The Java Language Implementation

Reading

- Chapter 13, sections 13.4 and 13.5
- Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, pages 1–5.

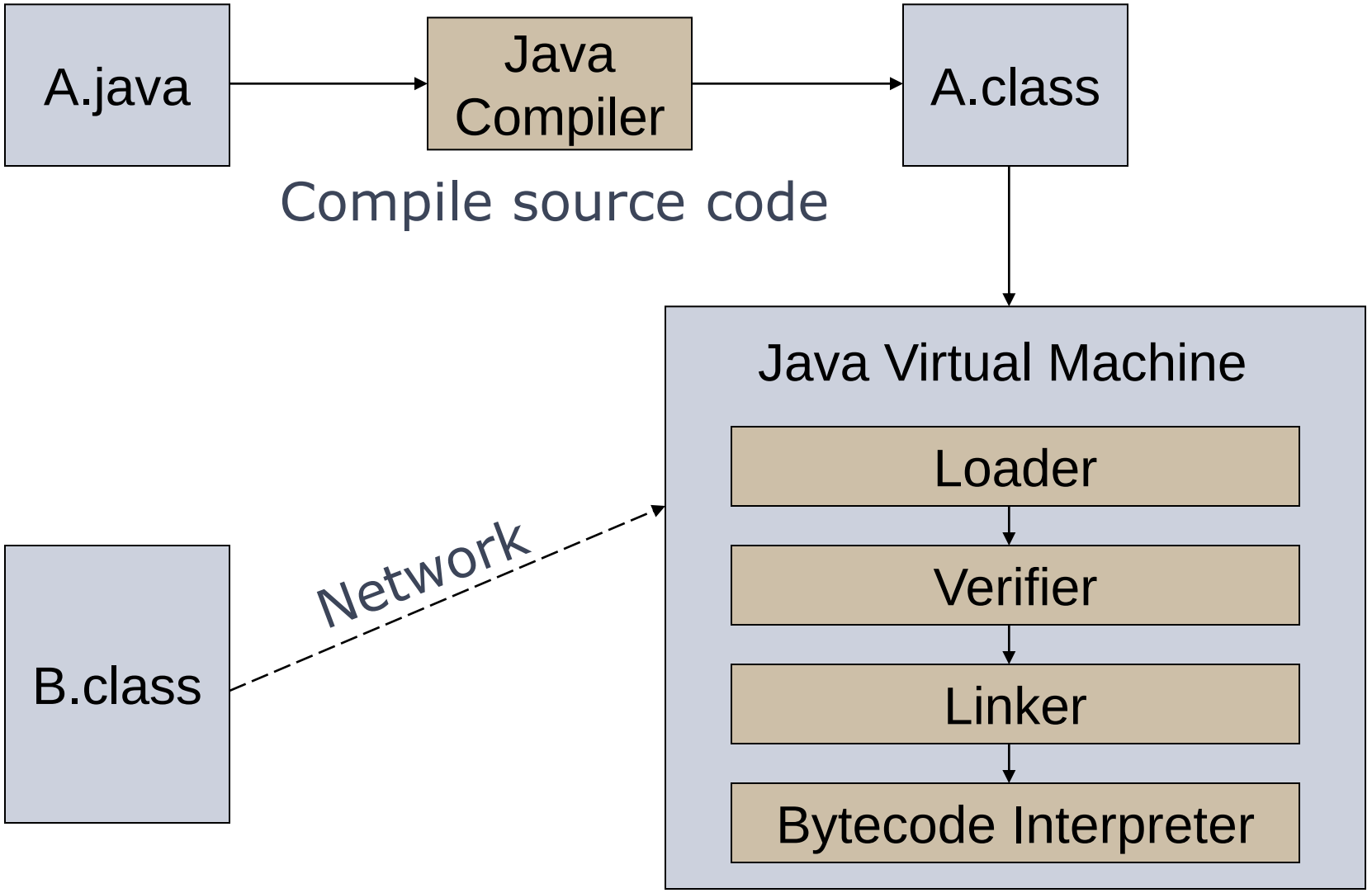
Outline

- Java virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- JVM Method lookup
  - four different bytecodes
- Verifier analysis
- Method lookup optimizations (beyond Java)
- Java security
  - Buffer overflow
  - Java “sandbox”
  - Stack inspection

