THE MAXINE VIRTUAL MACHINE

Dr. Bernd Mathiske, Senior Staff Engineer

TS-5169

Java

Sun microsystems

Welcome by Sun, welcome to the Maxine

Where have you been?

It's alright, we know where you've been.

You've been in vi and emacs, filling in macros,

Provided with make, scouting for ants.

You wrote a compiler to punish yourself,

And you didn't like coffee,
and you know you're nobody's user.

What did you dream?

It's alright, they told you what to dream.

You dreamed of a big nerd,

He wrote some mean code,

He always ate chili peppers,

He loved to run his benchmarks.

So welcome to the Maxine!

VM Implementation Challenges

- Lack of modularity
- Pervasive features with intricate interdependencies
  - Code generation, Application Binary Interfaces (ABI), Garbage Collection (GC), class loaders, security, threads, safepoints, synchronization, canonicalization…
- Platform dependencies
- Pressure to optimize performance leads to more complexity
- Much low-level programming
  - Explicit CPU and memory operations
  - Complex OS calls (signal, mmap, mprotect…)
  - Assembly code
  - Manipulation of intermediate representations
- High potential for hard-to-find bugs

- … <insert several more slides about how hard this is> …

As the sessions and exhibits at JavaOne℠ conference demonstrate, Java™ application and library programmers enjoy great productivity and agility advantages based on their choice of language, platform, OO patterns, tools and build process.

But what about VM developers?

How can we bring these benefits to them?
How do we disentangle intricately interdependent VM features?
How do we keep a VM modular without performance loss?
How do we debug large amounts of optimized machine code?
How do we overcome the lack of low-level features in the Java language?
How...

GOAL

Let's start with meta-circular VM design, then push the envelope.

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- Java Developer Kit (JDK™) Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Maxine Open Source Research VM

- GNU General Public License version 2
  - License-compatible with OpenJDK™
- Currently builds on JDK 1.6 release
- Not a fully compliant Java™ Virtual Machine (yet)
- Early access to pre-alpha source code:

```
https://maxine.dev.java.net/
```

Maxine Platforms

- *Supported:*
  - Solaris™ Operating System / SPARC® Technology
  - Solaris OS / x64
  - Xen / x64
- *Under development:*
  - Solaris OS / x32
  - Linux / x64
  - Linux / x32
  - Darwin / x64
  - Darwin / x32
- *Contemplated:*
  - Xen / x32
  - ARM
  - PowerPC
  - Windows

Pre-Alpha Release: Proof-Of-Concept

- Miscellaneous Meta-Circular Design Aspects
- Bootstrap
- Configurability
- Low-Level object interfaces and layering
- Unsafe features
- Portable JIT
- Safepoint mechanism
- ...

Not Yet

- Performance
- JCK Compliance
- JDK software's `java` loader
- All Java™ Virtual Machine interfaces
- Competitive GC
- Memory-Management Framework
- OSR, dynamic de-optimization
- Inline caching
- ...

- Profiling and tuning tools
- Multi-level debugging
- Asynchronous debugging
- Lock analyzer
- Heap analyzer
- ...

- JDK 1.7 release

Maxine VM Research Areas

- Addressed today:
  - Modularity
  - Tool support
  - Compilation
  - Garbage collection
  - Concurrency
  - Control flow
  - Data structures
  - Hypervisor
  - …
- Anticipated:
  - Task Isolation
  - Resource control
  - Adaptive runtime behavior
  - Reliability
  - Real-time
  - Multi-language
  - Emulation
  - …

Maxine Innovations

- Architecture:
  - Flexible configuration
  - Java environment interfaces for primitive operations
  - Portable fast JIT derived from optimizing compiler
  - Portable generation of efficient inlineable Java Native Interface (JNI™) implementation and reflective invocations stubs
  - Byte code generation for exception dispatching and synchronized methods
- Tools and Libraries:
  - Assembler and disassembler framework for the Java platform
  - Integrated low-level debugging and object browsing (Inspector)
- Miscellaneous Runtime Mechanisms

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Maxine Development

- IDE-powered (NetBeans™ IDE, Eclipse, IntelliJ, JBuilder)
- Java Technology powered
  - Almost all source code is written in the Java language
  - Integrates with JDK 6 release
  - Leverages Java Technology 5/6 features
  - OO-patterns
- Maxine Inspector

Maxine Inspector

- A kind of Serviceability Agent / Object Browser / Debugger
- Starts and inspects a new VM or inspects a boot image
  - planned: attach to a running VM, inspect core dump
- Displays objects, memory content, disassembled methods
  - planned: source code
- Queries meta data: class registry, code manager
- Displays threads, register contents, stack frames
- Performs instruction single stepping
- Breakpoints at several code representation levels

Maxine Inspector Demo – Part 1:

Object Browsing

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Meta-Circular VM Design

- The VM is written in the same language it executes
- The VM uses JDK packages
  and implements their downcalls into the VM
- Built around an optimizing compiler (not an interpreter)
- The optimizing compiler in the VM translates itself

Bootstrapping

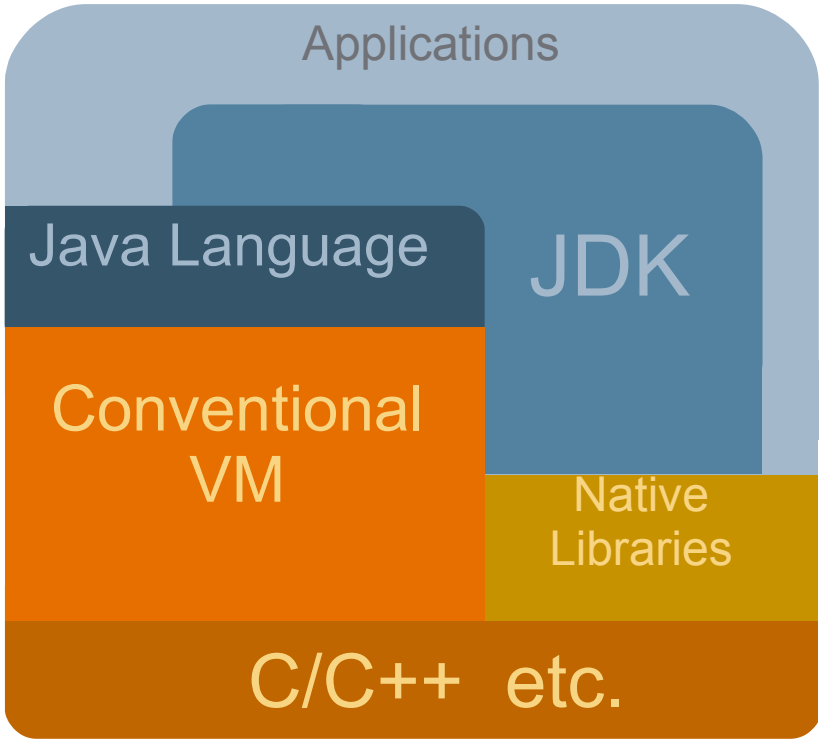
- Developer writes VM source code
- `javac` produces class files
- Boot Image Generator reads class files,
  creates VM data structures and code as Java objects,
  writes binary representations of these to a boot image file
- Boot image contains all necessary runtime functionality
  including class loader, compiler, GC, etc.

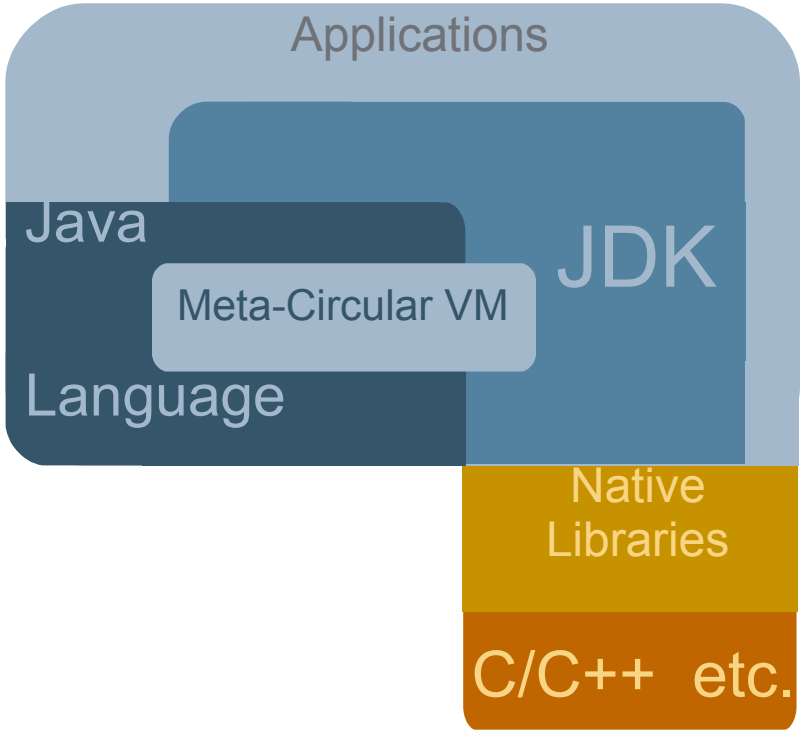
Very Limited C Code

- Simple way to create an executable for the OS's loader
- Starts up the VM by loading its boot image (mmap) and transferring control to it
- Shields some native methods and JNI API functions from the C preprocessor and C varargs
- Implements the “JVM” interface to native libraries
- Comes with easy-to-use make files

