3-4 DECEMBER IEC, SOFIA, BULGARIA 2013

Dissecting the Hotspot JVM

Martin Toshev @martin_fmi
Ivan St. Ivanov @ivan_stefanov

Agenda

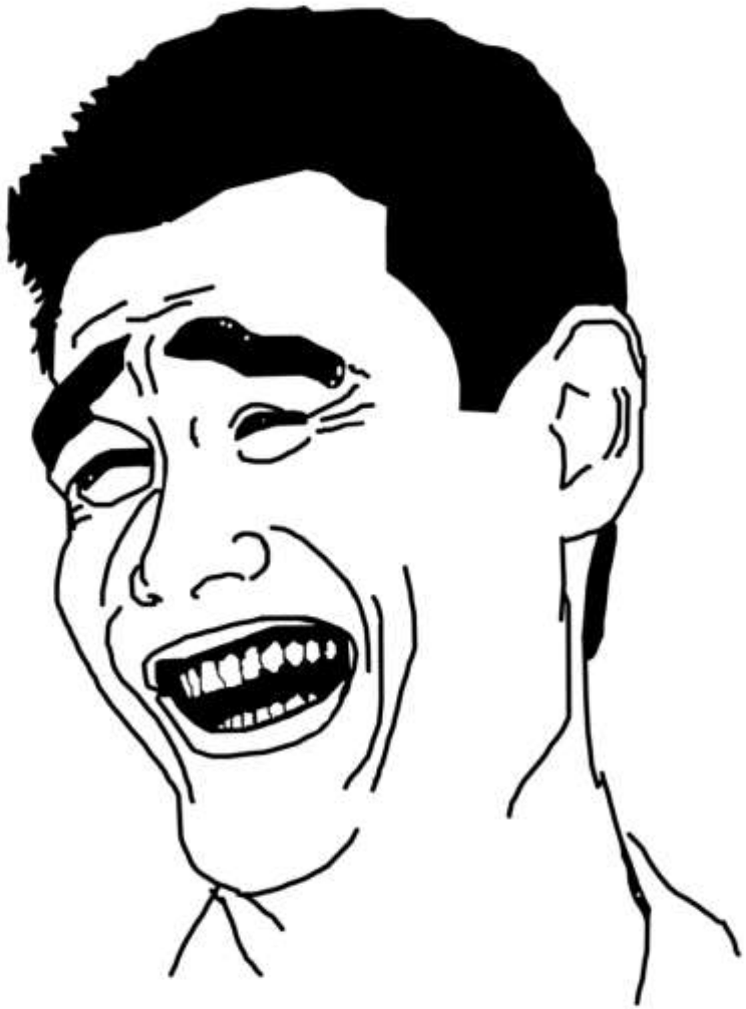
Virtual Machine Basics

The Hotspot JVM

Understanding the Hotspot Source Code

Debugging Hotspot

DISCLAIMER



Why this presentation ?

Feeling comfortable with the JVM source code allows you to:

- contribute to and improve the JVM itself
- fork and customize the JVM for whatever reason you want (experimental, commercial,
  academic …)

Why this presentation ?

Feeling comfortable with the JVM source code allows you to:

- be a step ahead of Chuck Norris

Virtual Machine Basics

A typical virtual machine for an interpreted language provides:

- Compilation of source language into VM specific bytecode
- Data structures to contains instructions and operands (the data the instructions process)

Virtual Machine Basics

A typical virtual machine for an interpreted language provides:

- A call stack for function call operations
- An ‘Instruction Pointer’ (IP) pointing to the next instruction to execute

Virtual Machine Basics

A typical virtual machine for an interpreted language provides:

- A virtual ‘CPU’ – the instruction dispatcher that:
  - Fetches the next instruction (addressed by the instruction pointer)
  - Decodes the operands
  - Executes the instruction

