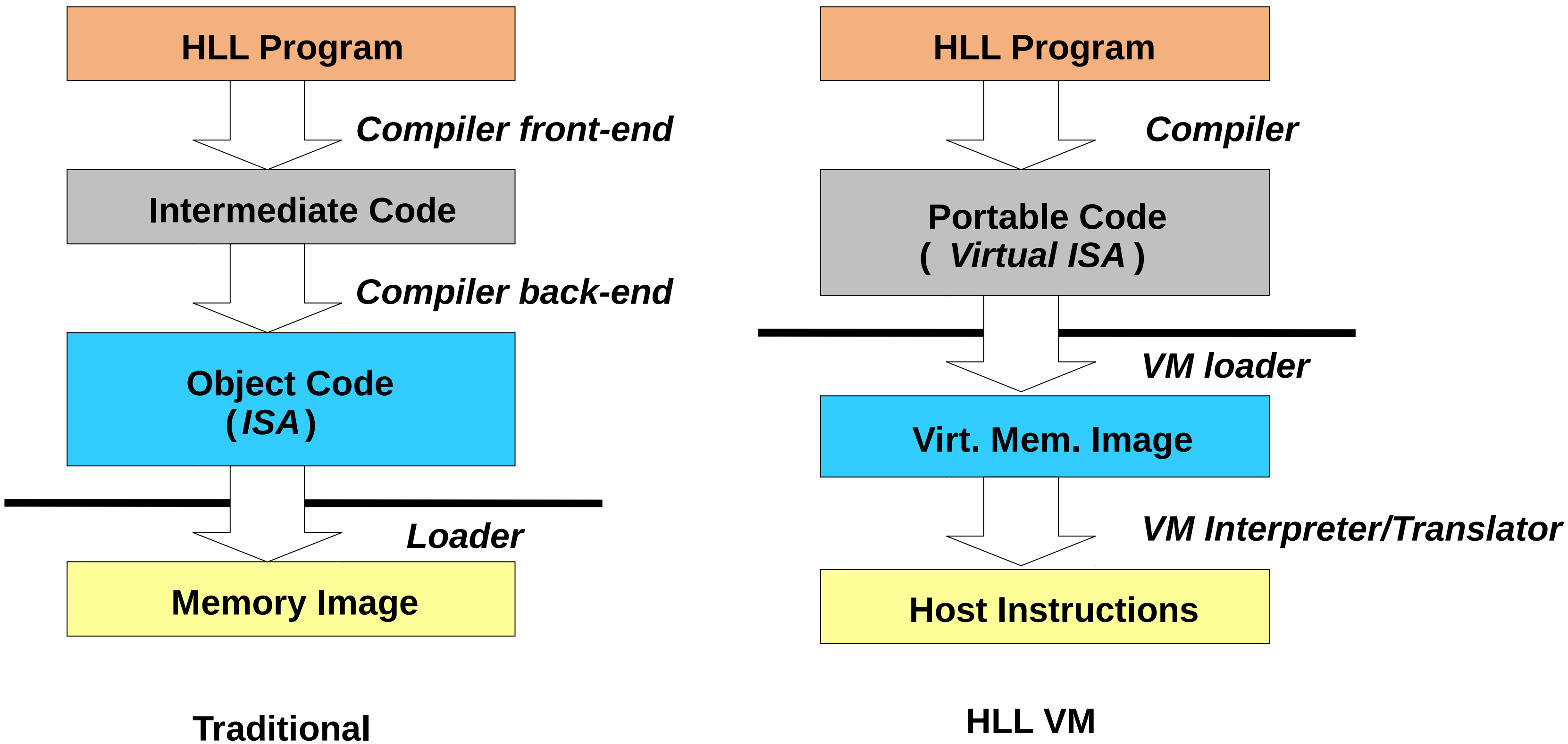
High-Level Language VM – Outline

- Introduction
- Virtualizing conventional ISA Vs. HLL VM ISA
- Pascal P-code virtual machine
- OO HLL virtual machines
  - properties, architecture, terms
- Implementation of HLL virtual machine
  - class loading, security, GC, JNI

Introduction

- HLL PVM similar to a *conventional* PVM
  – V-ISA not designed for a real hardware processor



Virtualizing Conventional ISA Vs. High-Level-Language VM ISA

- Drawbacks of virtualizing a conventional ISA
  - not developed for being virtualized!
  - operating system dependencies
  - issues with fixed-size address space, page-size
  - memory address formation
  - maintaining precise exceptions
  - instruction set features
  - instruction discovery during indirect jumps
  - self-modifying and self-referencing code

C-ISA Not for Being Virtualized

- Conventional ISA
  - after the fact solution for portability
  - no built-in ISA support for virtualization
- High-level language V-ISA
  - VM based portability is a primary design goal
  - generous use of *metadata*
  - metadata allows better type-safe code verification, interoperability, and performance

Operating System Dependencies

- Conventional ISA
  - most difficult to emulate
  - exact emulation may be impossible (different OS)
- High-level language V-ISA
  - find a least common denominator set of functions
  - programs interact with the library API
  - library interface is higher level than conventional OS interface

Memory Architecture

- Conventional ISA
  - fixed-size address spaces
  - specific addresses visible to user programs
- High-level language V-ISA
  - abstract memory model of indefinite size
  - memory *regions* allocated based on need
  - actual memory addresses are never visible
  - *out-of-memory* error reported if process requests more that is available of platform

Memory Address Formation

- Conventional ISA
  - unrestricted address computation
  - difficult to protect runtime from un-authorized guest program accesses
- High-level-language V-ISA
  - pointer arithmetic not permitted
  - memory access only through explicit memory pointers
  - static/dynamic type checking employed

Precise Exceptions

- Conventional ISA
  - many instructions trap, precise state needed
  - *global* flags enable/disable exceptions
- High-level language V-ISA
  - few instructions trap
  - test for exception encoded in the program
  - requirements for precise exceptions are relaxed

Instruction Set Features

- Conventional ISA
  - guest ISA registers > host registers is a problem
  - ISAs with condition codes are difficult to emulate
- High-level language V-ISA
  - stack-oriented
  - condition codes are avoided

Instruction Discovery

- Conventional ISA
  - indirect jumps to potentially arbitrary locations
  - variable-length instruction, embedded data, padding
- High-level-language V-ISA
  - restricted indirect jumps
  - no mixing of code and data
  - variable-length instructions permitted

Self-Modifying/Referencing Code

- Conventional ISA
  - pose problems for translated code
- High-level language V-ISA
  - self-modifying and self-referencing code not permitted

Pascal P-code

- Popularized the Pascal language
  - simplified porting of a Pascal *compiler*
- Introduced several concepts used in HLL VMs
  - stack-based instruction set
  - memory architecture is implementation independent
  - undefined stack and heap sizes
  - standard libraries used to interface with the OS
- Objective was compiler portability (and application portability)

Pascal P-Code (2)

- Protection via trusted interpreter.
- Advantages
  - porting is simplified
    - don't have to develop compilers for all platforms
  - VM implementation is smaller/simpler than a compiler
  - VM provides concise definition of semantics
- Disadvantages
  - achieving OS independence reduces API functionality to least common denominator
  - tendency to add platform-specific API extensions

