18-640 Foundations of Computer Architecture

Lecture 17:
"Virtual Machine Design and Implementation"

Antero Taivalsaari, Nokia Fellow
November 4, 2014

➢Recommended References:

- Jim Smith, Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, June 2005.
- Matthew Portnoy, *Virtualization Essentials*, Sybex Press, May 2012

Electrical & Computer ENGINEERING

My Background with Virtual Machines

- Built virtual machines since the mid-1980s (initially as a hobby).
  - Main interests in the 1980s/early 1990s: Forth, Smalltalk, Self, other dynamic OO languages.
- In 1997, moved to California to work on Java VMs at Sun Microsystems.
  - Wrote the K Virtual Machine (KVM) at Sun Labs in 1998.
  - KVM became the starting point for Java 2 Micro Edition (J2ME / JavaME), a popular version of the Java platform for mobile devices; ultimately KVM ran in over a billion mobile devices (mostly mobile phones).
- Engineering manager of the J2ME/KVM virtual machine team at Sun's Java Software unit, 1999-2001.
  - Shipped the first product versions of the J2ME platform
  - Led the early J2ME standards activities (CLDC 1.0/1.1)
  - Co-author of the first Java Series book on J2ME.

"Ten Year Increments"

Goals of this Lecture

- Introduce you to the world of virtual machine (VM) design.
- Provide an overview of key technologies that are needed for constructing virtual machines, such as automatic memory management, interpretation techniques, multithreading, and adaptive compilation.
- *Caveat: This is a very broad area – we will only scratch the surface in this lecture.*

Topics to be Covered

- History and overview of VM design
- Memory management
- Interpretation/execution techniques
- Multithreading, synchronization and I/O
- Native interface & VM-internal runtime structures
- Adaptive compilation (only briefly)
- Designing for portability

Introduction

What is a Virtual Machine?

- A *virtual machine* (VM) is an "abstract" computing architecture or computational engine that is independent of any particular hardware or operating system.
- Software machine that runs on top of a physical hardware platform and operating system.
- Allows the same applications to run "virtually" on any hardware for which a VM is available.

Two Broad Classes of Virtual Machines

- There are two broad classes of virtual machines:

1) *System virtual machines* – typically aimed at virtualizing the execution of an entire operating system.
   - Examples: VMware Workstation, VirtualBox, Virtual PC

2) *Language virtual machines* (process virtual machines) – typically aimed at providing a portable runtime environment for specific programming languages.
   - Examples: Java VM, Dalvik, Microsoft CLR, V8, LLVM, Squeak

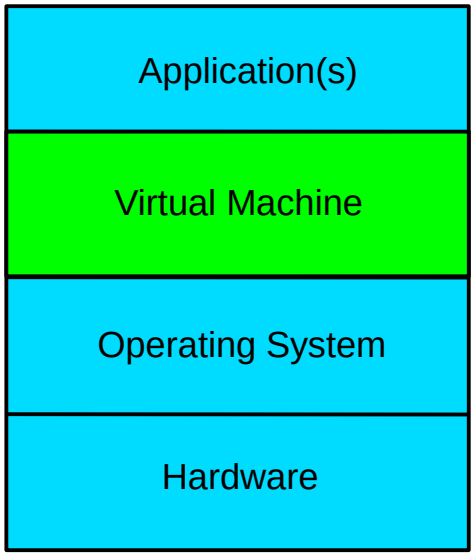
Focus in this lecture is on language VMs

Why are Virtual Machines Interesting?

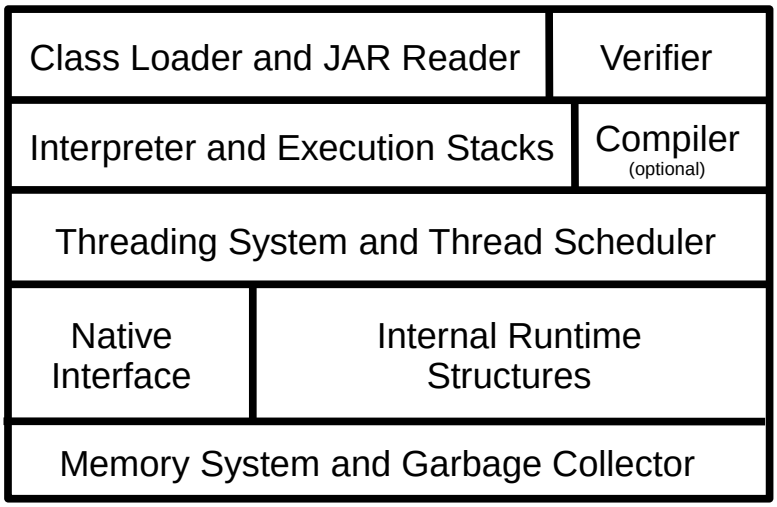
- Provide platform independence
- Isolate programs from hardware details
- Simplify application code migration across physical platforms
- Can support dynamic downloading of software
- Can provide additional security or scalability that hardware-specific implementations cannot provide
- Can hide the complexity of legacy systems

- Many interesting programming languages and systems are built around a virtual machine

Language VMs – Typical High-Level Architecture



Example: Components of a Java Virtual Machine (JVM)



VM vs. OS Design

- There is a lot of similarity between VM and operating system design.
  - The key component areas are pretty much the same (memory management, threading system, I/O, ...)
- A few key differences:
  - Operating systems are language-independent extensions of the underlying hardware. They are built to facilitate access to the underlying computing architecture and maximize the utilization of the hardware resources.
  - In contrast, language VMs implement a machine-independent instruction set and abstract away the details of the underlying hardware and the host operating system pretty much completely.

Languages that Use Virtual Machines

- Well-known languages using a virtual machine:
  - *Lisp* systems, 1958/1960-1980s
  - *Basic, 1964-1980s*
  - *Forth*, early 1970s
  - *Pascal* (P-Code versions), late 1970s/early 1980s
  - *Smalltalk*, 1970s-1980s
  - *Self*, late 1980/early 1990s
  - *Java*, late 1990s (2000's for Android)
- Numerous other languages:
  - ... *PostScript*, *TCL/TK*, *Perl*, *Python*, *C#*, ...

Highlight: Self Language and VM

- *Self* is a prototype-based variant of Smalltalk that was invented by David Ungar and Randall Smith in the end of 1980's; most of the implementation work was done at Sun Labs in the 1990's. See: http://selflanguage.org/.
- The Self language was so extremely dynamic that its implementers had to push the limits of VM technology very aggressively:
  - Adaptive compilation to speed up execution.
  - Generational garbage collection (originally invented by David Ungar in his Ph.D. work.)
  - Dynamic deoptimization to allow debugging of highly optimized programs.
  - Novel collaborative / visual programming and debugging environment (tightly integrated with the VM.)
- Many of the essential technologies that are used in today's mainstream virtual machines (e.g., for Java, Android) were invented by the Self group.
- Many of the students working in the Self team went on to accomplish great things in the industry later (e.g., the Java HotSpot VM, V8 JavaScript VM).

Designing and Implementing Virtual Machines

How are Virtual Machines Implemented?

- Virtual machines are typically written in "portable" and "efficient" programming languages such as C or C++.
- For performance-critical components, assembly language is used.
  - The more machine code is used, the less portability (duh…)
- Some virtual machines (Lisp, Forth, Smalltalk) are largely written in the language itself.
  - These systems have only a minimal core implemented in C or assembly language.
- Most Java VM implementations consist of a mixture of C/C++ and assembly code.

Virtual Machine Design Considerations

- Size
- Portability
- Performance
- Memory consumption
- Scalability
- Security
- ...

There are always trade-offs in VM design!

The Common Tradeoffs

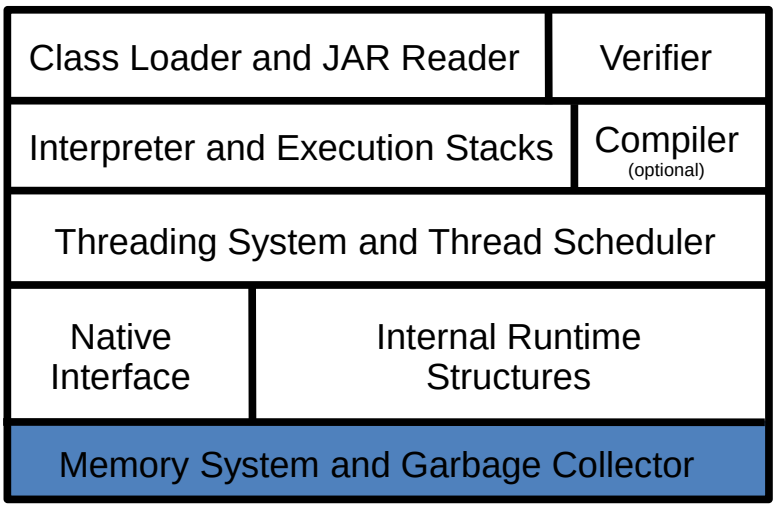
- Unfortunately, for nearly all aspects of the VM:
  - Simple implies slow
  - Fast implies more complicated
  - Fast implies less portable
  - Fast implies larger memory consumption

Examples of areas with significant tradeoffs:

- Interpretation
- Memory management
- Locking/Synchronization, exception handling
- Dynamic compilation, debugging

There are two "camps" of language VM designers:
(1) *speed enthusiasts* and (2) *portability enthusiasts*

Walkthrough of Essential Component Areas



Memory Management

Basic Memory Management Strategies

1) Static memory management
   - Everything allocated statically.

2) Linear memory management
   - Memory is allocated and freed in Last-In-First-Out (LIFO) order.

3) **Dynamic memory management**
   - Memory is allocated dynamically from a large preallocated "heap" of memory.

- Dynamic memory management is a prerequisite for most modern programming languages

Dynamic Memory Management

- In dynamic memory management, objects can be allocated and deallocated freely.
  - Allows the creation and deletion of objects in an arbitrary order.
  - Objects can be resized on the fly.
- Most modern virtual machines use some form of dynamic memory management.
- Depending on the implementation, dynamic memory management can be:
  - *Manual*: the programmer is responsible for freeing the unused areas explicitly (e.g., malloc/free/realloc in C)
  - *Automatic*: the virtual machine frees the unused areas implicitly without any programmer intervention.

Memory Management – Simple Example

```
public void foo() {
   MyClass object = new MyClass();
   object.doSomething();
   ...
   object = new MyClass();
   ...
}
```

The previously allocated object instance just became garbage!!

Automatic Memory Management & Garbage Collection

- Most modern virtual machines support *automatic dynamic memory management*.
- Automatic dynamic memory management frees the programmer from the responsibility of explicitly managing memory.
- The programmer can allocate memory without having to worry about deallocation.
- The memory system will automatically:
  - reclaim unused memory using a *garbage collector*,
  - expand and shrink data in the heap as necessary,
  - service weak pointers and perform finalization of objects (if necessary).

Benefits of Automatic Memory Management

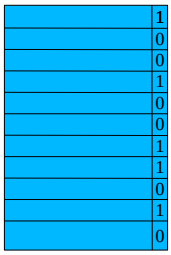
- Makes the programmer's life much easier
  - Removes the problems of explicit deallocation
  - Decreases the risk of memory leaks
  - Simplifies the use of abstract data types
  - Facilitates proper encapsulation
- Generally: ensures that programs are *pointer-safe*
  - No more dangling pointers
- Automatic memory management improves program reliability and safety significantly!!

Basic Challenges in Automatic Dynamic Memory Management

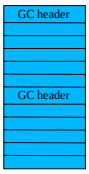
- How does the memory system know where all the pointers are?
- How does the memory system know when it is safe to delete an object?
- How does the memory system avoid memory fragmentation problems?
- If the memory system needs to move an object, how does the system update all the pointers to that object?
- When implementing a virtual machine, your VM must be able to handle all of this.

How to Keep Track of Pointers?

