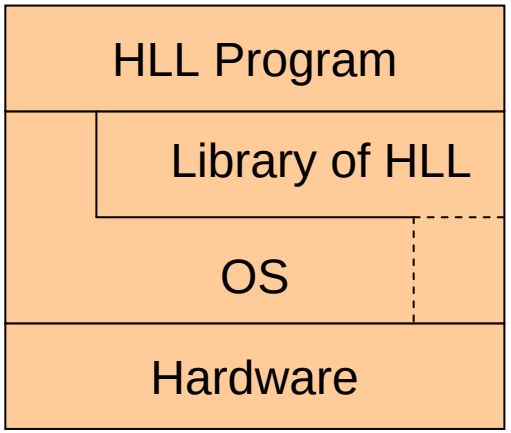
High-Level Language Virtual Machine Architecture

Contents

- Introduction of HLL VM
- The Pascal P-code VM
- Object-Oriented HLL VM
- Java VM Architecture

HLL program characteristics

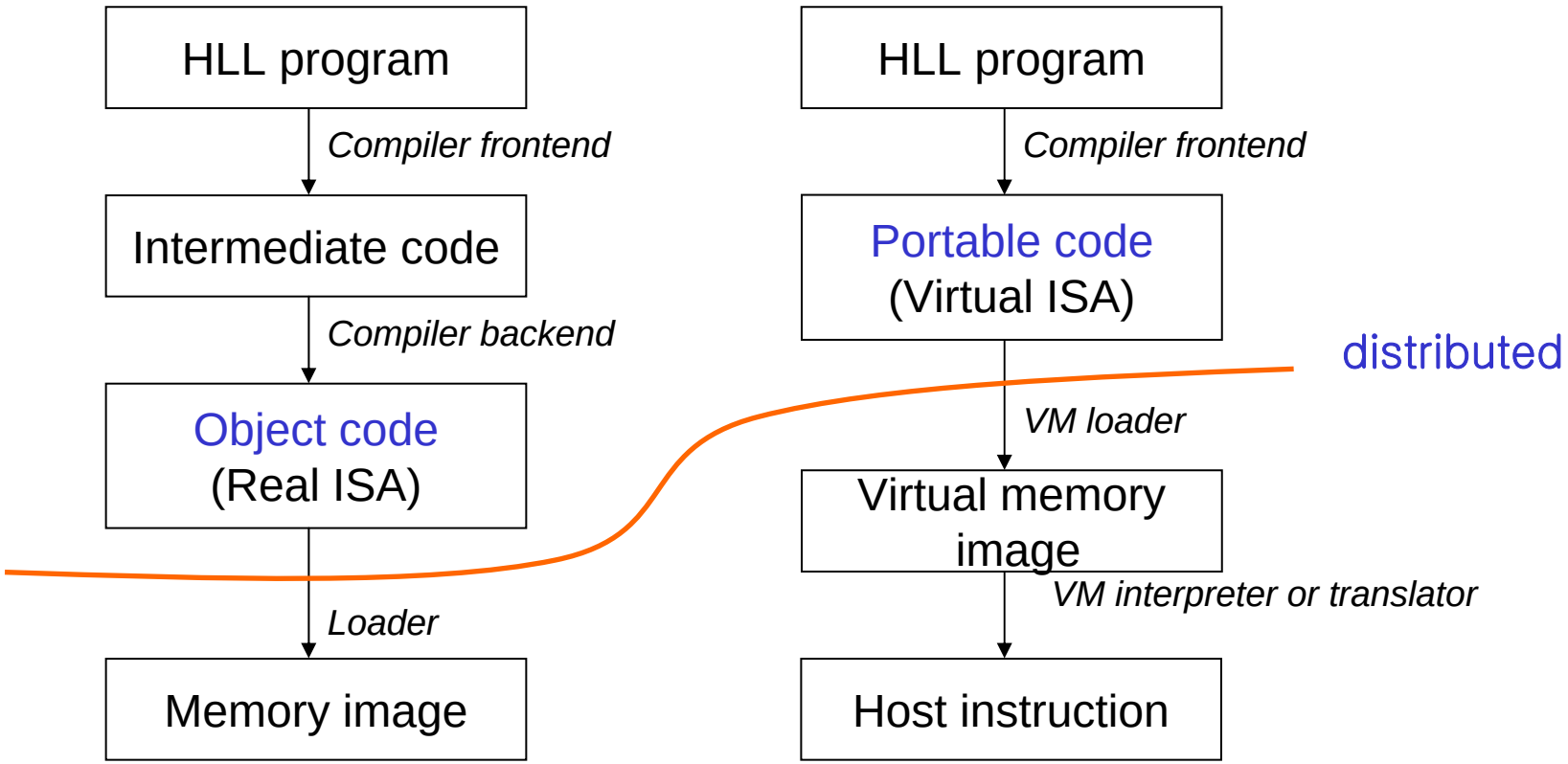
- Platform dependence
  - Library, OS system call, ISA
- Porting to another platform,
  - At least re-compile
  - Rewrite build environment
    - compiler, library, assembler, linker, etc.
  - Modify source code (e.g., system call)
  - Painful for S/W venders
- Employ process VM to avoid porting
  - Emulating application's ISA and OS calls
  - Difficult to emulate some OS interfaces completely
  - High-performance is hard to achieve