Object Oriented HLL Virtual Machines

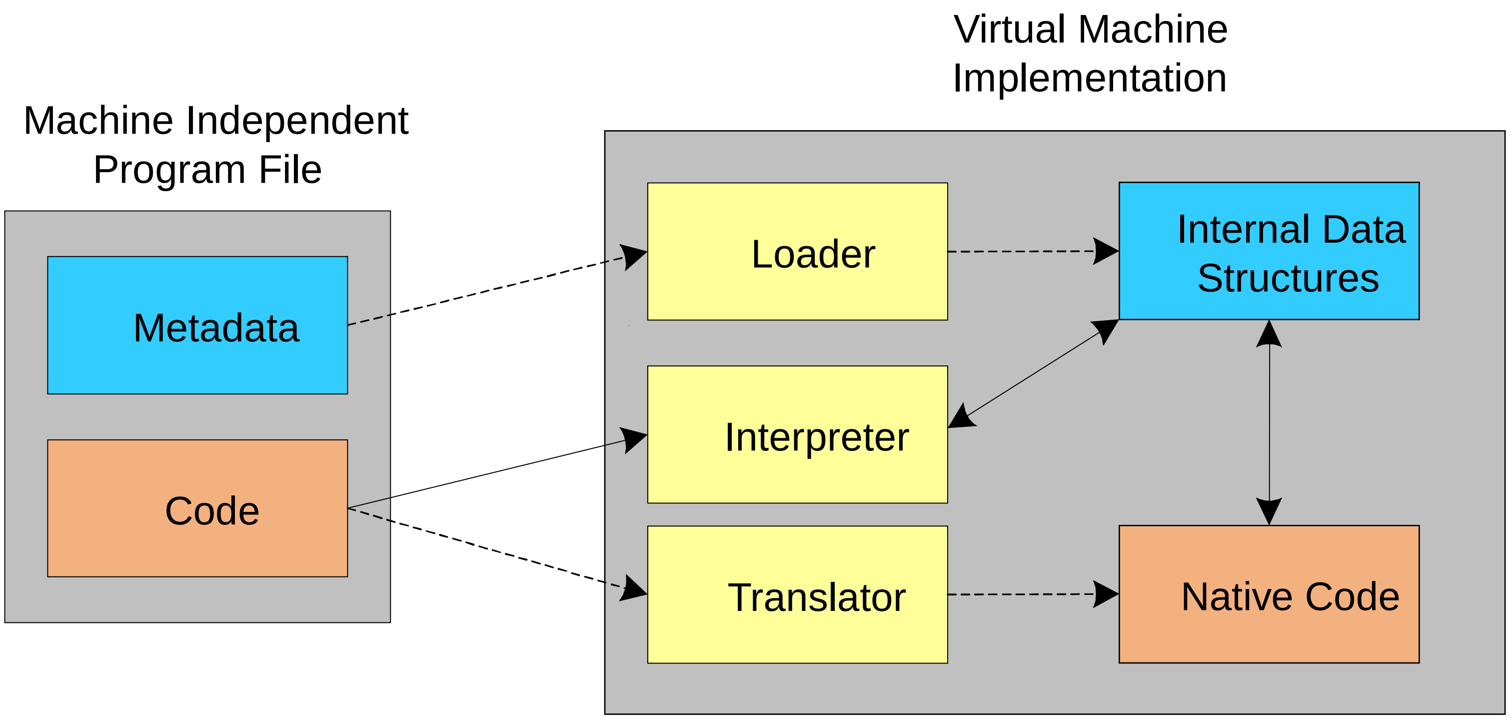
- Used in a networked computing environment
- Important features of HLL VMs
  - security and protection
    - protect remote resources, local files, VM runtime
  - robustness
    - OOP model provides component-based programming, strong type-checking, and garbage collection
  - networking
    - incremental loading, and small code-size
  - performance
    - easy code discovery allows entire method compilation

Terminology

- Java Virtual Machine Architecture □ CLI
  - analogous to an ISA
- Java Virtual Machine Implementation □CLR
  - analogous to a computer implementation
- Java bytecodes □ Microsoft Intermediate Language (MSIL), CIL, IL
  - the instruction part of the ISA
- Java Platform □ .NET framework
  - ISA + Libraries; a higher level ABI

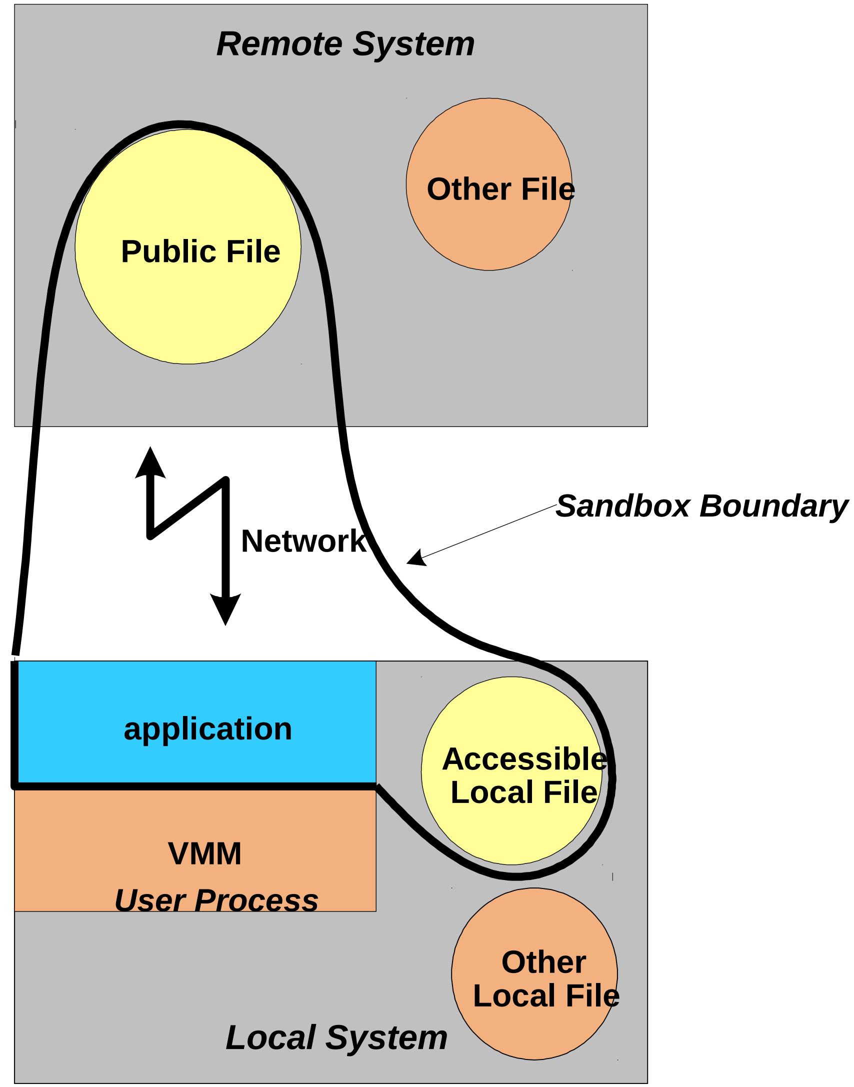
Modern HLL VM

- Compiler frontend produces binary files
  - – standard format common to all architectures
- Binary files contain both code and metadata