- The memory system must be able to know which memory locations contain pointers and which don't.
- Three basic approaches:

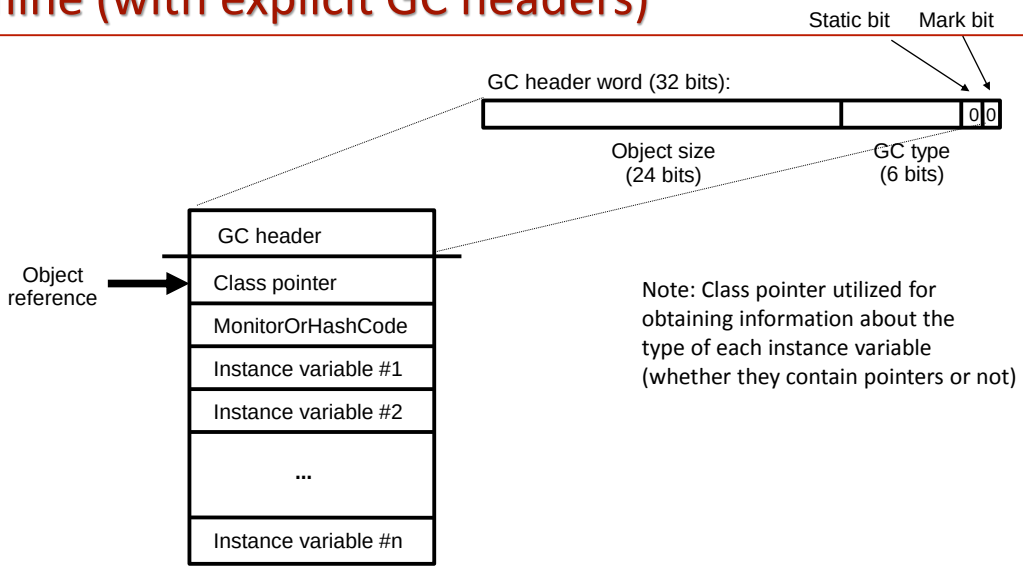


- In some systems, all memory words are *tagged* with pointer/type information.
- In some systems, objects have headers that contain pointer information.



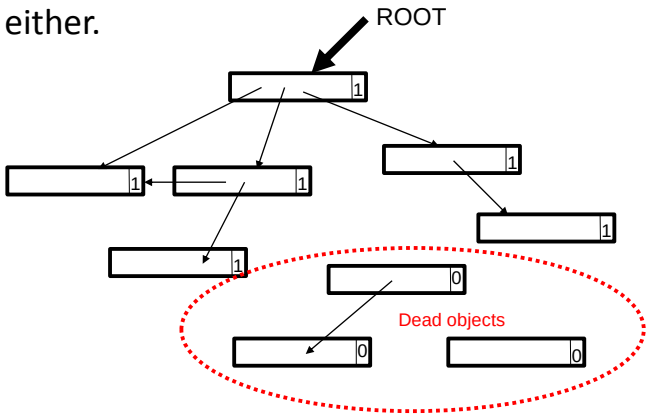
- In some systems, pointer information is kept in separate data structures.

Example: Object Layout in the K Virtual Machine (with explicit GC headers)



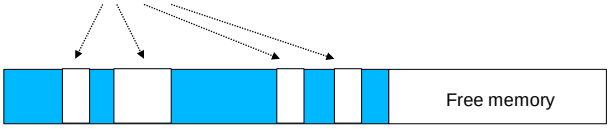
When Is It Safe to Delete An Object?

- Generally, an object can be deleted when there are no more pointers to it.
- All the dependent objects can be deleted as well, if there are no references to them either.



To Compact Memory or Not?

- When objects are deleted, the object heap will contain *holes* unless the heap is *compacted*.



- If a compaction algorithm is used, objects in the heap may move.
  - All pointers to the moved objects must be updated!
- If no compaction is used, the system must be able to manage free memory areas.
  - Often, a *free list* is used to chain together the free areas.
  - Memory allocation will become slower.
  - Fragmentation problems are possible!

Basic Heap Compaction Techniques

1) Two-finger algorithms
   - Two pointers are used, one to point to the next free location, the other to the next object to be moved. As objects are moved, a forwarding address is left in their old location.
   - Generally applicable only to systems that use fixed-size objects (e.g., Lisp).

2) Forwarding address algorithms
   - Forwarding addresses are written into an additional field within each object before the object is moved.
   - These methods are suitable for collecting objects of different sizes.

3) Table-based methods
   - A relocation map, usually called a *breaktable*, is constructed in the heap either before or during object relocation. This table is consulted later to calculate new values for pointers.
   - Best-known algorithm: Haddon-Waite breaktable algorithm; used in Sun's KVM.

4) Threaded methods
   - Each object is chained to a list of those objects that originally pointed to it. When the object is moved, the list is traversed to readjust pointer values.

5) Semi-space (copying) compaction
   - In copying collectors, compaction occurs as a side-effect to copying.

Fundamental Garbage Collection Algorithms

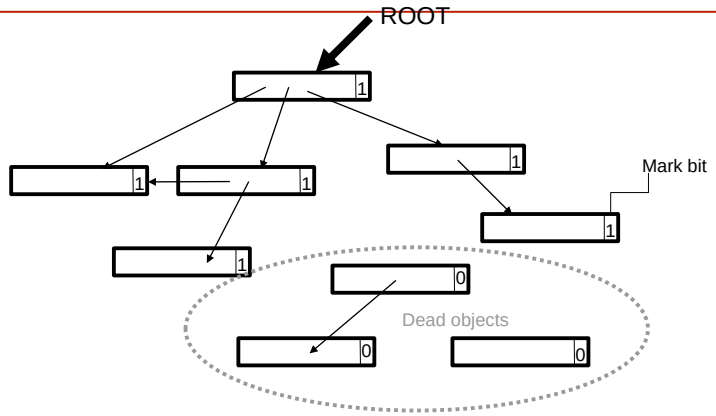
Garbage Collection: Fundamental Algorithms

- There are three "classical" categories of garbage collection algorithms:
  1) Marking collection
     - For instance, *mark-and-sweep*, *mark-and-compact*
  2) Copying collection
  3) Reference counting collection
- In addition, there are other algorithm categories that are variations of the above:
  4) Generational collection
  5) Hybrid/adaptive collection

1) Marking Collectors

- Marking collectors typically operate in two phases:
  - 1) In a *marking* phase, all reachable objects are marked "alive".
  - 2) In a *sweep* phase, all dead (unmarked) objects are added to a *free list*, and marks in live objects are cleared.
- Many marking collectors also perform a third phase:
  - 3) During *compact* phase, all the live objects are packed to the beginning of the heap.
- Marking collectors are not very efficient, but they are simple to implement and have many other desirable properties.

Example: Marking Collection



Phase 1: Starting from a set of *root objects*, mark all reachable objects alive.

Phase 2: Add all dead (unmarked) objects to a free list (and optionally compact memory). Clear all mark bits.

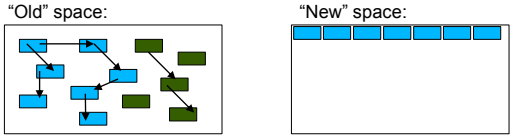
2) Copying Collectors

- Copying collectors divide memory into two (or more) *semi-spaces*.
- During collection, live objects are copied from one semi-space to another, leaving dead objects behind.
- Copying collectors are generally very efficient, since dead objects do not require processing.
- Copying collectors have other nice properties, such as the ability to automatically compact memory and group live objects together.
- However, copying collectors waste memory, since at least one semi-space must be empty.

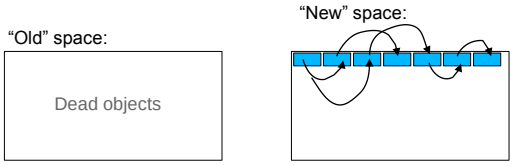
Example: Cheney Copying Collection

(Communications of the ACM, September 1970)

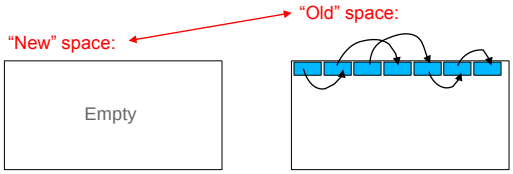
Phase 1: Starting from a set of root objects, copy all reachable (live) objects to a free semi-space.



Phase 2: Adjust pointers in the copied objects to point to correct locations, utilizing forwarding pointer information left in the old space ("pointer swizzling")



During the next garbage collection, the "old" space and "new" space are swapped, and copying is performed again.



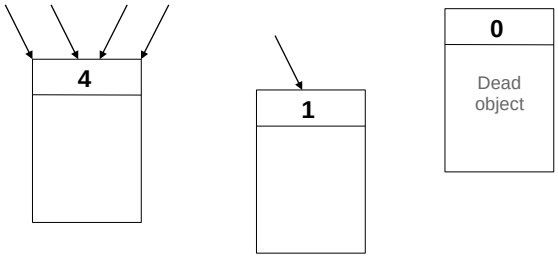
3) Reference Counting Collectors

- In reference counting garbage collection, each object keeps track of how many references are pointing to it.
- When the count becomes zero, the object can be reclaimed.
- Reference counting distributes the costs of memory management evenly during execution.
- However, assignment statements become rather expensive.
- Also, cyclic data structures are problematic.

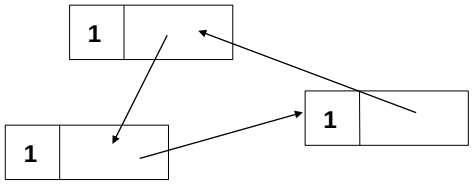
Example: Reference Counting

Each object has a reference count that holds the number of references to that object.

Each time an assignment is performed, reference count is incremented or decremented.

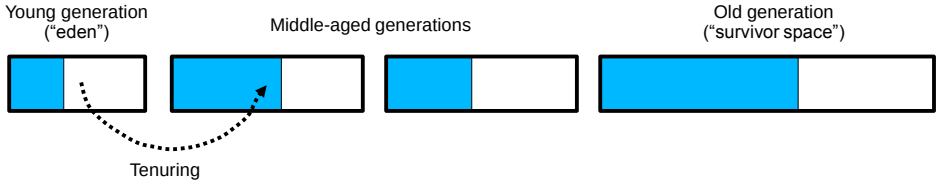


Problem: How to collect cyclic data structures?



4) Generational Collectors

- General observations about garbage collection:
  - Most objects die young!!
  - The more garbage collections an object survives, the more likely it is to stay alive for a long time.
- Modern garbage collectors utilize *object age* in determining the need for garbage collection.
  - Heap is divided into generations based on object age.
  - Objects are "tenured" (promoted) to older generations as they age.



5) Hybrid/Adaptive Collectors

- *Hybrid collectors* use a number of different collection algorithms.
  - For instance:
    - Young objects collected with a fast copying collector.
    - Old objects collected (infrequently) with a compacting collector that maximizes the amount of free available memory.
- *Adaptive collectors* gather statistics/feedback about previous garbage collections, and adapt the behavior of the collector dynamically based on the feedback.

Tradeoffs in Garbage Collection

- Best-performing collection algorithms are combinations of simpler algorithms.
- Simple and small basic algorithms have serious *space vs. time* tradeoffs.
  - For instance, the Cheney copying collector is very efficient, provides fast allocation and packs objects close to each other. However, it requires twice the heap space of a mark-and-sweep collector.
- Naïve mark-and-sweep collectors have serious memory fragmentation problems. Also, these algorithms use *recursion* which may be a problem in memory-constrained environments.
- Pause times may become excessive with large heaps unless generational collection or reference counting is used.

Performance Considerations

- According to studies, garbage collection overhead ranges from 2% to 20%.
  - 10% of total execution time is a good average estimate.
- The number can be larger for extremely dynamic programming languages.
  - For instance, studies in the 1970s/early 1980s indicated that Lisp programs spent 40% of execution time garbage collecting!
  - However, GC algorithms have improved a lot since then.
  - Smalltalk and Self create a huge number short-lived method instances that need to be garbage-collected.
- For Java, the GC overhead is generally much smaller (no more than 5% usually).

