HOW DOES JVM WORK?

AUTHOR: © KO THET KHINE

Presented by: Programming Wiki

https://programmingwiki.org

Table Of Contents

HOW DOES JVM WORKS PART-1

Modern programming language တွေမှာ ဟိုး C,C++ ကလို native platform (os+hardware) အပေါ်မှာ တိုက်ရိုက် run နိုင်တဲ့ exe လိုကောင်တွေ မထုတ်ကြ တော့ပါဘူး။ အဲ့အစား runtime intermediate code တွေကို ထုတ်ပြီးတော့ runtime system တွေကို သုံးပြီး ခုနက ထုတ်ထားတဲ့ intermediate code တွေကို execute လုပ်စေ ကြပါတယ်။ ဘာကောင်းသလဲဆိုတော့ intermediate code တခု ထဲ ရှိရုံနဲ့ Platform အမျိုးမျိုးမှာ run လို့ ရသွား တာပေါ့။ ဥပမာ Java မှာဆို bytecode ဆိုတာကို ထုတ်ပါတယ်။ Java Compiler ရဲ့ output ဟာ .class တခု သို.မဟုတ် .class ဖိုင်တွေ ဖြစ်ပါတယ်။

အဲ့ဒီ .class ဖိုင်တွေကို byte code လို့ ခေါ်ပါတယ်။ Microsoft Visual Studio Family ကတော့ (VB.NET, C#, F#) ကတော့ intermediate code အနေနဲ. MSIL (Microsoft Intermediate Language) ဆိုတဲ့ intermediate code ကိုထုတ်ပါတယ်။ အဲ့ဒီ MSIL ကိုတော့ CLR (Common Language Runtime) ဆိုတဲ့ Microsoft VM မှာ run ပါတယ်။ ဘာလို. Common Language Runtime လို့ ခေါ်သလဲ ဆိုတော့ C# ကထုတ်တဲ့ MSIL သည် VB.Net ကထုတ်ထားတဲ့ MSIL ကနေတင် class တွေ method တွေကို ယူသုံးတာ inherit လုပ်တာမျိုး ရပါတယ်။

အဲ့တော့ VB.net က ရေးထားတဲ့ code ကို C# ကနေခေါ်သုံးလို.ရသွားတာပေါ့ဗျာ။ Java ထက်နာတဖျား သာပြီး cross language Interoperability ရသွားတယ် ဆိုပါစို.ဗျာ။

WHAT IS JVM?

JVM ဆိုတာ Java bytecode ကို ယူပြီးတော့ execute လုပ်ပေးတဲ့ software သို.မဟုတ် hardware နဲ့ Implement လုပ်ထားတဲ့ machine တခုပါပဲ။ Physical Machine (ဒီနေရာမှာ CPU) သည် machine language ကို နားလည် ပါတယ်။ အဲ့ဒီ Physical Machine ပေါ်မှာ နားလည်တဲ့ machine code ထုတ်မယ် ဆိုရင် ဆိုကြပါစို့ Intel processor အတွက် machine code ထုတ်ရင် (ဒီနေရာမှာ ဘယ် OS အတွက် ဆိုတာကလဲ ဆိုင်သေးတယ်) တခြား machine တွေမှာ run လို့ အဆင်မပြေပါဘူး။ ဥပမာ window os + intel machine အတွက် ထုတ်ထားတဲ့ machine code သည် Linux အပေါ်မှာ သွား run မှာ မဟုတ်ပါဘူး။

ဒါကြောင့် Programming language designer တွေကနေပြီး language တွေကို platform အားလုံးမှာ run နိုင်အောင် ကြံဆ ကြပါတယ်။ အဲ့တာနဲ့. ဘာသုံးလဲ ဆိုတော့ intermediate code တခုထုတ်မယ်။ ဥပမာ byte code ပေါ့ အဲ့လို code ကို နောက်ထပ် software layer, သို့မဟုတ် specific hardware နဲ့. Run မယ် ဒါဆိုရင် ဒီ Language ဟာ platform အားလုံးမှာ run နိုင်မယ်ပေါ့။ အဲ့လိုရည်ရွယ်တာပါ။ Virtual Machine concept ဆိုတာ Small Talk language မှာကတည်းက ရှိခဲ့တာပါ။ သူ့နောက်ပိုင်း ထွက်တဲ့ Self မှာဆို JIT Concept မျိုးပါ ရှိနေပါပြီ။ ဒီနေရာမှာ Virtual Machine ဆိုတာ programming language တခုအတွက် Virtual Machine ကို ဆိုလိုတာလို. သတိချပ်စေ ချင်ပါတယ်။ ဘာလို့ လဲဆိုတော့ Hardware Virtualization လို့ခေါ်တဲ့ OS တွေ အတွက် Virtual Machine Technology ကလဲ သက်သက် ရှိလို့ပါပဲ။ ခပ်ဆင်ဆင်တူပေမဲ့ OS အတွက် Virtualization က ပိုကျယ်ပြန့်ပါတယ်။ ဒီနေရာမှာ language specific virtual machine ကိုပဲ ဆိုချင်တာပါ။ ကျောင်းသားဘဝက Presentation လုပ်နေတုံး VM code ကို ထိုင်ကြည့်နေတဲ့ ပါမောက္ခကြီးကတော့ ပြောရှာပါတယ်။ မင်းကတော့ VM ကို မင်းကိုယ်တိုင်ရေးတယ် ပြောတာပဲတဲ့။ ဌဲ့ဌဲ့။ သူထင်နေရှာတာက သူနဲ့ သက်ဆိုင်တဲ့ OS Virtual Machine တွေကို ရေးတယ်လို့ ထင်နေရှာတာပါ။ ပြောလဲ မထူးမှာမို့ (အမှန်က ထပ်ပြောရင် ရုံးခန်း ရောက်ဦးမှာ) မို့ ဘာမှတော့ ပြန်ပြော မနေတော့ ပါဘူး။

Virtual Machine တွေမှာ အဓိက မပါမဖြစ် သတ်မှတ်ရတာက intermediate code file format ပါ။ ဥပမာ Java ဆိုရင် JVM byte code class file format, .NET language တွေ ဆိုရင် MSIL ပေါ့ဗျာ။ နောက်မှ အဲ့ဒီ byte code ကို ဖတ်ပြီးတော့ virtual machine က execute လုပ်တာပါ။ ဒီနေရာမှာ အများအားဖြင့်တော့ software နဲ့. implement လုပ်ထားတဲ့ virtual machine တွေကို သုံးပါတယ်။ Hardware နဲ့လဲ virtual machine တခုကို တည်ဆောက်လို့ ရပါတယ် ဆိုတာလည်း သိစေချင်ပါတယ်။ အဲ့တော့ ကျွန်တော်တို့က Java program ရေးတယ်။ နောက် bytecode ထုတ်တယ်။ .class file ပေါ့။ အဲ့ဒီကောင်ကို window ပေါ်မှာ run ချင်ရင် window အတွက် ရေးထားတဲ့ JVM(Window Version) ကို သုံးတယ်။ Linux အတွက် ဆိုရင်တော့ Linux အတွက် ရေးထားတဲ့ JVM (Linux Version) ကို သုံးတယ်။ အဲ့တော့ Java Program တပုဒ်ဟာ ဘယ် Platform မှာမဆို run လို့ ရပြီပေါ့ဗျာ။ အဲ Virtual Machine လေးတော့ ရှိရမှာပေါ့နော် run ချင်တဲ့ platform မှာ။ များသောအားဖြင့် JVM တွေကို C++ နဲ့ ရေးကြပါတယ်။ Java နဲ့ ရေးထားတဲ့ JVM တွေ ရှိပေမဲ့လည်း ဒါက academic concept တခုကို proof လုပ်ချင်ရုံလောက်ပဲ သက်သက် အတွက်နဲ့ လုပ်ပြတာမျိုးမှာ။ Performance အပိုင်းကို C++ နဲ့ ရေးထားတဲ့ JVM တွေကို မမှီဘူးလို့ ဆိုရမှာပါ။

Java အတွက် byte code file format ကို JVM Specification ဆိုတာကို ထုတ်ပြီး byte code file မှာ ဘယ်လို ပုံစံတွေနဲ့ သိမ်းတယ် ဘယ်လို instruction set တွေ ပါတယ်။ အဲ့ဒီ instruction set တွေက ဘယ်လို အလုပ်လုပ်တယ် ဆိုတာကို ရေးသားထားပါတယ်။ အဲ့ဒီတော့ ကြိုက်တဲ့သူက ဒီ Specification ကို လိုက်နာပြီး JVM ကို ရေးလို့ ရပါတယ်။

STACK BASED MACHINE

JVM က stack based machine ပါ...။ ဒီနေရာမှာ Android မှာ သုံးတဲ့ ART(Android Run Time) ဆိုတာ ကတော့ register based machine ပါ။ Stack Based machine ဆိုတာ instruction တွေ execute လုပ်ဖို့ အတွက် stack ကို data structure အနေနဲ့ သုံးပြီး တည်ဆောက်ထားတဲ့ VM ကို ဆိုချင်တာပါ။ ဥပမာ ကျွန်တော်တို့က c = a + b အတွက် byte code ထုတ်မယ် ဆိုရင် ဒီလိုထုတ် ရမှာပါ...

load a
load b
add
store c

ဒါဆို ပထမဆုံး stack ပေါ်မှာ a ရဲ့ တန်ဖိုးကို တင်မယ် နောက် b ရဲ့ တန်ဖိုးကို တင်မယ်။ ပြီးတော့ add ဆိုတဲ့ op code(operation code) ကို တွေ့တော့ stack ပေါ်က a ရယ် b ရယ်ကို pop လုပ်ချမယ် ဒီနှစ်ခုကို ပေါင်းမယ် ပြီးတော့ stack ပေါ်ကို ပြန်တင်မယ်။ နောက် store c ဆိုတော့ ခုနက stack အပေါ်ဆုံးမှာ ရောက်နေတဲ့ a + b တန်ဖိုးကို c ထဲကို သိမ်းလိုက်မယ်။ ဒီလိုနည်းနဲ့ run တာကို stack based machine လို့ ခေါ်ပါတယ်။ register based machine ဆိုတာ ကတော့ လွယ်လွယ် ပြောရရင် assembly language နဲ့ ဆင်တူ ပါတယ်။

INTERPRETATION, COMPILATION, AOT, HOTSPOT, JIT

Bytecode တွေရဲ့ instruction တွေကို line by line read ပြီးတော့ execute လုပ်ရင် ဒါက interpretation ပါ။ အဲ့လို interpret လုပ်ရတာက နေးပါတယ် ဘာလို့လဲ ဆိုတော့ bytecode instruction opcode တွေက ဘာလုပ်မယ်(ဘယ် opcode လဲ add ဆိုရင် ပေါင်းရမယ် ဒါဆို stack ပေါ်က ကောင်တွေကို pop လုပ်ပြီးပေါင်း ပြီးပြန်တယ်) ဆိုတာကို decode လုပ်ရတဲ့ အတွက် ကြာပါတယ်။ အဲ့ဒီအစား အဲ့ဒီခုနက stack ပေါ်ကနေ pop ချ ပေါင်းပြီး ပြန်တင် အဲ့ဒီ အလုပ်တွေကို machine code အနေနဲ့ ပြောင်းလိုက်ပြီး ခုနက byte code နေရာမှာ အစားထိုး လိုက်ပါတယ် ။ Implementation အရဆို machine code ထုတ်ပြီး byte code array ထဲမှာ void pointer အနေနဲ့ ထားပါတယ်။ နောက်မှ pointer တွေ typecast လုပ်ပြီး function call အနေနဲ့ eexecute လုပ်ခိုင်းတာပါ။

အဲ့သည်လို Bytecode ကနေ machine code ကို runtime မှာ ပြောင်းပစ်တာကို Compilation လုပ်တယ် လို့ ခေါ်ပါတယ်။ Compilation မှာလဲ ပြဿနာ ရှိပါတယ် ။ ဒါက ဘာလဲ ဆိုတော့ ခုနက bytecode တွေကို machine code အနေနဲ့ compile လုပ်ဖို့ အချိန်ရယ် memory ရယ် ကုန်ပါတယ်။ ဒါကြောင့် ဘာမှ မ run ခင် ကြိုပြီး compile လုပ်ပါတယ် အားလုံးကို ဒါကို AOT (Ahead of Time) Compilation လို့ ဆိုပါတယ်။ သုံးသုံး မသုံးသုံး အားလုံးကို AOT လုပ်တော့လဲ အဆင်မပြေပါဘူး။ မသုံးတဲ့ method တွေ ဥပမာ exception class တွေ အတွက်ဆို အလဟဿ ဖြစ်ပါတယ်။ ဒါကြောင့် လိုမှပဲ compile လုပ်မယ်ပေါ့။ ဒါကို HotSpot လို.ခေါ်ပါတယ်။

HotSpot ဆိုတာ method တခုက run ပြီးရင် သူ့ကို counter လေးနဲ့ မှတ်ထားပါတယ်။ ခနခန run လေ counter လေးက တက်လာလေပေါ့ အဲ့ counter လေးက threshold တခု ရောက်လာမှ အဲ့ဒီ method ရဲ့ bytecode တွေကို machine code အနေနဲ့ Compile လုပ်ပါတယ်။ ဒါကို ပူနေတဲ့ နေရာလေးပဲ (hotspot) လေးတွေပဲ ကွက် compile လုပ်တဲ့ အတွက် hotspot compilation လို့ သုံးပါတယ်။ အပေါ်က ပြောခဲ့တဲ့ Interpretation , AOT, HotSpot အဲ့ဒါတွေကို စုပြီး runtime မှာ dynamic translation လုပ်တာကိုတော့ JIT (Just In Time) Compilation လို့ ခေါ်ပါသတဲ့ဗျာ.

GARBAGE COLLECTION

Java မှာ object တွေကို new operator သုံးပြီး dynamically allocate လုပ်ပါတယ်။ အဲ့ဒီ Object တွေကို Heap ဆိုတဲ့ memory မှာ သိမ်းပါတယ်။ Heap ဆိုတာ ကတော့ runtime system ကနေ memory ကို request လုပ်တဲ့အခါ လွတ်နေတဲ့ နေရာကနေ ယူပေးတယ်။ မြင်သာအောင် ပြောရရင် ကျွန်တော်တို့ စာအုပ်စားပွဲလိုပေါ့ တခုခု တင်ချင်ရင် လွတ်နေတဲ့ နေရာလေး ချလိုက်တယ်။ နောက်အဲ့ကောင်ကို မလိုချင်ရင် စားပွဲပေါ်က ချလိုက်တယ်။ Stack လို အပေါ်ဘက်ကနေပဲ တဖက်တည်း နေရာယူတာ မဟုတ်ပဲ စားပွဲပေါ်မှာ နေရာယူသလို လွတ်နေတဲ့ နေရာလေးကပဲ ယူလိုက်တဲ့ အတွက် ကြာလာရင် ဖရိဖရဲဖြစ်လာတယ်။ ဥပမာ နေရာလွတ်တော့ ရှိနေတယ် ဒါပေမဲ့ စားပွဲပေါ်မှာ တင်ထားတဲ့ Object တွေကို ပြန်စီရမယ်ပေါ့။ ဒါဆို နေရာလွတ် ထွက်လာနိုင်တယ် ပေါ့။ ဒါမျိုးကို Garbage Collection လို့ ခေါ်ပါတယ်။

Runtime system ကနေ Object တွေ allocate လုပ်ရင်း နေရာလိုလာတဲ့အခါမှာ အသုံးမလိုတော့တဲ့ memory location တွေကို ပြန် reclaim လုပ်တာမျိုးကို Garbage Collection လုပ်တယ်လို့ သုံးပါတယ်။ ဒါမျိုးကို JVM က လုပ်ပေးပါတယ်။ Oracle JVM မှာတော့ Generational Garbage Collection ဆိုတဲ့ Algorithm ကို သုံးပါတယ်။ Heap တွေကို Generation အပေါ် မူတည်ပြီး ခွဲပစ်ပါတယ်။

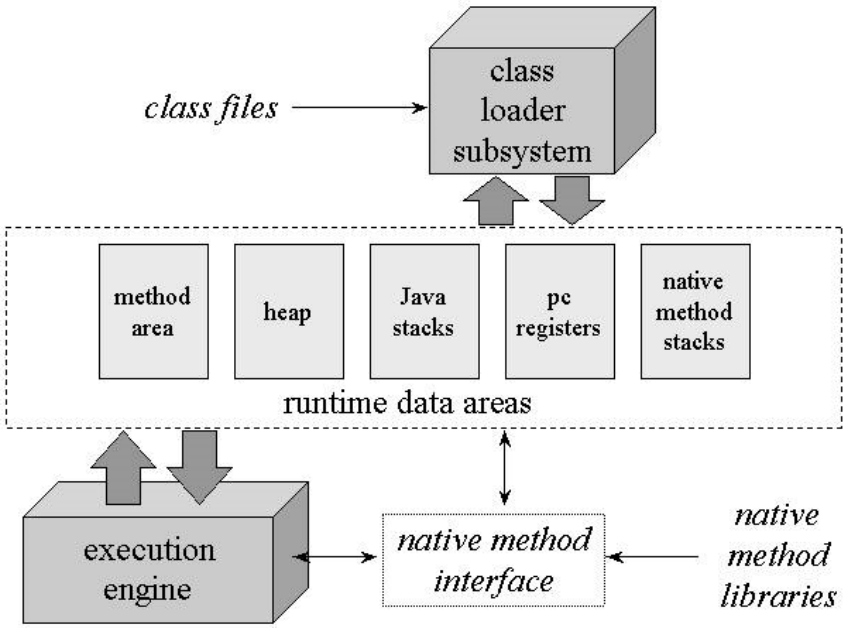
Generational GC ဆိုတာက အချိန်အကြာကြီး သုံးတဲ့ Object တွေနဲ့ အချိန် သက်တမ်း သိပ်ကြာကြာ မလိုအပ်တဲ့ Object တွေကို နေရာခွဲပြီး ရွှေ့ပစ် ပါတယ်။ ဘာလို့လဲ ဆိုတော့ GC ကို run ရင်လဲ အချိန်ကုန်လို့ပါပဲ။ Old generation တွေအပေါ်မှာ GC ကို ခနခန run စရာ မလိုပဲ short live Heap မှာပဲ GC ကို run ရင် ရပါပြီ။ Memory အကုန်လုံး အပေါ်မှာ GC ကို run စရာ မလိုပဲ။ Partition အပေါ် မူတည်ပြီး run ရတဲ့ အတွက် GC Collection time က သက်သာပါတယ်။ Oracle JVM မှာ သုံးတာက Generational GC ကို သုံးပြီး တခြား GC algorithms တွေနဲ့ ပေါင်းသုံးတာပါ။ ဘာမှမပြောရသေးပဲ စာရှည်သွားတဲ့အတွက် နောက်တပိုင်း မျှော်ပေါ့ ဗျာ... ။

HOW DOES JVM WORKS PART – 2

တွေ့ဖူးတဲ့ Java programmer တော်တော် များများက Java ကိုသာ လေ့လာမယ် JVM လို့ မေးရင် ဘာကြီးတုံး ဆိုတာမျိုး ပြန်မေးတာမျိုး ကြုံရဖူး ပါတယ်။ Java programmer တွေ အနေနဲ့. ဘာကြောင့် JVM ကို သိသင့်သလဲ ဆိုတော့ programmer တယောက်အနေနဲ့ ကို သုံးနေတဲ့ language runtime ကို သိဖို့ လိုပါတယ်။ ဒါမှ large application တွေ ရေးတဲ့အခါမှာ performance, scalability, safety, အစရှိတာတွေကို ကျွမ်းကျင်စွာ ကိုင်တွယ်နိုင်မှာပါ။

ဥပမာ Java က automatic garbage collection ပေးထားတယ် ဒါကြောင့် Java program တွေမှာ memory leak မရှိနိုင်ဘူး ဆိုတဲ့ အယူအဆ အမှားတွေ တွေ့ရပါတယ်။ ဒါဟာ ဘာကြောင့်လဲ ဆိုတော့ JVM အလုပ်လုပ်ပုံကို သေချာနားမလည်တာ ကြောင့်ပါ။ နောက်တခုက Garbage Collection ပါတဲ့ အတွက် ငါတို့က ကြိုက်သလောက် object ဆောက်မယ် ဘာဖြစ်လဲ memory ကို automatic reclaim ပြန်လုပ်ပေးမှာပဲ ဆိုတာမျိုး တွေးတာလဲ တွေ့ရပါတယ်။ ဘာဖြစ်လဲဆိုတော့ automatic reclaim လုပ်ပေးတာတော့ ဟုတ်တယ် ဒါပေမဲ့ အဲ့လိုလုပ်ဖို့ အတွက် GC algorithm က run ရတယ်။ GC algorithm ကလဲ computational time ကုန်တဲ့အတွက် program က run နေရင်းမှာ တခါတလေ GC ကြောင့် GC pause (Garbage Collection algorithm ကို run နေရတဲ့ အတွက် JVM က တခြား thread တွေကို အလုပ်မလုပ်နိုင်တော့ပဲ ခဏလောက် ရပ်နေရသလို အခြေအနေမျိုး) ဖြစ်လာနိုင်တာကို နားမလည်ကြပါဘူး။

