The Java Virtual machine

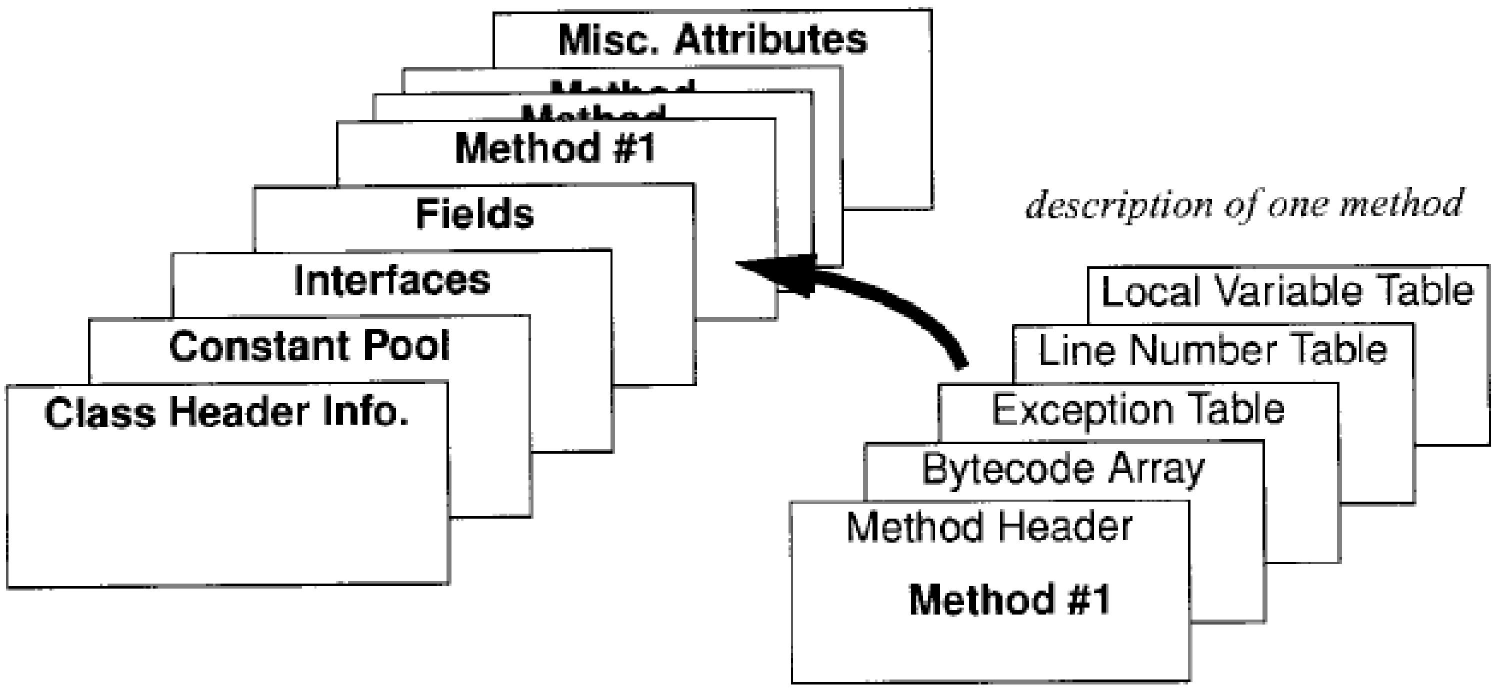
A Main Reference Source

The Java™ Virtual Machine Specification (2nd Ed)
by Tim Lindholm & Frank Yellin
Addison-Wesley, 1999

The book is on-line and available for download:

`http://java.sun.com/docs/books/vmspec/`

The Java Classfile



JVM Runtime Behaviour

- VM startup
- Class Loading/Linking/Initialization
- Instance Creation/Finalisation
- Unloading Classes
- VM exit

VM Startup and Exit

Startup

- Load, link, initialize class containing `main()`
- Invoke `main()` passing it the command-line arguments
- Exit when:
  - all non-daemon threads end, or
  - some thread explicitly calls the `exit()` method

Class Loading

- Find the binary code for a class and create a corresponding Class object
- Done by a class loader – bootstrap, or create your own
- Optimize: prefetching, group loading, caching
- Each class-loader maintains its own namespace
- Errors include: **`ClassFormatError`**, **`UnsupportedClassVersionError`**, **`ClassCircularityError`**, **`NoClassDefFoundError`**