HLL VM

- New design of ISA/interface with VM-based portability
  - Generic ISA free of implementation-specific features
  - Abstracted system interface easily supported by OS
  - Support for target HLL such as OO languages
  - Compared to process VM,
    - Supports virtual ISA (V-ISA)
    - Interface based on standard libraries (API) for network, file, graphics
  - V-ISA includes data specification as well as instruction set
    - Important for platform independency
    - *data set architecture* than *instruction set architecture*

HLL VMs from Language Perspectives

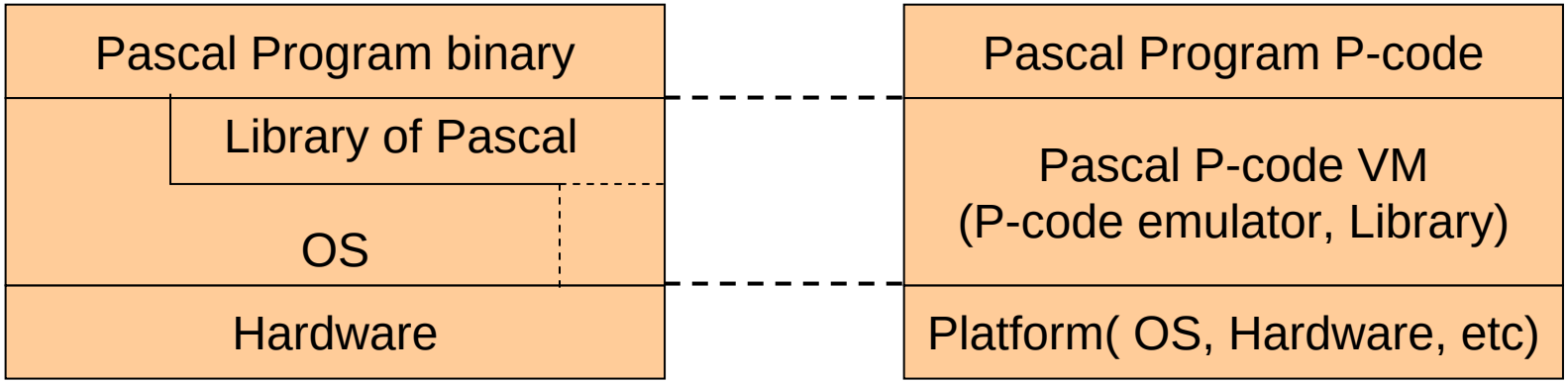


HLL VM Examples

- Pascal VM with a V-ISA called P-code
- Java VM with a V-ISA called bytecode
- Common language infrastructure (CLI) in .NET platform with a V-ISA called MSIL