တခြားအချက်တွေကတော့ ဟိုတရက်ကတင် ဆရာကြီးများ JavaEE များ ရပ်သွားလေမလား ဆိုပြီး စိုးရိမ်နေကြတာမျိုးပါ။ JVM ဘယ်လို အလုပ်လုပ်လဲဆိုတာ သိတော့ Framework တွေ library တွေရေးတဲ့နေရာမှာ တပမ်းသာ ပါလိမ့်မယ်။



STRUCTURE OF JVM

အပေါ်က ပုံမှာ ပြထားတာကတော့ JVM ရဲ့ အကြမ်းဖျင်း ပုံပါပဲ။ အဲ့ကောင်တွေ အကြောင်းကို အရင်မရှင်းခင်မှာ byte code class file အကြောင်းကို ရှင်းပါရစေ။ အောက်မှာ ပြထားတဲ့ Java program ဆိုပါစို့

```
class HelloWorld
{
    public static void main (String [] args)
    {
        System.out.println("Hello World");
    }
}
```

သူ့ကို javac command နဲ့ compile လုပ်လိုက်ရင် HelloWorld.class ဆိုတဲ့ byte code file ထွက်လာ ပါတယ်။ byte code file က binary file ပါ။ အဲ့တော့ ဖွင့်ဖတ်ဖို့ အဆင်မပြေပါဘူး။ bytecode ကို memonic code (binary အစား လူနားလည်နိုင်တဲ့ code) အနေနဲ့ ကြည့်ချင်ရင် JDK မှာပါတဲ့ javap ဆိုတဲ့ tool ကို သုံးလို့ ရပါတယ်။ Command prompt မှာ အောက်ကလို ရိုက်လိုက်ရင် (ပထမဆုံး HelloWorld.class ဖိုင် ထွက်အောင်တော့ အရင် compile လုပ်ထား ရပါမယ်)

javap -v HelloWolrd

HelloWorld အတွက် bytecode ကို ထုတ်ပေး ပါလိမ့်မယ်။

```
Classfile /D:/Program Coding/Java/HelloWorld.class
 Last modified Jul 16, 2016; size 425 bytes
 MD5 checksum 65c3a384152c83f16c6afcb42b788fab
 Compiled from "HelloWorld.java"
class HelloWorld
        SourceFile: "HelloWorld.java"
        minor version: 0
        major version: 51
        flags: ACC_SUPER
Constant pool:
#1 = Methodref #6.#15 // java/lang/Object."<init>":()V
#2 = Fieldref #16.#17 // java/lang/System.out:Ljava/io/PrintStream;
#3 = String #18 // Hello World
#4 = Methodref #19.#20 // java/io/PrintStream.println:(Ljava/lang/String;)V
#5 = Class #21 // HelloWorld
#6 = Class #22 // java/lang/Object
#7 = Utf8 <init>
#8 = Utf8 ()V
```

```
#9 = Utf8               Code
#10 = Utf8              LineNumberTable
#11 = Utf8              main
#12 = Utf8              ([Ljava/lang/String;)V
#13 = Utf8              SourceFile
#14 = Utf8              HelloWorld.java
#15 = NameAndType       #7:#8 // "<init>":()V
#16 = Class             #23 // java/lang/System
#17 = NameAndType       #24:#25 // out:Ljava/io/PrintStream;
#18 = Utf8              Hello World
#19 = Class             #26 // java/io/PrintStream
#20 = NameAndType       #27:#28 // println:(Ljava/lang/String;)V
#21 = Utf8              HelloWorld
#22 = Utf8               java/lang/Object
#23 = Utf8              java/lang/System
#24 = Utf8              out
#25 = Utf8              Ljava/io/PrintStream;
#26 = Utf8              java/io/PrintStream
#27 = Utf8              println
#28 = Utf8               (Ljava/lang/String;)V

{
        HelloWorld();
        flags:
        Code:
        stack=1, locals=1, args_size=1

        0: aload_0
        1: invokespecial     #1 // Method java/lang/Object."<init>":()V 4: return
        1: invokespecial     #1 // Method java/lang/Object."<init>":()V
        4: return
```

```
        LineNumberTable: line 1: 0

public static void main (java.lang.String []);
        flags: ACC_PUBLIC, ACC_STATIC
        Code:
                stack =2, locals=1, args_size = 1
                0: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
                3: ldc          #3      // String Hello World
                5: invokevirtual #4  // Method
                java/io/PrintStream.println:(Ljava/lang/String;)V 8: return

        LineNumberTable:
        line 5: 0
        line 6: 8
}
```

အပေါ်က line တွေက HelloWold.class ကို disassembly လုပ်ထားတာပါ။ class file တွေ ထိပ်ဆုံးမှာ ပါတာကတော့ CAFEBABE ဆိုတဲ့ hexa နဲ့ ရေးထားတဲ့ file descriptor ပါပဲ။ အဲ့ဒါကို အပေါ်က disassembly လုပ်တော့ ပြမထားပါဘူး။ minor major version ဆိုတာ ဒီ class file ကို ထုတ်ပေးထားတဲ့ language version ဘယ်လောက်လို့ ဆိုချင်တာပါ။ major version 51 ဆိုတာ JDK 1.7 နဲ့ ထုတ်ထားတာလို့ ဆိုချင်တာပါ။ JVM က class file တခုကို run တဲ့ အခါမှာ minor version major version ကို စစ်ပါတယ်။ အဲ့ဒီ class file ဟာ သူလုပ်နိုင်တဲ့ range အတွင်းမှာ ရှိမှ JVM က class file ကို run ခွင့် ပေးတာပါ။ Java Class file format ကို JVM Specification မှာ အသေးစိတ် ကြည့်နိုင်ပါတယ်။

CONSTANT POOL

Java class ဖိုင်မှာ အရေးကြီးတဲ့ data structure တခုပါ ပါတယ် သူက Constant Pool ဆိုတာပါ။ သူ့ မှာ ဘာတွေသိမ်းထားလဲ ဆိုတော့ byte code instruction တွေက လိုအပ်တဲ့ numerical literal တွေ။ string literal, တွေ class name တွေ method descriptor တွေ, parameter type တွေ, data type descriptor တွေ အစရှိတာတွေကို သိမ်းထားပါတယ်။

Java က Dynamic Linking ကိုသုံး ပါတယ်။ Dynamic Linking ဆိုတာ လိုအပ်တဲ့ runtime library တွေကို intermediate code ထဲမှာ တခါတည်း ထဲ့ပေါင်း မထားပဲနဲ့ runtime ရောက်တော့မှ runtime system ကနေ dynamically load ခေါ်တင်တာကို ဆိုလိုတာပါ။ ဥပမာ HelloWorld ကနေ System ဆိုတဲ့ java class ကို သုံးပါတယ်။ HelloWorld.class မှာ System ရဲ့ bytecode မပါပါဘူး။ Constant pool ထဲမှာ ထည့်ပြီးတော့ပဲ HelloWorld သည် System class ကိုထည့်သုံး ထားတယ်လို့ မှတ်ထား တာပါ။ ဒါကို JVM ကနေ runtime ရောက်တော့မှ အသုံးလိုတယ် ဆိုရင် memory ပေါ်ကို System class ကို ခေါ်တင်မှာပါ။ ဒါကို Dynamic Linking လုပ်တယ်လို့ ခေါ်ပါတယ်။ အဲ့ဒီလို လုပ်နိုင်ဖို့ အတွက် bytecode တခုမှာ သုံးထားတဲ့ external class တွေကို constant pool ထဲမှာ မှတ်ထား ရပါတယ်။ Constant Pool ထဲမှာ bytecode က သုံးထားတဲ့ class တွေ method တွေ field တွေကို မှတ်ထားရပါတယ်။ ဒါတွေကို Methodref , FieldRef, Class အစရှိတာတွေနဲ့ constant pool မှာ မှတ်ထားပါတယ်။
ဥပမာ

```
#1 = Methodref      #6. #15       // java/lang /Object."<init>":()V
```

#1 ဆိုတာက constant pool entry index ပါ။ MethodRef ဆိုတာက ဒီ entry သည် method တခုကို မှတ်ထားတယ် ဆိုတာ ပြတာပါ။ MethodRef တခုကိုပြချင်ရင် ဘယ် class ရဲ့ ဘယ် method ဆိုတာကိုပြရပါတယ်။ ဒီမှာဆို #6 သူက class ကိုညွှန်းတာပါ

```
#6 = Class    #22    // java/lang/Object
```

Class ကနေ ဘာကိုထပ်ညွှန်းလဲ ဆိုတော့ class နာမည်ကို string အနေနဲ့ ပြန်ညွှန်း ရပါတယ် entryt 22 ကိုညွှန်းတော့

```
#22 = Utf8              java/lang/Object ပါ …
```

ဒါဆို ခုနက ညွှန်းတဲ့ method ရဲ့ class သည် java/lang/Object ဆိုတာကို သိပါပြီ။
ဘယ် method လဲ ဆိုတာကိုတော့ #15 ကနေ ကြည့်ရမှာပါ။ #15 က

```
#15 = NameAndType        #7:#8          // "<init>":()V
```

NameAndType ဆိုတော့ method name ရယ် return type ရယ်ပေါ့ဗျာ…

<init> ဆိုတာ constructor ကို ကိုယ်စားပြုတာပါ။ ()V ဆိုတာ ကတော့ method က parameter မပါဘူး (ကွင်းထဲမှာ ဘာမှမရှိဘူး။) နောက်က V ဆိုတာကတော့ void ပေါ့။
ဒါဆို constant pool entry 1 သည်

#1 = Methodref #6.#15 // java/lang/Object."<init>":()V

ဘာကိုပြလဲဆိုတော့ java.lang.Object ထဲက no argument constructor ကို သုံးပါတယ်ဆိုတဲ့ reference ကိုပြတာပါ။ JVM က class file တခုကို load ခေါ်တင်မယ်ဆိုရင် byte code ကိုအရင်ဖတ်ရပါတယ် ပြီးတော့ constant pool ကို တည်ဆောက်ပါတယ်။
ခုနက MethodRef ဆိုရင် သူနဲ့.ဆိုင်ရာ ဆိုင်ရာ internal data structure တွေတည်ဆောက်ရပါတယ်။ ဒါက internally ပါ။ byte code မှာ နောက်ဘာထပ်ပါလဲ ဆိုတော့ constant pool ပြီးရင် သူ့ရဲ့ supre class တွေ super interface တွေ field တွေ method တွေပါရပါတယ်။ C style structure နဲ့ဆို byte code format က ဒီလိုရှိမှာပါ။

```
struct Class_File_Format {
   u4 magic_number;

   u2 minor_version;
   u2 major_version;

   u2 constant_pool_count;

   cp_info constant_pool[constant_pool_count - 1];

   u2 access_flags;

   u2 this_class;
   u2 super_class;

   u2 interfaces_count;

   u2 interfaces[interfaces_count];

   u2 fields_count;
   field_info fields[fields_count];

   u2 methods_count;
   method_info methods[methods_count];

   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

u4 ဆိုတာ 4 byte, u2 ဆိုတာ 2 byte ပေါ့ဗျာ။ အပေါ်က ပြထားတဲ့ HelloWorld.class မှာ method နှစ်ခု ပါပါတယ်။ Source Code မှာ main method တခုပဲ ရေးထားပေမဲ့ Java Compiler ကနေ default argument constructor ထည့်ပေးထားတဲ့ အတွက် method ၂ ခု ဖြစ်သွားတာပါ။ အောက်က main method လေးကို ဒီလို သိမ်းထားပါတယ်။

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC

  Code:

    stack=2, locals=1, args_size=1
      0: getstatic     #2              // Field java/lang/System.out:Ljava/io/PrintStream;
      3: ldc           #3              // String Hello World
      5: invokevirtual #4              // Method java/io/PrintStream.println:(Ljava
/lang/String;)V
      8: return

    LineNumberTable:
      line 5: 0
      line 6: 8

}
```

အဓိက က code section ပါ။ အခုမှာပါတဲ့ stack ဆိုတာက operand stack ပါ သူ့ အတွက် 2 word ပဲလိုမယ်တဲ့ locals ကတော့ local variable တွေ သိမ်းဖို့ပါ။ args_size parameter တွေ သိမ်းဖို့ ပါ ကျွန်တော်တို့က static method ဖြစ်တယ် နောက် String[] ကို လက်ခံတဲ့ အတွက် 1 word နဲ့ လုံလောက်ပါတယ်။ ဒီလိုလေးရေးထားပါတယ်...

stack = 2, locals=1, args_size=1

နောက်က

```
0: getstatic      #2    // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc            #3    // String Hello World
5: invokevirtual  #4    // Method java/io/PrintStream.println:
                        (Ljava/lang/String;)V
8: return
```

ဒါတွေကတော့ System.out.println("Hello World"); ကို ကိုယ်စားပြုတဲ့ byte code ပါ။ ပထမဆုံး 0 address မှာရှိတဲ့ bytecode က getstatic ပါ။ getstatic ဆိုတာ class တခုရဲ့ static field ကို ခေါ်တင်မယ်လို့ ပြောတာပါ။ ဒီနေရာမှာ System class ရဲ့ out ဆိုတဲ့ field ကို ခေါ်တင်တာပါ။ အောက်က code ရဲ့ meaning က java.lang.System class ထဲကနေ static field ဖြစ်တဲ့ out ဆိုတဲ့ field ကို operand stack ပေါ်တင်မယ် လို.ဆိုလိုပါတယ်။

```
0: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
```