Class Loaders

- System classes are automatically loaded by the bootstrap class loader
- To see which:

```
java -verbose:class Test.java
```

- Arrays are created by the VM, not by a class loader
- A class is unloaded when its class loader becomes unreachable (the bootstrap class loader is never unreachable)

Class Linking - 1. Verification

- Extensive checks that the .classfile is valid
- This is a vital part of the JVM security model
- Needed because of possibility of:
  - buggy compiler, or no compiler at all
  - malicious intent
  - (class) version skew
- Checks are independent of compiler and language

Class Linking - 2. Preparation

- Create static fields for a class
- Set these fields to the standard default values (N.B. not explicit initializers)
- Construct method tables for a class
- ... and anything else that might improve efficiency

Class Linking - 3. Resolution

- Most classes refer to methods/fields from other classes
- Resolution translates these names into explicit references
- Also checks for field/method existence and whether access is allowed

Class Initialization

Happens *once* just before first instance creation, or first use of static variable.

- Initialise the superclass first!
- Execute (class) static initializer code
- Execute explicit initializers for static variables
- May not need to happen for use of *final* static variable
- Completed before anything else sees this class

Instance Creation/Finalisation

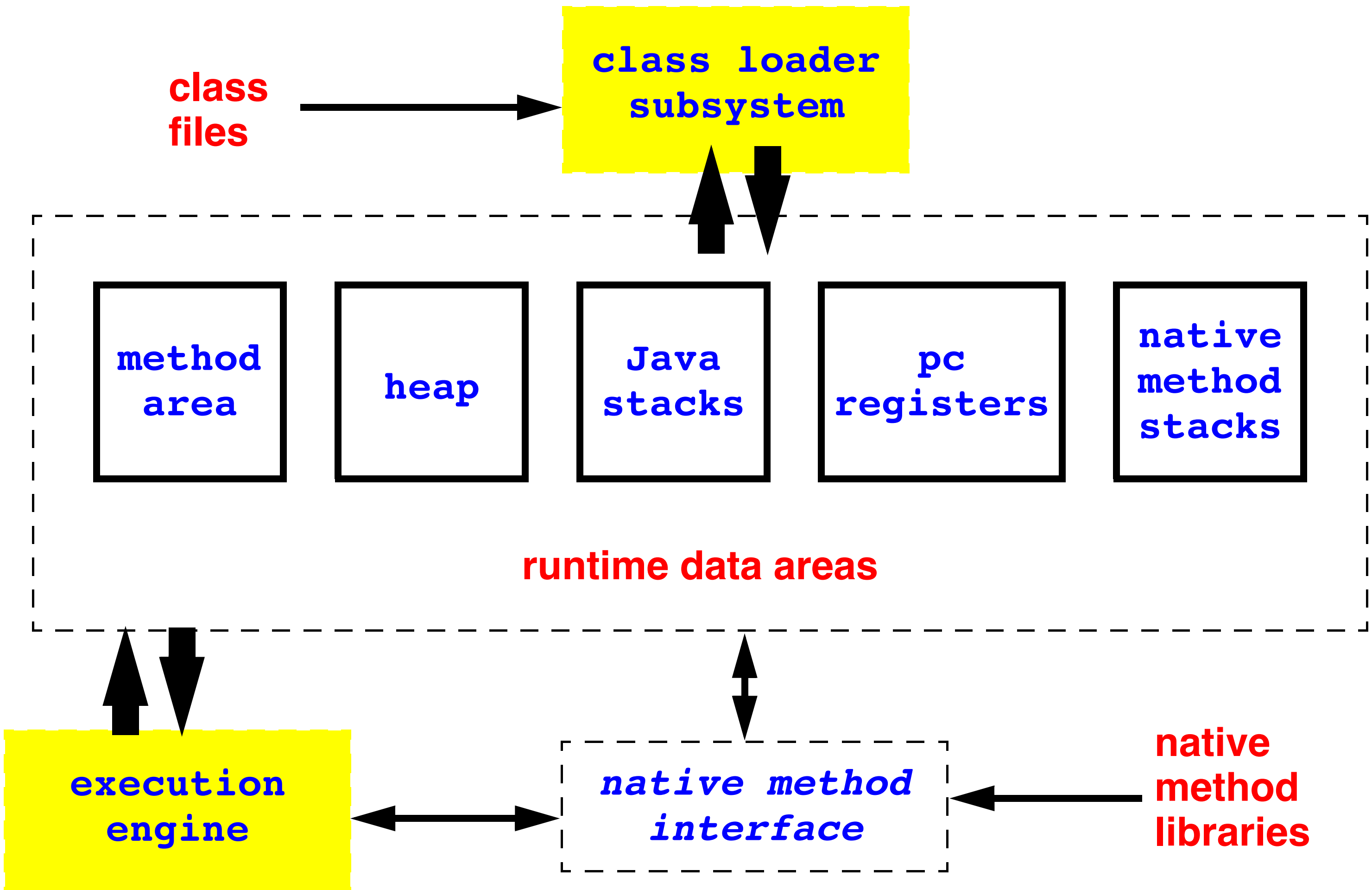
- Instances are created using `new`, or `newInstance()` from class `Class`
- Instances of `String` may be created (implicitly) for String literals
- Process:
  1. Allocate space for all the instance variables (including the inherited ones),
  2. Initialize them with the default values
  3. Call the appropriate constructor (do parent's first)
- _ `finalize()` is called just before garbage collector takes the object (so timing is unpredictable)