- This thin C “substrate” could later be removed by writing more platform-dependent Java code

Conventional vs. Meta-Circular





Immediate Payoffs

- All internal data structures expressed by Java objects
- No object handles in VM code to guard against GC
- Genuine Java code exception handling
  (not C/C++ macros or longjmp())
- No code and data structure duplication in C/C++
- Reuse VM code in companion tools (Inspector)

WEOO: **W**rite **E**verything **O**nly **O**nce

- Reuse runtime methods for different compilers by translation instead of manual re-implementation
- Diminished compiler interface
  - Only primitive operations and ABI specifics need porting
- Meta-evaluation by reflective invocation

Performance Potential

- The VM further optimizes itself by dynamic re-compilation
- Callbacks into the runtime are no longer native
- VM code can be inlined into app code and optimized together
- Optional runtime code features can be turned on and off by dynamic re-compilation
  - no overhead while off
- More efficient reflection

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Configurable Components ("Schemes")

- Garbage collector
- Object layout
- Object reference representation
- Read and write barriers
- Fast JIT
- Optimizing compiler
- Optimizing compiler's ABI
- Dynamic re-compilation policy
- Method call trampolines (for dynamic compilation)
- Thread synchronization (object monitors)
- Startup sequence

Configuration Mechanism

- At VM build time, select which alternative package to load for a particular “scheme”
- The package designates one class to implement the given `Scheme` interface
- This class is then loaded, initialized and instantiated
- The singleton instance represents the scheme to the VM
- Interface calls to the scheme instance are disambiguated and inlined by the optimizing compiler, which is guided by annotations
- A facade class may wrap the interface in static methods
- Implementation flexibility without performance overhead!

Configurable Cross-Compiling

- Operating system
- Instruction Set Architecture
- 64-bit vs. 32-bit
- Endianness
- Alignment

Guest VM

http://research.sun.com/projects/dashboard.php?id=185

- Sun Labs Project led by Mick Jordan

- Maxine VM runs as Xen Guest
- Runs a network stack written in the Java language
- Ported runtime substrate and inspector support in C to Xen
  - Uses a small micro-kernel providing threads, memory, I/O
  - File I/O via inter-guest communication to sibling guest OS
- Maxine Inspector inspects and controls debuggee guest VM via inter-guest communication

- Same Inspector functionality as on Solaris OS: single-step, find threads, read registers, read memory, ...

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Java 5 Platform Features Used by Maxine

- Annotations (JSR 175)
- Type Parameters (JSR 14)
- Return Type Overloading (JSR 14)
- Enhanced for Loop (JSR 201)
- Autoboxing/unboxing (JSR 201)
- Enums (JSR 201)
- Varargs (JSR 201)
- Static Import (JSR 201)

Maxine Annotations

- @INLINE
- @NEVER_INLINE
- @BUILTIN
- @SNIPPET
- @METHOD_SUBSTITUTIONS
- @SUBSTITUTE
- @SURROGATE
- @CONSTANT
- @CONSTANT_WHEN_NOT_ZERO
- @ACCESSOR
- @TEMPLATE
- @C_FUNCTION
- @JNI_FUNCTION

- @Generated
- @Hypothetical
- @JavacSyntax
- @JdtSyntax

- @FOLD
- @INITIALIZE
- @INSPECTED
- @PROTOTYPE_ONLY
- @TRAMPOLINE
- @UNSAFE
- @WRAPPER
- @WRAPPED

Source Code Example (1)

```
public abstract class
    EirSomeAllocator<EirRegister_Type extends EirRegister> extends
        EirAllocator<EirRegister_Type> {

    protected abstract PoolSet<EirRegister_Type> allocatableRegisters();
    ...
}
```

```
public final class SPARCEirSomeAllocator extends
   EirSomeAllocator<SPARCEirRegister> {
   ...
}
```

Source Code Example (2)

```
public enum Endianness {

    LITTLE {
        @Override
        public short readShort(InputStream stream) throws IOException {
            final int low = readByte(stream) & 0xff;
            final int high = readByte(stream);
            return (short) ((high << 8) | low);
        }
        ...
    },
    BIG {
        @Override
        public short readShort(InputStream stream) throws IOException {
            final int high = readByte(stream);
            final int low = readByte(stream) & 0xff;
            return (short) ((high << 8) | low);
        }
        ...
    }

    public abstract short readShort(InputStream stream) throws IOException;

}
...

dataModel.endianness().readShort(stream);
```