Java Implementation

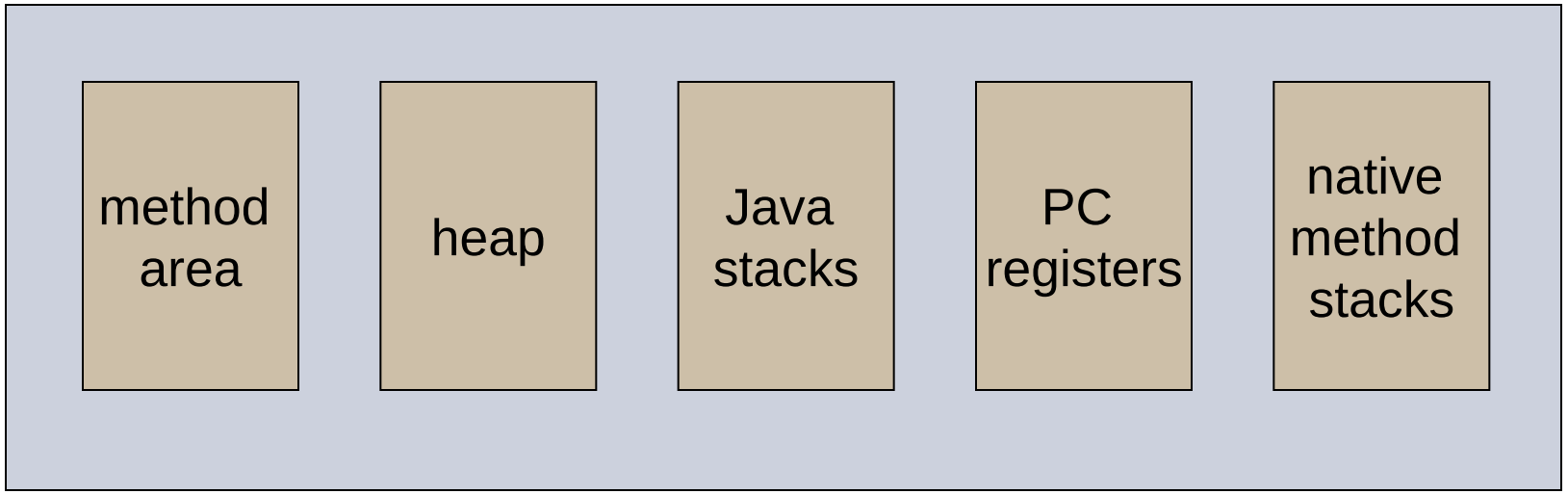
- Compiler and Virtual Machine
  - Compiler produces bytecode
  - Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode
- Why this design?
  - Bytecode interpreter/compilers used before
    - Pascal “pcode”; Smalltalk compilers use bytecode
  - Minimize machine-dependent part of implementation
    - Do optimization on bytecode when possible
    - Keep bytecode interpreter simple
  - For Java, this gives portability
    - Transmit bytecode across network

Java Virtual Machine Architecture



JVM memory areas

- Java program has one or more threads
- Each thread has its own stack
- All threads share same heap



Class loader

- Runtime system loads classes as needed
  - When class is referenced, loader searches for file of compiled bytecode instructions
- Default loading mechanism can be replaced
  - Define alternate ClassLoader object
    - Extend the abstract ClassLoader class and implementation
    - ClassLoader does not implement abstract method loadClass, but has methods that can be used to implement loadClass
  - Can obtain bytecodes from alternate source
    - VM restricts applet communication to site that supplied applet

Example issue in class loading and linking:

Static members and initialization

```
class ... {
   /*  static variable with initial value */
  static int x = initial_value
  /* ---- static initialization block       --- */
  static {   /* code executed once, when loaded */  }
}
```

- Initialization is important
  - Cannot initialize class fields until loaded
- Static block cannot raise an exception
  - Handler may not be installed at class loading time

JVM Linker and Verifier

- Linker
  - Adds compiled class or interface to runtime system
  - Creates static fields and initializes them
  - Resolves names
    - Checks symbolic names and replaces with direct references
- Verifier
  - Check bytecode of a class or interface before loaded
  - Throw VerifyError exception if error occurs

Verifier

- Bytecode may not come from standard compiler
  - Evil hacker may write dangerous bytecode
- Verifier checks correctness of bytecode
  - Every instruction must have a valid operation code
  - Every branch instruction must branch to the start of some other instruction, not middle of instruction
  - Every method must have a structurally correct signature
  - Every instruction obeys the Java type discipline

Last condition is complicated .

Bytecode interpreter

- Standard virtual machine interprets instructions
  - Perform run-time checks such as array bounds
  - Possible to compile bytecode class file to native code
- Java programs can call native methods
  - Typically functions written in C
- Multiple bytecodes for method lookup
  - invokevirtual - when class of object known
  - invokeinterface - when interface of object known
  - invokestatic - static methods
  - invokespecial - some special cases

Type Safety of JVM

- Run-time type checking
  - All casts are checked to make sure type safe
  - All array references are checked to make sure the array index is within the array bounds
  - References are tested to make sure they are not null before they are dereferenced
- Additional features
  - Automatic garbage collection
  - No pointer arithmetic

If program accesses memory, that memory is allocated to the program and declared with correct type