JVM Architecture

The internal runtime structure of the JVM consists of:

- One: (i.e. shared by all threads)
  - method area
  - heap
- For each thread, a:
  - program counter (pointing into the method area)
  - Java stack
  - native method stack (system dependent)

Run-Time Data Areas (Venners Figure 5-1)



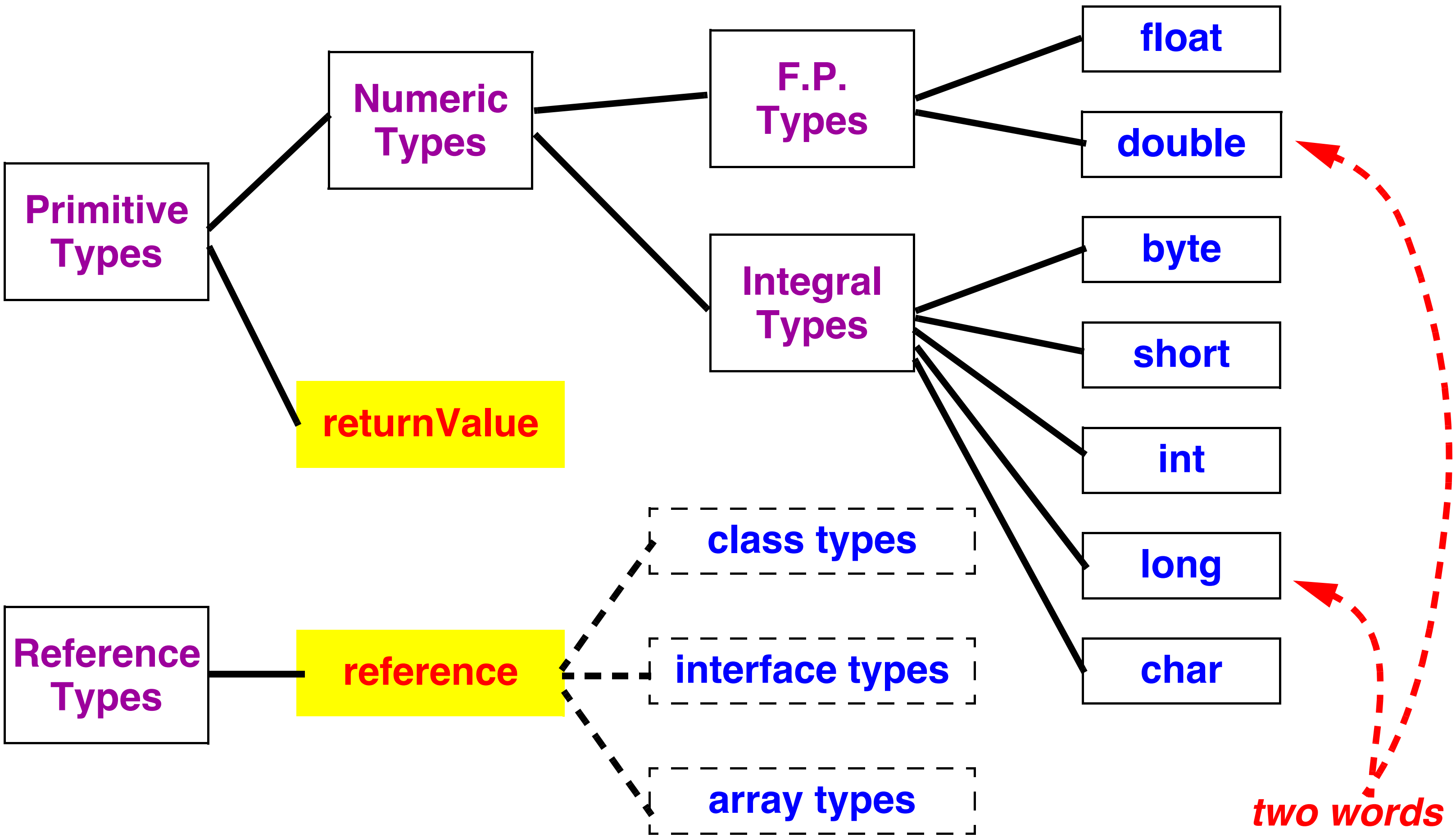
Java Bytecode

Java Intermediate Bytecode

- By James Gosling; presented at IR'95.

- Quick overview:
  - argue for the presence of type information in the bytecode
  - benefits for checkability (because speed/security)
  - reduced dependencies on environment

Datatypes of the JVM (Venners 5-4)



Control Transfer

- ifeq, iflt, ifle, ifne, ifgt, ifge
- ifnull, ifnonnull
- if_icmpeq, if_icmplt, if_icmple, if_icmpne, if_icmpgt, if_icmpge
- if_acmpeq, if_acmpne
- goto, goto_w, jsr, jsr_w, ret

Switch statement implementation

- tableswitch, lookupswitch

Comparison operations for long, float & double types

- lcmp, fcmpl, fcmpg, dcmpl, dcmpg

Load and Store Instructions

Transferring values between local variables and operand stack

- iload, lload, fload, dload, aload

  and special cases of the above: iload_0, iload_1 ...
- istore, lstore, fstore, dstore, astore

Pushing constants onto the operand stack

- bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1

  and special cases: iconst_0, iconst_1, ...

Arithmetic Operations

Operands are normally taken from operand stack and the result pushed back there

- iadd, ladd, fadd, dadd
- isub ...
- imul ...
- idiv ...
- irem ...
- ineg ...
- iinc

Bitwise Operations

- ior, lor
- iand, land
- ixor, lxor
- ishl, lshl
- ishr, iushr, lshr, lushr

Type Conversion Operations

Widening Operations

- i2l, i2f, i2d, l2f, l2d, f2d

Narrowing Operations

- i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, d2f