Agenda

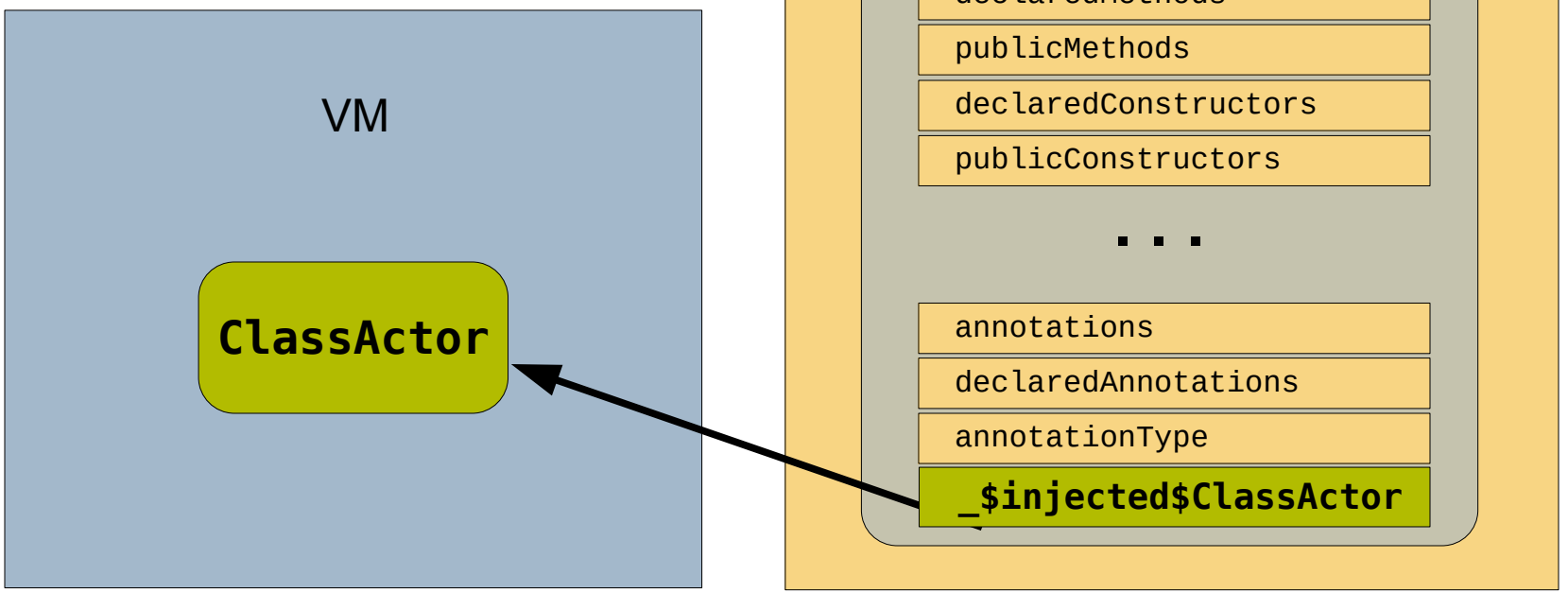
- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

JDK Software Integration

- No JDK source code changes
- No class/jar file changes
- Simplified implementation of some features
- Seamless bidirectional reuse between JDK library and VM
- No native methods for downcalls into the VM, no call overhead
- JDK library implementation methods in the VM are subject to dynamic optimization, especially inlining
  (just like the rest of the VM and everything else)

