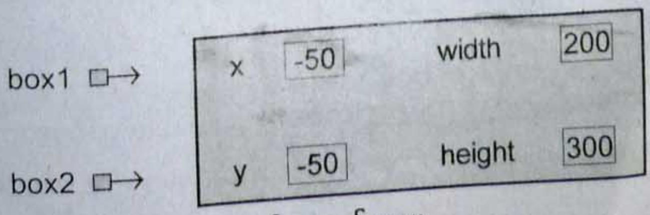

box1 နဲ့ box2 နှစ်ခုစလုံးဟာ တူညီတဲ့ဝတ္ထုတစ်ခုကို ညွှန်းဆိုပါတယ်။ ဒါကို C++ စကားလုံးနဲ့ပြောရရင် ညွှန်ပြတယ် (point) လို့လည်း ဆိုနိုင်ပါတယ်။ Java မှာ ညွှန်ပြမှုတွေပြုလုပ်တဲ့ pointer တွေမပါတဲ့အတွက် ညွှန်းတယ် (refer) ဆိုတဲ့စကားဟာ ပိုပြီးအဓိပ္ပာယ်ရှိပါတယ်။

စကားပြန်ဆက်ရရင် အဲဒီတူညီတဲ့ဝတ္ထုမှာ box1 နဲ့ box2 ဆိုပြီး နာမည်နှစ်ခုရှိပါတယ်။ နာမည်နှစ်ခုပေးတာကို အင်္ဂလိပ်လို aliase လုပ်တယ်လို့ ခေါ်ပါတယ်။

Variable နှစ်ခုကို aliase လုပ်ရင် တစ်ခုမှာ ပြုပြင်မှုလုပ်တိုင်း နောက်တစ်ခုမှာလည်း သက်ရောက်မှုရှိပါတယ်။ ဥပမာ

```
System.out.println(box2.width);
box1.grow(50, 50);
System.out.println(box2.width);
```

ပထမစာကြောင်းဟာ 100 ကို print လုပ်ပြီး သူဟာ box2 ကညွှန်းဆိုတဲ့ Rectangle ရဲ့ အနံဖြစ်ပါတယ်။ ဒုတိယစာကြောင်းဟာ box1 ပေါ်မှာ grow method ကိုနှိုးဆွပြီး Rectangle ကို အရပ်မျက်နှာနှစ်ဖက်မှာ 50 pixels ပြန့်ကားစေပါတယ်။ အကျိုးသက်ရောက်မှုကို ပုံမှာပြထားပါတယ်။



ဒီပုံကိုကြည့်ပြီး ရှင်းသွားသင့်တာက box1 မှာပြုလုပ်တဲ့ အပြောင်းအလဲတိုင်းဟာ box2 မှာလည်း သက်ရောက်မှုရှိစေပါတယ်။ ဒီတော့ တတိယစာကြောင်းမှာ print အလုပ်ခံရတဲ့ တန်ဖိုးဟာ 200 ဖြစ်ပြီး တိုးချဲ့ထားတဲ့ ထောင့်မှန်စတုဂံရဲ့ အနံဖြစ်ပါတယ်။ (Rectangle တစ်ခုရဲ့ ကိုဩဒိနိတ်တွေဟာ အနုတ်တန်ဖိုးလည်း ဖြစ်နိုင်ပါတယ်။)

ဒီရိုးရှင်းတဲ့ဥပမာကနေ ပြောနိုင်တာတစ်ခုကတော့ နာမည်တစ်ခုထက်ပိုပေးတာတွေ ပါဝင်တဲ့ code ဟာ အလွယ်တကူ နားလည်ရခက်စေပြီး အမှားပြင်ရ ရှုပ်ထွေးစေပါတယ်။ ပုံမှန်အနေအထားမှာ နာမည်တစ်ခုထက်ပိုပေးတာကို ရှောင်သင့်ရင်ရှောင်ပြီး သုံးရင်လည်း သတိနဲ့သုံးရပါမယ်။

## Null

ဝတ္ထု variable တစ်ခုကို ဖန်တီးတဲ့အခါ ဝတ္ထုတစ်ခုမှာ အညွှန်းတစ်ခုကို ပြုလုပ်နေတယ်ဆိုတာကို သတိရပါ။ အဲဒီ variable ကို ဝတ္ထုတစ်ခုမှာ ညွှန်ပြမခိုင်းမချင်း variable ဟာ null ဖြစ်နေပါတယ်။ null ဟာ Java မှာ ဘာဝတ္ထုမှမရှိဘူးလို့ အဓိပ္ပာယ်ရတဲ့ အထူးတန်ဖိုးတစ်ခုဖြစ်ပြီး keyword တစ်ခုလည်း ဖြစ်ပါတယ်။

Point blank; ဆိုတဲ့ ကြေညာချက်ဟာ Point blank = null; ဆိုတဲ့ စတင်မှုနဲ့ တူညီပြီး အောက်မှာပြထားတဲ့ အခြေအနေပုံနဲ့ သရုပ်ဖော်ပါတယ်။

blank □

null တန်ဖိုးကို မြားမပါတဲ့ အစက်တစ်စက်နဲ့ သရုပ်ဖော်ပါတယ်။

null ဝတ္ထုတစ်ခုကို instance variable သုံးပြီးဖြစ်ဖြစ်၊ method နှိုးဆွပြီးဖြစ်ဖြစ် သုံးဖို့ကြိုးစားရင် NullPointerException တစ်ခုကို ရပါမယ်။ ဒီစနစ်ဟာ အမှား message တစ်ခုကို print လုပ်ပြီး ပရိုဂရမ်ကို ရပ်တန့်စေပါလိမ့်မယ်။

```
Point blank = null;
int x = blank.x;             // NullPointerException
blank.translate(50, 50);     // NullPointerException
```

အခြားတစ်ဖက်မှာတော့ null ဝတ္ထုတစ်ခုကို argument တစ်ခုအနေနဲ့ တရားဝင်ပေးပို့နိုင်ပြီး ပြန်တဲ့တန်ဖိုးတစ်ခုအနေနဲ့လည်း တရားဝင်လက်ခံနိုင်ပါတယ်။ တကယ်တမ်းကျတော့ ဒီလိုလုပ်တာဟာ အသုံးတောင် များနေပါတယ်။ ဥပမာ empty set ကို ကိုယ်စားပြုဖို့ ဒါမှမဟုတ် အမှားအခြေအနေတစ်ခုကို ညွှန်းဆိုဖို့ သုံးလေ့ရှိကြပါတယ်။

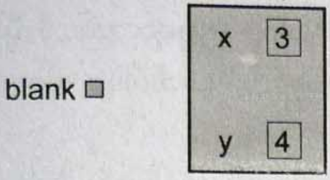
### အမှိုက်သိမ်းခြင်း

ပြီးခဲ့တဲ့ တစ်ပိုင်းကျော်မှာတုန်းက ဝတ္ထုတစ်ခုကို variable တစ်ခုထက်ပိုပြီး ညွှန်းဆိုရင် ဘာဖြစ်သလဲဆိုတာ ပြောခဲ့ပါတယ်။ ဝတ္ထုတစ်ခုကို ဘယ် variable ကမှ မညွှန်းဆိုတော့ရင်

ဘယ်လိုဖြစ်သွားမလဲ။ ဥပမာ

```
Point blank = new Point(3, 4);
blank = null;
```

ပထမစာကြောင်းဟာ Point ဝတ္ထုအသစ်တစ်ခုကို ဖန်တီးပြီး blank ကို သူ့မှာ ညွှန်းခိုင်းလိုက်ပါတယ်။ ဒုတိယစာကြောင်းဟာ အဲဒီဝတ္ထုကို ညွှန်းဆိုနေမယ့်အစား ဘာကိုမှ မညွှန်းဆိုအောင် blank ကို ပြုပြင်လိုက်ပါတယ်။




ဝတ္ထုတစ်ခုကို ဘယ်သူကမှ မညွှန်းဆိုတော့ရင် ဘယ်သူကမှ သူ့ရဲ့တန်ဖိုးတွေကို မရေးနိုင် မဖတ်နိုင်တော့ပါဘူး။ သူ့ပေါ်မှာ method တစ်ခုကို နှိုးဆွလို့လည်း မရတော့ပါဘူး။ သူ့ရဲ့ရှင်သန်မှု ရပ်ဆိုင်းသွားတယ်လို့ ဆိုလိုနိုင်ပါတယ်။ ဒီဝတ္ထုကို မှတ်ဉာဏ်ထဲမှာ ဆက်သိမ်းထားလို့ရပါတယ်။ ဒါပေမယ့် ဒါဟာ နေရာပြုန်းတီးစေပါတယ်။ ဒီတော့ ပရိုဂရမ် run နေချိန်မှာ အချိန်အပိုင်းအခြားတစ်ခုအတိုင်း Java စနစ်ဟာ ဘေးရောက်နေတဲ့ ဝတ္ထုတွေကို လိုက်ရှာပြီး ပြန်သိမ်းယူပါတယ်။ ဒီဖြစ်စဉ်ကို အမှိုက်သိမ်းတယ် (garbage collection) လို့ ခေါ်ပါတယ်။ ဒီလိုလုပ်ပြီးနောက်မှာ ဝတ္ထုကယူထားတဲ့ မှတ်ဉာဏ်နေရာကို နောက်ထပ် ဝတ္ထုအသစ်တစ်ခုရဲ့ တစ်စိတ်တစ်ဒေသအဖြစ် ပြန်သုံးနိုင်အောင် လုပ်ထားပါတယ်။

**ဝတ္ထုများနှင့် အခြေခံအမျိုးအစားများ**

Java မှာ အမျိုးအစား နှစ်မျိုးရှိပါတယ်။ အခြေခံအမျိုးအစား (primitive type) နဲ့ ဝတ္ထုအမျိုးအစား (object type) ဆိုပြီး ဖြစ်ပါတယ်။ int နဲ့ boolean တို့လို အခြေခံတွေဟာ စာလုံးသေးတွေနဲ့စပြီး ဝတ္ထုအမျိုးအစားတွေဟာ အက္ခရာအကြီးနဲ့ စတင်ပါတယ်။ ဒီခွဲခြားမှုဟာ အရေးပါပါတယ်။ ဘာလို့လဲဆိုတော့ သူတို့နှစ်မျိုးကြားက ကွာဟချက်တွေကို သတိပေးသလို ဖြစ်နေလို့ပါပဲ။

- အခြေခံ variable တစ်ခုကို ကြေညာတဲ့အခါမှာ အခြေခံတန်ဖိုးတစ်ခုအတွက် သိမ်းဆည်းမှုနေရာကို ရပါတယ်။ ဝတ္ထု variable တစ်ခုကို ကြေညာရင် ဝတ္ထုတစ်ခုဆီသွားတဲ့အညွှန်းအတွက် နေရာရပါတယ်။ ဝတ္ထုအတွက် နေရာယူချင်ရင် new command ကို သုံးရပါမယ်။
- အခြေခံအမျိုးအစားတစ်ခုကို မစတင်ခဲ့ရင် အမျိုးအစားပေါ်မူတည်ပြီး ပုံမှန်တန်ဖိုးတစ်ခုကို ပေးပါလိမ့်မယ်။ ဥပမာ int တွေအတွက် 0 ကိုပေးပြီး boolean တွေအတွက် true ကိုပေးပါတယ်။ ဝတ္ထုအမျိုးအစားတွေအတွက် ပုံမှန်တန်ဖိုးကတော့ ဘာ

ဝတ္ထုမှ မရှိဘူးဆိုတာပြောတဲ့ null ဖြစ်ပါတယ်။

- အခြေခံ variable တွေဟာ သီးသန့်တည်ရှိနိုင်ပါတယ်။ Method တစ်ခုက ဆောင်ရွက်တဲ့အလုပ်တစ်ခုဟာ တခြား method မှာရှိတဲ့ variable ကို ပြောင်းလဲစေစရာ အကြောင်းမရှိပါဘူး။ ဝတ္ထု variable တွေဟာ အလုပ်လုပ်ရတာ နည်းနည်းပဟေဠိဆန်နိုင်ပါတယ်။ အကြောင်းက သူတို့ဟာ သီးသန့်တည်ရှိမှု မရှိတာကြောင့်ပါပဲ။ ဝတ္ထုအညွှန်းတစ်ခုကို argument တစ်ခုအနေနဲ့ ပေးပို့ရင် ကိုယ်နှိုးဆွတဲ့ method ဟာ ဝတ္ထုကို ပြုပြင်ကောင်း ပြုပြင်ပါလိမ့်မယ်။ ဒီနေရာမှာ အကျိုးသက်ရောက်မှုကို မြင်ရပါမယ်။ ဝတ္ထုတစ်ခုပေါ်မှာ method တစ်ခုကို နှိုးဆွရင်လည်း ဒီလိုကိစ္စမျိုး ဖြစ်ပေါ်ပါတယ်။ ဒါဟာ ကောင်းမွန်တဲ့အရာတစ်ခုတော့ ဟုတ်ပါရဲ့။ ဒါပေမယ့် သူ့ကို သတိနဲ့သုံးဖို့တော့ လိုပါတယ်။

အခြေခံတွေနဲ့ ဝတ္ထုအမျိုးအစားတွေကြားမှာ ကွာခြားချက်တစ်ခု ရှိနေပါသေးတယ်။ Java ဘာသာစကားမှာ အခြေခံအသစ်တွေ ကိုယ့်ဘာသာကိုယ် သွားပေါင်းထည့်လို့ မရပါဘူး။ (Java မူဝါဒကော်မတီအဖွဲ့ဝင် မဟုတ်ရင်ပေါ့။) ဒါပေမယ့် ဝတ္ထုအမျိုးအစား အသစ်တွေကိုတော့ ကိုယ့်ဘာသာကိုယ် ဖန်တီးနိုင်ပါတယ်။ ဘယ်လိုလုပ်ရမလဲဆိုတာ ရှေ့အခန်းမှာ သိသွားပါလိမ့်မယ်။

**အခန်း ၄**

# ကိုယ်ပိုင်ဝတ္ထုများတည်ဆောက်ခြင်း

## Class အဓိပ္ပာယ်သတ်မှတ်ချက်များနှင့် ဝတ္ထုအမျိုးအစားများ

Class အဓိပ္ပာယ်သတ်မှတ်ချက်တစ်ခုကို ရေးတဲ့အခါတိုင်း အဲဒီ class နဲ့ နာမည်တူတဲ့ ဝတ္ထုအမျိုးအစားအသစ်တစ်ခုကို ဖန်တီးပါတယ်။ အစပိုင်းမှာတုန်းက Hello လို့ နာမည်ပေးထားတဲ့ class ကို သတ်မှတ်ခဲ့ချိန်မှာ Hello နာမည်ရှိတဲ့ ဝတ္ထုအမျိုးအစားကို ဖန်တီးခဲ့ပါတယ်။ Hello အမျိုးအစားရှိတဲ့ variable တွေကို မဖန်တီးခဲ့သလို Hello ဝတ္ထုတွေကို ဖန်တီးဖို့ new command ကို မသုံးခဲ့ပါဘူး။ ဒါပေမယ့် ဒါတွေအားလုံးကို လုပ်ချင်ရင် လုပ်ခဲ့နိုင်ပါတယ်။

ဒီဥပမာဟာ အဓိပ္ပာယ်မရှိပါဘူး။ ဘာလို့လဲဆိုတော့ Hello ဝတ္ထုတစ်ခုကို ဖန်တီးစရာအကြောင်းမရှိလို့ပါပဲ။ လုပ်ခဲ့ရင်တောင် ဘယ်နေရာမှာ သွားကောင်းသလဲဆိုတာ မရှင်းလင်းပါဘူး။ ဒီအခန်းမှာတော့ အသုံးဝင်တဲ့ ဝတ္ထုအမျိုးအစားအသစ်တွေကို ဖန်တီးတဲ့ class အဓိပ္ပာယ်သတ်မှတ်ချက် ဥပမာတချို့ကို ကြည့်ရှုရပါမယ်။

ဒီအခန်းမှာ အရေးကြီးတဲ့ အယူအဆတချို့ ရှိပါတယ်။

- Class အသစ်တစ်ခုကို သတ်မှတ်တဲ့အခါ နာမည်တူတဲ့ ဝတ္ထုအမျိုးအစား အသစ်တစ်ခုကို ဖန်တီးပါတယ်။

- Class သတ်မှတ်ချက်တစ်ခုဟာ ဝတ္ထုတွေကိုပုံလောင်းတဲ့ ပုံစံခွက်တစ်ခုနဲ့ တူပါတယ်။ သူက ဝတ္ထုတွေပိုင်ဆိုင်မယ့် instance variable တွေကို သတ်မှတ်ပေးပြီး သူတို့အပေါ်မှာ ဘယ်လို method မျိုးတွေက လုပ်ဆောင်ချက်တွေ ပြုလုပ်နိုင်မလဲ ဆိုတာကိုလည်း ဆုံးဖြတ်ပါတယ်။
- ဝတ္ထုတိုင်းဟာ ဝတ္ထုအမျိုးအစားတစ်ခုခုနဲ့ ပတ်သက်ပါတယ်။ ဒီတော့ သူဟာ class တစ်ခုရဲ့ instance တစ်ခု ဖြစ်ပါတယ်။
- ဝတ္ထုတစ်ခုကိုဖန်တီးဖို့ new command ကို နှိုးဆွတဲ့အခါ Java ဟာ instance variable တွေကိုစတင်ဖို့ constructor လို့ခေါ်တဲ့ အထူး method တစ်ခုကို ခေါ်ယူပါတယ်။ Class သတ်မှတ်ချက်ရဲ့ တစ်စိတ်တစ်ဒေသအဖြစ် constructor တချို့ကို ပံ့ပိုးပေးရပါတယ်။
- ပုံမှန်အားဖြင့် အမျိုးအစားတစ်ခုပေါ်မှာ လုပ်ဆောင်တဲ့ method အားလုံးဟာ အဲဒီအမျိုးအစားအတွက် class သတ်မှတ်ချက်ထဲမှာ ရှိနေပါတယ်။
  Class သတ်မှတ်ချက်တွေနဲ့ ပတ်သက်ပြီး သဒ္ဒါကိစ္စတွေလည်း ရှိနေပါတယ်။
- Class နာမည်တွေနဲ့ ဝတ္ထုအမျိုးအစားတွေဟာ အက္ခရာအကြီးနဲ့ အမြဲစတင်ပါတယ်။ ဒါဟာ သူတို့ကို အခြေခံအမျိုးအစားတွေနဲ့ variable နာမည်တွေကနေ ကွဲပြားစေပါတယ်။
- အများအားဖြင့် class အဓိပ္ပာယ်သတ်မှတ်ချက်တစ်ခုကို ဖိုင်တစ်ခုစီနဲ့ သိမ်းပါတယ်။ ဖိုင်ရဲ့နာမည်ဟာ class ရဲ့နာမည်နဲ့ အတူတူရှိရပြီး သူ့နောက်မှာ `.java` လိုက်ရပါတယ်။ ဥပမာ `Time` class ကို `Time.java` ဖိုင်ထဲမှာ သတ်မှတ်ရပါတယ်။
- ပရိုဂရမ်တိုင်းမှာ class တစ်ခုကို စတင်တဲ့ class အနေနဲ့ အလုပ်ပေးရပါတယ်။ စတင်တဲ့ class မှာ `main` ဆိုတဲ့နာမည် ပါရမယ်။ ဒါဟာ ပရိုဂရမ် စ run တဲ့နေရာ ဖြစ်ပါတယ်။
  တခြား class တွေမှာလည်း `main` လို့ နာမည်ပေးထားတဲ့ method တစ်ခု ပါဝင်နိုင်ပါတယ်။
  ဒါပေမယ့် သူ့ကို run မှာတော့ မဟုတ်ပါဘူး။

ဒီကိစ္စတွေနဲ့အတူ အသုံးပြုသူက သတ်မှတ်လိုက်တဲ့ `Time` ဆိုတဲ့ အမျိုးအစားကို ဥပမာတစ်ခုအနေနဲ့ ကြည့်ကြပါမယ်။

`Time`

ဝတ္ထုအမျိုးအစား အသစ်တစ်ခုကို ဖန်တီးဖြစ်စေတဲ့ ဆွဲဆောင်မှုတစ်ခုကတော့ data အပိုင်းအစတွေကိုယူပြီး argument အဖြစ် ပို့ပေးတာဖြစ်ဖြစ်၊ လုပ်ဆောင်ချက်တစ်ခုကို ဆောင်ရွက်တာဖြစ်ဖြစ် ကိုင်တွယ်မှုတွေပြုလုပ်နိုင်တဲ့ ဝတ္ထုတစ်ခုအနေနဲ့ အဲဒီ data တွေကို အုပ်ယူပြီး

တစ်ခုတည်းသောယူနစ်အဖြစ် စုစည်းယူတာ ဖြစ်ပါတယ်။ ဒီလိုပုံစံမျိုးကို Point နဲ့ Rectangle ဆိုတဲ့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ အမျိုးအစားနှစ်ခုမှာ တွေ့ခဲ့ပြီးပါပြီ။

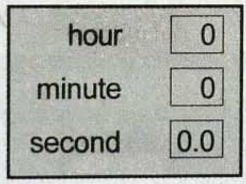
နောက်ထပ်ဥပမာတစ်ခုကတော့ Time ဖြစ်ပြီး သူ့ကို ကိုယ့်ဘာသာကိုယ် အကောင်အထည်ဖော်ပါမယ်။ အချိန်တစ်ခုကို ဖြစ်ပေါ်စေတဲ့ အချက်အလက်တွေကတော့ နာရီ၊ မိနစ်နဲ့ စက္ကန့်ဖြစ်ပါတယ်။ Time ဝတ္ထုတိုင်းဟာ ဒီ data တွေကို သယ်ဆောင်မှာဖြစ်တဲ့အတွက် သူတို့ကိုသိမ်းဖို့ instance variable တွေကို ဖန်တီးရပါမယ်။

ပထမဆုံးအဆင့်ကတော့ variable တစ်ခုစီဟာ ဘယ်လိုအမျိုးအစား ဖြစ်သင့်သလဲဆိုတာ ဆုံးဖြတ်ဖို့ဖြစ်ပါတယ်။ hour နဲ့ minute ဟာ ကိန်းပြည့်တွေ ဖြစ်သင့်တယ်ဆိုတာတော့ ရှင်းပါတယ်။ ဒါပေမယ့် ကိစ္စတွေကို ပိုပြီးစိတ်ဝင်စားစရာကောင်းအောင် second ကို double လုပ်ပါမယ်။ ဒါမှ စက္ကန့်တစ်ခုရဲ့ အစိတ်အပိုင်းတွေကို သိုလှောင်နိုင်ပါမယ်။

Instance variable တွေကို class သတ်မှတ်ချက်ရဲ့ အစ method သတ်မှတ်ချက်တိုင်းရဲ့ ပြင်ပမှာ ရေးသားရပါတယ်။

```
class Time {
    int hour, minute;
    double second;
}
```

သူ့ဘာသာသူတောင် ဒီ code အပိုင်းအစဟာ တရားဝင် class သတ်မှတ်ချက်တစ်ခုဖြစ်နေပါပြီ။ Time ဝတ္ထုတစ်ခုအတွက် အခြေအနေပုံဟာ အောက်မှာပြထားသလိုပုံမျိုး ရှိနေပါတယ်။




Instance variable တွေ ကြေညာပြီးနောက်မှာ ဆက်လုပ်ရမယ့်အဆင့်ကတော့ class အသစ်အတွက် constructor တစ်ခုကို သတ်မှတ်တာဖြစ်ပါတယ်။

## Constructor များ

Constructor တစ်ခုရဲ့ ပုံမှန်အခန်းကဏ္ဍဟာ instance variable တွေကို စတင်ဖို့ဖြစ်ပါတယ်။ Constructor တွေအတွက် သဒ္ဒါဟာ တခြား method တွေရဲ့သဒ္ဒါနဲ့ ဆင်တူပါတယ်။

- Constructor ရဲ့နာမည်ဟာ class ရဲ့နာမည်နဲ့ အတူတူဖြစ်ပါတယ်။
- Constructor တွေမှာ ပြန်တဲ့အမျိုးအစားနဲ့ ပြန်တဲ့တန်ဖိုး မရှိပါဘူး။
- static ဆိုတဲ့ keyword ကို ချန်လှပ်ထားခဲ့ပါတယ်။

Time class အတွက် ဥပမာကို အောက်မှာပေးထားပါတယ်။

```
public Time() {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
```

public နဲ့ Time ကြားက ပြန်တဲ့တန်ဖိုးကို မြင်ရမယ့်နေရာမှာ ဘာမှမရှိတာကို သတိပြုပါ။ ဒါဟာ အခုရေးတဲ့ method က constructor တစ်ခု ဖြစ်တယ်ဆိုတာ compiler ကို သိစေတဲ့နည်းပဲ ဖြစ်ပါတယ်။

ဒီ constructor ဟာ argument တစ်ခုကိုမှ လက်မခံပါဘူး။ ဒါကို လက်သည်းကွင်း အလွတ်တစ်စုံကြည့်ပြီး သိနိုင်ပါတယ်။ Constructor ရဲ့စာကြောင်းတစ်ကြောင်းစီဟာ instance variable တစ်ခုကို ပရိုဂရမ်ရေးသားသူက သတ်မှတ်တဲ့တန်ဖိုးတွေနဲ့ စတင်ပေးပါတယ်။ this ဆိုတဲ့နာမည်ဟာ ကိုယ်ဖန်တီးနေတဲ့ဝတ္ထုကို ဆိုလိုတဲ့ အထူး keyword တစ်ခု ဖြစ်ပါတယ်။ this ကို ဝတ္ထုနာမည်တွေသုံးတဲ့အတိုင်း သုံးနိုင်ပါတယ်။ ဥပမာ this ရဲ့ instance variable တွေကို ရေးနိုင်ဖတ်နိုင်ပါတယ်။ this ကို တခြား method တစ်ခုဆီ argument တစ်ခုအနေနဲ့ ပေးပို့နိုင်ပါတယ်။

ဒါပေမယ့် this ကို ကြေညာစရာမလိုသလို သူ့ကိုဖန်တီးဖို့ new command ကိုလည်း သုံးစရာမလိုပါဘူး။ တကယ်တမ်းကျတော့ သူ့မှာ နေရာချထားခွင့်တောင် မပေးပါဘူး။ this ကို ကွန်ပျူတာစနစ်က ဖန်တီးပါတယ်။ ကိုယ်လုပ်ဖို့လိုသမျှ အရာအားလုံးဟာ သူ့ရဲ့ instance variable တွေအတွက် စတင်မှုတွေ ပြုလုပ်ပေးဖို့ပါပဲ။

Constructor တွေရေးတဲ့အခါ ဖြစ်တတ်တဲ့အမှားတစ်ခုဟာ အဆုံးမှာ return ဖော်ပြချက်တစ်ခုကို ထည့်မိတာပဲဖြစ်ပါတယ်။ ဒီလိုလုပ်ချင်စိတ်ကို ချိုးနှိမ်ထားပါ။

## Constructor များအကြောင်းကို ပိုမိုလေ့လာခြင်း

တခြား method တွေမှာလိုပဲ constructor တွေကို overload လုပ်နိုင်ပါတယ်။ ဆိုလိုတာက ကွဲပြားတဲ့ parameter တွေကိုပိုင်ဆိုင်တဲ့ constructor အများကြီးကို ပံ့ပိုးပေးနိုင်ပါတယ်။ Constructor တွေရဲ့ parameter တွေကို new command ရဲ့ argument တွေနဲ့ တွဲပေးခြင်းအားဖြင့် ဘယ် constructor ကို နှိုးဆွရမလဲဆိုတာ Java က သိပါတယ်။

အပေါ်မှာပြထားသလို argument မယူတဲ့ constructor တစ်ခုကို တည်ဆောက်တာ အသုံးများပါတယ်။ ရှိသမျှ instance variable တွေနဲ့ တစ်ထပ်တည်းတူတဲ့ parameter စာရင်းတစ်ခုကို constructor တစ်ခုက လက်ခံတာမျိုးလည်း ရှိပါတယ်။ ဥပမာ

```
public Time(int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
```

```
    this.second = second;
}
```

Parameter တွေရဲ့ နာမည်တွေနဲ့ အမျိုးအစားတွေဟာ instance variable တွေရဲ့ နာမည်တွေ၊ အမျိုးအစားတွေနဲ့ အတူတူဖြစ်ပါတယ်။ Constructor လုပ်တဲ့အလုပ်ဟာ parameter တွေဆီက သတင်းအချက်အလက်တွေကို ကူးယူပြီး instance variable တွေမှာ နေရာချထားတာဖြစ်ပါတယ်။ Point နဲ့ Rectangle အတွက် စာရွက်စာတမ်းကို ပြန်ကြည့်ရင် class နှစ်ခုစလုံးဟာ ဒီလို constructor တွေကို ပံ့ပိုးမိတယ်ဆိုတာ တွေ့ရပါလိမ့်မယ်။ Constructor တွေကို overload လုပ်တာဟာ ဝတ္ထုကို အရင်ဖန်တီးပြီးမှ ကျန်တဲ့ကွက်လပ်တွေ လိုက်ဖြည့်တာ ဒါမှမဟုတ် ဝတ္ထုမဖန်တီးခင် အချက်အလက် စုဆောင်းတာဆိုတဲ့ အလုပ်နှစ်ခုကြား ရွေးချယ်ခွင့်ပေးပါတယ်။

အခုထက်ထိတော့ ဒါဟာ သိပ်စိတ်ဝင်စားစရာ ကောင်းတယ်လို့ မထင်ရသေးပါဘူး။ တကယ်တမ်းလည်း စိတ်ဝင်စားစရာ မကောင်းသေးပါဘူး။ Constructor တွေကိုရေးရတာဟာ စက်ရုပ်ဆန်ပြီး အင်မတန်ပျင်းစရာကောင်းတဲ့ အလုပ်တစ်ခုဖြစ်ပါတယ်။ နှစ်ခုလောက်ရေးပြီးသွားရင် အိပ်ချင်စိတ်တွေ ပေါက်လာပါတယ်။ Instance varaible တွေရဲ့စာရင်းကို မြင်ရင်တောင်မှ အိပ်ရာထဲပြေးအိပ်ချင်စိတ်တွေ ပေါက်လာပါတယ်။

#### ဝတ္ထုအသစ်တစ်ခုကို ဖန်တီးခြင်း

Constructor တွေဟာ ဝတ္ထုတွေနဲ့ အမြင်မှာတူပေမယ့် သူတို့ကို ဘယ်တော့မှ တိုက်ရိုက်နှိုးဆွလို့ မရပါဘူး။ အဲဒီအစား new command ကိုသုံးတဲ့အခါ Java စနစ်ဟာ ဝတ္ထုအသစ်အတွက် ကွက်လပ်ကို နေရာချထားပြီး instance variable တွေကိုစတင်ဖို့ constructor ကို နှိုးဆွပါတယ်။

အောက်မှာပြထားတဲ့ ပရိုဂရမ်ဟာ Time ဝတ္ထုတွေကိုဖန်တီးပြီး စတင်နိုင်တဲ့ နည်းနှစ်နည်းကို သရုပ်ဖော်ပါတယ်။

```
class Time {
    int hour, minute;
    double second;

    public Time() {
        this.hour = 0;
        this.minute = 0;
        this.second = 0.0;
    }

    public Time(int hour, int minute, double second) {
```

```
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }

  public static void main(String[] args) {

    // one way to create and initialise a Time object
    Time t1 = new Time();
    t1.hour = 11;
    t1.minute = 8;
    t1.second = 3.14159;
    System.out.println(t1);

    // another way to do the same thing
    Time t2 = new Time(11, 8, 3.14159);
    System.out.println(t2);
  }
}
```

