JavaOne℠

Sun's Worldwide Java Developer Conference

Java™ Internals

Frank Yellin
Tim Lindholm
JavaSoft

About This Talk

- The JavaSoft implementation of the Java Virtual Machine (JDK 1.0.2)
- Some companies have "tweaked" our implementation
- Alternative implementations also exist
  - Microsoft
  - Natural Intelligence
  - *Companies we don't even know about*

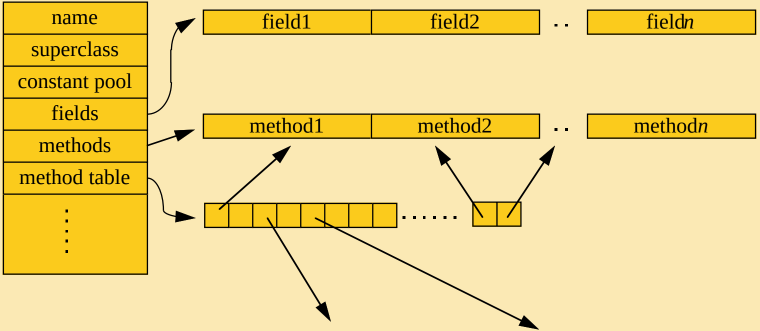
Overview

- Class file format
- Object format
- Memory layout
- Instruction Set
- Security
- Security Manager
- Garbage collection
- Native methods
- Class loading
- Threads and monitors

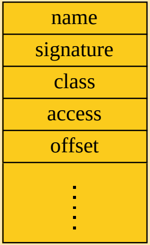
Class File (External)

- Machine independent
  - “Stream of bytes”
  - No byte sex dependency
  - No pointer size dependency
- Constant Pool
- Attributes
- http://java.sun.com/newdocs.html

Class Format (Internal)



Field Information



name

signature

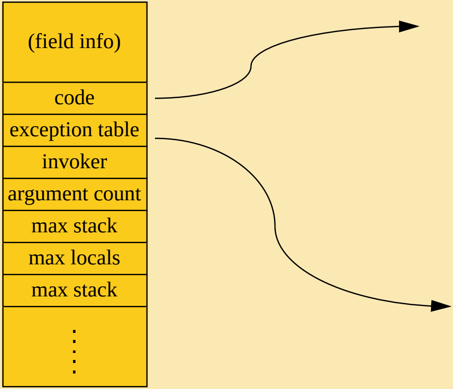
class

access

offset

⋮

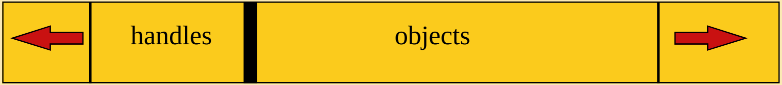
Method Block



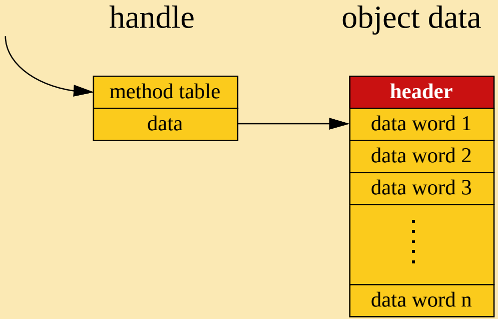
Memory Areas

- Malloc’ed space
  - Methods, classes
  - Random data structures
- Java heap
  - Handle space
  - Object space
- Moving more stuff to the Java heap

Java Heap Layout

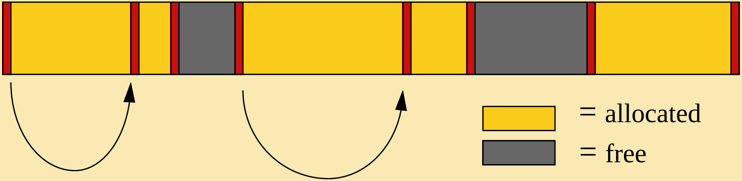


Object Format



- Java objects refer to other objects via handles

Object Data Header



- Next fit allocation
- Both allocated and free space kept in linked list
- (Handle allocation, however, is trivial)