Implementing Garbage Collectors: Key Design Choices

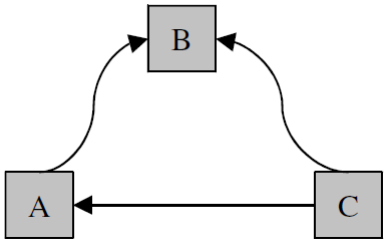
1) Exact vs. conservative
2) Handle-free vs. handle-based
3) Full vs. partial
4) Cooperative vs. concurrent
5) Single vs. multi-threaded

1) Exact vs. Conservative Garbage Collection

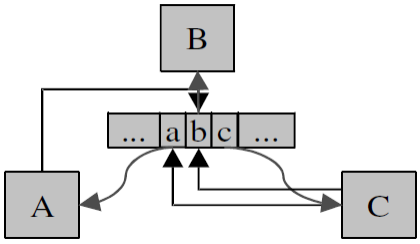
- Exact (precise) collection
  - Assumes that all pointers can be identified precisely
    - No accidental memory leaks
    - Enables object migration/copying
    - Enables full heap compaction
    - More complex algorithms to implement
- Conservative collection
  - Not all pointers are known; some "guesses" made
    - "If it looks like a pointer, assume it is a pointer"
    - Occasional memory leaks possible
    - Can cause heap fragmentation
    - Object migration unsafe
    - Simple to implement

2) Handle-free vs. Handle-based Garbage Collectors

- Handle-free collectors
  - Object references are direct
    - Higher performance
    - Harder to relocate objects



- Handle-based collectors
  - Object references are indirect
    - Slower performance
    - Separate handle area needed
    - Easy to relocate objects



3) Full vs. Partial Garbage Collection

- Full collectors
  - Perform full garbage collection each time GC is invoked
    - Pause times proportional to heap size
    - Often unacceptable for interactive applications and servers with strict response time requirements
    - Simple to implement; less performance overhead
- Partial collectors
  - Heap subdivided into areas or generations that are collected separately
    - Reduces pause times; pause times more evenly distributed
    - Requires a *write barrier*, increasing complexity and slowing down overall performance

4) Cooperative vs. Concurrent Garbage Collection

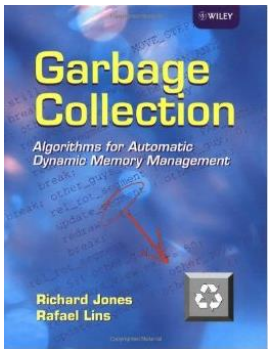
- Cooperative (stop-the-world) collection
  - Application threads temporarily stopped during GC
    - "Frozen" heap during GC simplifies collection
    - Introduces pauses that may be undesirable
    - Simple to implement; less overall performance overhead
- Concurrent collection
  - Application and the collector may run simultaneously
    - Heap contents may change during collection
    - Application threads might compete with the collector
    - Synchronization and load balancing difficult to implement

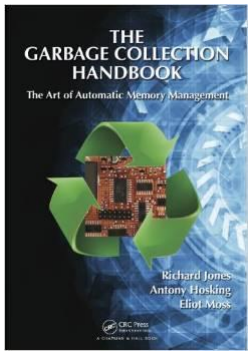
5) Single vs. Multi-Threaded Garbage Collectors

- Single-threaded collectors
  - Only one thread performs garbage collection
    - Simple to implement
    - Not scalable to multi-processor machines
- Multi-threaded collectors
  - Multiple threads performing garbage collection
    - Scales better on multi-processor machines
    - Difficult to implement because of synchronization issues, possible race conditions, etc.

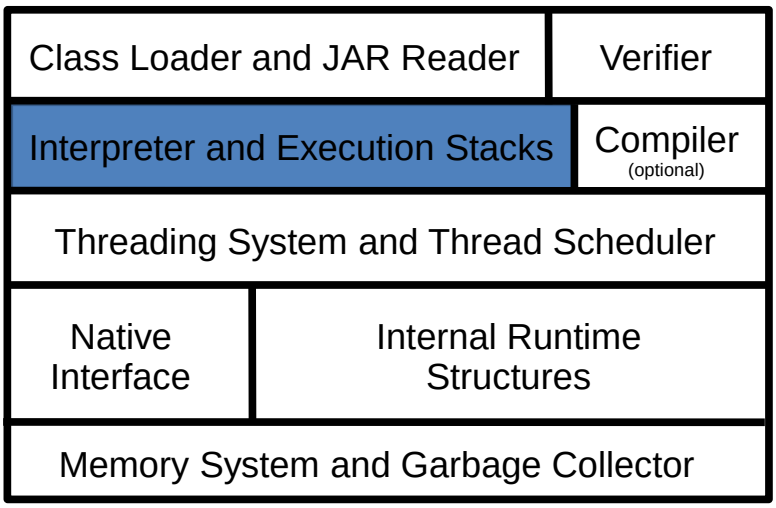
Further Reading on Memory Management and GC

- There are hundreds of garbage collection algorithms.
- For a great overview, read the "bible" of garbage collection (the original 1996 version & the 2012 update).
- http://www.gchandbook.org/





Walkthrough of Essential Component Areas



Interpretation and Execution

Background

- Executing source code directly can be very expensive and difficult.
  - Parsing of source code takes a lot of time and space.
  - In general, source code is intended to be human-readable; it is not intended for direct execution.
- Most virtual machines use some kind of an *intermediate representation* to store programs.
- Most virtual machines use an *interpreter* to execute code that is stored in the intermediate representation.

Two Kinds of Interpreters

Virtual machines commonly use two types of interpreters:

❶ Command-line interpreter ("outer" interpreter / parser)
  - Reads and parses instructions in source code form (textual representation).
  - Only needed in those systems that can read in source code at runtime.

❷ Instruction interpreter ("inner" interpreter)
  - Reads and executes instructions using an intermediate execution format such as bytecodes.

Parsers are covered well in traditional compiler classes; in this lecture we will focus on instruction interpretation

Basics of Inner Interpretation

- The heart of the virtual machine is the inner interpreter.
- The behavior of the inner interpreter:

1) Read the current instruction,
2) Increment the instruction pointer,
3) Parse and execute the instruction,
4) Go back to (1) to read the next instruction

A Minimal Inner Interpreter Written in C

```
int* ip; /* instruction pointer */
while (true) {
    ((void (*)())*ip++)();
}
```

Components of an Interpreter

- Interpreters usually have the following components:

Interpreter loop

```
while (true) {
    ((void (*)())*ip++)();
}
```

Instruction set

add, mul, sub, ...
load, store, branch, ...
...

Virtual registers

ip fp ...
sp lp

Execution stacks

Operand stack
Execution stack
...

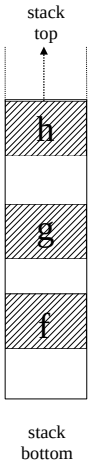
Virtual Registers

- *Virtual registers* hold the state of the interpreter during execution. Typical virtual registers:
  - *ip*: instruction pointer
    - Points to the current (or next) instruction to be executed.
  - *sp*: stack pointer
    - Points to the topmost item in the operand stack.
  - *fp*: frame pointer
    - Points to the topmost frame (activation record) in the execution stack (call stack).
  - *lp*: local variable pointer
    - Points to the beginning of the local variables in the execution stack.
  - *up*: current thread pointer (if multithreading is required)

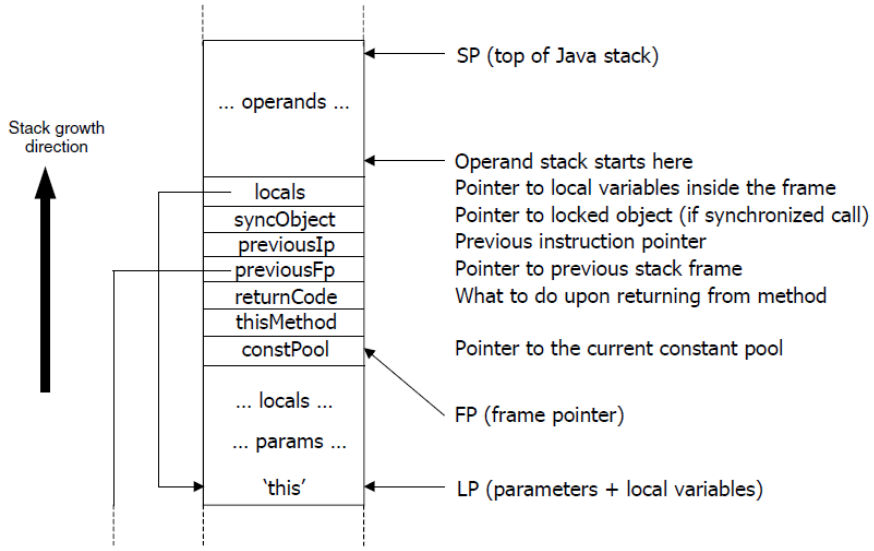
Execution Stacks

- In order to support method (subroutine) calls and proper control flow, an *execution stack* is typically needed.
  - Also known as the *call stack*.
- *Execution stack* holds the *stack frames* (activation records) at runtime.
  - Allows the interpreter to invoke methods/subroutines and to return to correct locations once a method call ends.
  - Each thread in the VM needs its own execution stack.
- Some VMs use a separate *operand stack* to store parameters and operands.

```
f() {
    g();
}

g() {
    h();
}
```



Concrete Example: Stack Frames in the KVM

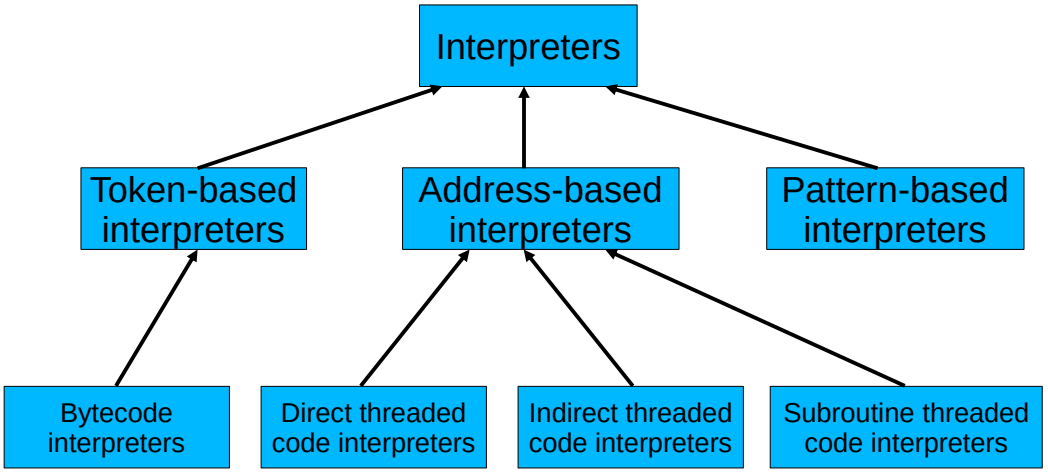


Fundamental Interpretation Techniques

Interpretation: Fundamental Techniques

- Token-based interpretation
  - P Code (UCSD Pascal)
  - Bytecode interpretation (Smalltalk, Java)
- Address-based interpretation
  - Indirect threaded code (Forth)
  - Direct threaded code (Forth)
  - Subroutine threading
- String/pattern-based interpretation (PostScript, Snobol, ...)

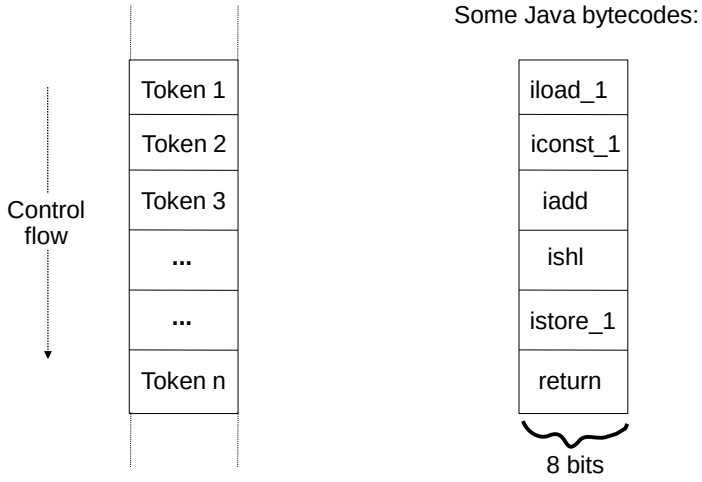
Taxonomy of Interpreters



Token-Based Interpretation

- In token-based interpreters, the fundamental instruction unit is a *token*.
  - Token is a predefined numeric value that represents a certain instruction.
    - E.g., 1 = LOAD LITERAL, 2 = ADD, 3 = MULTIPLY, …
  - Token values are independent of the underlying hardware or operating system.
- The most common subcase:
  - In a *bytecode interpreter*, instruction (token) width is limited to 8 bits.
  - Total instruction set limited to 256 instructions.
  - Bytecode interpreters are very commonly used, e.g., for Smalltalk, Java, and many other interpreted programming languages.