လေ့ကျင့်ခန်းတစ်ခုအနေနဲ့ ဒီပရိုဂရမ် run တဲ့ လမ်းကြောင်းကို ရှာကြည့်ပါ။

main မှာ new command ကို ပထမဆုံးအကြိမ် နှိုးဆွတဲ့အခါ argument တစ်ခုမှ ပံ့ပိုးပေးမထားတဲ့အတွက် Java ဟာ ပထမဆုံး constructor ကို နှိုးဆွပါတယ်။ သူ့နောက်က စာကြောင်းတွေဟာ instance variable တစ်ခုစီမှာ တန်ဖိုးတွေ နေရာချထားပါတယ်။

new command ကို ဒုတိယအကြိမ် နှိုးဆွတဲ့အခါ ဒုတိယ constructor ရဲ့ parameter တွေနဲ့ကိုက်ညီအောင် argument တွေကို ပံ့ပိုးပေးပါတယ်။ Instance variable တွေကို ဒီလိုစတင်တဲ့နည်းဟာ ပိုမိုတိုတောင်းပြီး အလုပ်တွင်ကျယ်မှုကို အနည်းငယ်ပိုများစေပါတယ်။ ဒါပေမယ့် သူ့ကိုဖတ်ရတာ ပိုခက်စေပါတယ်။ ဘာလို့လဲဆိုတော့ ဘယ် instance variable တွေမှာ ဘယ်တန်ဖိုးတွေကို နေရာချထားသလဲဆိုတာ ကွဲကွဲပြားပြား မသိရလို့ပါပဲ။

### ဝတ္ထုတစ်ခုကို Print လုပ်ခြင်း

ဒီပရိုဂရမ်ရဲ့ output ဟာ အောက်မှာပြထားသလို ဖြစ်ပါတယ်။

```
Time@80cc7c0
```

```
Time@80cc807
```

Java က အသုံးပြုသူက သတ်မှတ်ထားတဲ့ ဝတ္ထုအမျိုးအစားတစ်ခုရဲ့တန်ဖိုးကို print လုပ်တဲ့အခါ အမျိုးအစားရဲ့နာမည်နဲ့ ဝတ္ထုတိုင်းအတွက် သီးသန့်ရှိနေတဲ့ အခြေ 16 code ကို print လုပ်ပါတယ်။ ဒီ code ဟာ သူ့ဘာသာသူနေရင် အဓိပ္ပာယ်မရှိပါဘူး။ ရှင်းရှင်းပြောရရင် စက်တစ်လုံးကနေ တစ်လုံးအပြောင်းမှာ ကွဲပြားနိုင်သလို တစ်ခါနဲ့တစ်ခါ run ရင်လည်း မတူပါဘူး။ ဒါပေမယ့် သူ့ကိုအမှားပြင်တဲ့အခါ လိုအပ်ပါလိမ့်မယ်။ ဝတ္ထုတစ်ခုစီကို နောက်ကြောင်းပြန်လိုက်ချင်ရင် ဒါဟာ အသုံးဝင်လာပါလိမ့်မယ်။

ပရိုဂရမ်မာတွေမဟုတ်တဲ့ သာမန်အသုံးပြုသူတွေအတွက် အဓိပ္ပာယ်ရှိစေမယ့်နည်းနဲ့ ဝတ္ထုတွေကို print လုပ်ချင်ရင် အဲဒီအလုပ်ကိုလုပ်မယ့် method တစ်ခုကို ရေးရပါတယ်။ ကျွန်တော်တို့ရဲ့ဥပမာမှာ `printTime` ကိုသုံးပါမယ်။

```
public static void printTime(Time t) {
    System.out.println(t.hour + ":" + t.minute + ":" + t.second);
}
```

ဒီ method ရဲ့ output ဟာ t1 ဒါမှမဟုတ် t2 ကို argument တစ်ခုအနေနဲ့ ပေးပို့လိုက်ရင် 11:8:3.14159 ကို ရပါလိမ့်မယ်။ ဒါကိုမြင်ရင် အချိန်လို့သိပေမယ့် စံပုံစံတော့မဝင်ပါဘူး။ ဥပမာ မိနစ်နဲ့ စက္ကန့်အရေအတွက်ဟာ 10 ထက်ငယ်နေရင် ဂဏန်းရှေ့မှာ 0 ထည့်ရပါမယ်။ ပြီးတော့ စက္ကန့်မှာ ဒသမအပိုင်းကို ဖြုတ်ချင်ပါတယ်။ နောက်တစ်မျိုးပြန်ပြောရရင် ကျွန်တော်တို့လိုချင်တဲ့ပုံစံက 11:08:03 လိုပုံစံ ဖြစ်ပါတယ်။

ဘာသာစကားတော်တော်များများမှာ ကိန်းသေပုံစံသွင်းတာကို ထိန်းချုပ်တဲ့နည်းတွေဟာ ရိုးရှင်းပါတယ်။ ဒါပေမယ့် Java မှာတော့ ဒီလိုမဟုတ်ပါဘူး။

Java ဟာ အချိန်နဲ့နေ့စွဲတွေကို ပုံစံသွင်းတဲ့ကိရိယာတွေကို အင်မတန်စွမ်းအားကြီးတဲ့ပုံစံနဲ့ ပံ့ပိုးပေးထားပါတယ်။ ပုံစံသွင်းထားတဲ့ input လက်ခံကို နားလည်ယူဆမှုအတွက်လည်း ကိရိယာတွေလည်း ပါဝင်ပါတယ်။ ကံမကောင်းတာက ဒီကိရိယာတွေကိုသုံးရတာ သိပ်ပြီးမလွယ်ကူလို့တဲ့အတွက် လွယ်ကူတဲ့ ပရိုဂရမ်းမင်းအခြေခံတွေကို လေ့လာတဲ့ဒီစာအုပ်မှာ ဒီအကြောင်းကို ချန်လုပ်ထားခဲ့ပါတယ်။ ပုံစံသွင်းနည်းကို Core Java စာအုပ်မှာ အကျယ်တဝင့် လေ့လာနိုင်ပါတယ်။

### ဝတ္ထုများပေါ်တွင် လုပ်ဆောင်ချက်များဆောင်ရွက်ခြင်း

အကောင်းဆုံးပုံစံသုံးပြီး အချိန်တွေကို print မလုပ်နိုင်ပေမယ့် `Time` တွေကို ကိုင်တွယ်တဲ့ method တွေကိုတော့ ရေးနိုင်သေးပါတယ်။ ရှေ့လာမယ့်အပိုင်းတွေမှာ ဝတ္ထုတွေအပေါ်မှာ ဆောင်ရွက်မယ့် method တွေအတွက် လုပ်ကိုင်နိုင်တဲ့ ရေးသားနည်းတွေကို ဖော်ပြသွားပါမယ်။ တချို့လုပ်ဆောင်ချက်တွေအတွက် လုပ်ကိုင်နည်းတွေထဲက ရွေးချယ်မှုတွေကို ပြုလုပ်ရပါတယ်။ ဒီတော့ သူတို့တစ်ခုချင်းစီအတွက် ကောင်းကျိုးဆိုးကျိုးတွေကို သိထားသင့်ပါတယ်။

**သန့်ရှင်းသော function**။ ။ ဝတ္ထုတွေ၊ အခြေခံတန်ဖိုးတွေကို argument တွေအဖြစ် လက်ခံပြီး ဝတ္ထုတွေကို ပြုပြင်မှုမရှိပါဘူး။ ပြန်တဲ့တန်ဖိုးဟာ အခြေခံတန်ဖိုးတစ်ခု ဖြစ်နိုင်သလို method ထဲမှာ ဖန်တီးလိုက်တဲ့ ဝတ္ထုအသစ်တစ်ခု ဒါမှမဟုတ် အခြေခံတန်ဖိုးတစ်ခုလည်း ဖြစ်နိုင်ပါတယ်။

**ပြုပြင်သော function**။ ။ ဝတ္ထုတွေကို argument တွေအနေနဲ့လက်ခံပြီး သူတို့အားလုံး ဒါမှမဟုတ် အချို့ကိုပြုပြင်ပါတယ်။ တစ်ခါတစ်ရံမှာ `void` ကို ပြန်ထုတ်ပေးပါတယ်။

**ဖြည့်ပေးသော method**။ ။ Argument တစ်ခုဟာ method က ဖြည့်ပေးတာကို ခံရတဲ့ ဝတ္ထုအလွတ်တစ်ခု ဖြစ်ပါတယ်။ နည်းပညာသဘောအရ ဒါဟာ ပြုပြင်ပေးတဲ့ function တစ်မျိုးပဲ ဖြစ်ပါတယ်။

## သန့်ရှင်းသော Function

Method တစ်ခုရဲ့ရလဒ်ဟာ argument တွေအပေါ်မှာပဲမူတည်ပြီး argument ကို ပြုပြင်တာနဲ့ တစ်ခုခုကို print လုပ်တာလို ဘေးထွက်သက်ရောက်မှုတွေမရှိရင် အဲဒီ method ကိုသန့်ရှင်းတဲ့ function တစ်ခုအဖြစ် ယူဆပါတယ်။ သန့်ရှင်းတဲ့ function တစ်ခုကို နှိုးဆွရာကရတဲ့ တစ်ခုတည်းသောရလဒ်ဟာ ပြန်တဲ့တန်ဖိုးဖြစ်ပါတယ်။

ဥပမာတစ်ခုကတော့ `Time` နှစ်ခုကိုနှိုင်းယှဉ်ပြီး ပထမ operand ဟာ ဒုတိယ operand ထက် ကြီးမကြီးကိုဖော်ပြတဲ့ `boolean` တစ်ခုကို ထုတ်ပေးတဲ့ `after` ဖြစ်ပါတယ်။

```
public static boolean after(Time time1, Time time2) {
    if (time1.hour > time2.hour)        return true;
    if (time1.hour < time2.hour)        return false;

    if (time1.minute > time2.minute)    return true;
    if (time1.minute < time2.minute)    return false;

    if (time1.second > time2.second)    return true;
    return false;
}
```

ဒုတိယဥပမာကတော့ အချိန်တွေရဲ့ ပေါင်းလဒ်တွေကိုရှာပေးတဲ့ `addTime` ဖြစ်ပါတယ်။ ဥပမာ လက်ရှိအချိန်ဟာ `9:14:30` ဖြစ်ပြီး ငါးသလောက်ပေါင်းတာ 3 နာရီ 35 မိနစ်ကြာတယ်ဆိုရင် `addTime` ကိုသုံးပြီး ဘယ်အချိန် ငါးသလောက်ရမယ်ဆိုတာ သိနိုင်ပါတယ်။

အောက်မှာပြထားတာကတော့ သိပ်မဟန်ပေမယ့် ဒီအလုပ်ကိုလုပ်ပေးတဲ့ method ရဲ့ အကြမ်းပုံစံဖြစ်ပါတယ်။

```
public static Time addTime(Time t1, Time t2) {
```

```
    Time sum = new Time();
    sum.hour = t1.hour + t2.hour;
    sum.minute = t1.minute + t2.minute;
    sum.second = t1.second + t2.second;
    return sum;
}
```

ဒီ method ဟာ Time ဝတ္ထုတစ်ခုကိုပြန်ပေးမယ့် constructor တစ်ခုတော့ မဟုတ်ပါဘူး။ ဒီလို method မျိုးရဲ့သဒ္ဒါကို constructor တစ်ခုရဲ့သဒ္ဒါနဲ့ ပြန်ပြီးနှိုင်းယှဉ်သင့်ပါတယ်။ ဘာလို့လဲဆိုတော့ အလွယ်တကူ မျက်စေ့လည်နိုင်လို့ပါပဲ။

အောက်မှာပြထားတာကတော့ ဒီလို method မျိုးကို သုံးရတဲ့နည်းပါပဲ။ currentTime မှာ လက်ရှိအခြေအနေပါဝင်ပြီး fishTime မှာ ငါးသလောက်ပေါင်းရတဲ့ ကြာချိန်ပါဝင်ရင် ငါးသလောက်ကျက်မယ့်အချိန်ကို တွက်ထုတ်နိုင်ပါတယ်။

```
Time currentTime = new Time(9, 14, 30.0);
Time fishTime = new Time(3, 35, 0.0);
Time doneTime = addTime(currentTime, fishTime);
printTime(doneTime);
```

ဒီပရိုဂရမ်ရဲ့ output ဟာ 12:49:30.0 ဖြစ်ပြီး အဖြေမှန်ထုတ်ပေးပါတယ်။ အခြားတစ်ဖက်မှာတော့ မှန်ကန်တဲ့ရလဒ်ကိုမရတဲ့ အခြေအနေတွေရှိနိုင်ပါတယ်။ ဘယ်လိုပြဿနာမျိုးပါလဲ။

အကြောင်းကတော့ ဒီ method ဟာ ပေါင်းလိုက်ရင် 60 ကျော်တဲ့ စက္ကန့်တွေ၊ မိနစ်တွေကို မကိုင်တွယ်နိုင်ပါဘူး။ ဒီနေရာမှာ ပိုနေတဲ့စက္ကန့်တွေကို မိနစ်နေရာမှာပေါင်းထည့်ပြီး ပိုနေတဲ့ မိနစ်တွေကို နာရီနေရာမှာ ပေါင်းထည့်ရပါတယ်။

အောက်မှာပြထားတာကတော့ ဒီ method ကို ပြန်ပြင်ထားတဲ့ပုံစံဖြစ်ပါတယ်။

```
public static Time addTime(Time t1, Time t2) {
    Time sum = new Time();
    sum.hour = t1.hour + t2.hour;
    sum.minute = t1.minute + t2.minute;
    sum.second = t1.second + t2.second;

    if (sum.second >= 60.0) {
            sum.second -= 60.0;
            sum.minute += 1;
```

```
    }

    if (sum.minute >= 60) {
            sum.minute -= 60;
            sum.hour += 1;
    }
    return sum;
}
```

အခုရေးလိုက်တဲ့ method ဟာ မှန်ကန်ပေမယ့် အရွယ်အစားကြီးမားလာပါတယ်။ နောက်ပိုင်းကျရင် အများကြီးပိုတိုတဲ့ ချဉ်းကပ်နည်းတွေကို အကြံပြုပါမယ်။

ဒီ code မှာ အရင်တုန်းက မမြင်ဖူးခဲ့တဲ့ += နဲ့ -= ဆိုတဲ့ လက္ခဏာနှစ်ခုကို သရုပ်ဖော်ထားတာကို တွေ့ရပါလိမ့်မယ်။ ဒီလက္ခဏာတွေဟာ variable တွေကို increment နဲ့ decrement တိတိကျကျလုပ်နိုင်တဲ့ နည်းလမ်းတစ်ခုကို ပေးပါတယ်။ သူတို့ဟာ ++ နဲ့ -- အသုံးနဲ့ ဆင်တူပါတယ်။ ခြားနားချက်တွေကတော့ (၁) သူတို့ဟာ `int` တွေသာမကဘဲ `double` တွေပေါ်မှာလည်း ဆောင်ရွက်နိုင်ပါတယ်၊ (၂) တိုးတဲ့လျော့တဲ့ပမာဏဟာ 1 ဖြစ်စရာ မလိုပါဘူး။ `sum.second = 60.0;` ဆိုတဲ့ ဖော်ပြချက်ဟာ `sum.second -= sum.second - 60;` နဲ့ တူညီပါတယ်။

## ပြုပြင်သော Function များ

ပြုပြင်တဲ့ function (modifier) တစ်ခုရဲ့ဥပမာအနေနဲ့ `Time` ဝတ္ထုတစ်ခုမှာ ပေးထားတဲ့ ကိန်းတစ်ခုကို စက္ကန့်အနေနဲ့ပေါင်းထည့်တဲ့ `increment` ကို စဉ်းစားကြည့်ပါ။ ဒီ method အကြမ်းပုံစံဟာ အောက်မှာပြထားသလို ဖြစ်ပါတယ်။

```
public static void increment(Time time, double secs) {
    time.second += secs;

    if (time.second >= 60.0) {
            time.second -= 60.0;
            time.minute += 1;
    }
    if (time.minute >= 60) {
            time.minute -= 60;
            time.hour += 1;
    }
```

ပထမစာကြောင်းဟာ အခြေခံလုပ်ဆောင်ချက်ကိုဆောင်ရွက်ပြီး ကျန်တဲ့အပိုင်းဟာ အရင်ကမြင်ခဲ့တဲ့ အခြေအနေမျိုးကို ကိုင်တွယ်ပါတယ်။

ဒီ method ဟာ မှန်ကန်ပါသလား။ secs ဆိုတဲ့ argument ဟာ 60 ထက်ကြီးရင် ဘာဖြစ်မလဲ။ ဒီအနေအထားမှာ 60 ကို တစ်ခါနုတ်ရုံနဲ့ မလုံလောက်ပါဘူး။ secs ဟာ 60 အောက်မရောက်မချင်း ဒါကို ထပ်ကာထပ်ကာ လုပ်ရပါမယ်။ ဒီလိုလုပ်ချင်ရင် if ဖော်ပြချက်တွေကို while ဖော်ပြချက်တွေနဲ့ အစားထိုးရုံနဲ့ ရပါတယ်။

```
public static void increment(Time time, double secs) {
    time.second += secs;

    while (time.second >= 60.0) {
            time.second -= 60.0;
            time.minute += 1;
    }
    while (time.minute >= 60) {
            time.minute -= 60;
            time.hour += 1;
    }
}
```

ဒီဖြေရှင်းနည်းဟာ မှန်ကန်ပါတယ်။ ဒါပေမယ့် သိပ်ပြီး အလုပ်မတွင်ကျယ်ပါဘူး။ သံသရာမလိုတဲ့ အဖြေတစ်ခုကို စဉ်းစားနိုင်မလား။ အဖြေကတော့ စက္ကန့်တွေနဲ့စဉ်းစားရင် ရပါလိမ့်မယ်။

## ဖြည့်ပေးသော Method များ

ရံဖန်ရံခါမှာ addTime လို method မျိုးတွေကို ကွဲပြားတဲ့ပုံစံတစ်ခုနဲ့ ရေးထားတာတွေရပါလိမ့်မယ်။ ကွဲပြားတဲ့ပုံစံဆိုတာ ကွဲပြားတဲ့ argument တွေနဲ့ ကွဲပြားတဲ့ ပြန်တဲ့တန်ဖိုးကို ဆိုလိုပါတယ်။ addTime ကို နှိုးဆွတဲ့အခါတိုင်း ဝတ္ထုအသစ်တစ်ခုကို ဖန်တီးမယ့်အစား ခေါ်ဆိုသူကို ဝတ္ထုအလွတ်တစ်ခု ပံ့ပိုးပေးခိုင်းပြီး addTime က ရလဒ်ကို အဲဒီဝတ္ထုမှာ သိမ်းဆည်းနိုင်ပါတယ်။ အောက်မှာပြထားတာကို ပြီးခဲ့တဲ့အပိုင်းကပုံစံနဲ့ နှိုင်းယှဉ်ကြည့်ပါ။

```
public static void addTimeFill(Time t1, Time t2, Time sum) {
    sum.hour = t1.hour + t2.hour;
    sum.minute = t1.minute + t2.minute;
    sum.second = t1.second + t2.second;
```

```
    if (sum.second >= 60.0) {
        sum.second -= 60.0;
        sum.minute += 1;
    }
    if (sum.minute >= 60) {
        sum.minute -= 60;
        sum.hour += 1;
    }
}
```

ဒီချဉ်းကပ်ပုံရဲ့ အားသာချက်တစ်ခုကတော့ ခေါ်ဆိုသူဟာ အပေါင်းလုပ်ဆောင်ချက် တစ်သီတစ်တန်းကြီးကို ဆောင်ရွက်ဖို့ ဝတ္ထုတစ်ခုတည်းကို အပြန်ပြန်အလှန်လှန် ပြန်သုံးနိုင်တဲ့ ရွေးချယ်ခွင့်ကို ရရှိပါတယ်။ ဒါဟာ အလုပ်တွင်ကျယ်မှုကို အနည်းငယ်ပိုမြင့်နိုင်စေပေမယ့် သိမ့်မွေ့တဲ့အမှားတွေ ပေါ်ပေါက်လောက်အောင် ရှုပ်ထွေးနိုင်စေပါတယ်။ ပရိုဂရမ်မင်းအလုပ်တော်တော်များများအတွက် run တဲ့အချိန်ကို နည်းနည်းပိုပေးပြီး အမှားပြင်ရတဲ့အချိန် နည်းအောင်လုပ်တာဟာ သင့်လျော်တဲ့ အပေးအယူကိစ္စတစ်ခု ဖြစ်ပါတယ်။

### ဘယ်ဟာအကောင်းဆုံးလဲ

ပြုပြင်တဲ့ function တွေနဲ့ ဖြည့်ပေးတဲ့ function တွေကိုသုံးပြီး လုပ်နိုင်တဲ့အလုပ်အားလုံးကို သန့်စင်တဲ့ function တွေသုံးပြီးလည်း လုပ်နိုင်ပါတယ်။ တကယ်တမ်းပြောရရင် လုပ်ဆောင်ချက်ဦးတည်တဲ့ ပရိုဂရမ်မင်းဘာသာစကားတွေလို့ခေါ်တဲ့ ဘာသာစကားတွေတောင်ရှိပြီး သူတို့ဟာ သန့်စင်တဲ့ function တွေကိုပဲ ခွင့်ပြုပါတယ်။ အချို့ပရိုဂရမ်မာတွေက သန့်စင်တဲ့ function တွေကို သုံးတဲ့ပရိုဂရမ်တွေဟာ ရေးရပိုလွယ်ပြီး အမှားဖြစ်နိုင်ခြေ နည်းစေတယ်လို့ ယုံကြည်ကြပါတယ်။ ဒါပေမယ့် ပြုပြင်တဲ့ function တွေသုံးရင် အဆင်ပြေစေမယ့်နေရာတွေ ရှိပါတယ်။ လုပ်ဆောင်ချက်ဦးတည်တဲ့ ပရိုဂရမ်တွေကိုသုံးရင် အလုပ်တွင်ကျယ်မှုနည်းတဲ့ အခြေအနေတွေလည်း ရှိပါတယ်။

ယေဘုယျအားဖြင့် ကျိုးကြောင်းဆီလျော်တဲ့အခါတိုင်း သန့်စင်တဲ့ function တွေကိုရေးပြီး မက်လောက်စရာ အကျိုးကျေးဇူးတစ်ခုရှိမှ ပြုပြင်တဲ့ function တွေကိုသုံးဖို့ အကြံပေးချင်ပါတယ်။ ဒီချဉ်းကပ်နည်းကို လုပ်ဆောင်ချက်ဦးတည်တဲ့ ပရိုဂရမ်ရေးဟန်လို့ ခေါ်နိုင်ပါတယ်။

### အဆင့်ဆင့်ရေးသားခြင်းနှင့် အစီအစဉ်ချခြင်း

ဒီအခန်းမှာ အကြိမ်ကြိမ်တိုးတက်နေသည့် လျင်မြန်သောစမ်းသပ်ခြင်းလို့ခေါ်တဲ့ ပရိုဂရမ်ရေးသားချဉ်းကပ်နည်းကို သရုပ်ဖော်ခဲ့ပါတယ်။ ဥပမာတစ်ခုစီမှာ စမ်းသပ်ချက်အတွက်သုံးတဲ့ အကြမ်းပုံစံတစ်ခုကို အခြေခံတွက်ချက်မှုဆောင်ရွက်ဖို့ ရေးသားခဲ့ပြီး တစ်ချိန်တည်းမှာ အမှား

ပြင်ဆင်မှုကို လုပ်ခဲ့ပါတယ်။

ဒီချဉ်းကပ်နည်းဟာ အကျိုးသက်ရောက်မှုရှိပေမယ့် မလိုအပ်ဘဲရှုပ်ထွေးတဲ့ code မျိုးကို ရေးမိစေပါတယ်။ အကြောင်းရင်းက ခြွင်းချက်အခြေအနေအများကြီးကို ကိုင်တွယ်ရလို့ပါပဲ။ အမှားတွေအားလုံးကို ရှာတွေ့ပြီးပြီလို့ ကိုယ့်ကိုယ်ကို အာမမခံနိုင်တဲ့အတွက်လည်း စိတ်မချရပါဘူး။

နောက်ထပ် ရွေးစရာတစ်လမ်းကတော့ အဆင့်မြင့်ကြိုတင်စီမံနည်းပဲ ဖြစ်ပါတယ်။ ဒီနည်းမှာ ပြဿနာကို အာရုံထည့်ပြီး တွေးတောမှုနည်းနည်းလုပ်လိုက်ရင် ပရိုဂရမ်းမင်းကို အများကြီး ပိုလွယ်စေပါတယ်။ ဒီနေရာမှာ ထည့်ရမယ့်အာရုံက Time တစ်ခုဟာ တကယ်တော့ အခြေ 60 ရှိတဲ့ ဂဏန်းသုံးလုံးပါကိန်းတစ်ခုပဲ ဖြစ်တယ်ဆိုတာပဲ ဖြစ်ပါတယ်။ second ဟာ 1 တွေရဲ့ ကော်လံဖြစ်ပြီး minute ဟာ 60 တွေရဲ့ ကော်လံဖြစ်ပါတယ်။ hour ကတော့ 3600 တွေရဲ့ ကော်လံဖြစ်ပါတယ်။

addTime နဲ့ increment ကိုရေးခဲ့တဲ့အခါမှာ အမှန်တကယ်လုပ်နေတာကတော့ အခြေ 60 နဲ့ ကိန်းတွေပေါင်းနေတာ ဖြစ်ပါတယ်။ အခြေ 10 စနစ်မှာ 10 ပြည့်သွားရင် သူ့ရှေ့က ကော်လံတိုင်ဆီကို 1 တင်ရသလိုပဲ စက္ကန့် 60 ပြည့်ရင် မိနစ်မှာ 1 တိုးရပါတယ်။

ဒီတော့ ချဉ်းကပ်နည်းနောက်တစ်မျိုးဟာ Time တွေကို double အဖြစ် ပြောင်းပစ်ပြီး ကွန်ပျူတာဟာ double တွေနဲ့ သင်္ချာဘယ်လိုတွက်ရမလဲဆိုတာ သိပြီးသားဆိုတဲ့ အခြေအနေကို အခွင့်ကောင်းယူဖို့ပါပဲ။ အောက်မှာပြထားတာကတော့ Time တစ်ခုကို double တစ်ခုအဖြစ် ပြောင်းပေးတဲ့ method ပါပဲ။

```
public static double convertToSeconds(Time t) {
    int minutes = t.hour * 60 + t.minute;
    double seconds = minutes * 60 + t.second;
    return seconds;
}
```

အခုအချိန်မှာ လိုအပ်သမျှကတော့ double တစ်ခုကနေ Time ဝတ္ထုတစ်ခုကို ပြောင်းပေးတဲ့ နည်းတစ်နည်းပါပဲ။ ဒီအလုပ်လုပ်ပေးတဲ့ method တစ်ခုကို ရေးလို့ရပါတယ်။ ဒါပေမယ့် သူ့ကို တတိယ constructor တစ်ခုအနေနဲ့ထားလိုက်ရင် ပိုပြီးအဓိပ္ပာယ်ရှိပါလိမ့်မယ်။

```
public Time(double secs) {
    this.hour = (int) (secs / 3600.0);
    secs -= this.hour * 3600.0;
    this.minute = (int) (secs / 60.0);
    secs -= this.minute * 60;
    this.second = secs;
```

```
}
```

ဒီ constructor ဟာ တခြား constructor တွေနဲ့ နည်းနည်းကွဲပါတယ်။ သူ့မှာ instance variable တွေကို နေရာချထားမှုလုပ်တာအပြင် တွက်ချက်မှုတွေလည်း လုပ်ရပါသေးတယ်။

အခြေတစ်ခုကနေ နောက်တစ်ခုကိုပြောင်းဖို့ အခုသုံးနေတဲ့နည်းစနစ်ဟာ မှန်ကန်တယ်လို့ ကိုယ့်ဘာသာကိုယ်ယုံအောင် တွေးတောမှုနည်းနည်း လုပ်ရပါတယ်။ ယုံသွားပြီဆိုရင် ဒီ method တွေကိုသုံးပြီး addTime ကို ပြန်ရေးနိုင်ပါတယ်။

```
public static Time addTime(Time t1, Time t2) {
    double seconds = convertToSeconds(t1) + convertToSeconds(t2);
    return new Time(seconds);
}
```

အခုလိုလုပ်လိုက်တာဟာ မူလပုံစံထက် အများကြီးပိုတိုပါတယ်။ မှန်ကြောင်းသက်သေပြရလည်း ပိုလွယ်ပါတယ်။ (ခေါ်ယူတဲ့ method တွေဟာ မှန်တယ်ဆိုရင်ပေါ့။) လေ့ကျင့်ခန်းအနေနဲ့ increment ကို ဒီလိုနည်းနဲ့ ပြန်ရေးကြည့်ပါ။

**ယေဘုယျပြုခြင်း**

တချို့အခြေအနေတွေမှာ အခြေ 60 နဲ့ အခြေ 10 ကို ဟိုပြောင်းဒီပြောင်းလုပ်ရတာ ပုံမှန်အချိန်တွက်တာထက် ပိုခက်တဲ့အခါ ရှိပါတယ်။ အခြေပြောင်းတာဟာ စိတ္တဇပိုဆန်ပါတယ်။ လူသားတွေရဲ့ပင်ကိုယ်အချိန်နားလည်နိုင်စွမ်းဟာ ပိုကောင်းတတ်ပါတယ်။

ဒါပေမယ့် အချိန်တွေကို အခြေ 60 ရှိတဲ့ ကိန်းတွေအဖြစ်မြင်တဲ့ အသိကိုရပြီး convertToSeconds နဲ့ တတိယ constructor လို ပြောင်းလဲပေးတဲ့ method တွေ ရေးတဲ့အလုပ်ကိုလုပ်ခဲ့ရင် ပိုပြီးတိုတဲ့၊ ဖတ်ရပိုလွယ်တဲ့၊ အမှားပြင်ရအဆင်ပြေတဲ့ ပရိုဂရမ်တစ်ပုဒ်ကို စိတ်ချရတဲ့အနေအထားမှာ ရရှိပါတယ်။

နောက်ပိုင်းမှာ လက္ခဏာတွေပေါင်းထည့်ရတာလည်း ပိုလွယ်လာပါတယ်။ ဥပမာ Time နှစ်ခုကိုနုတ်ပြီး သူတို့ကြားကကြာချိန်ကို ရှာမယ်ဆိုပါစို့။ ခပ်တုံးတုံးချဉ်းကပ်မှုတစ်ခုဟာ အနုတ်ကို ဟိုချေးလိုက် ဒီချေးလိုက်နဲ့ အကောင်အထည်ဖော်ပါလိမ့်မယ်။ အခုလိုပြောင်းလဲပေးတဲ့ method တွေကိုသုံးရတာ ပိုပြီးလွယ်ကူစေပါတယ်။

ရယ်စရာကောင်းတာက ပြဿနာတစ်ခုကို ယေဘုယျပိုကျအောင်လုပ်ပြီး ပိုခက်စေတာဟာ ခြွင်းချက်ပိုနည်းအောင်နဲ့ အမှားဖြစ်ခွင့်ပိုနည်းအောင် လုပ်ပေးတဲ့အတွက် ဖြေရှင်းရ ပိုလွယ်ကူစေပါတယ်။

## Algorithm များ

ပြဿနာတစ်ခုတည်းအတွက် သီးသန့်ဖြေရှင်းနည်းတစ်ခုမဟုတ်ဘဲ ပြဿနာအတန်းအစားတစ်ခုအတွက် ယေဘုယျဖြေရှင်းနည်းတစ်ခုကို ရေးတာကို algorithm တစ်ခုလို့ ခေါ်ပါတယ်။

ဒီအသုံးနှုန်းကို အဓိပ္ပာယ်ဖွင့်ဆိုရတာ မလွယ်လှပါဘူး။ ဒီတော့ ချဉ်းကပ်နည်းနှစ်မျိုးသုံးမျိုးကို သုံးကြည့်ကြပါမယ်။

ပထမဆုံးအနေနဲ့ algorithm တွေမဟုတ်တဲ့ ကိစ္စတွေကို စဉ်းစားကြည့်ရအောင်။ ဥပမာ အနေနဲ့ ဂဏန်းတစ်လုံးတည်းရှိတဲ့ ကိန်းအလီတွေကို လေ့လာခဲ့တုန်းက အလီဇယားကို အလွတ်ကျက်မှတ်ခဲ့ကြပါတယ်။ တကယ်တမ်းကျတော့ ဖြေရှင်းနည်း 100 ကို အလွတ်ကျက်ခဲ့ပါတယ်။ ဒီတော့ ဒီအသိပညာဟာ algorithm တစ်ခု မဟုတ်ပါဘူး။