The Pascal P-Code VM

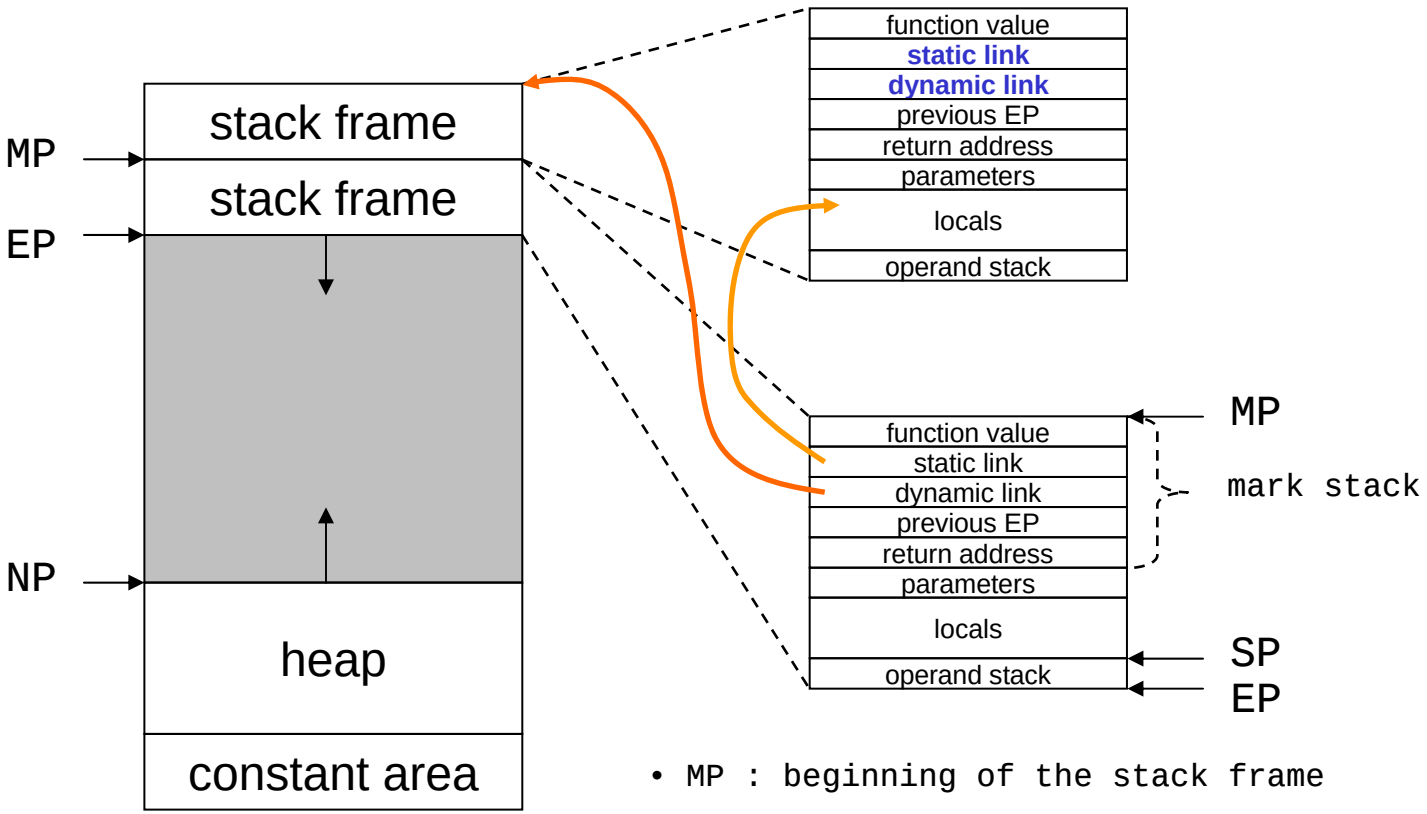
- P-code is a simple, stack-based ISA
- Only one Pascal compiler needs to be developed
  - Distributed as a P-code version
- Porting Pascal is implementing the VM only
  - Much simpler than writing a Pascal compiler for the platform



Memory Architecture

- Program memory, constant area, stack, and heap
  - All data areas are divided into *cells,*
    - Each cell can hold a single value
    - Actual size of cell is implementation dependent but large enough
  - Constant area (immediate operands)
  - A stack
    - Procedure stack
    - Operand stack for instruction execution
  - No garbage collection

Memory Architecture



- MP : beginning of the stack frame
- EP : maximum extent of current frame
- NP : maximum extent of the heap
- SP : current top of the operand stack