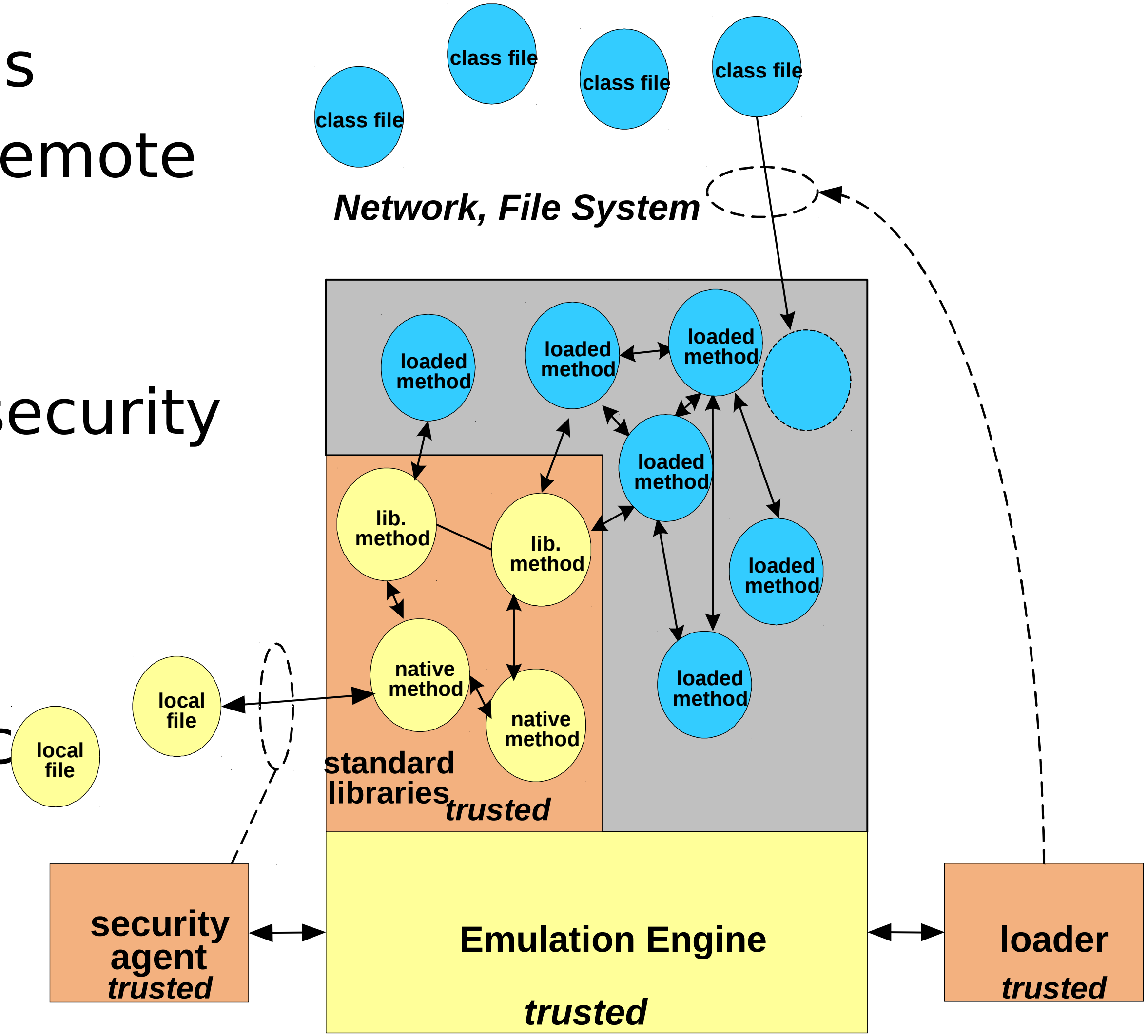
Security

- A key aspect of modern network-oriented Vms
  - "protection sandbox"
- Must protect:
  - remote resources (files)
  - local files
  - runtime
- Java's first generation security method
  - still the default



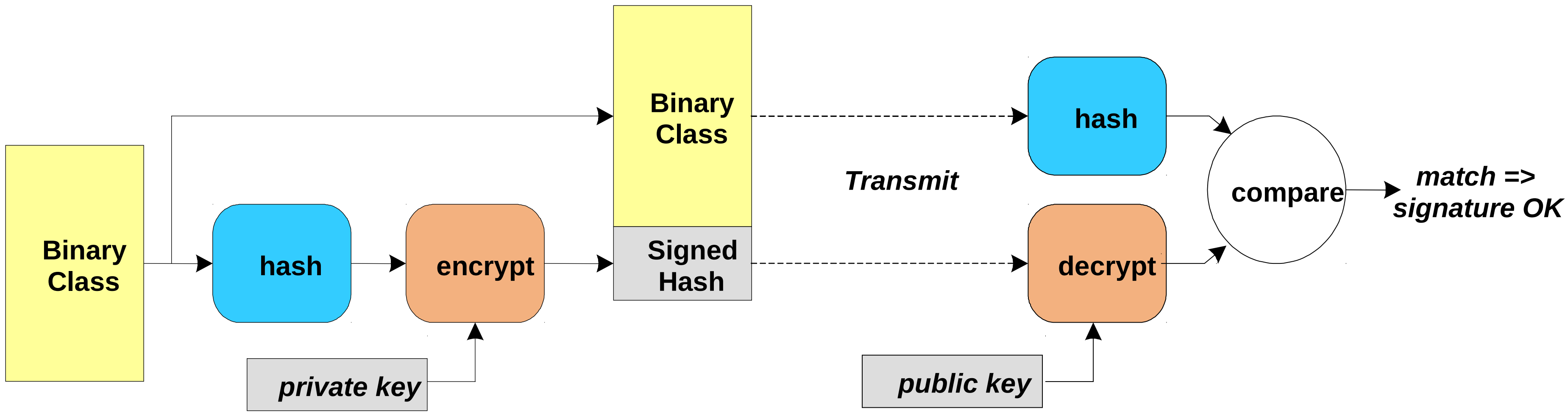
Protection Sandbox

- Remote resources
  - protected by remote system
- Local resources
  - protected by security manager
- VM software
  - protected via static/dynamic checking



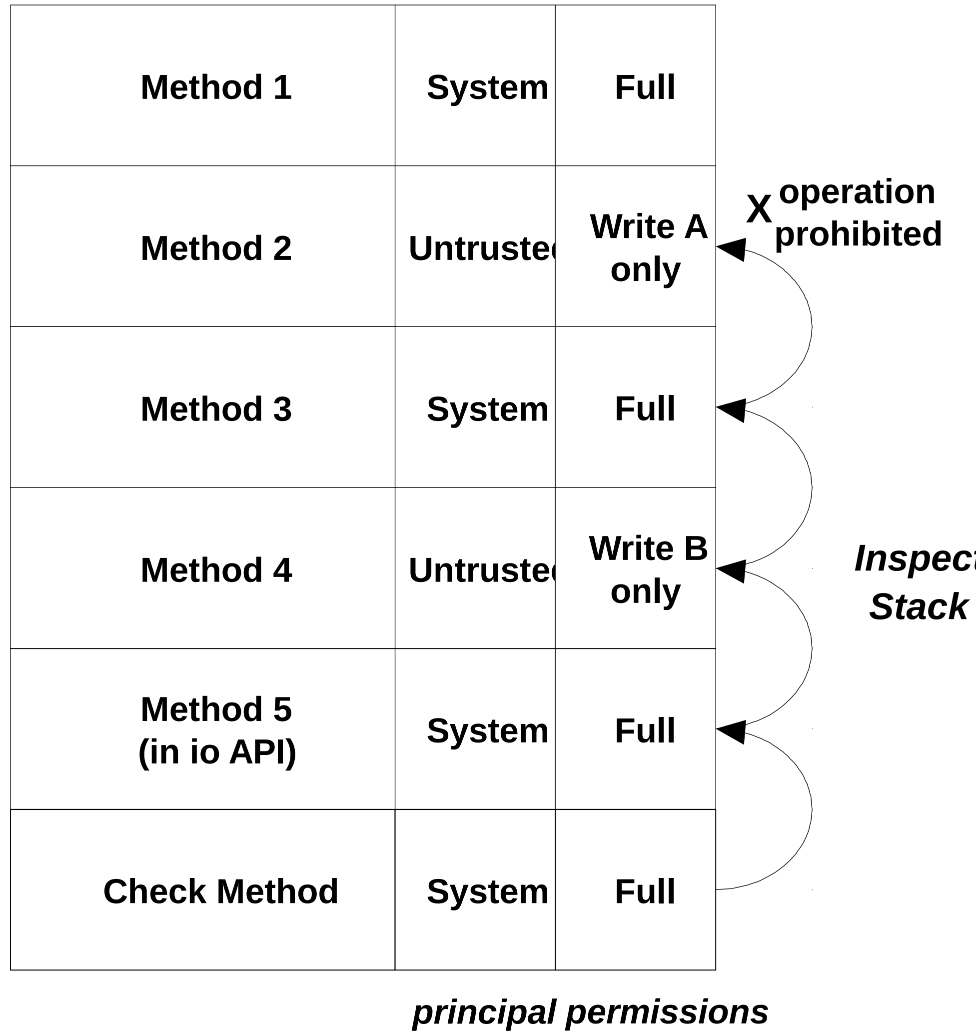
Java 1.1 Security: Signing

- Identifies source of the input program
  - can implement different security policies for programs from different vendors



Java 2 Security: Stack Walking

- Inspect privileges of all methods on stack
  - append method permissions
  - method 4 attempts to write file B via io.method5
  - call fails since method2 does not have privileges



Garbage Collection

- Issues with traditional *malloc/free, new/delete*
  - explicit memory allocation places burden on programmer
  - dangling pointer, double free errors
- Garbage collection
  - objects with no references are garbage
  - must be collected to free up memory
    - for future object allocation
    - OS limits memory use by a process
  - eliminates programmer pointer errors

Network Friendliness

- Support dynamic class loading on demand
  - load classes only when needed
  - spread loading over time
- Compact instruction encoding
  - zero-address stack-based bytecode to reduce code size
  - contain significant *metadata*
    - maybe a slight code size win over RISC fixed-width ISAs

Java ISA

- Formalized in *classfile* specification.
- Includes instruction definitions (*bytecodes*).
- Includes data definitions and interrelationships (*metadata*).