Operand Stack Management

- pop, pop2
- dup, dup2, dup_x1, dup_x2, dup2_x2, swap

Object Creation and manipulation

- new
- newarray, anewarray, multinewarray
- getfield, putfield, getstatic, putstatic
- baload, caload, saload, iaload, laload, faload, daload, aaload
- bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore
- arraylength
- instanceof, checkcast

Object Creation and manipulation

- new
- newarray, anewarray, multinewarray
- getfield, putfield, getstatic, putstatic
- baload, caload, saload, iaload, laload, faload, daload, aaload
- bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore
- arraylength
- instanceof, checkcast

Method Invocation / Return

- invokevirtual
- invokespecial
- invokeinterface
- invokestatic
- ireturn, freturn, dreturn, areturn
- return

Java Intermediate Bytecode

- Observation:
  - Original goals where modularity, small footprint, verifiability, but not speed.
  - the bytecode had to be statically typed (speed/safety argument)
  - control flow merges must have the same incoming stack types
  - use symbolic references to environment (fragile base class)

Class Resolution

- CP entry tagged `CONSTANT_Class` can be either class/interface.
- Execution of an instruction that refers to a class:

    1. *search for class in the classloader hierarchy*
    2. *if not found, initiate class loading*

- ... much more to the story.

Method Invocation

`INVOKEVIRTUAL,`	- instance method
`INVOKEINTERFACE,`	- interface method
`INVOKESPECIAL`	- constructor/private/super method
`INVOKESTATIC`	- class method

```
foo/baz/Myclass/myMethod(Ljava/lang/String;)V
---------------        ---------------------
      |         --------          |
      |             |             |
  classname   methodname     descriptor
```

- When an invocation is executed the method must be resolved.