ဉာဏ်ကောင်းတဲ့သူတွေဟာ သမာရိုးကျအလုပ်တွေ လုပ်တဲ့အခါ အပျင်းကြီးတတ်ကြပါတယ်။ ဥပမာ n နဲ့ 9 ရဲ့မြှောက်လဒ်ကိုရှာဖို့ ပထမဂဏန်းအတွက် n - 1 နဲ့ ဒုတိယဂဏန်းအတွက် 10 - n ကို ရှာပြီး ပေါင်းရေးနိုင်ပါတယ်။ ဒီနည်းဟာ 9 နဲ့ ဂဏန်းတစ်လုံးရှိတဲ့ ကိန်းတိုင်းအတွက် မြှောက်လဒ်တန်ဖိုးရှာဖို့ ယေဘုယျတွက်ချက်နည်း ဖြစ်ပါတယ်။ ဒါဟာ algorithm တစ်ခု ဖြစ်ပါတယ်။

အဲဒီလိုပဲ ကိန်းဂဏန်းတွေပေါင်းရာမှာ 10 ကျော်ရင် ရှေ့ဂဏန်းမှာ 1 တင်တာ၊ ကန်းတွေနုတ်ရင် မနုတ်လောက်လို့ 1 ချေးတာတွေဟာ algorithm တွေဖြစ်ပါတယ်။ Algorithm တွေရဲ့ ဝိသေသလက္ခဏာတစ်ခုက သူတို့ကိုလုပ်ဆောင်ဖို့ သိပ်ပြီး ဉာဏ်ရှိစရာမလိုပါဘူး။ သူတို့ဟာ စည်းမျဉ်းအစုအဝေးတစ်ခုကို လိုက်နာပြီး တစ်ဆင့်ပြီးတစ်ဆင့်ဆောင်ရွက်ရတဲ့ စက်ရုပ်ဆန်ဆန် ဖြစ်စဉ်တွေ ဖြစ်ပါတယ်။

တကယ်ဆိုရင် ဉာဏ်ရှိစရာလုံးဝမလိုဘဲ တွက်ချက်နိုင်တဲ့ algorithm တွေကိုလေ့လာဖို့ ကျောင်းတက်ခဲ့ရတဲ့ အချိန်တွေအများကြီးကို ပေးခဲ့ရတာဟာ ရှက်စရာကောင်းပါတယ်။ အခြားတစ်ဖက်မှာတော့ algorithm တွေကို ဒီဇိုင်းထုတ်ရတာဟာ စိတ်ဝင်စားစရာကောင်းပါတယ်။ ဉာဏ်ရှိရှိနဲ့ လုပ်ရတဲ့အလုပ်ဖြစ်ပါတယ်။ ပရိုဂရမ်းမင်းလို့ခေါ်တဲ့အရာရဲ့ အဓိကအပိုင်းဖြစ်ပါတယ်။

မတွေးတောဘဲ အခက်အခဲမရှိလုပ်ကိုင်ကြတဲ့ အရာတချို့ကို algorithm ပုံစံနဲ့ ဖော်ပြရတာဟာ အခက်ဆုံးအလုပ် ဖြစ်ပါတယ်။ သဘာဝဘာသာစကားကို နားလည်အောင်လုပ်ကြရတဲ့ ဖြစ်စဉ်ကို ဥပမာပေးရင်ရပါတယ်။ ဘယ်သူမဆို လူရယ်လို့ဖြစ်လာရင် ဘာသာစကားတစ်ခုတော့ ပြောတတ်ကြပါတယ်။ ဒါပေမယ့် ဘယ်လိုပြောတတ်သွားကြသလဲ ဆိုတာကိုတော့ ရှင်းပြလို့မရပါဘူး။ အနည်းဆုံးတော့ algorithm ပုံစံတစ်ခုအနေနဲ့တော့ ရှင်းပြလို့ မရနိုင်ပါဘူး။

နောက်ပိုင်းကျရင် ပြဿနာမျိုးစုံအတွက် ရိုးရှင်းတဲ့ algorithm ရေးနည်းကို လေ့လာရပါမယ်။

အခန်း ၁၀

# Array များ

Array တွေဟာ တန်ဖိုးတစ်ခုစီကို index တစ်ခုနဲ့ သတ်မှတ်တဲ့ တန်ဖိုးအစုအဝေးတစ်ခု ဖြစ်ပါတယ်။ int တွေ၊ double နဲ့ တခြားအမျိုးအစားတွေအတွက် array တစ်ခုကို ပြုလုပ်နိုင်ပါတယ်။ ဒါပေမယ့် array တစ်ခုထဲက တန်ဖိုးတွေအားလုံးဟာ တူညီတဲ့ အမျိုးအစားတစ်ခုကို ပိုင်ဆိုင်ကြပါမယ်။

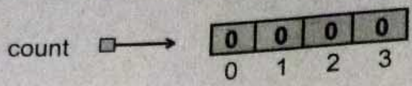
သဒ္ဒါသဘောအရ array အမျိုးအစားတွေဟာ သူတို့နောက်က [] လိုက်တာကလွဲရင် တခြား Java အမျိုးအစားတွေနဲ့ အတူတူဖြစ်ပါတယ်။ ဥပမာ int[] ဟာ ကိန်းပြည့်တွေရဲ့ array အမျိုးအစားဖြစ်ပြီး double[] ဟာ double တွေရဲ့ array အမျိုးအစားဖြစ်ပါတယ်။ ဒီလိုအမျိုးအစားတွေရှိတဲ့ variable တွေကို ပုံမှန်နည်းလမ်းတွေသုံးပြီး ကြေညာနိုင်ပါတယ်။

```
int[] count;
double[] values;
```

ဒီ variable တွေကို စတင်မှုမပြုမချင်း သူတို့ကို null မှာထားပါတယ်။ Array ကိုယ်တိုင်ကိုဖန်တီးဖို့ new command ကို အသုံးပြုရပါတယ်။

```
count = new int[4];
values = new double[size];
```

ပထမနေရာချထားမှုဟာ count ကို ကိန်းပြည့်လေးခုပါတဲ့ array တစ်ခုဆီ ညွှန်းနိုင်းလိုက်ပါတယ်။ ဒုတိယနေရာချထားမှုဟာ values ကို double တွေရဲ့ array တစ်ခုဆီ ညွှန်းဆိုနိုင်းပါတယ်။ values မှာရှိတဲ့ အဖွဲ့ဝင်အရေအတွက်ဟာ size ပေါ်မှာ မူတည်ပါတယ်။ Array အရွယ်အစားတစ်ခုအဖြစ် ကြိုက်တဲ့ကိန်းပြည့်ဖော်ပြချက်ကို အသုံးပြုနိုင်ပါတယ်။ အောက်မှာပြထားတဲ့ပုံဟာ array တွေကို အခြေအနေပုံတွေမှာ ဘယ်လိုကိုယ်စားပြုရမလဲဆိုတာ ပြသပါတယ်။



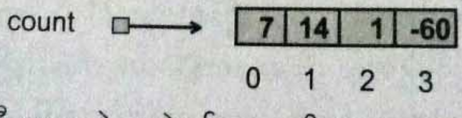

လေးထောင့်ကွက်တွေထဲက စာလုံးထူနဲ့ ရေးထားတဲ့ကိန်းတွေဟာ array ရဲ့ အဖွဲ့ဝင်တွေဖြစ်ပါတယ်။ လေးထောင့်ကွက်တွေအပြင်က ရိုးရိုးစာလုံးနဲ့ ရေးထားတဲ့ကိန်းတွေဟာ လေးထောင့်ကွက်တစ်ကွက်ကို ကိုယ်စားပြုတဲ့ index တွေဖြစ်ပါတယ်။ Array အသစ်တစ်ခုကို နေရာချထားရင် အဖွဲ့ဝင်တွေကို သုညနဲ့ စတင်လိုက်ပါတယ်။

**အဖွဲ့ဝင်များဆီသို့ရောက်ရှိခြင်း**

Array ထဲမှာ တန်ဖိုးတွေကိုသိုလှောင်ဖို့ [] လက္ခဏာကိုသုံးပါ။ ဥပမာ count[0] ဟာ array ရဲ့ သုညခုမြောက်အဖွဲ့ဝင်ကို ညွှန်ပြပြီး count[1] ဟာ တစ်ခုမြောက်အဖွဲ့ဝင်ကို ညွှန်ပြပါတယ်။ [] လက္ခဏာကို ဖော်ပြချက်တစ်ခုရဲ့ ဘယ်နေရာမှာမဆို သုံးနိုင်ပါတယ်။

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

အပေါ်မှာပြထားတာအားလုံးဟာ တရားဝင်နေရာချထားတဲ့ ဖော်ပြချက်တွေဖြစ်ပါတယ်။ ဒီ code အပိုင်းအစရဲ့အကျိုးသက်ရောက်မှုကို အောက်မှာပြထားပါတယ်။




အခုချိန်မှာ ဒီ array ရဲ့ အဖွဲ့ဝင်လေးခုကို 0 ကနေ 3 အထိ နံပါတ်တပ်ထားတယ်ဆိုတာ သိနေလောက်ပါပြီ။ ဆိုလိုတာက index 4 ရှိတဲ့ အဖွဲ့ဝင်မရှိဘူးဆိုတာပါပဲ။ ဒီလိုစကားမျိုးနဲ့ ရင်းနှီးနေသင့်ပါတယ်။ ဘာလို့လဲဆိုတော့ String index တွေမှာတုန်းက သူနဲ့ထပ်တူထပ်မျှတူတဲ့ ကိစ္စမျိုးကို မြင်ခဲ့ရလို့ပါပဲ။ ဒါတောင်မှ array တစ်ခုရဲ့ အကန့်အသတ်ပြင်ပကို ကျော်လွန်ပြီး အမှားလုပ်တတ်ကြတာတွေ ရှိပါတယ်။ ဒါဆိုရင် ArrayOutOfBoundsException ကို ဖြစ်ပေါ်စေပါလိမ့်မယ်။ ခြွင်းချက်တိုင်းမှာလိုပဲ အမှား message တစ်ခုကိုရပြီး ပရိုဂရမ်ဟာ ထွက်သွားပါလိမ့်မယ်။

ဘယ်ဖော်ပြချက်ကိုမဆို int ဖြစ်နေသရွေ့ index တစ်ခုအနေနဲ့ သုံးနိုင်ပါတယ်။ Array တစ်ခုကို index လုပ်နိုင်တဲ့ အသုံးအများဆုံး နည်းတစ်ခုကတော့ သံသရာထဲက variable တစ်ခုဖြစ်ပါတယ်။ ဥပမာ

```
int i = 0;
while (i < 4) {
    System.out.println(count[i]);
    i++;
}
```

ဒါဟာ 0 ကနေ 4 မတိုင်ခင်အထိ ရေတွက်တဲ့ while သံသရာတစ်ခု ဖြစ်ပါတယ်။ သံသရာ variable i ဟာ 4 ဖြစ်သွားရင် အခြေအနေမပြေလည်တော့ဘဲ သံသရာ ပြီးဆုံးသွားပါတယ်။ ဒီတော့ သံသရာရဲ့ကိုယ်ထည်ကို i ဟာ 0, 1, 2 နဲ့ 3 ရှိနေတုန်းမှာပဲ run ပါတယ်။

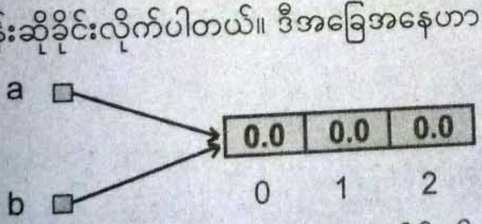
သံသရာကို တစ်ကြိမ်ပတ်တိုင်း i ကို index အနေနဲ့ array ထဲ ထည့်ပေးပါတယ်။ i ခု မြောက်အဖွဲ့ဝင်ကို print လုပ်ပါတယ်။ Array တွေရဲ့ သံသရာတွေဟာ လက်ဖက်ရည်နဲ့ အီကြာကွေးလို လိုက်ဖက်မှုရှိပါတယ်။

## Array များကိုကူးယူခြင်း

Array variable တစ်ခုကိုကူးယူတဲ့အခါ array ဆီသွားတဲ့ အညွှန်းကို ကူးယူနေတယ်ဆိုတာကို အမှတ်ရပါ။ ဥပမာ

```
double[] a = new double[3];
double[] b = a;
```

ဒီ code ဟာ double သုံးခုပါတဲ့ array တစ်ခုကိုတည်ဆောက်ပြီး ကွဲပြားတဲ့ variable နှစ်ခုကို သူ့ဆီ ညွှန်းဆိုခိုင်းလိုက်ပါတယ်။ ဒီအခြေအနေဟာ နာမည်နှစ်ခုပေးတဲ့ ပုံစံတစ်မျိုးပါပဲ။




Array တစ်ခုဆီမှာ ပြုလုပ်တဲ့အပြောင်းအလဲတိုင်းကို နောက် array တစ်ခုမှာ မှန်ပေါ်ကပုံရိပ်လို မြင်ရပါလိမ့်မယ်။ ဒါဟာ ပရိုဂရမ်မာတွေလိုချင်တဲ့အပြုအမူမျိုးမဟုတ်ပါဘူး။ ဒီလိုလုပ်မယ့်အစား array အသစ်တစ်ခုကို နေရာချထားပြီး အဖွဲ့ဝင်တိုင်းကို တစ်နေရာကနေ တစ်နေရာကူးယူရင် array ရဲ့ မိတ္တူတစ်ခုကို ရရှိပါလိမ့်မယ်။

```
double[] b = new double[3];
```

```
        b[i] = a[i];
        i++;
    }
```

**for သံသရာများ**

အခုထက်ထိရေးခဲ့သမျှ သံသရာတွေမှာ တူညီတဲ့အကြောင်းအရာတွေ ရှိပါတယ်။ သူတို့အားလုံးဟာ variable တစ်ခုကိုစတင်ပြီး ဘဝစပါတယ်။ သူတို့မှာ အဲဒီ variable ပေါ်မူတည်တဲ့ အခြေအနေ ဒါမှမဟုတ် စမ်းသပ်ချက်တစ်ခုရှိပါတယ်။ သံသရာထဲမှာ အဲဒီ variable ကို တစ်ခုခုလုပ်ပါတယ်။ ဥပမာ ဂဏန်းတစ်ခု တိုးပါတယ်။

ဒီလိုသံသရာမျိုးကို အသုံးများလွန်းလို့ သူ့လိုမျိုး သံသရာဖော်ပြချက်တစ်ခု ရှိပါတယ်။ for လို့ခေါ်ပြီး ကျစ်လစ်မှုပိုရှိပါတယ်။ ယေဘုယျသဒ္ဒါဟာ အောက်မှာပြထားသလို ဖြစ်ပါတယ်။

```
for (INITIAILISER; CONDITION; INCREMENTOR) {
    BODY
}
```

ဒီဖော်ပြချက်ဟာ အောက်မှာပြထားတဲ့ ဖော်ပြချက်နဲ့ တသဝေမတိမ်းတူပါတယ်။

```
INITIALISER;
while (CONDITION) {
    BODY
    INCREMENTOR
}
```

ပိုပြီးကျစ်လစ်မှုရှိတာကလွဲရင် အောက်ကပုံစံဟာ ပထမပုံစံနဲ့ အတူတူဖြစ်ပါတယ်။ သံသရာနဲ့ပတ်သက်တဲ့ ဖော်ပြချက်တွေအားလုံးကို တစ်နေရာထဲမှာ ထားခြင်းအားဖြင့်လည်း ဖတ်ရလွယ်စေပါတယ်။ ဥပမာ

```
for (int i = 0; i < 4; i++) {
    System.out.println(count[i]);
}
```

ဆိုတာဟာ

```
int i = 0;
while (i < 4) {
    System.out.println(count[i]);
    i++;
}
```

လေ့ကျင့်ခန်းတစ်ခုအနေနဲ့ array ရဲ့အဖွဲ့ဝင်တွေကို ကူးယူတဲ့ for သံသရာတစ်ခု

ရေးကြည့်ပါလား။

## Array များနှင့် ဝတ္ထုများ

Array တွေဟာ ရှုထောင့်အမျိုးမျိုးမှာ ဝတ္ထုတွေလို ပြုမူကြပါတယ်။

- Array variable တစ်ခုကိုကြေညာရင် array တစ်ခုဆီသွားတဲ့ အညွှန်းတစ်ခုကို ရပါတယ်။
- Array ကိုယ်တိုင်ကို ဖန်တီးဖို့ new command ကို သုံးရပါတယ်။
- Array တစ်ခုကို argument တစ်ခုအနေနဲ့ ပေးပို့တဲ့အခါ အညွှန်းတစ်ခုကိုပေးပို့ပြီး နှိုးဆွခံရတဲ့ method ဟာ array ရဲ့ ပါဝင်မှုတွေကို ပြုပြင်နိုင်တယ်လို့ ဆိုလိုပါတယ်။

မြင်တွေ့ခဲ့ရတဲ့ ဝတ္ထုတချို့ဟာ (ဥပမာ Rectangle တွေ) array တွေနဲ့ ဆင်တူပါတယ်။ သူတို့ဟာ တန်ဖိုးတွေကိုစုစည်းထားတဲ့ နာမည်တစ်ခုဖြစ်တယ်ဆိုတဲ့ အမြင်နဲ့ကြည့်ရင် ဆင်တူတယ်လို့ ဆိုနိုင်ပါတယ်။ ဒါဟာ ကိန်းပြည့်လေးလုံးပါတဲ့ array တစ်ခုနဲ့ ထောင့်မှန်စတုဂံ ဝတ္ထုတစ်ခု ဘာကွာခြားလဲဆိုတဲ့ မေးခွန်းကို ပေါ်ပေါက်စေပါတယ်။

ဒီအခန်းရဲ့အစက array ရဲ့ အဓိပ္ပာယ်သတ်မှတ်ချက်ကို ပြန်ကြည့်မယ်ဆိုရင် ကွာခြားချက်တစ်ခုကို တွေ့ရပါလိမ့်မယ်။ Array တစ်ခုရဲ့ အဖွဲ့ဝင်တွေကို index တွေနဲ့ ကိုယ်စားပြုပေမယ့် ဝတ္ထုတစ်ခုရဲ့ အဖွဲ့ဝင် (instance variable) တွေကတော့ x တို့၊ width တို့လို နာမည်တွေပဲ ရှိပါတယ်။

Array တွေနဲ့ ဝတ္ထုတွေကြားက ကွာခြားချက် နောက်တစ်ခုကတော့ array တစ်ခုရဲ့ အဖွဲ့ဝင်အားလုံးမှာ အမျိုးအစားအတူတူ ရှိရပါတယ်။ ဒါဟာ Rectangle တွေအတွက်လည်း မှန်ပေမယ့် Time လို အမျိုးအစားအမျိုးမျိုးရှိတဲ့ instance variable တွေကိုပိုင်ဆိုင်တဲ့ တခြားဝတ္ထုတွေကိုလည်း မြင်ခဲ့ရပါတယ်။

## Array အလျား

တကယ်တမ်းကျတော့ array တွေမှာ နာမည်ပေးထားတဲ့ instance variable တစ်ခုရှိပါတယ်။ သူကတော့ length ဖြစ်ပါတယ်။ သူ့မှာ array ရဲ့ အလျားပါဝင်ပါတယ်။ အလျားဆိုတာ အဖွဲ့ဝင်အရေအတွက် ဖြစ်ပါတယ်။ ကိန်းသေတန်ဖိုးတစ်ခုကို သံသရာရဲ့ အကြီးဆုံးအကန့်အသတ်အဖြစ် သုံးမယ့်အစား ဒီတန်ဖိုးကို အသုံးပြုသင့်ပါတယ်။ ဒီလုပ်ခြင်းအားဖြင့် array ရဲ့ အရွယ်အစားပြောင်းသွားရင် သံသရာတွေကိုလိုက်ပြောင်းဖို့ ပရိုဂရမ်တစ်ပုဒ်လုံးကို လိုက်စစ်စရာမလိုပါဘူး။ ဘယ်အရွယ်အစားရှိတဲ့ array မဆိုအတွက် သူတို့ဟာ မှန်မှန်ကန်ကန် အလုပ်ဖြစ်နေပါလိမ့်မယ်။

```
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

သံသရာရဲ့ကိုယ်ထည်ကို နောက်ဆုံး run တဲ့အခါ i ရဲ့တန်ဖိုးဟာ a.length - 1 ရှိနေပါတယ်။ သူဟာ နောက်ဆုံးအဖွဲ့ဝင်ရဲ့ index ဖြစ်ပါတယ်။ i ဟာ a.length နဲ့တူတဲ့အခါ အခြေအနေပျက်သွားပြီး ကိုယ်ထည်ကို မ run တော့ပါဘူး။ ဒါဟာ ကောင်းတဲ့ကိစ္စတစ်ခု ဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတော့ ဒီလိုမဟုတ်ရင် ခြွင်းချက်တစ်ခုကို ဖြစ်ပေါ်စေမှာ ဖြစ်လို့ပါပဲ။ ဒီ code ဟာ array b မှာ array a ရဲ့ အဖွဲ့ဝင်အရေအတွက်အတိုင်း အဖွဲ့ဝင်အနည်းဆုံးရှိတယ်လို့ ယူဆပါတယ်။

**ကျပန်းကိန်းများ**

ကွန်ပျူတာပရိုဂရမ် တော်တော်များများဟာ သူတို့ကို run တဲ့အခါတိုင်း တူညီတဲ့ရလဒ်တစ်ခုကို ထုတ်ပေးပါတယ်။ အဲဒီအတွက် ကွန်ပျူတာပရိုဂရမ်တွေကို မှန်းဆလို့ရနိုင်တယ်လို့ ပြောကြပါတယ်။ ပုံမှန်အားဖြင့် မှန်းဆနိုင်စွမ်းရည်ဟာ ကောင်းတဲ့အလုပ်တစ်ခု ဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတော့ တူညီတဲ့တွက်ချက်မှုတွေဟာ တူညီတဲ့ရလဒ်တွေကို ထုတ်ပေးမယ်လို့ မျှော်လင့်ကြလို့ပါပဲ။ တချို့အသုံးချမှုတွေအတွက်တော့ ကွန်ပျူတာကို မှန်းဆလို့မရစေချင်တဲ့ အခြေအနေမျိုးတွေ ရှိပါတယ်။ ဂိမ်းတွေဟာ မြင်သာတဲ့ဥပမာတစ်ခု ဖြစ်ပါတယ်။ ဒါပေမယ့် ဒီလိုဖြစ်စေလိုတဲ့ ကိစ္စမျိုးတွေ အများကြီးရှိနေပါသေးတယ်။

ပရိုဂရမ်တစ်ပုဒ်ကို အပြည့်အဝ မှန်းဆလို့မရအောင်လုပ်တာဟာ တော်တော်မလွယ်တဲ့ ကိစ္စတစ်ခု ဖြစ်ပါတယ်။ ဒါပေမယ့် အနည်းဆုံး မှန်းဆလို့မရဘူးလို့ထင်ရအောင် လုပ်နိုင်တဲ့နည်းတွေရှိပါတယ်။ သူတို့ထဲကတစ်ခုကတော့ ကျပန်းကိန်းတွေကိုထုတ်လုပ်ပြီး ပရိုဂရမ်ရဲ့ရလဒ်ကို ဆုံးဖြတ်ခိုင်းဖို့ပါပဲ။ Java ဟာ ကျဘမ်းကိန်းအတုတွေကို ထုတ်ပေးတဲ့ method တစ်ခုကို ထည့်သွင်းတည်ဆောက်ပေးထားပါတယ်။ သူတို့ဟာ သင်္ချာသဘောနဲ့ကြည့်ရင် သိပ်ပြီးကျပန်းမဖြစ်ပါဘူး။ ဒါပေမယ့် လက်ရှိအသုံးအတွက်တော့ အဆင်ပြေစေပါလိမ့်မယ်။

Math class ထဲက ramdom method ရဲ့ စာရွက်စာတမ်းကိုဖတ်ကြည့်ရင် ပြန်တဲ့တန်ဖိုးဟာ 0.0 နဲ့ 1.0 ကြားက double တစ်ခုဖြစ်ပါတယ်။ ပိုပြီးတိတိကျကျပြောရရင် သူဟာ 0.0 နဲ့ညီရင်ညီ မညီရင် ကြီးပေမယ့် 1.0 တော့ ဖြစ်လို့မရပါဘူး။ ramdom ကို နှိုးဆွတဲ့အခါတိုင်း ကျပန်းကိန်းတုအစီအစဉ်ထဲက နောက်လာတဲ့ကိန်းကို ရပါတယ်။ နမူနာကြည့်ဖို့ အောက်က သံသရာကို run ကြည့်ပါ။

```
for (int i = 0; i < 10; i++) {
    double x = Math.random();
    System.out.println(x);
}
```

0.0 နဲ့ high လို အထက်ပိုင်း အကန့်အသတ်ကြားက ကျပန်း double တစ်ခုကို ထုတ်လုပ်ချင်ရင် x ကို high နဲ့ မြှောက်နိုင်ပါတယ်။ ဥပမာ မျက်နှာပြင်ခြောက်ဖက်ပါတဲ့ အံစာတုံးတစ်

ခုအတွက် ပရိုဂရမ်ရေးချင်ရင် x ကို 6 နဲ့ မြှောက်နိုင်ပါတယ်။ အောက်ပိုင်းအကန့်အသတ်ဖြစ်တဲ့ low နဲ့ အထက်ပိုင်းအကန့်အသတ်ဖြစ်တဲ့ high ကြားက ကျပန်းကိန်းတစ်ခုကို ထုတ်လုပ်ချင်ရင် ဘယ်လိုလုပ်မလဲ။ ကျပန်းကိန်းပြည့်တစ်ခုကို ဘယ်လိုထုတ်မလဲ။ အဖြေသိချင်ရင် Core Java နဲ့ Core JavaScript စာအုပ်မှာ ရှာကြည့်နိုင်ပါတယ်။

## ကျပန်းကိန်းများ၏ Array

```
public static int randomInt(int low, int high) {
    double x = Math.random();
    double prob = x * (high - low) + low;
    return (int) prob;
}
```

randomInt ရဲ့အကောင်အထည်ဖော်ပုံဟာ မှန်ကန်တယ်ဆိုရင် low ကနေ high အထိကြားမှာရှိတဲ့ တန်ဖိုးအားလုံးဟာ ဖြစ်နိုင်ခြေအတူတူ ရှိရပါမယ်။ ရှည်လျားတဲ့ ကိန်းတန်းကြီးတစ်ခုကို ထုတ်လုပ်တဲ့အခါ တန်ဖိုးတိုင်းပေါ်ပေါက်သင့်ပြီး အနီးစပ်ဆုံးပေါ်တဲ့ အကြိမ်တွေလည်း တူရပါမယ်။

အဲဒီ method မှန်ကြောင်း စမ်းသပ်နည်းကတော့ ကျပန်းတန်ဖိုးအရေအတွက် အများကြီးကိုထုတ်လုပ်ပြီး သူတို့ကို array တစ်ခုထဲမှာ သိုလှောင်ရပါမယ်။ ပြီးရင် တန်ဖိုးတစ်ခုစီပေါ်ပေါက်တဲ့ အရေအတွက်ကို ရေတွက်ဖို့ပါပဲ။

အောက်မှာပြထားတဲ့ method ဟာ array ရဲ့အရွယ်အစားကို argument အနေနဲ့ လက်ခံပါတယ်။ သူက ကိန်းပြည့်တွေရဲ့ array အသစ်တစ်ခုကို နေရာချထားပါတယ်။ အဲဒီ array ကို ကျပန်းတန်ဖိုးတွေနဲ့ ဖြည့်ပါတယ်။ Array အသစ်ဆီသွားတဲ့ အညွှန်းတစ်ခုကို ပြန်ထုတ်ပေးပါတယ်။

```
public static int[] randomArray(int n) {
    int[] a = new int[n];
    for (int i = 0; i < a.length; i++) {
        a[i] = randomInt(0, 100);
    }
    return a;
}
```

ပြန်တဲ့အမျိုးအစားဟာ int[] ဖြစ်ပြီး ကိန်းပြည့်တွေရဲ့ array တစ်ခုကို ပြန်ထုတ်ပေးတယ်လို့ ဆိုလိုပါတယ်။ ဒီ method ကို စမ်းသပ်ဖို့ array တစ်ခုရဲ့ ပါဝင်မှုတွေကို print လုပ်ပေးတဲ့ method တစ်ခုကိုရေးရင် အဆင်ပြေပါလိမ့်မယ်။

```
public static void printArray(int[] a) {
    for (int i = 0; i < a.length; i++) {
        System.out.println(a[i]);
    }
}
```

အောက်မှာပြထားတဲ့ code ဟာ array တစ်ခုကို ထုတ်လုပ်ပြီး print လုပ်ပါတယ်။

```
int numValues = 8;
int[] array = randomArray(numValues);
printArray(array);
```

တစ်ကြိမ်စမ်းကြည့်တာ အောက်မှာပြထားတဲ့ output ထွက်လာပါတယ်။

```
27
6
54
62
54
2
44
81
```

ရလဒ်တွေဟာ ကျပန်းပုံစံရှိပါတယ်။ တစ်ခါနဲ့တစ်ခါ run ရင် ရလဒ်တွေ ကွဲပြားနိုင်ပါတယ်။

ဒီနံပါတ်တွေဟာ စာမေးပွဲဖြေလို့ရတဲ့ အမှတ်တွေဆိုရင် တော်တော်ဆိုးရွားတဲ့ အမှတ်တွေဖြစ်ပါလိမ့်မယ်။ သူတို့ဆရာက ဒီရလဒ်တွေကို histogram လို့ခေါ်တဲ့ တန်ဖိုးတစ်ခု ဘယ်နှခါဖြစ်ပေါ်သလဲဆိုတာ နောက်ကြောင်းလိုက်တဲ့ ရေတွက်စာရင်းတစ်ခုကို ပြုစုပါလိမ့်မယ်။

စာမေးပွဲရလဒ်တွေအတွက် ကျောင်းသားဘယ်နှယောက် 90 ကျော်ရလဲ၊ 80 ကျော်ရလဲ စသဖြင့် ရေတွက်တဲ့ variable ဆယ်ခု ရှိနိုင်ပါတယ်။ နောက်လာမယ့်အပိုင်းက histogram တစ်ခုကို ထုတ်လုပ်နိုင်တဲ့ code ကို ရေးသားပါတယ်။

## ရေတွက်ခြင်း

ဒီလိုပြဿနာမျိုးကို ချဉ်းကပ်နိုင်တဲ့ နည်းလမ်းကောင်းတစ်ခုကတော့ ရေးရလွယ်တဲ့ method ရိုးရိုးရှင်းရှင်းလေးတွေကို တွေးကြည့်ဖို့ပါပဲ။ ဒီလိုလုပ်တာဟာ အသုံးဝင်တယ်လို့ တွေ့လာပါလိမ့်မယ်။ ပြီးရင် သူတို့အားလုံးကို ဖြေရှင်းနည်းတစ်ခုမှာ ပေါင်းစည်းနိုင်ပါတယ်။ ဘယ် method တွေဟာ အသုံးဝင်လိမ့်မလဲလို့ ကြိုတင်တွက်ဆရတာဟာ သိပ်ပြီးလွယ်တဲ့အလုပ်တော့ မဟုတ်ပါဘူး။ ဒါပေမယ့် အတွေ့အကြုံရှိလာတာနဲ့အမျှ အတွေးအခေါ်တွေ ပိုထွက်လ

ပါလိမ့်မယ်။

ပြီးတော့ ဘယ်လိုအရာမျိုးတွေဟာ ရေးရလွယ်လိမ့်မလဲဆိုတာ အမြဲတမ်း ထင်ထင်ရှားရှား မသိရပါဘူး။ ဒါပေမယ့် ကောင်းမွန်တဲ့ ချဉ်းကပ်နည်းတစ်ခုကတော့ အရင်ကမြင်ဖူးထားတဲ့ ပုံစံနဲ့ကိုက်ညီတဲ့ လက်အောက်ခံပြဿနာတွေကို ရှာဖို့ပါပဲ။

ပြီးခဲ့တဲ့ အပိုင်းတစ်ပိုင်းမှာတုန်းက စာသားတစ်ခုကို ဖြတ်လျှောက်ပြီး ပေးထားတဲ့အက္ခရာတစ်ခုဘယ်နှခါပါသလဲဆိုတာ ရေတွက်တဲ့ သံသရာတစ်ခုကို ကြည့်ရှုခဲ့ပါတယ်။ ဒီပရိုဂရမ်ကို "ဖြတ်လျှောက်ပြီး ရေတွက်လို့" ခေါ်တဲ့ ပုံစံတစ်ခုအနေနဲ့ မြင်နိုင်ပါတယ်။ ဒီပုံစံရဲ့ ပါဝင်မှုတွေကတော့

- Array တစ်ခုနဲ့ စာသားတစ်ခုလို ဖြတ်လျှောက်လို့ရတဲ့ အစုအဝေး ဒါမှမဟုတ် သယ်ဆောင်သူတစ်ခု
- သယ်ဆောင်သူရဲ့အဖွဲ့ဝင်တိုင်းအပေါ် ဆောင်ရွက်နိုင်မယ့် စစ်ဆေးမှုတစ်ခု
- စစ်ဆေးမှုကိုအောင်မြင်တဲ့ အဖွဲ့ဝင်အရေအတွက်ကို နောက်ကြောင်းလိုက်နိုင်တဲ့ ရေတွက် counter တစ်ခု

တို့ဖြစ်ကြပါတယ်။

အခုကိစ္စမှာတော့ သယ်ဆောင်သူဟာ ကိန်းပြည့်တွေရဲ့ array တစ်ခုဖြစ်ပါတယ်။ စစ်ဆေးမှုကတော့ တန်ဖိုးအပိုင်းအခြားတစ်ခုထဲမှာ ပေးထားတဲ့အမှတ် ရောက်နေမနေဖြစ်ပါတယ်။

အောက်မှာပေးထားတာကတော့ ပေးထားတဲ့ အပိုင်းအခြားတစ်ခုထဲမှာကျရောက်တဲ့ array တစ်ခုထဲက အဖွဲ့ဝင်အရေအတွက်ကို ရေတွက်နိုင်တဲ့ inRange method ဖြစ်ပါတယ်။ Parameter တွေကတော့ အသုံးပြုမယ့် array နဲ့ အပိုင်းအခြားရဲ့ အထက်အောက် အကန့်အသတ်တွေကို သတ်မှတ်ပေးတဲ့ ကိန်းပြည့်နှစ်ခု ဖြစ်ကြပါတယ်။