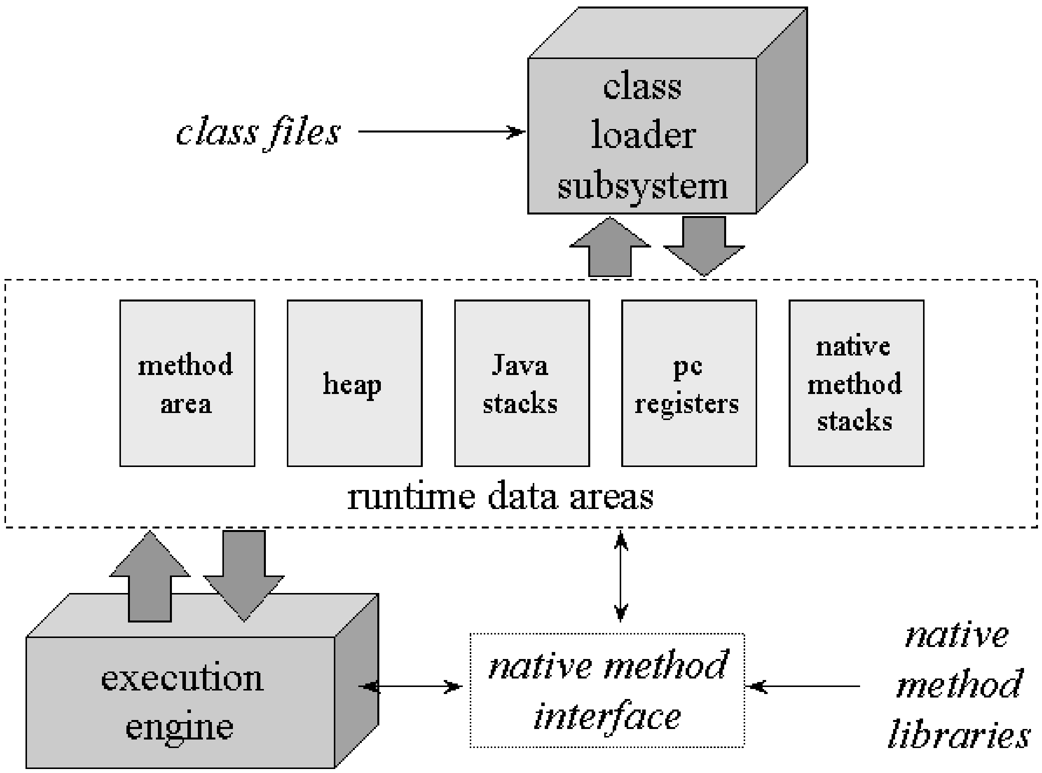
The Hotspot JVM

Provides:

- bytecode execution - using an interpreter, two runtime compilers or On-Stack Replacement
- storage allocation and garbage collection
- runtimes - start up, shut down, class loading, threads, interaction with OS and others

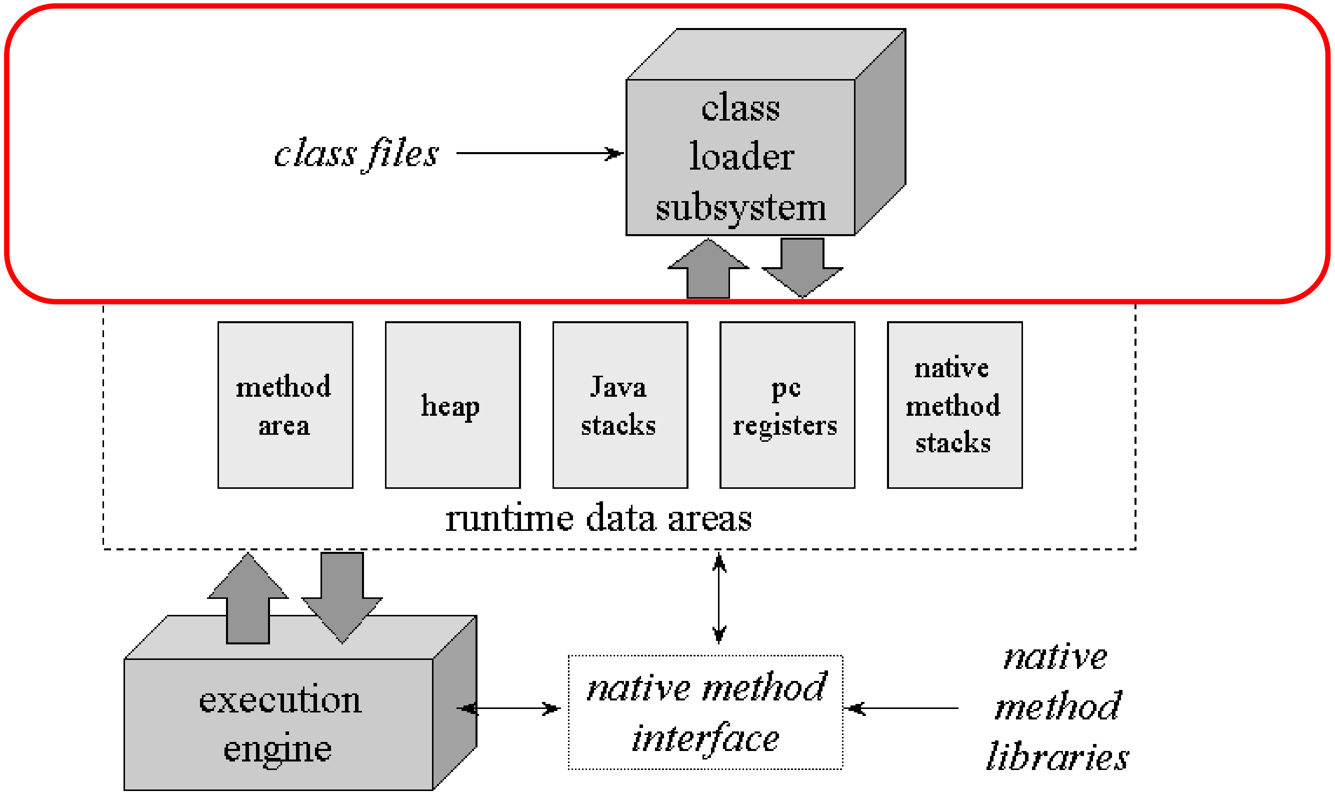
The Hotspot JVM

Architecture:



The Hotspot JVM

Architecture:



The Hotspot JVM

```
ClassFile {
    u4             magic;
    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];
}
```