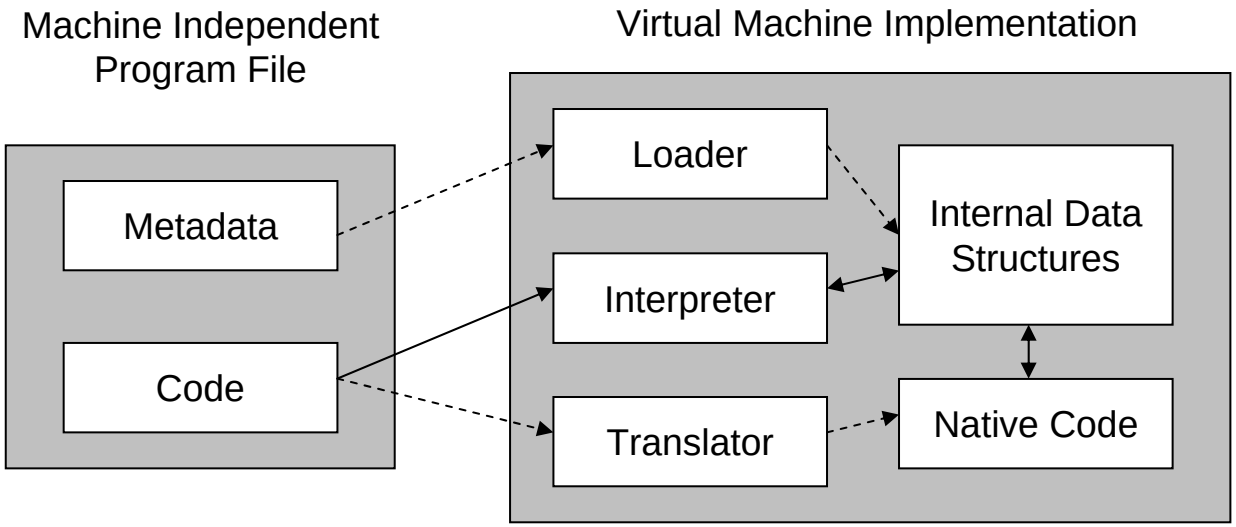
P-code

- Basis for later HLL VM's V-ISA
  - Stack ISA with minimum registers is easily portable to any ISA
  - Stack ISA leads to small binaries
  - Cell-based memory model whose details are not part of ISA
  - Interface to OS is through libraries only
    - Minimum libraries common to all platforms lead to weak I/O
- Difference to modern HLL VM
  - Support for OO programming
  - Support for networked computing

Modern HLL VMs

- Similar to P-code scheme like stack-oriented ISA
  - Not just for easing the porting of compiler, though
  - For platform-independent distribution of applications
- Platform-independence of data as well as instructions
  - Data in P-code is encoded in the P-code itself
  - Abstraction of data in modern HLL VMs
    - Metadata: a platform independent form of data
    - Data structures or resource-related information is encoded in a platform-independent way
    - VM runtime convert metadata into internal machine-dependent data

Transformation done by VM



Transform machine-independent program files to machine-dependent code and data

Security and Protection of HLL VM

- Allow load/execute programs from untrusted sources
  - VM implementation is protected from applications
  - Rely on “protection sandbox”
    - Access to remote files
      - ✓ Protected by the remote system
    - Access to local files
      - ✓ Trusted libraries and security manager
    - Prevent application from accessing memory and code that is outside the sandbox, even though they are shared with VM
      - ✓ Static checking by a trusted loader
      - ✓ Dynamic checks by a trusted emulation engine

Static and Dynamic Checking

- Branch instructions
  - All branches are fixed offsets within the code region
  - The only indirection control transfer is through
    - Explicit call instruction
    - Explicit return instruction
- Load and store instruction
  - Statically checked by loader w.r.t the metadata
  - Loader also checks control flow instructions
    - Most out-of-bound accesses or transfers are checked
  - Dynamically checked by runtime
    - Array accesses or null derefernces

Robustness : Object-Orientation

- Class and objects
- Inheritance, Methods, Polymorphism
- Objects can only be accessed via references
  - *Array* is also intrinsic form of object in JVM and CLI

Garbage Collection

- Objects are created and *float* in memory space
- *Garbage*
  - During program execution, many objects are created then abandoned, so become garbage
- *Collection*
  - Due to limited memory, we collect *garbage*
  - To improve robustness, make the VM collect garbage ***automatically***