Instruction Set

- Instructions are typed
- Operate on the stack and local variables
- Non-orthogonal
- All arithmetic operations use the stack

Instruction Set *(cont.)*

- The Java Virtual Machine is not:
  - The world's greatest virtual machine
- We wanted the Java Virtual Machine instruction set to be:
  - Easy to verify
  - Easy to compile
  - Easy to interpret
  - Portable
  - Contain extensive type information

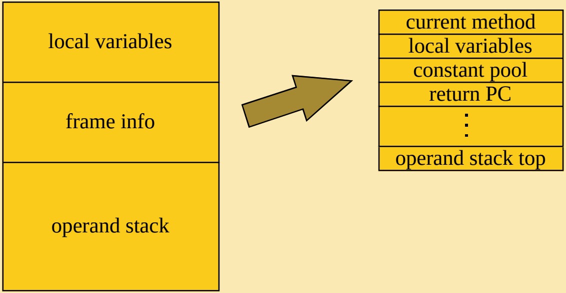
Instruction Categories

- Load/store local variable
- Arithmetic and type conversion
- Conditional/unconditional branch
- Object creation and manipulation
- Array creation and manipulation
- Method invocation
- Stack manipulation

Execution

- Stack machine model
  - One Java stack per thread
  - Java stack contains *frames*
  - All instructions use the *operand stack*
  - Local variables are per method invocation
- Method invocation arguments are pushed on the operand stack
- Separate (per thread) “C” stack

Java Stack Frames



local variables

frame info

operand stack

current method

local variables

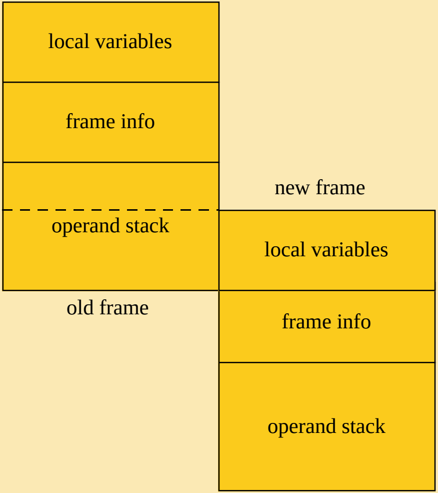
constant pool

return PC

⋮

operand stack top

Java Stack Frames (*cont.*)



Linking, Loading, and Initialization

- Class files are loaded as needed
- Class files are loaded from the same source as the code requesting the class
- Several stage process

Constant Pool Resolution

- All references to `String`'s, methods, fields, and (most) constants are done through the constant pool
- References are initially symbolic
- References are "snapped" the first time an instruction is executed

Constant Pool *(cont.)*

- Causes new classes to be loaded, as necessary

Security

- Low-level security
  - The definition of the language
  - The verifier
- High-level security
  - The Security Manager

When Is the Verifier Run?

- Any class files that comes from an untrusted source (such as the network)
- Not on class files that come from your local disk
- Choice really depends on your interpreter or browser and on your security requirements

Why a Verifier?

- Hostile compilers (or just broken ones)
- Improve the speed of the interpreter
- Protect against changing APIs
- Protect against stack overflow, underflow, etc.

Verifier Passes

1. Check the class files syntactically
2. Check the class files semantically
3. Check the bytes codes
4. Runtime checks [optimization]

But. . .

~~Halting Problem~~

Java High-Level Security

- Code that is downloaded over the Net is *untrusted*
- The Java language runs untrusted code in a trusted environment
- The Security Manager keeps a watchful eye on untrusted code

The Security Manager

- The Security Manager is the cop:
  - Implements *security policies*
  - Throughout the system, security checks are done at sensitive points
  - The system’s Security Manager performs those checks

What Does it Protect?

- The Security Manager *restricts access* to:
  - The file system
  - The network
  - Other dangerous runtime calls:
    - Setting the Security Manager
    - Exiting
    - Executing programs

How Does it Work?