Method Resolution

1. Checks if C is class or interface.

    If C is interface, throw `IncompatibleClassChangeError`.

2. Look up the referenced method in C and superclasses:

    - Success if C has method with same name & descriptor
    - Otherwise, if C has a superclass, repeat 2 on super.

3. Otherwise, locate method in a superinterface of C

    - If found success.
    - Otherwise, fail.

Method Invocation

- Resolution is rather work intensive. Can this be done faster?

class initialization

- Before use of static field, static method, object creation, a class must be initialized.
- Initialization involves creating a new Class object, and running the static initializers.
- Every operation that could trigger initialization must check the status of the class.

subroutines

- Subroutines were added to the bytecode to reduce the space requirements of exception handler's finally clauses.

Example

```java
int bar(int i) {
    try {
        if (i == 3) return this.foo();
    } finally {
        this.ladida();
    }
    return i;
}
```

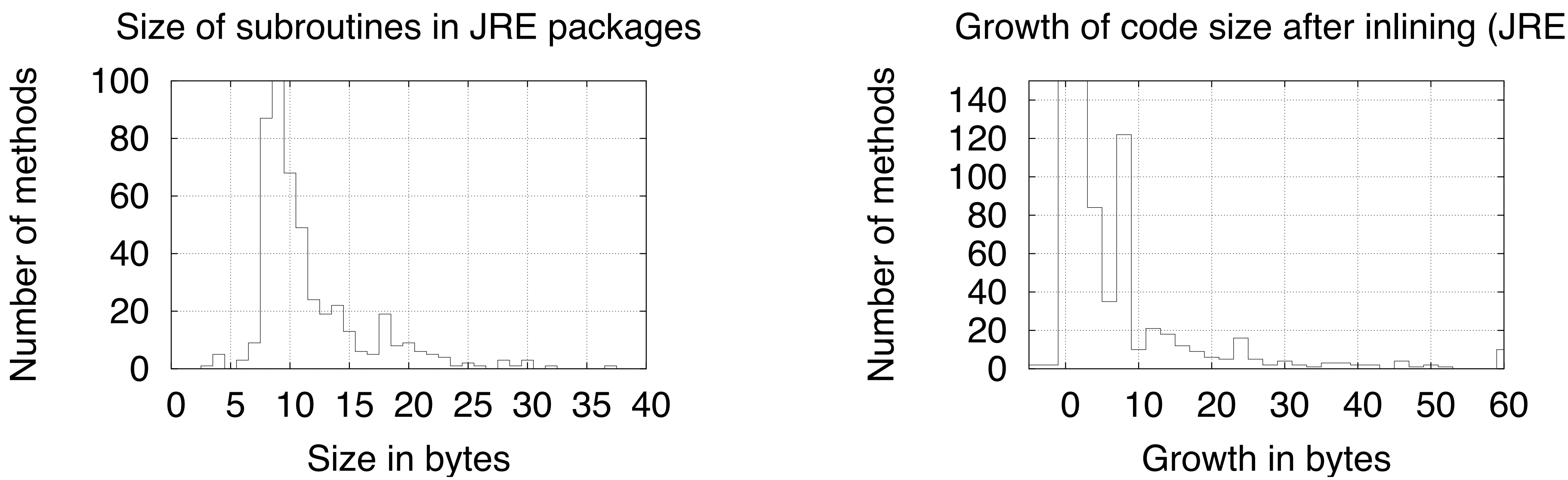
Region	Target
1–12	17
13–16	21

```
01   iload_1               // Push i
02   iconst_3              // Push 3
03   if_icmpne 10          // Goto 10 if i does not e

04   aload_0               // Push this
05   invokevirtual foo     // Call this.foo
06   istore_2              // Save result of this.foo
07   jsr 13                // Do finally block before
08   iload_2               // Recall result from this
09   ireturn               // Return result of this.f

10   jsr 13                // Do finally block before

11   iload_1               // Push i
12   ireturn               // Return i
     // finally block
13   astore_3              // Save return address in
14   aload_0               // Push this
15   invokevirtual ladida  // Call this.ladida()
16   ret 3                 // Return to address saved
     // Exception handler for try body
17   astore_2              // Save exception
18   jsr 13                // Do finally block
19   aload_2               // Recall exception
20   athrow                // Rethrow exception
     // Exception handler for finally body
21   athrow                // Rethrow exception
```

PURDUE UNIVERSITY

subroutines

- Over the JDK 1.1, subroutines save a total of 2427 bytes [Freund98].
- Java5 does not use them. They can be inlined by tools.