JVM uses stack machine

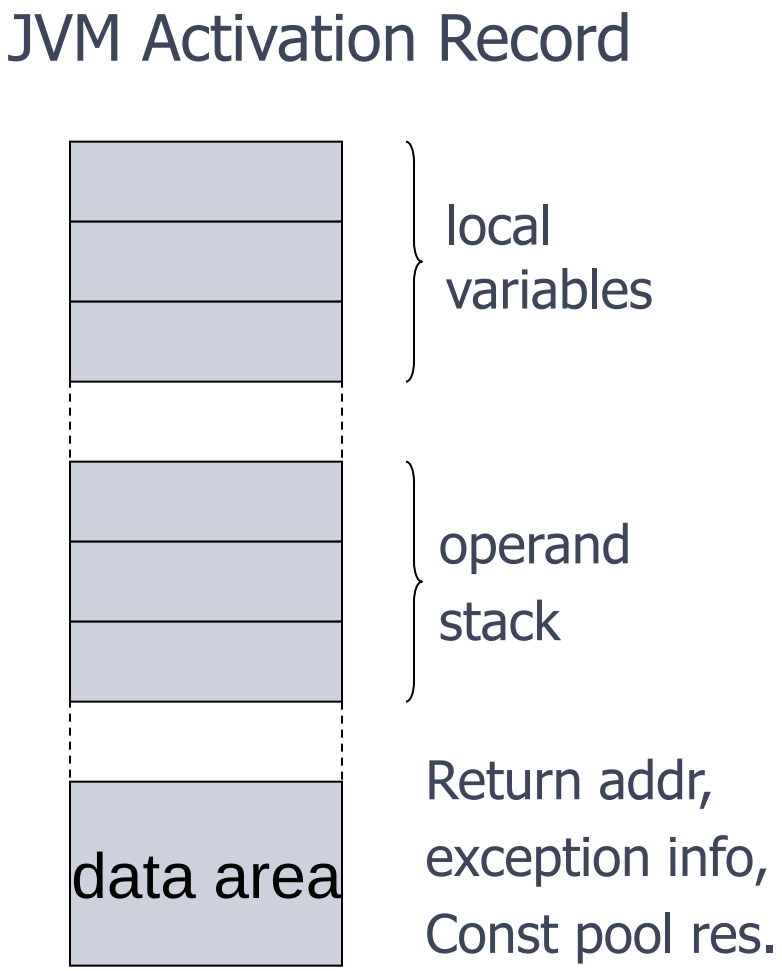
- Java

```
Class A extends Object {
    int I;
    void f(int val) { i = val + 1;}
}
```

- Bytecode

```
Method void f(int)
    aload 0   ; object ref this
    iload 1   ; int val
    iconst 1
    iadd      ; add val +1
    putfield #4 <Field int i>
    return
```

refers to constant pool



Field and method access

- Instruction includes index into constant pool
  - Constant pool stores symbolic names
  - Store once, instead of each instruction, to save space
- First execution
  - Use symbolic name to find field or method
- Second execution
  - Use modified “quick” instruction to simplify search

Outline

- Java virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- JVM Method lookup
  - four different bytecodes
- Verifier analysis
- Method lookup optimizations (beyond Java)
- Java security
  - Buffer overflow
  - Java “sandbox”
  - Stack inspection

Two cases in more detail

- Source code provides interface to object
  - Method lookup using Smalltalk-like search process
  - Cache last offset in case next lookup is same class
- Source code provides class or superclass
  - Method lookup uses Smalltalk-like search first time
    - Reason: run-time class loading; compiler doesn't know representation of classes in different class files
  - Rewrite bytecode so that fixed offset on next lookup

invokeinterface <method-spec>

- Sample code

  void add2(Incrementable x) { x.inc(); x.inc(); }

- Search for method
  - find class of the object operand (operand on stack)
    - must implement the interface named in <method-spec>
  - search the method table for this class
  - find method with the given name and signature
- Call the method
  - Usual function call with new activation record, etc.

Why is search necessary?

```
interface A {
    public void f();
}
interface B {
    public void g();
}
class C implements A, B {
    …;
}
```

Class C cannot have method f first *and* method g first

But if class instead of interface…

- Sample code

```
void deposit1(Account a) { a.deposit(1) …}
```

- Class hierarchy

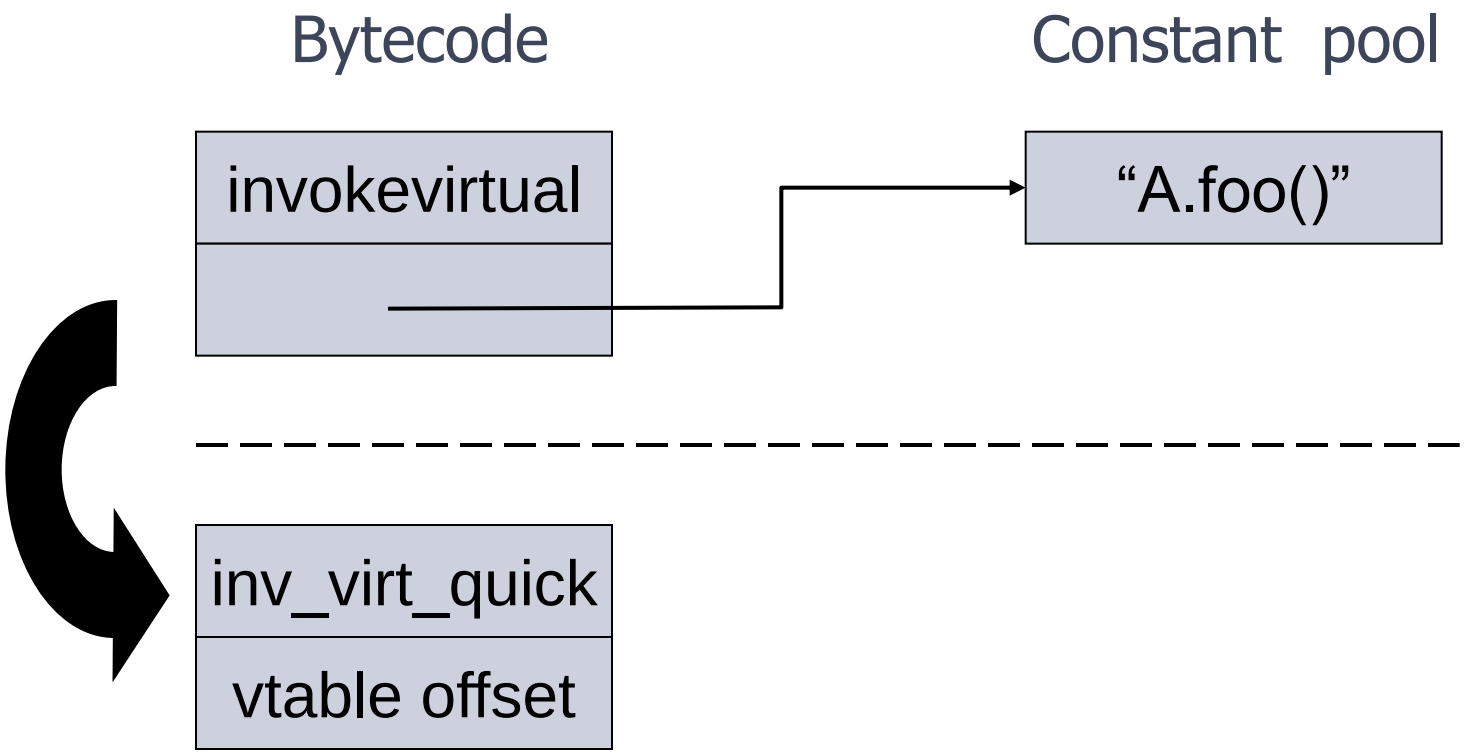
```
class Account {
   public void deposit(int i);
}
class  InterestAccount extends Account {
   …
}
```