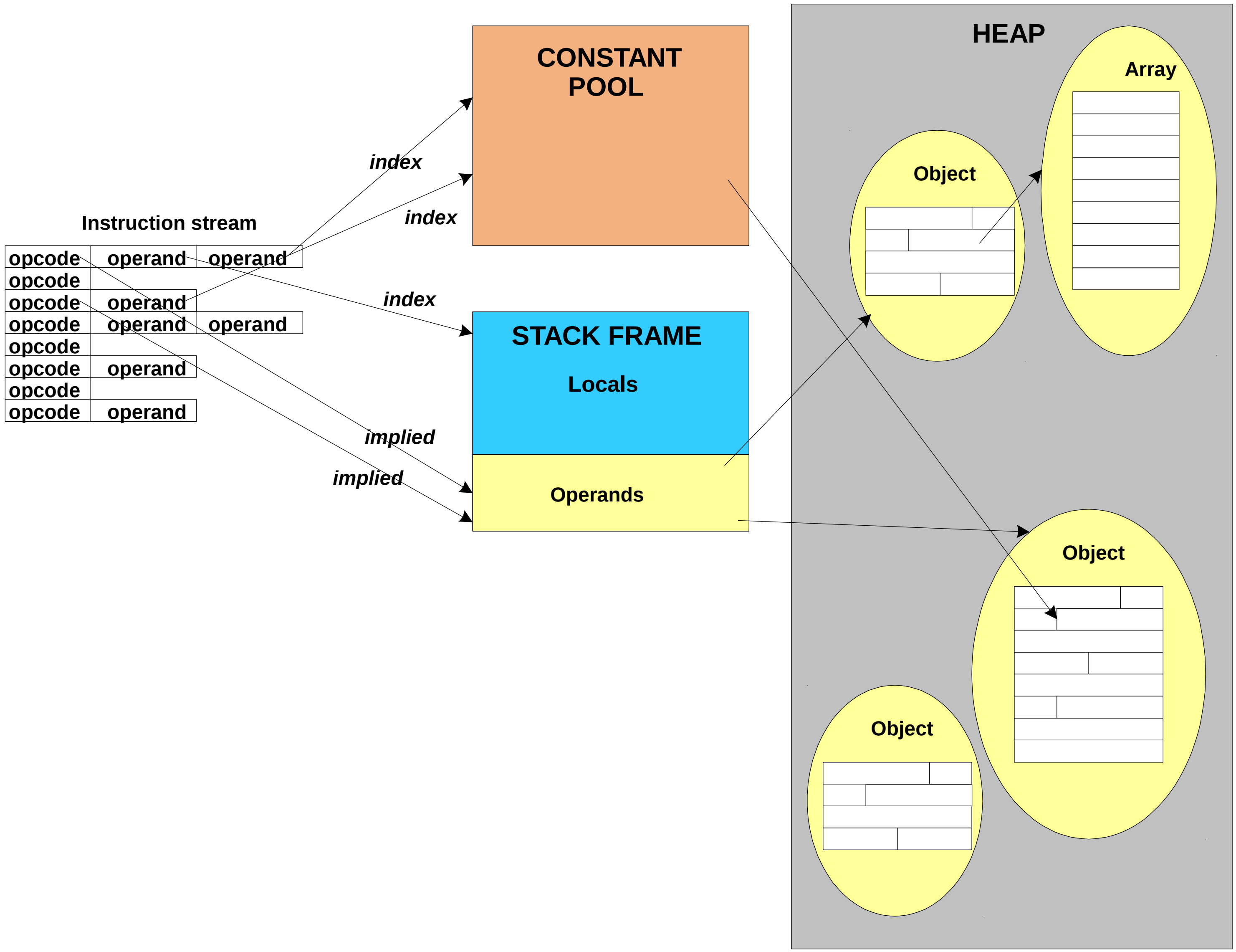
Java Architected State

- Implied registers
  - program counter, local variable pointer, operand stack pointer, current frame pointer, constant pool base
- Stack
  - arguments, locals, and operands
- Heap
  - objects and arrays
  - implementation-dependent object representation
- Class file content
  - constant pool holds immediates (and other constant information)

Data Items

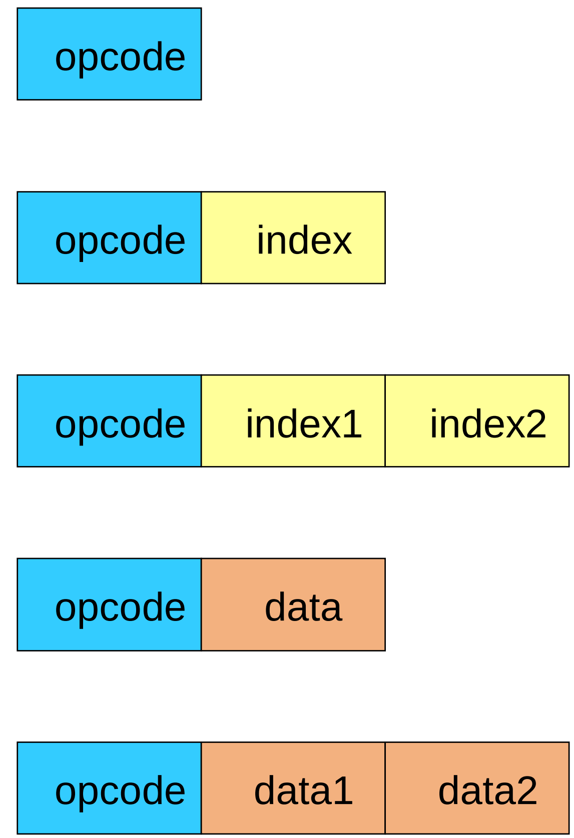
- Types are defined in specification
  - implementation free to choose representation
  - reference (pointers) and primitive (byte, int, etc.) types
- Range of values that can be held are given
  - e.g., byte is between -127 and +128
  - data is located via
    - references; as fields of objects in heap
    - offsets using constant pool pointer, stack pointer

Data Accessing



Instruction Set

- Bytecodes
  - single byte opcode
  - zero or more operands
- Can access operands from
  - instruction
  - current constant pool
  - current frame local variables
  - values on operand stack



Instruction Types

- Pushing constants onto the stack
- Moving local variable contents to and from the stack
- Managing arrays
- Generic stack instructions (dup, swap, pop & nop)
- Arithmetic and logical instructions
- Conversion instructions
- Control transfer and function return
- Manipulating object fields
- Method invocation
- Miscellaneous operations
- Monitors

Stack Tracking

- At any point in program operand stack has
  - same number of operands
  - of same types
  - and in same order
  - *regardless of the control path getting there !*
- Helps with static type checking

Stack Tracking – Example

- Valid bytecode sequence:

```
                iload      A          //push int. A from local mem.
                iload      B          //push int. B from local mem.
                If_cmpne 0  else      // branch if B ne 0
                iload      C          // push int. C from local mem.
                goto       endelse
      else:     iload      F          //push F
   endelse:     add                   // add from stack; result to stack
                istore     D          // pop sum to D
```