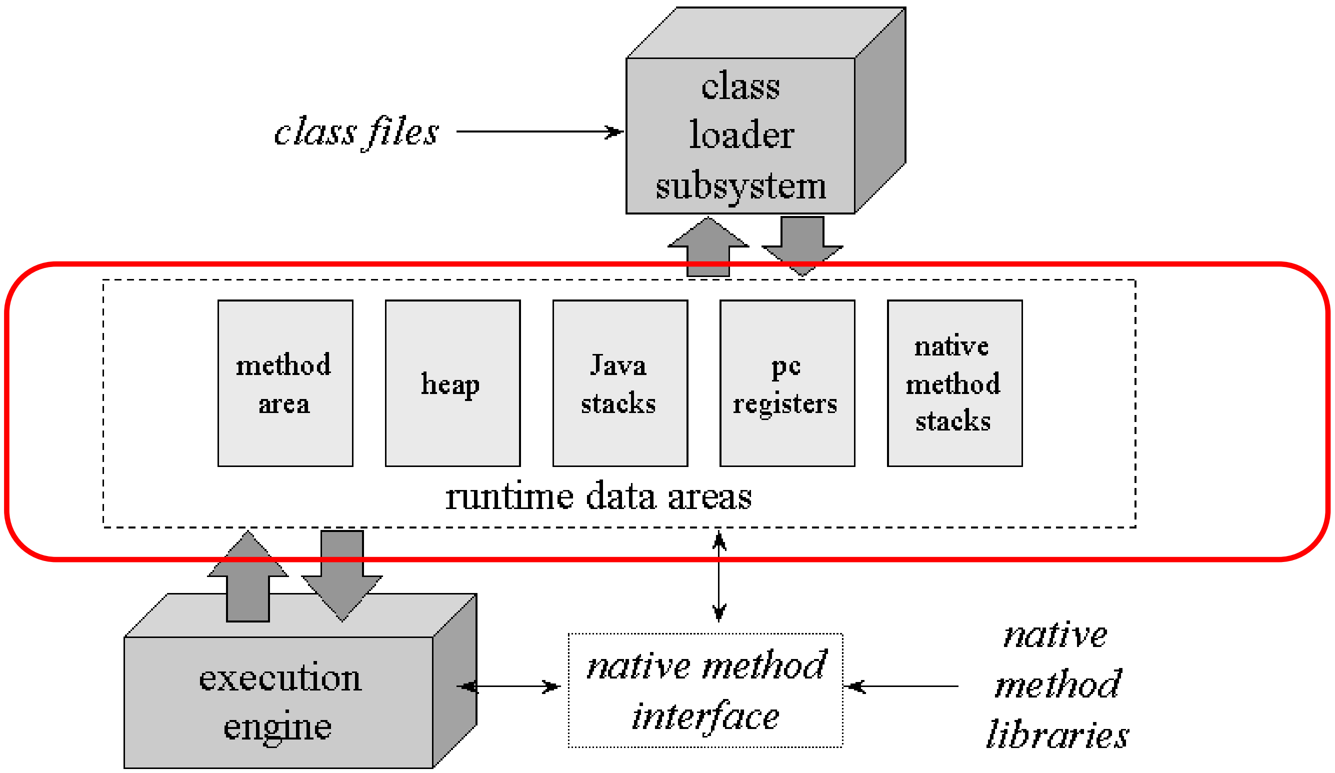
The Hotspot JVM

Three phases of class-loading:

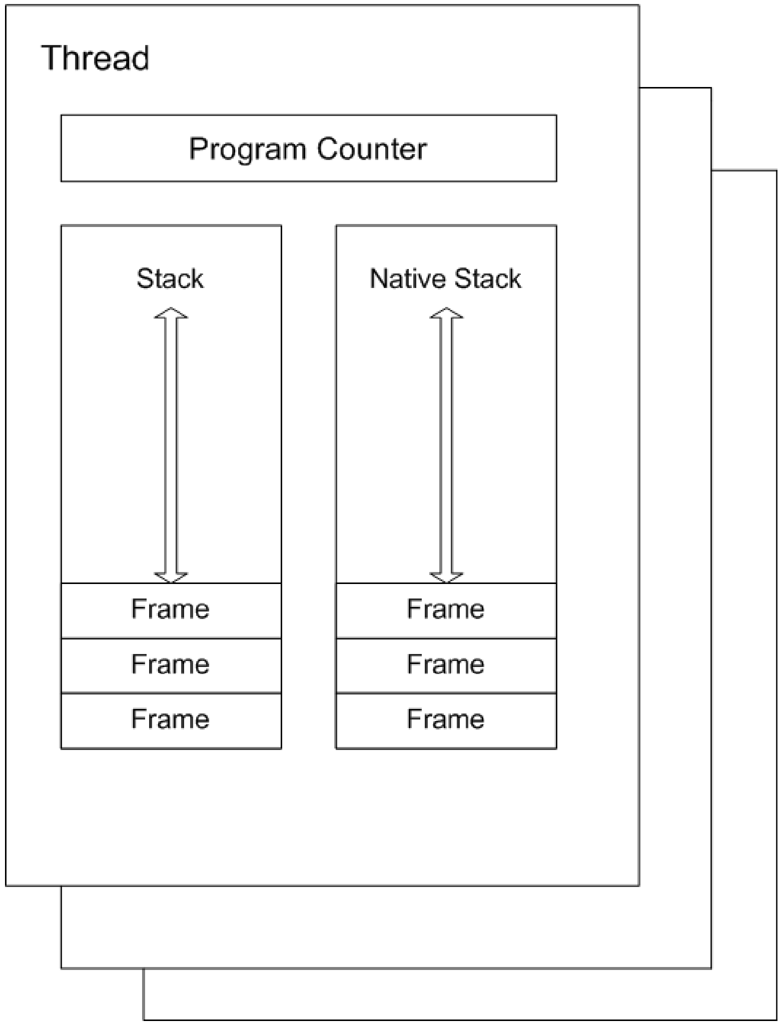
- Loading
- Linking
- Initialization

The Hotspot JVM

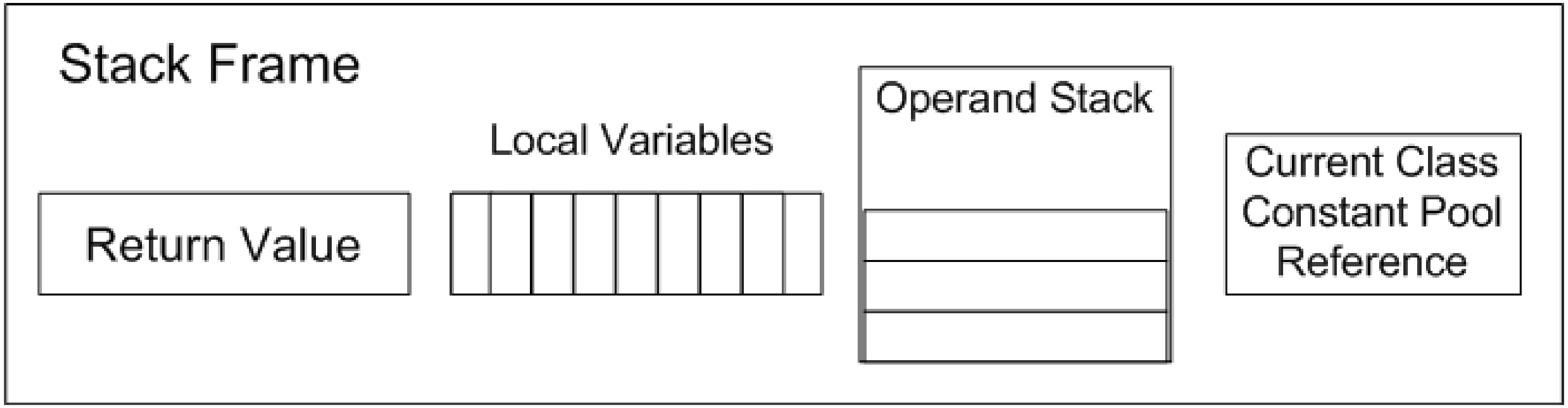
Architecture:



The Hotspot JVM



The Hotspot JVM

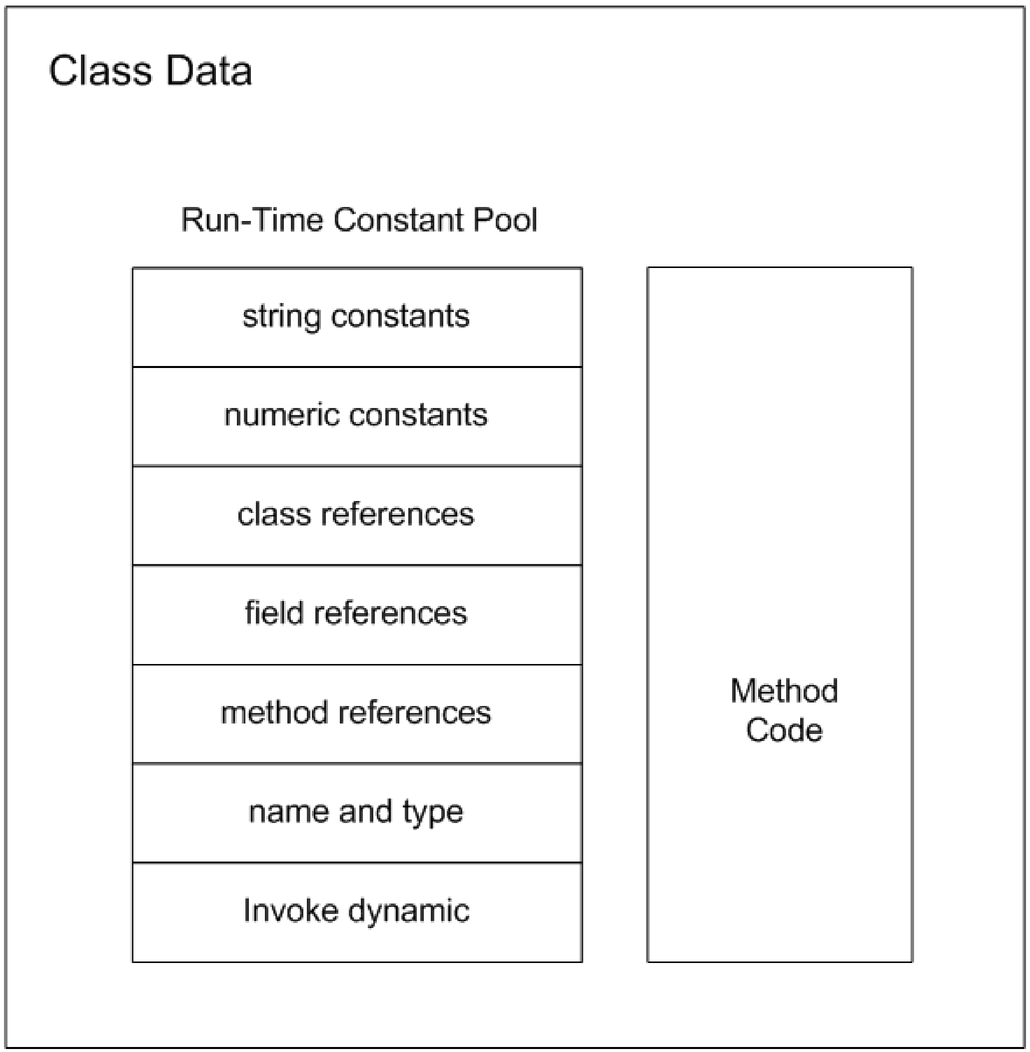


The Hotspot JVM

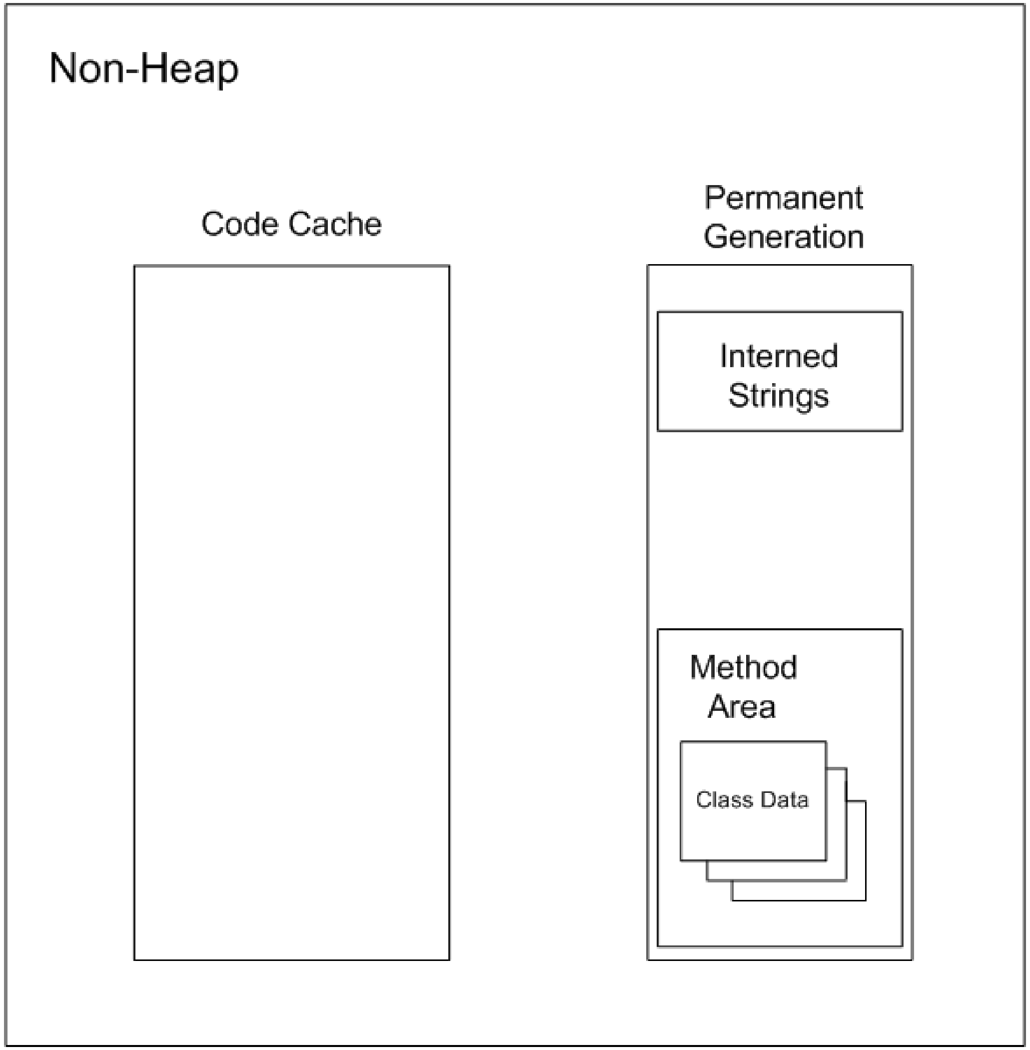
```
static int volume(int width,
        int depth,
        int height) {
    int area = width * depth;
    int volume = area * height;
    return volume;
}
```

```
0 iload_0
1 iload_1
2 imul
3 istore_3
4 iload_3
5 iload_2
6 imul
7 istore 4
9 iload 4
11ireturn
```