Single inheritance guarantees derived class vtable uses same order as base class vtable; remains true if class also *implements* many interfaces

invokevirtual <method-spec>

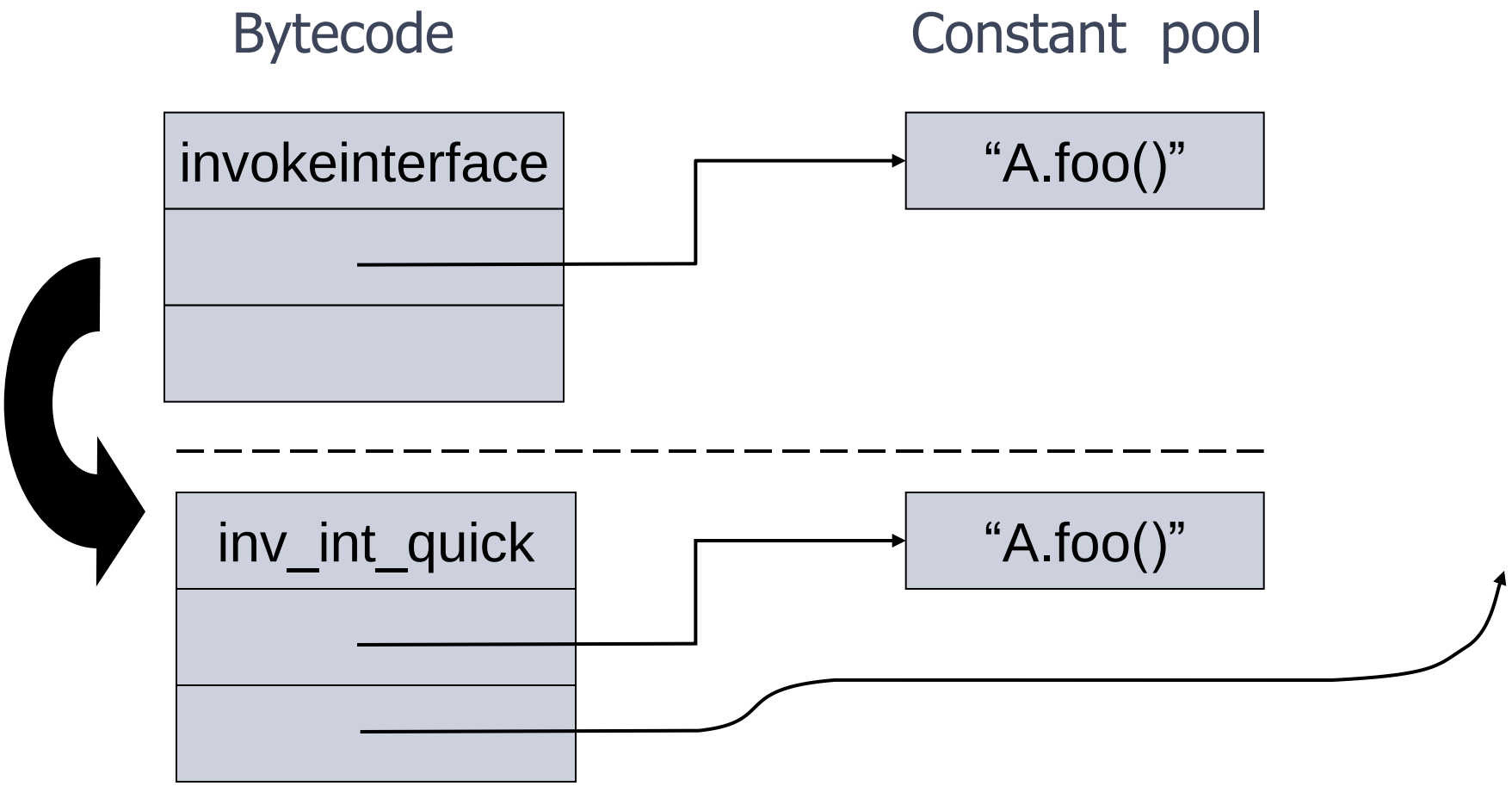
- Similar to invokeinterface, but class is known
- Search for method
  - search the method table of this class
  - find method with the given name and signature
- Can we use static type for efficiency?
  - Each execution of an instruction will be to object from subclass of statically-known class
  - Constant offset into vtable
    - like C++, but dynamic loading makes search useful first time
  - See next slide

Bytecode rewriting: invokevirtual



- After search, rewrite bytcode to use fixed offset into the vtable. No search on second execution.