Networking Considerations

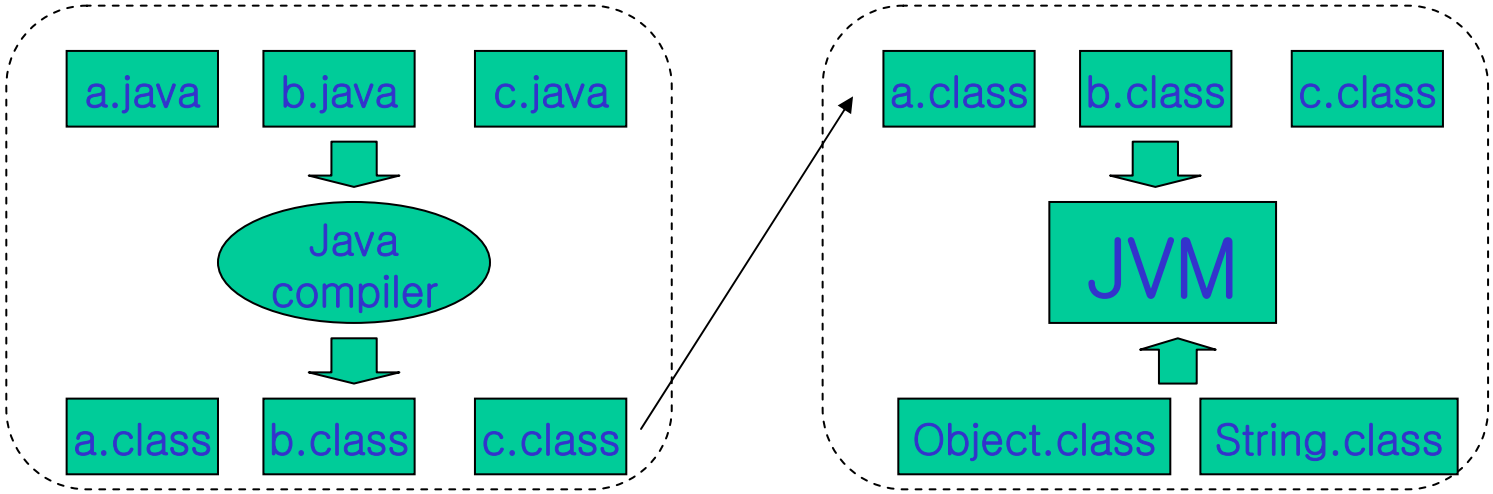
- Support for limited network bandwidth
- Reduce network bandwidth for downloading
  - Small code size
    - V-ISA code shows just the specification
      - ✓ Emulation converts the specification into real native instructions
    - Stack-oriented instruction set
  - Support dynamic class loading on demand
    - Load only classes on an as-needed basis
    - Spread loading overhead out over whole running time

Performance Issues

- OO languages are slower than non-OO languages
- VM-based HLL are even slower than native execution
- CPU speed increases can reduce performance issues
- But most VMs employ high-performance techniques
  - Most emulation techniques can be used
  - Interpretation, dynamic binary translation, static translation
- Translation is much simpler than in other VMs
  - No code discovery problem
    - Dynamic translation can be done even w/o interpretation
    - Even static translation is possible

Java VM Architecture

- Java VM is for executing Java programs
  - Usually referred to as Java Runtime Environment (JRE)
    - Contains the Java virtual machine, classes comprising the Java 2 Platform API, and supporting files
    - JDK (Java development kit): JRE, Development Tools (compiler, debugger), additional library