```
public static int inRange(int[] a, int low, int high) {
    int count = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] >= low && a[i] < high) count++;
    }
    return count;
}
```

ဒီ method ကိုရှင်းပြခဲ့ရာမှာ high နဲ့ low တို့နဲ့တူတဲ့ အမှတ်တစ်ခုဟာ အပိုင်းအခြားထဲမှာ ပါဝင်မှုရှိမရှိ မရှင်းပြခဲ့ပါဘူး။ ဒါပေမယ့် ဒီ code မှာမြင်ရတဲ့အတိုင်း low ပါဝင်ပြီး high ကတော့ မပါဝင်ပါဘူး။ ဒီလိုလုပ်ခြင်းအားဖြင့် အဖွဲ့ဝင်နှစ်ခုကို နှစ်ခါရေမိတဲ့ပြဿနာကနေ ဝေးသွားပါလိမ့်မယ်။

အခုအချိန်မှာတော့ ကိုယ်စိတ်ဝင်စားတဲ့ အပိုင်းအခြားထဲက အမှတ်အရေအတွက်ကို ရေ

တွက်နိုင်ပါပြီ။

```
int[] scores = randomArray(30);
int a = inRange(scores, 90, 100);
int b = inRange(scores, 80, 90);
int c = inRange(scores, 70, 80);
int d = inRange(scores, 60, 70);
int f = inRange(scores, 0, 60);
```

## Histogram

အခုရရှိခဲ့တဲ့ code ဟာ ဒီအလုပ်တစ်ခုတည်းကိုပဲ ထပ်ကာထပ်ကာ လုပ်နေရသလို ဖြစ်နေပါတယ်။ ဒါပေမယ့် အပိုင်းအခြားအရေအတွက် နည်းနေသရွေ့တော့ ဒါကို လက်ခံနိုင်ပါတယ်။ ဒါပေမယ့် အမှတ်တစ်ခုစီပေါ် ပေါက်တဲ့ အရေအတွက်ကို လိုက်ချင်တယ်ဆိုပြီး စဉ်းစားကြည့်ပါ။ ဖြစ်နိုင်တဲ့တန်ဖိုး 100 ရှိပါတယ်။ အောက်မှာပြထားသလို ရေးကြမလား။

```
int count0 = inRange(scores 0, 1);
int count1 = inRange(scores 1, 2);
int count2 = inRange(scores 2, 3);
...
int count99 = inRange(scores 99, 100);
```

ဒီလောက် မိုက်လုံးကြီးကြလိမ့်မယ်လို့တော့ မထင်မိပါဘူး။ တကယ်လိုအပ်တာက ကိန်းပြည့်အရေအတွက် 100 ကို သိမ်းဆည်းနိုင်တဲ့ နည်းတစ်နည်းပါ။ ကိန်းပြည့်တစ်ခုစီဆီ ရောက်ရှိဖို့ index တစ်ခုကိုသုံးနိုင်ရင် ပိုကောင်းပါတယ်။ ချက်ချင်းဆိုသလိုပဲ array ကို မြေးမြင်သင့်ပါတယ်။

ရေတွက်တဲ့ပုံစံဟာ ရေတွက် counter တစ်ခုပဲသုံးသုံး၊ counter တွေရဲ့ array တစ်ခုကိုပဲသုံးသုံး အတူတူပဲဖြစ်ပါတယ်။ ဒီအခြေအနေမှာ array ကို သံသရာရဲ့ပြင်ပမှာ စတင်ပါမယ်။ ပြီးမှ သံသရာထဲမှာ inRange ကိုနှိုးဆွပြီး ရလဒ်ကို သိမ်းဆည်းပါမယ်။

```
int[] counts = new int[100];

for (int i = 0; i < 100; i++) {
    counts[i] = inRange(scores, i, i + 1);
}
```

ဒီနေရာမှာ ဂရုစိုက်ရမယ့်အရာကတော့ သံသရာ variable ကို အခန်းကဏ္ဍနှစ်ခုအနေနဲ့ အသုံးပြုနေတာပဲ ဖြစ်ပါတယ်။ အခန်းကဏ္ဍတစ်ခုက array ရဲ့ index အနေနဲ့ဖြစ်ပြီး နောက်တစ်

ခုကတော့ inRange ရဲ့ parameter အနေနဲ့ပဲ ဖြစ်ပါတယ်။

**တစ်ချိန်တည်းနှင့် အလုပ်ပြီးသောဖြေရှင်းနည်း**

အပေါ်မှာပြထားတဲ့ code ဟာ အလုပ်ဖြစ်ပေမယ့် အလုပ်တွင်သင့်သလောက် တွင်ကျယ်မှုမရှိပါဘူး။ inRange ကို နှိုးဆွလိုက်တဲ့အခါတိုင်း array တစ်ခုလုံးကို ဖြတ်လျှောက်ရပါတယ်။ အပိုင်းအခြား အရေအတွက်များလာတာနဲ့အမျှ ဖြတ်လျှောက်ရတဲ့ အကြိမ်တွေလည်း မထိန်းနိုင် မသိမ်းနိုင် များလာပါတယ်။

Array ကို တစ်ကြိမ်တည်းဖြတ်လျှောက်ရတဲ့ နည်းတစ်ခုရှိရင် ပိုကောင်းပါလိမ့်မယ်။ အဲဒီနေရာမှာလည်း တန်ဖိုးတိုင်းအတွက် သူကျရောက်မယ့် အပိုင်းအခြားကို တွက်ထုတ်ပေးသင့်ပါတယ်။ ပြီးမှ သင့်တော်ရာရေတွက် counter ကို increment လုပ်နိုင်ပါတယ်။ အခုဥပမာမှာ တန်ဖိုးကိုယ်၌ကို counter ရဲ့ array အတွက် index အဖြစ်သုံးထားလို့ တွက်ချက်မှုဟာ နည်းနည်းလေးပဲ ရှိပါတယ်။

```
int[] counts = new int[100];

for (int i = 0; i < scores.length; i++) {
   int index = scores[i];
   counts[index]++;
}
```

**အခန်း ၁၁**

# ဝတ္ထုများ၏ Array များ

**ဖွဲ့စည်းခြင်း**

အခုအချိန်ရောက်ရင်တော့ ဘာသာစကားလက္ခဏာတွေကို အစီအစဉ်အမျိုးမျိုးနဲ့ ပေါင်းစပ်နိုင်စွမ်းဆိုတဲ့ ဖွဲ့စည်းနိုင်စွမ်းရည်ကို ဥပမာအမျိုးမျိုးမှာ မြင်ခဲ့ပြီးပါပြီ။ ပထမဆုံးမြင်ခဲ့တဲ့ဥပမာတစ်ခုကတော့ ဖော်ပြချက်တစ်ခုရဲ့တစ်စိတ်တစ်ဒေသအဖြစ် method နှိုးဆွမှုကို အသုံးပြုတာဖြစ်ပါတယ်။ နောက်ဥပမာကတော့ ဖော်ပြချက်တွေကို အဆင့်ဆင့်တည်ဆောက်တဲ့ပုံစံနဲ့ ရေးသားတာ ဖြစ်ပါတယ်။ if ဖော်ပြချက်တစ်ခုကို while သံသရာတစ်ခုထဲမှာ ထည့်နိုင်သလို နောက်ထပ် if ဖော်ပြချက်တစ်ခုထဲမှာလည်း ထည့်နိုင်ပါတယ်။

ဒီလိုပုံစံကိုမြင်ပြီးနောက်မှာ array တွေနဲ့ ဝတ္ထုတွေအကြောင်း လေ့လာပြီးတဲ့အခါ ဝတ္ထုတွေရဲ့ array တွေကို ပိုင်ဆိုင်နိုင်တယ်ဆိုတာ တအံ့တဩ ဖြစ်မနေသင့်တော့ပါဘူး။ တကယ်ပြောရရင် instance variable တွေအနေနဲ့ array တွေပါဝင်တဲ့ ဝတ္ထုတွေလည်း ရှိနိုင်ပါတယ်။ Array တွေပါဝင်တဲ့ array တွေလည်း ရှိနိုင်ပါတယ်။ ဝတ္ထုတွေပါဝင်တဲ့ ဝတ္ထုတွေလည်း ရှိနိုင်ပါတယ်။ စသဖြင့် အမျိုးမျိုးရှိနိုင်ပါတယ်။

လာမယ့်အခန်းတစ်ခန်းမှာ Card ဝတ္ထုတွေကို ဥပမာအနေနဲ့ကြည့်ပြီး ဒီပေါင်းစည်းမှုတွေကို လေ့လာကြပါမယ်။

Card ဝတ္ထုများ

ပုံမှန်ဖဲချပ်တွေနဲ့ မရင်းနှီးဘူးဆိုရင် အခုအချိန်ဟာ ဖဲတစ်ထုပ်ဆောင်ထားဖို့ အချိန်ကောင်းပါပဲ။ မဟုတ်ရင် ဒီအခန်းဟာ အဓိပ္ပာယ်သိပ်ရှိမှာ မဟုတ်ပါဘူး။ ဖဲတစ်ထုပ်မှာ ဖဲချပ် 52 ချပ်ပါပါတယ်။ သူတို့ဟာ အပွင့်တူတဲ့အုပ်စုလေးစုရဲ့ အဖွဲ့ဝင်တွေဖြစ်ကြပါတယ်။ အဆင့် 13 ဆင့်ထဲက တစ်ခုခုဖြစ်ပါလိမ့်မယ်။ အပွင့်တွေကတော့ Spade, Heart, Diamond နဲ့ Club ဆိုပြီး ရှိကြပါတယ်။ ဒါဟာ Bridge ကစားနည်းမှာ ကြီးစဉ်ငယ်လိုက် အစီအစဉ်ဖြစ်ပါတယ်။ ဖဲသမားမဟုတ်တဲ့အတွက် သူတို့ကို မြန်မာလို ဘယ်လိုခေါ်သလဲဆိုတာကိုတော့ မသိပါဘူး။ အဆင့်တွေကတော့ Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King ဆိုပြီး ဖြစ်ပါတယ်။ ကစားနည်းပေါ် မူတည်ပြီး Ace ရဲ့အဆင့်ဟာ King ထက်မြင့်နိုင်သလို 2 ထက်လည်း နိမ့်နိုင်ပါတယ်။

ဖဲချပ်တစ်ချပ်ကိုကိုယ်စားပြုတဲ့ ဝတ္ထုအသစ်တစ်ခုကို သတ်မှတ်တယ်ဆိုရင် ရှိသင့်တဲ့ instance variable က အဆင့်ကိုကိုယ်စားပြုတဲ့ `rank` နဲ့ အပွင့်အမျိုးအစားကို ကိုယ်စားပြုတဲ့ `suit` တို့ဖြစ်ပါတယ်။ ဒီ instance variable တွေရဲ့အမျိုးအစားဟာ ဘာဖြစ်သင့်သလဲဆိုတာတော့ ဒီလောက် လွယ်လွယ်ကူကူမသိနိုင်ပါဘူး။ ဖြစ်နိုင်ခြေတစ်ခုကတော့ အပွင့်တွေအတွက် `"Spade"` နဲ့ အဆင့်တွေအတွက် `"Queen"` လိုဟာမျိုးတွေပါဝင်တဲ့ `String` တွေ ဖြစ်ပါတယ်။ ဒီအကောင်အထည်ဖော်ပုံရဲ့ ပြဿနာတစ်ခုကတော့ ဘယ်ဖဲချပ်ဟာ အဆင့်ပိုမြင့်သလဲ၊ အပွင့်ပိုမြင့်သလဲဆိုတာ နှိုင်းယှဉ်ဖို့ခက်ခဲတာပဲ ဖြစ်ပါတယ်။

နောက်ရွေးချယ်နိုင်တဲ့ လမ်းတစ်ခုကတော့ အဆင့်တွေနဲ့အပွင့်တွေကို ကိန်းပြည့်တွေသုံးပြီး code ပုံစံနဲ့ ရေးဖို့ပါပဲ။ Code ပုံစံနဲ့ရေးတယ်ဆိုတာ တချို့လူတွေထင်သလို လျှို့ဝှက် code နံပါတ်တစ်ခုအဖြစ် ပြောင်းရေးတာမဟုတ်ပါဘူး။ ကွန်ပျူတာသမားတွေပြောကြတဲ့ code ပြောင်းရေးတယ်ဆိုတာ ဖော်ပြလိုတဲ့အရာတွေနဲ့ ကိန်းစဉ်တွေကြားမှာ ဆက်သွယ်ချက်တစ်ခုကို သတ်မှတ်ပေးတာဖြစ်ပါတယ်။ ဥပမာ -

| | | |
|---|---|---|
| Spades | $\mapsto$ | 3 |
| Hearts | $\mapsto$ | 2 |
| Diamonds | $\mapsto$ | 1 |
| Clubs | $\mapsto$ | 0 |

ဒီဆက်သွယ်ချက်ရဲ့ မြင်သာတဲ့လက္ခဏာကတော့ အပွင့်အမျိုးအစားတွေဟာ ကြီးစဉ်ငယ်လိုက် ကိန်းပြည့်တွေအဖြစ် ပြောင်းလဲသွားပါတယ်။ ဒီတော့ ကိန်းပြည့်တွေကိုနှိုင်းယှဉ်ရင် အပွင့်တွေကိုလည်း နှိုင်းယှဉ်နိုင်ပါတယ်။ အဆင့်တွေအတွက် ကိုယ်စားပြုပုံကတော့ မြင်သာပါတယ်။ နံပါတ်ရှိတဲ့ အဆင့်တွေဟာ သူနဲ့ဆိုင်တဲ့ကိန်းပြည့်အဖြစ် ပြောင်းသွားပါတယ်။ မျက်နှာပုံပါတဲ့ကတ်ပြားတွေအတွက်တော့ အောက်မှာပြထားတဲ့ပုံအတိုင်း ပြောင်းသွားပါတယ်။

| | | |
|---|---|---|
| Jack | $\mapsto$ | 11 |
| Queen | $\mapsto$ | 12 |

King ↦ 13

ဒီဆက်သွယ်ချက်တွေအတွက် သင်္ချာသင်္ကေတတွေ သုံးရခြင်းအကြောင်းရင်းက သူတို့ဟာ Java ပရိုဂရမ်ရဲ့ တစ်စိတ်တစ်ဒေသ မဟုတ်လို့ပါပဲ။ သူတို့ဟာ ပရိုဂရမ်ဒီဇိုင်းမှာပါဝင်ပေမယ့် code ထဲမှာ ပွင့်ပွင့်လင်းလင်းမပါဝင်ပါဘူး။ Card အမျိုးအစားအတွက် class အဓိပ္ပာယ်သတ်မှတ်ချက်ဟာ အောက်မှာပြထားသလို ရှိပါတယ်။

```
class Card {
    int suit, rank;

    public Card() {
        this.suit = 0;
        this.rank = 0;
    }

    public Card(int suit, int rank) {
        this.suit = suit;
        this.rank = rank;
    }
}
```

အရင်လိုပဲ constructor နှစ်ခုကို ပံ့ပိုးပေးနေတာဖြစ်ပါတယ်။ Constructor တစ်ခုဟာ instance variable နှစ်ခုစလုံးအတွက် parameter တွေကိုလက်ခံပြီး ကျန်တဲ့တစ်ခုကတော့ parameter လက်မခံပါဘူး။ Club ရဲ့ 3 ကို ကိုယ်စားပြုတဲ့ ဝတ္ထုတစ်ခုကို ဖန်တီးချင်ရင် new command ကိုသုံးနိုင်ပါတယ်။

```
Card threeOfClubs = new Card(0, 3);
```

0 ဆိုတဲ့ ပထမဆုံးတွေ့ရတဲ့ argument ဟာ Club အပွင့်ကို ကိုယ်စားပြုပါတယ်။

## printCard method

Class အသစ်တစ်ခုကို ဖန်တီးတဲ့အခါ ပုံမှန်အားဖြင့် ပထမဦးဆုံးလုပ်ရမယ့်အဆင့်ကတော့ instance variable တွေကြေညာတာနဲ့ constructor တွေရေးတာပဲ ဖြစ်ပါတယ်။ ဒုတိယအဆင့်က ဝတ္ထုတိုင်းပိုင်ဆိုင်သင့်တဲ့ method တွေကို ရေးသားတာဖြစ်ပါတယ်။ ဒီလို method တွေထဲမှာ ဝတ္ထုကို print လုပ်တဲ့ method တစ်ခုနဲ့ ဝတ္ထုတွေကိုနှိုင်းယှဉ်တဲ့ method တစ်ခုနှစ်ခုပါဝင်ပါတယ်။ printCard method နဲ့စလိုက်ရအောင်။

Card ဝတ္ထုတွေကို လူတွေအလွယ်တကူဖတ်နိုင်အောင် print လုပ်ဖို့အတွက် ကိန်းပြည့်

code တွေကို စကားလုံးတွေအဖြစ် ပြောင်းပေးတဲ့နည်းကို လိုအပ်ပါတယ်။ ဒါကို သဘာဝကျကျ လုပ်နိုင်တဲ့နည်းကတော့ String တွေရဲ့ array တစ်ခုပဲ ဖြစ်ပါတယ်။ String တွေရဲ့ array တစ်ခုကို အခြေခံအမျိုးအစားတွေရဲ့ array တွေကြေညာသလိုပဲ ဖန်တီးနိုင်ပါတယ်။

```
String[] suits = new String[4];
```

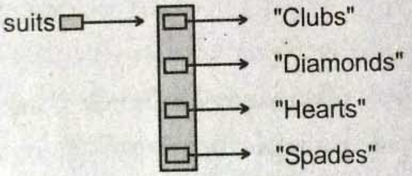
အဲဒီနောက်မှာ array ရဲ့ အဖွဲ့ဝင်တန်ဖိုးတွေကို ပေးနိုင်ပါတယ်။

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

Array တစ်ခုကို ဖန်တီးတာနဲ့ အဖွဲ့ဝင်တွေကို စတင်တာဟာ အသုံးများတဲ့လုပ်ဆောင်ချက်တွေဖြစ်တဲ့အတွက် Java ဟာ ဒီလုပ်ဆောင်ချက် နှစ်ခုစလုံးကိုပေါင်းပြီး အထူးသဒ္ဒါတစ်ခုကို ပံ့ပိုးပေးထားပါတယ်။

```
String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
```

ဒီဖော်ပြချက်ရဲ့ သက်ရောက်မှုကတော့ ကြေညာတာ၊ အရွယ်အစားသတ်မှတ်တာနဲ့ နေရာချထားတာကို ပေါင်းပြီးလုပ်သလိုပါပဲ။ ဒီ array ရဲ့အခြေအနေပုံဟာ အောက်မှာပြထားသလို ရှိပါလိမ့်မယ်။




Array ရဲ့အဖွဲ့ဝင်တွေဟာ String ဆီသွားတဲ့ အညွှန်းတွေဖြစ်ပါတယ်။ အညွှန်းတွေကိုယ်တိုင်မဟုတ်ပါဘူး။ ဒါဟာ ဝတ္ထုတွေရဲ့ array တိုင်းအတွက် မှန်ပါတယ်။ နောက်ပိုင်းကျမှ ဒီအကြောင်းကို အသေးစိတ် ဆက်ဆွေးနွေးပါမယ်။ အခုတော့ လိုအပ်နေတာက အဆင့်တွေကို code ပြန်ဖြည့်ဖို့ နောက်ထပ် String array တစ်ခုပဲ ဖြစ်ပါတယ်။

```
String[] ranks = {"narf", "Ace", "1", "2", "3", "4", "5", "6",
                  "7", "8", "9", "10", "Jack", "Queen", "King"};
```

"narf" ကိုရေးရခြင်းအကြောင်းက array ရဲ့သုညခုမြောက် အဖွဲ့ဝင်နေရာကို ယူဖို့ပဲဖြစ်ပါတယ်။ အဲဒီနေရာကို ဘယ်တော့မှ သုံးမှာမဟုတ်ပါဘူး။ တရားဝင်အဆင့်တွေက 1-13 ပဲဖြစ်ပါတယ်။ ဒီလိုဖြုန်းတီးလိုက်တဲ့ အဖွဲ့ဝင်ကို အလဟဿမဖြစ်အောင် လုပ်ခဲ့ရင်လည်းရပါတယ်။ အရင်တုန်းကလို 0 ကနေ စတင်ခဲ့နိုင်ပါတယ်။ ဒါပေမယ့် 2 ကို 2 လို့၊ 3 ကို 3 လို့ code လုပ်တာ အကောင်းဆုံးပါပဲ။

ဒီ array တွေကိုသုံးပြီး suit နဲ့ rank ကို index တွေအနေနဲ့ ထည့်ပေးခြင်းအားဖြင့်

သင့်တော်တဲ့ String တွေကို ရွေးချယ်နိုင်ပါတယ်။ printCard method ကို အောက်မှာရေးထားပါတယ်။

```
public static void printCard(Card c) {
    String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
    String[] ranks = {"narf", "Ace", "1", "2", "3", "4", "5", "6",
                      "7", "8", "9", "10", "Jack", "Queen", "King"};

    System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
}
```

suits[c.suit] ရဲ့အဓိပ္ပာယ်က c ဝတ္ထုထဲက suits ဆိုတဲ့ instance variable ကို suits လို့ နာမည်ပေးထားတဲ့ array ထဲမှာ index အနေနဲ့သုံးပြီး သင့်လျော်တဲ့စာသားကို ရွေးထုတ်ပါလို့ ဖြစ်ပါတယ်။ အောက်မှာပြထားတဲ့ code

```
Card card = new Card(1, 11);
printCard(card);
```

ရဲ့ output ကတော့

Jack of Diamonds ဆိုပြီးဖြစ်ပါတယ်။

## sameCard Method

တူတယ်ဆိုတဲ့ဝေါဟာရဟာ သဘာဝဘာသာစကားမှာ နည်းနည်းတွေးကြည့်လို့ ကိုယ်မျှော်လင့်ထားတာထက် အဓိပ္ပာယ်ပိုနက်နဲတယ်လို့ သတိမပြုမိမချင်း သေသေချာချာကြီးကို ကွဲကွဲပြားပြားမသိတဲ့ အရာတစ်ခုပဲ ဖြစ်ပါတယ်။

ဥပမာ "ကိုလတ်ရဲ့ကားနဲ့ ကျွန်တော့်ရဲ့ကား အတူတူပဲ" လို့ပြောရင် အဓိပ္ပာယ်က သူ့ကားနဲ့ကျွန်တော့်ကား အမျိုးအစားတူပြီး မော်ဒယ်လည်းတူတယ်လို့ ဆိုလိုပါတယ်။ "ကိုလတ်အမေနဲ့ ကျွန်တော့်အမေ အတူတူပဲ" လို့ပြောရင် သူ့အမေနဲ့ ကျွန်တော့်အမေဟာ လူတစ်ယောက်တည်းလို့ ဆိုလိုပါတယ်။ ဒီတော့ တူတယ်ဆိုတဲ့ အယူအဆဟာ အခြေအနေပေါ်လိုက်ပြီး ပြောင်းလဲပါတယ်။

ဝတ္ထုတွေအကြောင်းပြောရင်လည်း ခပ်ဆင်ဆင်တူတဲ့ ဒွိဟဖြစ်မှုကို တွေ့ရပါလိမ့်မယ်။ ဥပမာ Card နှစ်ခုကို တူတယ်လို့ပြောရင် သူတို့မှာ အဆင့်နဲ့အပွင့်ဆိုတဲ့ data တွေတူညီကြသလား၊ ဒါမှမဟုတ် သူတို့ဟာ တကယ်ပဲ Card ဝတ္ထုတစ်ခုတည်းလားဆိုတာ ဒွိဟဖြစ်စရာ ရှိပါတယ်။

အညွှန်းနှစ်ခုဟာ ဝတ္ထုတစ်ခုတည်းကို ညွှန်းဆိုနေမနေ သိချင်ရင် == လက္ခဏာကို သုံးနိုင်ပါတယ်။ ဥပမာ

```
Card card1 = new Card(1, 11);
Card card2 = card1;

if (card1 == card2) {
   System.out.println("card1 and card2 are the same objects.");
}
```

ဒီလိုညီမျှချက်မျိုးကို အပေါ်ယံတူညီခြင်းလို့ ခေါ်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူဟာ အညွှန်း တွေကိုပဲနှိုင်းယှဉ်ပြီး ဝတ္ထုတွေရဲ့ပါဝင်မှုတွေကို နှိုင်းယှဉ်ဖို့ မကြိုးစားလို့ပါပဲ။

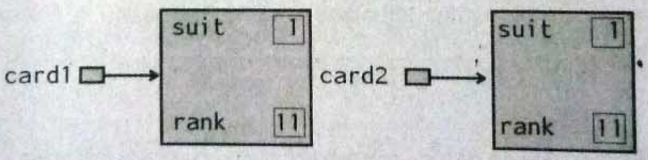
ဝတ္ထုတွေရဲ့ပါဝင်မှုတွေ နှိုင်းယှဉ်တာကို လေးလံသောနှိုင်းယှဉ်ခြင်းလို့ ခေါ်ပါတယ်။ ဒီလို လုပ်ဖို့အတွက် sameCard လို့နာမည်မျိုးရှိတဲ့ method တစ်ခုကို ရေးလေ့ရှိကြပါတယ်။

```
public static boolean sameCard(Card c1, Card c2) {
   return(c1.suit == c2.suit && c1.rank == c2.rank);
}
```

အခုအချိန်မှာ တူညီတဲ့ data ပါဝင်တဲ့ ဝတ္ထုနှစ်ခုကို ကွဲကွဲပြားပြားဖန်တီးပြီး တူညီတဲ့ဖဲ ချပ်ကို ကိုယ်စားပြုသလားဆိုတာ sameCard ကိုသုံးပြီး ကြည့်ရှုနိုင်ပါပြီ။

```
Card card1 = new Card(1, 11);
Card card2 = new Card(1, 11);

if (sameCard(card1, card2)) {
   System.out.println("card1 and card2 are the same card.");
}
```

ဒီနေရာမှာ card1 နဲ့ card2 ဟာ တူညီတဲ့ data ပါဝင်တဲ့ ဝတ္ထုကွဲနှစ်ခုဖြစ်ပါတယ်။




အဲဒီအတွက် အခြေအနေဟာ ပြေလည်ပါတယ်။ card1 == card2 မှန်တဲ့အခါမှာ အခြေအနေပုံဟာ ဘယ်လိုဖြစ်နေမယ် ထင်သလဲ။

ပြီးခဲ့တဲ့ အပိုင်းတစ်ပိုင်းမှာတုန်းက == လက္ခဏာကို String တွေမှာသုံးရင် ကိုယ်မျှော် လင့်တဲ့အလုပ်ကို လုပ်မှာမဟုတ်တဲ့အတွက် ရှောင်ကျဉ်ဖို့ အကြံပေးခဲ့ပါတယ်။ String ရဲ့ပါဝင် မှုတွေကို နှိုင်းယှဉ်ပြီး နက်နဲတဲ့ညီမျှချက်ကို စစ်ဆေးမယ့်အစား သူက String နှစ်ခုဟာ

တူညီတဲ့ ဝတ္ထုတစ်ခုတည်းလားဆိုတဲ့ အပေါ်ယံညီမျှမှုကိုပဲ စစ်ဆေးပါတယ်။

## compareCard Method

အခြေခံအမျိုးအစားတွေအတွက် တန်ဖိုးတွေကိုနှိုင်းယှဉ်ပြီး ဘယ်ဟာက ဘယ်ဟာထက် ပိုကြီးသလဲဆိုတာ အဆုံးအဖြတ်ပေးတဲ့ ဖြစ်ရပ်မှီခို လက္ခဏာတွေရှိပါတယ်။ < နဲ့ > အပါအဝင် ဒီလက္ခဏာတွေဟာ ဝတ္ထုအမျိုးအစားတွေပေါ်မှာတော့ မလုပ်ဆောင်ပါဘူး။ String တွေအတွက် compareTo ဆိုတဲ့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ method တစ်ခုရှိပါတယ်။ Card တွေအတွက်တော့ ကိုယ်ပိုင် method တစ်ခုကို ရေးရပါမယ်။ ဒါကို compareCard လို့ခေါ်ပါမယ်။ နောက်ပိုင်းကျရင် ဒီ method ကို ဖဲပြန်စီတဲ့နေရာမှာ သုံးကြပါမယ်။ တချို့အစုတွေဟာ အစီအစဉ်ကျမှု အပြည့်အဝရှိကြပါတယ်။ ဆိုလိုတာက ကြိုက်တဲ့အဖွဲ့ဝင်နှစ်ခုကို နှိုင်းယှဉ်ပါ။ ဘယ်သူပိုကြီးသလဲဆိုတာ ပြောနိုင်ပါတယ်။ ဥပမာ ကိန်းပြည့်တွေနဲ့ floating-point ကိန်းတွေမှာ အစီအစဉ်ကျနမှု အပြည့်ရှိပါတယ်။ တချို့အစုတွေကတော့ အစီအစဉ်ကျနမှုမရှိဘဲ ဘယ်အဖွဲ့ဝင်က ဘယ်အဖွဲ့ဝင်ထက် ပိုကြီးသလဲဆိုတာ အဓိပ္ပာယ်ရှိရှိ မပြောနိုင်ပါဘူး။ ဥပမာ သစ်သီးတွေဟာ အစီအစဉ်မကျပါဘူး။ အဲဒါကြောင့်မို့လို့ ပန်းသီးတွေနဲ့ လိမ္မော်သီးတွေကို နှိုင်းယှဉ်လို့မရနိုင်တာ ဖြစ်ပါတယ်။ Java မှာ boolean အမျိုးအစားဟာ အစီအစဉ်မကျပါဘူး။ true က false ထက်ပိုကြီးတယ်လို့ ပြောလို့မရနိုင်ပါဘူး။

ဖဲတစ်ထုပ်ဟာ တစ်စိတ်တစ်ဒေသ အစီအစဉ်ကျပါတယ်။ ဆိုလိုချက်က တစ်ခါတစ်ရံမှာ ဖဲချပ်တွေကို နှိုင်းယှဉ်လို့ရနိုင်သလို တစ်ခါတလေ ဒီလိုလုပ်လို့မရတာတွေ ရှိတတ်ပါတယ်။ ဥပမာ Club ရဲ့ 3 ဟာ Club ရဲ့ 2 ထက်ပိုမြင့်တယ်ဆိုတာ သိပါတယ်။ ဒီလိုပဲ Diamond ရဲ့ 3 ဟာ Club ရဲ့ 3 ထက် ပိုမြင့်တယ်ဆိုတာလည်း သိပါတယ်။ ဒါပေမယ့် Club ရဲ့ 3 နဲ့ Diamond ရဲ့ 2 ဘယ်ဟာပိုမြင့်သလဲ။ တစ်ခုက အဆင့်နေရာပိုမြင့်ပြီး နောက်တစ်ခုက အပွင့်ပိုမြင့်ပါတယ်။

ဖဲချပ်တွေကို နှိုင်းယှဉ်လို့ရအောင် အဆင့်နဲ့အပွင့် ဘယ်ဟာပိုအရေးကြီးသလဲဆိုတာ ဆုံးဖြတ်ရပါမယ်။ ရိုးရိုးသားသားပြောရရင် ဒီရွေးချယ်မှုဟာ ပုဂ္ဂလထင်မြင်ချက်ပေါ်မှာ မူတည်ပါတယ်။ ဒီနေရာမှာတော့ အပွင့်ကပိုအရေးကြီးတယ်လို့ ဆိုချင်ပါတယ်။ ဘာလို့လည်းဆိုတော့ ဖဲထုပ်အသစ်တစ်ထုပ်ဝယ်လိုက်ရင် အပေါ်ဆုံးမှာ Club တွေနဲ့ စီထားပြီး သူ့နောက်မှာ Diamond တွေ လိုက်စီပါတယ်။ ဒီလိုပုံစံနဲ့ ဆက်ပြီးစီသွားပါတယ်။

ဒါကိုဆုံးဖြတ်ပြီးတဲ့အခါ compareCard ကို ရေးနိုင်ပါပြီ။ သူက Card နှစ်ခုကို parameter တွေအနေနဲ့လက်ခံပြီး ပထမဖဲချပ်ကနိုင်ရင် 1 ပြန်ပါတယ်။ ဒုတိယဖဲချပ်ကနိုင်ရင် -1 ပြန်ပါတယ်။ သရေကျရင် (နက်နဲတဲ့တူညီမှုရှိရင်) 0 ပြန်ပါတယ်။ ဒီပြန်တဲ့တန်ဖိုးတွေကို လိုက်မှတ်ရတာ ဦးနှောက်နည်းနည်းတော့ ရှုပ်ပါတယ်။ ဒါပေမယ့် သူတို့ဟာ နှိုင်းယှဉ်တဲ့ method တွေအတွက် စံတစ်ခုလိုတောင် ဖြစ်နေပါပြီ။

အရင်ဆုံး အပွင့်တွေကို နှိုင်းယှဉ်ပါမယ်။