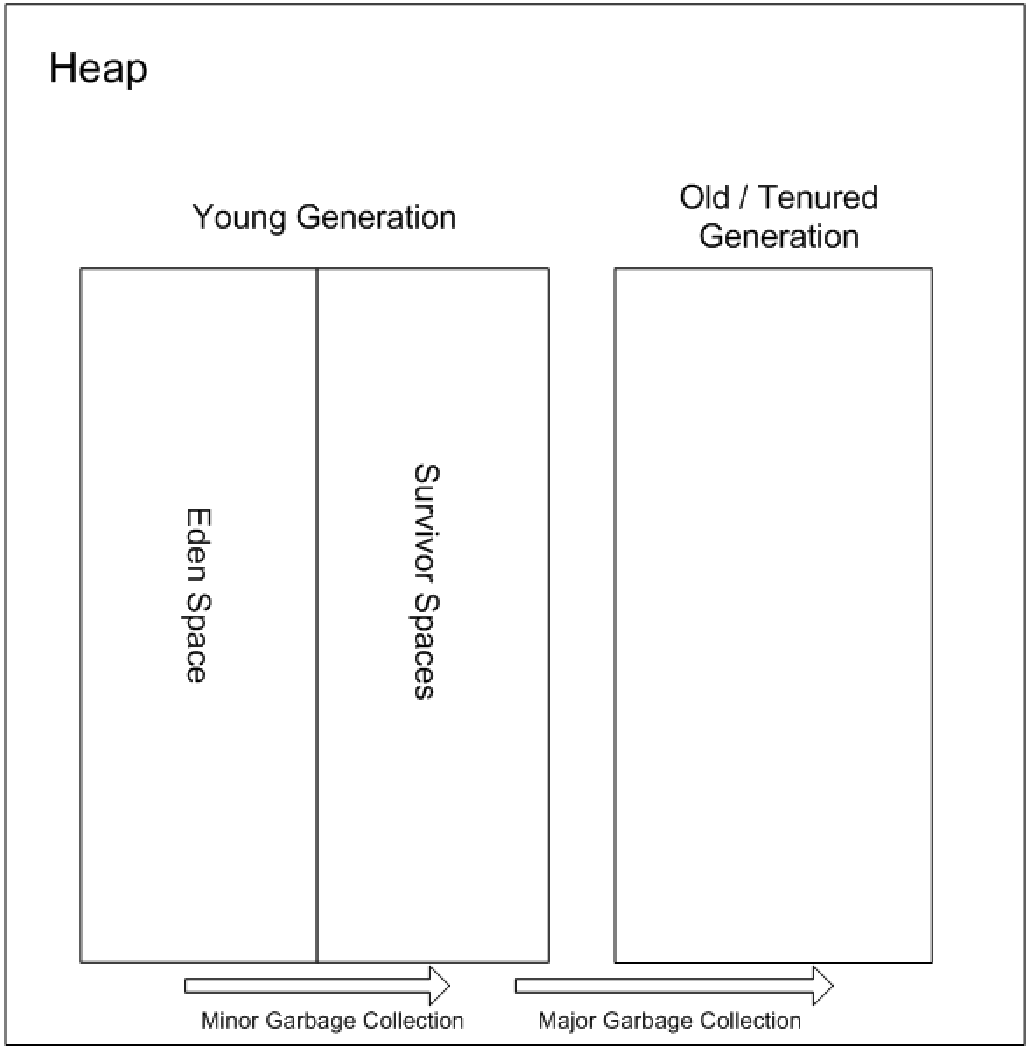
The Hotspot JVM



The Hotspot JVM

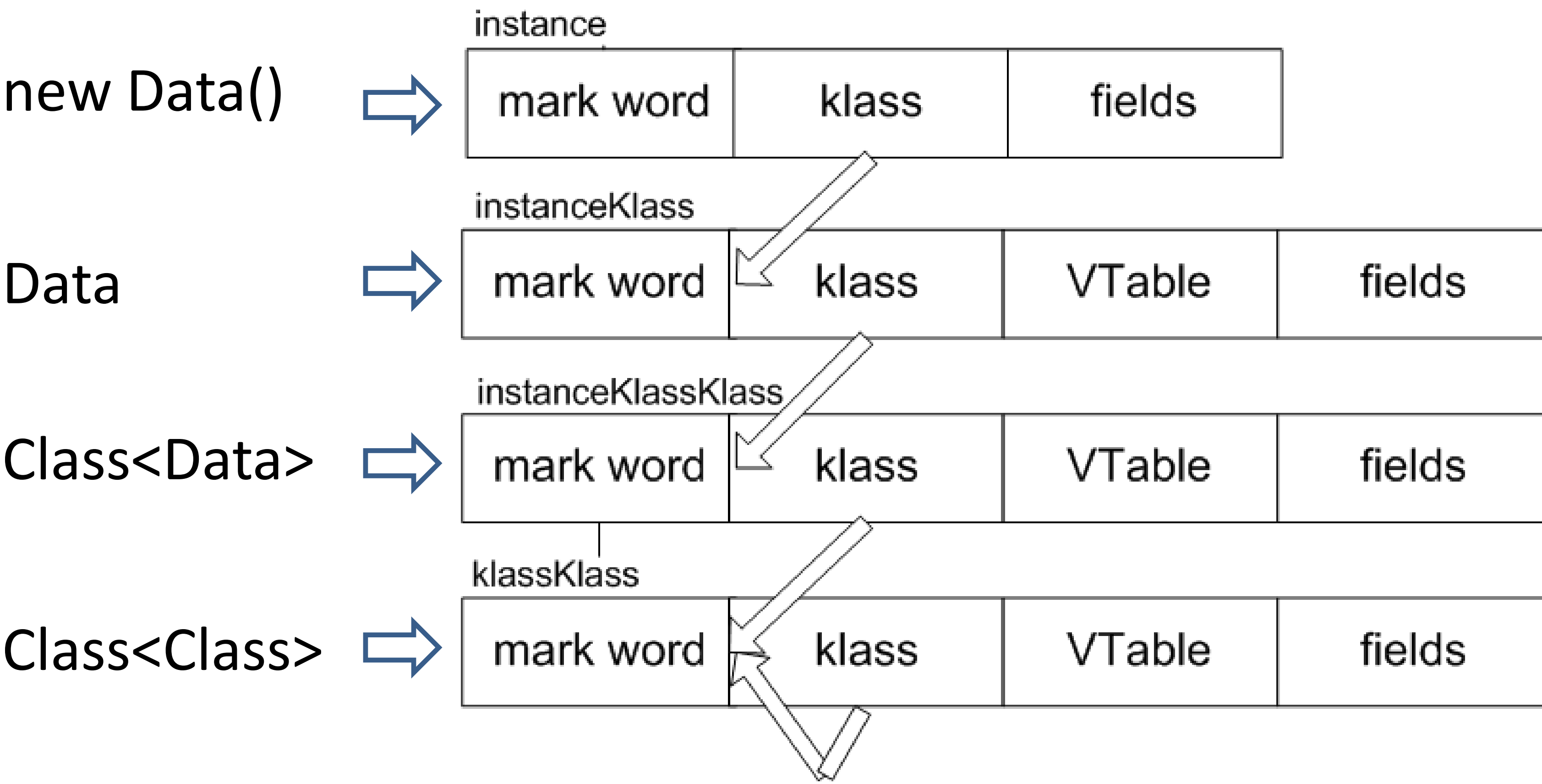


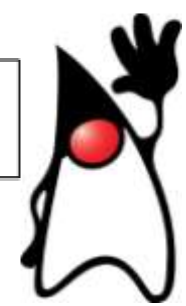