နောက်တကြောင်းက

```
ldc             #3      // String Hello World
```

ldc ဆိုတာ load constant ကို ပြောတာပါ။ HelloWorld ဆိုတဲ့ String ကိုလဲ operand stack ပေါ်တင်မယ်လို့ ဆိုလိုတာပါ။ JVM ဟာ Stack based ဖြစ်တဲ့ အတွက် လိုအပ်တဲ့ method တွေ မခေါ်ခင် method ရဲ့ object ရယ် parameter တွေရယ် ကို operand stack ပေါ် ခေါ်တင်ရပါတယ်။ ခုဆို stack ပေါ်မှာ System.out ဆိုတဲ့ object ရယ် Hello World ဆိုတဲ့ string ရဲ့ reference ရယ် ရောက်နေပါပြီ။ နောက်တကြောင်းက

```
invokevirtual   #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

invokevirtual ဆိုတာက virtual method (static မဟုတ်တဲ့ constructor မဟုတ်တဲ့ interface method မဟုတ်တဲ့ method တွေကိုခေါ်ရင် သုံးရပါတယ်) execute လုပ်ခိုင်းတာပါ။ ဒီမှာဆို println ဆိုတဲ့ method ကို execute လုပ်ခိုင်းတာပါ။ သူက parameter တခု လက်ခံပါတယ် (Ljava/lang/String;) ရှေ့က L ဆိုတာက object type ကို ပြတာပါ။ parameter သည် reference type java.lang.String ကို လက်ခံမယ်။ return type ကတော့ V ဆိုတော့ void ပေါ့။

ဒါဆို void println(String arg)ဆိုတဲ့ method ကိုလှမ်းခေါ်တာပါ။ ဘယ်သူ့ class ထဲကလဲ ဆိုတော့ ava/io/PrintStream ထဲကပါ။ ဒါဆို ဒီ method ကို execute လုပ်ဖို့ PrintStream object ဖြစ်တဲ့ out ရယ် parameter ဖြစ်တဲ့ HelloWorld ရယ် က operand stack ပေါ်မှာ ရှိပြီးသား ဖြစ်တဲ့အတွက် ဒါတွေကို pop လုပ်ချပြီး method ကို execute လုပ်မှာပါ။ method က return ပြန်လာတဲ့ ကောင်တွေကို stack ပေါ်တင်ရမှ ဖြစ်ပေမဲ့ void ဆိုတဲ့ အတွက် တင်စရာတခုမှ မရှိပါဘူး။
Code section က

stack=2, locals=1, args_size=1

ဒီအကြောင်းလေးမှာပါတဲ့ stack =2 ဆိုတာ operand stack သည် maximum depth အနေနဲ့ 2 ပဲ ရှိမယ်လို့ ဆိုလိုတာပါ။ ဒီနေရာမှာ out ရယ် Hello World string ရယ် (method မခေါ်ခင်က) stack ပေါ်မှာ ရှိတဲ့အတွက် max-depth က 2ပါပဲ။ နောက်တကြောင်း ကတော့

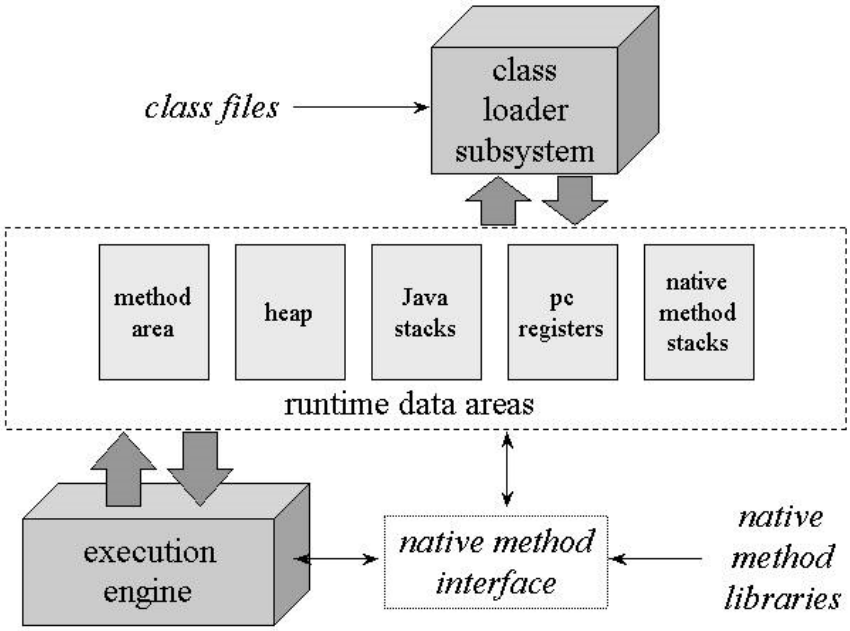
8: return ပါ

သူကတော့ method ကို execution လုပ်တာ ရပ်ပြီး Method call stack ပေါ်ကနေ method frame တခုကို pop လုပ်ခိုင်းတာပါ။ ဒီနေရာမှာ သူ့အောက်မှာ ဘာ method မှမရှိတဲ့အတွက် JVM ကနေ exit လုပ်ပါပြီ။ ဆက်ရန် :D
ဘာပဲပြောပြောဖတ်လိုက်တော့ အူတူတူတော့နိုင်သွားတယ် မို့လား :P

HOW DOES JVM WORK PART-3

CLASSLOADER

အောက်ကပုံမှာ JVM ရဲ့ structure ကို ပြထားပါတယ်။ ဒီအပိုင်းမှာတော့ သူ့မှာပါတဲ့ subsystem တွေနဲ့ runtime data structure တွေ... နောက် Execution Engine အကြောင်းကို ရှင်းသွားမှာပါ။



STRUCTURE OF JVM

CLASSLOADER

Java program တပုဒ်ကို (အမှန်က java compiler က ထုတ်ပေးထားတဲ့ byte code java class file) ကို JVM က စ run တော့မယ် ဆိုတာနဲ့ ပထမဆုံး လုပ်ရတဲ့ အလုပ်က အဲ့ဒီ run မဲ့ Java class file ကို memory ပေါ်ကို ခေါ်တင်ရတာပါ။ အဲ့ဒီ အလုပ်ကို ClassLoader တွေက လုပ်ပါတယ်။ ClassLoader တွေက byte code ဖိုင်နဲ့ ကိုက်ညီတဲ့ class တွေကို file system ကနေ ဖတ်မယ် ပြီးတော့... ပြီးတော့ လိုအပ်တဲ့ runtime data structure တွေဆောက်မယ် ဥပမာ runtime constant pool လို ကောင်မျိုးတွေ... နောက် JVM ကနေ internal class (C++ representation of Java class) ဖြစ်အောင် ဖွဲ့မယ်... နောက် loading ပြီးသွားရင် linking, verification, preparation အစရှိတဲ့ step တွေကို လုပ်ဆောင်ရ ပါတယ်။ JVM မှာ Class Loader တွေကို hierarchical အရ အဆင့်ဆင့် ခွဲထားပါတယ်။

Bootstrap class loader

သူက ပင်မ Class Loader ပါပဲ။ သူ့ကိုတခြား Class Loader တွေကနေ parent class laoder အနေနဲ့ သုံးကြပါတယ်။ သူ့ကို core java libraries တွေ load ခေါ်တင်ဖို့သုံးပါတယ်။ သူ့ကိုကျတော့ JVM implementation language ဖြစ်တဲ့ C++ နဲ့ရေးထားရပါတယ်။ java.lang.Object အစရှိတဲ့ API class တွေကို ဒီ Boostrap Class Loader နဲ့ ခေါ်တင်တာပါ။

Extension class loader

သူကခုနက Boostrap class loader ရဲ့ child အနေနဲ့သုံးပါတယ်။ သူ့ကိုကျတော့ extension library တွေကို load လုပ်ဖို့ အတွက်သုံးပါတယ်။ Extension library တွေ ဆိုတာက JRE ရဲ့ lib/ext folder အောက်က library တွေပါပဲ။ Sun ရဲ့ JCE Jav Cryptography Extension လိုကောင်မျိုးတွေကို ဒီ Extension Class Loader သုံးပြီး ခေါ်တင်ပါတယ်။

System class loader

သူကတော့ ခုနက Extension Class Loader ရဲ့ child အနေနဲ့ သုံးပါတယ်။ သူ့ကိုကျတော့ application ရဲ့ classpath မှာရှိတဲ့ jar file တွေ library တွေကို ခေါ်တင်ရင်သုံးပါတယ်။ ကျန်တော်တို့ သုံးနေတဲ့ application library တွေ JavaEE library တွေဟာ ဒီ System Class Loader ကနေ ခေါ်တင်တာပါ။

User-defined class loader

Java မှာ java.lang.ClassLoader ကို extend လုပ်ပြီး Custom Class Loader တွေ တည်ဆောက်လို့ ရပါတယ်။ အဲ့ဒီ class loader တွေအမျိုးအစားကို User-defined class loader လို့ ခေါ်ပါတယ်။ ဘာအတွက်သုံးလဲဆိုရင် တခါတလေမှာ ကျန်တော်တို့သုံးတဲ့ library တခုဆီက မတူညီတဲ့ class file တွေကို ဒါမှမဟုတ် jar file တွေကို သုံးနေရတာရှိပါတယ်။ အဲ့ဒီအခါမှာ jar file ၂ ခုကို တခါတည်း runtime classpath ထဲ ထည့်လိုက်ရင် မူလက class file တွေပျောက်ပြီး အသစ်တွေပဲရှိမှာပါ။ ဒါဆို version အဟောင်းကိုသုံးတဲ့ API က method not found လို error မျိုး တက်နိုင်ပါတယ်။ ဒါမျိုးကို resolve လုပ်တဲ့နည်းက (version မတူတဲ့ class file ၂ခုကို JVM ပေါ်ခေါ်တင်ချင်တဲ့ အခါမျိုးမှာ) User defined class loader တွေရေးပြီး သုံးကြပါတယ်။ JVM Class Loader တွေ အလုပ်လုပ်ပုံအရ တူညီတဲ့ class တခုပင် ဖြစ်ပါစေဦး... မတူညီတဲ့ class loader တွေကနေ ခေါ်တင်ခံရရင် အဲ့ဒီ class ၂ခုကို မတူဘူးလို့သတ်မှတ်ပါတယ်။

Class loading process

Classs တခုကို JVM ပေါ် load ခေါ်တင်တော့မယ်ဆိုရင် သူ့ရဲ့ parent class loader ကနေ ခေါ်တင်ပြီးပြီလား အရင်ကတည်းက ခေါ်တင်ပြီးသားလားဆိုတာကိုအရင်စစ်ပါတယ်။ သူ့ရဲ့ parent class loader မှာ parent class loader ရှိနေသေးရင် ဆက်ပြီး ခေါ်တင်ပြီးပြီလားဆိုတာစစ်ပါတယ်။ parent class loader တွေကနေ ခေါ်မတင်ရသေး မှသာ current class loader က loading လုပ်ပါတယ်။ ဒါကို delegation principle လို့သုံးပါတယ်။ Child class loader ဟာ parent class loader ကနေ ခေါ်တင်ထားတဲ့ cached လုပ်ထားတဲ့ class တွေကိုမြင်လို့ရပါတယ်။

ဒါကို Visibility principle လို့ ခေါ်ပါတယ်။ Parent က load လုပ်ပြီးသား class ကို child class loader တွေက ဘယ်တော့မှ ပြန်ပြီး reload မလုပ်ပါဘူး။ ဒါကိုတော့ uniqueness principle လို့ခေါ်ပါတယ်။ JVM class loading မှာ အခဲ့ဒီ principle သုံးခုကိုသုံးပါတယ်။ Class loading ကို Java က Dynamic Linking ကို support လုပ်တဲ့ language ဖြစ်လို့ runtime မှာ လိုမှသာလျှင် load လုပ်ပါတယ်။ တကယ်လို့ အစောကတည်းက Class တခုကို load လုပ်စေချင်ရင်...

ဥပမာ Database driver class တွေဆို datbase related API call တွေမခေါ်ခင် အရင် load ခေါ်ရပါတယ်။ အခဲ့ဒီအခါ Class.forName (String className) ဆိုတဲ့ method ကိုသုံးပြီး ခေါ်တင်လို့ ရပါတယ်။ ဂါသနာပါလို့ java program တပုဒ်က ဘယ်လို class တွေ ခေါ်တင်သွားလဲဆိုတာ သိချင်ရင် ဆိုပါစို့. HelloWorld ဆိုတဲ့ program ကို compile လုပ်ပြီး HelloWorld.class ဖိုင် ထွက်ပြီးပြီ java command နဲ့ run ပြီး JVM က ဘယ် class တွေကို ခေါ်တင်သွားလဲသိချင်ရင်

java –verbose :class HelloWorld

ဆိုပြီး ရိုက်လိုက်ရင် JVM ကခေါ်တင်သွားတဲ့ class list တွေ jar file တွေရယ်ကို တသီတတန်း ကြီး မြင်ရပါလိမ့်မယ်။ ClassLoader ကနေ parent classloader အဆင့်ဆင့်သွား load လုပ်တယ် မရလို့ class file ကို ဖတ်မယ် ရှာမတွေ့ဘူးဆိုရင်တော့ NoClassDefFoundError exception ကို JVM က throw ပါလိမ့်မယ်။

Linking

C,C++ ဆို language မျိုးမှာ လိုအပ်တဲ့ library တွေကို dll မထုတ်ပဲ static linking ကို သုံးပြီး လိုအပ်တဲ့ library function တွေကို target exe တွေမှာ တခါတည်း ထည့် ပေးလိုက်ပါတယ်။ အခဲ့ဒီနည်းကို static linking လို့ ခေါ်ပါတယ်။ runtime မတိုင်ခင် ကတည်းက လုပ်တဲ့အတွက် static linking ပါ။ Java မှာ class ဖိုင်တွေဟာ သူတို့ အတွက် ဆိုင်တဲ့ bytecode ကိုသာ မှတ်သားထားတာပါ။ ဥပမာ class တခုမှာ nested inner class ပါလာရင်တောင် inner class အတွက်သီးသန့်. class file တခု ထုတ်ပါတယ်။ အခဲ့ဒီအတွက် Java bytecode တဖိုင်မှာ လိုအပ်တဲ့ တခြား library တွေ API class တွေ ပါမလာပါဘူး။ ဒီ class file က ဘယ် library ,API class တွေကိုသုံးတယ်ဆိုတာ byte code ရဲ့ constant pool ဆိုတဲ့ data structure မှာသာ symbolic reference အနေနဲ့ မှတ်သား ထားတာပါ။ Symbolic reference ဆိုတာ သူသုံးတဲ့ API, class တွေကို string အနေနဲ့သာ နာမည်ပဲ မှတ်ထားတယ်လို့ ဆိုချင်တာပါ။

Runtime ရောက်တဲ့အခါကျတော့ ခုနက library တွေကို သုံးထားတာတွေ့လို့ လိုအပ်ပြီဆိုရင် JVM memory ပေါ်ကို ခေါ်တင်ရပါတယ်။ ဒါက loading ပါ။ loading ပြီးလို့ ရှိရင် ခုနက symbolic reference ဖြစ်တဲ့ string နေရာမှာ actual class native implementation နဲ့ အစားထိုး ရပါတယ်။ ဒါကို linking လို့ ခေါ်ပါတယ်။ runtime မှာ လုပ်တာ ဖြစ်တဲ့အတွက် dynamic linking လို့ ခေါ်ပါတယ်။

Verification

Linking ပြီးရင် verification ကို လုပ်ရပါတယ်။ Java က security ကို language level တင် မကပဲ နဲ့ JVM level မှာပါ ထိန်းထားပါတယ်။ C/C++ မှာလို buffer overflow တွေက security ကို ကောင်းကောင်း ဒုက္ခပေးနိုင်တယ်ဆိုတာ Java langauge desinger က သိပါတယ်။ ဒါကြောင့် class file တခုဟာ runtime stack ကို တနည်းနည်းနဲ့ မဟုတ်တာ လုပ်လေမလားဆိုပြီး စစ်ပါတယ်။ ဒါကို verification လို့ ခေါ်ပါတယ်။ ဥပမာ မြင်သာအောင် ပြောရရင် add ဆိုတဲ့ byte code ဟာ integer add ဆိုရင် သူ့အတွက် stack ပေါ်မှာ operand 2 ခု ရှိရပါမယ်။ operand 2 ခု မရှိပဲ add ကို ခေါ်ရင် stack က သူ.အောက်က runtime memory ကို တနည်းနည်းနဲ့ ထိနိုင်ပါတယ်။ Buffer overflow attack လို ကောင်မျိုးတွေ ဖြစ်လာနိုင်ပါတယ်။ ဒါမျိုး မဖြစ်အောင် bytecode တိုင်းဟာ သူတို့ instruction sequence အရ operand stack အရ valid operation ဖြစ်ရဲ့လား ဆိုတာကို စစ်ပါတယ်။ ဒါကို Verification လို့ဆိုပါတယ်။ Class တခုကို verification လုပ်တဲ့အခါမှာသူ့ရဲ့ parent class တွေ interface တွေကိုပါ တခါတည်း verification မလုပ်ရသေးရင် လုပ်ရပါတယ်။

Preparation

Preparation ကတော့ class file ကို verification ပြီးသွားရင် သူ့အတွက် static data တွေကို allocated လုပ်ပေးတာကို ခေါ်တာပါ။ လက်ရှိ prepared လုပ်နေတဲ့ class တင်မကပဲ သူ့ရဲ့ parent class တွေ interface တွေ အတွက်ပါ static field တွေရှိရင် preparation လုပ်ရပါတယ်။

Resolution

Class တခုရဲ့ method တွေကို run တော့မယ် ဆိုရင် အဲ့ဒီ class က တခြား Class ရဲ့ method တွေ field တွေ အစရှိတာကို ယူသုံးနိုင်ပါတယ်။ ဒါတွေကို constant pool ထဲမှာ မှတ်ထားတာပါ။ ခုနက သုံးတဲ့ constant pool entry MethodRef, ClassRef, FieldRef အစရှိတာ တွေကို loaded ခေါ်တင်ပြီး runtime representation (C++ representation of method, field and class) နဲ့ အစားထိုးတာကို resolution လို့ ခေါ်ပါတယ်။

ဆက်ရန် :3

HOW DOES JVM WORK PART-4

OBJECT LAYOUT AND VIRTUAL TABLE

OO language တွေမှာ Object creation ကို မဖြစ်မနေ ပေးရပါတယ်။ အဲ့အတွက် JVM မှာ Object creation နဲ့ပတ်သတ်ပြီး data structure တွေ သိမ်းရပါတယ်။ နောက် polymorphism feature အတွက်လဲ virtual method table လို entries တွေသိမ်းရပါတယ်... ဥပမာ java class တွေကို internal native C++ class structure နဲ့ မှတ်ပြီး သိမ်းဖို့ လိုပါတယ်။ အဲ့ဒီ internal class structure ထဲမှာ ဒီ class က object file layout အရ memory ဘယ်လောက် allocate လုပ်ရမယ် Virtual method ဘယ်နခု ပါတယ်။ ဒါကြောင့် object layout မှာ virtual table entries ဘယ်လောက် ထားရမယ် ဆိုတာတွေကို မှတ်ထားရပါတယ်။

Object layout

Java Object တွေကို new operator သုံးပြီး Object ဆောက်တာဖြစ်ဖြစ် တခြားနည်းနဲ့ Object ပဲ ဆောက်ဆောက် JVM က Object creation ကို လုပ်ပေးရပါတယ်။ အဲ့လို လုပ်တဲ့အခါ ဆောက်မဲ့ Object သည် ဘယ်လောက် နေရာ ယူရမယ်။ သူ့ထဲမှာ ဘယ်လို filed တွေ ပါတယ် ဆိုတာကို Object layout နဲ့ သိမ်းထား ရပါတယ်။ Object layout ဆိုတာ class တခုမှာ field ဘယ်နစ်ခု ပါတယ် တခုချင်းဆီအတွက် word ဘယ်လောက်ယူမယ်။ ဘယ် filed ကို access လုပ်မယ်ဆိုရင် memory အရ word index ဘယ်လောက် အစရှိတာတွေကို မှတ်ထားတာပါ။

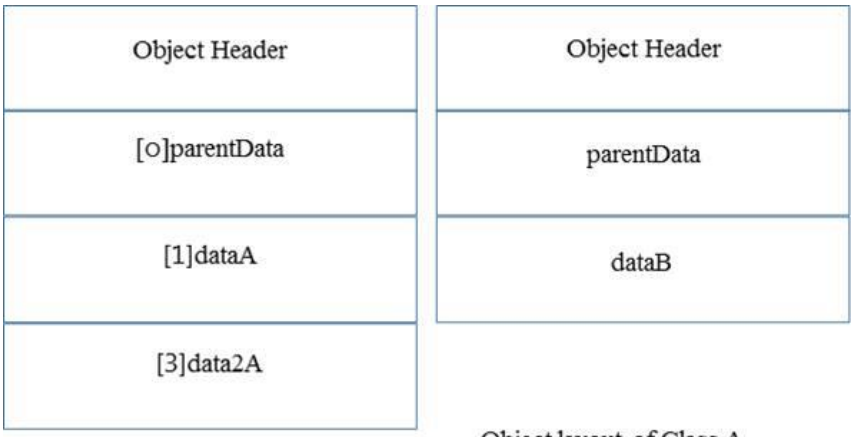
C++ နဲ့ ရေးမယ် ဆိုရင်တော့ သူ့ကို byte array နဲ့ပဲသိမ်းပါတယ်။ Object data တွေကို ပြီးမှ data type အပေါ် မူတည်ပြီး pointer type cast လုပ်ပြီး double, integer အစရှိတာတွေကို byte array ထဲမှာ သိမ်းပါတယ်။ ဒါမှ Java Object တခု ဆောက်လိုက်တာနဲ့ JVM က သူရဲ့ internal classs strucutre ကို ကြည့်ပြီး ဒီ object အတွက် memory ဘယ်လောက် allocated လုပ်ပေးပါ ဆိုပြီး runtime ကို တောင်းပါတယ်။ Object layout ထဲမှာ ဘာတွေပါသေးလဲ ဆိုတော့ garbage collection လုပ်ဖို့ အတွက် လိုအပ်တဲ့ memory reference တွေ ပါပါသေးတယ်။ နောက်ဒီ Object က ဘယ် class ရဲ့ instance လဲဆိုတာ သိဖို့အတွက် object တိုင်းမှာ class information ကု မှတ်ထားပါတယ်။ Java class ကို represent လုပ်ထားတဲ့ native C++ class ရဲ့ pointer နဲ့ သုံးပြီး မှတ်ပါတယ်။

ဥပမာ instanceof operator လို ကောင်မျိုးကို စစ်မယ်ဆိုရင် object layout ထဲက class information ကိုသုံးပြီး စစ်မှာပါ။ နောက်တခြား meta data တွေလိုပါသေးတယ်

ဥပမာ flag လိုကောင်မျိုးပါ... သူ့ကိုကျတော့ Object ကဘာ type လဲ ဥပမာ array လား ရိုးရိုး Object လား ဆိုတာမျိုး မှတ်ဖို့သုံးပါတယ်။ နောက် thread synchronization လုပ်ဖို့ အတွက် လိုအပ်တဲ့ lock လို ကောင်မျိုးတွေကိုလဲ object layout မှာ သိမ်းပါတယ်။ အဲ့ဒီတော့ Object layout ဆိုတာ JVM ကလိုတဲ့ meta data+ all field လို့ ဆိုရမှာပါ။။ JVM specification အရတော့ Object layout က ဘယ်လိုပုံစံရှိရမယ် ဆိုတာ မပြောထားပါဘူး။ Metadata တွေကိုတော့ Object header လို့ခေါ်ပြီးတော့ ကျန်တဲ့ all field ကိုတော့ object body လို့ခေါ်ပါတယ်။ Field တွေ တွက်တဲ့ နေရာမှာ ယခုလက်ရှိ class တင်မကပဲနဲ့ သူ့ရဲ့ parent class အစဉ်အလိုက် အတွက်ပါ ထည့် မှတ်ထား ပေးရပါတယ်။ ဒါမှသာ allocated လုပ်တဲ့အခါ base class က field တွေ အတွက်ပါ သိမ်းလို့ ရအောင်ပါ။ Object size ကို သိချင်ရင်တော့ OpenJDK မှာ jol ဆိုတဲ့ tool တခုပါပါတယ်။ http://openjdk.java.net/projects/co... ဒီလင့်ကနေ သွားကြည့်ပါ။ java.lang.Integer အတွက် Object size က object header အတွက် (12 byte) နဲ့ integer data အတွက် (4 byte) စုစုပေါင်း 16 byte ယူပါတယ်။ java.lang.Object ရဲ့ object size ကလဲ 16 bytes ပါပဲ။ 64 bit VM ပေါ်မှာ စမ်းထားတာပါ။ အဲ့တော့ အနည်းဆုံး java object တခုဖန်တီးမယ် ဆိုတိုင်း 16 bytes ကတော့ သုံးရပါပြီ။ Hibernate လို framework တွေမှာ primitive type တွေ မသုံးပဲနဲ့ warpper object တွေနဲ့ data ကို သိမ်းတာဟာ memory အရ ဘာဖြစ်သွားမယ် ဆိုတာကို တွေးကြည့်နိုင်လောက်ပြီလို့ ထင်ပါတယ်။