Stack Tracking – Example

- Invalid bytecode sequence
  - stack at *skip1* depends on control-flow path

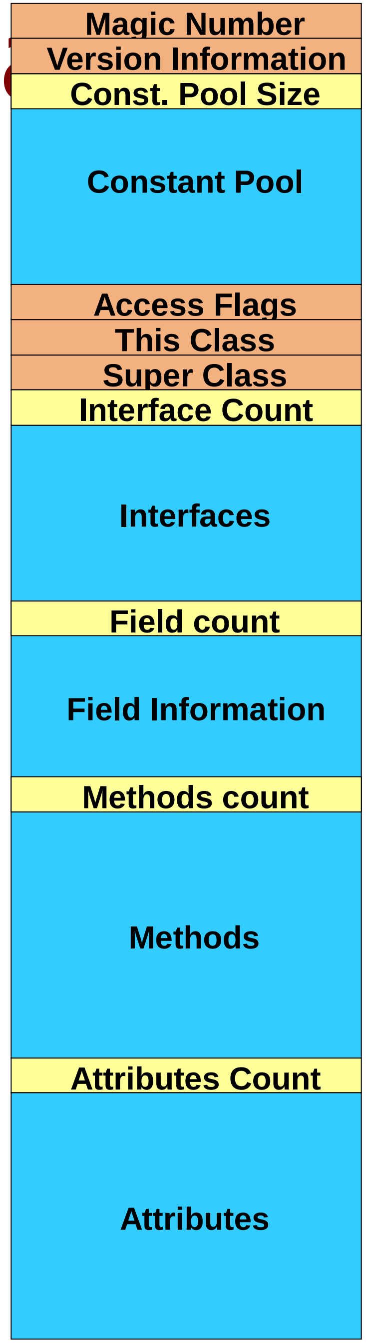
```
        iload     B              // push int. B from local mem.
        If_cmpne 0  skip1        // branch if B ne 0
        iload     C              // push int. C from local mem.
skip1:  iload     D              // push D
        iload     E              // push E
        if_cmpne 0  skip2        // branch if E ne 0
        add                      // add stack; result to stack
skip2:  istore    F              // pop to F
```

Exception Table

- Exceptions identified by table in class file
  - address Range where checking is in effect
  - target if exception is thrown
    - operand stack is emptied
- If no table entry in current method
  - pop stack frame and check calling method
  - default handlers at main

From	To	Target	Type
8	12	96	Arithmetic Exception

Binary Class Forma



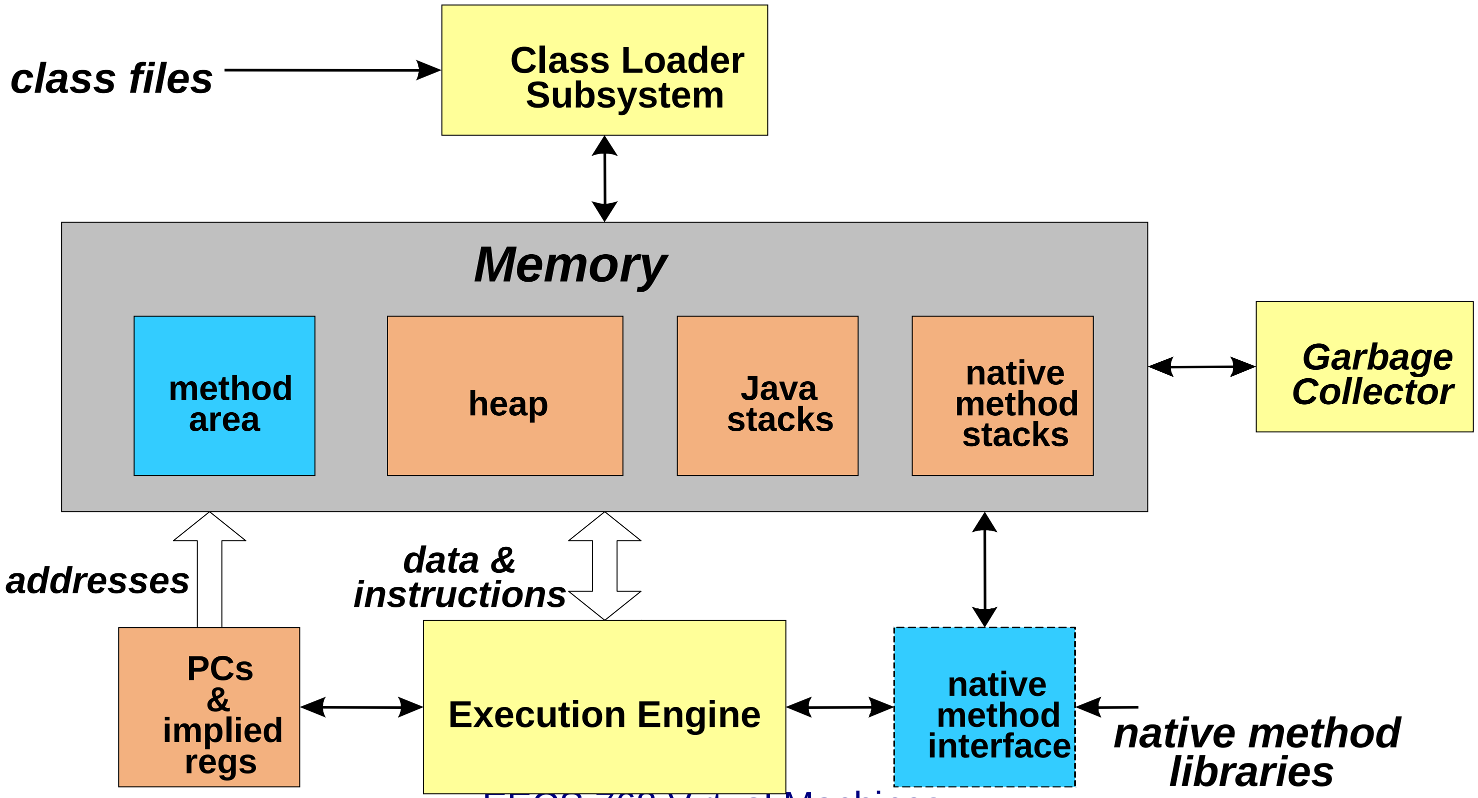
- Magic number and header
- Regions preceded by counts
  - constant pool
  - interfaces
  - field information
  - methods
  - attributes

Java Virtual Machine

- Abstract entity that gives meaning to class files
- Has many concrete implementations
  - hardware
  - interpreter
  - JIT compiler
- Persistence
  - an instance is created when an application starts
  - terminates when the application finishes