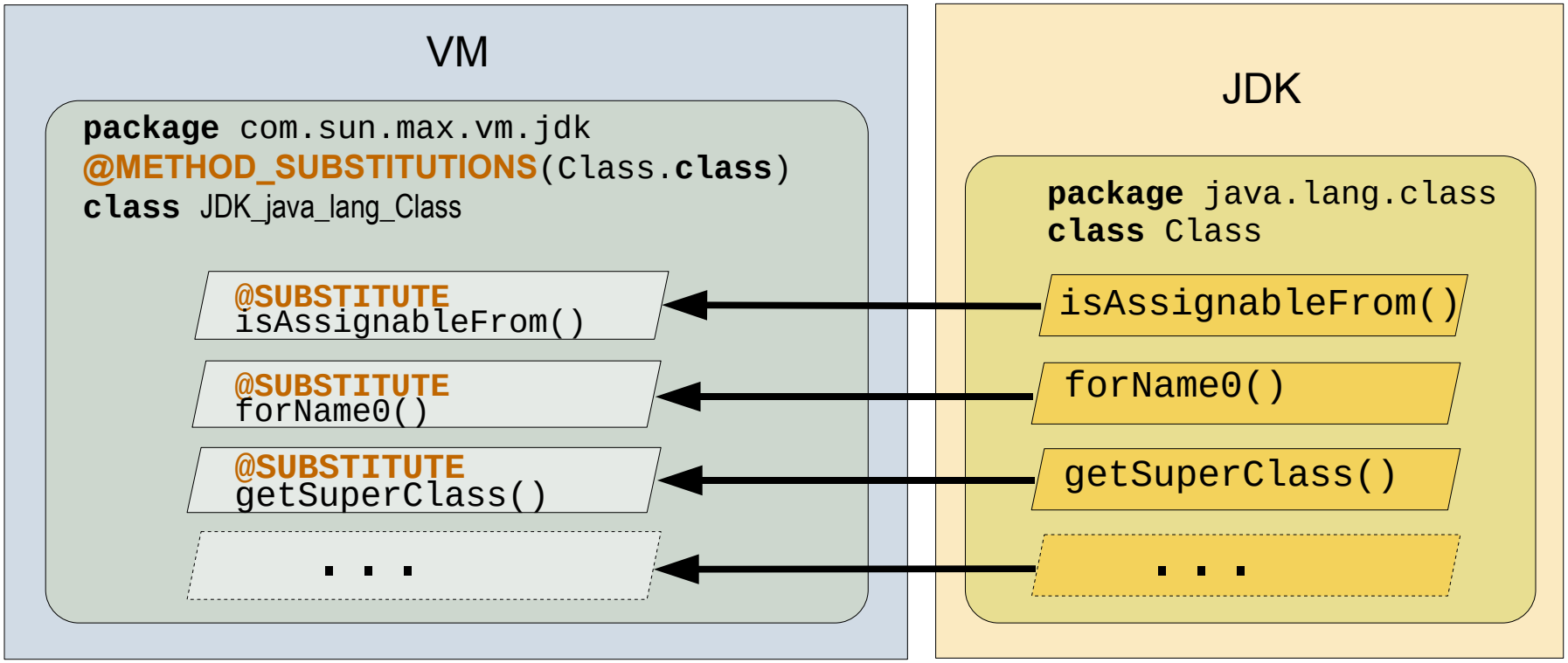
Injected Fields

The Maxine VM synthesizes and injects extra fields into core classes to link instances to internal representations



Method Substitution

- Guided by annotations, the VM substitutes certain JDK library methods with alternative implementations and compiles those in their stead
- It does not matter whether the original methods are native



Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Low Level Programming Support

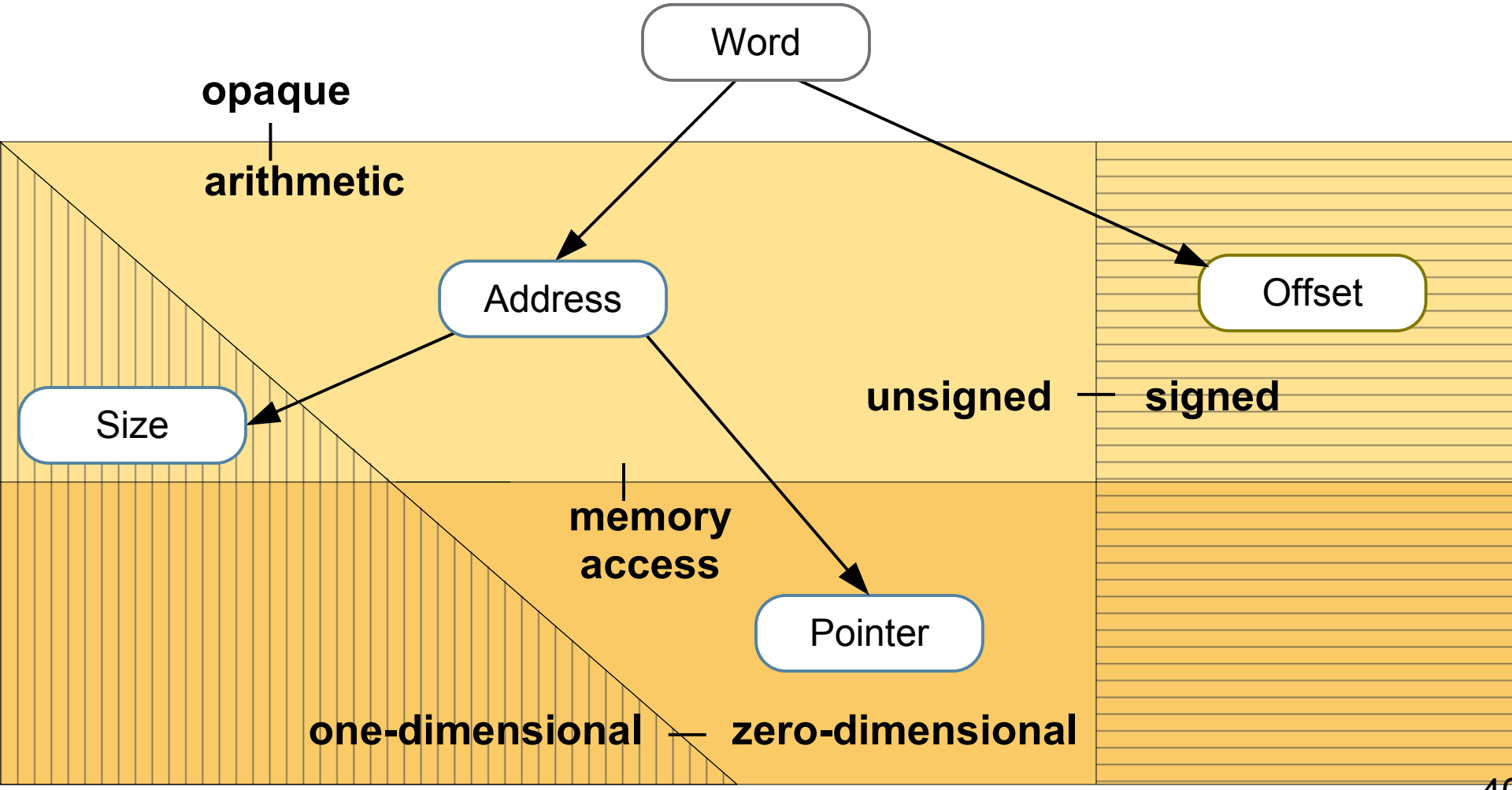
- Static type loophole
- Unsafe type loophole