Bytecode rewriting: invokeinterface



Cache address of method; check class on second use

Outline

- Java virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- JVM Method lookup
  - four different bytecodes
- Verifier analysis
- Method lookup optimizations (beyond Java)
- Java security
  - Buffer overflow
  - Java “sandbox”
  - Stack inspection

Bytecode Verifier

- Let's look at one example to see how this works
- Correctness condition
  - No operations should be invoked on an object until it has been initialized
- Simplified bytecode instructions
  - new ⟨class⟩ allocate memory for object
  - init ⟨class⟩ initialize object on top of stack
  - use ⟨class⟩ use object on top of stack

(idealization for purpose of presentation)

Object creation

- Example:

  Point p = new Point(3) Java source

  1: new Point
  2: dup
  3: iconst 3
  4: init Point } bytecode

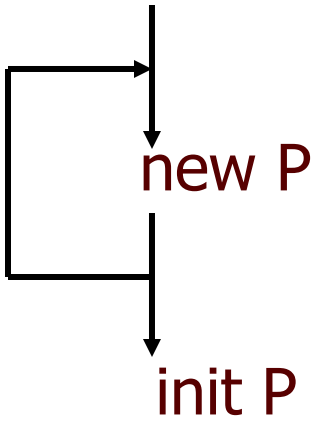
- No easy pattern to match
- Multiple refs to same uninitialized object
  - Need some form of alias analysis

Alias Analysis

- Other situations:

1: new P
2: new P
3: init P

or



- Equivalence classes based on line where object was created

Tracking initialize-before-use

- Alias analysis uses line numbers
  - Two pointers to “unitialized object created at line 47” are assumed to point to same object
  - All accessible objects must be initialized before jump backwards (possible loop)
- Oversight in early treatment of local subroutines
  - Used in implementation of **`try-finally`**
  - Object created in **`finally`** not necessarily initialized
- No clear security consequence
  - Bug fixed

Have proved correctness of modified verifier for init

Aside: bytecodes for try-finally

- Idea
  - Finally clause implemented as lightweight subroutine
- Example code

```
static int f(boolean bVal) {
        try {
            if (bVal) { return 1; }
            return 0;
         }
         finally {
             System.out.println(“About to return");
} }
```

- Bytecode on next slide
  - Print before returning, regardless of which return is executed

(from http://www.javaworld.com/javaworld/jw-02-1997/jw-02-hood.html?page=2)

Bytecode

```
0 iload_0       // Push local variable 0
1 ifeq 11       // Jump on test
4 iconst_1      // Push int 1
5 istore_3      // Pop an int  (the 1), store into local variable 3
6 jsr 24        // Jump to the mini-subroutine for the finally clause
9 iload_3       // Push local variable 3 (the 1)
10 ireturn      // Return int on top of the stack (the 1)
.
.
24 astore_2     // Pop the return address, store it in local variable 2
25 getstatic #8 // Get a reference to java.lang.System.out
28 ldc #1       // Push <String "About to return."> from the constant pool
30 invokevirtual #7 // Invoke System.out.println()
33 ret 2        // Return to return address stored in local variable 2
```

Bug in Sun's JDK 1.1.4

```
1: jsr 10
2: store 1
3: jsr 10
4: store 2
5: load 2
6: init P
7: load 1
8: use P
9: halt
```

```
10: store 0
11: new P
12: ret 0
```

variables 1 and 2 contain references to two different objects which are both "uninitialized object created on line 11"

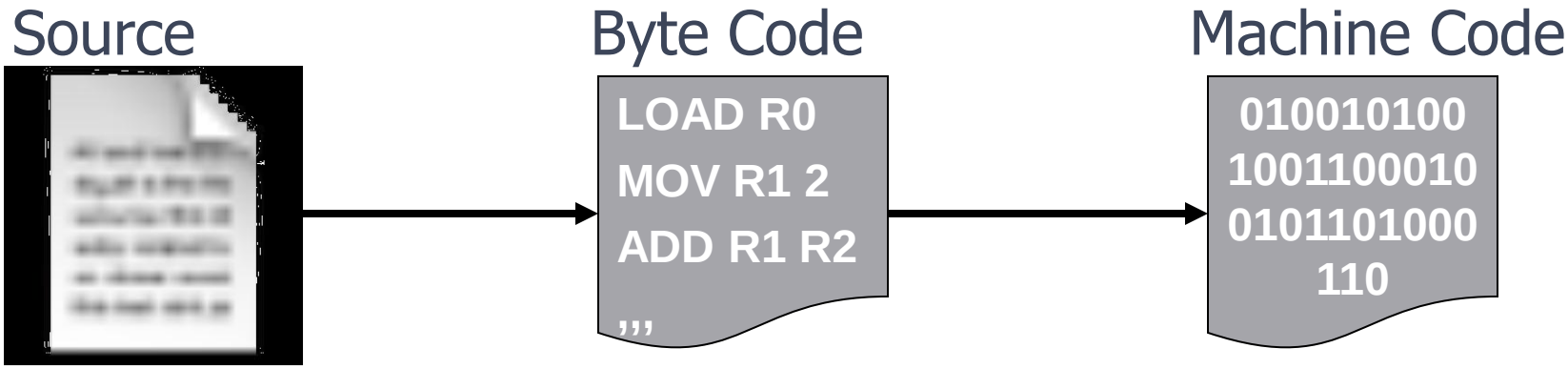
Bytecode verifier not designed for code that creates uninitialized object in jsr subroutine