From Artho, Biere, Bytecode 2005.

Figure 7. Sizes of subroutines and size increase after inlining.

PURDUE UNIVERSITY

class compression

- Observation:
  - class file size dominated by symbolic information in the CP
  - JAR files (containing multiple classes) contain redundancies

[Pugh99]

	swingall	javac
Total size	3,265	516
excluding jar overhead	3,010	485
Field definitions	36	7
Method definitions	97	10
Code	768	114
Other	72	12
Constant pool	2,037	342
Utf8 entries	1,704	295
if shared	372	56
if shared and factored	235	26

compression

- Observation:
  - class file size dominated by symbolic information in the CP
  - JAR files (containing multiple classes) contain redundancies

[Bradley,Horspool,Vitek,98]

File Format	Size	% orig. size
JAR file, uncompressed	260,178	100.0%
JAR file, compressed	132,600	51.0%
Clazz	97,341	37.4%
Gzip	97,223	37.4%
Jazz	59,321	22.8%

Java Virtual Machine, part three

Verification

- Ensures that the type (i.e. the loaded class) obeys Java semantics, and
- will not violate the integrity of the JVM.

There are many aspects to verification

Verification, cont'd

Some Checks during Loading

- If it's a classfile, check the magic number (`0xCAFEBABE`),
- make sure that the file parses into its components correctly

Additional Checks after/during Loading

- make sure the class has a superclass (only Object does not)
- make sure the superclass is not final
- make sure final methods are not overridden
- if a nonabstract class, make sure all methods are implemented
- make sure there are no incompatible methods
- make sure constant pool entries are consistent

Additional Checks after/during Loading, cont'd

- check the format of special strings in the constant pool (such as method signatures etc)

A Final Check (required before method is executed)

- verify the integrity of the method's bytecode

This last check is very complicated (so complicated that Sun got it wrong a few times)

Verifying Bytecode

The requirements

- All the opcodes are valid, all operands (e.g. number of a field or a local variable) are in range.
- Every control transfer operation (goto, ifne, ...) must have a destination which is in range and is the start of an instruction
- Type correctness: every operation receives operands with the correct datatypes
- No stack overflow or underflow
- A local variable can never be used before it has been initialized
- Object initialization – the constructor must be invoked before the class instance is used

The requirements, cont'd

- Execution cannot fall off the end of the code
- The code does not end in the middle of an instruction
- For each exception handler, the start and end points must be at the beginnings of instructions, and the start must be before the end
- Exception handler code must start at the beginning of an instruction

Sun’s Verification Algorithm

A *before* state is associated with each instruction.

The state is:

- contents of operand stack (stack height, and datatype of each element), plus
- contents of local variables (for each variable, we record *uninitialized* or *unusable* or the datatype)

A datatype is integral, long, float, double or any reference type

Each instruction has an associated *changed* bit:

- all these bits are false,
- except the first instruction whose changed bit is true.

Sun's Verification Algorithm, cont'd

```
do forever {
   find an instruction I whose changed bit is true;
   if no such instruction exists, return SUCCESS;
   set changed bit of I to false;
   state S = before state of I;

   for each operand on stack used by I
      verify that the stack element in S has correct datatype
      and pop the datatype from the stack in S;
   for each local variable used by I
      verify that the variable is initialized and
      has the correct datatype in S;
   if I pushes a result on the stack,
      verify that the stack in S does not overflow, and
      push the datatype onto the stack in S;
   if I modifies a local variable,
      record the datatype of the variable in S
                       ... continued
```

Sun's Verification Algorithm, cont'd

```
    determine SUCC, the set of instructions which can follow I;
        (Note: this includes exception handlers for I)

    for each instruction J in SUCC do
        merge next state of I with the before state of J
        and set J's changed bit if the before state changed;
        (Special case: if J is a destination because of an exception
        then a special stack state containing a single instance of
        the exception object is created for merging with the before
        state of J.)
} // end of do forever
```

Verification fails if a datatype does not match with what is required by the instruction, the stack underflows or overflows, or if two states cannot be merged because the two stacks have different heights.