- The Security Manager can:
  - Scan the stack
  - Check the caller's:
    - Thread group
    - Namespace
    - Digital signature
- Or anything else: *it's extensible*!
- You can provide your own

Forthcoming: Signed Classes

1. User declares *trusted entity*
2. Entity *signs* Java-powered applet or application
3. Applet or application is now trusted, and thus granted *more privileges*

Java-based technology enables *true internet applications*!

Garbage Collection

- Three causes of garbage collection
  - Synchronous
  - Asynchronous
    - (Not always as useful as you might think)
  - Explicit

GC Buzzwords

- Conservative or Exact?
- Compacting or Non-compacting?
- Generational?
- “Stop and Copy”?
- Real-time?

Sun’s Garbage Collector

Sun’s garbage collector is:

partially conservative,

optionally compacting,

non-generational,

stop and copy,

and generally pretty fast

Finalization

- A generalization of garbage collection
- Normally asynchronous, may be synchronous
- Guarantees?

Possible New Stuff

- Better low-memory behavior
- Heap contraction (staying small)
- Class garbage collection
- Tunable garbage collection
- Garbage collector “plug ins”
- Better algorithms (faster, generational, etc.)

Native Methods

- Declaring native methods in the Java language
- Defining native methods in C
- The “javah” glue
- Dynamic linking at runtime

Declaring Native Methods in Java:

```
public native int read();

public static native double
  sin(double x);
```

Defining Native Methods in C

```
#include “java_io_FileInputStream.h”

long
java_io_FileInputStream_read(
    Hjava_io_FileInputStream *this) {
  . . .
}
```

Defining Native Methods in C

```
#include “java_lang_Math.h”

double
java_lang_Math_sin(
    Hjava_lang_Math *this, double f) {
  return sin(f)
}
```

javah

- To generate .h include files

```
$ javah -stubs
      java.lang.FileInputStream
```

- To generate “glue” files

```
$ javah
      java.lang.FileInputStream
```

Runtime Dynamic Linking

- The “glue” file generated by javah must be included in the shareable library or dll
- What happens internally

Class Loading

- System classes
- The `ClassLoader` class
  - Used by HotJava to download classes over the network
  - Can be used by sophisticated applets to create classes “on the fly”

Threads

- Priority preemptive
  - Not guaranteed time-sliced
- Use platform facilities when possible
- Don’t specify what we can’t deliver
- Program defensively

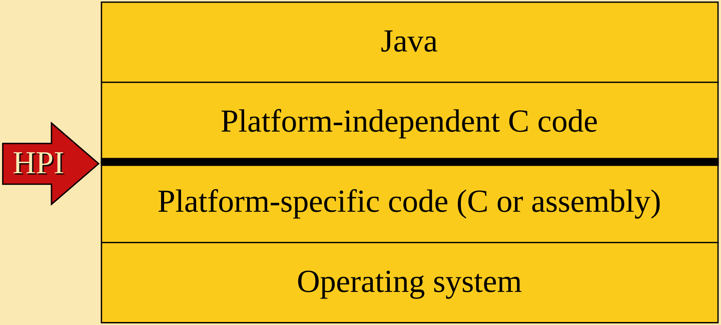
Threads Implementations

- Solaris
- Windows 95/Windows NT
- MacOS

Monitors

- Java’s synchronization primitive
- Use platform facilities
- Monitor cache
- Implementations
  - Solaris
  - Windows 95/Windows NT
  - MacOS

The Host Programming Interface



Java

Platform-independent C code

Platform-specific code (C or assembly)

Operating system

Conclusions

- The Java Virtual Machine
  - Synthesis of successful ideas from other languages and other projects
  - Designed to meet a goal, not to be aesthetically pure

For More Information. . . .

The Java Virtual Machine Specification, by Tim Lindholm and Frank Yellin

The Java Language Specification, by James Gosling, Bill Joy, and Guy Steele

`http://java.sun.com/newdocs.html`

JavaOne℠

Sun's Worldwide Java Developer Conference