Outline

- Java virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- JVM Method lookup
  - four different bytecodes
- Verifier analysis
- Method lookup optimizations (HotSpot VM, …)
- Java security
  - Buffer overflow
  - Java “sandbox”
  - Stack inspection

Avoid interpreting

JIT Compilation



Method can be executed as interpreted byte code
Compiled to machine code, initially or on nth execution
Compiled native code stored in cache

If cache fills, previously compiled method flushed

Lookup Cache

- Cache of recently used methods
  - indexed by (receiver impl-type, message name) pairs
- When a message is sent, compiler first consults cache
  - if found: invokes associated code
  - if absent: performs general lookup and potentially updates cache
- Berkeley Smalltalk would have been 37% slower without this optimization

Note: some researchers use "type" to refer to the method lookup table. This is different from "type" as object interface.

Static “Type” Prediction

- Compiler predicts “types” that are unknown but likely:
  - “Type” here means method lookup table, or “implementation type”
  - Arithmetic operations (+, -, <, *etc*.) have small integers as their receivers 95% of time in Smalltalk-80.
  - ifTrue had Boolean receiver 100% of the time.

- Compiler inlines code (and tests to confirm guess):

```
if impl-type = smallInt  jump to method_smallInt
                else call general_lookup
```

Inline Caches

- Track message-send from a *call site* :
  - general lookup routine invoked
  - call site back-patched
    - is previous method still correct?
      - yes: invoke code directly
      - no: proceed with general lookup & backpatch
- Successful about 95% of the time
- All compiled implementations of Smalltalk and Self use inline caches

Polymorphic Inline Caches

- Typical call site has <10 distinct receiver types
  - So often can cache *all* receivers
- At each call site, for each new receiver, extend patch code:

```
if impl-type = rectangle jump to method_rect
if impl-type = circle jump to method_circle
call general_lookup
```

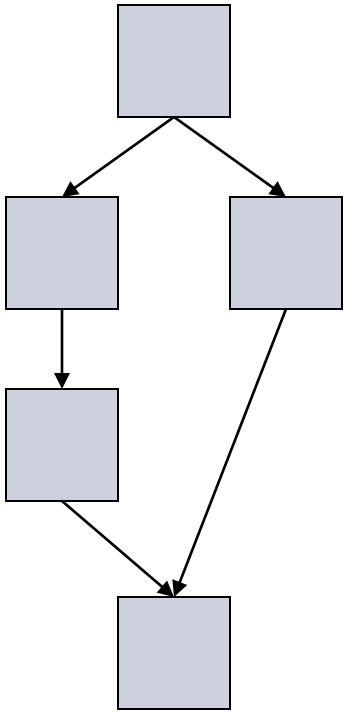
  - Order clauses by frequency
  - Inline short methods into PIC code
  - After some threshold, revert to simple inline cache

Customized Compilation

- Compile several copies of each method, one for each receiver type
- Within each copy:
  - Compiler knows the type of self
  - Calls through self can be statically selected and inlined
- Enables downstream optimizations
- Increases code size

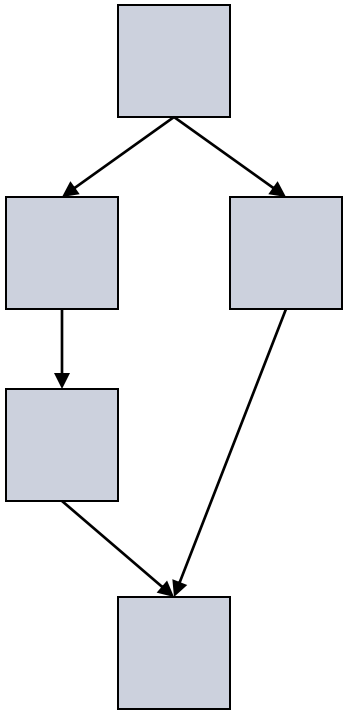
Implementation Type Analysis

- Constructed by compiler by flow analysis.
- “Type": set of possible vtables for object
  - Singleton: know vtable statically
  - Union/Merge: know expression has one of a fixed collection of vtables
  - Unknown: know nothing about expression
- If singleton, we can inline method
- If type set is small, we can insert type test and create branch for each possible receiver (type casing)



Message Splitting

- Type information above a merge point is often better
- Move message send “before” merge point:
  - duplicates code
  - improves type information
  - allows more inlining