Token-Based Code: Examples



Code is represented as linear lists that contain fixed-size tokens.
In bytecode, token width is 8 bits.

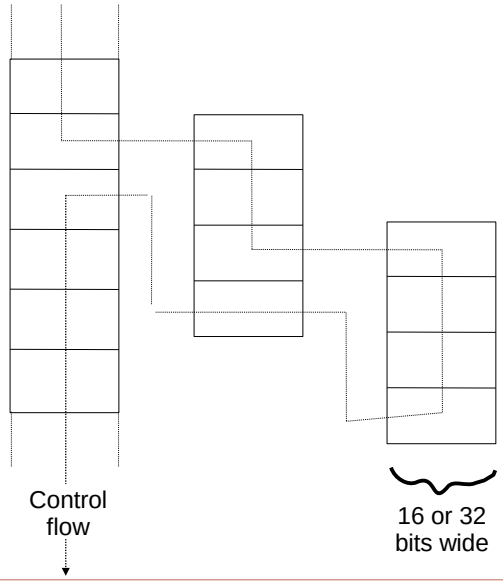
A Simple Bytecode Interpreter Written in C

```
void Interpreter() {
   while (true) {
      byte token = (byte)*ip++;

      switch (token) {
               case INSTRUCTION_1:
                     break;
               case INSTRUCTION_2:
                     break;
               case INSTRUCTION_3:
                     break;
      }
   }
}
```

Address-based Interpreters

- In address-based interpreters, the fundamental instruction unit is an *address* rather than a token.
- Address-based interpreters can be more efficient than token-based, especially when written in machine code.
  - No token decoding overhead.
  - Can jump directly from one instruction to the next.
- The term *threaded code* refers to a code representation where every instruction is implicitly a function call to another address.
  - Bell, J.K., Threaded code, Communications of the ACM vol 16, nr 6 (Jun) 1973, pp.370-372.

Threaded Code: The Basic Idea



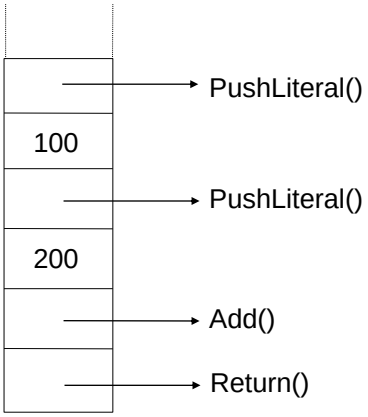
In threaded code, code is stored as a linear list of addresses.

Every instruction (word) contains an address that represents a function to execute.

Some words are *primitives*; primitives are executed directly by calling the address stored in the word itself.

Other words are *non-primitives*; when executing a non-primitive, control flow is transferred to another list of addresses.

Threaded Code: Example



Explanation:

Read the literal value 100 from the instruction stream and push it onto the operand stack.

Read the literal value 200 from the instruction stream and push it onto the operand stack.

Add two topmost items in the operand stack.

Return control back to caller, leaving the result on the operand stack.

Equivalent Forth code:
: foo 100 200 + ;

A Simple Threaded Code Interpreter Written in C

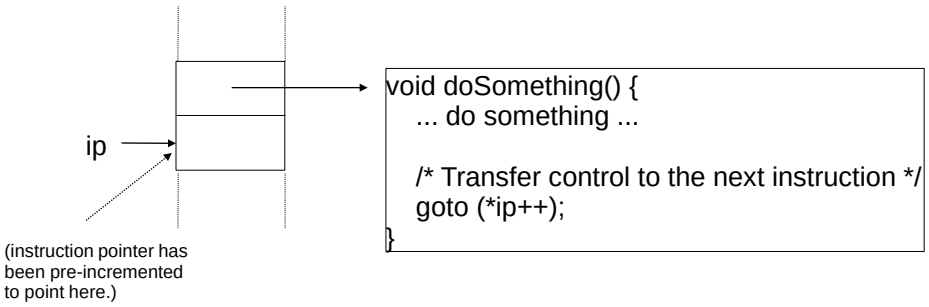
```
void Interpreter() {
   while (true) {
      word* address = (word*)*ip++;

      if (isPrimitiveInstruction(address)) {
         /* Run primitive */
         ((void (*)())*address)();
      }
      else {
         /* Execute subroutine */
         pushReturn(ip);
         ip = address;
      }
   }
}
```

Direct vs. Indirect Threaded Code

- There are three commonly used forms of threaded code:
  - Direct threaded code
    - Bell, J.K., Threaded code. Communications of the ACM vol 16, nr 6 (Jun) 1973, pp.370-372.
  - Indirect threaded code
    - Dewar, R.B.K., Indirect threaded code, Communications of the ACM vol 18, nr 6 (Jun) 1975, pp.330-331.
  - Subroutine threaded code
- Direct threaded code is most efficient, but it cannot be implemented in a portable fashion using C or other high-level languages.

Worth Mentioning… Direct Threaded Code

- In direct threaded code, there is no centralized interpreter loop at all.
- When the execution of a primitive ends, control is transferred directly to the next instruction by performing a "*computed goto*" instruction.



- Unfortunately, C/C++ language does not have a standardized computed goto instruction, so there is no way to implement this in a 100% portable fashion.

Instruction Sets

Instruction Sets

- Each virtual machine typically has its own instruction set based on the requirements of the language(s) the VM must support.
- These instruction sets are similar to instruction sets of hardware CPUs.
- Common types of instructions:
  - Local variable load and store operations
  - Constant value load operations
  - Array load and store operations
  - Arithmetic operations (add, sub, mul, div, ...)
  - Logical operations (and, or, xor, ...)
  - Type conversions
  - Conditional and unconditional branches
  - Method invocations and returns
  - ...

Stack-Oriented vs. Register-Oriented Instruction Sets

- Two types of instruction sets:
  1. In *stack-oriented* instruction sets, operands to most instructions are passed in an operand stack; this stack can grow and shrink dynamically as needed.
  2. In *register-oriented* instruction sets, operands are accessed via "register windows": fixed-size areas that are allocated automatically upon method calls.
- Historically, most virtual machines used a stack-oriented instruction set.
  - Stack machines are generally simpler to implement.
  - No problems with "running out of registers"; the instruction set can be smaller.
  - Less encoding/decoding needed to parse register numbers.
- Unlike JVM, Dalvik uses a register-based instruction set.

Criteria	Dalvik	JVM
Architecture	Register-based	Stack-based
OS-Support	Android	All
Executables	DEX	JAR
Constant-pool	per application	per Class

Example: The Java Bytecode Interpreter

- The JVM uses a straightforward stack-oriented bytecode instruction set with approximately 200 instructions.
  - Fairly similar to the Smalltalk bytecode set, except that in Java primitive data types are not objects.
- One execution stack is required per each Java thread.
  - No separate operand stack; operands are kept on top of the current stack frame.
- Four virtual registers are commonly assumed:
  - *ip* (instruction pointer): points to current instruction
  - *sp* (stack pointer): points to the top of the stack
  - *fp* (frame pointer): provides fast access to stack frame
  - *lp* (locals pointer): provides fast access to local variables

Example: The Java Virtual Machine Instruction Set

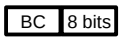
aaload	daload	f2l	getstatic	if_icmplt	invokevirtual_quick	ldc_w	new
aastore	dastore	fadd	getstatic	if_icmpne	invokevirtual_quick_w	ldc_w_quick	new_quick
aconst_null	dcmpg	faload	getstatic_quick	ifeq	invokevirtualobject_quick	ldc2_w	newarray
aload	dcmpl	fastore	getstatic2_quick	ifge	ior	ldc2_w_quick	nop
aload_0	dconst_0	fcmpg	goto	ifgt	irem	ldiv	pop
aload_1	dconst_1	fcmpl	goto_w	ifle	ireturn	lload	pop2
aload_2	ddiv	fconst_0	i2b	iflt	ishl	lload_0	putfield
aload_3	dload	fconst_1	i2c	ifne	ishr	lload_1	putfield
anewarray	dload_0	fconst_2	i2d	ifnonnull	istore	lload_2	putfield_quick
anewarray_quick	dload_1	fdiv	i2f	ifnull	istore_0	lload_3	putfield_quick_w
areturn	dload_2	fload	i2l	iinc	istore_1	lmul	putfield2_quick
arraylength	dload_3	fload_0	i2s	iload	istore_2	lneg	putstatic
astore	dmul	fload_1	iadd	iload_0	istore_3	lookupswitch	putstatic
astore_0	dneg	fload_2	iaload	iload_1	isub	lor	putstatic_quick
astore_1	drem	fload_3	iand	iload_2	iushr	lrem	putstatic2_quick
astore_2	dreturn	fmul	iastore	iload_3	ixor	lreturn	ret
astore_3	dstore	fneg	iconst_0	impdep1	jsr	lshl	return
athrow	dstore_0	frem	iconst_1	impdep2	jsr_w	lshr	saload
baload	dstore_1	freturn	iconst_2	imul	l2d	lstore	sastore
bastore	dstore_2	fstore	iconst_3	ineg	l2f	lstore_0	sipush
bipush	dstore_3	fstore_0	iconst_4	instanceof	l2i	lstore_1	swap
breakpoint	dsub	fstore_1	iconst_5	instanceof_quick	ladd	lstore_2	tableswitch
caload	dup	fstore_2	iconst_m1	invokeinterface	laload	lstore_3	wide
castore	dup_x1	fstore_3	idiv	invokeinterface_quick	land	lsub	xxxunusedxxx
checkcast	dup_x2	fsub	if_acmpeq	invokenonvirtual_quick	lastore	lushr	
checkcast_quick	dup2	getfield	if_acmpne	invokespecial	lcmp	lxor	
d2f	dup2_x1	getfield	if_cmpge	invokestatic	lconst_0	monitorenter	
d2i	dup2_x2	getfield_quick	if_icmpeq	invokestatic_quick	lconst_1	monitorexit	
d2l	f2d	getfield_quick_w	if_icmpgt	invokesuper_quick	ldc	multianewarray	
Dadd	f2i	getfield2_quick	if_icmple	invokevirtual	ldc_quick	multianewarray_quick	

(Note: "_quick" bytecodes are non-standard and implementation-dependent)

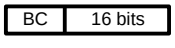
JVM Instruction Formats

1. Most Java bytecodes do not require any parameters from the instruction stream.
   - They operate on the values provided on the execution stack (e.g., IADD, IMUL, ...)

2. Some bytecodes read an additional 8-bit parameter from the instruction stream.
   - For instance, NEWARRAY, LDC, *LOAD, *STORE

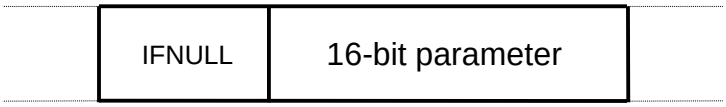


3. Many bytecodes read additional 16 bits from the instruction stream.
   - INVOKE* instructions, GET/PUTFIELD, GET/PUTSTATIC, branch instructions, ...