JVM Implementation

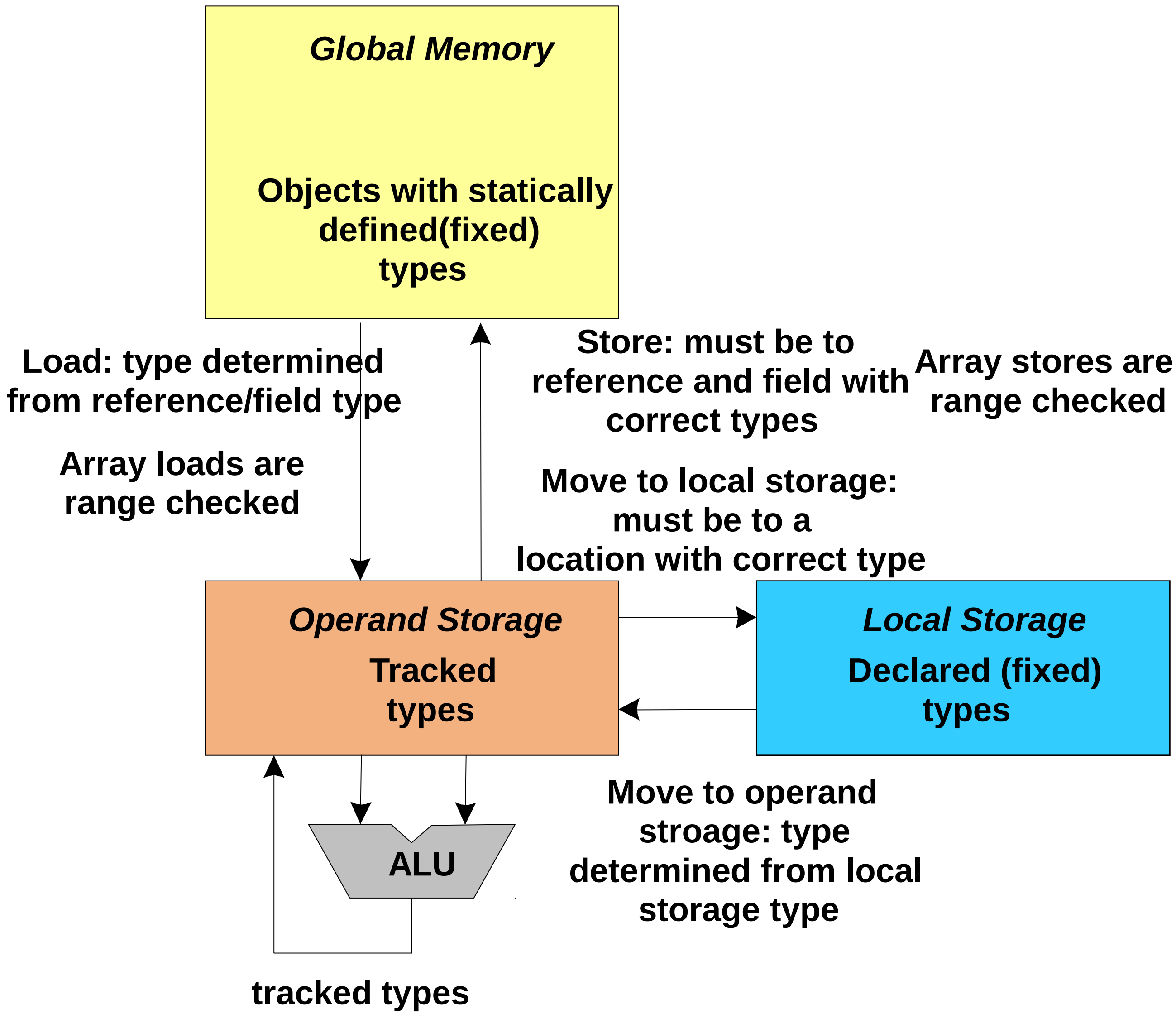
- A typical JVM implementation consists of
  - class loader subsystem , memory subsystem, emulation/execution engine, garbage collector



Class Loader

- Functions
  - find the binary class
  - convert class data into implementation-dependent memory image
  - verify correctness and consistency of the loaded classes
- Security checks
  - checks class magic number
  - component sizes are as indicated in class file
  - checks number/types of arguments
  - verify integrity of the bytecode program

Protection Sandbox



Protection Sandbox: Security Manager

- A trusted class containing check methods
  - attached when Java program starts
  - cannot be removed or changed
- User specifies checks to be made
  - files, types of access, etc.
- Operation
  - native methods that involve resource accesses (e.g. I/O) first call check method(s)

Verification

- Class files are checked when loaded
  - to ensure security and protection
- Internal Checks
  - checks for magic number
  - checks for truncation or extra bytes
    - each component specifies a length
  - make sure components are well-formed

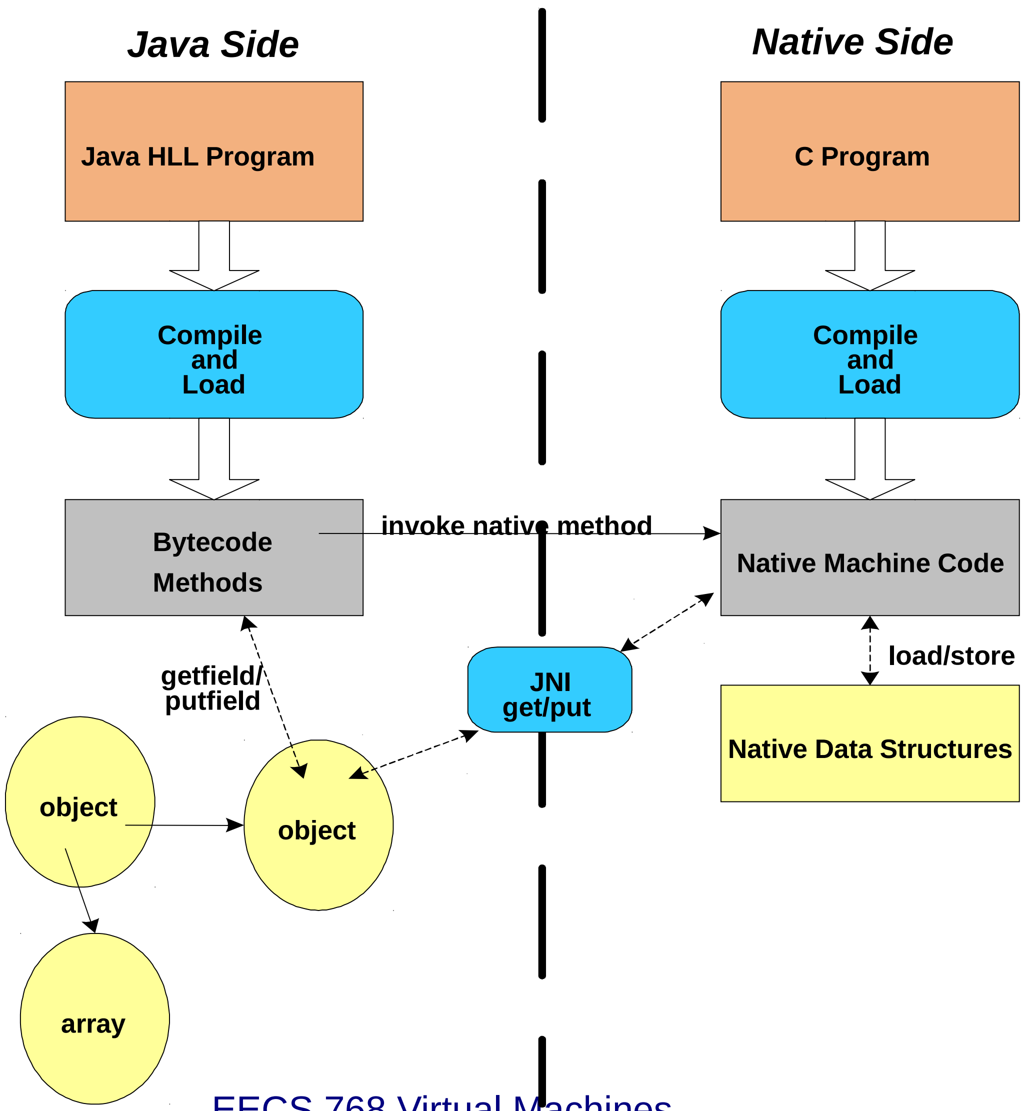
Verification (2)

- Bytecode checks
  - check valid opcodes
  - perform full path analysis
    - regardless of path to an instruction contents of operand stack must have same number and types of items
    - checks arguments of each bytecode
    - check no local variables are accessed before assigned
    - makes sure fields are assigned values of proper type

Java Native Interface (JNI)

- Allows java code and native code to interoperate
  - access legacy code, system calls from Java
  - access Java API from native functions
- see figure on next slide
  - each side compiles to its own binary format
  - different java and native stacks maintained
  - arguments can be passed; values/exceptions returned