```
if (c1.suit > c2.suit) return 1;
if (c1.suit < c2.suit) return -1;
```

အပေါ်မှာပြောထားတဲ့ ဖော်ပြချက်နှစ်ခုလုံးမမှန်ရင် အပွင့်တွေထားပြီး အဆင့်တွေကို ဆက်ပြီး နှိုင်းယှဉ်ရပါမယ်။

```
if (c1.rank > c2.rank) return 1;
if (c1.rank < c2.rank) return -1;
```

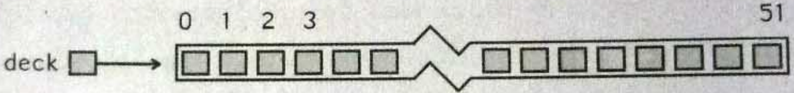
ဒီနှစ်ခုစလုံးမမှန်ဘူးဆိုရင် အဆင့်တွေလည်း တူရပါမယ်။ ဒီတော့ 0 ကိုပြင်ပါတယ်။ ဒီစီစဉ်ပုံမှာ Ace တွေဟာ 2 တွေထက် ပိုနိမ့်ပါလိမ့်မယ်။

### ဖဲချပ်များ၏ Array များ

ဒီအခန်းအတွက် ဝတ္ထုတွေအဖြစ် Card တွေကို ရွေးချယ်ခဲ့ရခြင်းအကြောင်းက ဖဲချပ်တွေရဲ့ array တစ်ခုအတွက် ထင်သာမြင်သာရှိတဲ့ အသုံးတစ်ခု ရှိတာကြောင့်ပဲဖြစ်ပါတယ်။ အဲဒါကတော့ ဖဲတစ်ထုပ်ပါ။ အောက်မှာပြထားတာကတော့ ဖဲချပ် 52 ချပ်ပါတဲ့ ဖဲထုပ်အသစ်တစ်ထုပ်ကိုဖန်တီးတဲ့ code တွေပဲဖြစ်ပါတယ်။

```
Card[] deck = new Card[52];
```

အောက်မှာပြထားတာကတော့ ဒီဝတ္ထုအတွက် အခြေအနေပုံပဲ ဖြစ်ပါတယ်။




ဒီနေရာမှာ မြင်ဖို့အရေးကြီးတာက array ဟာ ဝတ္ထုတွေဆီသွားတဲ့ အညွှန်းတွေကိုပဲ ပိုင်ဆိုင်တယ်ဆိုတာ ဖြစ်ပါတယ်။ သူ့မှာ Card ဝတ္ထုတွေ မပါပါဘူး။ Array အဖွဲ့ဝင်တွေရဲ့ တန်ဖိုးတွေကိုလည်း null မှာစတင်ထားပါတယ်။ Array ရဲ့အဖွဲ့ဝင်တွေကို ပုံမှန်နည်းသုံးပြီး ရရှိနိုင်ပါတယ်။

```
if (deck[3] == null) {
   System.out.println("No cards yet!");
}
```

ဒါပေမယ့် တကယ်ရှိမနေတဲ့ Card တွေရဲ့ instance variable တွေကို ရရှိဖို့ကြိုးစားရင် NullPointerException ကို ရပါမယ်။

```
deck[2].rank;            // NullPointerException
```

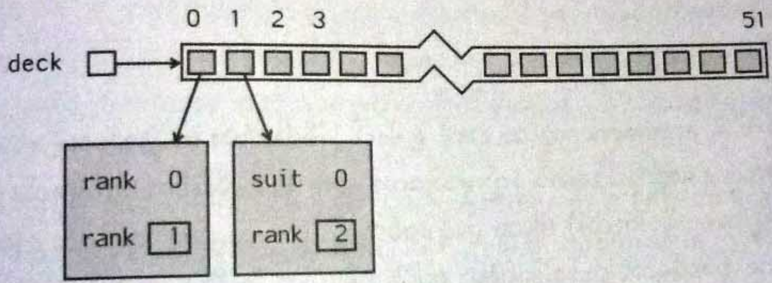
ဒါပေမယ့်လည်း သူဟာ ဖဲထုပ်ရဲ့ နှစ်ခုမြောက် rank ကိုရရှိတဲ့ သဒ္ဒါအမှန်ဖြစ်ပါတယ်။ (တကယ်တမ်းကျတော့ တတိယမြောက်ဖြစ်ပါတယ်။ ရေတွက်မှုကို သုညကစတင်ခဲ့တယ်ဆိုတာ သတိရပါ။) ဒါဟာ ဖွဲ့စည်းမှုရဲ့ နောက်ထပ်ဥပမာတစ်ခုပါပဲ။ Array တစ်ခုရဲ့ အဖွဲ့ဝင်တစ်ခုကို ရရှိနိုင်တဲ့သဒ္ဒါနဲ့ ဝတ္ထုတစ်ခုရဲ့ instance variable တစ်ခုကို ရရှိနိုင်တဲ့သဒ္ဒါကို ပေါင်းစပ်ထား

တာဖြစ်ပါတယ်။ ဖဲထုပ်ကို Card ဝတ္ထုတွေနဲ့ဖြည့်ဖို့ အလွယ်ဆုံးနည်းကတော့ အဆင့်ဆင့်ထပ်ထား တဲ့ သံသရာတစ်ခုကို ရေးဖို့ပါပဲ။

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
   for (int rank = 1; rank <= 13; rank++) {
         deck[index] = new Card(suit, rank);
         index++;
   }
}
```

အပြင်ကသံသရာဟာ အပွင့်တွေကို 0 ကနေ 3 အထိ ရေပါတယ်။ အပွင့်တစ်မျိုးစီအတွက် အတွင်းကသံသရာဟာ အဆင့်တွေကို 1 ကနေ 13 အထိ ရေပါတယ်။ အပြင်ကသံသရာဟာ 4 ကြိမ်လည်ပတ်ပြီး အတွင်းကသံသရာကတော့ 13 ကြိမ်လည်ပတ်တဲ့အတွက် ကိုယ်ထည်ကို run တဲ့အကြိမ်ဟာ (13 x 4) 52 ကြိမ် ဖြစ်ပါတယ်။

index ကို နောက်လာတဲ့ဖဲချပ်ဘယ်နေရာမှာရှိနေသလဲဆိုတာ လိုက်မှတ်ဖို့သုံးခဲ့ပါတယ်။ အောက်မှာပြထားတဲ့ပုံဟာ ပထမဆုံးကတ်ပြားနှစ်ခုကို နေရာချထားပြီးတဲ့အခါ ရရှိလာမယ့် ဖဲထုပ်ရဲ့အခြေအနေကို ပြသပါတယ်။




## printDeck Method

Array တွေနဲ့ အလုပ်လုပ်တဲ့အခါတိုင်း array ရဲ့ပါဝင်မှုတွေကို print လုပ်ပေးမယ့် method တစ်ခုရှိထားရင် အဆင်ပြေပါတယ်။ Array တစ်ခုကို ဖြတ်လျှောက်တဲ့ပုံစံကို အကြိမ်ကြိမ်မြင်ခဲ့ ပြီးပါပြီ။ ဒီတော့ အောက်မှာပြထားတဲ့ method ဟာ ရင်းနှီးနေသင့်ပါတယ်။

```
public static void printDeck(Card[] deck) {
   for (int i = 0; i < deck.length; i++) {
         printCard(deck[i]);
   }
}
```

deck မှာ Card[] အမျိုးအစားရှိတဲ့အတွက် deck ရဲ့အဖွဲ့ဝင်တွေမှာ Card အမျိုးအစား

ရှိပါတယ်။ ဒီတော့ deck[i] ဟာ printCard အတွက် တရားဝင် argument တစ်ခုဖြစ်ပါတယ်။

**ရှာဖွေခြင်း**

နောက်ဆက်ရေးချင်တဲ့ method ကတော့ findCard ပဲ ဖြစ်ပါတယ်။ သူက Card တွေရဲ့ array မှာ ဖဲချပ်တစ်ခုပါမပါ စစ်ဆေးဖို့ ရှာဖွေမှုတွေ ပြုလုပ်ပါတယ်။ ဒီ method ဟာ ဘာကြောင့် အသုံးဝင်လာလိမ့်မလဲဆိုတာ အခုအချိန်မှာ မြင်ဖို့ခက်ခဲပါတယ်။ ဒါပေမယ့် လောလောဆယ်တော့ ရှာဖွေနည်းနှစ်နည်းကို သရုပ်ဖော်ဖို့ အခွင့်အရေးရပါတယ်။ သူတို့ကတော့ linear search နဲ့ bisection search တို့ ဖြစ်ပါတယ်။

Linear search ဟာ ရှာဖွေနည်းနှစ်နည်းထဲက ပိုပြီးသိသာထင်ရှားတဲ့ ရှာဖွေနည်းဖြစ်ပါတယ်။ သူက ဖဲထုပ်ကိုဖြတ်လျှောက်ပြီး ကိုယ်ရှာနေတဲ့ဖဲချပ်ကို ဖဲချပ်တစ်ခုချင်းနဲ့ လိုက်တိုက်ကြည့်ပါတယ်။ ရှာတွေ့ရင် ဖဲချပ်ပေါ်ပေါက်တဲ့ index ကို ပြန်ထုတ်ပေးပါတယ်။ ဖဲထုပ်ထဲမှာ မရှိရင် -1 ကို ပြန်ထုတ်ပေးပါတယ်။

```
public static int findCard(Card[] deck, Card card) {
    for (int i = 0; i < deck.length; i++) {
        if (sameCard(deck[i], card) return i;
    }
    return -1;
}
```

findCard ရဲ့ argument တွေဟာ card နဲ့ deck ဖြစ်ပါတယ်။ အမျိုးအစားနဲ့ နာမည်အတူတူရှိတဲ့ variable တွေကိုမြင်ရတာ နည်းနည်းတော့ ထူးဆန်းသလိုဖြစ်နေပါလိမ့်မယ်။ (card မှာ Card အမျိုးအစားရှိပါတယ်။) ဒါဟာ တရားဝင်သလို အသုံးလည်းများပါတယ်။ ဒါပေမယ့် သူ့ကြောင့် code ကိုဖတ်ရတာ တစ်ခါတစ်ရံမှာ ခက်ခဲစေနိုင်ပါတယ်။ ဒီနေရာမှာတော့ အလုပ်ဖြစ်မြောက်ပါတယ်။

ဒီ method က ဖဲချပ်ကိုရှာဖွေလို့ရတာနဲ့ တန်ဖိုးပြန်ထုတ်ပေးပါတယ်။ ဆိုလိုတာက ရှာဖွေနေတဲ့ဖဲချပ်ကိုတွေ့ရင် ဖဲတစ်ထုပ်လုံးကို ဖြတ်လျှောက်စရာမလိုပါဘူး။ ဖဲချပ်ကိုရှာမတွေ့ဘဲ သံသရာဆုံးသွားရင် ဖဲချပ်ဟာ ဖဲထုပ်ထဲမှာမရှိဘူးဆိုတာ သိတဲ့အတွက် -1 ပြန်ပါတယ်။

ဖဲထုပ်ထဲကဖဲချပ်တွေဟာ အစီအစဉ်တကျ ရှိမနေဘူးဆိုရင် ဒီထက်မြန်အောင် ရှာနိုင်တဲ့နည်း မရှိတော့ပါဘူး။ ဖဲချပ်တိုင်းကို လိုက်ကြည့်ရပါမယ်။ ဘာလို့လဲဆိုတော့ ဒီလိုမှမလုပ်ရင် လိုချင်တဲ့ဖဲချပ်ဟာ အဲဒီမှာရှိနေမနေ သိနိုင်တဲ့နည်း မရှိလို့ပါပဲ။

ဒါပေမယ့် အဘိဓာန်တစ်ခုထဲမှာ စကားလုံးတစ်လုံးကိုရှာချင်ရင် ဝေါဟာရတစ်ခုချင်းစီ လိုက်ကြည့်မနေပါဘူး။ အကြောင်းကတော့ စကားလုံးတွေဟာ အက္ခရာအစီအစဉ်အတိုင်း ရှိနေ

လို့ပါပဲ။ ရလဒ်ကတော့ bisection search တစ်ခုနဲ့ဆင်တူတဲ့ algorithm တစ်ခုကို သုံးကြပါတယ်။

၁။ အလယ်တစ်နေနေရာက စရပါတယ်။

၂။ စာမျက်နှာပေါ်မှာ စကားလုံးတစ်လုံးကိုရွေးပြီး ကိုယ်ရှာနေတဲ့စကားလုံးနဲ့ နှိုင်းယှဉ်ကြည့်ရပါတယ်။

၃။ ကိုယ်ရှာနေတဲ့ စကားလုံးကိုတွေ့ရင် ရပ်တန့်ကရပ်ရပါတယ်။

၄။ ကိုယ်ရှာနေတဲ့စကားလုံးက စာမျက်နှာပေါ်မှာတွေ့ထားတဲ့ စကားလုံးရဲ့နောက်ကို ရောက်နေရင် အဘိဓာန်ရဲ့ နောက်ပိုင်းတစ်နေရာမှာ ရှာကြည့်ရပါတယ်။ အဆင့် ၂ ကနေ ပြန်စရပါတယ်။

၅။ ကိုယ်ရှာနေတဲ့စကားလုံးဟာ စာမျက်နှာပေါ်ကစကားလုံးရဲ့ ရှေ့ကိုရောက်နေရင် အဘိဓာန်ရဲ့ အစောပိုင်းတစ်နေရာကိုသွားပြီး အဆင့် ၂ ကို ပြန်လုပ်ရပါတယ်။

စာမျက်နှာပေါ်မှာ ရှေ့နောက်ကပ်လျက်ရှိနေတဲ့ စကားလုံးနှစ်ခုကိုတွေ့ပြီး ကိုယ်ရှာနေတဲ့စကားလုံးဟာ သူတို့ကြားမှာ ရောက်နေတယ်ဆိုရင် ကိုယ်ရှာနေတဲ့စကားလုံးဟာ အဘိဓာန်ထဲမှာ မရှိဘူးဆိုတာ သေချာပါပြီ။ မျှော်လင့်ရတာတစ်ခုကတော့ ကိုယ့်စကားလုံးကို တစ်နေရာမှာမှား ထည့်ထားခဲ့ပါတယ်။ ဒီလိုဆိုရင်တောင်မှ ဒီအဖြစ်အပျက်ဟာ စကားလုံးတွေကို အက္ခရာစဉ်အတိုင်း စီထားတယ်ဆိုတဲ့ မူလယူဆချက်နဲ့ ဖီလာဆန့်ကျင်နေပါတယ်။

ဖဲထုပ်အခြေအနေအတွက် ဖဲချပ်တွေအစီအစဉ်အတိုင်း ရှိနေတယ်ဆိုတာသိရင် ပိုပြီးမြန်တဲ့ `findCard` ကို ရေးနိုင်ပါတယ်။ Bisection search တစ်ခုကို ရေးနိုင်တဲ့အကောင်းဆုံးနည်းကတော့ အပြန်ပြန်အလှန်လှန် ဆောင်ရွက်တဲ့နည်းပဲဖြစ်ပါတယ်။ ဒါဟာ ဘာဖြစ်လို့လဲဆိုတော့ bisection ရဲ့သဘာဝဟာ အလိုအလျောက် အပြန်ပြန်အလှန်လှန် သဘောရှိတဲ့အတွက်ပဲ ဖြစ်ပါတယ်။

လှည့်ကွက်ကတော့ index နှစ်ခုကို `low` နဲ့ `high` ဆိုတဲ့ parameter အဖြစ်လက်ခံတဲ့ method တစ်ခုကို ရေးဖို့ပါပဲ။ သူတို့ဟာ `low` နဲ့ `high` အပါအဝင် ရှာဖွေရမယ့် array ရဲ့ အပိုင်းကို သတ်မှတ်ပါတယ်။

၁။ Array ကိုရှာဖွေဖို့ `low` နဲ့ `high` ကြားက index တစ်ခုကို ရွေးချယ်ပါ။ သူ့ကို `mid` လို့ နာမည်ပေးပြီး ကိုယ်ရှာနေတဲ့ဖဲချပ်နဲ့ တူမတူ နှိုင်းယှဉ်ပါ။

၂။ ရှာတွေ့ရင် ရပ်တန့်ကရပ်ပါ။

၃။ `mid` မှာရှိတဲ့ဖဲချပ်ဟာ ကိုယ့်ဖဲချပ်ထက်ပိုမြင့်နေရင် `low` နဲ့ `mid-1` ကြားက အပိုင်းအခြားထဲမှာ ရှာဖွေရပါမယ်။

၄။ `mid` မှာရှိတဲ့ဖဲချပ်ဟာ ကိုယ့်ဖဲချပ်ထက်နိမ့်နေရင် `mid+1` နဲ့ `high` ကြားက အပိုင်းအခြားမှာ ရှာရပါတယ်။

အဆင့် ၃ နဲ့ ၄ ဟာ သံသယဖြစ်စရာကောင်းလောက်အောင် အထပ်ထပ်အခါခါ လုပ်ကိုင်တဲ့ တီထွင်မှုတွေနဲ့ တူပါတယ်။ အောက်မှာပြထားတာက ဒါတွေအားလုံးကို Java code အနေနဲ့

ဘာသာပြန်လိုက်ရင် မြင်ရမယ့်ပုံပါပဲ။

```
public static int findBisect(Card[] deck, Card card, int low, int high)
{
    int mid = (high + low) / 2;
    int comp = compareCard(deck[mid], card);

    if (comp == 0) {
        return mid;
    } else if (comp > 0) {
        return findBisect(deck, card, low, mid-1);
    } else {
        return findBisect(deck, card, mid+1, high);
    }
}
```

compareCard ကိုသုံးခါခေါ်မယ့်အစား သူ့ကိုတစ်ခါခေါ်ပြီး ရလဒ်ကို သိမ်းထားလိုက်ပါတယ်။

ဒီ code ထဲမှာ bisection search ရဲ့ အတွင်းပိုင်းအယူအဆပါပေမယ့် တစ်ခုခုတော့ လိုနေပါသေးတယ်။ လက်ရှိရေးထားတဲ့ပုံအတိုင်းဆိုရင် ဖဲထုပ်ထဲမှာ ဖဲချပ်ရှိနေတဲ့အခါ သံသရာလည်လို့ ဆုံးတော့မှာမဟုတ်ပါဘူး။ ဒီအခြေအနေကို စောစောစီးစီးသိပြီး ထိထိရောက်ရောက် ကိုင်တွယ်နိုင်တဲ့ နည်းတစ်နည်း ရှာဖို့လိုပါတယ်။ -1 ပြန်ပေးရပါမယ်။

ဒီစာကြောင်းကိုပေါင်းထည့်လိုက်ရင် ဒီ method ဟာ အလုပ်ကောင်းကောင်း လုပ်ပါလိမ့်မယ်။

```
public static int findBisect(Card[] deck, Card card, int low, int high)
{
    System.out.println(low + ", " + high);

    if (high < low) return -1;

    int mid = (high + low) / 2;
    int comp = compareCard(deck[mid], card);

    if (comp == 0) {
```

```
        return mid;
    } else if (comp > 0) {
        return findBisect(deck, card, low, mid-1);
    } else {
        return findBisect(deck, card, mid+1, high);
    }
}
```

အစပိုင်းမှာ print ဖော်ပြချက်တစ်ခုကို ပေါင်းထည့်ပြီး အပြန်ပြန်အလှန်လှန် ခေါ်ဆိုမှုဖြစ်စဉ်တွေကို ကြည့်ရှုနိုင်အောင် လုပ်ထားပါတယ်။ ဒါမှ တဖြည်းဖြည်းနဲ့ အခြေခံအခြေအနေကို ရောက်ရှိမယ်ဆိုတာ ကိုယ့်ကိုကိုယ်ယုံအောင် လုပ်နိုင်မှာဖြစ်ပါတယ်။ အောက်မှာပြထားတဲ့ code ကို စမ်းလုပ်ကြည့်မယ်ဆိုပါစို့။

```
Card card1 = new Card(1, 11);
System.out.println(findBisect(deck, card1, 0, 51));
```

ဒါဟာ အောက်မှာပြထားတဲ့ output မျိုးကို ထုတ်ပေးပါလိမ့်မယ်။

```
0, 51
0, 24
13, 24
19, 24
22, 24
23
```

ဖဲထုပ်ထဲမှာ မရှိတဲ့ဖဲချပ် (Diamond ရဲ့ 15) ကို လုပ်ကြံဖန်တီးပြီး သူ့ကိုရှာဖို့ ကြိုးစားကြည့်လိုက်ပါတယ်။ အောက်မှာပြထားသလို ရပါလိမ့်မယ်။

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
-1
```

ဒီစမ်းသပ်ချက်တွေဟာ ဒီပရိုဂရမ်မှန်ကြောင်း သက်သေမပြပါဘူး။ တကယ်ဆိုရင် ဘယ်လောက်ပဲစမ်းကြည့်ကြည့် ပရိုဂရမ်မှန်ကြောင်းကို သက်သေမပြနိုင်ပါဘူး။ အခြားတစ်ဖက်မှာတော့ အခြေအနေအနည်းငယ်ကို လေ့လာပြီး code ကိုစစ်ဆေးကြည့်ရင် ကိုယ့်ကိုယ်ကိုယုံအောင်

လုပ်နိုင်ပါတယ်။

အကြိမ်ကြိမ်ခေါ်ဆိုမှုအရေအတွက်ဟာ အတန်အသင့်နည်းပါတယ်။ ပုံမှန်အားဖြင့် ၆ ကြိမ် 7 ကြိမ်လောက် ရှိတတ်ပါတယ်။ ကွန်ပျူတာက compareCard ကို 6 ကြိမ်နဲ့ 7 ကြိမ်ကြားပဲခေါ်ရပါတယ်။ Linear search နဲ့ရှာရင် 52 ကြိမ် ခေါ်ဆိုရတာနဲ့ ယှဉ်ကြည့်တဲ့အခါ တော်တော်နည်းတယ်လို့ ဆိုရမှာပါ။ ယေဘုယျအားဖြင့် bisection ဟာ Linear search ထက်ပိုမြန်ပါတယ်။ ကြီးမားတဲ့ array တွေမှာ ပိုလို့တောင်မြန်ပါသေးတယ်။

အပြန်ပြန်အလှန်လှန်ခေါ်ဆိုတဲ့ ပရိုဂရမ်တွေမှာ ဖြစ်တတ်တဲ့အမှားနှစ်ခုကတော့ အခြေအနေကို ထည့်ရေးဖို့မေ့တာနဲ့ အခြေခံအခြေအနေကို ဘယ်တော့မှမရောက်အောင် အပြန်ပြန်အလှန်လှန်ခေါ်ဆိုမှုကို ရေးသားမိတာ ဖြစ်ပါတယ်။ ဒီအမှားနှစ်ခုစလုံးဟာ အဆုံးမသတ်နိုင်တဲ့ အပြန်ပြန်အလှန်လှန်လုပ်ကိုင်မှုကို ဖြစ်ပေါ်စေပြီး နောက်ဆုံးမှာတော့ Java ဟာ StackOverflowException တစ်ခုကို ဖြစ်ပေါ်စေပါလိမ့်မယ်။

**ဖဲထုပ်များနှင့် လက်အောက်ခံဖဲထုပ်များ**

```
public static int findBisect(Card[] deck, Card card, int low, int high)
```

findBisect ရေးထားတဲ့ပုံကိုကြည့်ရင် deck, low နဲ့ high ဆိုတဲ့ parameter သုံးခုကို လက်အောက်ခံဖဲထုပ်တစ်ခုကို သတ်မှတ်တဲ့ parameter တစ်ခုတည်းအဖြစ် မြင်ကြည့်ရင် အဓိပ္ပာယ်ရှိပါလိမ့်မယ်။ ဒီလိုတွေးနည်းမျိုးကို မကြာခဏပြုလုပ်လေ့ရှိပြီး စိတ်ကူးပြုလုပ်ထားတဲ့ parameter လို့ ဆိုနိုင်ပါတယ်။ စိတ်ကူးပြုလုပ်ထားတယ်ဆိုတာ ပရိုဂရမ်စာသားရဲ့တစ်စိတ်တစ်ဒေသအဖြစ် တိုက်ရိုက်မပါဝင်ပေမယ့် ပရိုဂရမ်ရဲ့လုပ်ကိုင်ပုံကို အဆင့်မြင့်အနေအထားတစ်ခုမှာ ဖော်ပြတဲ့အရာတစ်ခု ဖြစ်ပါတယ်။

ဥပမာ method တစ်ခုကိုနှိုးဆွပြီး array တစ်ခုနဲ့ အထက်အောက်အကန့်အသတ် low နဲ့ high ကို ပေးပို့တဲ့အခါ နှိုးဆွခံရတဲ့ method အနေနဲ့ အကန့်အသတ်ပြင်ပက array အစိတ်အပိုင်းတွေကို ရောက်ရှိနိုင်တဲ့အဖြစ်အပျက်ကနေ ကာကွယ်ပေးထားတာ ဘာမှမရှိပါဘူး။ ဒီတော့ တကယ်လုပ်နေတာက ဖဲထုပ်ရဲ့အစိတ်အပိုင်းတစ်ခုကို ပေးပို့နေတာမဟုတ်ဘဲ ဖဲတစ်ထုပ်လုံးကို ပေးပို့နေတာဖြစ်ပါတယ်။ ဒါပေမယ့် လက်ခံသူက စည်းမျဉ်းစည်းကမ်းတွေကို လိုက်နာသရွေ့ သူ့ကို စိတ်ကူးပြုလုပ်ထားတဲ့ သဘောအတိုင်း လက်ခံဖဲထုပ်အဖြစ် တွေးကြည့်ရင် အဓိပ္ပာယ်ရှိပါတယ်။

ပြီးခဲ့တဲ့အပိုင်းတစ်ပိုင်းမှာတုန်းက ဒီလိုစိတ်ကူးပြုလုပ်တာမျိုးကို ဥပမာအနေနဲ့ တွေ့ခဲ့ပါတယ်။ ဒါကို လွတ်နေတဲ့ data ပုံစံလို့ ပြောခဲ့ပါတယ်။ ဒါပေမယ့် လွတ်နေတယ်ဆိုတာ တကယ်တွေးကြည့်ရင်တော့ သိပ်မမှန်ပါဘူး။ ဘယ် variable မဆို အချိန်တိုင်း တန်ဖိုးတွေရှိပါတယ်။ သူတို့ကိုဖန်တီးတဲ့အခါ မူလတန်ဖိုးပေးခြင်း ခံရပါတယ်။ ဒီတော့ လွတ်နေတဲ့ဝတ္ထုရယ်လို့ မရှိပါဘူး။

ဒါပေမယ့် ပရိုဂရမ်ဟာ variable တစ်ခုရဲ့ လက်ရှိတန်ဖိုးကို ရေးသားသမှုမပြုခင် ဖတ်ရှုတဲ့အလုပ် မလုပ်ဘူးလို့ တာဝန်ခံနိုင်တဲ့အခါ လက်ရှိတန်ဖိုးဟာ အကြုံးမဝင် ဖြစ်သွားပါတယ်။ စိတ်ကူးပြုလုပ်တဲ့ သဘောတရားအရ ဒီလို variable တစ်ခုကို လွတ်နေတယ်လို့တွေးကြည့်ရင် အဓိပ္ပာယ်ရှိပါတယ်။

ပရိုဂရမ်တစ်ပုဒ်အနေနဲ့ ရေးသားပုံကနေ တိုက်ရိုက်ထုတ်ယူလို့ရတဲ့ အဓိပ္ပာယ်ထက် ပိုမိုနက်နဲမှုရှိလာရင် ဒီလိုတွေးနည်းမျိုးကို စိတ်ကူးပြုလုပ်တယ်လို့ခေါ်ပြီး ကွန်ပျူတာသမားလိုတွေးနည်းရဲ့ တစ်စိတ်တစ်ဒေသဖြစ်ပါတယ်။ တစ်ခါတစ်ရံမှာ စိတ်ကူးပြုလုပ်တယ်ဆိုတဲ့ ဝေါဟာရကို အသုံးများလွန်းလို့ အဓိပ္ပာယ်တောင် ပျောက်သွားပါတယ်။ ဒါတောင်မှ စိတ်ကူးပြုလုပ်တာဟာ ကွန်ပျူတာသိပ္ပံနဲ့ တခြားဘာသာရပ်တွေမှာ အဓိကကျတဲ့ တွေးခေါ်နည်းဖြစ်ပါတယ်။

စိတ်ကူးပြုလုပ်တာရဲ့ ယေဘုယျပိုကျတဲ့ အဓိပ္ပာယ်ဖွင့်ဆိုချက်ဟာ ရှုပ်ထွေးတဲ့စနစ်တစ်ခုမှာ မလိုအပ်တဲ့ အသေးစိတ်အချက်အလက်တွေကို ချုံ့နိုင်ဖို့ သင့်လျော်တဲ့အပြုအမူကို ထိန်းသိမ်းထားတဲ့ ဖော်ပြချက်ရိုးရိုးရှင်းရှင်းလေးတစ်ခုနဲ့ ပုံစံထုတ်တဲ့ ဖြစ်စဉ်ဖြစ်ပါတယ်။

အခန်း ၁၂

# Array များ၏ ဝတ္ထုများ

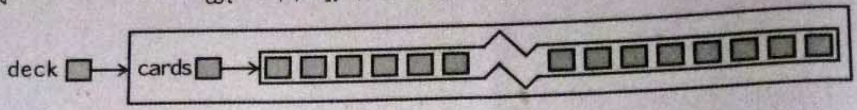
## Deck Class

ပြီးခဲ့တဲ့အခန်းမှာတုန်းက ဝတ္ထုတွေရဲ့ array တစ်ခုနဲ့ အလုပ်တွေလုပ်ခဲ့ပါတယ်။ ပြီးတော့ ဝတ္ထုတစ်ခုမှာ array တစ်ခုကို instance variable အနေနဲ့ ပါဝင်နိုင်တယ်ဆိုတာကိုလည်း ပြောခဲ့ပါတယ်။ ဒီအခန်းမှာ Card တွေရဲ့ array တစ်ခုကို instance variable တစ်ခုအနေနဲ့ ပိုင်ဆိုင်တဲ့ Deck ဝတ္ထုတစ်ခုကို ဖန်တီးကြပါမယ်။

Class အဓိပ္ပာယ်သတ်မှတ်ချက်ဟာ အောက်မှာပြထားသလို ရှိပါတယ်။