4. Three instructions are varying-length.
   - LOOKUPSWITCH, TABLESWITCH, WIDE

Example: IFNULL

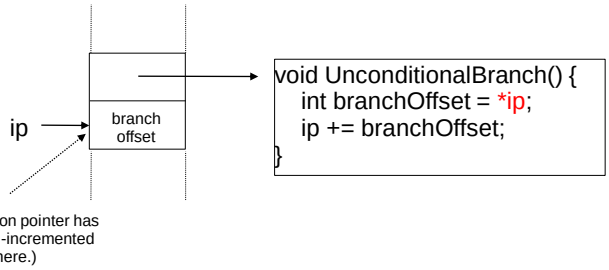


Operand stack: ..., *value* => ...

- IFNULL: Branch if reference is null.
- The instruction pops value off the operand stack, and checks if the value is NULL.
- The 16-bit parameter contains a branch offset that is added to the instruction pointer if value is NULL.
- Otherwise, execution continues normally from the next instruction.

Accessing Inline Parameters

- Inline parameters are generally very easy to access.
- Use the instruction pointer to determine the location.
- For instance, a bytecode for performing an unconditional jump could be written as follows:



- Important: Keep in mind the endianness issues!
  - In Java, all numbers in classfiles are *big-endian*; if a machine-specific endianness was used, Java class files wouldn't be portable across different machines.

A Few Notes on Stack Frames and Exception Handling

Implementing Stack Frames (Activation Records)

- Each time you do a method call (e.g., execute an INVOKE* bytecode in Java), a new *stack frame* needs to be created.
  - The stack frame contains the method call arguments, has space for the local variables, and contains some administrative data as well.
- When the method returns, the stack frame is popped (removed) from the execution stack.
  - The values of the virtual registers (ip, sp, fp, ...) are restored so that the earlier method can continue to run.
- Each Java thread has its own execution stack.
- There is *no separate operand stack* in a JVM.
  - Operands of bytecodes are kept on top of the current stack frame.

Activation Records in Other VMs

- Not all programming languages use or need stack-allocated activation records.
- In Smalltalk, method block instances are allocated dynamically from the heap.
  - Every method invocation creates garbage!
  - The actual VM implementation may optimize the semantics and use stack allocation instead.
- In Forth, there are no activation records.
  - The only thing that needs to be pushed onto the execution stack (called the '*return stack*' in Forth) is the current instruction pointer.
  - Parameters and return values are stored explicitly in a separate operand stack.

Advanced Topics: Adding Exception Handling Support

- The Java programming language supports *exception handling*:

```
try {
    ... do something ...
} catch (IOException e1) {
    ... handle I/O failure ...
} catch (MyException e2) {
    ... handle app-specific failure ...
}
```

- When an exception occurs, control flow must return to the exception handler no matter how many levels of nested subroutine/method calls have been made inside the **`try`** block!!

Implementing Exception Handling

- When an exception occurs:
  1. The virtual machine scans the execution stack for matching *exception handlers*.
     - Scanning is performed by comparing the exception handler type and the bytecode range in which the handler is valid.
     - Scanning is started from the topmost stack frame, and continues to the bottommost frame if necessary.
  2. If a matching handler is found in a stack frame, the VM has to unwind (pop and remove) all those stack frames that are above the matching one; control is then transferred to the execution handler code.
- Possible monitors/locks associated with the removed stack frames will have to be released!!
  - *This can get very tricky if there are pending I/O operations associated with the removed stack frames!!*

Remarks on Interpreter Performance

Interpretation Overhead

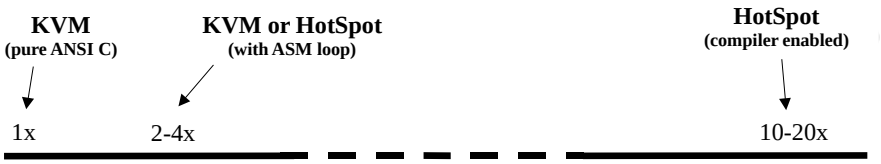
- Interpreted code is generally a lot slower than compiled code/machine code.
  - Studies indicate an order of magnitude difference.
  - Actual range is something like 2.5x to 50x.
- Why? Because there are extra costs associated with interpretation:
  - Dispatch (fetch, decode and invoke) next instruction
  - Access virtual registers and arguments
  - Perform primitive functions outside the interpreter loop
- *"Interpreter performance is primarily a function of the interpreter itself and is relatively independent of the application being interpreted."*
- In principle, direct threaded code has much lower overhead.

Interpreter Tuning

- Common interpreter optimizations techniques:
  - Writing the interpreter loop and key instructions in assembly code.
  - Keeping the virtual registers (ip, sp, ...) in physical hardware registers – this can improve performance dramatically.
  - Splitting commonly used instructions into a separate interpreter loop & making the core interpreter so small that it fits in HW cache.
  - Top of stack caching (keeping topmost operand in a register).
  - Padding the instruction lookup table so that it has exactly 16/32/64/128/256 entries.
- Actual impact of such optimizations will vary considerably based on underlying hardware.

High-Performance VMs

- Small, simple & portable VM == slow VM
- Interpreted code has a big performance overhead compared to native code:



- If you need speed, you need a compiler!
  - Unfortunately, compilers are always rather machine/CPU-specific.
  - This introduces a lot of additional complexity & requires a lot more manpower to implement.
- Almost always: Fast == more complex

Compilation: Basic Strategies

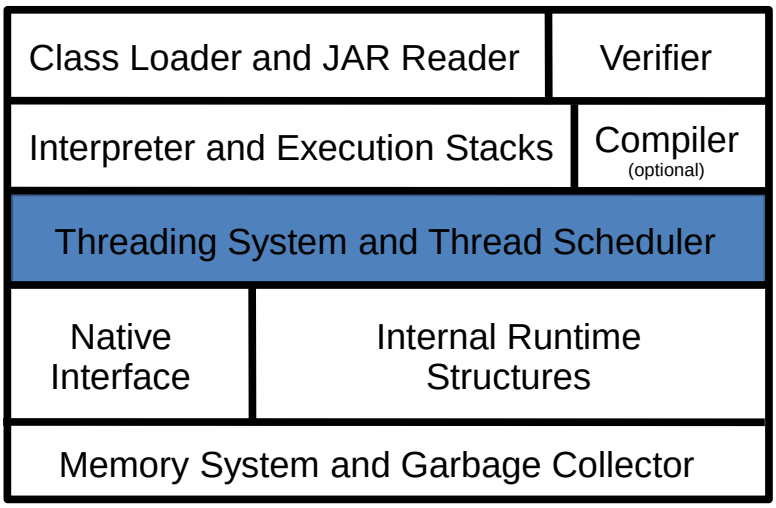
❶ Static / Ahead-Of-Time (AOT) Compilation
  - Compile code before the execution begins.

❷ Dynamic (Just-In-Time, JIT) Compilation
  - Compile code on the fly when the VM is running.

- Different flavors of dynamic compilation:
  - Compile everything upon startup (impractical)
  - Compile each method when executed first time
  - Adaptive compilation based on “hotspots” (frequently executed code)

These topics would deserve a separate lecture or two...

Walkthrough of Essential Component Areas



Adding Multithreading Support

Multithreading and Synchronization

- A key feature of many programming languages is *multithreading.*
  - *Multithreading*: the ability to create multiple concurrently running threads/programs.
  - Smalltalk, Forth, Ada, Self, Java, ...
- Each thread behaves as if it owns the entire virtual machine
  - ... except when the thread needs to access external resources such as storage, network, display, or perform I/O operations in general.
  - ...or when the thread needs to communicate with the other threads.
- *Synchronization/locking* mechanisms are needed to ensure controlled communication and controlled access to external resources.

Multithreading Support in Java: Code Example

```
public class MyClass implements Runnable {
  public void run() {
    ... do something ...
  }

  public static void main(String[] args) {
      for (int i = 0; i < 10; i++) {
          Runnable r = new MyClass();
          Thread t = new Thread(r);
          t.start();
      }
  }
}
```

Implementing Multithreading: Technical Challenges

- Each thread must have its own virtual registers and execution stacks.
- Critical places of the VM (and libraries) must use mutual exclusion to avoid deadlocks and resource conflicts.
- Access to external resources (e.g., storage, network) must be controlled so that two threads do not interfere with each other.
- I/O operations must be designed so that one thread's I/O operations do not block the I/O of other threads.
- Generally: All native function calls must be non-blocking.
- Locking / synchronization operations must be provided also at the application level.

Recap: Components of an Interpreter

- Interpreters usually have the following components:

Interpreter loop

```
while (true) {
    ((void (*)())*ip++)();
}
```

Instruction set

add, mul, sub, ...
load, store, branch, ...
...

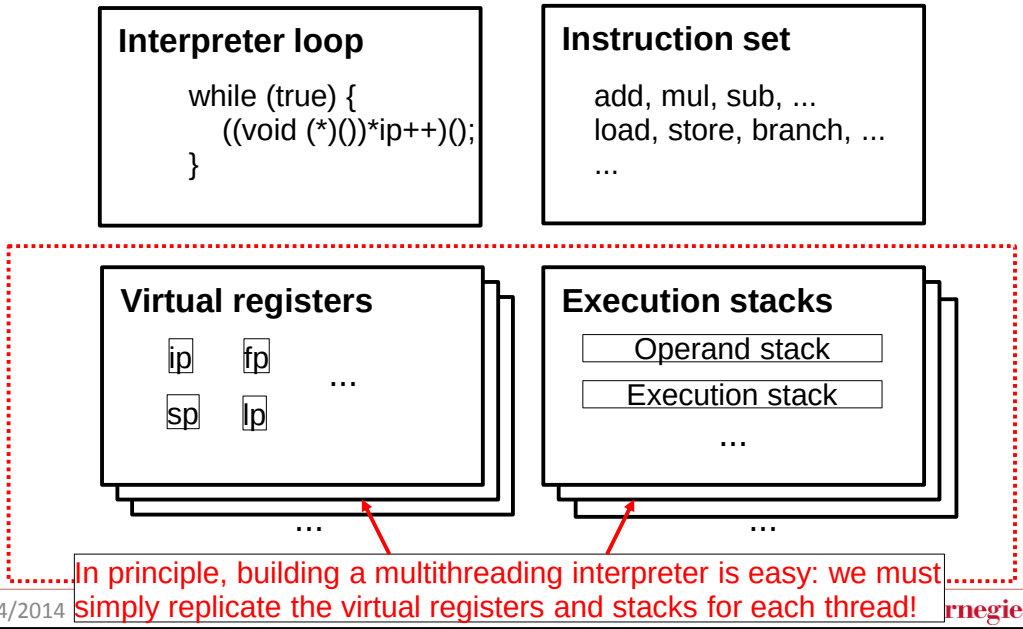
Virtual registers

ip fp ...
sp lp

Execution stacks

Operand stack
Execution stack
...

Building a Multithreading VM



Building a Multithreading VM: Supporting Interrupts

- In addition, we must modify the system so that it allows the current thread to be *interrupted*.
- When a thread is interrupted, a *context switch* is performed.

Example:

```
int* ip; /* instruction pointer */
while (true) {
    if (isInterrupted()) ContextSwitch();
    ((void (*)())*ip++)();
}
```

Building a Multithreading VM: Context Switching

- What happens during a context switch?
  1. The virtual registers of the current thread are stored (context save).
  2. Current thread pointer is changed to point to the new current thread.
  3. Virtual registers are replaced with the saved virtual registers of the new current thread (context load).
- Context switching must be performed as an uninterrupted operation!
  - No further interrupts may be processed until the context switch operation has been performed to completion.
  - Operating systems commonly have a "supervisor mode" for running system-critical code.

Avoiding Atomicity Problems Using Safepoints

- In operating systems, threads can usually be interrupted at arbitrary locations.
  - Interrupts may be generated by hardware at any time.
  - The entire operating system must be designed to take into account mutual exclusion problems!
  - Must use monitors or semaphores to protect code that can be executed only by one thread at the time.
- In VMs, simpler solutions are often used.
  - Threads can only be interrupted in certain locations inside the VM source code.
  - These locations are known as "*safepoints*".
  - In the simplest case, thread switching is only allowed in one place inside the VM.
  - Makes VM design a lot simpler and more portable!