```
class Parent
{
    String parentData;
    void method()
    {
    }
    void methodTwo()
    {}
}
class A extends Parent
{
    int dataA;
    int data2A;
}
class B extends Parent
{
    int dataB;
    void method()
    {}
}
```

အပေါ်ကပြထားတဲ့ class တွေအတွက် A နဲ့ B ရဲ့ object layout ကို အောက်ကပုံမှာပြထားပါတယ်။



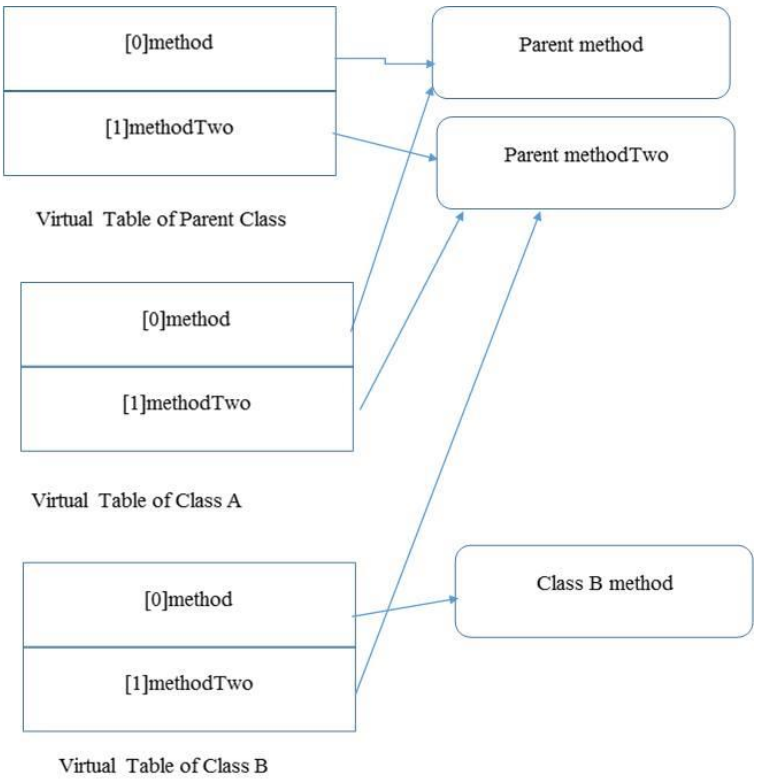
အပေါ်ကပုံမှာ ပြထားတဲ့အတိုင်း Class တခုရဲ့ object layout ထဲမှာ သူ့ parent တွေရဲ့ data တွေ အတွက်ပါ နေရာ မှတ်ထားတယ်လို့ ဆိုရမှာပါ။ ဒီနေရာမှာ java.lang.Object အတွက် ထည့် မဆွဲထားဘူး လို့ပဲ သဘောထားပေးပါ။ တကယ်တမ်းပုံမှာ ပြထားတာက size တူနေပေမဲ့ string က reference ဖြစ်တဲ့အတွက် 1word ပဲ နေရာယူပြီး data , dataB က integer ဖြစ်တဲ့အတွက် 2 word နေရာယူမယ်ဆိုတာ သတိပြုပါခင်ဗျာ။

အမှန်က Object layout ဆိုတာ object data တကယ် သိမ်းတာမဟုတ်ပဲနဲ့ ဘယ် member data က ဘယ် index မှာ နေရယူမယ် ဆိုတာမျိုး သိမ်းပေးထားယုံ ပါပဲ။ Object creation အတွက် memory size ဘယ်လောက်လိုမယ် field တွေ access လုပ်တဲ့ bytecode တွေအတွက် object ထဲက memory index ဘယ်လောက်ကို ထိရမလဲဆိုတာတွေကိုပဲ မှတ်ထားတာပါ။ ဥပမာ class A အတွက် parentData က index 0 မှာ သိမ်းပါတယ်။ parentData သည် 1 word ပဲ ယူတဲ့ အတွက် နောက်က field dataA သည် index 1 မှာ နေရာယူပါတယ်။
dataA ကတော့ integer 2word ဖြစ်တဲ့ အတွက် သူ့နောက်က data2A သည် index 3 မှာ နေရာယူရပါတယ်။ ဒီနေရာမှာ တခုသတိထားရမှာက Object layout တွက်တဲ့နေရာမှာ parent က object တွေရဲ့ data ကိုပါ အစဉ်အလိုက် တွက်ရမယ်ဆိုတာပါပဲ။ ဒါမှ inheritance လုပ်ထားတဲ့ class တွေအတွက် parent data တွေကိုပါ သိမ်းနိုင်မှာပါ။

Virtual method table

OO Programming မှာ နောက်ထပ် အရေးပါတဲ့ feature တခုက polymorphism ပါ။ သူ့ ကိုအလုပ်လုပ်ဖို့ ဆိုရင် JVM ကနေ parent မှာရှိတဲ့ virtual method တွေကို child virtual method တွေနဲ့ အစားထိုးပြီး လုပ်ပေးရပါတယ် ။ ဒါမှ parent reference ကို သုံးပြီး method call ခေါ်လိုက်လဲ child method က... ထ run မှာပါ။ ဒါမျိုး polymorphism ကို အထောက်အကူပြုဖို့ အတွက် JVM မှာ class internal implementation မှာ virtual method table ဆိုတာကို ဆောက်ထား ရပါတယ်။

အောက်ကပုံမှာ အပေါ်က class ၃ ခုအတွက် virtual table ကို ပြထားပါတယ်။



Virtual Table entry တွေ ဆောက်တဲ့နေရာမှာ ပထမဆုံး parent class ရဲ့ method တွေကို အစဉ်အတိုင်း အရင်ထည့် ပါတယ်။ ပုံမှာပြထားတဲ့ အတိုင်း class parent ရဲ့ method တွေကို virtual table ထဲမှာ အစဉ်အတိုင်း တွေ့ ရမှာပါ။ Class A ကတော့ သူကိုယ်တိုင် ဘာ method မှ မရှိပေမဲ့ parent class ကနေ extends လုပ်ထားတာ ဖြစ်တဲ့အတွက် သူ့ virtual table ထဲမှာ Parent class ကရတဲ့ virtual table အတိုင်း ဖြစ်နေမှာပါ။

Class A object ရဲ့ method ဒါမှမဟုတ် methodTwo ကို invoke လုပ်မယ်ဆိုရင် Parent class ရဲ့ method , methodTwo ကိုပဲ invoke လုပ်ပေးမှာပါ။ Class Bကကျတော့ ပထဆုံး သူ့ parent ဖြစ်တဲ့ Parent Class ရဲ့ virtual table ကို အရင်ဖြည့်ပါတယ်။ နောက် သူ့ကိုယ်ပိုင် method တခု ပါလာပါတယ် ။ Override လုပ်ထားတဲ့ method ဆိုတဲ့ method ပါ။ Override လုပ်ထားတဲ့ အတွက် သူ့ရဲ့ virtual table entry 0 မှာ သူ့ ကိုယ်ပိုင် class ရဲ့ method pointer နဲ့ အစားထိုး လိုက်ပါတယ်။

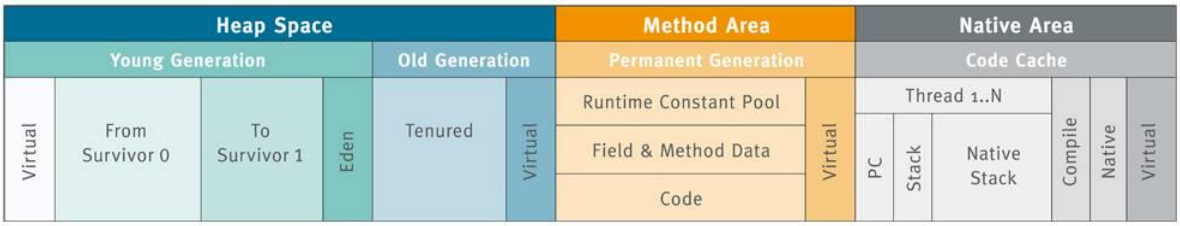
methodTwo ကတော့ override လုပ် မထားတဲ့ အတွက် Parent class က ကောင်ကိုပဲ ညွှန်နေဦးမှာပါ။ Class B ရဲ့ method ကို invoke လုပ်မယ်ဆိုရင် virtual table အရ B ရဲ့ own method ကိုသွား run မှာပါ။ ဒါဆို polymorphism ပေးနိုင်ပါပြီ။ B ရဲ. methodTwo ကို invoke လုပ်မယ် ဆိုရင်တော့ B ရဲ့ Virtual table entry အရ Parent class ရဲ. methodTwo ကိုပဲ invoke လုပ်သွားမှာပါ။ ဆက်ရန် :3

HOW DOES JVM WORK PART-5

Runtime data area

JVM မှာ byte code တွေကို execute လုပ်ဖို့ internal class တွေ method တွေ သိမ်းဖို့ Java Program ကနေ create လုပ်တဲ့ object တွေ သိမ်းဖို့ အတွက် memory area တွေကို အောက်ကအတိုင်း သတ်မှတ်ပေးထား ပါတယ်။ ပုံကတော့ အင်တာနက်ကနေ အဆင်ပြေတာ ကောက်ယူ ထားတာပါ။

JVM Runtime Data Area



- PC
- Java Virtual Machine Stack
- Heap
- Method Area
- Runtime Constant Pool
- Native method stack

ဆိုပြီး memory area တွေ ထားရမယ်လို့ JVM specification မှာ ပါပါတယ်။ အပိုင်း ၄ မှာပြခဲ့တဲ့ Object File layout ကတော့ဘယ်လိုထားရမယ် ဆိုတာ JVM မှာ မပါပါဘူး။ အဲ့ဒီကောင်ကတော့ ကိုယ်ကြိုက်တဲ့ပုံစံနဲ့ တည်ဆောက်နိုင်ပါတယ်။

PC

JVM မှာ thread အများကြီးကို run ခွင့် ပေးထားပါတယ်။ ဒါကြောင့် thread တခုချင်းဆီမှာ အဲ့ဒီ thread လက်ရှိ run နေတဲ့ method ရဲ့ byte code line ကို မှတ်ထား ရပါတယ်။ ဒါကို PC (program counter) လို့ ခေါ်ပါတယ်။ Thread တခုချင်း ဆီအတွက် PC တခုချင်းဆီ ရှိနေမှာပါ။ တကယ်လို့ လက်ရှိ run နေတဲ့ method က native (Java method မဟုတ်ဘဲ JNI သုံးထားတဲ့ C++ method) ဖြစ်နေရင်တော့ PC က undefined ဖြစ်နေမှာပါ။

JAVA VIRUTAL MACHINE STACK

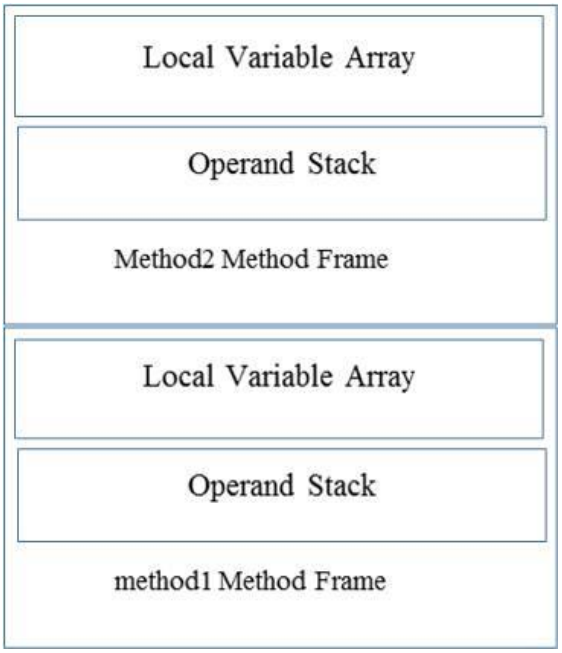
Java, C ,C++,C# လို language တွေမှာ method တွေ create လုပ်လို့ရပါတယ်။ အဲ့ဒီ method တွေကနေ တခြား method တွေကို လှမ်းခေါ်လို့ ရပါတယ်။ ဥပမာ method method1() ကနေ method2() ကို လှမ်းခေါ်တယ်ဆိုပါစို့ method2 က ပြီးသွားရင် method1() ဆီပြန်လာရပါတယ်။ ဒါမျိုးကိုထိန်းဖို့ အတွက် method frame တွေကို သိမ်းဖို့အတွက် calling sequence (method တခုကနေ ဘယ် method ကိုခေါ်သလဲ) ကိုထိန်းဖို့အတွက် Jav Virtual Machine Stack ကိုသုံးပါတယ်။

JVM stack မှာ method တခုအတွက် method frame တွေကို stack အနေနဲ့သိမ်းပါတယ်။ ဘာလို့ stack အနေနဲ့သိမ်းလဲ ဆိုတော့ ဆိုပါစို့... method1 ကိုအရင်ခေါ်တယ်... ဒါဆို stack ပေါ်မှာ method1 ရဲ့ method frame ကိုတင်ပါတယ်။ နောက် method1 ကနေ method2 ကိုခေါ်ရင် stack ပေါ်မှာ method2 ရဲ့ method frame ကို တင်ပါတယ်။ ဒါဆို method2 က return ဆိုတဲ့ bytecode ကို execute လုပ်ရင် သူ့ကိုခေါ်တဲ့ method1 ဆီပြန်သွားရမှာပါ။

အဲဒါကို Virtual Machine မှာဘယ်လို implement လုပ်လဲဆိုတော့ method တွေ run ပြီးသွားရင် return byte code တွေ့ရင် JVM Stack ပေါ်ကနေ pop လုပ်ချပါတယ်။ ဒါဆို stack ထိပ်ဆုံးမှာ ကျန်ခဲ့တဲ့ကောင်က ခုနကပြီးသွားတဲ့ method ကိုခေါ်တဲ့ method frame ပါပဲ။ အဲ့ဒီတော့ JVM stack ဆိုတာ method တခုချင်းဆီကို run တိုင်း JVM stack ပေါ်မှာ method frame တခုထပ်ထပ်တင်ရပါတယ်။ ဒါကြောင့် တချို့ recursive method တွေမှာ method တွေ ခဏ ခဏ ခေါ်လွန်းရင် JVM stack က ပြည့်လာပြီး StackOverflowError တက်ပါလိမ့်မယ်။

အောက်က JVM stack က method1 ကနေ method2 ကိုခေါ်ထားတဲ့ အခြေအနေမှာ method2 က run နေတုန်း ပုံကိုပြထားတာပါ။

JVM Stack



Method frame

JVM က method တခုကို invoke လုပ်တိုင်း method frame တခုကို တည်ဆောက်ပြီး JVM stack ပေါ် တင်ရပါတယ်။ Method frame ထဲမှာ ဘာတွေပါလဲ ဆိုတော့ ဒီ method ကို execute လုပ်ဖို့ အတွက် parameter တွေ local variable တွေကို သိမ်းဖို့ လိုပါတယ်။ ဒါတွေကို method frame ထဲမှာ သိမ်းပါတယ်။ Parameter တွေရယ် local variable တွေရယ် ကို ပေါင်းပြီး local variable array ဆိုတဲ့ word array ထဲမှာ သိမ်းပါတယ်။

static မဟုတ်တဲ့ method တွေဆိုရင် ခု run နေတဲ့ current object ဖြစ်တဲ့ this ကို local variable array ရဲ့ 0 အခန်းမှာ သိမ်းပါတယ်။ byte code instruction တွေက local variable တွေကို နာမည် မှတ်မထားပါဘူး... compiler ကနေ ဘယ် local variable သည် local variable array ရဲ့ ဘယ် index မှာ သိမ်းတယ်ဆိုတာပဲ မှတ်ထားတာပါ။

Operand stack ဆိုတာကတော့ method ထဲက byte code တွေရဲ့ instruction တွေ expression တွေ ကို execute လုပ်တဲ့အခါ လိုတဲ့ stack ပါပဲ။ Operand stack ကိုသုံးပြီး JVM instruction တွေကို execute လုပ်လို့ JVM ကို stack based machine လို့ဆိုကြတာပါ။ Operand stack က maximum ဘယ်လောက်ယူရမယ် ဆိုတာကိုလဲ compiler ကနေ တွက်ထုတ်ပေးပါတယ်။ အဲ့ဒီအကြောင်းကို class file format အပိုင်းမှာ ရေးထားပါတယ်။

ဆိုပါစို့ အောက်က java statement ပေါ့
a = 10 +2
သူ့ကို Byte code instruction ပြောင်းရင် ဒီလိုပုံစံရပါလိမ်မယ်။ အတိအကျ မဟုတ်ပါဘူး JVM မှာ instruction တွေကို type ပေါ်မူတည်ပြီး ခွဲပါသေးတယ် ဥပမာ integer load နဲ့ double load က မတူပါဘူး။

load 10
load 2
add store a
(တကယ်က ဒီနေရာမှာ store a လို့ မရေးပါဘူး byte code မှာ a ရဲ့ local variable index နံပါတ်ကိုပဲ ညွှန်းမှာပါ... store 1 ဆိုပါစို့... local variable က တလုံးတည်း ဆိုရင် 0 အခန်းက this ယူသွားတော့ a က 1 ဖြစ်မှာပါ)
ခုနက instruction 4 ကြောင်းကို execute လုပ်မယ်ဆိုရင် ပထမဆုံး load 10 ဆိုရင် operand stack အပေါ်ကို 10 ကိုတင်ပါတယ်။ operand stack က [10] ဒီလို ဖြစ်နေပါပြီ။ နောက် load 2 ဆိုတော့ operand stack [10,20] ဖြစ်နေပါပြီ add ဆိုလို့ရှိရင် stack ပေါ်က ၂ ခုကို pop လုပ်ပြီး ပေါင်းပါတယ်။ stack ပေါ်ပြန်တင်ပါတယ်... ဒါဆို stack ပေါ်မှာ [30] ဖြစ်သွားပါပြီ ။ အဲ့ဒါကို store 1 ဆိုပါစို့... ဒါဆိုရင် ခုနက operand stack ထိပ်ဆုံးက 30 ကိုယူပြီး local varible အခန်းနံပါတ် 1 မှာသွားသိမ်းပါတယ်။

Heap

Java program တွေ မှာ object creation , array creation တွေ ပါလေ့ရှိပါတယ်။ အဲ့ဒီ object instance တွေ array object တွေကို heap ဆိုတဲ့ memory area မှာသိမ်းပါတယ်။ Stack က အထုတ်အသွင်းကို ထိပ်ဆုံးကနေပဲလုပ်ပါတယ် ရိုးရှင်းပါတယ် stack ကိုထိန်းရတာက။ Heap ကျတော့ နေရာလွတ်တဲ့နေရာကနေ object တခုစာကိုယူပါတယ် ။နောက် အဲ့ဒီ object က အသုံးမလိုတော့ဘူး ဆိုရင် ခုနကနေရာကို empty ဖြစ်လိုက်ပြီလို့မှတ်လိုက်ပါတယ်။ ဒါကြောင့် heap management က ရှုပ်ပါတယ်။ Dynamically allocated လုပ်ထားတဲ့ object တွေကို heap မှာသိမ်းပါယ်။ နောက်အဲ့ဒီ object တွေ အသုံးမလိုလာတဲ့အခါ heap ကိုပြန် compact ဖြစ်အောင် ခုနက မသုံးတော့တဲ့ object နေရာကို နေရာလွတ်ဖြစ်အောင် ဖန်တီးရပါတယ်။ ဒါတွေကို Garbage Collection Algorithm တွေက လုပ်ပါတယ်။ JVM Spec မှာတော့ ဘယ်လို GC algorithm သုံးရမယ်ဆိုတာမပါပါဘူး။ Oracle ကတော့ generational GC ကိုသုံးပါတယ်။ သူ့အကြောင်းက သက်သက်ရေးမှ ပြည့်စုံမှာပါ။

Method area

Java class တွေရဲ့ method တခုချင်းစီက instruction တွေ ကို method area မှာသိမ်းပါတယ်။ Constructor တွေ ရိုးရိုး method တွေ constant pool တွေ field တွေအစရှိတဲ့ class တခုရဲ့ information တွေ အကုန်လုံးကို method area မှာ သိမ်းပါတယ်။

Method Area သည် Heap ရဲ အစိတ်အပိုင်း တခုသာ ဖြစ်ပါတယ်။ ဆိုချင်တာက compile code တွေကို heap ရဲ့ method area ဆိုတဲ့ section မှာ သိမ်းထားတာ ပါပဲ။ ဒီနေရာမှာ method တခုရဲ့ code က byte code ဖြစ်နေနိုင်သလို JVM ကနေ ထပ်ပြီး byte code ကို native machine code အနေနဲ့ ပြောင်းထားတာလဲ ဖြစ်နိုင်ပါတယ်။

Class တွေ interface တွေရဲ့ internal implementation detail ကို method area မှာ သိမ်းတယ်လို့ပြောလို့ ရပါတယ်။ နာမည်အရသာ method area လို့ ခေါ်တာပါ။ တကယ်က class representation, field, method, constant pool အကုန် သိမ်းပါတယ်။

Runtime constant pool

JVM က dynamic linking ကို သုံးပါတယ်။ ဒါကြောင့် class တခုဟာ တခြား class တခုရဲ့ method ဖြစ်ဖြစ် field ဖြစ်ဖြစ် ခေါ်သုံးတယ်ဆိုရင် ဒါတွေကို symbolic reference အနေနဲ့ သိမ်းပါတယ်။ အဲ့ဒီ method reference, field reference, class reference တွေရယ် string ,number, boolean constant အစရှိတာတွေကို byte code file ရဲ့ constant pool ဆိုတဲ့ နေရာမှာသိမ်းပါတယ်။ Constant Pool အကြောင်းကို အရင် bytecode file format တွေရှင်းတုံးကရှင်းခဲ့ပြီးပါပြီ။ အဲ့ဒီ byte code မှာပါတဲ့ constant pool ကိုသိမ်းဖို့ အတွက် runtime constant pool ဆိုတဲ့ memory area ကို သိမ်းပါတယ်။ Runtime contant pool ကိုတော့ method area ထဲက class တခုချင်းဆီရဲ့ representation တိုင်းမှာ သိမ်းပါတယ်။

Native method stack

Java API တွေကို ဘယ်လို implement လုပ်ထားလဲဆိုတော့ တချို့ method တွေက Java နဲ့ ရေးထားပါတယ်။ တချို့ method တွေကျတော့ native implementation ကို လှမ်းခေါ်ရပါတယ်။ ဒီနေရာမှာ native implementation ဆိုတာ C/C++ နဲ့ ရေးထားတဲ့ method တွေကိုပြောတာပါ။ Java program တွေက native method, native method ကနေ OS system call အဲ့ဒီလိုသွားပါတယ်။

အခုတော့ java method တွေက နေ native method ကိုလှမ်းခေါ်ရင် သုံးဖို့အတွက် JVM stack လိုပဲ native method stack ဆိုတာကို သုံးရပါတယ်။ Native method stack ကို C stack လို့လည်း ခေါ်ပါသေးတယ်။ သူ့ကို implement လုပ်တဲ့အခါကျရင်တော့ သူ့ရဲ့ stack frame သည် C stack frame ရဲ့ format ကို လိုက်နာရပါတယ်။ ဒါမှ C/C++ method တွေကိုခေါ်လို့ အဆင်ပြေမှာဖြစ်ပါတယ်။ ဆက်ရန် ☺

HOW DOES JVM WORK PART-6

BYTE CODE INSTRUCTION

Execution Engine ဆိုတာကတော့ Java ရဲ့ byte code တွေကို execute လုပ်ပေးတဲ့ ကောင်ပါပဲ။ သူ့ကို computer တွေရဲ့ CPU လို့ တင်စားရမှာပါ။ Machine ရဲ့ CPU နဲ့ ကွာတဲ့ အချက်ကတော့ သူက native instruction တွေအစား java byte code instruction တွေကို execute လုပ်တာပါပဲ။ Executed လို့ သုံးတာက interpretation လုပ်တာ ဖြစ်နိုင်သလို JIT compile လုပ်ပြီး compilation လုပ်တာလဲ ဖြစ်နိုင်လို့ပါ။ Oracle HotSpot VM မှာဆိုရင် interpretation ရော compliation ကိုရော တွဲလုပ်ပါတယ်။

Interpretation

Byte code တွေကို တိုက်ရိုက် ဖတ်ပြီး byte code ရဲ့ instruction အတိုင်း native instruction တွေ အပြောင်းပဲ directly execute လုပ်ပေးတာကို interpretation လို့ဆိုရမှာပါ။ ရိုးရိုးရှင်းရှင်းနဲ့ interpreter ကို မြင်အောင်ပြရရင်တော့ swtich based interpreter ကို ပြရပါလိမ့်မယ်။ အောက်က code example လို အလုပ်လုပ်ပါတယ်။ opcode ပေါ်မူတည်ပြီး လုပ်ရမဲ့ အလုပ်ကို တခါတည်း တန်းလုပ်ပါတယ်။