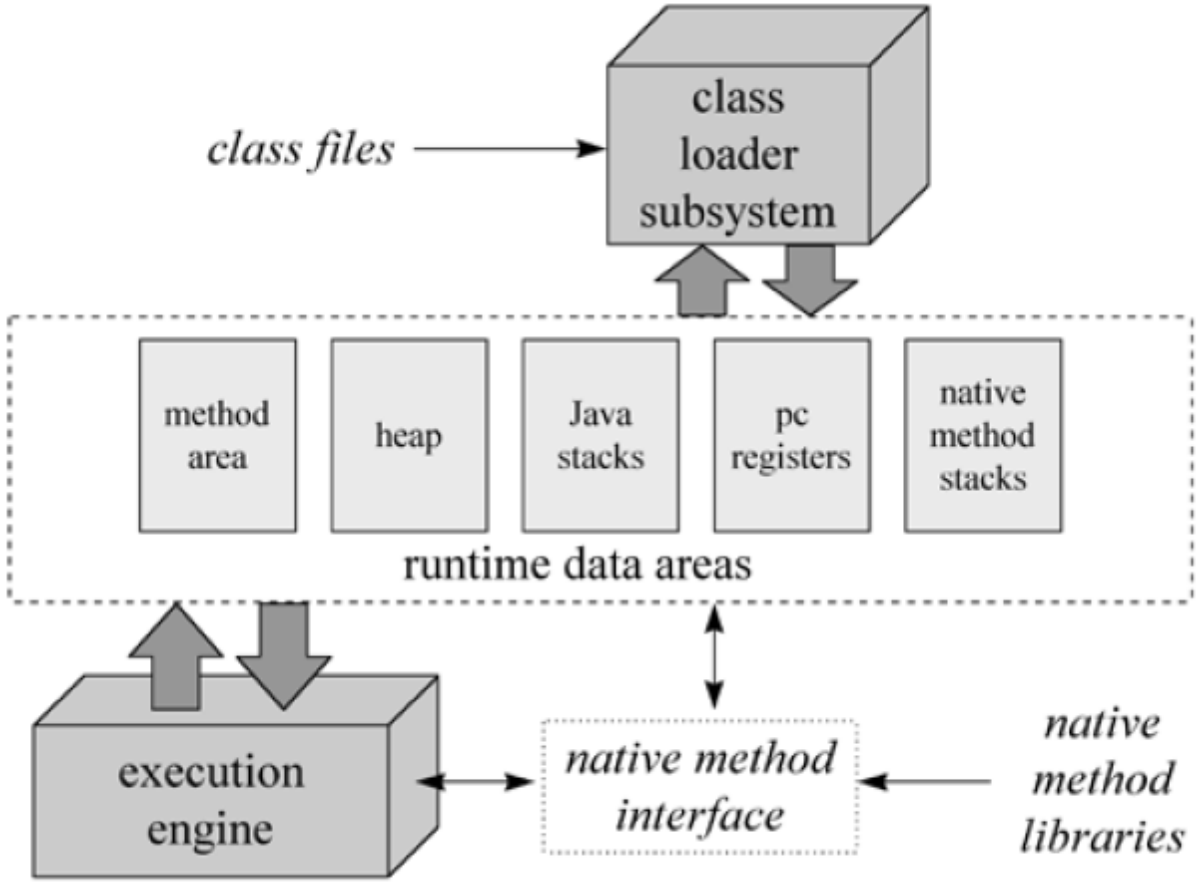
JRE Components

- Execution engine – virtual (or sometimes real) processor for executing bytecodes
- Memory manager – allocate memory for instances and arrays and perform garbage collection
- Error and exception manager – deal with exception
- Native method support – for calling c/c++ methods
- Threads interface – supporting threads and monitors
- Class loader – dynamically load Java classes from Java class files
- Security manager – verify that classes are safe and controlling access to system resources

Data Types

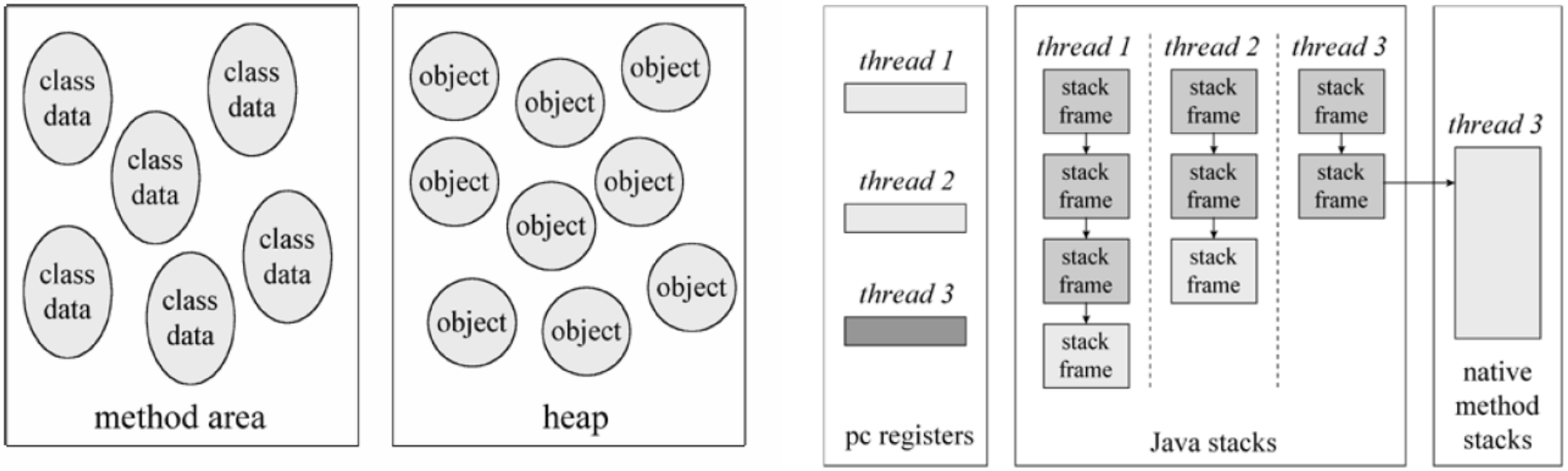
- Primitive data types
  - int, char, byte, short, float, double
- Reference type
  - Hold a reference value or null
- Objects
  - Carry data declared by the class
  - Composed of primitive data or references to other objects
  - `array` is a special object with ISA support