- Unboxed word types
  - 32 or 64 bits, depending on VM configuration
  - Unboxed representation like primitive types
  - Method call syntax like boxed object types

- Raw memory allocation and deallocation
- Virtual memory mapping and unmapping

- Assemblers, Disassemblers

Word-Sized Primitive Types



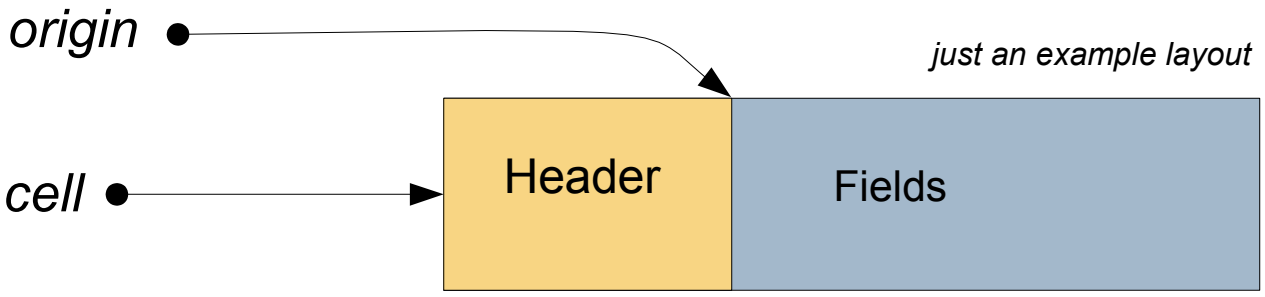
Layered Views on Objects

java.lang.Object — what the Java application program(mer) sees

Reference — what the mutator primarily uses:
low-level operations with read/write barriers

Grip — what the garbage collector primarily uses:
low-level operations without read/write barriers

Pointer



The Object Interface

```
public interface Accessor { ...
    boolean isZero();
    byte readByte(Offset offset);
    int readInt(int offset);
    Word readWord(Offset offset);
    Reference readReference(int offset);
    void write Boolean(Offset offset, boolean value);
    void writeDouble(int offset, double value);
    void writeReference(int offset, Reference value);
    ...
}
```

implemented by these classes (and all their subclasses):

Reference **Grip** **Pointer**

Low Level Code Example: sun.misc.Unsafe

```
package com.sun.max.vm.jdk;

import com.sun.max.unsafe.*;
...

@METHOD_SUBSTITUTIONS(sun.misc.Unsafe.class)
final class JDK_sun_misc_Unsafe {
    ...

    @SUBSTITUTE
    public Object getObject(Object object, long offset) {
        if (object == null) {
            return Pointer.fromLong(offset).getReference().toJava();
        }
        final Reference r = Reference.fromJava(object);
        return r.readReference(Offset.fromLong(offset)).toJava();
    }

    ...
}
```

Low Level Code Example: JNI API Function

```
public final class JniFunctions {
    ...

    @JNI_FUNCTION
    private static Pointer GetByteArrayElements(Pointer env,
                                                JniHandle array,
                                                Pointer isCopy) {
        if (!isCopy.isZero()) {
            isCopy.setBoolean(true);
        }
        final byte[] a = (byte[]) array.get();
        final Pointer pointer = Memory.allocate(a.length);
        for (int i = 0; i < a.length; i++) {
            pointer.setByte(i, a[i]);
        }
        return pointer;
    }

    ...
}
```