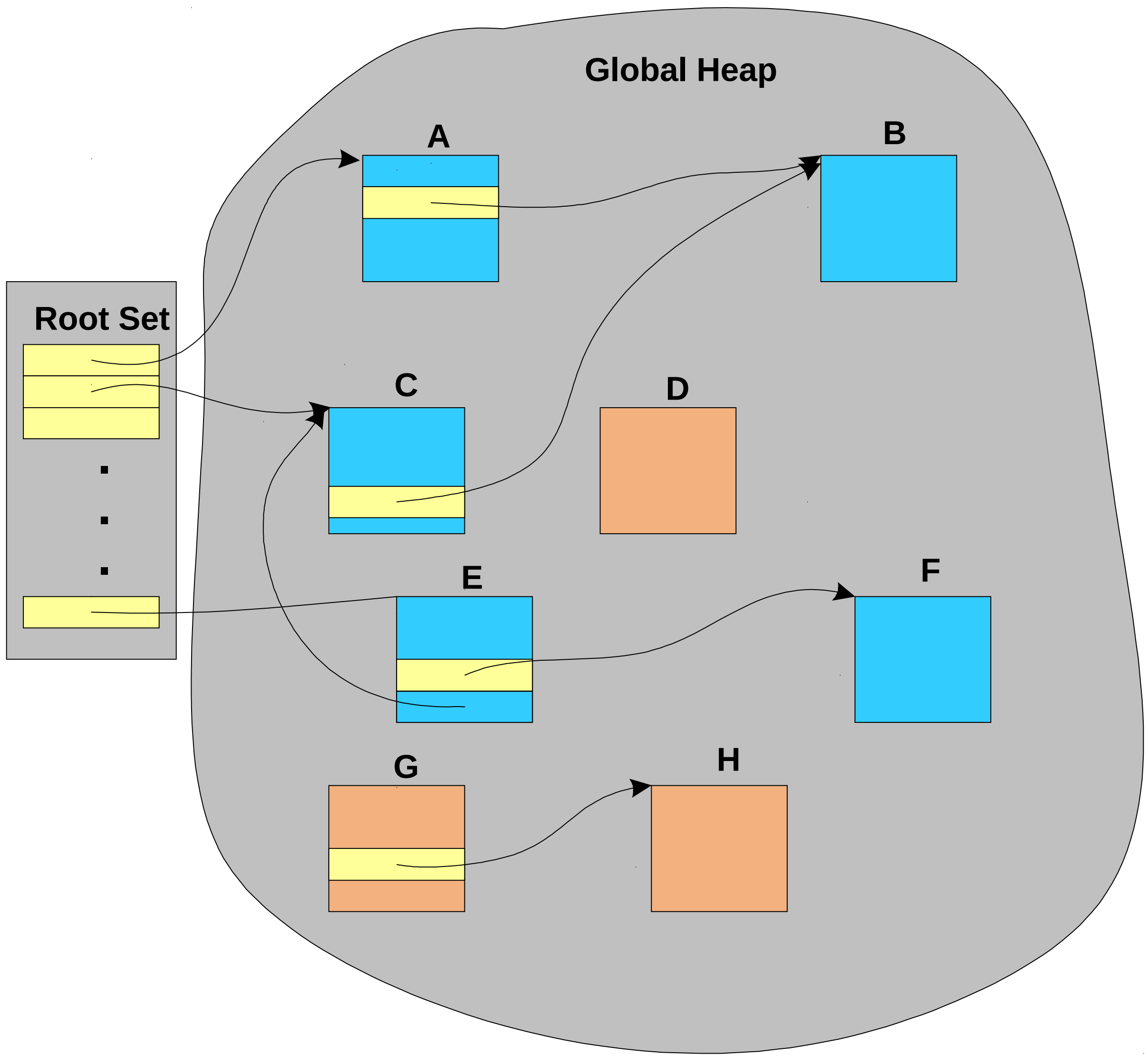
Java Native Interface (JNI)



Garbage Collector

- Provides implicit heap object space reclamation policy.
- Collects objects that have all their references removed or destroyed.
- Invoked at regular intervals, or when low on memory.
- see figure on next slide
  - root set point to objects in heap
  - objects not reachable from root set are garbage

Garbage Collector (2)



Types of Collectors

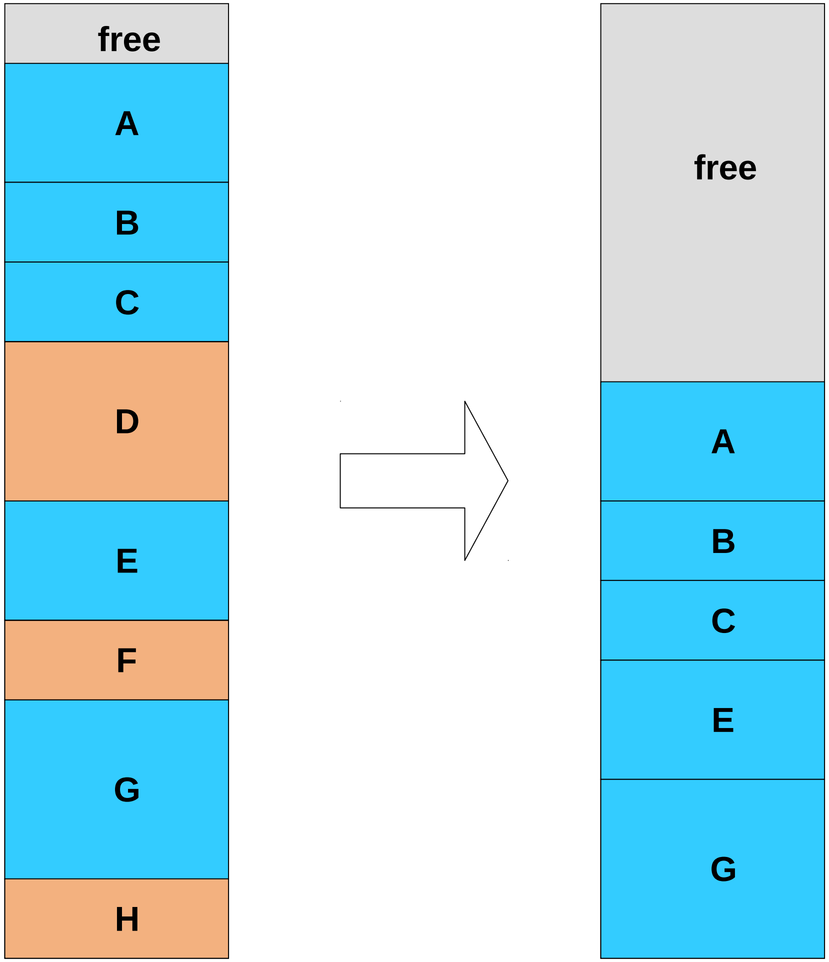
- Reference count collectors
  - keep a count of the number of references to each object
- Tracing collectors
  - using the root set of references

Mark and Sweep Collector

- Basic tracing collector
  - start with *root* set of references
  - trace and mark all reachable objects
  - sweep through heap collecting marked objects
- Advantages
  - does not require moving object/pointers
- Disadvantages
  - garbage objects combined into a linked list
    - leads to fragmentation
    - segregated free-lists can be used
    - consolidation of free space can improve efficiency