Sun’s Verification Algorithm, cont’d

Merging two states

- Two stack states with the same height are merged by pairwise merging the types of corresponding elements.
- The states of the two sets of local variables are merged by merging the types of corresponding variables.

The result of merging two types:

- Two types which are identical merge to give the same type
- For two types which are not identical:
  if they are both references, then the result is the first common superclass (lowest common ancestor in class hierarchy); otherwise the result is recorded as *unusable.*

Example (Leroy, Figure 1):

```
static int factorial( int n ) {
   int res;
   for (res = 1; n > 0; n--) res = res * n;
   return res;
}
```

Corresponding JVM bytecode:

```
method static int factorial(int), 2 variables, 2 stack slots
    0: iconst_1         // push the integer constant 1
    1: istore_1         // store it in variable 1 (res)
    2: iload_0          // push variable 0 (the n parameter)
    3: ifle 14          // if negative or null, go to PC 14
    6: iload_1          // push variable 1 (res)
    7: iload_0          // push variable 0 (n)
    8: imul             // multiply the two integers at top of stack
    9: istore_1         // pop result and store it in variable 1
   10: iinc 0, -1       // decrement variable 0 (n) by 1
   11: goto 2           // go to PC 2
   14: iload_1          // load variable 1 (res)
   15: ireturn          // return its value to caller
```

Sun’s Analysis Algorithm

Chng’d	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
X	`()`	`(I,T)`	`0: iconst_1`		
-	`?`	`(?,?)`	`1: istore_1`		
-	`?`	`(?,?)`	`2: iload_0`		
-	`?`	`(?,?)`	`3: ifle 14`		
-	`?`	`(?,?)`	`6: iload_1`		
-	`?`	`(?,?)`	`7: iload_0`		
-	`?`	`(?,?)`	`8: imul`		
-	`?`	`(?,?)`	`9: istore_1`		
-	`?`	`(?,?)`	`10: iinc 0, -1`		
-	`?`	`(?,?)`	`11: goto 2`		
-	`?`	`(?,?)`	`14: iload_1`		
-	`?`	`(?,?)`	`15: ireturn`		

where I = integral; T = *uninitialized/unusable*; ? = $\perp$ = *unknown*

Sun's Analysis Algorithm - after 1 step

Chng'd	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
-	()	(I,T)	0: iconst_1	(I)	(I,T)
X	(I)	(I,T)	1: istore_1		
-	?	(?,?)	2: iload_0		
-	?	(?,?)	3: ifle 14		
-	?	(?,?)	6: iload_1		
-	?	(?,?)	7: iload_0		
-	?	(?,?)	8: imul		
-	?	(?,?)	9: istore_1		
-	?	(?,?)	10: iinc 0, -1		
-	?	(?,?)	11: goto 2		
-	?	(?,?)	14: iload_1		
-	?	(?,?)	15: ireturn		