PICS as Type Source

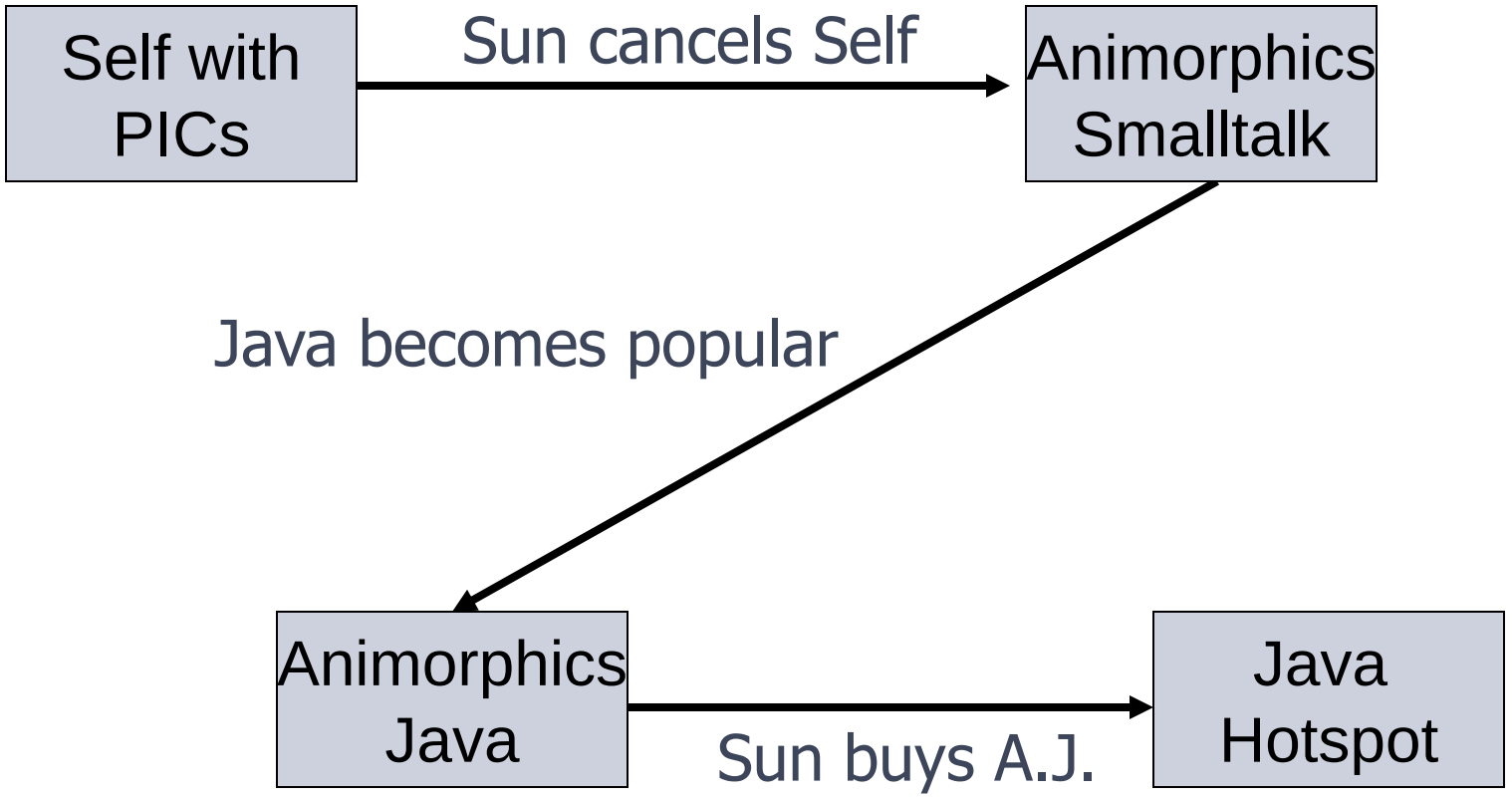
- Polymorphic inline caches build a call-site specific type database *as the program runs*
- Compiler can use this runtime information rather than the result of a static flow analysis to build type cases
- Must wait until PIC has collected information
  - When to recompile?
  - What should be recompiled?
- Initial fast compile yielding slow code; then dynamically recompile *hotspots*

Performance Improvements

- Initial version of Self was 4-5 times slower than optimized C.
- Adding type analysis and message splitting got within a factor of 2 of optimized C.
- Replacing type analysis with PICS improved performance by further 37%.

Best Self compiler was within a factor of 2 of optimized C
These techniques are commonly used for Java, JavaScript, …

Impact on Java



Outline

- Java virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- JVM Method lookup
  - four different bytecodes
- Verifier analysis
- Method lookup optimizations (beyond Java)
- Java security
  - Buffer overflow
  - Java “sandbox”
  - Stack inspection

Java Security

- Security
  - Prevent unauthorized use of computational resources
- Java security
  - Java code can read input from careless user or malicious attacker
  - Java code can be transmitted over network –

    code may be *written* by careless friend or malicious attacker

Java is designed to reduce many security risks

Java Security Mechanisms

- Sandboxing
  - Run program in restricted environment
    - Analogy: child's sandbox with only safe toys
  - This term refers to
    - Features of loader, verifier, interpreter that restrict program
    - Java Security Manager, a special object that acts as access control "gatekeeper"
- Code signing
  - Use cryptography to establish origin of class file
    - This info can be used by security manager

Buffer Overflow Attack

- Most prevalent *general* security problem today
  - Large number of CERT advisories are related to buffer overflow vulnerabilities in OS, other code
- General network-based attack
  - Attacker sends carefully designed network msgs
  - Input causes privileged program (e.g., Sendmail) to do something it was not designed to do
- Does not work in Java
  - Illustrates what Java was designed to prevent

Sample C code to illustrate attack

```
void f (char *str) {
   char buffer[16];
   …
   strcpy(buffer,str);
}
void main() {
   char large_string[256];
   int i;
   for( i = 0; i < 255; i++)
      large_string[i] = 'A';
   f(large_string);
}
```

- Function
  - Copies str into buffer until null character found
  - Could write past end of buffer, *over function retun addr*
- Calling program
  - Writes 'A' over f activation record
  - Function f "returns" to location 0x4141414141
  - This causes segmentation fault
- Variations
  - Put meaningful address in string
  - Put *code* in string and jump to it !!