```
class Deck {
   Card[] cards;

   public Deck(int n) {
      cards = new Card[n];
   }
}
```

Deck ဝတ္ထုကနေ သူ့မှာပါဝင်တဲ့ Card တွေရဲ့ array နဲ့ ကွဲပြားအောင် instance variable ကို Card လို့နာမည်ပေးလိုက်ပါတယ်။ အောက်မှာပြထားတာကတော့ ဖဲချပ်တွေကို နေရာချမထားခင် Deck ဝတ္ထုအနေနဲ့ ရှိနေတဲ့အခြေအနေပုံပါ။




အရင်လိုပဲ constructor က instance variable ကို စတင်ပါတယ်။ ဒါပေမယ့် ဒီနေရာမှာတော့ သူက new command ကိုသုံးပြီး ဖဲချပ်တွေရဲ့ array တစ်ခုကို ဖန်တီးပါတယ်။ သူက ဖဲချပ်တွေကို သူ့ထဲမှာ ဖန်တီးတာကိုတော့ မလုပ်ပါဘူး။ အဲဒီအတွက် စံချိန်စံညွှန်းမီဖဲချပ် 52 ချပ်ပါတဲ့ ဖဲထုပ်တစ်ခုကို ဖန်တီးပြီး သူ့ကို Card ဝတ္ထုတွေနဲ့ လိုက်ဖြည့်ရပါမယ်။

```
public Deck() {
    cards = new Card[52];
    int index = 0;
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            cards[index] = new Card(suit, rank);
            index++;
        }
    }
}
```

အခုအချိန်မှာ Deck class တစ်ခုကို ရပြီဖြစ်တဲ့အတွက် Deck ရဲ့ class အဓိပ္ပာယ်သတ်မှတ်ချက်ထဲမှာ Deck တွေနဲ့ဆိုင်တဲ့ method အားလုံးကို ထည့်ပေးသင့်ပါတယ်။ အရင်တုန်းက ရေးခဲ့တဲ့ method တွေကိုကြည့်ရင် အခုရေးသင့်တဲ့ method တစ်ခုကတော့ printDeck ပဲ ဖြစ်ပါတယ်။ အောက်မှာပြထားတာကတော့ Deck ဝတ္ထုနဲ့ အလုပ်ဖြစ်အောင် ပြင်ရေးထားတဲ့ printDeck ဖြစ်ပါတယ်။

```
public static void printDeck(Deck deck) {
    for (int i = 0; i < deck.cards.length; i++) {
        Card.printCard(deck.cards[i]);
    }
}
```

ပြောင်းလဲရမယ်လို့ ကွက်ကွက်ကွင်းကွင်းသိနိုင်တဲ့ ကိစ္စကတော့ parameter အမျိုးအစားပါပဲ။ Card[] ကနေ Deck ဆီကို ပြောင်းရပါမယ်။ ဒုတိယပြောင်းလဲမှုကတော့ array ရဲ့အလျားကိုရဖို့ deck.length ကို ဆက်သုံးလို့မရတော့ပါဘူး။ ဘာဖြစ်လို့လဲဆိုတော့ deck ဟာ Deck

ဝတ္ထုတစ်ခု ဖြစ်သွားပါပြီ။ Array တစ်ခု မဟုတ်တော့ပါဘူး။ သူ့မှာ array တစ်ခုပါဝင်ပေမယ့် သူကိုယ်တိုင်က array တစ်ခု မဟုတ်တော့ပါဘူး။ ဒီတော့ Deck ဝတ္ထုတစ်ခုကနေ array တစ်ခုကိုထုတ်နုတ်ပြီး အလျားကိုရယူဖို့ deck.cards.length ကို ရေးဖို့လိုပါတယ်။

ဒီအကြောင်းကြောင့်ပဲ array ရဲ့ အဖွဲ့ဝင်ဆီရောက်ရှိဖို့ deck.cards[i] ကို deck[i] တစ်ခုတည်းအစား သုံးရတာဖြစ်ပါတယ်။ နောက်ဆုံးပြုလုပ်ရတဲ့ ပြောင်းလဲမှုကတော့ printCard နှိုးဆွမှုမှာ printCard ကို Card class ထဲမှာ သတ်မှတ်ထားတယ်လို့ ရှင်းရှင်းလင်းလင်း ပြောရတာဖြစ်ပါတယ်။

တခြား method တချို့အတွက်ကတော့ သူတို့ကို Card class ထဲမှာ ထည့်ရမလား၊ Deck class ထဲမှာ ထည့်ရမလားဆိုတာ မကွဲပြားပါဘူး။ ဥပမာ findCard ဟာ Card တစ်ခုနဲ့ Deck တစ်ခုကို argument တွေအနေနဲ့ လက်ခံတဲ့အတွက် သူ့ကို ဘယ် class ထဲမှာပဲထည့်ထည့် ယုတ္တိရှိပါတယ်။ လေ့ကျင့်ခန်းတစ်ခုအနေနဲ့ findCard ကို Deck class ထဲရွှေ့ပြီး ပထမဆုံး parameter ဟာ Card တွေရဲ့ array တစ်ခုအစား Deck ဝတ္ထုတစ်ခုကို လက်ခံအောင် ပြန်ပြင်ရေး ကြည့်ပါလား။

## ဖဲချိုးခြင်း

ဖဲကစားနည်း တော်တော်များများအတွက် ဖဲချိုးတတ်ဖို့လိုပါတယ်။ အင်မတန်ရိုးသားသူများ အတွက် ဖဲချိုးတယ်ဆိုတာ ဖဲချပ်တွေကို ကြုံရာကျပန်း အစီအစဉ်နဲ့ရှိနေအောင် နေရာရွှေ့တာ ဖြစ်ပါတယ်။ ပြီးခဲ့တဲ့အပိုင်းတစ်ပိုင်းမှာတုန်းက ကျပန်းကိန်းတွေကို ဘယ်လိုထုတ်လုပ်ရတယ် ဆိုတာ မြင်ခဲ့ပါတယ်။ ဒါပေမယ့် သူတို့ကိုသုံးပြီး ဖဲတစ်ထုပ်ကို ဘယ်လိုချိုးရမယ်ဆိုတာကိုတော့ သိပ်မထင်ရှားပါဘူး။

ဖြစ်နိုင်ခြေတစ်ခုကတော့ လူတွေဖဲချိုးကြတဲ့ပုံကို အတုယူတာဖြစ်ပါတယ်။ ပုံမှန်ဖဲချိုးပုံဟာ ဖဲထုပ်ကိုနှစ်ပိုင်းပိုင်းပြီး တစ်ပိုင်းဆီကဖဲချပ်တွေကို တစ်လှည့်စီပြန်စီပြီး ဖဲထုပ်ကို ပြန်ဖွဲ့စည်းတာ ဖြစ်ပါတယ်။ လူသားတွေရဲ့ဖဲချိုးပုံဟာ အပြစ်ကင်းစင်မှုမရှိတဲ့အတွက် (အမှားမကင်းတဲ့အတွက်) ခုနစ်ခါလောက် အပြန်ပြန်အလှန်လှန်လုပ်လိုက်ရင် ဖဲထုပ်ရဲ့အစီအစဉ်ဟာ အတော်အတန်ကျ ပန်းဖြစ်သွားပါတယ်။ ဒါပေမယ့် ကွန်ပျူတာပရိုဂရမ်တစ်ပုဒ်ဟာ အချိန်တိုင်း အင်မတန်အပြစ် ကင်းစင်တဲ့ ဖဲချိုးနည်းနဲ့ ဖဲထုပ်ကို မွှေနှောက်ပြပါလိမ့်မယ်။ ဒါဟာ သိပ်ပြီးကျပန်းမဖြစ်ပါဘူး။ တကယ်တမ်းကျတော့ ဖဲထုပ်ကို ရှစ်ကြိမ်ဖဲချိုးအပြီးမှာ ဖဲထုပ်ကို ကိုယ်စခဲ့တဲ့အစီအစဉ်နဲ့ ပုံစံ မပျက် ပြန်တွေ့ရပါလိမ့်မယ်။ ဒီအဆိုပြုချက် မှန်မမှန် ကိုယ်တိုင်ကိုယ်ကျ လေ့လာကြည့်ချင်ရင် Google မှာ perfect shuffle လို့ရိုက်ထည့်ပြီး ရှာကြည့်ပါ။

ပိုပြီးကောင်းမွန်တဲ့ ဖဲချိုးနည်း algorithm တစ်ခုကတော့ ဖဲထုပ်ကို တစ်ချိန်မှာ ဖဲတစ်ချပ် စီဖြတ်လျှောက်ပြီး အပြန်ပြန်အလှန်လှန် တစ်ကြိမ်လုပ်ချိန်မှာ ဖဲနှစ်ချပ်ကိုရွေးပြီး လဲလိုက်ဖို့ပါ ပဲ။ အောက်မှာပြထားတာကတော့ ဒီ algorithm ဘယ်လိုအလုပ်လုပ်သလဲဆိုတာရဲ့ မူကြမ်းပဲ ဖြစ်ပါတယ်။ ဒီပရိုဂရမ်ကို အကြမ်းရေးဖို့ Java ဖော်ပြချက်တွေနဲ့ အင်္ဂလိပ်စကားလုံးတွေကို

ပေါင်းရေးထားပါတယ်။ ဒါကိုတစ်ခါတလေ pseudocode လို့လည်း ခေါ်ကြပါတယ်။

```
for (int i = 0; i < deck.length; i++) {
    // choose a random number between i and deck.cards.length
    // swap the ith card and the randomly choosen card
}
```

Pseudocode တွေသုံးရာကရတဲ့ ကောင်းကျိုးတစ်ခုကတော့ ကိုယ်လိုအပ်လာမယ့် method အမျိုးအစားတွေကို ရှင်းလင်းကွဲပြားစေပါတယ်။ ဒီနေရာမှာ လိုအပ်တာက low နဲ့ high parameter တွေကြားက ကျပန်းကိန်းပြည့်တစ်ခုကို ရွေးချယ်ပေးမယ့် randomInt နဲ့ index နှစ်ခုကိုလက်ခံပြီး ညွှန်းထားတဲ့နေရာတွေက ဖဲချပ်တွေကို နေရာလဲပေးတဲ့ swapCards တို့ ဖြစ်ကြပါတယ်။

ပြီးခဲ့တဲ့ ကျပန်းနဲ့သက်ဆိုင်တဲ့ အပိုင်းတစ်ပိုင်းကိုကြည့်ပြီး randomInt ကို ဘယ်လိုရေးမလဲ ဆိုတာ မှန်းဆနိုင်ပါလိမ့်မယ်။ ဒါပေမယ့် အပိုင်းအခြားပြင်ပကို ရောက်ရှိနေတဲ့ index တွေကို ထုတ်လုပ်မိမယ့်အဖြစ် မရောက်အောင်တော့ သတိထားဖို့ လိုပါလိမ့်မယ်။

swapCards ရေးနည်းကိုလည်း ကိုယ့်ဘာသာကိုယ် မှန်းကြည့်နိုင်ပါတယ်။ နည်းနည်းတွေးရခက်မယ့်ကိစ္စကတော့ ဖဲချပ်တွေရဲ့ အညွှန်းတွေကိုပဲ လဲမလား၊ ဖဲချပ်တွေရဲ့ ပါဝင်မှုတွေကိုပဲ လဲထည့်မလားဆိုတာပဲ ဖြစ်ပါတယ်။ ဘယ်နည်းကိုရွေးသလဲဆိုတာ အရေးကြီးသလား။ ဘယ်နည်းက ပိုမြန်သလဲ။ ဒီ method တွေ အကောင်အထည်ဖော်ပုံကိုတော့ လေ့ကျင့်ခန်းတစ်ခု အနေနဲ့ ချန်ထားခဲ့လိုက်ပါမယ်။

**ဖဲစီခြင်း**

အခုအချိန်မှာ ဖဲထုပ်ကြီးကို ရှုပ်ယှက်ခတ်သွားအောင် လုပ်ပစ်လိုက်ပြီးပြီဖြစ်တဲ့အတွက် သူ့ကို ပုံမှန်အတိုင်းပြန်စီဖို့ နည်းတစ်နည်းလိုပါတယ်။ ရယ်စရာကောင်းတာက ဖဲချိုးတဲ့ algorithm နဲ့တူတဲ့ ဖဲစီနည်း algorithm တစ်ခုရှိပါတယ်။ ဒီ algorithm ကို ရွေးချယ်ပြီးစီတဲ့နည်းလို့ခေါ်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူ့အလုပ်လုပ်ပုံက array ကို အကြိမ်ကြိမ်ဖြတ်လျှောက်ပြီး ကျန်နေတဲ့ အငယ်ဆုံးဖဲချပ်ကို တစ်ကြိမ်စီမှာ ရွေးချယ်လို့ပါပဲ။

ပထမဆုံးပတ်တဲ့အချိန်မှာ အနိမ့်ဆုံးဖဲချပ်ကိုတွေ့ပြီး သူ့ကို 0 နေရာမှာရှိတဲ့ ဖဲချပ်နဲ့ နေရာလဲလိုက်ပါတယ်။ i နေရာမှာ အငယ်ဆုံးဖဲချပ်ကို i ရဲ့ညာဘက်မှာတွေ့ပြီး သူ့ကို i နေရာက ဖဲချပ်နဲ့ နေရာလဲလိုက်ပါတယ်။

အောက်မှာပြထားတာကတော့ ရွေးချယ်ပြီး ပြန်စီတဲ့နည်းအတွက် pseudocode ပဲ ဖြစ်ပါတယ်။

```
for (int i = 0; i < deck.length; i++) {
    // find the lowest card at or to the right of i
```

```
    // swap the ith card and the lowest card
  }
```

ဒီနေရာမှာလည်း pseudocode ဟာ အကူအညီပေးတဲ့ method တွေရဲ့ ဒီဇိုင်းရေးသားရာမှာ အထောက်အပံ့ပြုပါတယ်။ ဒီအခြေအနေမှာ swapCards ကို ပြန်သုံးနိုင်ပါတယ်။ ဒီတော့ method အသစ်တစ်ခုပဲ ထပ်လိုပါတယ်။ သူကတော့ ဖဲချပ်တွေရဲ့ array တစ်ခုနဲ့ သူစရှာသင့်တဲ့နေရာကို ကိုယ်စားပြုတဲ့ index တစ်ခုကို လက်ခံတဲ့ findLowestCard ပဲ ဖြစ်ပါတယ်။

ဒီနေရာမှာလည်း အကောင်အထည်ဖော်ပုံကို ကိုယ်တိုင်ရေးသားနိုင်ဖို့ ချန်ထားခဲ့လိုက်ပါမယ်။

**လက်အောက်ခံဖဲထုပ်များ**

ဖဲရိုက်တဲ့သူလက်ထဲက ဖဲတွေ ဒါမှမဟုတ် ဖဲထုပ်ရဲ့အခြားလက်အောက်ခံ အစုအဝေးတစ်ခုကို ဘယ်လိုကိုယ်စားပြုမလဲ။ ဖြစ်နိုင်ခြေတစ်ခုကတော့ Hand လို့ခေါ်တဲ့ class အသစ်တစ်ခုကို ဖန်တီးပြီး Deck ကို တိုးချဲ့ဖို့ပါပဲ။ အခုရှင်းပြမယ့် နောက်တစ်နည်းကတော့ ဖဲရိုက်သူရဲ့ ဖဲချပ်တွေကို ဖဲချပ် 52 ချပ်ထက် ပိုနည်းတဲ့ Deck ဝတ္ထုတစ်ခုနဲ့ ကိုယ်စားပြုဖို့ပါပဲ။

Deck တစ်ခုနဲ့ index အပိုင်းအခြားတစ်ခုကို လက်ခံပြီး ဖဲချပ်တွေရဲ့ သတ်မှတ်ထားတဲ့ လက်အောက်ခံအစုအဝေးတစ်ခုပါဝင်တဲ့ Deck အသစ်တစ်ခုကို ပြန်ပေးအောင် supdeck ဆိုတဲ့ method ကို ရေးကြပါမယ်။

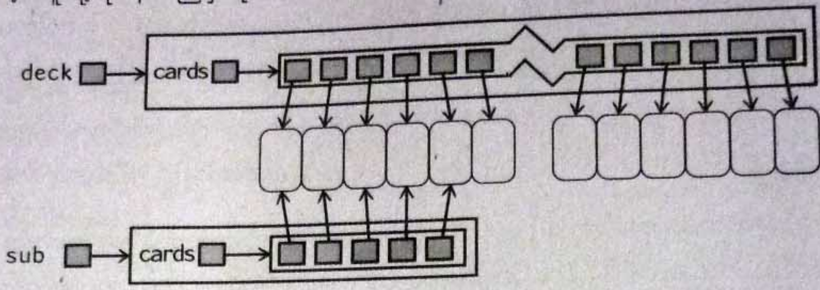
```
public static Deck subdeck(Deck deck, int low, int high) {
    Deck sub = new Deck(high - low + 1);

    for (int i = 0; i < sub.cards.length; i++) {
        sub.cards[i] = deck.cards[low + i];
    }
    return sub;
}
```

လက်အောက်ခံ ဖဲထုပ်ရဲ့အလျားဟာ high - low + 1 ဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတော့ အနိမ့်ဆုံးဖဲချပ်နဲ့ အမြင့်ဆုံးဖဲချပ် နှစ်ခုစလုံး ပါဝင်ရမှာကြောင့်ပါပဲ။ ဒီလိုတွက်ချက်နည်းမျိုးဟာ အနိမ့်ဆုံးဖဲချပ်နဲ့ အမြင့်ဆုံးဖဲချပ် နှစ်ခုစလုံး ပါဝင်ရမှာကြောင့်ပါပဲ။ ဒီလိုတွက်ချက်နည်းမျိုးဟာ ရှုပ်ထွေးနိုင်ပြီး တစ်ခုကွာနေတဲ့ အမှားမျိုးနဲ့ ကြုံတွေ့ရတတ်ပါတယ်။ ပုံဆွဲတာဟာ သူတို့ကို ရှောင်ဖို့ အကောင်းဆုံးနည်းပါပဲ။

new command နဲ့အတူ argument တစ်ခုကို ပံ့ပိုးတဲ့အတွက် နှိုးဆွခံရတဲ့ constructor ဟာ ပထမဆုံး constructor ဖြစ်ပါလိမ့်မယ်။ သူက array ကိုပဲ နေရာချထားပြီး ဖဲချပ်တွေကို လိုက်မဖြည့်ပါဘူး။ for သံသရာထဲမှာ လက်အောက်ခံဖဲထုပ်ဟာ ဖဲထုပ်ကြီးထဲက array အညွှန်း

တွေနဲ့ ဖြည့်ဆည်းခြင်း ခံရပါတယ်။

အောက်မှာပြထားတာကတော့ low = 3 နဲ့ high = 7 ဆိုတဲ့ parameter တွေကိုသုံးပြီး ဖန်တီးထားတဲ့ လက်အောက်ခံဖဲထုပ်တစ်ခုရဲ့ အခြေအနေပုံတစ်ခုပဲ ဖြစ်ပါတယ်။ ရလဒ်ကတော့ မူလဖဲထုပ်နဲ့ ကွန်ပျူတာမှတ်ဉာဏ်ပေါ်မှာ မျှဝေတည်ရှိနေတဲ့ ဖဲငါးချပ်ဖြစ်ပါတယ်။ အဓိပ္ပာယ်ကတော့ သူတို့ကို နာမည်နှစ်ခုပေးထားတယ်ဆိုတာနဲ့ အတူတူပါပဲ။




နာမည်နှစ်ခုပေးတာဟာ ပုံမှန်အားဖြင့် ကောင်းမွန်တဲ့အလေ့အထ မဟုတ်ဘူးလို့ ပြောခဲ့ပါတယ်။ လက်အောက်ခံဖဲထုပ်တစ်ခုရဲ့ အပြောင်းအလဲတွေဟာ ကျန်တဲ့ဖဲထုပ်တွေမှာလည်း ရောင်ပြန်ဟပ်သလို ဖြစ်ပေါ်တတ်ပါတယ်။ ဒါဟာ တကယ့်ဖဲချပ်တွေ ဖဲထုပ်တွေမှာဖြစ်ပေါ်ဖို့ မျှော်လင့်ရတဲ့ အပြုအမူမျိုးမဟုတ်ပါဘူး။ ဒါပေမယ့် ကိုင်တွယ်နေတဲ့ဝတ္ထုတွေဟာ ပြောင်းလဲလို့မရနိုင်ရင် နာမည်နှစ်ခုပေးတာဟာ ဥပါဒ်ဖြစ်နိုင်စွမ်း တော်တော်နည်းသွားပါတယ်။ အခုကိစ္စမှာ ဖဲချပ်တစ်ခုရဲ့ အဆင့်နဲ့အပွင့်ကို ပြောင်းလဲဖို့ အကြောင်းမရှိလောက်ပါဘူး။ ဒီအတွက် ဖဲချပ်တစ်ချပ်စီကို တစ်ကြိမ်ဖန်တီးပြီး သူ့ကို ပြောင်းလဲလို့မရနိုင်တဲ့ ဝတ္ထုအဖြစ် ဆက်ဆံသွားပါမယ်။ ဒီတော့ Card တွေအတွက် နာမည်နှစ်ခုပေးတာဟာ သဘာဝကျတဲ့ ရွေးချယ်မှုတစ်ခုပဲ ဖြစ်ပါတယ်။

## ဖဲချိုးခြင်းနှင့် ဖဲဝေခြင်း

ပြီးခဲ့တဲ့အပိုင်း အနည်းငယ်ကျော်မှာတုန်းက ဖဲချိုးတဲ့ algorithm ကို pseudocode ရေးခဲ့ပါတယ်။ ဖဲထုပ်တစ်ထုပ်ကို argument အဖြစ်လက်ခံပြီး ဖဲချိုးပေးတဲ့ shuffleDeck method တစ်ခု ရှိတယ်ဆိုရင် ဖဲထုပ်ကိုဖန်တီးပြီး ဖဲချိုးနိုင်ပါတယ်။

```
Deck deck = new Deck();
shuffleDeck(deck);
```

ပြီးရင် ဖဲသမားနှစ်ယောက်ဆီကို ဖဲဝေဖို့အတွက် subdeck ကိုသုံးနိုင်ပါတယ်။

```
Deck hand1 = subdeck(deck, 0, 4);
Deck hand2 = subdeck(deck, 5, 9);
Deck pack = subdeck(deck, 10, 51);
```

ဒီ code ဟာ ပထမဖဲငါးချပ်ကို ဖဲကစားသူတစ်ဦးလက်ထဲ ထည့်ပေးပြီး ကျန်တဲ့ငါးချပ်ကို နောက်တစ်ယောက်လက်ထဲ ထည့်ပေးလိုက်ပါတယ်။ ဝေပြီးကျန်တဲ့ဖဲတွေကို pack ထဲ ပုံထားလိုက်ပါတယ်။

ဖဲဝေတဲ့ကိစ္စကို တွေးကြည့်တဲ့အခါ ဝိုင်းဖွဲ့နေတဲ့အုပ်စုထဲက ဖဲသမားတစ်ယောက်စီအတွက် တစ်လှည့်စီ ဖဲတစ်ချပ်ဝေပေးဖို့ စဉ်းစားမိခဲ့သလား။ ဒီလိုစဉ်းစားမိခဲ့ပေမယ့် နောက်ပိုင်းမှာတော့ ကွန်ပျူတာပရိုဂရမ်တစ်ပုဒ်အတွက် ဒီလိုကိစ္စမျိုးကို မလိုအပ်ဘူးဆိုတာ သတိပြုမိလိုက်ရတယ်။ ဖဲကို တစ်ဝိုင်းလှည့်ပြီး ဝေတဲ့စနစ်ဟာ အပြစ်ကင်းစင်မှုမရှိတဲ့ဝေနည်းကို မမျှတမှုနည်းသွားအောင် ရည်ရွယ်ထားတာဖြစ်ပါတယ်။ ဒီလိုလုပ်လိုက်တဲ့အတွက် ဖဲဝေသူမှာလည်း ဖဲလိမ်နိုင်တဲ့ အခွင့်အလမ်းနည်းသွားပါတယ်။ ဒီပြဿနာနှစ်ခုစလုံးဟာ ကွန်ပျူတာတစ်လုံးအတွက်တော့ စကားထဲထည့်ပြောစရာတောင် မဟုတ်ပါဘူး။

ဒီဥပမာဟာ ကွန်ပျူတာအင်ဂျင်နီယာပညာရပ်မှာ ခိုင်းနှိုင်းမှုပြုလုပ်တာရဲ့ အန္တရာယ်တစ်ခုကို သတိပြုမိစေပါတယ်။ တစ်ခါတစ်ရံမှာ ကွန်ပျူတာတွေအပေါ် မလိုအပ်တဲ့အကန့်အသတ်တွေ ချမှတ်တတ်ပါတယ်။ တစ်ခါတစ်ရံ ကွန်ပျူတာတွေမှာမရှိတဲ့ အရည်အချင်းတွေကို မျှော်လင့်တတ်ပါတယ်။ ဘာလို့လဲဆိုတော့ ခိုင်းနှိုင်းမှုတစ်ခုကို အဓိပ္ပာယ်ရှိတဲ့ပမာဏထက် ကျော်လွန်ပြီး မဆင်မခြင်အသုံးချမိလို့ ဖြစ်ပါတယ်။ အဓိပ္ပာယ်မရှိတဲ့ ဥပမာယူပုံတွေကို သတိပြုပါ။

## ပေါင်းစည်းနည်းဖြင့်စီခြင်း

ဒီအခန်းရဲ့ အစောပိုင်း အပိုင်းတစ်ပိုင်းမှာတုန်းက သိပ်ပြီးအလုပ်မဖြစ်ဘူးလို့ သိခဲ့ရတဲ့ ဖဲစီနည်း algorithm ရိုးရိုးရှင်းရှင်းလေးတစ်ခုကို တွေ့ခဲ့ရပါတယ်။ ပါဝင်တဲ့အရေအတွက် n ရှိရင် array ကို n ကြိမ် ဖြတ်လျှောက်ရပါတယ်။ တစ်ကြိမ်ဖြတ်လျှောက်ရင် n ခုရှိတဲ့အတွက် တစ်ခုကြာချိန်ရဲ့ n ဆရှိတဲ့ အချိန်ကာလကို ယူပါလိမ့်မယ်။ n ကြိမ်ဖြတ်လျှောက်ရရင် စုစုပေါင်းကြာချိန်ဟာ n နှစ်ခုမြှောက်လဒ်နဲ့ အချိုးကျနေပါလိမ့်မယ်။

ဒီအပိုင်းမှာတော့ ပိုပြီးအလုပ်တွင်ကျယ်တဲ့ algorithm တစ်ခုဖြစ်တဲ့ mergesort ကို အကြမ်းဖျင်း တင်ပြသွားပါမယ်။ Mergesort ဟာ နှစ်ခြမ်းဖြစ်နေတဲ့ အစုအဝေးကို ပေါင်းတဲ့နည်းနဲ့ ပြန်စီတာဖြစ်ပါတယ်။ n အရေအတွက်ကိုပြန်စီဖို့ mergesort ဟာ n log n နဲ့ အချိုးညီတဲ့အချိန်ကို ယူပါလိမ့်မယ်။ ဒါဟာ သိပ်ပြီးအထင်ကြီးစရာမဟုတ်ဘူးလို့ ထင်စရာရှိပါတယ်။ ဒါပေမယ့် n ကြီးထွားလာတာနဲ့အမျှ $n^2$ နဲ့ n log n တို့ကြားက ခြားနားချက်ဟာ ပြောပရလောက်အောင်ကြီးမားလာပါတယ်။ n ရဲ့တန်ဖိုးအနည်းငယ်အတွက် စမ်းသပ်ကြည့်ပြီး ဘယ်လိုဖြစ်လာမလဲဆိုတာ စောင့်ကြည့်ပါ။

Mergesort ရဲ့နောက်ကွယ်က အခြေခံအယူအဆကတော့ ဒီလိုဖြစ်ပါတယ်။ စီပြီးသားဖြစ်နေတဲ့ ဖဲနှစ်ထုပ်ရှိရင် သူတို့ကို အစီအစဉ်တကျဖြစ်နေတဲ့ ဖဲတစ်ထုပ်တည်းအဖြစ် ပြန်ပေါင်းရတာ လွယ်ကူသလို မြန်လည်းမြန်ပါတယ်။ ဖဲတစ်ထုပ်ယူပြီး အောက်မှာပြထားတာတွေကို စမ်းလုပ်ကြည့်ပါ။

၁။ ဖဲချပ်ဆယ်ချပ်လောက်ပါတဲ့ ဖဲနှစ်ပုံကို မျက်နှာအပေါ်ဘက်လှည့်ထားရင် အငယ်ဆုံးဖဲက အပေါ်ဆုံးမှာရောက်အောင် စီပါ။ ဖဲနှစ်ပုံစလုံးကို ကိုယ့်ရှေ့မှာထားလိုက်ပါ။

၂။ ဖဲတစ်ပုံဆီက ထိပ်ဆုံးဖဲတွေကိုယူပြီး ပိုငယ်တဲ့ဖဲကို ရွေးပါ။ ပြောင်းပြန်လှန်ပြီး ပေါင်းမယ့်အပုံမှာ ပေါင်းထည့်လိုက်ပါ။

၃။ ဒုတိယအဆင့်ကို ဖဲတစ်ပုံမကုန်မချင်း ဆက်လုပ်ပါ။ ပြီးရင် ကျန်တဲ့ဖဲတွေကိုယူပြီး ပေါင်းထားတဲ့ဖဲထုပ်မှာ ထပ်လိုက်ပါ။

ရလဒ်ဟာ တစ်ခုတည်းဖြစ်နေတဲ့ စီပြီးသားဖဲတစ်ထုပ် ဖြစ်နေသင့်ပါတယ်။ အောက်မှာပြထားတာကတော့ pseudocode မှာ ဒါကို ဘယ်လိုမြင်ရမလဲ ဆိုတာပါပဲ။

```
public static Deck merge(Deck d1, Deck d2) {
    // create a new deck big enough for all the cards
    Deck result = new Deck(d1.cards.length + d2.cards.length);

    // use the index i to keep track of where we are in
    // the first deck, and the index j for the second deck
    int i = 0;
    int j = 0;

    // the index k traverses the result deck
    for (int k = 0; k < result.cards.length; k++) {

    // if d1 is empty d2 wins; if d2 is empty d1 wins;
    // otherwise, compare the two cards