Maxine Assembler System

- Used by Maxine's optimizing compiler
- Java 5/6 technology packages
- Assemblers, Disassemblers
- Generator framework
- Automated testing
- Almost complete instruction sets:
  SPARC processor (32/64), AMD64, IA32, PowerPC (32/64),
  (ARM under development)
- Earlier version: Project Maxwell Assembler System
  - `https://maxwellassembler.dev.java.net`

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Optimizing Compiler Highlights

- Layered architecture
- Reduced compiler interface
- Meta-evaluation by reflective invocation
- Annotation-driven optimizations
- Interpreters for intermediate representations
- Continuation passing style
- No intermediate representation manipulation outside the translator and optimizer
- Almost no hand-written assembly code in the runtime
- Portable register allocator framework

Bytecode Generation

- Exception dispatch
- Synchronized methods
- Reflective invocation
- Native method invocation

=> streamlined compiler,
translating bytecodes without incongruous diversions

Ultra-Light JIT

- Goal: produce code as quickly as possible, hopefully preventing the need for an interpreter
- Single pass!
- Code quality is of secondary concern
- Closely matches the JVM spec's execution model
- No stack map production while generating code
  - Stack maps can be filled in at GC safepoints as needed, by abstract interpretation without allocating

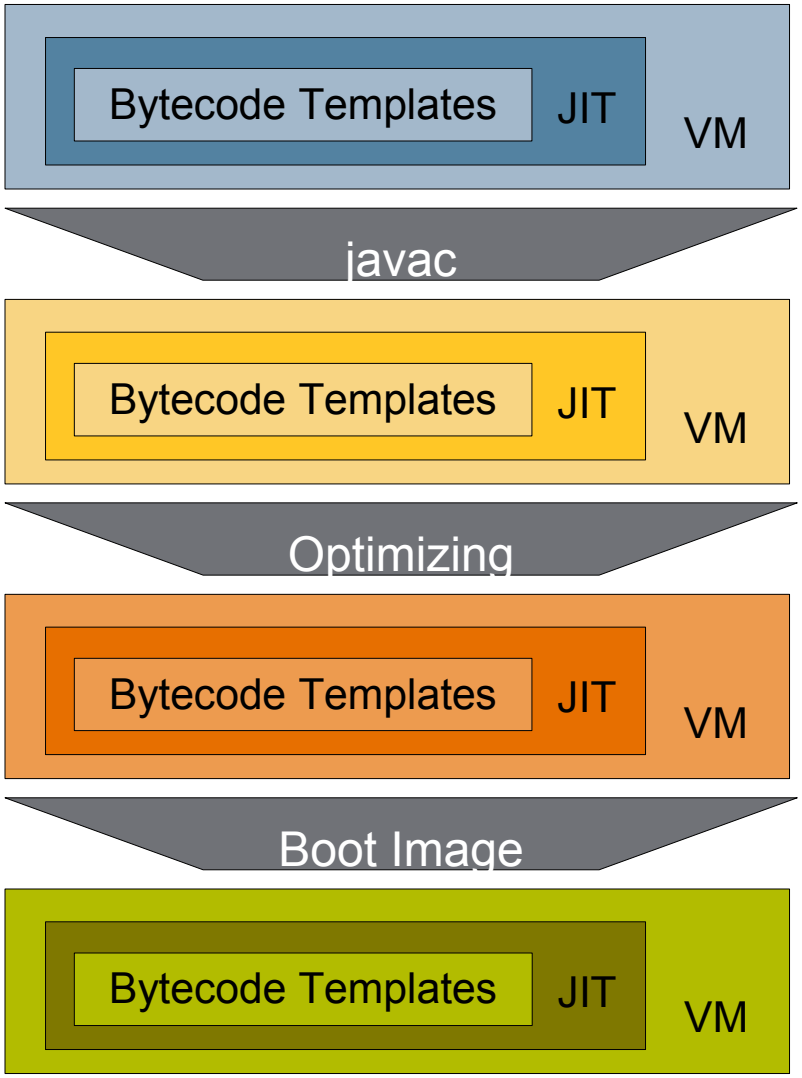
Portable JIT Implementation

- Copy machine code snippets from a template table
- Templates generated by the optimizing compiler at VM build time from Java source code
- One-instruction template prefix arguments specify local variable indices, constant pool indices
- Adapter frames mediate between stack-oriented JIT code and register-oriented optimized code and vice-versa

JIT Portability

	Porting Effort	Variations
ABI-sensitive: • Stack walking procedures • Adapter frame layout • Prefix argument instructions	*moderate*	*few*
Platform: • Prefix argument instructions are ISA-specific • Some byte codes hand-coded, e.g., branches • Templates written in portable Java source code	*low*	*few*
Runtime features: • Templates reuse runtime code	*none*	*some* *many*