```
switch(opcode)
{
        case iadd:
                int opTwo = operandStack.pop();
                int opOne = operandStack.pop();
                operandStack.push(opTwo+opOne);
        break;
        .
        .
        .
}
```

ဒီနေရာမှာ JVM က stack based machine ဆိုတာကို သတိရဖို့ လိုပါတယ်။ ဒါက ဘာကို ဆိုလိုတာလဲ ဆိုတော့ ခုနက switch based interpreter မှာ ဥပမာ ပြထားတဲ့ iadd ဆိုတဲ့ instruction ရဲ့ အဓိပ္ပာယ်က integer ၂ခုကိုပေါင်းမယ်... ပြီးတော့ stack ပေါ်ကို result ကိုတင်မယ်လို့ ဆိုလိုတာပါ။
Machine language ဒါမှမဟုတ် assembly language တွေမှာဆိုရင် add instrution ဟာ operand 2 ခု လိုက်မှာပါ။ JVM မှာကျတော့ opcode တခုတည်းနဲ့ လောက်ပါတယ်။ iadd (ပေါင်းမယ်ဆိုရင်) operand တွေသည် stack ပေါ်မှာရှိပြီးသားလို့ ယူဆလို့ရပါတယ်။
Operand Stack က ဘယ်မှာရှိသလဲဆိုတော့ Method Frame ရဲ့ အထဲမှာရှိပါတယ် ဒါကို အရင်က အပိုင်း ၅ runtime data area မှာ ပြန်ကြည့်လို့ ရပါတယ်။
Interpretation ရဲ့ ကောင်းတဲ့အချက်ကတော့ implementation ဟာ ရိုးရှင်းလွယ်ကူပါတယ်။ Java က စထွက်တုန်း က အစောပိုင်းတုံးက interpretation ကိုပဲ သုံးပါတယ် ။ မကောင်းတာကတော့ သူက တော်တော်နေးပါတယ်။ ဒါကြောင့်နောက်ပိုင်းမှာ compilation နဲ့ တွဲသုံးပါတယ်။

Compilation

Compilation က ဘယ်လိုအလုပ်လုပ်သလဲ ဆိုတော့ ခုနက method တခုထဲမှာ run ရမဲ့ code block တွေ ရှိပါတယ်။ အဲ့ဒီ code block တွေကို native machine code အနေနဲ့ ပြောင်းပြီး byte array ထဲမှာ သိမ်းထားပါတယ် (native machine code ဆိုတာကလဲ အမှန်တော့ just a binary string ပါပဲ) ။ ပြီးတော့မှာ ခုနက byte array နဲ့ သိမ်းထားတဲ့ native code ကို C++ ကနေ void function pointer အနေနဲ့ typecast လုပ်ပြီး function တခုလို call လုပ်ပါတယ်။ ဒါဆိုရင် ခုနက native machine code က ထ run မှာပါ။
တခု သတိထားရမှာက ခုနက native machine code ကို OS က run ဖို့ ခွင့်ပေးထားတဲ့ memory area မှာ allocated လုပ်ထား ရမှာပါ။ ဘာလို့ လဲဆိုတော့ OS ကနေ read-write ရတဲ့ memory area ကနေ instruction တွေကို execute လုပ်ခွင့် ပေးမှာမဟုတ်လို့ ပါပဲ။ Compilation ရဲ့ အားသာချက်ကတော့ သူကမြန်ပါတယ်။ ဒါပေမဲ့ သူ့ကိုယ် သူမှာဆိုရင် byte code ကို native code ဖြစ်အောင် အရင် compile လုပ်ရပါတယ်။ ဒီတော့ တကယ် ကြာတဲ့အချိန်သည် compile time+ execution time ဖြစ်သွားပါတယ်။
ဒါကြောင့် ရှိသမျှ method တွေကို တခါတည်း compile မလုပ်ပဲ interpreter လေးနဲ့အရင် run နောက်မှ သုံးတာ များလာတော့မှာ hot spot ဖြစ်လာတော့မှ compile လုပ်ပြီး native code အနေနဲ. Run ပါတယ်။ ဒါကြောင့် Oracle ကထုတ်တဲ့ JVM ကို hot spot လို့ ခေါ်ရခြင်းပါ။ ဆိုချင်တာက method တိုင်း ကို JIT compilation မလုပ်ပဲ hot ဖြစ်တဲ့ method (treshold တခုလောက် အတိုင်း အတာထိ run ရတဲ့) method ကိုမှ compile လုပ်လို့ပါ။

Jvm instruction set

JVM byte code instruction set ကို ဒီလိုပုံစံနဲ့ format ချထားပါတယ်။

mnemonic
operand1
operand2

menomoic ဆိုတာ စာနဲ့ ဖတ်ရလွယ်တဲ့ opcode ပါ ဥပမာဆိုရင် iadd ပေါ့ ။ တကယ်က iadd သည် byte code instruction opcode ဖြစ်တဲ့ number တခုပါပဲ။ Hexa နဲ့ ဆိုရင် iadd ရဲ့ number သည် 60 ပါ။ နောက်က operand1 operand2 ဆိုတာ တွေကတော့ operand တွေပါ ။ တချို့ opcode တွေက operand တခုမှ မလိုက်တာလဲဖြစ်နိုင်ပါတယ်။ JVM မှာ byte, short, int , long ,float, double, char နဲ့ object reference type ဆိုပြီး ခွဲထားပါတယ် instruction တွေကို… instruction တွေက type အပေါ်မူတည်ပြီးကွဲပါတယ် ဥပမာ iadd ဆိုတာ integer add ကို သုံးတာ ဖြစ်ပြီး dadd ဆိုတာကတော့ double ၂ ခုကို add လုပ်တာပါ။ Byte code တွေကို Category အရ ခွဲမယ်ဆိုရင် ဒီလိုရမှာပါ။

Pushing constants onto the stack

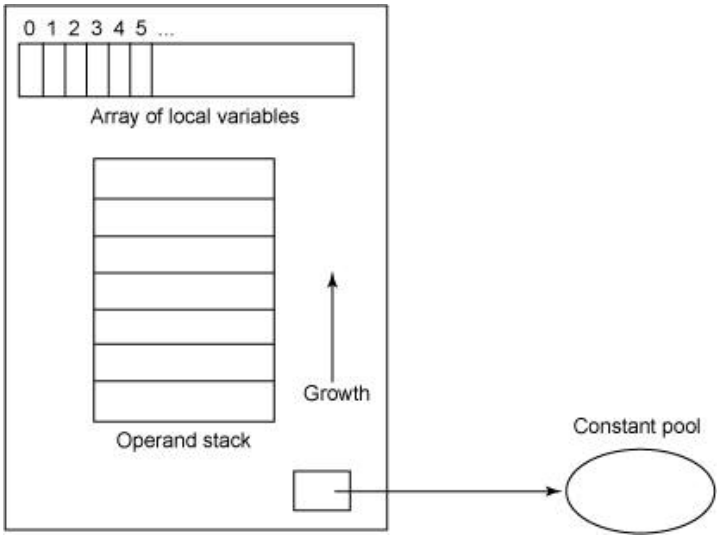
iconst_0, dconst_0 ဒီကောင်တွေ ကတော့ integer constant 0 ကို operand stack ပေါ်တင်မယ်။ dconst_0 ဆိုတာ ကတော့ operand stack ပေါ်တင်မယ် ဆိုတဲ့ constant တွေကို operand stack ပေါ်တင်တဲ့ ကောင်တွေပါ။ ဒီနေရာမှာ numerical type တွေ အတွက် အကုန် ရှိတယ်လို့ ယူဆရမှာပါ။ Constant Pool ထဲမှာရှိတဲ့ constant တွေကို operand stack ပေါ် တင်ချင်ရင်တော့ အောက်က opcode မျိုးကို သုံးပါတယ်.

ldc1 constant_pool_index , lcd1 ဆိုတာက constant pool ထဲက 32 bit constant ကို operand stack ထဲ တင်မယ်လို့ ဆိုချင်တာပါ။ ဒီ instruction group ကတော့ constant literal တွေပါတဲ့ expression တွေကို execute လုပ်ရမှာ သုံးပါတယ်။

Pushing local variables onto the stack

ဒီ Group ကတော့ local variable တွေကို operand stack ပေါ် တင်တာပါ။ local variable တွေ ပါတဲ့ expression တွေကို execute လုပ်တဲ့အခါ ဒီ group ကို သုံးပါတယ်။ local variable တွေကို method frame ရဲ့ local variable array ထဲမှာ သိမ်းထား ပါတယ်။

ဥပမာ iload_0 ဆိုရင် local variable array index 0 မှာ ရှိတဲ့ integer ကို stack ပေါ် တင်မယ်ဆိုတဲ့ အဓိပ္ပာယ် ရပါတယ်။ aload_0 ဆိုရင်တော့ (byte code menomoic တွေမှာ object reference ကို a နဲ့ သုံးပါတယ်) အခန်းနံပါတ် 0 မှာ ရှိတဲ့ object reference type အမျိုး အစား variable ကို operand stack ပေါ် တင်မယ်လို့ ဆိုချင်တာပါ။ အောက်ကပုံမှာ method frame တခုကို ပြထားပါတယ်။ local variable array နဲ့ operand stack ကိုပါ ပြထားပါတယ်။ တခု သိထားရမှာက instance method (non-static) method တွေဆိုရင် current object this ဆိုတဲ့ reference ကို local variable array ထိပ်ဆုံးခန်း 0 မှာ သိမ်းတယ် ဆိုတာပါပဲ။ field access တွေ method call တွေဆိုရင် ပထမဆုံး field access or method call ခေါ်မဲ့ object ကို operand stack ပေါ် အရင်တင်ရပါတယ်။



Popping to local variables

ဒီ Group ကတော့ operand stack ရဲ. ထိပ်ဆုံးမှာရှိတဲ့ value ကို local variable array ထဲကိုသိမ်းတာပါ။ ဥပမာ istore_0 ဆိုရင် operand stack ထိပ်ဆုံးက integer ကို pop လုပ်ပြီးတော့ local variable အခန်း 0 မှာသွားသိမ်းမှာပါ။ အခန်း 0 မှာသိမ်းတဲ့အတွက် အဲ့ဒီ method ဟာ static method ဖြစ်မှာပါ။ ဥပမာ..

```
int a = 10 +2
```

ဒီ Statement ကို byte code ထုတ်ရင်

```
iload 10        // အမှန်က contant pool index နဲ့ သုံးမှာပါ
iload 2
iadd istore_0 ဆိုပြီး ထုတ်ပါလိမ့်မယ်။
```

ဒါဆိုရင် stack ပေါ်ကို iload_10 ဆိုရင် 10ကိုတင် နောက်တကြောင်း iload 2 ဆိုရင် 2 ကိုတင်... iadd ဆိုတော့ stack ပေါ်က 1၀ နဲ. 2 ကို pop လုပ် ပေါင်း အဖြေ 12 ကို stack ပေါ်ပြန်တင်... နောက်ဆုံး istore_0 ဆိုတော့ stack ပေါ်က 12 ကို local variable a အတွက် အခန်း 0 မှာသွားသိမ်း အဲ့လို လုပ်တာပါ။ Assignment statement တွေဆိုရင် ဒီ bytecode group ကိုသုံးရမှာပါ။

Type conversions

ဒီ group ကတော့ Java language မှာ ပေးထားတဲ့ type casting, type promotion တွေလုပ်ဖို့အတွက် သုံးတာပါ။ ဥပမာ integer ကနေ float ကိုပြောင်းချင်ရင် i2f ဆိုတဲ့ byte code ကို သုံးပါတယ်။ ဘယ်လိုအလုပ်လုပ်လဲ ဆိုတော့ operand stack ပေါ်မှာရှိတဲ့ integer ကို pop လုပ်မယ် နောက် float ကိုပြောင်းမယ် operand stack ပေါ်ပြန်တင်မယ်ပေါ့ ခင်ဗျာ။ Automatic type promotion တွေ type cast တွေ coercion တွေအတွက်ဆို ဒီ group ကိုသုံးပါတယ်။

Arithemtic

Java မှာရှိတဲ့ arithmetic operation တွေ အားလုံးအတွက် ဒီ opcode group ကို သုံးပါတယ်။ တခုသိထားရမှာက Java Langauge မှာသာ byte+int လိုမျိုး မတူတာကို ပေါင်းလို့ရပေမဲ့ java compiler ကနေ byte code ထုတ်ရင် byte ကိုအရင် int ပြောင်းတဲ့ promotion operation ကိုလုပ်ပါတယ် ပြီးမှ integer နဲ့ပေါင်းပါတယ်။ အဲ့လို type တွေကို widening လုပ်ပြီးမှ arithmetic opcode တွေကို လုပ်တယ်လို့ မှတ်ထားရမှာပါ။ opcode ဥပမာတွေကတော့... iadd, idiv,imul တို့ပါ ။ iadd ဆိုရင် operand stack ပေါ်က integer ၂ခုကို pop လုပ်မယ်... ပေါင်းမယ်... ပြီးရင် result ကို operand stack ပေါ်ပြန်တယ်မယ် ဒီလို အလုပ်လုပ်ပါတယ်။

Logic

ဒီ Group ကတော့ logic operation တွေလုပ်ရတာပါ။ boolean operation တွေဖြစ်တဲ့ and or not တွေအတွက် သုံးတဲ့ byte code ပါ။ ဥပမာကတော့ iand, ior,ixor တို့ပါပဲ။ iand ဆိုရင် operand stack ပေါ်မှာရှိတဲ့ integer 2 လုံးကို pop လုပ်ချမယ်... ပြီးတော့ and operation လုပ်မယ် (ဒီနေရာမှာ JVM မှာ boolean မရှိတဲ့အတွက် boolean အစား integer ကိုသုံးတယ်လို့ ဆိုရမှာပါ) ပြီးတော့ ရလာတဲ့ result ကို operand stack ပေါ်တင်မယ်။ ဒီလိုအလုပ်လုပ်ပါတယ်။

Object and Array Operation

ဒီ group ထဲမှာပါတဲ့ opcodeတွေကတော့ object creation , field access,array creation, array member access လုပ်တဲ့ instruction တွေအတွက်ပါ။ object ဆောက်ရင်သုံးတဲ့ opcode ကတော့ anew ပါ။ သူ့ format က ဒီလိုပါ။

anew index1,index2

index1 နဲ. index2 ၂ခု ပေါင်းက constant pool ရဲ့ ClassRef ကိုညွှန်းမှာပါ။ အဲ့ဒီ opcode ကိုလုပ်ဖို့ ဆိုရင် constant pool index က class ရဲ့ object ကို heap ပေါ်မှာဆောက်ပြီးတော့ object ရဲ့ reference ကို operand stack ပေါ်ကိုတင်မှာပါ။ နောက် putfield,getfield ဆိုတဲ့ opcode ၂ခုကတော့ object တခုရဲ့ field ထဲကို operand stack ပေါ်က တန်ဖိုးကို ထည့်ပေးတာရယ်။ object ရဲ့ field ကို operand stack ပေါ်တင်တာရယ်ကိုလုပ်တာပါ။
object field access ပါတဲ့ expression တွေမှာ putfield,getfield ကိုသုံးရပါတယ်။
putfield,getfield က instance variable တွေအတွက် သုံးတာပါ။ သူတို့ကို မခေါ်ခင်မှာ operand stack ပေါ်မှာ object reference ရှိနေရပါလိမ့်မယ်။ static field တွေကို reference လုပ်မယ် ဆိုရင်တော့ putstatic, getstatic ဆိုတဲ့ opcode တွေကို သုံးပါတယ်။ သူတို့ကိုခေါ်မယ် ဆိုရင်တော့ အဲ့ဒီ opcode တွေရဲ့ နောက်မှာ ClassRef ကို ကိုယ်စားပြုတဲ့ constant pool index ကို ပေးရမှာပါ။
Array operation တွေအတွက်ကတော့ ရိုးရိုး one dimensional array ဆောက်ချင်ရင် newarray ဆိုတဲ့ opcode ကိုသုံးပါတယ်။

Control flow

ဒီကောင်တွေကတော့ Assembly မှာလို condition တခုကိုစစ်ပြီး byte code ဘယ် index ကို သွားဆိုပြီးခိုင်းတဲ့ control flow opcode တွေပါ။ Java ရဲ့ for, while, do while, break အစရှိတဲ့ control structure တွေကို compile လုပ်ဖို့သုံးပါတယ်။ နမူနာပြရရင်တော့

if_icmpeq branchbyte1, branchbyte2

အပေါ်က byte code က operand stack ပေါ်က integer ၂ခုကို တန်ဖိုးအားဖြင့် တူသလားလို့ စစ်ပါတယ်... တူခဲ့ရင် branchbyte1 ကညွှန်တဲ့ byte code index ကို PC ကိုရွှေ့မှာပါ (if ရဲ့ then part ပေါ့) မဟုတ်ရင်တော့ branchbyte2 ကညွှန်တဲ့ bytecode index ကို PC ကိုရွှေ့ပါလိမ့်မယ်။

Method invocation and return

ဒီကောင်တွေကတော့ method call တွေ return statement တွေ ကို compile လုပ်တဲ့အခါ သုံးပါတယ်။

invokevirtual indexbyte1, indexbyte2 (Virtual method တွေခေါ်ရင် သုံးပါတယ်)
invokestatic indexbyte1, indexbyte2 (Static method တွေ ခေါ်ရင်သုံးပါတယ်)
invokespecial indexbyte1, indexbyte2 (Constructor call အတွက်ပါ) invokeinterface indexbyte1, indexbyte2 (Interface ကနေ တဆင့်ခေါ်တဲ့ method တွေပါ)

နောက်ထပ် JVM မှာ ထပ်ဖြည့်ထားတာကတော့ dynamic language တွေအတွက် invokedynamic ဆိုတဲ့ opcode ပါ။ invokevirtual ကိုခေါ်မယ်ဆိုရင် ပထမ ဆုံးoperand stack ပေါ်မှာ object instance ကိုတင်ရပါတယ်။ နောက်ပြီး parameter တွေကို အစဉ်အလိုက် operand stack ပေါ်မှာတင်ရပါတယ်။ အဲ့လိုပြီးမှ invokevirtual ဆိုတဲ့ opcode ကိုခေါ်ရပါတယ်။

invokevirtual နောက်မှာလိုက်တာကတော့ constant poool entry MethodRef ဖြစ်တဲ့ကောင်ရဲ့ index ပါ။ ကျန်တဲ့ method call တွေကလဲ ထိုနည်းလည်းကောင်းပါပဲ။ JVM က method call ကို execute လုပ်ရင် ခုနက constant pool entry index မှာထောက်ထားတဲ့ method အတွက် method frame ကို allocate လုပ်ပါတယ်။ နောက် operand stack ပေါ်က parameter တွေကို အသစ်ခေါ်တဲ့ method frame ရဲ့ local variable array ထဲကို copy ကူးပါတယ်။
(ဒီနေရာမှာ local variable array သည် parameter တွေရော local variable တွေကိုပါ သိမ်းတယ်ဆိုတာ သတိထားရမှာပါ)။ နောက်အဲ့ဒါတွေပြီးသွားရင် PC ကို အခေါ်ခံရတဲ့ method ရဲ့ byte code index 0 ကို ညွှန်းပေးလိုက်ပါတယ်။ ဒါဆို JVM ကနောက် method ကိုအလုပ်စလုပ်တော့မှာပါ။
Return statement တွေအတွက်ကျတော့ ireturn(integer return), areturn (object return) အစရှိတဲ့ bytecode တွေကို သုံးပါတယ်။ ဥပမာ ireturn ကို တွေ့ပြီဆိုရင် လက်ရှိ run နေတဲ့ method ရဲ့ operand stack ထိပ်ဆုံးက integer value ကိုယူ... သူ့ အောက်က method frame ရဲ့ operand stack ထိပ်ဆုံးကို တင်လိုက်ပါတယ်။ ပြီးရင် return ပြန်တဲ့ method ရဲ့ method frame ကို Method stack ကနေ pop လုပ်ချရပါတယ်။ သူ့ကိုခေါ်ထားတဲ့ method frame ရဲ့ PC တန်ဖိုးကို လက်ရှိ PC တန်ဖိုးအနေနဲ့ပြောင်းထည့် ရပါတယ်။ ဒီလိုနည်းနဲ့ JVM က return ကို အလုပ်လုပ်ပါတယ်။

နောက် JVM မှာပါတဲ့ opcode group တွေက execption handling နဲ့ thread synchronization အတွက် opcode တွေပဲဖြစ်ပါတယ်။ အဲ့ကောင်တွေကိုတော့ ကို.ဘာသာပဲလေ့လာကြည့်ပါလို့ စာရှည်လို့ ခဏ ရပ်ပါရစေ ☺ ဆက်ရန် ☺

HOW DOES JVM WORK PART-7

Execution engine

အောက်ကပေးထားတဲ့ Java Program ရဲ့ byte code ကို execution engine က ဘယ်လိုအလုပ်လုပ်သွားသလဲ ဆိုတာ ကြည့်ရအောင်ပါ။ Program ကတော့ bytecode အကြောင်း ရှင်းလို့လွယ်အောင် ခပ်သေးသေးပေါ့။