    // add the winner to the new deck
    }
    return result;
}
```

အင်္ဂလိပ်စာ သိပ်မလေ့လာထားသူတွေအတွက်တော့ ဒါကိုနားလည်ရနည်းနည်းခက်နေပါလိမ့်မယ်။ သူလုပ်တာက ဖဲနှစ်ပုံစာဖဲတစ်ပုံကို ဖန်တီးပြီး ဖဲတစ်ထုပ်စီရဲ့ ရောက်နေတဲ့ဖဲချပ်နေရာကို နောက်ကြောင်းလိုက်ဖို့ i နဲ့ j ဆိုတဲ့ ကိန်းပြည့်တွေကို သုံးပါတယ်။ ပြီးရင် for သံသရာထဲမှာ ဖဲထုပ်တွေကုန်နေသလား အရင်စစ်ပါတယ်။ ကျန်တဲ့ဖဲထုပ်ကို ပေါင်းထားတဲ့ဖဲတွေနေရာမှာ သွားပေါင်းခိုင်းပါတယ်။ မဟုတ်ရင် ဖဲတွေကိုနှိုင်းယှဉ်ပြီး ပြေလည်စေတဲ့ဖဲကို ဖဲထုပ်

အသစ်မှာ ပေါင်းထည့်ခိုင်းပါတယ်။ ပရိုဂရမ်ရေးသူအတွက် အင်္ဂလိပ်စာနဲ့ ရင်းနှီးမှုကို မရှိမ
ဖြစ်ဆိုသလို လိုအပ်ပါတယ်။ တရားဝင်ထုတ်ပြန်ထားတဲ့ စာရွက်စာတမ်းတွေကို အင်္ဂလိပ်ဘာ
သာနဲ့ ရေးသားထားတာဖြစ်ပြီး ပရိုဂရမ်ရေးသားမှု ပတ်ဝန်းကျင်တွေရဲ့ စာရွက်စာတမ်းရေးသား
မှု ကိရိယာတွေဟာလည်း အင်္ဂလိပ်ဘာသာစကားကိုပဲ အသုံးပြုပါတယ်။ ဒီတော့ ကိုယ့်ရဲ့စာရွက်
စာတမ်းကိုလည်း အင်္ဂလိပ်လိုရေးရမှာ ဖြစ်ပါတယ်။

စကားပြန်ဆက်ရရင် merge ကိုစစ်ဆေးဖို့ အကောင်းဆုံးနည်းက ဖဲတစ်ထုပ်ကို တည်
ဆောက်ပြီး ဖဲချိုး၊ subdeck ကိုသုံးပြီး ဖဲအပုံသေးသေး နှစ်ပုံခွဲ၊ ပြီးရင် ပြီးရင်ခဲ့တဲ့အခန်းက စီ
တဲ့နည်းကိုသုံးပြီး နှစ်ခြမ်းကို ပြန်စီကြည့်ဖို့ပါပဲ။ ပြီးရင် သူတို့နှစ်ပိုင်းကို merge ဆီပေးပို့ပြီး
အလုပ်ဖြစ်မဖြစ် စစ်ဆေးနိုင်ပါတယ်။

ဒီလိုလုပ်ကြည့်လို့မှန်ရင် mergeSort ရဲ့ ရိုးရှင်းတဲ့အကောင်အထည်ဖော်ပုံကို စမ်းကြည့်
ပါ။

```
public static Deck mergeSort(Deck deck) {
    // find the midpoint of the deck
    // devide the deck into two subdecks
    // sort the subdecks using sortDeck
    // merge the two halves and return the result
}
```

အဆင့်တွေက ဖဲထုပ်ရဲ့အလယ်မှတ်ကိုရှာပြီး နှစ်ပိုင်းပိုင်းပါ။ sortDeck သုံးပြီး လက်
အောက်ခံဖဲထုပ်တွေကိုစီပါ။ တစ်ဝက်တွေကိုပေါင်းပြီး ရလဒ်ကိုပြန်ပါ။

ဒါကို ရအောင်လုပ်နိုင်ရင် တကယ်ပျော်စရာကောင်းတဲ့ အလုပ်ကို စတင်ရပါမယ်။
Mergesort ရဲ့ အံ့သြစရာကောင်းတဲ့အရာက သူဟာ အပြန်ပြန်အလှန်လှန်ဖြစ်ပါတယ်။ လက်
အောက်ခံဖဲထုပ်တွေကို ပြန်စီတဲ့နေရာမှာ ဘာလို့ဈေးကွေးတဲ့ sort ပုံစံဟောင်းကြီးကို နှိုးဆွနေ
တော့မှာလဲ။ ဘာဖြစ်လို့ ကိုယ်ရေးသားနေတဲ့ mergeSort အသစ်ကို နှိုးဆွလို့ မရနိုင်ရမှာလဲ။

ဒါဟာ ကောင်းမွန်တဲ့အရာ ဖြစ်ရုံသာမကဘဲ အရင်ကတိပေးခဲ့တဲ့ စွမ်းဆောင်ရည်အား
သာချက်ကိုရရှိဖို့ သူ့ကို မရှိမဖြစ်လိုပါတယ်။ ဒါကို အလုပ်ဖြစ်အောင်တော့ အဆုံးမရှိ အပြန်ပြန်
အလှန်လှန်ဖြစ်မနေအောင် အခြေခံအခြေအနေအတွက် ဖြေရှင်းချက်တစ်ခု ပေါင်းထည့်သင့်
ပါတယ်။ ရိုးရှင်းတဲ့ အခြေအနေဖြေရှင်းချက်ဟာ 0 ဒါမှမဟုတ် 1 အရေအတွက်ရှိတဲ့ ဖဲချပ်ပါတဲ့
လက်အောက်ခံဖဲထုပ်ဖြစ်ပါတယ်။ အကယ်၍ mergeSort ဟာ ဒီလိုသေးငယ်တဲ့ လက်အောက်
ခံဖဲထုပ်ကို လက်ခံရရင် သူ့ကိုနဂိုကတည်းက စီပြီးသားဖြစ်တဲ့အတွက် မပြုမပြင်ဘဲ ပြန်နိုင်ပါ
တယ်။

mergeSort ရဲ့ အပြန်ပြန်အလှန်လှန်ပုံစံဟာ အောက်မှာပြထားသလို ရှိပါတယ်။

```
public static Deck mergeSort(Deck deck) {
```

```
    // if the deck is 0 or 1 cards, return it

    // find the midpoint of the deck
    // devide the deck into two subdecks
    // sort the subdecks using sortDeck
    // merge the two halves and return the result
}
```

ပရိုဂရမ်တွေနဲ့ပတ်သက်ပြီး တွေးစရာနှစ်နည်း ရှိပါတယ်။
အရင်လိုပဲ အပြန်ပြန်အလှန်လှန် ပရိုဂရမ်တွေနဲ့ပတ်သက်ပြီး ခုန်ကျော်တဲ့နည်းဆိုပြီး ရှိပါတယ်။
Run တဲ့ အစီအစဉ်အတိုင်း တွေးတဲ့နည်းနဲ့ ယုံကြည်မှုသုံးပြီး ခုန်ကျော်တဲ့နည်းဆိုပြီး ရှိပါတယ်။ လက်ရှိပုံစံအတိုင်း
ဒီဥပမာမှာ ယုံကြည်မှုနဲ့ ခုန်ကျော်ရဲအောင် တမင်တကာ တည်ဆောက်ပုံကို လက်ရှိပုံစံအတိုင်း
တင်ပြထားပါတယ်။

sortDeck ကိုသုံးပြီး လက်အောက်ခံဖဲထုပ်တွေကို ပြန်စီနေတယ်ဆိုရင် ပရိုဂရမ် run
တဲ့လမ်းကြောင်းကို လိုက်ချင်တဲ့စိတ်တွေ ပေါ်မလာဘူးမဟုတ်လား။ sortDeck method ကို
အမှားပြင်ပြီးသား ဖြစ်တဲ့အတွက် အလုပ်ဖြစ်လိမ့်မယ်လို့ ယူဆခဲ့ပါတယ်။ mergeSort ကို
အပြန်ပြန်အလှန်လှန်ဖြစ်အောင် လုပ်ဖို့လိုသမျှဟာ ပြန်စီတဲ့ algorithm တစ်ခုကို နောက်တစ်ခုနဲ့
ပြန်လဲထည့်တာပဲ ဖြစ်ပါတယ်။ ပရိုဂရမ်ကို နောက်တစ်နည်းနဲ့ပြန်ဖတ်ဖို့ အကြောင်းမရှိပါဘူး။

တကယ်တမ်းကျတော့ အခြေခံအခြေအနေကို မှန်မှန်ကန်ကန်ရအောင် နည်းနည်းတွေး
ဖို့တော့ လိုခဲ့ပါတယ်။ တဖြည်းဖြည်းနဲ့ ဒီဘူတာပဲဆိုက်ရမယ်လို့ ယုံကြည်ပြီး လုပ်ခဲ့ပါတယ်။
အဲဒါကလွဲရင် အပြန်ပြန်အလှန်လှန်ဖြစ်တတ်ပုံကို ပြန်ရေးရတာဟာ ပြဿနာမရှိပါဘူး။ ကံ
ကောင်းပါစေ။

အခန်း ၁၃

# ဝတ္ထုဦးတည်သော ပရိုဂရမ်းမင်း

**ပရိုဂရမ်းမင်းဘာသာစကားများနှင့် ရေးဟန်များ**

ကမ္ဘာပေါ်မှာ ပရိုဂရမ်းမင်းဘာသာစကား အမြောက်အများရှိပါတယ်။ အဲဒီလောက် အရေအတွက်နီးပါး ပရိုဂရမ်ရေးဟန်တွေ ရှိပါတယ်။ သူတို့ကို paradigm တွေလို့လည်း ခေါ်ပါတယ်။ ဒီစာအုပ်မှာတွေ့ခဲ့ရတဲ့ ရေးဟန်သုံးမျိုးကတော့ ဖြစ်စဉ်ရေးသားနည်း၊ လုပ်ဆောင်ချက်ကိုရည်ရွယ်တဲ့ ရေးသားနည်းနဲ့ ဝတ္ထုကိုဦးတည်တဲ့ ရေးသားနည်းတို့ ဖြစ်ကြတယ်။ Java ကို အများအားဖြင့် ဝတ္ထုကိုဦးတည်တဲ့ ဘာသာစကားတစ်ခုအဖြစ် ပြောဆိုတတ်ကြပေမယ့် Java ပရိုဂရမ်တွေကို ဘယ်ရေးဟန်နဲ့မဆို ရေးသားနိုင်ပါတယ်။ ဒီစာအုပ်မှာ သရုပ်ဖော်ခဲ့တဲ့ ရေးဟန်ဟာ ဖြစ်စဉ်တော်တော်ဆန်ပါတယ်။ ပြင်ပမှာတည်ရှိနေတဲ့ Java ပရိုဂရမ်တွေနဲ့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ Java package တွေကို ရေးဟန်သုံးမျိုးစလုံးရောပြီး ရေးသားထားပါတယ်။ ဒါပေမယ့် သူတို့
ဟာ ဒီစာအုပ်က ပရိုဂရမ်တွေထက်ပိုပြီး ဝတ္ထုဦးတည်တဲ့ဘက် ရောက်ပါတယ်။

ဝတ္ထုကိုဦးတည်တဲ့ ပရိုဂရမ်းမင်းဆိုတာဘာလဲလို့ အဓိပ္ပာယ်သတ်မှတ်ရ မလွယ်ပါဘူး။ ဒါပေမယ့် အောက်မှာပြောထားတာတွေကတော့ သူ့ရဲ့ ဝိသေသလက္ခဏာတွေပဲ ဖြစ်ပါတယ်။

- Class တွေလို့ခေါ်ကြတဲ့ ဝတ္ထုအဓိပ္ပာယ်သတ်မှတ်ချက်တွေဟာ လက်တွေ့ဘဝက သင့်

တော်တဲ့ ဝတ္ထုတွေကို ကိုယ်စားပြုပါတယ်။ ဥပမာ ပြီးခဲ့တဲ့အခန်းက Deck class ကို ရေးသားတာဟာ ဝတ္ထုဦးတည်တဲ့ ပရိုဂရမ်းမင်းဆီ ခြေလှမ်းတစ်လှမ်း တိုးလိုက်တာပါပဲ။

- Method တော်တော်များများဟာ ဝတ္ထု method တွေ ဖြစ်ပါတယ်။ ဝတ္ထု method ဆိုတာ ဝတ္ထုတစ်ခုပေါ်မှာ နှိုးဆွတဲ့ method တွေဖြစ်ပါတယ်။ သူတို့ကို class method တွေထက် ပိုအသုံးများပါတယ်။ Class method တွေဆိုတာ class ပေါ်မှာနှိုးဆွပြီး အဲဒီ class ရဲ့ သီးသန့်ဝတ္ထုတွေနဲ့ မသက်ဆိုင်တဲ့ method ဖြစ်ပါတယ်။ အခုအချိန်ထိ ရေးဖူးခဲ့သမျှ method တွေဟာ class တွေဖြစ်ခဲ့ပါတယ်။ ဒီအခန်းက ဝတ္ထု method တချို့ကို စတင်ရေးသားပါမယ်။
- ဝတ္ထုကိုဦးတည်တဲ့ ပရိုဂရမ်းမင်းနဲ့သက်ဆိုင်တဲ့ ဘာသာစကားလက္ခဏာ တစ်ခုကတော့ inheritance လို့ခေါ်တဲ့ အမွေဆက်ခံတာ ဖြစ်ပါတယ်။ အမွေဆက်ခံတဲ့ အကြောင်းကို ဒီအခန်းနောက်ပိုင်းမှာ ရှင်းပြပါမယ်။

မကြာသေးမီကာလအတွင်းမှာ ဝတ္ထုဦးတည်တဲ့ ပရိုဂရမ်းမင်းဟာ တော်တော်လေး ရေပန်းစားလာခဲ့ပါတယ်။ ပြီးတော့ သူ့ကို အခြားရေးဟန်တွေထက် နေရာအမျိုးမျိုးမှာ ပိုပြီးသာတယ်လို့ ပြောကြတဲ့သူတွေလည်း ရှိပါတယ်။ ပရိုဂရမ်ရေးဟန်အမျိုးမျိုးနဲ့ ထိတွေ့ပေးခြင်းအားဖြင့် ဒီလိုပြောဆိုတာတွေကို နားလည်ပြီး ကိုယ့်ဘာသာကိုယ် စဉ်းစားဆင်ခြင်ကြည့်ကြဖို့ လိုအပ်တဲ့အယူအဆတွေကို ရရှိပြီးပြီလို့ ယုံကြည်ပါတယ်။

## ဝတ္ထုနှင့် Class Method များ

Java မှာ method နှစ်မျိုးနှစ်စားရှိပါတယ်။ သူတို့ကို class method တွေနဲ့ ဝတ္ထု method တွေဆိုပြီး ခေါ်ပါတယ်။ အခုအချိန်အထိတော့ ရေးဖူးခဲ့သမျှ method တွေအားလုံးဟာ class method တွေချည်း ဖြစ်ခဲ့ပါတယ်။ Class method ပထမဆုံးစာကြောင်းမှာ `static` ဆိုတဲ့ keyword နဲ့ အမှတ်အသားပြုပါတယ်။ `static` keyword မပါတဲ့ method တိုင်းဟာ ဝတ္ထု method တွေ ဖြစ်ပါတယ်။

ဝတ္ထု method တွေကို မရေးသားဘူးပေမယ့် နှိုးဆွဖူးတာတွေတော့ ရှိပါတယ်။ ဝတ္ထုတစ်ခုအပေါ်မှာ method တစ်ခုကို နှိုးဆွတဲ့အခါတိုင်း သူဟာ ဝတ္ထု method တစ်ခုဖြစ်ပါတယ်။ ဥပမာ `String` တွေပေါ်မှာနှိုးဆွခဲ့တဲ့ `charAt` နဲ့ တခြား method တွေအားလုံးဟာ ဝတ္ထု method တွေ ဖြစ်ပါတယ်။

Class method တစ်ခုအနေနဲ့ ရေးသားနိုင်သမျှကို ဝတ္ထု method တစ်ခုအနေနဲ့ ရေးသားနိုင်သလို ဝတ္ထု method တစ်ခုအနေနဲ့ ရေးသားထားတဲ့ method တိုင်းကိုလည်း class method အနေနဲ့ ပြန်ပြီးရေးနိုင်ပါတယ်။ တစ်ခါတစ်ရံမှာ ဘယ် method ကိုသုံးသလဲဆိုတာ လုပ်လေ့လုပ်ထရှိတဲ့ အကျင့်ပေါ်မှာမူတည်ပါတယ်။ ရှေ့ပိုင်းကျရင် ရှင်းလင်းသွားမယ့် အကြောင်းတွေကြောင့် ဝတ္ထု method တွေဟာ သူတို့နဲ့သက်ဆိုင်တဲ့ class method တွေထက် ပိုပြီးတိုတတ်ပါတယ်။

**လက်ရှိဝတ္ထု**

ဝတ္ထုတစ်ခုပေါ်မှာ method တစ်ခုကို နှိုးဆွတဲ့အခါ အဲဒီဝတ္ထုဟာ လက်ရှိဝတ္ထုဖြစ်သွားပါတယ်။ Method ထဲမှာ လက်ရှိဝတ္ထုရဲ့ instance variable တွေကို နာမည်ခေါ်ပြီး ညွှန်းဆိုနိုင်ပါတယ်။ ဝတ္ထုနာမည်ကို ထည့်ပြောစရာမလိုပါဘူး။

ပြီးတော့ လက်ရှိဝတ္ထုကို this keyword သုံးပြီးလည်း ညွှန်းဆိုနိုင်ပါတယ်။ this ကို constructor တွေမှာသုံးတာကို မြင်ခဲ့ပြီးပါပြီ။ တကယ်တမ်းကျတော့ constructor တွေကို အထူးဝတ္ထု method အမျိုးအစားတွေလို့ တွေးကြည့်နိုင်ပါတယ်။

## Complex ကိန်းများ

အထက်တန်းသင်္ချာမှာတုန်းက အနုတ်ကိန်းတစ်ခုကို နှစ်ထပ်ကိန်းရင်းတင်လို့ မရဘူးလို့ သင်ခဲ့ရပါတယ်။ ဒါဟာ ကိန်းစစ်စနစ်မှာတော့ မှန်ပါတယ်။ အလယ်တန်းမှာတုန်းက ကိန်းတွေကို အပြည့်ကိန်းကနေ ကိန်းပြည့်၊ ကိန်းပြည့်ကနေ ရာရှင်နယ်ကိန်း၊ ရာရှင်နယ်ကိန်းကနေ ကိန်းစစ်ဆိုပြီး ကိန်းစနစ်တစ်ခုကို သူ့မှာမပါတဲ့ ကိန်းအမျိုးအစားတွေ ပေါင်းထည့်ခြင်းအားဖြင့် တိုးချဲ့ခဲ့ပါတယ်။ အခုလည်း ကိန်းစစ်တွေကို ထပ်တိုးချဲ့လို့ မရတော့ဘူးလား။ ကိန်းစစ်စနစ်မှာ မပါတဲ့ကိန်းတွေဟာ ဘာတွေလဲ။

-4 ရဲ့ နှစ်ထပ်ကိန်းရင်းကိုရှာရင် ဘယ်လောက်ရမလဲ။ 4 ရဲ့နှစ်ထပ်ကိန်းရင်းဟာ 2 ဖြစ်ပါတယ်။ -4 ဆိုတာ 4 နဲ့ -1 ကို မြှောက်ထားတာဖြစ်ပါတယ်။ -4 ကိုနှစ်ထပ်ကိန်းရင်းတင်ရင် 2 နဲ့ -1 ရဲ့ နှစ်ထပ်ကိန်းရင်းတန်ဖိုး မြှောက်ထားတာကို ရပါလိမ့်မယ်။ ကိန်းစစ်တွေရဲ့ ပြင်ပက complex ကိန်းစနစ်မှာ -1 ရဲ့နှစ်ထပ်ကိန်းရင်းကို i နဲ့ သတ်မှတ်ပါတယ်။

ဒီအခန်းမှာ complex ကိန်းတွေအတွက် class အဓိပ္ပာယ်သတ်မှတ်ချက်တစ်ခုကို ဥပမာအဖြစ် စဉ်းစားမှာဖြစ်ပါတယ်။ Complex ကိန်းတွေဟာ သင်္ချာနဲ့အင်ဂျင်နီယာဘာသာကွဲ တော်တော်များများအတွက် အရေးပါသလို တွက်ချက်မှုတော်တော်များများကိုလည်း complex ကိန်းတွေပါတဲ့ ဂဏန်းသင်္ချာသုံးပြီး ဆောင်ရွက်ရပါတယ်။ Complex တစ်ခုကို ကိန်းစစ်တစ်ခုနဲ့ စိတ်ကူးယဉ်ကိန်းတို့ ပေါင်းလဒ်အနေနဲ့ ရေးသားပါတယ်။ သင်္ချာပုံစံနဲ့ပြောရရင် x + yi ပုံစံရှိပါတယ်။ x ဟာ ကိန်းစစ်ဖြစ်ပြီး yi ဟာ စိတ်ကူးယဉ်ကိန်း ဖြစ်ပါတယ်။ i ကို i နဲ့ ပြန်မြှောက်ရင် အနုတ်တစ်ရပါတယ်။

အောက်မှာပြထားတာကတော့ Complex လို့ခေါ်တဲ့ ဝတ္ထုအမျိုးအစားသစ် တစ်ခုအတွက် class အဓိပ္ပာယ်သတ်မှတ်ချက်တစ်ခု ဖြစ်ပါတယ်။

```
class Complex {
    // instance variables
    double real, imag;
```

```
    // constructor
    public Complex() {
       this.real = 0.0;
       this.imag = 0.0;
    }

    // constructor
    public Complex(double real, double imag) {
       this.real = real;
       this.imag = imag;
    }
}
```

ဒီနေရာမှာ ဘာမှ အံ့သြတကြီးဖြစ်စရာမရှိပါဘူး။ Instance variable တွေဟာ ကိန်းစစ်နဲ့ စိတ်ကူးယဉ်အပိုင်းတွေပါဝင်တဲ့ double တွေဖြစ်ပါတယ်။ Constructor နှစ်ခုဟာ ပုံမှန်အတိုင်း ဖြစ်ပါတယ်။ Parameter တွေကိုလက်မခံဘဲ instance variable တွေမှာ မူလတန်ဖိုးတွေနေရာချထားတဲ့ constructor တစ်ခုနဲ့ instance variable တွေနဲ့တူတဲ့ parameter တွေကိုလက်ခံတဲ့ constructor တစ်ခုတို့ ဖြစ်ပါတယ်။ အရင်ကတွေ့ခဲ့သလိုပဲ စတင်ခံရတဲ့ဝတ္ထုကို ညွှန်းဆိုဖို့ this keyword ကို သုံးပါတယ်။

main ဒါမှမဟုတ် တခြားတစ်နေရာမှာ complex ဝတ္ထုတွေကို တည်ဆောက်ချင်ရင် ရွေးစရာနှစ်လမ်းရှိပါတယ်။ ဝတ္ထုကိုဖန်တီးပြီးမှ instance variable တွေကို သတ်မှတ်တဲ့လမ်းနဲ့ တစ်ချိန်တည်းမှာ နှစ်ခုစလုံး တစ်ပြိုင်တည်းလုပ်တဲ့လမ်းတို့ ဖြစ်ပါတယ်။

```
Complex x = new Complex();
x.real = 1.0;
x.imag = 2.0;
Complex y = new Complex(3.0, 4.0);
```

## Complex ကိန်းများအတွက် လုပ်ဆောင်ချက်တစ်ခု

Complex ကိန်းတွေပေါ်မှာ ဆောင်ရွက်လိုမယ့် လုပ်ဆောင်ချက်တချို့ကို လေ့လာရအောင်။ Complex ကိန်းတစ်ခုရဲ့ ပကတိတန်ဖိုးဟာ $x^2 + y^2$ ရဲ့ နှစ်ထပ်ကိန်းရင်းဖြစ်ပါတယ်။ abs method ဟာ ပကတိတန်ဖိုးကိုတွက်တဲ့ သန့်စင်တဲ့ function တစ်ခုဖြစ်ပါတယ်။ Class method တစ်ခုအနေနဲ့ ရေးတဲ့အခါ အောက်မှာပြထားသလို ရှိပါတယ်။

```
// class method
```

```
public static double abs(Complex c) {
    return Math.sqrt(c.real * c.real + c.imag * c.imag);
}
```

abs ရဲ့ ဒီပုံစံဟာ parameter အဖြစ်လက်ခံရရှိတဲ့ c ဆိုတဲ့ Complex ဝတ္ထုရဲ့ ပကတိတန်ဖိုးကို တွက်ပါတယ်။ အောက်မှာပြန်ပြင်ရေးထားတဲ့ abs ဟာ ဝတ္ထု method တစ်ခုဖြစ်ပါတယ်။ သူက method ကိုနှိုးဆွခံရတဲ့ လက်ရှိဝတ္ထုရဲ့ ပကတိတန်ဖိုးကို တွက်ပါတယ်။ ဒီတော့ ဘယ် parameter ကိုမှ လက်မခံပါဘူး။

```
// object method
public double abs() {
    return Math.sqrt(real * real + imag * imag);
}
```

ဒါဟာဝတ္ထု method တစ်ခုဖြစ်တယ်ဆိုတာကို ပြောဖို့ static keyword ကို ဖြုတ်လိုက်ပါတယ်။ မလိုအပ်တဲ့ parameter ကိုလည်း ဖျောက်လိုက်ပါတယ်။ Method ထဲမှာ real နဲ့ imag ဆိုတဲ့ instance variable နှစ်ခုကို ဘယ်ဝတ္ထုကလည်းလို့ ညွှန်းစရာမလိုဘဲ နာမည်ခေါ်ပြီး သုံးလိုက်ပါတယ်။ လက်ရှိဝတ္ထုရဲ့ instance variable တွေကို ညွှန်းဆိုနေတယ်ဆိုတာ Java က တိတ်တဆိတ် သိလိုက်ပါတယ်။ ဒါကို သိသိသာသာဖြစ်သွားစေချင်ရင် this keyword ကို သုံးလို့ရပါတယ်။

```
// object method
public double abs() {
    return Math.sqrt(this.real * this.real + this.imag * this.imag);
}
```

ဒါပေမယ့် ဒါဟာ ပိုပြီးရှည်လာသလို ထူးထူးခြားခြားလည်း ရှင်းမသွားပါဘူး။ ဒီ method ကိုနှိုးဆွဖို့ ဝတ္ထုတစ်ခုပေါ်မှာ ပြုလုပ်ရပါမယ်။ ဥပမာ

```
Complex y = new Complex(3.0, 4.0);
double result = y.abs();
```

## Complex ကိန်းများအတွက် လုပ်ဆောင်ချက်နောက်တစ်ခု

Complex ကိန်းတွေပေါ်မှာ ဆောင်ရွက်လိုမယ့် နောက်ထပ် လုပ်ဆောင်ချက်တစ်ခုကတော့ Complex ကိန်းတွေပေါ်မှာ ဆောင်ရွက်လိုမယ့် နောက်ထပ် လုပ်ဆောင်ချက်တစ်ခုကတော့ ပေါင်းခြင်းပဲ ဖြစ်ပါတယ်။ ကိန်းစစ်အပိုင်းတွေအချင်းချင်းနဲ့ စိတ်ကူးယဉ်အပိုင်းတွေအချင်းချင်း ပေါင်းခြင်းပဲ ဖြစ်ပါတယ်။ ကိန်းစစ်အပိုင်းတွေအချင်းချင်းနဲ့ စိတ်ကူးယဉ်အပိုင်းတွေအချင်းချင်း ပေါင်းပြီး complex ကိန်းတွေကို ပေါင်းနိုင်ပါတယ်။ Class method တစ်ခုအနေနဲ့ ရေးတဲ့အခါ ဒါဟာ အောက်မှာပြထားသလို ပုံရှိပါတယ်။

```
public static Complex add(Complex a, Complex b) {
```

```
        return new Complex(a.real + b.real, a.imag + b.imag);
    }
```

ဒီ method ကိုနှိုးဆွဖို့ လုပ်ဆောင်ချက် ဆောင်ရွက်ခံရမယ့် operand နှစ်ခုစလုံးကို argument တွေအနေနဲ့ ပေးပို့ရပါတယ်။

```
    Complex sum = add(x, y);
```

ဝတ္ထု method တစ်ခုအနေနဲ့ ရေးတဲ့အခါ သူဟာ argument တစ်ခုတည်းကိုပဲ လက်ခံပါတယ်။ အဲဒီ argument ကို လက်ရှိဝတ္ထုမှာ ပေါင်းထည့်ပါတယ်။

```
public Complex add(Complex b) {
    return new Complex(real + b.real, imag + b.imag);
}
```

ဒီနေရာမှာလည်း လက်ရှိဝတ္ထုရဲ့ instance variable တွေကို သိုသိုသိပ်သိပ် ညွှန်းဆိုနိုင်ပါတယ်။ ဒါပေမယ့် b ရဲ့ instance variable တွေကို ညွှန်းဆိုတဲ့အခါ b ဆိုတဲ့ နာမည်ကို အစက်သင်္ကေတသုံးပြီး သိသိသာသာ ညွှန်းဆိုရပါတယ်။ ဒီ method ကိုနှိုးဆွဖို့ သူ့ကို operand တစ်ခုပေါ်မှာ ခေါ်ယူပြီး နောက်ထပ် operand တစ်ခုကို argument အနေနဲ့ ပေးပို့ရပါတယ်။

```
    Complex sum = x.add(y);
```

ဒီဥပမာတွေကနေပြီး this ဆိုတဲ့လက်ရှိဝတ္ထုဟာ parameter တစ်ခုရဲ့နေရာကို ယူနိုင်တယ်ဆိုတာ မြင်နိုင်ပါတယ်။ ဒီအကြောင်းကြောင့် လက်ရှိဝတ္ထုကို တစ်ခါတစ်ရံ သိုသိုသိပ်သိပ် parameter လို့ ခေါ်ကြပါတယ်။

## ပြုပြင်ပေးသော Method တစ်ခု

နောက်ထပ်ဥပမာတစ်ခုအနေနဲ့ conjugate ကို ကြည့်ကြပါမယ်။ သူက Complex ကိန်းတစ်ခုကို သူရဲ့ Complex conjugate အဖြစ် ပြောင်းလဲပေးတဲ့ method တစ်ခု ဖြစ်ပါတယ်။ x + yi ရဲ့ complex conjugate ဟာ x - y ဖြစ်ပါတယ်။

Class method တစ်ခုအနေနဲ့ သူဟာ အောက်မှာပြထားသလို ပုံစံရှိပါတယ်။

```
public static void conjugate(Complex c) {
    c.imag = -c.imag;
}
```

ဝတ္ထု method တစ်ခုအနေနဲ့ သူဟာ ဒီလိုပုံစံရှိပါတယ်။

```
public void conjugate() {
    imag = -imag;
}
```

အခုလောက်ဆိုရင် method တစ်ခုကို အမျိုးအစားပြောင်းတာဟာ စက်ရုပ်ဆန်တဲ့ ဖြစ်စဉ်

တစ်ခုဖြစ်တယ်လို့ မြင်နေလောက်ပါပြီ။ အလေ့အကျင့်နည်းနည်းရသွားရင် ဒါကို သိပ်တွေးစရာ မလိုဘဲ လုပ်တတ်လာပါလိမ့်မယ်။ ဒါဟာ ကောင်းပါတယ်။ ဘာလို့လဲဆိုတော့ method ရေးနည်း တစ်ခုနဲ့ ပိတ်မိနေတာဟာ ကောင်းမွန်တဲ့ ပရိုဂရမ်ရေးနည်းအလေ့အထ မဟုတ်လို့ပါပဲ။ ယင်းနှစ် မျိုးစလုံးနဲ့ ညီတူညီမျှ အကျွမ်းဝင်နေသင့်ပါတယ်။ ဒါမှပဲ ကိုယ်ရေးချင်တဲ့ လုပ်ဆောင်ချက်အတွက် သင့်တော်တဲ့ ရွေးချယ်မှုကို ပြုလုပ်နိုင်မှာပါ။

ဥပမာ add ကို class method တစ်ခုအနေနဲ့ ရေးသင့်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူဟာ operand နှစ်ခုရဲ့ ခေါက်ချိုးညီတဲ့ လုပ်ဆောင်ချက်ဖြစ်နေလို့ပါပဲ။ ဒီတော့ operand နှစ်ခုစလုံး ကို parameter တွေအနေနဲ့ ပေါ်ပေါက်စေတာဟာ အဓိပ္ပာယ်ရှိပါတယ်။ Operand တစ်ခုအ ပေါ်မှာ method ကိုနှိုးဆွပြီး ကျန်တဲ့ operand ကို argument အနေနဲ့ ပေးပို့တာဟာ နည်းနည်း ထူးဆန်းနေပါတယ်။

အခြားတစ်ဖက်မှာတော့ ဝတ္ထုတစ်ခုတည်းအပေါ်မှာသာ သက်ရောက်စေတဲ့ လုပ်ဆောင် ချက် ရိုးရိုးရှင်းရှင်းလေးတွေကို သူတို့ဟာ ထပ်ဆောင်း argument တွေ လက်ခံရင်တောင်မှ ဝတ္ထု method တွေအဖြစ် တိုတိုတုတ်တုတ် ရေးသားနိုင်ပါတယ်။

## toString Method

ဝတ္ထုတိုင်းမှာ ဝတ္ထုရဲ့ စာသားကိုယ်စားပြုချက်ကို ထုတ်လုပ်တဲ့ toString method ကို ပိုင်ဆိုင်ပါတယ်။ ဝတ္ထုတစ်ခုကို print ဒါမှမဟုတ် println ကိုသုံးပြီး print လုပ်တဲ့အခါ Java ဝတ္ထုရဲ့ toString method ကို နှိုးဆွလိုက်ပါတယ်။ toString ရဲ့ မူလပုံစံဟာ ဝတ္ထုအ မျိုးအစားနှင့် ဝတ္ထုမှတ်ဉာဏ်နေရာကို ကိုယ်စားပြုတဲ့ နံပါတ်တစ်ခုပါဝင်တဲ့ စာသားတစ်ခုကို ပြန်ပါတယ်။ ဝတ္ထုအမျိုးအစား အသစ်တစ်ခုကို သတ်မှတ်တဲ့အခါ မူလအပြုအမူကို ကိုယ်လိုချင် တဲ့ အပြုအမူရှိတဲ့ method အသစ်တစ်ခု ပံ့ပိုးပေးပြီး ဖျက်ရေးနိုင်ပါတယ်။

အောက်မှာပြထားတာကတော့ Complex class အတွက် toString method တစ်ခုပါ။