Building the JIT from Java Source Code




```
public static void iadd() {
    int value2 = JavaStackFrame.peekInt(0);
    int value1 = JavaStackFrame.peekInt(1);
    JavaStackFrame.removeSlots(1);
    JavaStackFrame.pokeInt(0, value1 + value2);
}
```

```
iconst_0
invokestatic JavaStackFrame.peekInt()
istore_0
iconst_1
invokestatic JavaStackFrame.peekInt()
istore_1
iconst_1
invokestatic JavaStackFrame.removeSlots()
iconst_0
iload_1
iload_0
iadd
invokestatic JavaStackFrame.pokeInt()
```

```
movsxd      rcx,[rsp]           0x48 0x63 0x0C 0x24
movsxd      rax,[rsp + 8]       0x48 0x63 0x44 0x24 0x08
addq        rsp,8               0x48 0x83 0xC4 0x08
add         eax,rcx             0x01 0xC8
mov         [rsp],rax           0x89 0x04 0x24
```

```
byte[]{0x48,0x63,0x0C,0x24,0x48,0x63,0x44,0x24,
       0x08,0x48,0x83,0xC4,0x08,0x01,0xC8,0x89,
       0x04,0x24}
```

Bytecode Template with **Unresolved** Constant

```
public static void igetfield(ReferenceResolutionGuard guard) {
    IntFieldActor fieldActor = ResolutionSnippet.ResolveInstanceFieldForReading.resolve(guard);
    Object object = JavaStackFrame.peekReference(0);
    JavaStackFrame.pokeInt(0, FieldReadSnippet.ReadInt.readInt(object, fieldActor));
}
```

```
aload_0
invokestatic ResolutionSnippet.ResolveInstanceField.resolveInstanceFieldForReading()
astore_1
iconst_0
invokestatic JavaStackFrame.peekReference()
astore_2
iconst_0
aload_2
aload_1
invokestatic FieldReadSnippet.ReadInt.readInt
invokestatic JavaStackFrame.pokeInt()
```

- How to improve the quality of such JIT code?
  - Conventional: manually change the JIT
  - Meta-circular: do not touch the JIT, improve the optimizing compiler, which needs to be done anyway

```
        mov     rdi,[-515] // custom argument

        mov     [rbp],rdi
        mov     rax,[rbp]
        mov     rcx,[rax + 32]
        xor     rax,rax
        cmp     rcx,rax
        jnz     L2:+43
        mov     rdi,[rbp]
        push    rbp
        call    resolve()
        pop     rbp
        mov     rax,[rbp]
        mov     rax,[rax + 32]
        mov     rcx [rbp]
        mov     rcx, rax
L1:     mov     rdx[rsp]
        movsxd  rcx,[rcx + 64]
        movsxd  rax,rdx[rcx]
        mov     [rsp],eax
        jump    L3:+17
L2:     mov     rax,[rbp]
        mov     rax,[rax + 32]
        mov     rcx,[rbp]
        mov     rcx,rax
        jmp     L1:-34
L3:
```

BytecodeTemplate with **Resolved** Constant

```
public static void igetfield(int offset) {
    Object object = JavaStackFrame.peekReference(0);
    JavaStackFrame.pokeInt(0, TupleAccess.readInt(object, offset));
}
```

```
iconst_0
invokestatic JavaStackFrame.peekReference()
astore_1
iconst_0
aload_1
iload_0
invokestatic TupleAccess.ReadInt.readInt
invokestatic JavaStackFrame.pokeInt()
```

```
mov        rdi,0x10        // custom argument: field offset

mov        rdx,[rsp]
movsxd     rax,rdx[rdi]
mov        [rsp],eax
```

- To do: top-of-stack caching
  - Idea: additional template parameter passing, e.g., `rdi == [rsp]`
  - Then this code will become possible:

```
mov        rcx,0x10
movsxd     rdi,rdi[rcx]
```

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Maxine Inspector Demo – Part 2:

Debugging

Agenda

- Open Source Release
- Maxine Development Environment
- Meta-Circular VM Design
- Configurability
- Use of New Language Features
- JDK Software Hookup
- Low-Level Features
- Compiler system
- Debugging Demo
- Summary

Summary

- New research VM
- Pervasive meta-circular design
- Configurable, malleable, maintainable, portable
- Integrates with JDK software
- Inspector tool: low-level debugging and object browsing

- Agile systems programming with Java technologies

- Pre-alpha open source release

http://research.sun.com/projects/maxine

THANK YOU

The Maxine Virtual Machine

Dr. Bernd Mathiske
Senior Staff Engineer
Principal Investigator, Maxine Project
Sun Microsystems Laboratories

TS-5169