Sun’s Analysis Algorithm - after 4 steps

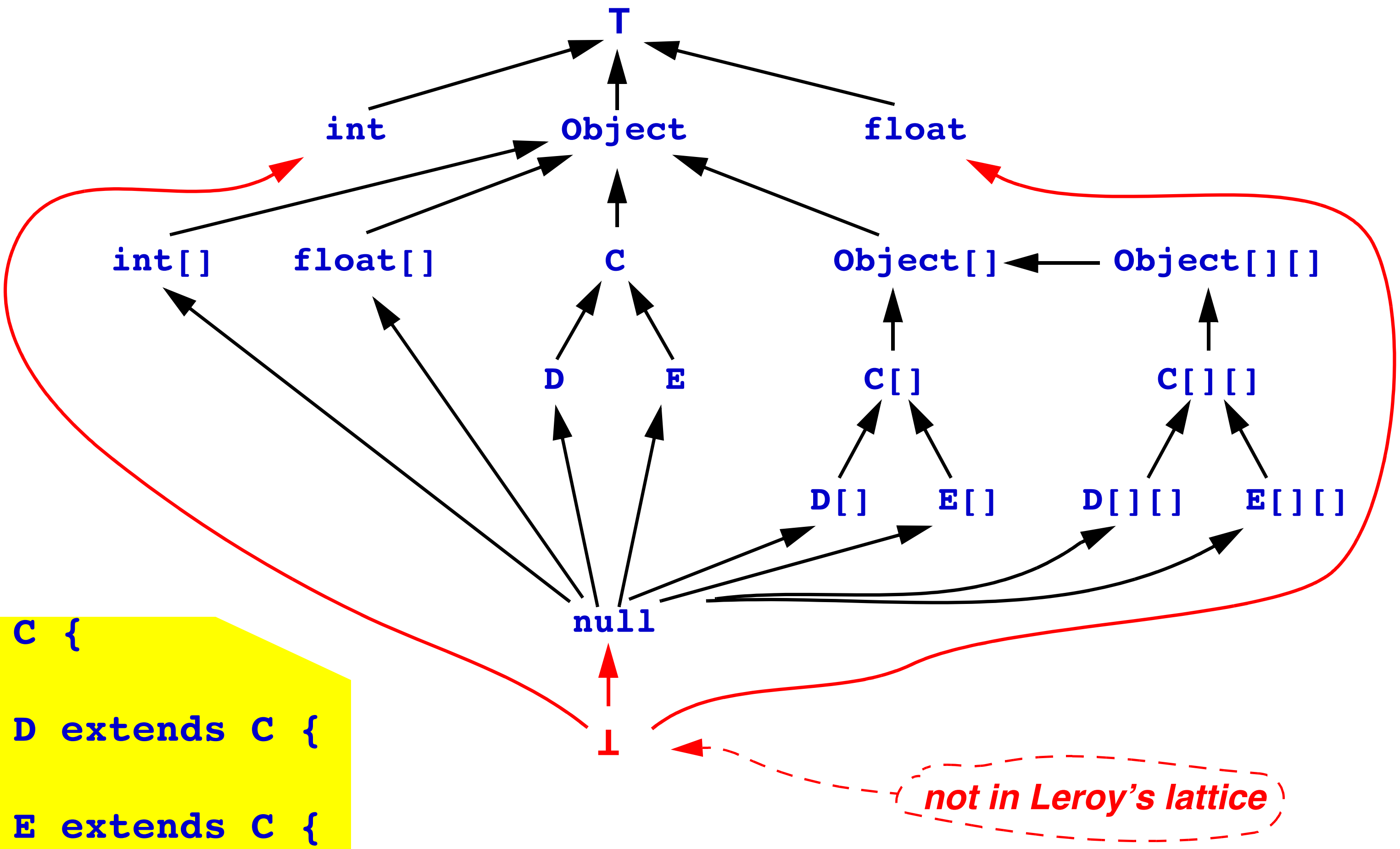
Chng’d	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
-	()	(I,T)	0: iconst_1		
-	(I)	(I,T)	1: istore_1		
-	()	(I,I)	2: iload_0		
-	(I)	(I,I)	3: ifle 14	()	(I,I)
X	()	(I,I)	6: iload_1		
-	?	(?,?)	7: iload_0		
-	?	(?,?)	8: imul		
-	?	(?,?)	9: istore_1		
-	?	(?,?)	10: iinc 0, -1		
-	?	(?,?)	11: goto 2		
X	()	(I,I)	14: iload_1		
-	?	(?,?)	15: ireturn		

Analysis Algorithm - after 12 steps

Chng'd	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
-	()	(I,T)	0: iconst_1		
-	(I)	(I,T)	1: istore_1		
-	()	(I,I)	2: iload_0		
-	(I)	(I,I)	3: ifle 14		
-	()	(I,I)	6: iload_1		
-	(I)	(I,I)	7: iload_0		
-	(I,I)	(I,I)	8: imul		
-	(I)	(I,I)	9: istore_1		
-	()	(I,I)	10: iinc 0, -1		
-	()	(I,I)	11: goto 2		
-	()	(I,I)	14: iload_1		
-	(I)	(I,I)	15: ireturn	()	(I,I)

and we have completed the verification without error.

Some of the Lattice of Types (Leroy, Figure 3)




```
class C {
}
class D extends C {
}
class E extends C {
}
```

Merging Types

- The lattice represents an ordering relation on types
- The lattice is derived from the semantics of Java (and is based on the class hierarchy)
- Given any two types t_1 and t_2, there is a least upper bound type, *lub*(t_1,t_2)
- Given any type t, the length of the path from t to top, T, is finite (the well-foundedness property).

The step in Sun's verification algorithm where types are merged is implemented as *lub*.

The finiteness property guarantees that Sun's algorithm will converge in a finite number of steps.