```
public class Account {
   private double balance = 0;


   public void deposit(double amount) {
      balance += amount;
   }

   public static void main(String[] args) {
          Account account = new Account();
           account.deposit(1000);
          System.out.println("Balance = " + account.balance);

      }
}
```

အပေါ်က java program ကို compile လုပ်ပြီး javap -v Account ဆို ပြီး run လိုက်ရင် bytecode listing ကိုဒီလိုတွေ့ရမှာပါ။

```
Classfile /D:/Program Coding/Java/Account.class
  Last modified Aug 3, 2016; size 782 bytes
  MD5 checksum 19222ced0291e09d82a29b3239835909
  Compiled from "Account.java"
public class Account
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER

Constant pool:
   #1 = Methodref          #16.#29        // java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#30         // Account.balance:D
   #3 = Class              #31            // Account
   #4 = Methodref          #3.#29         // Account."<init>":()V
   #5 = Double             1000.0d
   #7 = Methodref          #3.#32         // Account.deposit:(D)V
   #8 = Fieldref           #33.#34        // java/lang/System.out:Ljava/io/PrintStream;
   #9 = Class              #35            // java/lang/StringBuilder
  #10 = Methodref          #9.#29         // java/lang/StringBuilder."<init>":()V
  #11 = String             #36            // Balance =
  #12 = Methodref          #9.#37         // java/lang/StringBuilder.append:(Ljava
/lang/String;)Ljava/lang/StringBuilder;
  #13 = Methodref          #9.#38         // java/lang/StringBuilder.append:(D)Ljava/lang/StringBuilder;
  #14 = Methodref          #9.#39         // java/lang/StringBuilder.toString:()Ljava/lang/String;
  #15 = Methodref          #40.#41        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = Class              #42            // java/lang/Object
  #17 = Utf8 balance
  #18 = Utf8 D
  #19 = Utf8 <init>
  #20 = Utf8 ()V
  #21 = Utf8 Code
  #22 = Utf8 LineNumberTable
  #23 = Utf8 deposit
```

```
 #24 = Utf8               (D)V
 #25 = Utf8               main
 #26 = Utf8               ([Ljava/lang/String;)V
 #27 = Utf8               SourceFile
 #28 = Utf8               Account.java
 #29 = NameAndType        #19:#20        // "<init>":()V
 #30 = NameAndType        #17:#18        // balance:D
 #31 = Utf8               Account
 #32 = NameAndType        #23:#24         // deposit:(D)V
#33 = Class               #43            // java/lang/System
#34 = NameAndType         #44:#45        // out:Ljava/io/PrintStream;
#35 = Utf8                java/lang/StringBuilder
#36 = Utf8                Balance =
 #37 = NameAndType      #46:#47      // append:(Ljava/lang/String;)Ljava
/lang/StringBuilder;
 #38 = NameAndType      #46:#48      // append:(D)Ljava/lang/StringBuilder;
 #39 = NameAndType      #49:#50      // toString:()Ljava/lang/String;
 #40 = Class          #51         // java/io/PrintStream
 #41 = NameAndType      #52:#53      // println:(Ljava/lang/String;)V
 #42 = Utf8           java/lang/Object
 #43 = Utf8           java/lang/System
 #44 = Utf8           out
 #45 = Utf8           Ljava/io/PrintStream;
 #46 = Utf8           append
 #47 = Utf8           (Ljava/lang/String;)Ljava/lang/StringBuilder;
 #48 = Utf8           (D)Ljava/lang/StringBuilder;
 #49 = Utf8           toString
 #50 = Utf8           ()Ljava/lang/String;
 #51 = Utf8          java/io/PrintStream
 #52 = Utf8           println
 #53 = Utf8           (Ljava/lang/String;)V
```

```
{

  public Account();

    descriptor: ()V
    flags: ACC_PUBLIC
    Code:

      stack=3, locals=1, args_size=1

         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: dconst_0
         6: putfield      #2                  // Field balance:D
         9: return
       LineNumberTable:
        line 1: 0
        line 2: 4
            public void deposit(double);
              descriptor: (D)V
              flags: ACC_PUBLIC
              Code:
                stack=5, locals=3, args_size=2
                   0: aload_0
                   1: dup
                   2: getfield      #2                  // Field balance:D
                   5: dload_1
                   6: dadd
                   7: putfield      #2                  // Field balance:D
                  10: return

                LineNumberTable:
                  line 5: 0
                  line 6: 10
```

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:

    stack=4, locals=2, args_size=1
       0: new           #3                  // class Account
       3: dup
       4: invokespecial #4                  // Method "<init>":()V
       7: astore_1
       8: aload_1
       9: ldc2_w        #5                  // double 1000.0d
      12: invokevirtual #7                  // Method deposit:(D)V
      15: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      18: new           #9                  // class java/lang/StringBuilder
      21: dup
      22: invokespecial #10                 // Method java/lang/StringBuilder."<init>":()V
      25: ldc           #11                 // String Balance =
      27: invokevirtual #12                 // Method java/lang/StringBuilder.append:(Ljava
/lang/String;)Ljava/lang/StringBuilder;

      30: aload_1
      31: getfield      #2                  // Field balance:D
      34: invokevirtual #13                 // Method java/lang/StringBuilder.append:(D)Ljava
/lang/StringBuilder;
         37: invokevirtual #14              // Method java/lang/StringBuilder.toString:()Ljava
  /lang/String;
         40: invokevirtual #15              // Method java/io/PrintStream.println:(Ljava
  /lang/String;)V
         43: return
       LineNumberTable:

         line 8: 0
         line 9: 8
         line 10: 15
         line 13: 43

  }
```

Bytecode file format နဲ့.ပတ်သတ်ပြီး အရင်အပိုင်းတွေမှာ ရှင်းပြီးသားပါ။ ခု အခဲ့ဒီ Account program ရဲ့ byte code ကို JVM ကနေ run တော့မယ်ဆိုရင်...

ပထမဆုံးလုပ်ရမှာက System ClassLoader တွေသုံးပြီး java.lang.Object လို ကောင်တွေကို အရင်ခေါ်တင်ရမှာပါ။ နောက်ပြီးတော့မှ run မဲ့ Account byte code file ကို ClassLoader တွေသုံးပြီး ခေါ်တင်ပါတယ်။ Runtime မှာလိုအပ်မဲ့ native implementation class တွေ object file layout တွေ... အစရှိတာတွေကို JVM ရဲ့ method area မှာ နေရာယူပြီး သိမ်းထားလိုက်ပါတယ်။ နောက် Class ကို loading, linking,verification လုပ်ပါတယ်။ အဲ့ဒီ အဆင့်တွေ ပြီးသွားရင်တော့ public static void main (String[]args) ကိုလိုက်ရှာပါတယ်။ Account byte code မှာရှိရင် သူ့ကို entry အနေနဲ့ run မှာပါ။

main method ရဲ့ byte code ကို အပေါ်က byte code listing ရဲ့ အောက်ဆုံးအပိုင်းမှာ တွေ့နိုင်ပါတယ်။ အောက်က listing တွေနဲ့စပါတယ်။

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:

    stack=4, locals=2, args_size=1
```

အဲဒီမှာ method ရဲ့ နာမည် ၊ public access modifier နဲ့ parameter တွေရယ် return type ရယ်ကို ပြထားပါတယ်။ အောက်က main method ရဲ့ descriptor ပါ။

descriptor: ([Ljava/lang/String;)V

သူ့ အဓိပ္ပာယ်ကတော့ java.lang.String object အမျိုးအစား array ဖြစ်တဲ့ parameter ကို လက်ခံတယ်။ Object ကို L နဲ့သတ်မှတ်ပါတယ်။ ဒါကြောင့် Ljava/lang/String ဆိုတာ java.lang.String object လို့သိရမှာပါ။ array ကိုကျတော့ [(square bracket) နဲ့ သတ်မှတ်ပါတယ်။ ဒါကြောင့် [Ljava/lanng/String သည် java.lang.String object ဖြစ်သော one dimensional array လို့ ဆိုရမှာပါ။ paranethesis () အဖွင့်အပိတ်က parameter လို့ဆိုချင်တာပါ။ ဒါကြောင့် main method ရဲ့ parameter သည် java.lang.String object one dimensional array လို့ အဓိပ္ပာယ်ရပါတယ်။ parameter သည် တခုတည်းဖြစ်တဲ့အတွက် ; (semicolon) နဲ့ အဆုံးသတ်ပါတယ်။

parameter paranethesis() နောက်ကလိုက်တာကတော့ return type ပါ။ ဒီနေရာမှာ () V ဆိုတာက void လို့ဆိုချင်တာပါ။ ဒါကြောင့် main method သည် void return ပြန်မယ်လို့ ရမှာပါ။ နောက်တကြောင်းမှာ အောက်က stack,locals,args size ဆိုတာတွေ့ရပါလိမ့်မယ်။

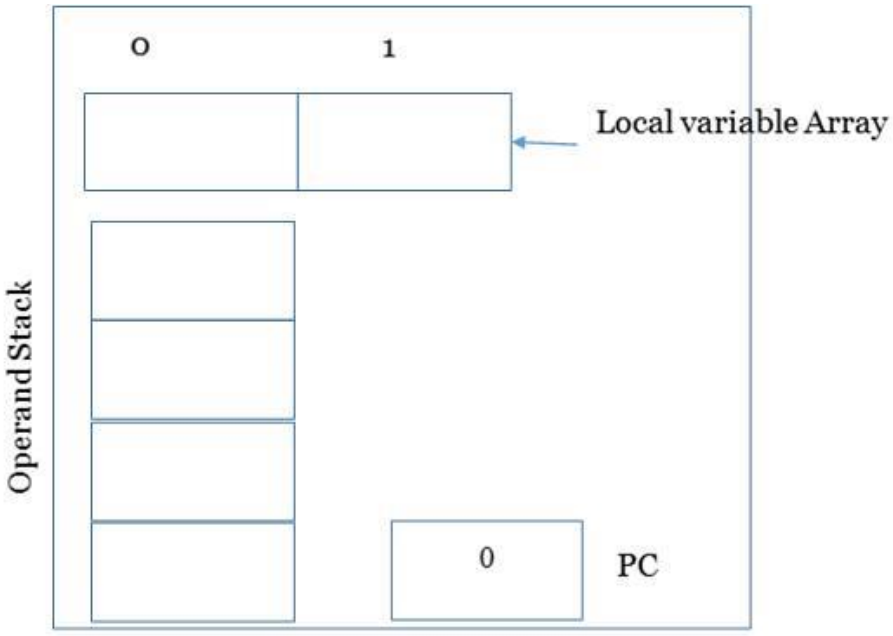
stack = 4, locals=2, args_size=1

stack = 4 ဆိုတာက main method ရဲ့ method frame ထဲမှာပါတဲ့ operand stack သည် maxixmum size 4 word ထိပဲ ယူမယ်လို့ဆိုချင်တာပါ။ locals သည် local variable + parameter တွေ အားလုံးပေါင်း ရဲ့ word size ပါ။

ဒီနေရာမှာ main method မှာ local variable အနေနဲ့ object type account ဆိုတဲ့ variable တလုံးရယ်၊ parameter ဖြစ်တဲ့ String[]args ဆိုတဲ့ array reference ရယ်ရှိပါတယ်။ Local variable အတွက် က 1 word parameter String[]args အတွက်က 1 word ဆိုတော့ locals သည် 2 word ယူရမှာပါ။ သူက array အနေနဲ့သိမ်းမှာပါ။ ဒါကြောင့် locals=2 ဖြစ်နေတာပါ။

နောက်တခုက args_size=1 ဆိုတာကတော့ရှင်းပါတယ်။ argument သည် 1 word စာသာနေရာယူမယ်လို့ဆိုလိုတာပါ၊ Local variable array ဆိုတာ method frame ထဲမှာပါတဲ့ parameter တွေရယ် local variable တွေရယ်သိမ်းထားတဲ့ one dimensional array ပါ။ parameter တွေကို array ရှေ့မှာသိမ်းပါတယ်။ parameter တွေကုန်မှ local variable တွေသိမ်းပါတယ်။ ဒီနေရာမှာ main သည် static method ဖြစ်တဲ့အတွက် current object this ကိုသိမ်းစရာမလိုပါဘူး။ မဟုတ်ရင် local variable 0 အခန်းမှာ this object ကိုသိမ်းရမှာပါ။ ဒါဆို ဒီ information တွေကြည့်ပြီးတော့ JVM ကနေ အောက်ကပုံလို method frame ကိုဆောက်လိုက်ပါပြီ။

METHOD FRAME OF MAIN METHOD



Local variable arry index 0 မှာ String[]args user က command promopt ကနေ ပေးတဲ့ parameter ကို သိမ်းပြီး main method ထဲက account ဆိုတဲ့ variable ကိုတော့ local variable array index 1 မှာ သိမ်းမှာပါ။ ဒီနေရာမှာ Compiler က byte code ထုတ်လိုက်တဲ့ အချိန်မှာ local variable တွေရဲ့ name တွေပျောက်သွားတယ်... သူတို့ အစား index နဲ့ပဲ သုံးတော့တယ်ဆိုတာ ကို သိထားရမှာပါ။ Main method က ခုမှ စ run မှာ ဖြစ်တဲ့အတွက် PC (Program Counter) က 0 ပါ။ main method ရဲ့ byte code instruction 0 ကို စထောက်ပြီး run ပါပြီ။ အောက်က code တွေကို run ပါမယ်...

```
 0: new           #3                  // class Account
 3: dup
 4: invokespecial #4                  // Method "<init>":()V
 7: astore_1
```

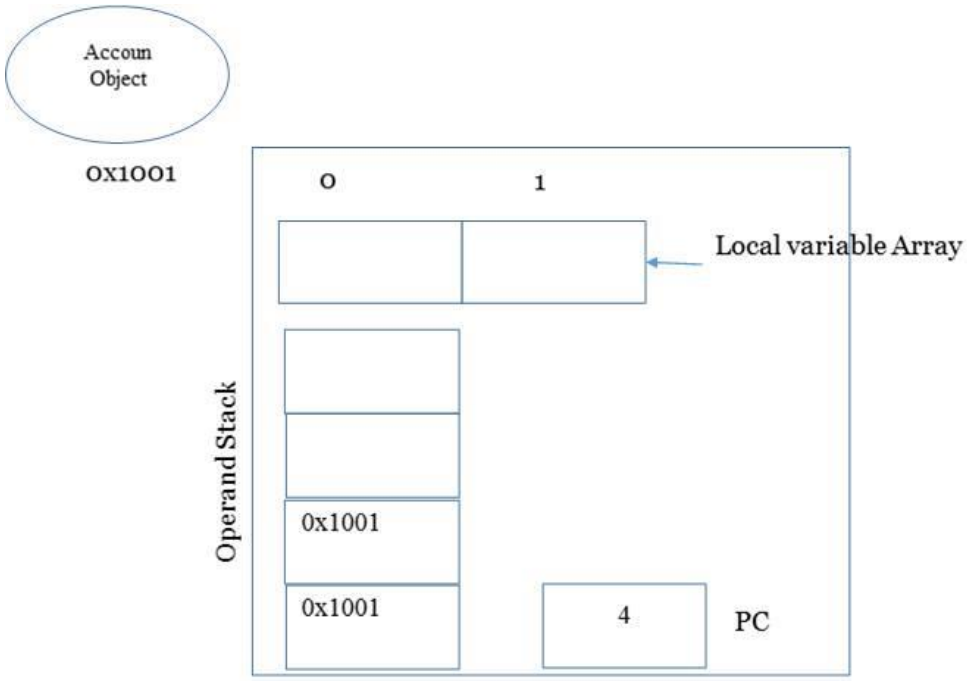
ပထမဆုံး