```
public String toString() {
    return real + " + " + imag + "i";
}
```

toString ရဲ့ ပြန်တဲ့တန်ဖိုးဟာ သဘာဝကျစွာပဲ String ဖြစ်ပါတယ်။ သူဟာ parameter လက်မခံပါဘူး။ toString ကို ပုံမှန်နည်းလမ်းနဲ့ နှိုးဆွနိုင်ပါတယ်။

```
Complex x = new Complex(1.0, 2.0);
String s = x.toString();
```

ဒါမှမဟုတ် သူ့ကို println တစ်ဆင့် သွယ်ဝိုက်ခေါ်ယူနိုင်ပါတယ်။

```
System.out.println(x);
```

ဒီနေရာမှာ output ဟာ 1.0 + 2.0i ဖြစ်ပါတယ်။

စိတ်ကူးယဉ်အပိုင်းက အနုတ်ဖြစ်ရင် ဒီ toString ဟာ ကြည့်မကောင်းပါဘူး။ လေ့ကျင့်ခန်းအနေနဲ့ ကြည့်ကောင်းတဲ့ပုံစံကို ပြန်ရေးပါ။

## equals Method

== လက္ခဏာကို ဝတ္ထုနှစ်ခုနှိုင်းယှဉ်ဖို့ သုံးတဲ့အခါ ကိုယ်တကယ်မေးနေတဲ့ မေးခွန်းက ဒီအရာနှစ်ခုဟာ ဝတ္ထုတစ်ခုတည်း ဟုတ်ရဲ့လားဆိုပြီး ဖြစ်ပါတယ်။ ဆိုလိုတာကတော့ ဝတ္ထုနှစ်ခုစလုံးဟာ မှတ်ဉာဏ်ထဲမှာ တူညီတဲ့နေရာတစ်ခုတည်းကို ညွှန်းဆိုရဲ့လားဆိုတာ ဖြစ်ပါတယ်။

အမျိုးအစားတော်တော်များများအတွက် ဒါဟာ လျော်ကန်တဲ့ တူညီမှုအဓိပ္ပာယ်သတ်မှတ်ချက် မဟုတ်ပါဘူး။ ဥပမာ ကိန်းစစ်အပိုင်းတွေနဲ့ စိတ်ကူးယဉ်အပိုင်းတွေ တူညီကြရင် complex ကိန်းနှစ်ခုဟာ တူညီကြပါတယ်။ သူတို့ဟာ ဝတ္ထုတစ်ခုတည်း ဖြစ်စရာမလိုပါဘူး။

အမျိုးအစားသစ်တစ်ခုကို သတ်မှတ်တဲ့အခါ equals လို့ခေါ်တဲ့ ဝတ္ထု method တစ်ခုကို ပံ့ပိုးပေးပြီး ကိုယ့်ကိုယ်ပိုင် တူညီမှုအဓိပ္ပာယ်ကို သတ်မှတ်ပေးနိုင်ပါတယ်။ Complex class အတွက် ဒါဟာ အောက်မှာပြထားသလို ပုံရှိပါတယ်။

```
public boolean equals(Complex b) {
    return (real == b.real && imag == b.imag);
}
```

ထုံးတမ်းစဉ်လာအရ equals ဟာ boolean တစ်ခုကိုပြန်တဲ့ ဝတ္ထု method တစ်ခုဖြစ်ပါတယ်။

Object class ထဲက equals ရဲ့ စာရွက်စာတမ်းဟာ ကိုယ်ပိုင်တူညီမှု အဓိပ္ပာယ်သတ်မှတ်ချက်ကိုရေးသားရင် ခေါင်းထဲထည့်ထားသင့်တဲ့ လမ်းညွှန်ချက်တချို့ကို ပံ့ပိုးပေးပါတယ်။

equals method ဟာ ညီမျှချက် ဆက်သွယ်ချက်တစ်ခုကို အကောင်အထည်ဖော်ပါတယ်။

- သူဟာ ရောင်ပြန်ဟပ်ပါတယ်။ x ရဲ့ ဘယ်အညွှန်းတန်ဖိုးအတွက်မဆို x.equals(x) ဟာ true ပြန်သင့်ပါတယ်။
- သူဟာ ခေါက်ချိုးညီပါတယ်။ x, y နဲ့ z ဆိုတဲ့ အညွှန်းတန်ဖိုးတိုင်းအတွက် x.equals(y) ဟာ y.equals(x) က true ပြန်မှ သူလည်း true ပြန်သင့်ပါတယ်။
- သူဟာ ဖက်စပ်ရပါတယ်။ x, y နဲ့ z ဆိုတဲ့ အညွှန်းတန်ဖိုးတိုင်းအတွက် x.equals(y) ဟာ true ပြန်ပြီး y.equals(z) ဟာ true ပြန်တယ်ဆိုရင် x.equals(z) ဟာလည်း true ပြန်သင့်ပါတယ်။
- သူဟာ သမာသမတ်ကျပါတယ်။ x နဲ့ y ဆိုတဲ့ အညွှန်းတန်ဖိုးတိုင်းအတွက် x.equals(y) ကို အကြိမ်ကြိမ်ခေါ်ဆိုတဲ့အခါ true ဒါမှမဟုတ် false ကို တသမတ်တည်း ထုတ်ပေးပါတယ်။
- x ရဲ့အညွှန်းတန်ဖိုးတိုင်းအတွက် x.equals(null) ဟာ false ပြန်သင့်ပါတယ်။

ခုနကပေးထားတဲ့ equals သတ်မှတ်ချက်မှာ ဒီစည်းမျဉ်းတွေကို တစ်ခုကလွဲပြီး ကျန်တာ

ကို အကုန်လိုက်နာပါတယ်။ ဘယ်ဟာကို မလိုက်နာတာပါလဲ။ လေ့ကျင့်ခန်းအနေနဲ့ ဖြစ်ကြည့်ပါ။

## ဝတ္ထု Method တစ်ခုကို နောက်ဝတ္ထု Method တစ်ခုမှ ခေါ်ယူခြင်း

ဝတ္ထု method တစ်ခုကို တခြားဝတ္ထု method တစ်ခုကနေ လှမ်းပြီးနှိုးဆွတာဟာ တရားဝင်သလို အသုံးလည်းများပါတယ်။ ဥပမာ complex ကိန်းတစ်ခုကို ပုံမှန်လုပ်ဖို့ ကိန်းစစ်နဲ့ စိတ်ကူးယဉ် နှစ်ပိုင်းစလုံးကို ပကတိတန်ဖိုးနဲ့ စားရပါတယ်။ ဒါဟာ ဘာကြောင့်အသုံးဝင်သလဲဆိုတာ သိသာချင်မှ သိသာပါလိမ့်မယ်။ ဒါပေမယ့် အသုံးတော့ ဝင်ပါတယ်။

ပုံမှန်လုပ်ပေးတဲ့ normalise method ကို ဝတ္ထု method တစ်ခုအနေနဲ့ ရေးရအောင်။ သူ့ကို ပြုပြင်တဲ့ method တစ်ခုအနေနဲ့ ထားပါမယ်။

```
public void normalise() {
    double d = this.abs();
    real = real/d;
    imag = imag/d;
}
```

ပထမစာကြောင်းဟာ လက်ရှိဝတ္ထုပေါ်မှာ abs ကိုနှိုးဆွပြီး လက်ရှိဝတ္ထုရဲ့ပကတိတန်ဖိုးကို ရှာပါတယ်။ ဒီနေရာမှာ လက်ရှိဝတ္ထုကို သိသိသာသာနာမည်ပြောပြီး ညွှန်းထားပါတယ်။ ဒါပေမယ့် ချန်ထားခဲ့ရင်လည်း ရပါတယ်။ ဝတ္ထု method တစ်ခုကို နောက်ဝတ္ထု method တစ်ခုထဲကနေ နှိုးဆွတဲ့အခါ Java ဟာ သူ့ကို လက်ရှိဝတ္ထုပေါ်မှာ နှိုးဆွနေတယ်လို့ ယူဆလိုက်ပါတယ်။

## အဆန်းတကြယ်များနှင့် အမှားများ

Class အဓိပ္ပာယ်သတ်မှတ်ချက် တစ်ခုတည်းမှာ ဝတ္ထု method တွေနဲ့ class method တွေ နှစ်မျိုးစလုံးရှိရင် ရှုပ်ထွေးရလွယ်ပါတယ်။ Class အဓိပ္ပာယ်သတ်မှတ်ချက်တစ်ခုကို အစီအစဉ်တကျဖြစ်အောင် စုစည်းဖို့ အသုံးများတဲ့နည်းက အစမှာ constructor တွေအားလုံးကိုထား၊ သူ့နောက်မှာ ဝတ္ထု method တွေရေးပြီး class method အားလုံးကို နောက်ဆုံးမှာထားတာဖြစ်ပါတယ်။

ဝတ္ထု method တစ်ခုနဲ့ class တစ်ခုကို နာမည်အတူတူ ပေးလို့ရပါတယ်။ Parameter အရေအတွက်နဲ့ အမျိုးအစားတွေ မတူသရွေ့ ဒါဟာ တရားဝင်ပါတယ်။ Overload လုပ်တဲ့ တခြားနေရာတွေမှာလိုပဲ Java ဟာ ပံ့ပိုးပေးတဲ့ argurment တွေကိုကြည့်ပြီး ဘယ်ပုံစံကို နှိုးဆွရမယ်ဆိုတာ ဆုံးဖြတ်ပါတယ်။

အခုချိန်မှာ static keyword ရဲ့အဓိပ္ပာယ်ကို သိပြီဖြစ်တာကြောင့် main ဟာ class method တစ်ခုဖြစ်တယ်ဆိုတာ တွေးကြည့်မိကြပါလိမ့်မယ်။ ဒီတော့ main ကိုနှိုးဆွနေချိန်မှာ လက်ရှိဝတ္ထုမရှိပါဘူး။ Class method တစ်ခုမှာ လက်ရှိဝတ္ထုမရှိတဲ့အတွက် this keyword

ကိုသုံးရင် မှားပါတယ်။ သွားကြိုးစားရင် this ဆိုတဲ့ variable ကို သတ်မှတ်မထားဘူးဆိုတဲ့ အမှား message တစ်ခုကို ရပါလိမ့်မယ်။ Instance variable တွေကိုလည်း အစက်လက္ခဏာနဲ့ ဝတ္ထုနာမည်မပံ့ပိုးပေးဘဲ ညွှန်းဆိုလို့ မရပါဘူး။ သွားလုပ်ကြည့်ရင် တည်ငြိမ် variable မဟုတ်တာကို တည်ငြိမ်အညွှန်းတစ်ခုကနေ မရောက်ရှိနိုင်ဘူးလို့ပြောတာကို ရပါလိမ့်မယ်။ ဒါဟာ စံချိန်စံညွှန်းပြုလုပ်မထားတဲ့ ဘာသာစကားကိုသုံးတာကြောင့် ကောင်းမွန်တဲ့အမှား message တစ်ခု မဟုတ်ပါဘူး။ ဥပမာ တည်ငြိမ်မဟုတ်တဲ့ variable ဆိုတာ instance variable ကို ဆိုလိုပါတယ်။ သူတို့ ဘာကိုဆိုလိုသလဲဆိုတာကို သိတာနဲ့ ဆိုလိုရင်းကို သိသွားပါတယ်။

**အမွေဆက်ခံခြင်း**

ဝတ္ထုကိုဦးတည်တဲ့ပရိုဂရမ်းမင်းနဲ့ ဆက်စပ်နေတဲ့ ဘာသာစကားလက္ခဏာဟာ အမွေဆက်ခံတာဖြစ်ပါတယ်။ အမွေဆက်ခံတယ်ဆိုတာ ထည့်သွင်းတည်ဆောက်ထားတဲ့ class တွေအပါအဝင် အရင်ကြိုတင်သတ်မှတ်ထားတဲ့ class တစ်ခုကို ပြုပြင်ပြီး class အသစ်တစ်ခုအနေနဲ့ ပြန်လည်သတ်မှတ်နိုင်စွမ်းပဲ ဖြစ်ပါတယ်။

ဒီလက္ခဏာရဲ့ အဓိကအကျိုးကျေးဇူးဟာ လက်ရှိ class ကို ပြုပြင်စရာမလိုဘဲ method အသစ်တွေနဲ့ instance variable တွေကို လက်ရှိ class မှာ ပေါင်းထည့်နိုင်တာ ဖြစ်ပါတယ်။ ဒါဟာ ထည့်သွင်းတည်ဆောက်ထားတဲ့ class တွေအတွက် အထူးတလည် အသုံးဝင်ပါတယ်။ ဘာလို့လဲဆိုတော့ သူတို့ကို ကိုယ်ကပြုပြင်ချင်ရင်တောင်မှ ပြုပြင်လို့ မရလို့ပါပဲ။

အမွေဆက်ခံတာကို အမွေဆက်ခံတယ်လို့ ခေါ်ရတဲ့အကြောင်းက class အသစ်ဟာ ရှိနေတဲ့ class ရဲ့ instance variable အားလုံးနဲ့ method အားလုံးကို အမွေဆက်ခံလို့ပါပဲ။ ဒီဆက်သွယ်ချက်ကို တိုးချဲ့လိုက်ရင် ရှိနေတဲ့ class ကို မိဘ class လို့လည်း ခေါ်ပါတယ်။

**ဆွဲသားနိုင်သောထောင့်မှန်စတုဂံများ**

အမွေဆက်ခံတာရဲ့ ဥပမာတစ်ခုအနေနဲ့ ရှိနေတဲ့ Rectangle class ကိုယူပြီး ဆွဲသားနိုင်အောင်လုပ်ပါမယ်။ ဆိုလိုတာက DrawableRectangle လို့ခေါ်တဲ့ class အသစ်တစ်ခုကို ဖန်တီးပြီး Rectangle တစ်ခုရဲ့ instance variable တွေနဲ့ method တွေမှာ Graphics လို့ခေါ်တဲ့ ဝတ္ထုတစ်ခုကို parameter အနေနဲ့ လက်ခံတဲ့ method အသစ်တစ်ခုကို ပေါင်းထည့်ပါမယ်။ သူက ထောင့်မှန်စတုဂံကို ဆွဲပါလိမ့်မယ်။

Class အဓိပ္ပာယ်သတ်မှတ်ချက်ဟာ အောက်မှာပြထားသလို ရှိပါတယ်။

```
import java.awt.*;


class DrawableRectangle extends Rectangle {


    public void draw(Graphics g) {
```

```
    g.drawRect(x, y, width, height);
  }
}
```

ဟုတ်ပါတယ်။ ဒါဟာ class အဓိပ္ပာယ်သတ်မှတ်ချက် တစ်ခုလုံးထဲမှာ ရှိရှိသမျှပါပဲ။ ပထမစာကြောင်းဟာ java.awt package ကို သွင်းယူပါတယ်။ သူ့ထဲမှာ Rectangle နဲ့ Graphics ကို သတ်မှတ်ထားပါတယ်။

နောက်စာကြောင်းဟာ DrawableRectangle က Rectangle ကို အမွေဆက်ခံတယ်ဆိုတာ ပြောပြပါတယ်။ extends ဆိုတဲ့ keyword ဟာ အမွေဆက်ခံတဲ့ မိဘ class ကို ခွဲခြားဖော်ပြဖို့ စီစဉ်ထားတာ ဖြစ်ပါတယ်။

ကျန်တာကတော့ draw method ရဲ့ အဓိပ္ပာယ်သတ်မှတ်ချက် ဖြစ်ပါတယ်။ သူက x, y, width နဲ့ height ဆိုတဲ့ variable တွေကို ညွှန်းပါတယ်။ Class အဓိပ္ပာယ်သတ်မှတ်ချက်ထဲမှာ မမြင်ရတဲ့ instance variable တွေကို ညွှန်းဆိုရတာဟာ နည်းနည်းဆန်းသလို ဖြစ်နေပါတယ်။ ဒါပေမယ့် သူတို့ကို မိဘ class ကနေ အမွေဆက်ခံယူထားတယ်ဆိုတာ သတိရပါ။

DrawableRectangle တစ်ခုကို ဖန်တီးပြီးဆွဲဖို့ အောက်မှာပြထားတာကို သုံးနိုင်ပါတယ်။

```
public static void draw(Graphics g, int x, int y, int width, int height)
{
    DrawableRectangle dr = new DrawableRectangle();
    dr.x = 10;
    dr.y = 10;
    dr.width = 200;
    dr.height = 200;
    dr.draw(g);
}
```

draw ရဲ့ parameter တွေက Graphics ဝတ္ထုတစ်ခုနဲ့ ပုံဆွဲမယ့်ဧရိယာရဲ့ ပတ်ပတ်လည်အကွက်တို့ ဖြစ်ပါတယ်။ ထောင့်မှန်စတုဂံရဲ့ ကိုဩဒိနိတ်တွေ မဟုတ်ပါဘူး။

Constructor မရှိတဲ့ class တစ်ခုအတွက် new command ကိုသုံးရတာ နည်းနည်းထူးဆန်းသလိုပါပဲ။ DrawableRectangle ဟာ မိဘ class ရဲ့ မူလ constructor ကို အမွေဆက်ခံပါတယ်။ ဒီတော့ အဲဒီမှာပြဿနာမရှိပါဘူး။

dr ရဲ့ instance variable တွေကို သတ်မှတ်တာနဲ့ dr ပေါ်မှာ method တွေနိုးဆွတာကို အရင်လိုပဲ လုပ်နိုင်ပါတယ်။ draw ကိုနိုးဆွတဲ့အခါ Java ဟာ DrawableRectangle မှာ သတ်မှတ်ထားတဲ့ method တွေကို နိုးဆွပါတယ်။ grow လို Rectangle method တစ်ခုကို dr ပေါ်မှာနိုးဆွရင် Java ဟာ မိဘ class မှာ သတ်မှတ်ထားတဲ့ method ကို သုံးရမယ်ဆိုတာ သိ

ပါတယ်။

## Class အဆင့်ဆင့်

Java မှာ class အားလုံးဟာ နောက်ထပ် class တစ်ခုခုကို အမွေဆက်ခံပါတယ်။ အခြေခံ အကျဆုံး class ကို Object လို့ ခေါ်ပါတယ်။ သူ့မှာ instance variable တွေ မပါပါဘူး။ ဒါ ပေမယ့် equals နဲ့ toString လို method တွေကိုတော့ ပံ့ပိုးပေးပါတယ်။

Class တော်တော်များများဟာ Object ကို အမွေဆက်ခံခြင်းအားဖြင့် တိုးချဲ့ထားတာဖြစ် ပါတယ်။ Rectangle လို ရေးဖူးခဲ့တဲ့ class တွေ အားလုံးနီးပါးနဲ့ ထည့်သွင်းတည်ဆောက်ထား တဲ့ class တော်တော်များများဟာ Object ကို တိုးချဲ့တည်ဆောက်ထားတာ ဖြစ်ပါတယ်။ မိဘ ကို သိသိသာသာနာမည်မဖော်တဲ့ class တိုင်းဟာ Object ဆီက အမွေဆက်ခံပါတယ်။

တချို့ အမွေဆက်ခံတဲ့ကွင်းဆက်တွေဟာ ပိုရှည်ပါတယ်။ ဥပမာ ဂရပ်ဖစ် class တစ်ခုဖြစ် တဲ့ Slate ဟာ Frame ကို တိုးချဲ့ပါတယ်။ Frame ဟာ Window ကိုတိုးချဲ့ပါတယ်။ Window ဟာ Container ကိုတိုးချဲ့ပါတယ်။ Container ဟာ Component ကို တိုးချဲ့ပါတယ်။ Component ဟာ Object ကိုတိုးချဲ့ပါတယ်။ ကွင်းဆက်ဘယ်လောက်ရှည်ရှည် Object ဟာ class တွေအားလုံးရဲ့ ထိပ်ဆုံးက မိဘဖြစ်ပါတယ်။

Java မှာရှိတဲ့ class အားလုံးကို အဆင့်ဆင့်တည်ဆောက်ပုံလို့ခေါ်တဲ့ မိသားစုဝင် ဆက်သွယ် ချက်ပုံနဲ့ ဖွဲ့စည်းနိုင်ပါတယ်။ Object က အပေါ်ဆုံးမှာ ရှိပါတယ်။ သူ့အောက်မှာ ကလေး class တွေ ရှိကြပါတယ်။ Frame ရဲ့စာရွက်စာတမ်းကိုကြည့်မယ်ဆိုရင် သူ့ရဲ့မျိုးကွဲကို တည်ဆောက် တဲ့ ဖွဲ့စည်းပုံအဆင့်ဆင့်ကို မြင်နိုင်ပါတယ်။

## ဝတ္ထုကိုဦးတည်သော ဒီဇိုင်း

အမွေဆက်ခံတာဟာ စွမ်းအားကြီးတဲ့ လက္ခဏာတစ်ခုဖြစ်ပါတယ်။ သူ့ကိုမသုံးလို့ ရှုပ်ထွေး သွားတဲ့ ပရိုဂရမ်တချို့ကို သူနဲ့ပြန်ရေးရင် ရိုးရိုးရှင်းရှင်း တိုတိုတုတ်တုတ် ဖြစ်သွားတာတွေရှိပါ တယ်။ ပြီးတော့ အမွေဆက်ခံတာဟာ code ကို ပြန်သုံးခွင့်ပေးပါတယ်။ ဘာလို့လဲဆိုတော့ ထည့်သွင်းတည်ဆောက်ထားတဲ့ class တွေရဲ့ အပြုအမူကို ပြုပြင်စရာမလိုဘဲ ကိုယ်လိုသလို ပြင်ယူနိုင်ပါတယ်။

အခြားတစ်ဖက်မှာတော့ အမွေဆက်ခံတာဟာ ပရိုဂရမ်တွေကို ဖတ်ရခက်စေပါတယ်။ Method တစ်ခုကို နှိုးဆွတဲ့အခါ ဘယ်နေရာမှာသတ်မှတ်ထားသလဲဆိုတာ မရှင်းလင်းတဲ့အတွက် လိုက်ရှာရ ခက်စေပါတယ်။ ဥပမာ Slate ပေါ်မှာ နှိုးဆွနိုင်တဲ့ method တစ်ခုဟာ getBounds ဖြစ်ပါတယ်။ getBounds အတွက် စာရွက်စာတမ်းကို ရှာနိုင်မလား။ ဖြစ်ချင်တော့ getBounds ကို Slate ရဲ့ လေးဆင့်မြောက် မိဘမှာ သတ်မှတ်ထားပါတယ်။

နောက်ပြီးတော့ အမွေဆက်ခံတာကိုသုံးပြီး လုပ်နိုင်တဲ့အရာတွေကို သူမပါဘဲ လုလုပပ လေး လုပ်နိုင်တာတွေလည်း ရှိပါတယ်။

**ကျော်အောင်နေပါ**

အခုအချိန်မှာ သင်ခန်းစာတွေ ပြီးဆုံးသွားပါပြီ။ အခုရှင်းပြခဲ့တဲ့ သင်ခန်းစာတွေဟာ ပရိုဂရမ်လေ့လာတဲ့လမ်းခရီးမှာ ကြီးမားတဲ့အထောက်အကူတွေကို ပေးပါလိမ့်မယ်။ ဒါပေမယ့် သူတို့ကိုတတ်တာနဲ့ Java တတ်သွားပြီဆိုပြီး လိုက်ကြွားလို့တော့ မရသေးပါဘူး။ အခုလေ့လာခဲ့တဲ့ အသိအပေါ်မှာ အခြေခံပြီး Java နဲ့ပတ်သက်တဲ့ အယူအဆတွေ၊ နည်းစနစ်တွေ၊ လုပ်ကိုင်နည်းတွေ၊ နည်းပညာတွေကို ဆက်လက်လေ့လာရဦးပါမယ်။

ရှင်းပြထားတာအားလုံးကို နားမလည်လို့လည်း စိတ်မပျက်ပါနဲ့။ မကျေနပ်ရင် နောက်တစ်ခေါက်ပြန်ဖတ်ပါ။ တခြားဖတ်ရှုစရာရှိတာတွေကိုလည်း တတ်နိုင်သမျှ လိုက်ဖတ်ပေးပါ။ ကိုယ်တိုင်လည်း လုပ်ကြည့်ပါ။ မသိရင် သိတဲ့လူကိုမေးပါ။ အချိန်တန်ရင် တတ်ကျွမ်းသွားပါလိမ့်မယ်။ အဆင်ပြေပါစေ။

## ရှင်မတောင်စာပေမှ ဆက်လက်ထွက်ရှိမည့် စာအုပ်များ

၁။ ဒေါက်တာမြင့်သန်း (ညောင်လေးပင်)
၂၀ ကျော် အစိတ်ပိုင်း
လူငယ်ကျန်းမာရေးဆိုင်ရာဆောင်းပါးများ

၂။ ဒေါက်တာခင်မောင်လွင် (လမ်းမတော်)
ဆီးနှင့် ကျောက်ကပ်ရောဂါများနှင့် ကာကွယ်ကုသနည်း

၃။ အောင်ကျော်ကျော် (ကျန်းမာရေးပညာ) စုစည်းသည့်
ခုခံအားကျ စကားဝိုင်း AIDS FORUM

၄။ ချောင်းဆုံအုန်းသွင်၏ ဆောင်းပါးပေါင်းချုပ်

၅။ ဒေါက်တာအေးကျော် (ဇီဝကမ္မဗေဒ)
ယောက်ျားတစ်ယောက်၏ ဘဝခရီး
(လိင်ပိုင်းဆိုင်ရာ ကျန်းမာရေးသိသင့်သည်များ)

၆။ ဒေါက်တာအေးကျော် (ဇီဝကမ္မဗေဒ)
အမျိုးသားတို့ သိသင့်သော အမျိုးသားတို့အကြောင်း
အမျိုးသားတို့ သိသင့်သော အမျိုးသမီးတို့အကြောင်း

၇။ ဒေါက်တာအေးကျော် (ဇီဝကမ္မဗေဒ)
ဖေဖေ မေမေ မာပါစေ (လူကြီးကျန်းမာရေး)

၈။ ဒေါက်တာမြင့်သန်း (ညောင်လေးပင်)
စိတ်နိုင်လျှင် ကိုယ်နိုင်မည်

၉။ ဒေါက်တာမြင့်သန်း (ညောင်လေးပင်)
ခွေး၊ ခွေးရူးနှင့် ခွေးရူးရောဂါ

၁၀။ ဒေါက်တာမြင့်သန်း (ညောင်လေးပင်)
စိတ်အိုလျှင် ကိုယ်အိုမည်

၁၁။ ဆရာဝန်ကြီးများဖြေကြားသည့်
ကျန်းမာရေး အမေးအဖြေများ အမှတ် (၁)

၁၂။ ဒေါက်တာလှကြည်
(ကလေးကျန်းမာရေးနှင့် အဟာရဆရာဝန်ကြီး)
ကလေးများတွင် ဖြစ်ပွားတတ်သော
အထွေထွေရောဂါနှင့် ပြဿနာများ

PDF Created By Ko Chan Sar Thu 25/9/14

သန္တအောင်

# လွယ်ကူသော JAVA သင်ခန်းစာများ

Java ရဲ့ စွမ်းအားတွေနဲ့အတူ နာဆာရဲ့ လေ့လာရေးယာဉ်တွေဟာ အာကာသတစ်ခွင် ခရီးနှင်နေကြပါပြီ။

Java ပရိုဂရမ်တွေရဲ့ ထိန်းချုပ်မှုအောက်မှာ သုတေသနယာဉ်တွေဟာ ကြမ်းတမ်းတဲ့ သမုဒ္ဒရာကြမ်းပြင်ကို လေ့လာနေကြပါပြီ။

Java ပရိုဂရမ်သုံး စက်ရုပ်တွေဟာလည်း လူ့အသက်ပေါင်းများစွာကို ကယ်ဆယ်နေပါပြီ။

ကမ္ဘာ့အအောင်မြင်ဆုံး နည်းပညာကုမ္ပဏီကြီးတွေဖြစ်ကြတဲ့ IBM, Oracle နဲ့ SAP တို့ဟာ သူတို့ရဲ့ ဩဇာတိက္ကမကြီးမားတဲ့ ထုတ်ကုန်တွေမှာ Java နည်းပညာကိုပေါင်းထည့်ပြီး နည်းပညာလောကကို တစ်ခေတ်ဆန်းစေနေပါပြီ။

Nokia, Motorola, Samsung, NTT DoCoMo စတဲ့ လက်ကိုင်ဖုန်း ကုမ္ပဏီတွေဟာလည်း နောက်ဆုံးထွက်သမျှ Java နည်းပညာတိုင်းကို သူတို့ရဲ့ handset တွေမှာ အပြေးအလွှား အကောင်အထည်ဖော်နေပါတယ်။

ပရိုဂရမ်းမင်း နည်းပညာဈေးကွက် တစ်ဝက်ကျော်ကိုလည်း IBM WebSphere Studio, Eclipse, NetBeans စတဲ့ ရေးသားမှုဆော့ဖ်ဝဲ (IDE) တွေက သိမ်းပိုက်ထားပါပြီ။

စာဖတ်သူရော Java နဲ့ ဘာလုပ်ဖို့ စီစဉ်ထားလဲ။

စိတ်ဓာတ် (Spirit) နှင့် အခွင့်အလမ်း (Opportunity) ဟု နာမည်ပေးထားသော အာကာသယာဉ်နှစ်စင်းသည် ဖလော်ရီဒါပြည်နယ် Cape Canaveral လေတပ်စခန်းမှ ၂၀၀၃ ခု ဇွန်လ ၁၀ နှင့် ဇူလိုင်လ ၇ တို့တွင် ထွက်ခွာခဲ့သည်။ လေ့လာရေးယာဉ်များတွင် ကမ္ဘာ့အနှံ့ရှိ တက္ကသိုလ်များနှင့် သုတေသနဌာနများမှ ထုတ်လုပ်ပေးသော သိပ္ပံကိရိယာများကို အင်္ဂါဂြိုဟ်၏ ဘူမိဗေဒနှင့်လေထုကို ခွဲခြမ်းစိတ်ဖြာနိုင်ရန် တပ်ဆင်ထားသည်။ ၎င်းကိရိယာများကို ထိန်းချုပ်ရန်နှင့် ဂြိုဟ်မျက်နှာပြင်တွင် ထိန်းချုပ်သူမဲ့ မောင်းနှင်မှု ပရိုဂရမ်များအတွက် Java ပရိုဂရမ်မင်းဘာသာစကားနှင့် Java နည်းပညာကို အသုံးပြုမည်ဟု နာဆာက ဆုံးဖြတ်ခဲ့သည်။ ၎င်းယာဉ်များသည် အင်္ဂါဂြိုဟ်မျက်နှာပြင်တွင် ၂၀၀၄ ခုနှစ် ဇန်နဝါရီ ၄ နှင့် ဇန်နဝါရီ ၂၅ တွင် ဆင်းသက်ပြီး အင်္ဂါဂြိုဟ်တွင် ရေရှိခဲ့ကြောင်း အထောက်အထားနှင့်တကွ လူသားအသိဉာဏ်ပညာအတွက် အဖိုးမဖြတ်နိုင်သော အချက်အလက်များကို ပေးပို့ခဲ့သည်။