Using the "One Safepoint" Solution

- No separate interrupt handler routine.
- All the checking for interrupts happens inside the interpreter loop.
- Even if the actual interrupts are generated asynchronously, the actual context switching is not performed until the interpreter loop gets a chance to detect and process the interrupt:

```
int* ip; /* instruction pointer */
while (true) {
    if (isInterrupted()) ContextSwitch();
    ((void (*)())*ip++)();
}
```


Making Thread Switching 100% Portable

- In a virtual machine, you don't necessarily need an external clock to drive the interrupts!
- In the simplest case, you can count the number of executed instructions using a "timeslice":

```
int* ip; /* instruction pointer */
while (true) {
    if (--TimeSlice <= 0) ContextSwitch();
    ((void (*)())*ip++)();
}
```

  - Force a context switch every 1000 bytecodes or so.
- You can also enforce a thread switch at each I/O request (used in many Forth systems).

Thread Scheduling

- When you perform a context switch, which thread should run next?
- Various choices:
  - FCFS (First-Come-First-Serve)
  - Round robin approach
  - Priority-based scheduling
    - … with fixed priorities or varying priorities
- In operating systems, thread scheduling algorithms can be rather complicated.
- There is no "ideal" scheduling algorithm.
- The needs of interactive and non-interactive programs (and client vs. server software) can be fundamentally different in this area.

Case Study: KVM

- The original KVM implementation used a simple, portable *round robin* scheduler.
  - Threads stored in a circular list; each thread got to execute a fixed number of bytecodes until interrupt was forced.
  - Thread priority only affected the number of bytecodes a thread may run before it gets interrupted.
- In the actual product version, thread scheduling based on Java thread priority.
  - Higher-priority threads always run first.
- Fully portable thread implementation; interrupts driven by bytecode counting; thread switching handled inside the interpreter loop.

Actual Code: Thread Switching in KVM

Code inside the interpreter loop:

```
if (--Timeslice <= 0) {
    do {
        ulong64 wakeupTime;

        /* Check if it is time to exit the VM */
        if (AliveThreadCount == 0) return;

        /* Check if it is time to wake up */
        /* threads waiting in the timer queue */
        checkTimerQueue(&wakeupTime);

        /* Handle external events */
        InterpreterHandleEvent(wakeupTime);

    } while (!SwitchThread());
}
```

Thread switching code (simplified):

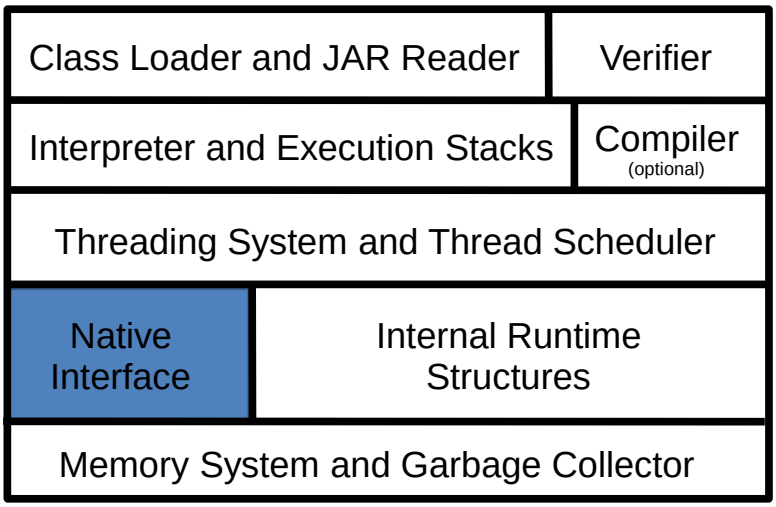
```
bool_t SwitchThread() {
    /* Store current context */
    StoreVirtualRegisters();

    /* Obtain next thread to run */
    CurrentThread =
        removeQueueStart(&RunnableThreads);
    if (CurrentThread == NULL) return FALSE;

    /* Set new context and timeslice */
    LoadVirtualRegisters(CurrentThread);
    Timeslice = CurrentThread->timeslice;

    return TRUE;
}
```

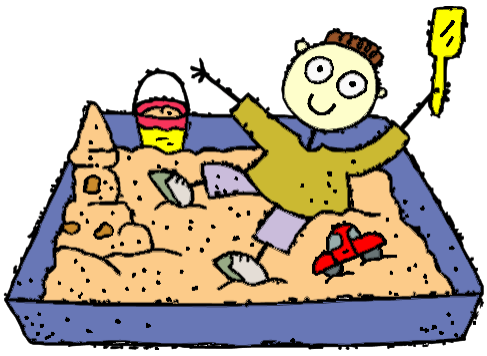

Native vs. Non-Native Multithreading

Different Multithreading Techniques

- Multithreading can be implemented in two basic ways:
  - *Non-native ("green") threads*: Multithreading is implemented internally by the VM; the threading system is independent of the multithreading capabilities provided by the underlying operating system.
  - *Native threads*: Multithreading is implementing using the capabilities provided by the underlying operating system (by mapping VM threads to the native threading system).
- The efficiency, portability and complexity of the virtual machine varies dramatically depending on the chosen approach!
- We will not dive into details in this lecture.

Walkthrough of Essential Component Areas





Native Interface

Escaping the
Virtual Machine Sandbox
to Access External Resources

Native Function Interface

- A virtual machine frequently needs to call various native functions to perform I/O, access external resources, etc.
  - Graphics, networking, file storage, etc. etc.
- Application developers usually access native functionality in a portable fashion using classes and functions that are specific to the programming languages (e.g., Java libraries).
- A *native function interface* is provided to map the language-specific function calls to underlying OS functionality.
- Among other things, the native interface will handle the parameter passing/conversion between the VM and the underlying OS.
- As usual, there are significant performance/portability tradeoffs in this area.

Native Function Interface – Technical Challenges

- Parameter passing conversions can add significant performance overhead.
  - E.g., mapping the VM stack frames to native calling conventions
- In a VM that uses green (non-native) threads, the rest of the VM is "blocked" when a native function is called.
  - There is only one physical thread of control.
  - If the native function does not return immediately, all the other threads in the VM are stuck indefinitely until the native function returns.
  - In the worst case, the entire system may deadlock if the current thread is waiting for something from another thread (e.g., in a blocking I/O operation).

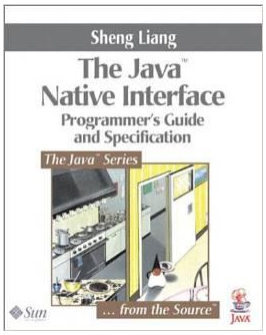
Possible Solutions

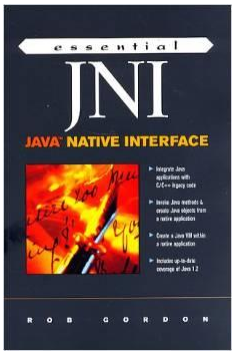
1. All native functions must be designed so that they return immediately.
   - Might have to wrap some I/O calls with code that uses polling / busy waits to check if data is available.
   - Difficult to enforce on third-party functions.
2. A pool of native threads might be used at the implementation level to perform I/O calls asynchronously.
   - Allows the VM to continue execution until data is present.

- (2) is a better choice but not fully portable.
  - KVM provided both choices (as a compile-time option).
- VMs that use native threads don't have these problems (but have many other thread-related problems instead; basically, interrupts can occur at any place/time).

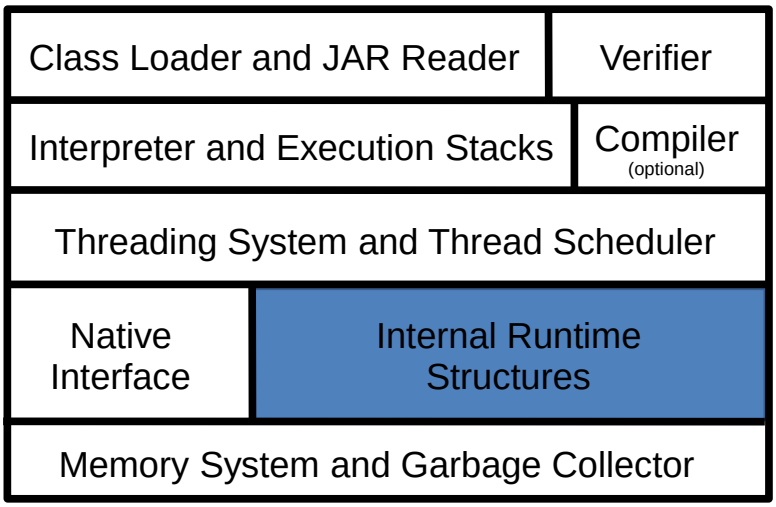
Further Reading in This Area

- In the Java area, Sheng Liang's handbook is still the *de facto* reference.





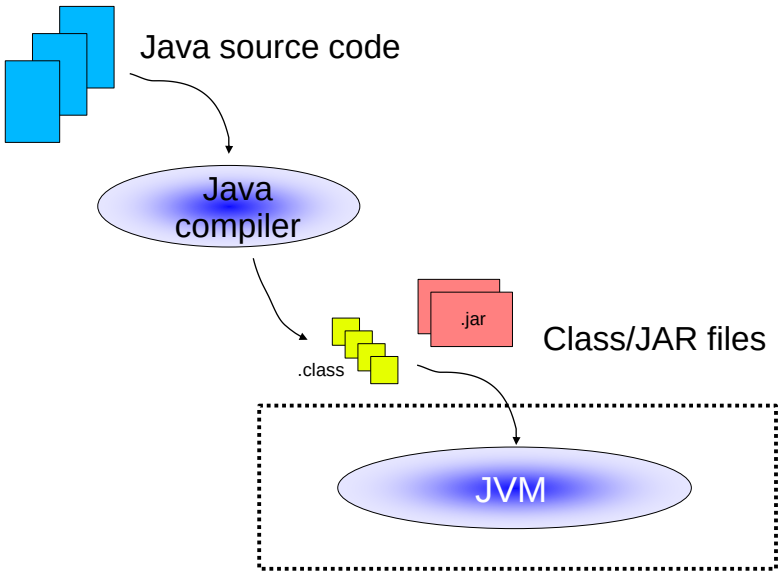
Walkthrough of Essential Component Areas



Internal Runtime Structures

- In addition to the topics discussed above, a virtual machine must implement various internal data structures for supporting the runtime behavior of the supported language.
  - E.g., classes, class hierarchies, field and method lookup tables ("vtables"), access flags, object instances
- These structures tend to be language-specific; furthermore, different virtual machines may end up creating these structures entirely differently.
- However, there is commonality across families of languages. For instance, all object-oriented programming languages require efficient lookup tables for instance method calls.
- The following material is applicable to Java (and indirectly Android).

Java Application Model



Java Class File Format

- Class file: binary representation of a Java class.
- Consists of a stream of 8-bit bytes.
- Data types inside the file:
  - Basic types:
    - `u1`, `u2`, `u4`: unsigned 8, 16 and 32-bit numbers.
  - Higher-level types are composed of the basic types.
  - In addition, there are variable-sized *tables*: arrays of data types that are preceded by the number of elements in the table.
- Big-endian representation used for portability.
  - All 16-bit, 32-bit and 64-bit quantities inside the class file are represented in *big-endian* format.
  - Big-endian: high byte comes first in memory.