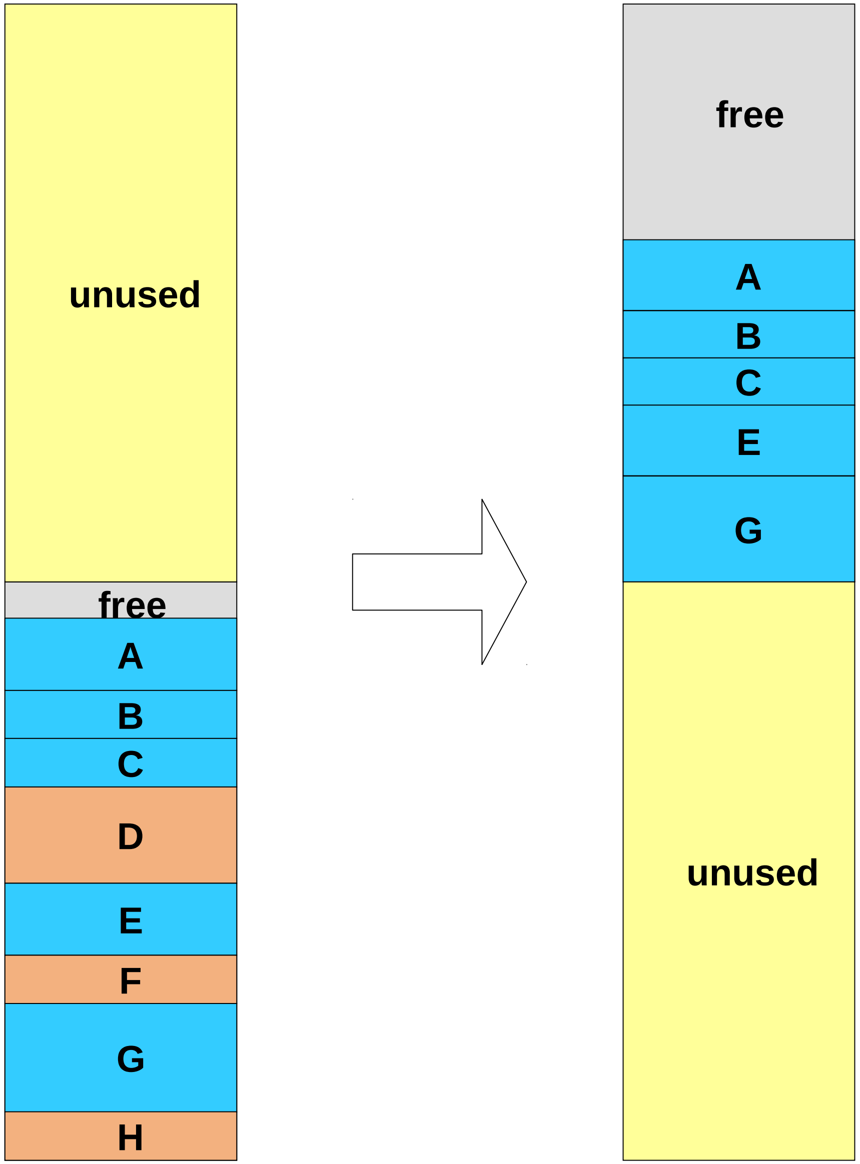
Compacting Collector

- Make free space contiguous
  - multiple passes through heap
  - lot of object movement
    - many pointer updates



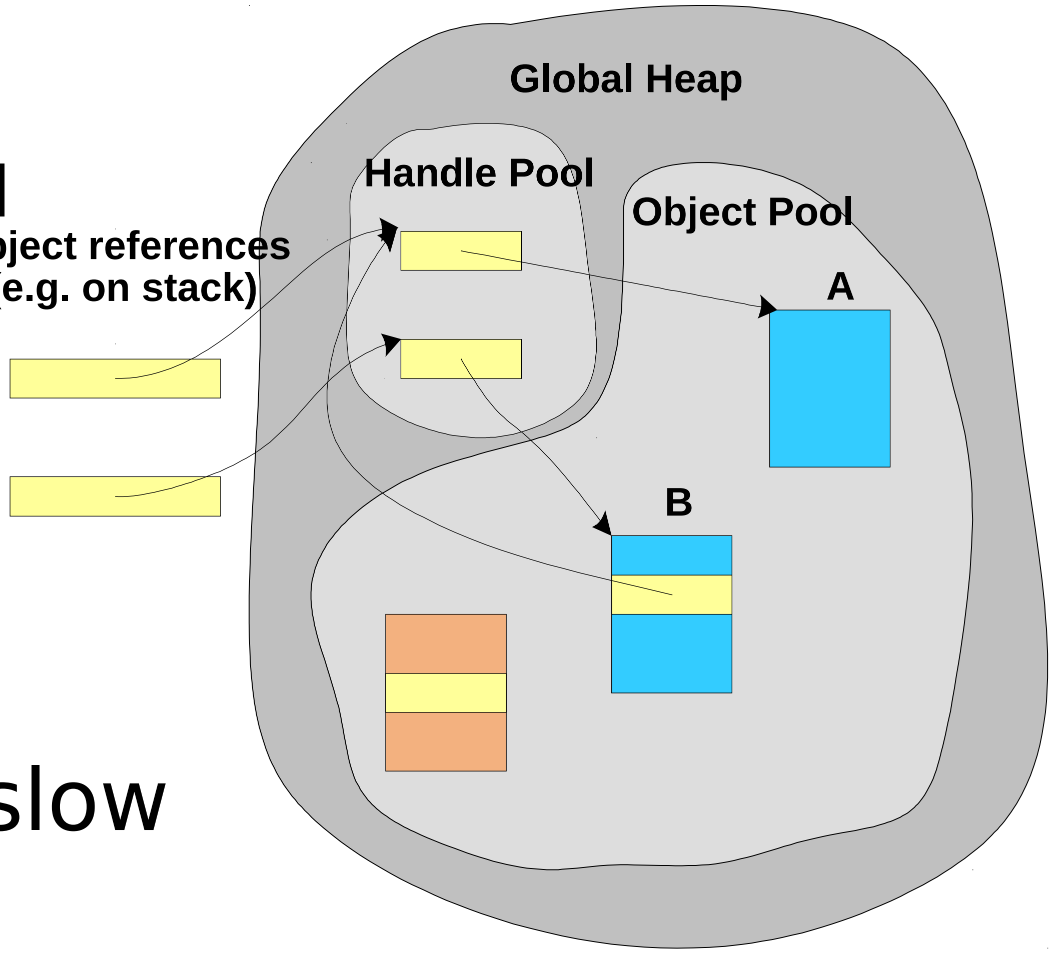
Copying Collector

- Divide heap into halves
  - collect when one half full
  - copy into unused half during sweep phase
- Reduces passes through heap
- *Wastes* half the heap



Simplifying Pointer Updates

- Add level of indirection
  - use handle pool
  - object moves update handle pool
- Makes every object access slow



Generational Collectors

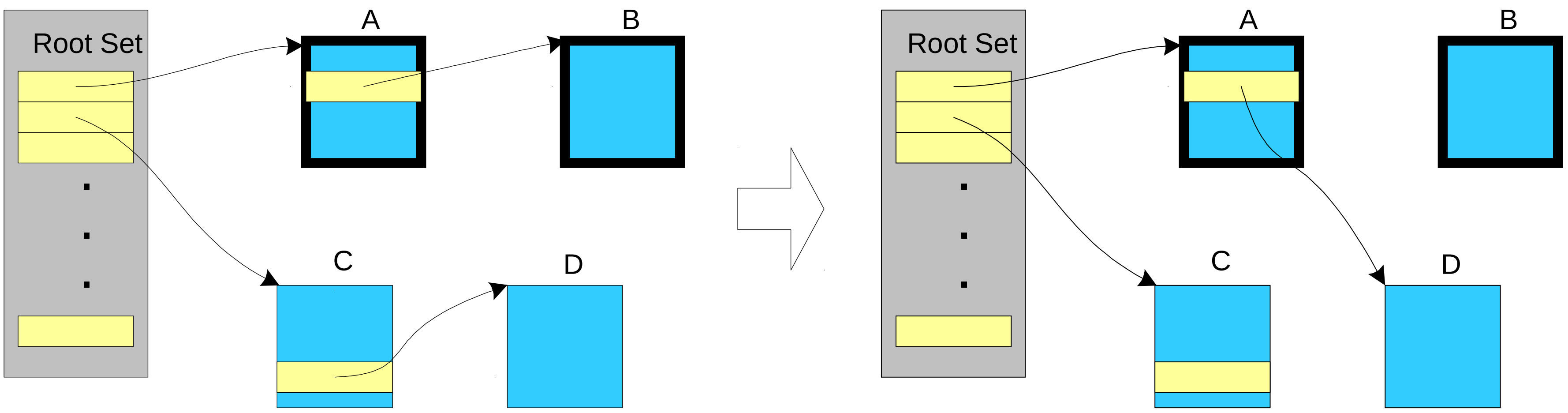
- Reduce number of objects moved during each collection cycle.
- Exploit the bi-modal distribution of object lifetimes.
- Divide heap into two sub-heaps
  - *nursery,* for newly created objects
  - *tenured,* for older objects
- Collect a smaller portion of the heap each time.

Generational Collectors (2)

- *Stop-the-world* collectors
  - time consuming, long pauses
  - unsuitable for real-time applications

Concurrent Collectors (2)

- GC concurrently with application execution
  - partially collected heap may be unstable (see figure)
  - synchronization needed between the application (mutator) and the collector



JVM Bytecode Emulation

- Interpretation
  - simple, fast startup, slow steady-state
- Just-In-Time (JIT) compilation
  - compile each method on first invocation
  - simple optimizations, slow startup, fast steady-state
- Hot-spot compilation
  - compile frequently executed code
  - can apply more aggressive optimizations
  - moderate startup, fast steady-state