```
 0: new           #3                  // class Account
```

bye code 0 အခန်းက instruction သည် new ဖြစ်တဲ့အတွက် Heap မှာ Object ဆောက်ပါမယ်။ new #3 လို.ပြောတဲ့အတွက် ဆောက်ရမဲ့ object သည် constant pool entry index 3 ကနေညွှန်တဲ့ class ဖြစ်တဲ့ Account အမျိုးအစား object ကိုဆောက်ပါတယ်။ ဆောက်ပြီးတော့ ရလာတဲ့ reference ကို operand stack ပေါ်တင်ပါတယ်။ နောက်byte code index က 3 ပါ။

3: dup

ဆိုတဲ့အတွက် operand stack ပေါ်က ခုနက object refrence ကို copy ပွား လိုက်ပါတယ်။ ဒါဆို operand stack ပေါ်မှာ Account object ကို ကိုယ်စားပြုတဲ့ reference ကို ၂ခါ တင်ထားတာ ဖြစ်နေမှာပါ။ ခုဆို main ရဲ့ method frame က ဒီပုံစံဖြစ်နေပါပြီ။ ဒီနေရာမှာ Account Object က heap မှာနေရာယူပြီးသူ့ရဲ့ address က 0x1001 လို့ မှတ်ထားရမှာပါ။ တကယ်တမ်း JVM မှာက memory address မဟုတ်ပဲ reference handle ဖြစ်နေမှာပါ။ ရှင်းအောင်လို့ ဒီလိုရေးထားတယ်ပဲ မှတ်ထားပေးပါ။ ခုချိန်မှာ PC က 4 ကို ရောက်တော့ byte code index 4 ကို run ပါမယ်။



4: invokespecial #4 // Method "<init>":()V

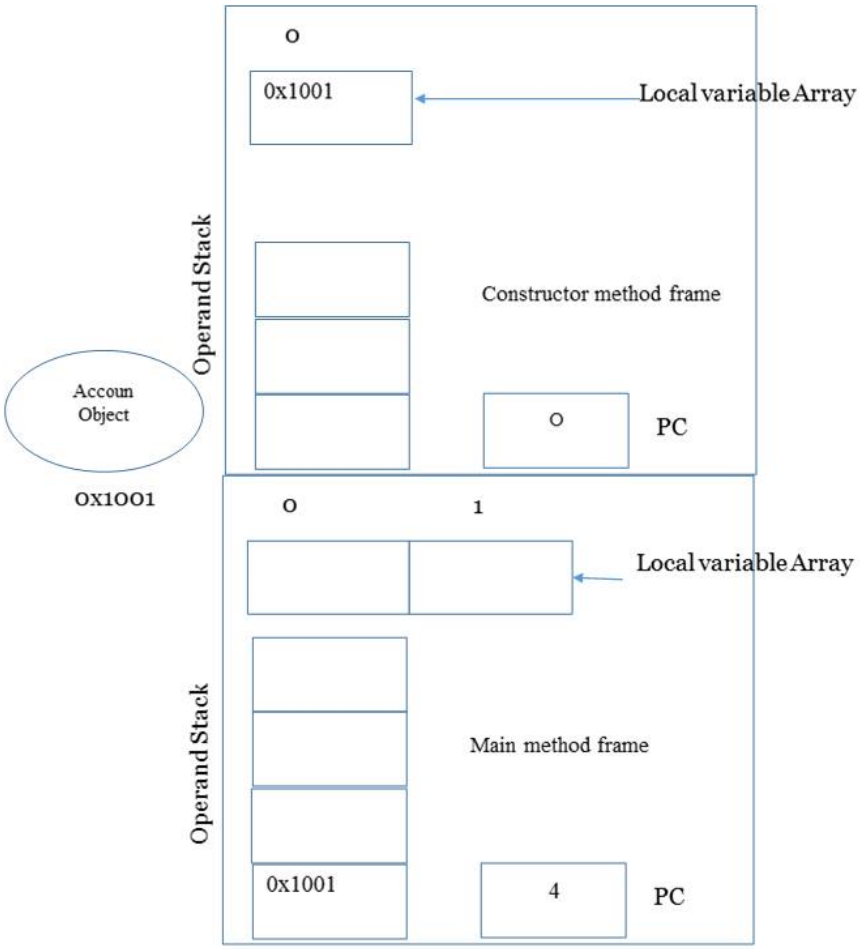
byte code index 4 ရဲ. instruction သည် invokespecial ပါ။ invokespecial ဆိုတာ constructor ကို ခေါ်တဲ့အခါ သုံးတာပါ။ သူ့နောက်က contant pool entry index 4 သည် <init>:()V လို့ ပြတဲ့အတွက် no argument constructor ကိုခေါ်မှာပါ။

ဒီနေရာမှာ Java language အရ ဘာ constructor မှ programmer က ပေးမထားဘူး ဆိုရင် no argument constructor တခုကို ဆောက်ပေးထားပါတယ်။ အောက်က byte code က Constructor အတွက်ပါ။ Constructor သည် no argument ဖြစ်ပေမဲ့ instance method အမျိုးအစား ဖြစ်တဲ့အတွက် မမြင်ရတဲ့ hidden object this ကို parameter အနေနဲ့ ပေးရမှာပါ။ အဲ့ဒီတော့ခုနက Main method frame ရဲ့ operand stack ထိပ်ဆုံးက ကောင်ကို နောက်အသစ်ဆောက်မဲ့ constructor method frame ရဲ့ parameter ထဲကို ထည့် ပေးရမှာပါ။ method call တွေ မခေါ်ခင် parameter တွေကို operand stack ပေါ်အရင်တင်ပေးထားရပါတယ်။ method call ခေါ်တဲ့အခါ method frame အဟောင်းရဲ့ operand stack ပေါ်ကတန်ဖိုးတွေနဲ့ method frame အသစ်(Constructor) ရဲ့ local variable အခန်းထဲကိုဖြည့်ပေးရပါတယ်။ အခုဘယ်လောက်ဖြည့်ပေးရမလဲဆိုရင် အောက်က Constructor code မှာ args_size 1 လို့ ဆိုတဲ့အတွက် main method frame ရဲ့ operand stack ကနေ constructor method ရဲ့ local variable array ထဲကို 1 word copy ကူးထည့် ရမှာပါ။

```
descriptor: ()V
  flags: ACC_PUBLIC
  Code:

    stack=3, locals=1, args_size=1
```

Constructor ကို စ run တဲ့အချိန်မှာ JVM stack ကဒီလိုဖြစ်နေမှာပါ။ JVM Stack ရဲ့ ထိပ်ဆုံးက method frame သည် Constructor ကို run နေတဲ့အတွက် constructor ရဲ့ method frame ဖြစ်ပါတယ်။ ခုက method ကို စ run တဲ့အတွက် PC သည် 0 ပါ။ Constructor ရဲ့ local variable 0 အခန်းထဲမှာ သူ.ကိုခေါ်တဲ့ main method ရဲ့ operand stack ထဲကတန်ဖိုး 0x1001 ကို copy ပေးလိုက်ပါတယ်။ ဒါကတော့ parameter passing ကို JVM က လုပ်သွားတာပါ။ instance method တွေမှာ local variable index ၀ သည် this ကိုသိမ်းပါတယ်။



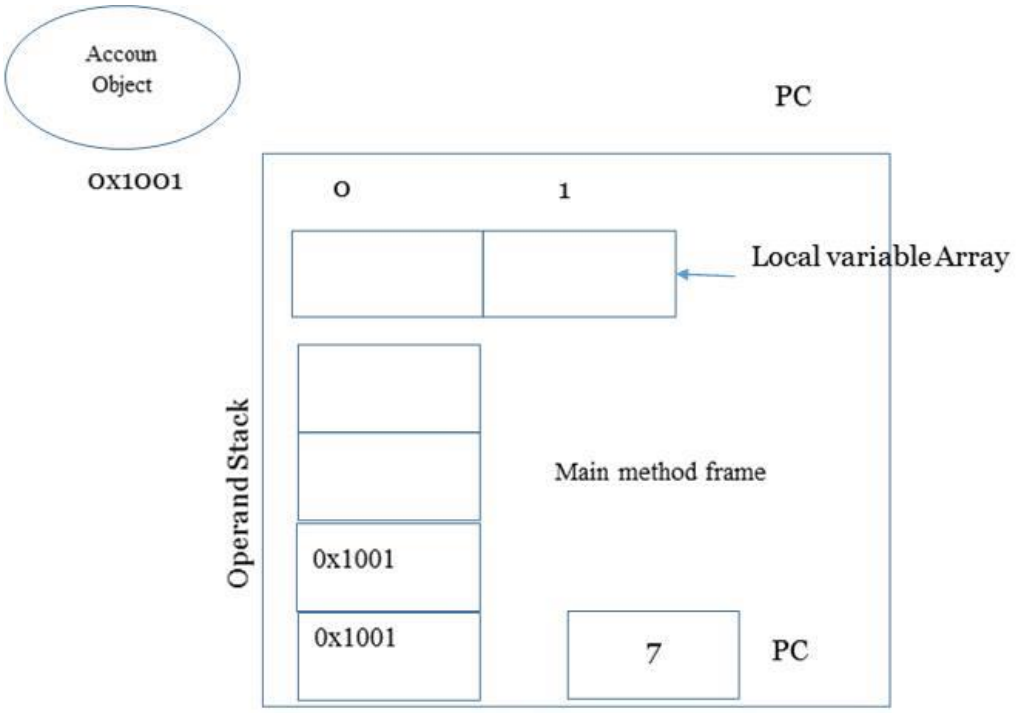
No argument constructor တွေရဲ့ လုပ်ရမဲ့တာဝန်ကတော့ Class ထဲမှာရှိတဲ့ instance variable တွေကို default တန်ဖိုး ဒါမှမဟုတ် instance initializer ရှိရင်ထည့်ပေးရမှာပါ။ ခုဒီ Program မှာ ကျတော့ instance initializer မရှိတဲ့အတွက် instance variable ဖြစ်တဲ့ balance ဆိုတဲ့ field ထဲကို default တန်ဖိုး double ဖြစ်တဲ့အတွက် 0 ကို ထည့် ပေး ရမှာ ပါ။ အောက်က code တွေက constructor ရဲ့ code တွေပါ။

```
 0: aload_0
 1: invokespecial #1                  // Method java/lang/Object."<init>":()V
 4: aload_0
 5: dconst_0
 6: putfield      #2                  // Field balance:D
 9: return
```

ပထဆုံး aload_0 ဆိုတာကတော့ current object this ကို operand stack ပေါ် တင်ပါတယ်။ နောက် invokespecial ဆိုပြီး java.lang.Object ရဲ. constructor ကိုလှမ်းခေါ်ပါတယ်။ နောက် byte code index 4 မှာ aload_0 ဆိုပြီး this object ကို stack ပေါ်တင်ပါတယ်။ နောက် 5 မှာ dconst_0 ဆိုတော့ constant 0 ကို stack ပေါ်တင်ပါတယ်။ နောက် 6 မှာ putfield ဆိုတဲ့အတွက် blance field ကို operand stack ပေါ်က 0 တန်ဖိုးကို assign လုပ်ပါတယ်။ နောက် 9 က return statement ကို run တဲ့အခါ လက်ရှိ Account ရဲ့ default method constructor method frame ကို JVM stack ပေါ်ကနေ pop လုပ်ချလိုက်ပါတယ်။

ဒါဆို JVM stack ပေါ်မှာ main method ပဲကျန်ပါတော့တယ်။

return run တဲ့အခါ stack ပေါ်မှာ byte code index 4 ရဲ့ command aload_0 ကြောင့် current object this ကို သူ့ကိုခေါ်တဲ့ main method ဆီကိုပြန်ပေးရမှာပါ။ အဲ့ဒီတော့ main method frame ရဲ့ operand stack ပေါ်ကို ခုနက constructor ရဲ့ return value ကို တင်ပေးရပါမယ်။ ဒါဆို main method frame သည် အောက်ပါ ပုံအတိုင်းဖြစ်သွားပါပြီ။



Main method frame ရဲ့ operand stack ထိပ်ဆုံးက 0x1001 သည် constructor က return ပြန်လာတဲ့ object ပဲဖြစ်ပါတယ်။ ဒါဆို main method ကိုဆက် run ရမှာပါ ဘာလို့လဲဆိုတော့ သူက ခု JVM stack ရဲ့ ထိပ်ဆုံးက ကောင် ဖြစ်နေလို့ပါ။ ဘယ်ကပြန် run ရမလဲဆိုတော့ PC တန်ဖိုး 7 ဖြစ်တဲ့အတွက် byte code index 7 ကနေပြန် run ရမှာပါ။ အောက်က byte code listing က main method ရဲ့ကျန်နေတဲ့ instruction တွေပါ။

```
7: astore_1
 8: aload_1
 9: ldc2_w      #5            // double 1000.0d
12: invokevirtual #7           // Method deposit:(D)V
```

bytecode index 7 မှာ astore_1 ဆိုတဲ့အတွက် operand stack ပေါ်က ခုနက contructor က return ပြန်တဲ့ return value new object reference ကို local variable account ရဲ့ index ဖြစ်တဲ့ 1 အခန်းမှာသိမ်းမှာပါ။ ဒါဆို main method ရဲ့ byte code 7 အထိသည်...

Account account = new Account();

ကို run လို့ပြီးသွားတယ်ဆိုတာသိမှာပါ။

နောက် byte code 8 က aload_1 ဆိုပြီး local variable account ကို operand stack ပေါ် ခေါ်တင်ပါတယ်။ နောက် ldc2_w ဆိုပြီး double တန်ဖိုး 1000 ကိုလဲ operand stack ပေါ် ခေါ်တင်ပါတယ်။ Method deposit ကို invoke လုပ်ဖို့ parameter တွေကို operand stack ပေါ်တင်နေတာပါ။ ပြီးတော့ မှ line 12 မှာ deposit method ကို လှမ်းခေါ်ပါတယ်။ ကျန်တဲ့ main method ရဲ့ byte code တွေကတော့ string concatenation နဲ့. System.out.println ကို call ခေါ်တဲ့ code တွေပါ။

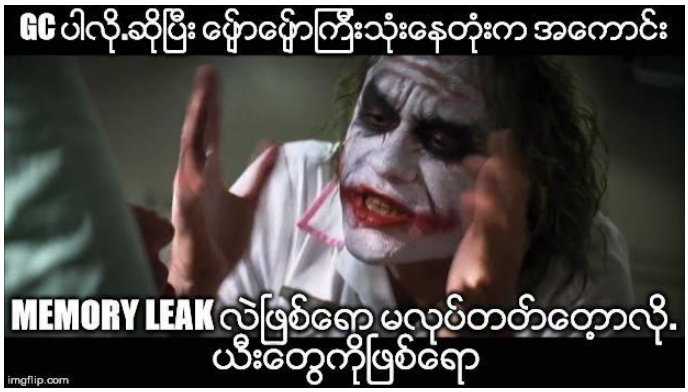
ဒီလောက်ဆို သဘောပေါက်လောက်မယ်ထင်ပါတယ်။
ဆက်ရန် ☺

HOW DOES JVM WORK PART – 8

Garbage collection, heap and references

JVM specification မှာ automatic storage management (Garbage Collector) ထားပေးရမယ်လို့ ဆိုထားပါတယ်။ ဒါပေမဲ့ ဘယ်လို Algorithm တွေသုံးပြီး GC ကို လုပ်ပါလို့တော့ ပြောမထားပါဘူး။ C/C++ လို language တွေမှာ memory ကို malloc လိုကောင်မျိုးနဲ့ allocate လုပ်ပြီး free လို command မျိုးနဲ့ deallocate လုပ်ရပါတယ်။ အဲ့ဒီမှာ ဘာပြဿနာတက်လဲဆိုတော့ Dangling reference ဆိုတဲ့ကောင်မျိုး ထွက်လာပါတယ်။ Dangling reference ဆိုတာက အသုံးမလိုလို့ ဖျက်လိုက်တဲ့ object ကိုညွှန်နေတဲ့ reference variable သို့မဟုတ် pointer variable ကိုခေါ်တာပါ။

Object ကတော့အသုံးမလိုတော့ဘူး ဒါပေမဲ့ သူ့ကို တခြား variable တွေ reference တွေကနေ ညွှန်ထားသေးတယ်ဆိုရင် ဒါဟာ dangling reference ပါ။ Dangling reference ကနေ object ရဲ့ Properties တွေ ယူသုံးတာ... method တွေခေါ်တာမျိုး လုပ်ခဲ့ရင် တကယ့် Object မရှိတော့တဲ့ အတွက် unpreditable result တွေ ဖြစ်လာနိုင်ပါတယ်။ Garbage ဆိုတာကတော့ reference ကနေ မရောက်နိုင်တော့တဲ့ Object တွေ အသုံးမလိုတော့တဲ့ Object တွေကို ဆိုလိုတာပါ။



HEAP

Heap memory ဆိုတာ Stack memory လို တဖက်တည်းက နေပြီး ယူလိုက် ပြန်ထားလိုက် လုပ်တာ မဟုတ်ပါဘူး။ Stack ကတော့ ထိပ်ဆုံးက ထည့်တယ် ထိပ်ဆုံးကပဲ ပြန်ထုတ်တယ် ဆိုတော့ memory management လုပ်ရတာ ရှင်းပါတယ်။ Heap ကျတော့ပထမ Object တခု တောင်းလိုက်ရင် လွတ်နေတဲ့နေရာလေးပေးလိုက်မယ် Object A ပေါ့... နောက်ဒုတိယ Object B တောင်းတယ် A ရဲ့ နောက်က နေရာလေး ပေးလိုက်မယ်ပေါ့။ နောက် Object C တောင်းတယ်... ဒါဆို B နောက်က နေရာလေး ပေးလိုက်မယ်ပေါ့။

ဒါပေမဲ့ အကြောင်းတခုခုကြောင့် B ကိုဖျက်တာရှိနိုင်သလို A ကိုဖျက်တာလည်း ရှိနိုင်ပါတယ်။ ဆိုချင်တာက allocate လုပ်ရင်လဲ လွတ်နေတဲ့နေရာကနေ ယူမယ် deallocate လုပ်ရင်လဲ ယူထားတဲ့ နေရာကိုပဲ ဖျက်လိုက်မယ်... ဒါက Heap သဘောပါ။ မျက်စိထဲမြင်သာအောင်ပြောရရင် စာအုပ်တွေ တင်ဖို့ ထားတဲ့စားပွဲအလွတ်လိုပေါ့...။ ပထမတော့စာအုပ်တအုပ်တင်မယ်... လွတ်နေတဲ့ နေရာလေး ထားလိုက်မယ်။ နောက် ကော်ဖီခွက်တွေ ဘာတွေ တင်မယ် စာအုပ်တင်ထားတဲ့ နေရာမဟုတ်တဲ့ တခြား လွတ်နေတဲ့နေရာ တင်လိုက်တယ်။ စာအုပ်ကို မသုံးတော့ဘူးဆိုရင် စားပွဲပေါ်ကနေ ပြန်ချမယ် ဒါဆို သူ့နေရာက လွတ်သွားမယ် ။ ဒါပေမဲ့ ကော်ဖီခွက် ကစားပွဲအလယ်ရောက်နေရင်တော့ စားပွဲနဲ့ ဆန့်တာတခုခု တင်ချင်ရင်တောင် ကော်ဖီခွက်ကို နေရာရွှေ့မှရတော့မယ်။ Heap မှာ ဒီသဘောတရား ကို fragmentation လို့သုံးပါတယ်။

Garbage

Gabage Collection ဆိုတာ Heap မှာ အသုံးမလိုတော့တဲ့ Object တွေကို ရှင်းထုတ်ပြီး Heap ကို နေရာလွတ်ရအောင် Compact ဖြစ်အောင် ပြန်လုပ်တာကို ခေါ်ပါတယ်။ Garbage Collection Algorithm များစွာရှိပါတယ်။ အသုံးမလိုတော့တဲ့ Object ဆိုတာဘယ်လိုမျိုးလဲဆိုတော့ အောက်က code example ကိုကြည့်ပါ။

```
if(---)
{
      Account a = new Account();
     a.deposit();
}
```

အပေါ်က code example မှာ Account object ကို reference variable a အနေနဲ့ သိမ်းထားပါတယ်။ Java မှာ variable scope သည် block scope ဖြစ်တဲ့အတွက် variable a ဟာ if block ပြီးတာနဲ့ ခေါ်သုံးလို့ မရတော့ပါဘူး။ ဒီတော့ reference a ကညွှန်ထားတဲ့ Account Object ကိုမသုံးတော့ဘူး ဆိုတဲ့အဓိပ္ပာယ်ပါပဲ။

တနည်းအားဖြင့် a ကညွှန်ထားတဲ့ Account object သည် if statement ကျော်သွားတာနဲ့ Garbage ဖြစ်သွားပြီ... အသုံးမလိုတော့ဘူး programmer ကလဲ သုံးလို့မရတော့ဘူး လို့ဆိုရမှာပါ။ အဲ့ဒီ Garbage ကိုmemory ပေါ်ကနေ ရှင်းထုတ်မှသာ နောက်တချိန် program က memory လိုအပ်လာတဲ့အချိန်မှာ အဆင်ပြေမှာပါ။ မဟုတ်ရင် OutOfMemory Error တက်နေမှာပါ။

Garbage Collection ကို modern langugage တွေမှာဘာလို့ ထည့်ပေးထားလဲဆိုတော့ မလိုအပ်တဲ့ Garbage object တွေကို progrmmer အနေနဲ့ manually လိုက်ထိန်းစရာမလိုပဲ language runtime ကနေ ထိန်းပေးထားတဲ့ သဘောပါ။ Programmer အလုပ်သက်သာတယ် ခေါင်းစားစရာ သိပ်မလိုဘူး။ Software ကလဲ ပို reliable ဖြစ်မယ် ဒါကြောင့်ထည့် ပေးထားတာပါ။ ဒါပေမဲ့ Garbage Collector ပါလို့ memory problem ဖြစ်တဲ့ memory leak မတတ်နိုင်တော့ဘူးလား ဆိုတော့ မဟုတ်ပါဘူး တတ်နိုင်ပါသေးတယ်။ ဆိုချင်တာက GC ပါလို့ ငါတို့တော့ဘာလုပ်လုပ် memory leak မဖြစ်နိုင်ဘူးလို့ တထစ်ချ မှတ်မထားဖို့ပါ။ Oracle Java 8 JVM မှာ Garbage Collector ၄ မျိုးကိုသုံးပါတယ်။ အဲ့ကောင်တွေကတော့...

The Serial Collector

The Parallel / Throughput collector

The CMS Collector (Councurrent Mark and Sweep Collector)

The G1 Collector (Garbage First Collector) ဆိုပြီး ၄ ခုရှိပါတယ်။

Java မှာ System.gc() ဆိုပြီး garbage collection algorithm ကို run ဖို့ ခိုင်းနိုင်ပါတယ်။ ဒါပေမဲ့ GC လုပ်မယ်... မလုပ်ဘူး... ဆိုတာကိုတော့ system ကပဲ ဆုံးဖြတ်မှာပါ။ လုပ်ချင်မှလဲ လုပ်မယ် မလုပ်ချင်လဲ မလုပ်ဘူးဆိုတဲ့ အကွက်ပါပဲ။ Garbage Collection လုပ်တဲ့ နေရာမှာ method frame ပေါ်က local variable တွေကနေတဆင့် ရောက်နိုင်တဲ့ reachable object တွေကို ချန်ထားပြီး မရောက်နိုင်တဲ့ unreachable ဒါမှမဟုတ် အသုံးမလိုတော့တဲ့ object တွေကို collect လုပ်မှာပါ။ Java Reference တွေကို 4 မျိုးခွဲထားပါသေးတယ်။

Strong references

သာမာန် သုံးနေကျ reference တွေကို strong refrence လို့ ခေါ်ပါတယ်...။ သူတို့ကိုကျတော့ Garbage Collection လုပ်ဖို့ အဲ့ဒီ reference တွေက ညွှန်တဲ့ Object တွေသည် Garbage ဖြစ်သလား ။ ဥပမာ Object တခုကို reference a ရော reference b ပါညွှန်တယ်ဆိုပါစို့ a က မညွှန်တော့ပေမဲ့ b က ညွှန်နေသေးတယ်ဆိုရင် object သည် reachable ဖြစ်တာကြောင့် GC collect လုပ်လို့မရပါဘူး။

အောက်က code sample က strong reference ကိုပြထားတာပါ။

StringBuffer buffer = new StringBuffer();

Weak Reference