Java Class File Format Illustrated

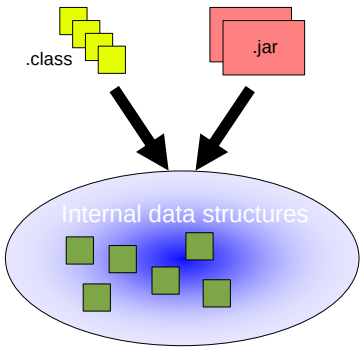
Magic value (0xCAFEBABE) and version info
Constant pool
Class/superclass info and class access flags
Interfaces
Fields and their attributes
Methods and their attributes
Extra attributes

Java Class File Format

```
ClassFile {
    u4             magic;           /* Always 0xCAFEBABE */
    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];
}
```


Creating Internal Data Structures

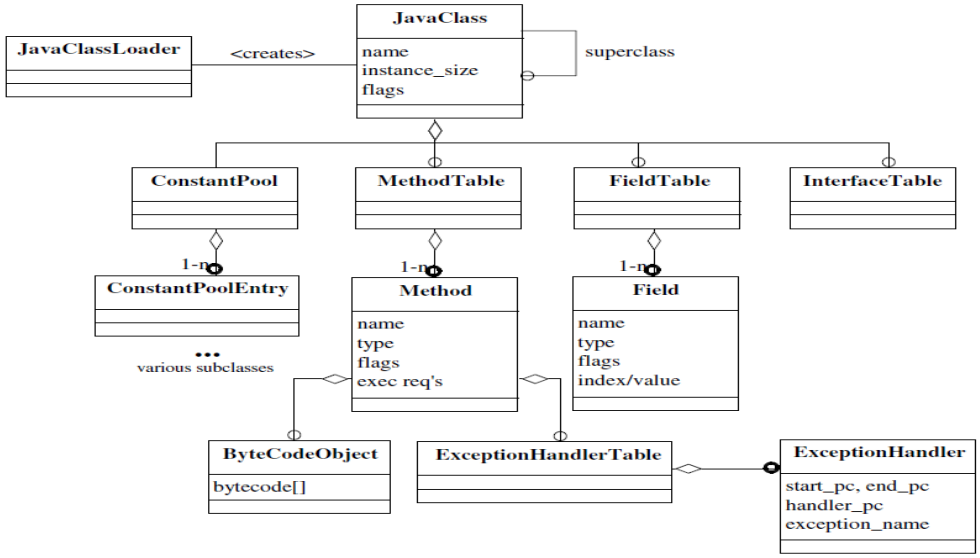
- The JVM creates internal data structures for all the information loaded from the class files.
- The internal data structures are different from the structures used in the class file.
  - For instance, symbolic references are usually converted to pointers inside the VM wherever possible.
- The internal data structures may be totally different in different JVMs.
- In some JVMs all the structures are in the heap while other JVMs may use special memory areas for some structures.



Internal Representation of Classes

- Each Java class needs to store the following information at runtime:
  - Name of the class
  - Access flags (public, abstract, final, ...)
  - Superclass pointer
  - Constant pool (symbols and constants of the class)
  - Field table, method table and interface table
  - Instance size
  - Class initialization status
  - The static variables of the class
- Classes and methods are typically kept in a hashtable to speed up class lookups and vtable lookups inside the VM.

Typical Runtime Data Structures For Java Classes

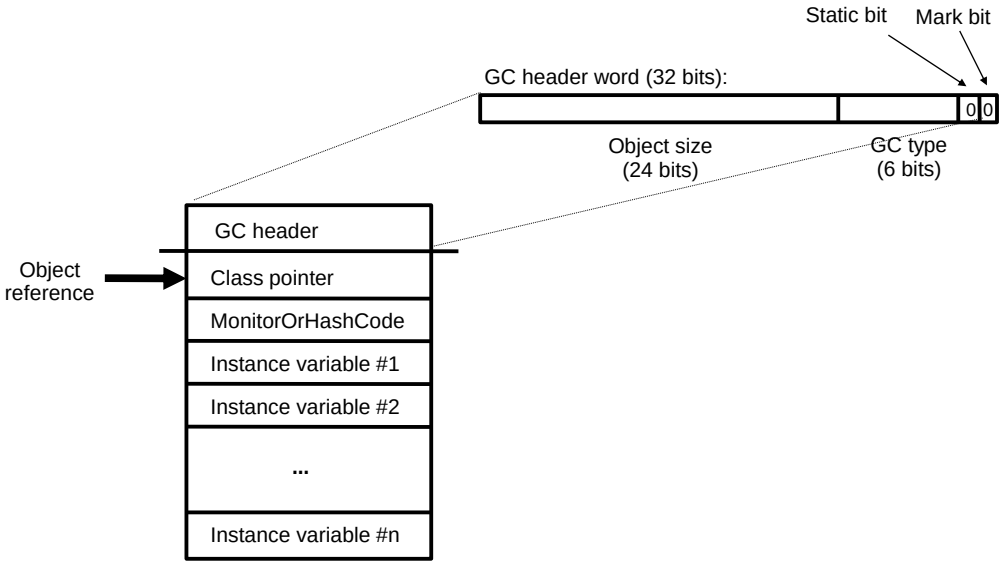


Object Layout

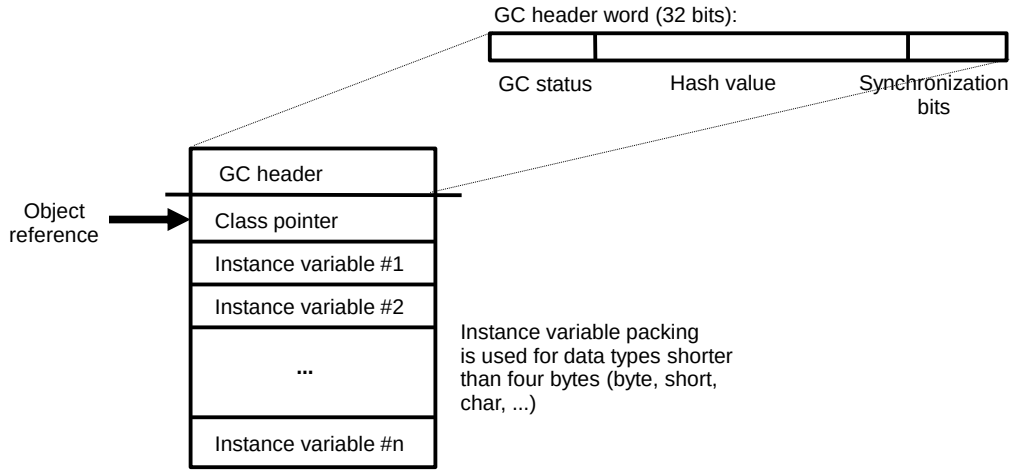
Object Layout in Java

- All Java object instances need to be able to potentially store the following data:
  - Instance variables
    - One slot for each instance variable, two slots for variables of type `long` and `double`.
  - Class pointer
    - `java.lang.Object.getClass()`
  - Monitor reference
    - `synchronized (x) { ... }`
  - Object identity (hashcode)
    - `java.lang.Object.hashCode()`
- Not all this information is needed for every object.
  - Various optimizations are possible

Example: Object Layout in the K Virtual Machine



Object Layout in the HotSpot Virtual Machine



Instance variable packing is used for data types shorter than four bytes (byte, short, char, ...)

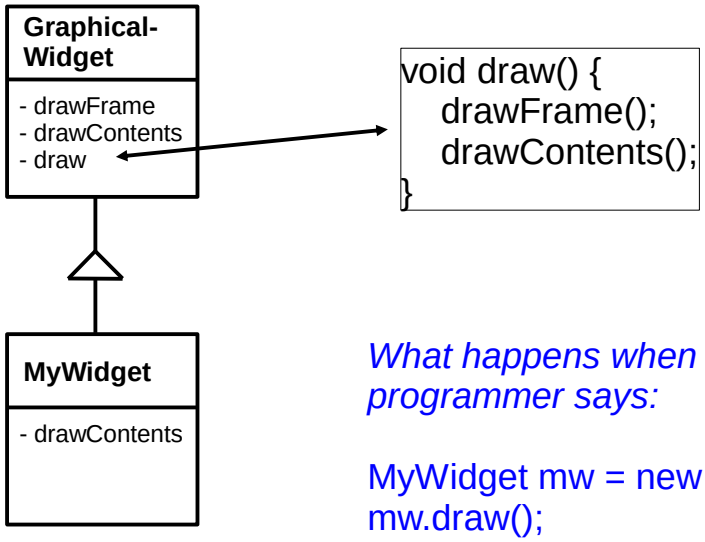
Implementing Method and Field Lookups Efficiently

- The method and field information in Java classfiles is not sufficient for implementing fast method and field lookups.
  - Methods are stored in a simple linear list; no sorting by name; no separation of static vs. instance methods.
- Quickening helps a lot, since each lookup has to be done only once.
  - Field lookups and static method lookups are "cheap"; after quickening is done, no lookup is needed at all.
- However, instance ("virtual") methods lookups are still a problem.
  - In virtual method lookups, the actual method to be invoked varies dynamically depending on the type of the object instance.

Static vs. Instance Methods in Object-Oriented Languages

- In object-oriented languages, static and instance method lookups behave differently.
- In static (class) method calls, the receiver of the message is the class.
  - A simple linear lookup of the class hierarchy is sufficient.
  - E.g., Object.hashCode(), MyClass.hashCode()
- In instance (virtual) method calls, the receiver of the message is an object instance.
  - The instance becomes the current '`this`' pointer (also known as the "self-pointer" or "self-reference").
  - The `this` pointer needs to be available in subsequent method calls so that inherited methods can call the methods of the subclasses when the object invokes its own instance methods.

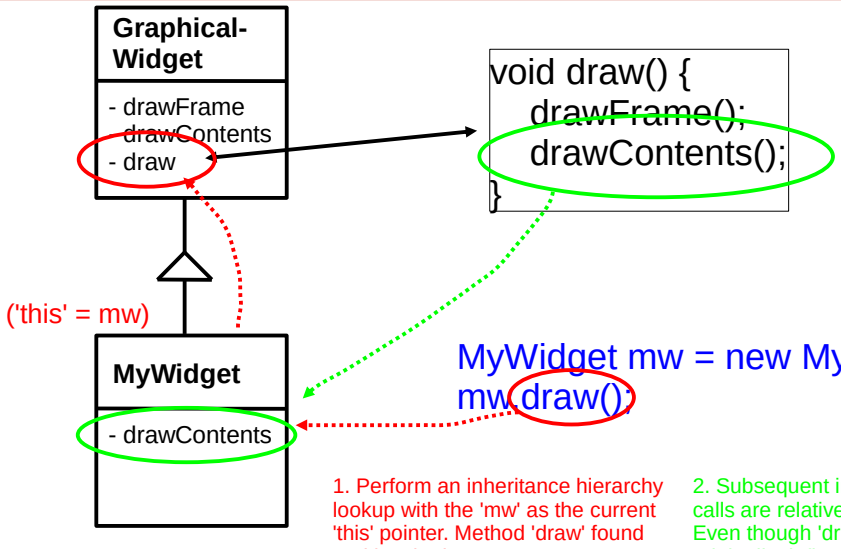
Instance Method Calls: Example



What happens when the programmer says:

MyWidget mw = new MyWidget();
mw.draw();

Instance Methods Calls: Example



1. Perform an inheritance hierarchy lookup with the 'mw' as the current 'this' pointer. Method 'draw' found and invoked.

2. Subsequent instance method calls are relative to 'this' pointer. Even though 'drawContents' was originally defined in the superclass, the subclass method gets invoked since it overrides the inherited method.

Implementing Instance Method Lookups

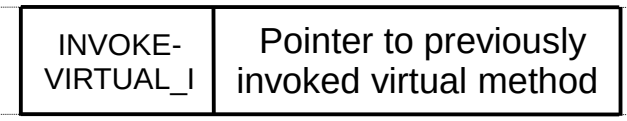
- In instance method lookups, methods are searched from the inheritance hierarchy starting from the (class of the) current '**`this`**' pointer.
  - Lookup algorithm needs to be fast, since dynamic lookups are needed very frequently.
- There are well-known techniques to implement/optimize method lookups:
  - *VTables*: Each class is associated with a table that contains the virtual methods of that class; the virtual methods of the superclasses are often copied down to each vtable to "flatten out" the inheritance hierarchy.
  - *Inline caching*: Cache one or more previous lookup results in the bytecode stream.

Example: Inline Caching

Invented by Deutsch & Schiffman for Smalltalk-80
Described in a POPL Conference article, 1984

Original bytecode:

Rewritten bytecode:



A dynamic lookup is needed only if the class of the cached method receiver is not the same as the class of 'this' pointer!

Studies show that cached virtual method is usually correct about 85% of the time.