The Hotspot JVM



The Hotspot JVM

Heap memory:





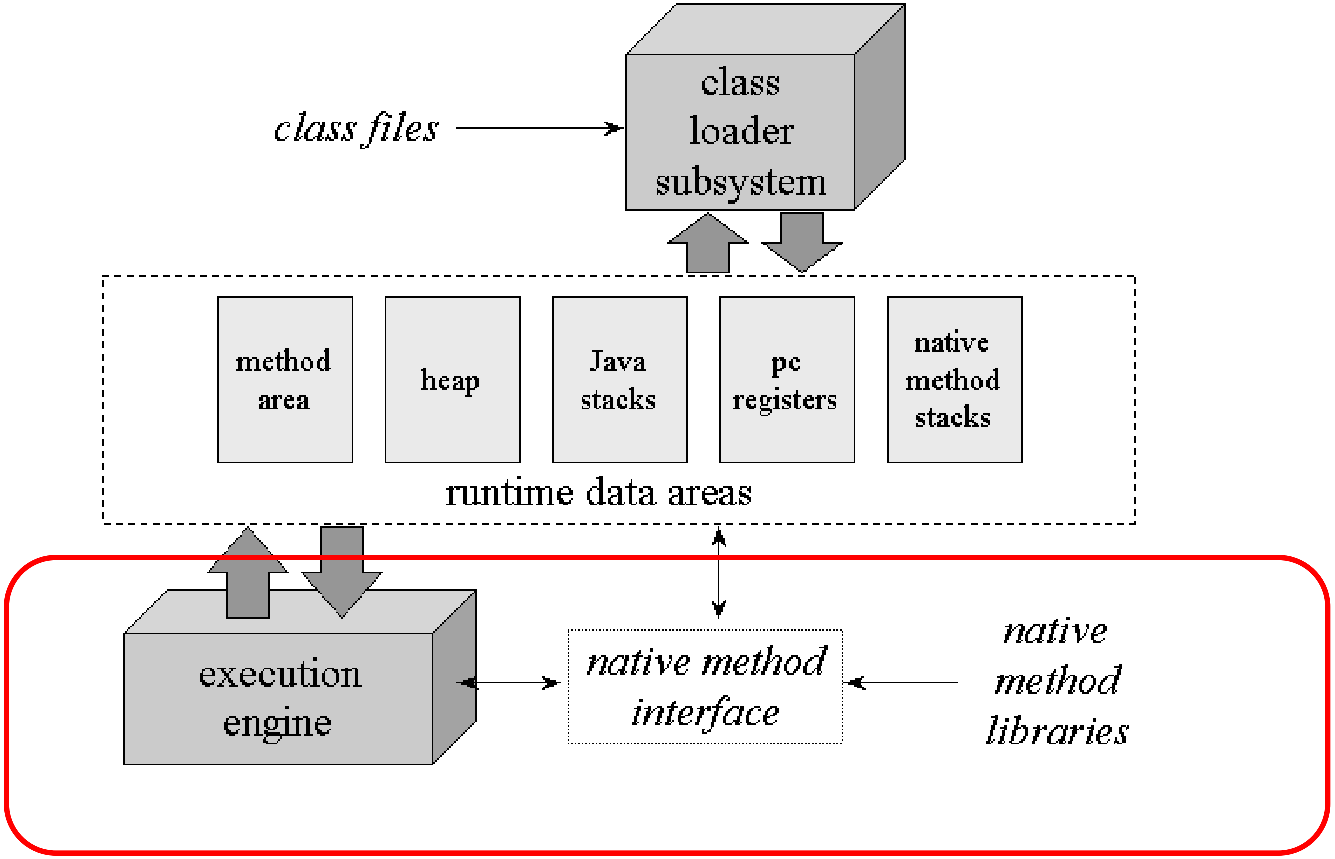
The Hotspot JVM

Mark word contains:

- Identity hash code
- age
- lock record address
- monitor address
- state (unlocked, light-weight locked, heavy-weight locked, marked for GC)
- biased / biasable (includes other fields such as thread ID)

The Hotspot JVM

Architecture:



The Hotspot JVM

Execution engine:

```
while(true) {
        bytecode b = bytecodeStream[pc++];
        switch(b) {
                case iconst_1: push(1); break;
                case iload_0: push(local(0)); break;
                case iadd: push(pop() + pop()); break;
        }
}
```

The Hotspot JVM

Execution engine:

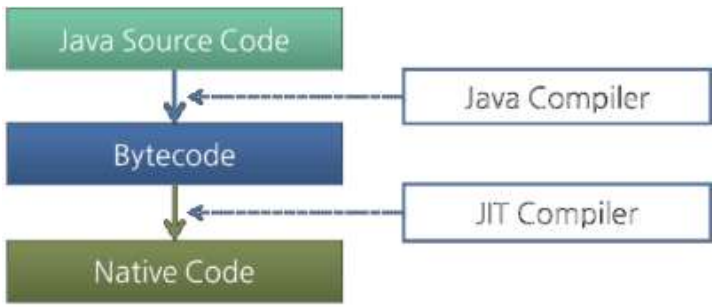
```
while(true) {
        bytecode b = bytecodeStream[pc++];
        switch(b) {
                case iconst_1: push(1); break;
                case iload_0: push(local(0)); break;
                case iadd: push(pop() + pop()); break;
        }
}
```

NOT that simple ...

The Hotspot JVM

Different execution techniques:

- interpreting
- just-in-time (JIT) compilation
- adaptive optimization (determines "hot spots" by monitoring execution)



The Hotspot JVM

JIT compilation:

- triggered asynchronously by counter overflow for a method/loop (interpreted counts method entries and loopback branches)
- produces generated code and relocation info (transferred on next method entry)
- in case JIT-compiled code calls not-yet-JIT-compiled code control is transferred to the interpreter

The Hotspot JVM

JIT compilation:

- compiled code may be forced back into interpreted bytecode (deoptimization)
- is complemented by On-Stack Replacement (turn dynamically interpreted to JIT compiled code and vise-versa - dynamic optimization/deoptimization)
- is more optimized for server VM (but hits start-up time compared to client VM)

The Hotspot JVM

JIT compilation flow (performed during normal bytecode execution):

1) bytecode is turned into a graph

2) the graph is turned into a linear sequence of operations that manipulate an infinite loop of virtual registers (each node places its result in a virtual register)

The Hotspot JVM

JIT compilation flow (performed during normal bytecode execution):

3) physical registers are allocated for virtual registers (the program stack might be used in case virtual registers exceed physical registers)

4) code for each operation is generated using its allocated registers

The Hotspot JVM

Typical execution flow (when using the **java**/**javaw** launcher):

1. Parse the command line options
2. Establish the heap sizes and the compiler type (client or server)
3. Establish the environment variables such as CLASSPATH
4. If the java Main-Class is not specified on the command line fetch the Main-Class name from the JAR's manifest
5. Create the VM using **JNI_CreateJavaVM** in a newly created thread (non primordial thread)

The Hotspot JVM

Typical execution flow (when using the **java**/**javaw** launcher):

6. Once the VM is created and initialized, load the Main-Class
7. Invoke the **main** method in the VM using **CallStaticVoidMethod**
8. Once the **main** method completes check and clear any pending exceptions that may have occurred and also pass back the exit status
9. Detach the main thread using DetachCurrentThread,

   by doing so we decrement the thread count so the **DestroyJavaVM** can be called safely

Demo

Summary

No JVMs were injured during this presentation …

Q & A

Thank you !

Resources

The Java Virtual Machine Specification Java SE7 Edition

http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

Mani Sarkar 's collection of Hotspot-related links

https://gist.github.com/neomatrix369/5743225

Hotspot documentation

http://openjdk.java.net/groups/hotspot/

Resources

Hacking Hotspot in Eclipse under Ubuntu 12.04

http://neomatrix369.wordpress.com/2013/03/12/hotspot-is-in-focus-again-aka-hacking-hotspot-in-eclipse-juno-under-ubuntu-12-04/

Garner R., The Design and Construction of High Performance Garbage Collectors, PhD thesis

https://digitalcollections.anu.edu.au/handle/1885/9053

Resources

Shi Y., Virtual Machine Shutdown: Stack versus Registers, PhD Thesis
https://www.cs.tcd.ie/publications/tech-reports/reports.07/TCD-CS-2007-49.pdf

Compiler and JVM Research at JKU

http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/

Resources

Xavier Leroy, Java bytecode verification: algorithms and formalizations
http://pauillac.inria.fr/~xleroy/publi/bytecode-verification-JAR.pdf

FOSDEM 2007

http://openjdk.java.net/groups/hotspot/docs/FOSDEM-2007-HotSpot.pdf

Resources

The Architecture of the Java Virtual Machine

http://www.artima.com/insidejvm/ed2/jvm2.html