See: Smashing the stack for fun and profit

Java Sandbox

- Four complementary mechanisms
  - Class loader
    - Separate namespaces for separate class loaders
    - Associates *protection domain* with each class
  - Verifier and JVM run-time tests
    - NO unchecked casts or other type errors, NO array overflow
    - Preserves private, protected visibility levels
  - Security Manager
    - Called by library functions to decide if request is allowed
    - Uses protection domain associated with code, user policy
    - Coming up in a few slides: stack inspection

Security Manager

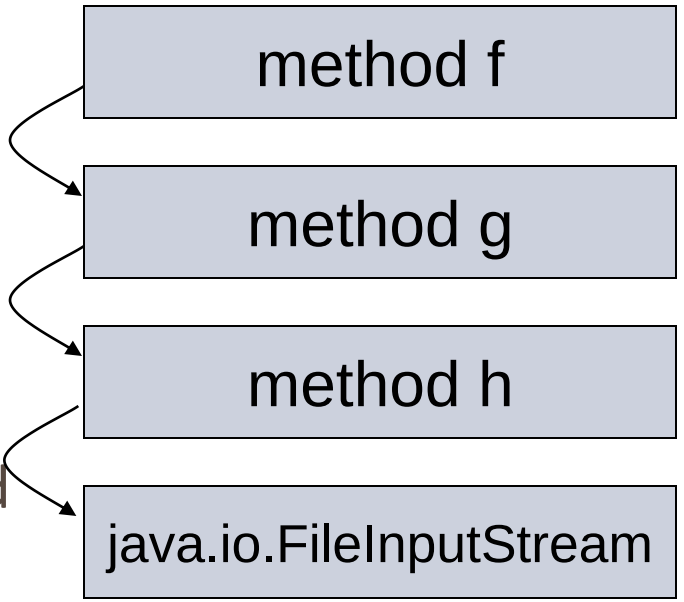
- Java library functions call security manager
- Security manager object answers at run time
  - Decide if calling code is allowed to do operation
  - Examine protection domain of calling class
    - Signer: organization that signed code before loading
    - Location: URL where the Java classes came from
  - Uses the system policy to decide access permission

Sample SecurityManager methods

checkExec	Checks if the system commands can be executed.
checkRead	Checks if a file can be read from.
checkWrite	Checks if a file can be written to.
checkListen	Checks if a certain network port can be listened to for connections.
checkConnect	Checks if a network connection can be created.
checkCreate ClassLoader	Check to prevent the installation of additional ClassLoaders.

Stack Inspection

- Permission depends on
  - Permission of calling method
  - Permission of all methods above it on stack
    - Up to method that is trusted and asserts this trust



Example: privileged printing

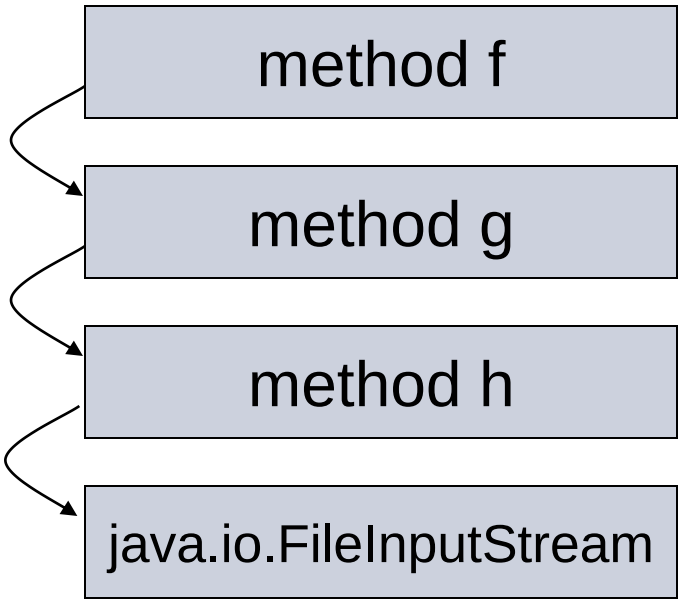
```
privPrint(f) =  (* owned by system *)
{
  checkPrivilege(PrintPriv);
  print(f);
}

foreignProg() = (* owned by Joe *)
{
  …; privPrint(file); …;
}
```

Stack Inspection

- Stack frames are annotated with names of owners and any enabled privileges
- During inspection, stack frames are searched from most to least recent:
  - fail if a frame belonging to someone not authorized for privilege is encountered
  - succeed if activated privilege is found in frame

Stack Inspection

- Permission depends on
  - Permission of calling method
  - Permission of all methods above it on stack
    - Up to method that is trusted and asserts this trust



Many details omitted here

Stories: Netscape font / passwd bug; Shockwave plug-in

Outline

- Java virtual machine overview
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- JVM Method lookup
  - four different bytecodes
- Verifier analysis
- Method lookup optimizations (beyond Java)
- Java security
  - Buffer overflow
  - Java “sandbox”
  - Stack inspection