Variations:

Monomorphic inline caching:
store only one (last) method target

Polymorphic inline caching:
store multiple method targets
(separate cache area needed)

More Information on Virtual Machine Design

- Bill Blunden, *Virtual Machine Design and Implementation in C/C++*, Wordware Publishing, March 2002
- Iain D. Craig, *Virtual Machines*, Springer Verlag, September 2005
- Jim Smith, Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, June 2005
- Xiao-Feng Li, Jiu-Tao Nie, Ligang Wang, *Advanced Virtual Machine Design and Implementation*, CRC Press, October 2014
- Matthew Portnoy, *Virtualization Essentials*, Sybex Press, May 2012

Not that good; very narrow scope

Better

Worth buying!

Brand new – Haven't read yet

Good introduction to OS virtualization!

Thank You!

Further Reading

A Brief History of Programming Languages that Utilize a Virtual Machine

Some Background

- Virtual machines have been studied and built since the late 1950s.
- Many early programming languages were built around the idea of a portable runtime system.
- Yet VM design was always a fairly specialized topic; not many books or articles were written until recently.
- Popularity of the area exploded in the mid-1990s when the Java programming language was introduced.
- OS virtualization became extremely popular in the mid-2000's.

LISP

- John McCarthy, 1958
  - http://www-formal.stanford.edu/jmc/history/lisp/lisp.html
  - LISP is the second oldest programming language still in widespread use (after Fortran)
- LISP is characterized by the following ideas:
  - Computing with symbolic expressions rather than numbers,
  - representation of symbolic expressions and other information by list structure,
  - composition of functions as a tool for forming more complex functions out of a few primitive operations,
  - the representation of LISP programs as LISP data, and the function *eval* that serves both as a formal definition of the language and as an interpreter.

Sample Lisp Code

```
(define (primes)
      (letrec ((sieve (lambda (s)
       (cons (car s)
             (delay (sieve (filter
              (lambda (n)
      (> (remainder n (car s)) 0))
             (force (cdr s)))))))))
      (sieve (force (cdr nat)))))
```


Why is Lisp Interesting from VM Designer's Viewpoint?

- The first language to widely use garbage collection as a means of automating memory management.
- The first language to use recursion extensively.
- One of the first truly interactive languages that didn't require a “compile-link-execute-crash-debug” cycle.
- Lisp was one of the first systems where programs run in a “sandbox”; access to the operating system is constrained and programs cannot crash the system.
- The first truly “reflective” programming language as well; LISP has a very small language core; the rest of the system can be written in itself; programs can be manipulated as data.

UCSD Pascal

- The Pascal language was developed by Nicklaus Wirth in 1969.
- A fairly "conventional" programming language.
  - Predecessor to a large family of other languages (Modula..., Oberon...)
- Pascal did not become popular until Ken Bowles of the University of California San Diego (UCSD) implemented the *P-Code system* in the late 1970s.
  - A portable pseudocode system/language runtime.
  - http://www.threedee.com/jcm/psystem/
  - The P-Code system made the implementation extremely portable, increasing the popularity of Pascal rapidly.

Sample Pascal Code

```
PROGRAM Fibonacci(input,output);
VAR
  lo : INTEGER; hi : INTEGER; n : INTEGER;
  golden_ratio : DOUBLE; ratio : DOUBLE;
BEGIN
  golden_ratio := (1.0 + sqrt(5.0))/2.0;
  lo := 1; hi := 1; n := 1;
  WHILE hi > 0 DO
  BEGIN
    n := n + 1; ratio := hi / lo;
    WRITELN(n : 2, ' ', hi, ratio : 25, ' ', (ratio - golden_ratio) : 21 : 18);
    hi := lo + hi; lo := hi - lo
  END
END
```

Why is UCSD Pascal Interesting from VM Designer's Viewpoint?

- The P-Code system popularized the idea of using *pseudocode* to improve portability of programming language runtime systems.
- Uses a stack-oriented instruction set and five virtual registers.
  - Only one stack (no separate operand & call stacks)
- The first virtual machine implementation widely available to hobbyists.
  - Especially the Apple II implementation was very popular.
  - http://homepages.cwi.nl/~steven/pascal/book/10pcode.html
  - http://www.wikipedia.org/wiki/P-Code_machine

P-Code Sample Instructions

Inst.	Stack before	Stack after	Description
ADI	i1 i2	i1+i2	add two integers
ADR	r1 r2	r1+r2	add two reals
DVI	i1 i2	i1/i2	integer division
INN	i1 s1	b1	set membership; b1 = whether i1 is a member of s1
LDCI	i1	i1	load integer constant
MOV	a1 a2		move
NOT	b1	~b1	boolean negation

BASIC

- Beginners All-purpose Symbolic Instruction Code.
- Developed by John Kemeny and Thomas Kurtz at Dartmouth College (USA) in mid-1960s.
  - http://www.kbasic.org/1/history.php3
- Interactive nature made it suitable for mini- and microcomputers (great timing!)
- Paul Allen and Bill Gates wrote the first interpreted implementation in 1975; this improved the portability of the language dramatically.

Sample BASIC Code

```
100 INPUT “Type a number”; N
120 IF N <= 0 GOTO 200
130 PRINT “Square root=” SQR(N)
140 GOTO 100
200 PRINT “Number must be > 0”
210 GOTO 100
```

Why is BASIC Interesting from VM Designer's Viewpoint?

- It really isn't very interesting...
  - The language has no specific contributions except ease of learning and ease of use.
  - Excessive use of GOTOs led to some horrible programs.
- However, the popularity of BASIC coincided with the microcomputer boom.
  - Many early microcomputer companies decided to integrate BASIC in their products.
  - You could either program in assembly language or BASIC...
- Some BASIC systems used pretty interesting intermediate code representation techniques.

Forth

- Invented by Charles Moore in the early 1970s.
  - http://www.forth.com/Content/History/History1.htm
- Originally designed to control radiotelescopes.
- Characteristics:
  - Forth is a "word-oriented" programming language; there is no syntax or grammar in the traditional sense.
  - All the primitive functions/words are also language keywords; open stack used for parameter passing.
  - Forth makes subroutine definition extremely cheap; this provides for extensibility and high level of procedural abstraction.
  - Extreme minimalism: The entire Forth system (including a simple multitasking programming environment) can fit in 8-15 kilobytes.

Sample Forth Code

```
: xReverse              \ reverse the horizontal direction of the ball
        xStep @ +/- xStep ! ;
: yReverse              \ reverse the vertical direction of the ball
        yStep @ +/- yStep ! ;
: checkLeft             \ check for the left edge of the screen
        x @ 1 <= IF xReverse THEN ;
: checkRight            \ check for the right edge of the screen
        x @ xMax >= IF xReverse THEN ;

ASCII o CONSTANT "ball"

: showBall              \ draw the ball on the screen
        "ball" xyPlot ;
: hideBall              \ hide (undraw) the ball
        "bl" xyPlot ;
: tryBall \ test the ball drawing routines
        BEGIN
         showBall
         checkLeft checkRight checkTop checkBottom
         hideBall xyStep
        AGAIN ;
```


Why is Forth Interesting from the VM Designer's Viewpoint?

- One of the easiest virtual machines to build.
- The VM consists of a small number of distinct components (stacks, dictionary, interpreter, virtual registers, primitives); no extra “fat”.
- The language itself is small, simple and efficient, and provides an unusual combination of high-level abstraction and very low level programming capabilities.
- High level of reflection (significant portions of the VM written in the language itself.)
- Ideal for embedded systems (if the awkward syntax is not exposed to the end user...)

Smalltalk

- Developed by Alan Kay's team at Xerox PARC
  - There are various versions (Smalltalk-72, -76, -80). Smalltalk-80 is the best known.
  - http://users.ipa.net/~dwighth/smalltalk/bluebook/bluebook_imp_toc.html
- Characteristics:
  - The first truly interactive object-oriented programming language (unlike Simula which was a compiler-based system.)
  - Took "everything is an object" and "message passing" metaphors to the extreme.
  - Everything is available for modification, even the VM itself (very high level of reflection.)
  - Even code is treated as objects (*blocks*).
  - The language is very closely coupled with a graphical interface; source code of a program cannot be easily separated from the programming environment.

Sample Smalltalk Code

```
| aString vowels |
aString := 'This is a string'.
vowels := aString select: [:aCharacter | aCharacter isVowel].

================================

| rectangles aPoint |
rectangles := OrderedCollection
  with: (Rectangle left: 0 right: 10 top: 100 bottom: 200)
  with: (Rectangle left: 10 right: 10 top: 110 bottom: 210).
aPoint := Point x: 20 y: 20.
collisions := rectangles select: [:aRect | aRect containsPoint: aPoint].
```

Why is Smalltalk Interesting From the VM Designer's Viewpoint?

- Various implementation challenges:
  - Everything can be changed on the fly.
  - No static type system.
  - Even numbers are objects that are manipulated by message passing (= arithmetic operations can be slow.)
  - Blocks (heap-allocated code objects/stack frames) are difficult to implement efficiently.
- Extremely interactive/reflective => fun.
- Well-designed and mature class libraries => easy to write interesting software.
- There are high-quality public domain Smalltalk implementations available.
  - http://www.squeak.org/

Self

- Invented by David Ungar and Randall Smith at Xerox PARC and Stanford University in 1986-1987.
  - The majority of the actual implementation work was done at Sun Labs in the 1990s.
  - http://www.sunlabs.com/self
- Prototype-based flavor/variant of Smalltalk.
  - Took the "everything is an object" metaphor even further than Smalltalk.
  - No more classes; objects can inherit ("delegate") behavior directly from each other.
  - Extremely dynamic language: even the control structures or the inheritance relationships of objects can be changed on the fly.

Sample Self Code

```
acc: bankAccount copy.
acc balance: 100.
b: [acc deposit: 50].
acc balance.          "returns 100"
b value.
b value.
acc balance.          "returns 200"
```


Why is Self Interesting from VM Designer's Viewpoint?

- The Self language is so extremely dynamic that the implementors had to push the limits of VM technology very aggressively:
  - Adaptive compilation to speed up execution.
  - Generational garbage collection (originally invented by David Ungar in his Ph.D. work.)
  - Dynamic deoptimization to allow debugging of highly optimized programs.
  - Novel collaborative / visual programming and debugging environment (tightly integrated with the VM.)
- Many of the key technologies that are used today in mainstream Java virtual machines were invented by the Self group.

Java

- Developed by James Gosling's team at Sun Microsystems in the early 1990s.
- Originally designed for programming consumer devices (as a replacement for C++).
  - Uses a syntax that is familiar to C/C++ programmers.
  - Uses a portable virtual machine that provides automatic memory management and a simple stack-oriented instruction set.
  - *Class file verification* was added to enable downloading and execution of remote code securely.
- Again, great timing: the development of the Java technology coincided with the widespread adoption of web browsers in the mid-1990s.

Sample Java Code

```
class Peg {
   int pegNum;
   int disks[ ] = new int[64];
   int nDisks;

   public Peg(int n, int numDisks) {
      pegNum = n;
      for (int i = 0; i < numDisks; i++) {
         disks[i] = 0;
      }
      nDisks = 0;
   }

   public void addDisk(int diskNum) {
      disks[nDisks++] = diskNum;
   }

   public int removeDisk() {
      return disks[--nDisks];
   }
}
```

Why is Java Interesting from VM Designer's Viewpoint?

- Most people had never heard of virtual machines until Java came along!
- Java brought virtual machines to the realm of mobile computing.
- Java combines a statically compiled programming language with a dynamic virtual machine.
- The Java virtual machine (JVM) is very well documented.
  - Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification*, Second Edition, Addison Wesley, Java Series, April 1999.
- A JVM is seemingly very easy to build.
- However, tight compatibility requirements make the actual implementation very challenging.
  - Must pass tens of thousands of test cases to prove compatibility.

Java for Android / Dalvik VM

- Android resurrected interest in Java in the mobile space.
- *Dalvik* is an alternative runtime environment for executing Java programs, using an Android-specific application format (.dex files).
- Unlike JVM, which uses a stack-based architecture, Dalvik uses a register-based architecture and bytecode set.
- As of Android 5.0 (Lollipop), the Dalvik VM will be replaced by Android Runtime (ART) – an architecture based on ahead-of-time compilation.