```
Counter counter = new Counter(); // strong reference - line 1
WeakReference<Counter> weakCounter = new WeakReference<Counter>
(counter); //weak reference
counter = null;
```

ပထမဆုံး line က counter variable သည် strong reference ပါ။ နောက် weak reference ကိုဆောက်ပါတယ်။ နောက် counter ကို null assign လုပ်ပါတယ်။ အဲ့ဒီတော့ weak reference သည် သူ့ကို ညွှန်တဲ့ strong reference မရှိတော့တဲ့ အတွက် GC collect လုပ်လို့ရပါပြီ။ Strong reference မရှိရင်တောင်မှ GC က memory မလိုသေးရင် GC မလုပ်စေချင်သေးရင်တော့ အောက်က soft reference ကိုသုံးရမှာပါ။

Soft reference

Soft Reference တွေကိုကျတော့ JVM က memory ကို ဆိုးဆိုးရွားရွား မလိုသေးသရွေ့ Garbage Collect လုပ်မှာ မဟုတ်သေးပါဘူး။ တကယ်လို့ JVM က memory ကို လိုနေပြီ ဆိုရင်တော့ သူတို့ဟာ GC အလုပ်ခံရပြီး အဲ့ဒီ reference တွေက null return ပြန်လာမှာပါ။

```
A a = new A();      //Strong Reference
//Creating Soft Reference to A-type object to which 'a' is also pointing
SoftReference<A> softA = new SoftReference<A>(a);
a = null;    //Now, A-type object to which 'a' is pointing earlier is eligible for garbage
collection. But, it will be garbage collected only when JVM needs memory.

a = softA.get();
```

အပေါ်က code example မှာ a သည် strong reference ပါ။ a ကို null ထားလိုက်ပေမဲ့ softA ဖြစ်တဲ့ soft reference ကနေ GC မလုပ်သေးဘူးဆိုရင် ပြန်ယူလို့ရပါသေးတယ်။ GC လုပ်လိုက်ရင်တော့ null ပြန်ရမှာပါ။ Cached data လို အမြဲတမ်းလည်း မသိမ်းချင်ဘူး လိုအပ်ရင် GC အလုပ်ခံနိုင်တဲ့ ကောင်မျိုးကိုဆို soft reference သုံးပါတယ်။

Phantom references

Java object တွေမှာ finalized ဆိုတဲ့ method ကို root Object class ကနေ override လုပ်လို့ရပါတယ်။ သူ့ကို GC မလုပ်ခင်လေးမှာ ထ run ပါလိမ့်မယ်။ ဒါပေမဲ့ finalized method မှာ strong reference တွေ ပြန်ဆောက်ခွင့် ပေးထားတဲ့အတွက် တခါတလေ အဆင်မပြေပါဘူး။ ဒီလိုအခြေအနေမျိုး clean up လုပ်ချင်ရင်တော့ phantom reference ဆိုတာကို သုံးရပါတယ်။ Phantom reference တွေကို JVM က garbage collect လုပ်ကိုလုပ်မှပါ။ ဒါပေမဲ့ သူတို့ကို GC မလုပ်ခင် ReferenceQueue ထဲကိုထည့် ပါတယ်။ ဒါကြောင့် clean up finalization လုပ်လို့ ရပါတယ်။ အောက်က code example ကတော့ Phantom Reference example ပါ။

```
A a = new A();     //Strong Reference
//Creating ReferenceQueue
ReferenceQueue<A> refQueue = new ReferenceQueue<A>();
//Creating Phantom Reference to A-type object to which 'a' is also pointing
PhantomReference<A> phantomA = new PhantomReference<A>(a, refQueue);
a = null;   //Now, A-type object to which 'a' is pointing earlier is available for garbage
//collection. But, this object is kept in 'refQueue' before removing it from the memory.

a = phantomA.get();   //it always returns null
```

GC လုပ်ပြီးပေမဲ့ refQueue ထဲကနေ move လုပ်ပြီး PhantomReference ကို remove လုပ်ပြီး cleanup code ကို run လို့.ရပါတယ်။
ဆက်ရန် ☺

How does jvm work part-9

Different garbage collection method and reachability detection

Gabage Collection Algorithm က ဘယ်လို အလုပ်လုပ်သလဲ
Garbage Collection Algorithm တွေဟာ အခြေခံအားဖြင့် အလုပ် အဆင့် ၂ ဆင့်ကို လုပ်ရပါတယ်။ ပထမ အဆင့်က ဘယ် Object တွေက Garbage ဖြစ်သလဲ ဆိုတာကို detection လုပ်ရပါတယ်။ Object တွေ Garbage ဖြစ်မဖြစ်ကို စစ်တဲ့နည်းလမ်းပေါင်းစုံ ရှိပါတယ်။ နောက်တဆင့်ကတော့ ခုနက detected လုပ်ထားတဲ့ Garbage Object တွေက ယူထားတဲ့ memory area တွေကို application ကနေ ပြန်သုံးလို့ရအောင် release လုပ်တဲ့အပိုင်းကို လုပ်ရပါတယ်။

Garbage Object တွေကို ဘယ်လိုသိနိုင်သလဲဆိုတော့ GC root ကနေပြီး Object တွေကို reachable ဖြစ်နိုင်သလား မဖြစ်နိုင်သလားဆိုပြီး စစ်တာကနေသိနိုင်ပါတယ်။ ဥပမာ အားဖြင့် Object တခုကို application ရဲ့ တစိတ်တပိုင်းကနေ reference လုပ်နေဦးမယ်ဆိုရင် ဒီ Object က reachable object ပါ။

တိုက်ရိုက်မဟုတ်ပဲ တဆင့်ခံ rechable ဖြစ်နိုင်လည်း... ဒါ rechable Object ပါ။ Rechable Object မဟုတ်တဲ့ ကောင်တွေက တော့ garbage object ဖြစ်သွားပါပြီ။ GC root ကနေ Object တွေကို program ကနေ reachable ဖြစ်သလား (အဲ့ဒီ Object ကလက်ရှိသုံးနေသလား သူ့ကိုသုံးနေတဲ့ reference ရှိသလားကို စစ်တာပါ) ဆိုပြီးစစ်ရပါတယ်။ GC root ဆိုတာ JVM stack ပေါ်မှာ ရှိတဲ့ local variable တွေရယ် Object ရဲ့ instance ,class variable တွေရယ်ကို ဆိုလိုတာပါ... java language အတွက်ဆိုရင်ပေါ့ဗျာ... Method stack ပေါ် ဒါမှမဟုတ် JVM stack ပေါ်မှာရှိတဲ့ local variable reference တွေကနေ တဆင့်ရောက်နိုင်တဲ့ object တွေဟာ reachable object ပါပဲ။ JVM method stack ရဲ့ local variable တွေကနေ တိုက်ရိုက် သို့မဟုတ် သွယ်ဝိုက် (ဆိုချင်တာက တဆင့်ခံပြီးတော့ ရောက်နိုင်သေးတာ) reachable ဖြစ်နေရင် ဒီ Object တွေကို live object လို့သတ်မှတ်ပါတယ်။ မဟုတ်ရင်တော့ Garbage object ပါပဲ။ အောက်က program ကိုကြည့်ပါ။

```
class GCDemo
{
   public static void main(String[] args) {
      String ref1 = new String( "String 1");
      String ref2 = new String("String 2");

       ref1 = ref2;
       ref1 = null;
   }
}
```

အပေါ်က program မှာ String object 2 ခုကိုတည်ဆောက်ပါတယ် String1 ရယ် String 2 ရယ်ပါ။ သူတို့ကို ref1 နဲ. ref2 ဆိုပြီး assignment နဲ့ assign လုပ်ပါတယ်။ ဒါဆိုရင် Object String1 ကို ref1 ကညွှန်နေပြီးတော့ Object String2 ကို ref2 က ညွှန်နေပါတယ်။ ဒီအချိန်မှာ အဲ့ဒီ Object ၂ခုက program variable ကနေ reachable ဖြစ်နေပါတယ်။ နောက်တကြောင်းမှာ

```
ref1 = ref2;
```

ဒီလိုရေးလိုက်ပါတယ် ။

ဒါဆို ref1 သည် ref2 က ညွှန်တဲ့ Object String 2 ကိုသွားညွှန်နေပါပြီ ။ ဒါဆိုခုနက heap ပေါ်မှာရှိတဲ့ Object ၂ခုထဲက ဒီအဆင့်မှာ Object String1 ကို ညွှန်တဲ့ကောင် program မှာ မရှိတော့ပါဘူး။ ဒါဆို ဒီအဆင့်မှာ Object String1 သည် unreachable ဖြစ်နေပါပြီ။ နောက်ဆုံးမှာ ref1 ထဲကို null ထဲ့လိုက်တယ် ဒါဆို String 2 ကိုညွှန်နေရာကနေ ref1 သည် null ကိုညွှန်တာဖြစ်သွားပါပြီ။

String 2ကို ref1 ကမညွှန်တော့ပေမဲ့ ref2 ကညွှန်နေတဲ့အတွက် String2 သည် ဒီအချိန်မှာ reachable ဖြစ်ပါတယ်။ ဒါပေမဲ့ java မှာ local variable တွေဟာ block scope ဖြစ်တဲ့အတွက် သူ့ကို ကြေငြာထားတဲ့ block ကုန်တာနဲ့ ဒီ variable က ပျက်ပါပြီ... ဒါဆို အဲ့ဒီ နောက်ဆုံး block အရောက်မှာ ဒီမှာဆို main method အဆုံးမှာ ref1 ရော ref2 ရော ပျက်ပြီ ဖြစ်တဲ့အတွက် သူတို့က ညွှန်ထားတဲ့ Object String1 ရော String 2 ရောသည် unreachable ဖြစ်ပါပြီ။
အဲ့ဒီ Object 2 ခုလုံးသည် GC collect လုပ်လို့ရပါပြီ။
တကယ်လို့ များ ခုနက String 1 ဒါမှမဟုတ် String2 ကို တခြား class ကို instance အနေနဲ့ ပေးလိုက်မယ်ဆိုရင် သူတို့သည် တခြား class ကနေ reachable ဖြစ်နေခဲ့ရင် Garbage Object ထဲပါမှာမဟုတ်ပါဘူး။
Object တွေဟာ reachable ဖြစ်လားမဖြစ်လားဆိုတာကို algorithm တွေသုံးပြီးစစ်ကြပါတယ်။ အဓိကကတော့ Algorithm 2 မျိုးရှိပါတယ်။ Reference Counting Collectors နဲ့ Tracing Collectors ဆိုပြီး ၂မျိုးခွဲပါတယ်။

Reference counting collectors

Reference Counting Collector တွေ အလုပ်လုပ်ပုံကတော့ Object တခုစီရဲ့ reference handle မှာ သူ့ကို reference ဘယ်နှစ်ခုက ညွှန်ထားသလဲဆိုတာရဲ့ counter လေးကို သိမ်းထားပါတယ်။
ဥပမာ...

```
String s1 = new String("Hello");// counter =1
String s2= s1 ; //then counter = 2
s1 = null;// then counter =1
```

အပေါ်က code မှာဆို ပထမဆုံး Hello ဆိုတဲ့ String object မှာ ပထမဆုံး s1 ကို assign လုပ်လိုက်တဲ့အတွက် counter က ၁ ပါ နောက် s2=s1 ဆိုတော့ Hello ဆိုတဲ့ String object ကိုညွှန်တာက s1 ရော s2 ရောဖြစ်တဲ့အတွက် ဒုတိယ assignment statement မှာ Hello String Object ရဲ့ reference count က ၂ ဖြစ်သွားပါပြီ။

တတိယ Statement မှာ s1 ကို null ထဲ့လိုက်တဲ့အတွက် s1 က ညွှန်နေတဲ့ Hello String Object သည် reference count 1 ပြန်ဖြစ်သွားပါပြီ ။ ဒီအချိန်မှာ Hello String object သည် GC collect လုပ်လို့ ရမလားဆိုရင် မရပါဘူး ။ဘာလို့လဲဆိုတော့ သူ့ရဲ့ reference count သည် zero မဟုတ်လို့ပါ ။

တခုခုက သူ့ကို ညွှန်နေသေးတဲ့အတွက် သူသည် non GC collectable ဖြစ်ပါတယ်။ reference counting ရဲ့ ပြဿနာကတော့ Circular Object တွေမှာ ကျိုင်ရှိပါတယ်...

ဥပမာ A က B ကိုညွှန်တယ်... B ကနေ A ကိုပြန်ညွှန်ရင် ဒါဆို Object တခုတည်းမှာ ညွှန်တဲ့ကောင် မရှိတော့လဲ reference count ရှိသေးတဲ့အတွက် Garbage ဖြစ်နေပေမဲ့ GC collect လို့ မရပါဘူး။ Java မှာဒီလို reference counting collector မသုံးပဲ tracing collector type တွေပဲသုံးပါတယ်။ PHP မှာတော့ reference counting collector သုံးပါတယ်။ အောက်က example က PHP ပါ။

```
<?php
$a = "new string";
$b = $a;
xdebug_debug_zval( 'a' );
?>
```

ဒါကို run လိုက်ရင် ဒီလိုထွက်လာပါလိမ့်မယ် a: (refcount=2, is_ref=0) ='new string'
php.ini ကိုတော့ xdebug ကိုသုံးဖို့ နည်းနည်းပြန် configure လုပ်ရ ပါလိမ့်မယ်
a က ညွှန်ထားတဲ့ object က reference count ၂ ခု ရှိနေတယ်ဆိုတာကိုပြတာပါ။

Tracing collectors

Tracing Collector ကတော့ GC root (လက်ရှိ JVM Stack ပေါ်ကနေသုံးနေတဲ့ variable တွေရယ် object instance, static variable တွေရယ်) ကနေ ရောက်နိုင်တဲ့ object တွေကို reachability graph ဆွဲပါတယ်။

Reachable ဖြစ်တယ်ဆိုရင် အဲ့ဒီ Object တွေကို mark လုပ်ထားပါတယ် ။ အားလုံး mark လုပ်ပြီးမှ (mark phase) reachability graph ကနေ မရောက်နိုင်တဲ့ Object တွေကို Garbage အနေနဲ့ သတ်မှတ်ပြီးတော့ GC collection လုပ်ပစ်ပါတယ် (sweep phase) ။ Tracing Collector တွေကို ထပ်ခွဲပါသေးတယ်။

Compacting collectors

Object တွေက Heap ပေါ်မှာနေရာ ခဏခဏ ယူပြီး ပြန် release လုပ်ရင် hard disk တွေလိုပဲ heap fragmentation ဖြစ်ပါတယ်။ ဒါဆိုရင် heap ကို GC collect လုပ်ယုံတင်မကပဲ... Fragementation ကို ဖျောက်ဖို့ အတွက် compact လုပ်ဖို့လိုပါတယ်။ Live Object တွေကို ပထမ အရင် live Object တွေ ရှိတဲ့ memory ဘေးနားကို ကပ်ခိုင်းပြီး compact လုပ်ပစ်ပြီး heap fragmentation ကို ပျောက်စေပါတယ်။

ဥပမာပေးရရင် ကားပေါ်မှာ နေရာလွတ်သွားရင် ဒရိုင်ဘာနဲ့ နီးတဲ့ဘက်ကိုပဲ တိုးကပ်ခိုင်းတာ မျိုးပါ။ ဒါက Compacting Collector တွေလုပ်တဲ့ပုံပါ။ Object တွေဟာ နေရာရွှေ့သွားတဲ့အတွက် သူတို့ရဲ့ internal reference တွေကို memory location အသစ်ကို ညွှန်ဖို့ update လုပ်ရပါမယ်။

Copying collector

ဒီမှာကျတော့ Heap ကို နှစ်ပိုင်း ပိုင်းလိုက်ပါတယ်။ live object တွေကို တပိုင်း... Garbage object တွေကို တပိုင်း... ဆိုပြီး နှစ်ပိုင်း ပိုင်းပါတယ်။ Live Object တွေ့ရင် သူတို့ နေရမယ့် အပိုင်းကို copy ကူးပို့ ပါတယ်။ Compacting တုန်း ကလို အရင်လွတ်နေတဲ့ နေရာလေးနား သွားကပ်ခိုင်းတာ မဟုတ်ပဲ live Object တွေပဲ ရှိရမယ့် ဘက်ကိုသာ copy လုပ်ခိုင်းတာပါ။ အဲ့ဒီတော့ Mark and sweep ပြီးသွားတဲ့အခါ heap တခြမ်းဟာ live object တွေ တဆက်တည်း ဖြစ်သွားပြီး heap fragmentation ကို ကွာကွယ်စေနိုင်ပါတယ်။

Generational collector

Mark and sweep collector တွေရဲ့ မကောင်းတဲ့ အချက်က performance ကို တော်တော်နှေး ပါတယ်။ ဘာလို့လဲဆိုတော့ object တွေကိုရော လိုက်ရွေ့ နေရတဲ့အတွက်ပါ။ အဲ့ဒီတော့ နောက် GC algorithm တွေကို Heap တခုလုံးကြီးကို အဲ့လိုလုပ်မယ့် အစား Heap ကို အပိုင်းလေးတွေ ထပ်ခွဲလိုက်ပါတယ်။

ဥပမာ short live object တွေကို heap ပထမပိုင်းမှာ ထားပြီး long live object တွေကို နောက်အပိုင်းမှာ ထားပါတယ်။ ဒါဆို GC ကို heap အပိုင်းလေးတွေမှာပဲ လုပ်လို့ ရသွားပါပြီ။ Object တွေကို lifetime ပေါ်မူတည်ပြီး ဘယ် heap မှာထားမယ်ဆိုတာဆုံးဖြတ်ပါတယ်။ ဥပမာ Object တွေကို ပထမဆုံးမှာ short live heap မှာ ထားပါတယ်။ နောက် GC ကို short live heap မှာလုပ်ပါတယ် short live heap မှာ GC လုပ်တဲ့ အကြိမ်အရေအတွက် treshold တခုကို ကျော်လာရင် ဒီ Object ကို long live heap ထဲကို ပြောင်းပါတယ်။ long live heap ထဲက Object တွေက တော်ရုံနဲ့ GC မလုပ်ကြပါဘူး။ GC ကို heap ရဲ. တပိုင်းခြင်းဆီမှာပဲခွဲလုပ်ရတဲ့အတွက် Performance ကို ပိုကောင်းစေပါတယ် ။ ဒီနည်းကိုတော့ Generational Collector လို့ခေါ်ပါတယ်။ ဆက်ရန် ☺