JRE Components and Runtime Data



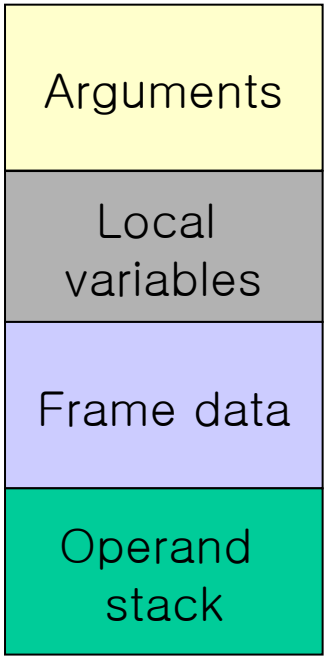
Runtime Data Area

- Method area, heap, stack, PC registers



Data Area

- PC: each JVM thread has its own program counter which is the index in bytecode stream
- Stack
  - Each JVM thread has its own stack
  - Invoke a method -> a new frame is allocated on the stack
  - Arguments and local variables: numbered from 0
  - Frame data holds return address, data for CP resolution, and exception table
  - Operand stack is used for computation
  - Caller pushes arguments on its operand stack which becomes local variables of the callee



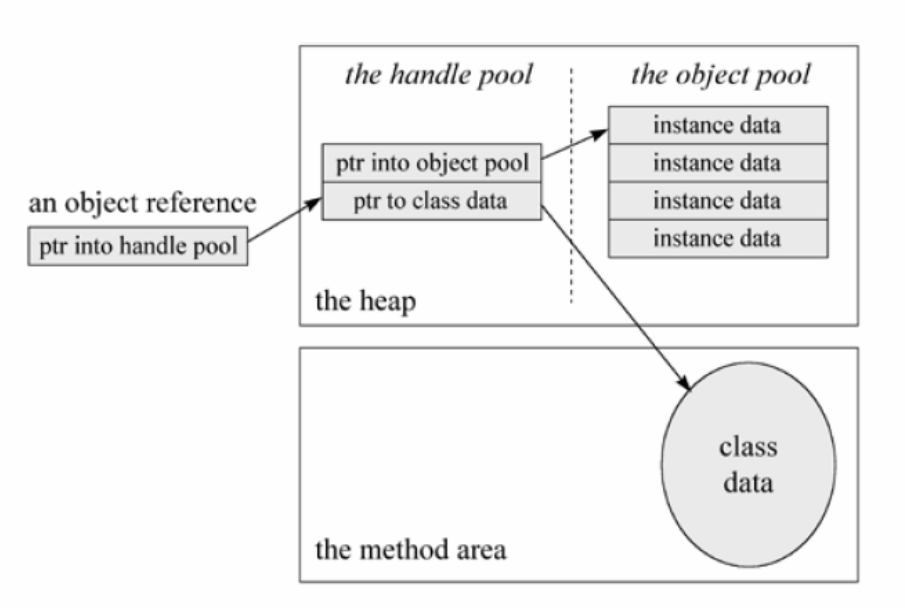
Stack Frame Structure

Data Area

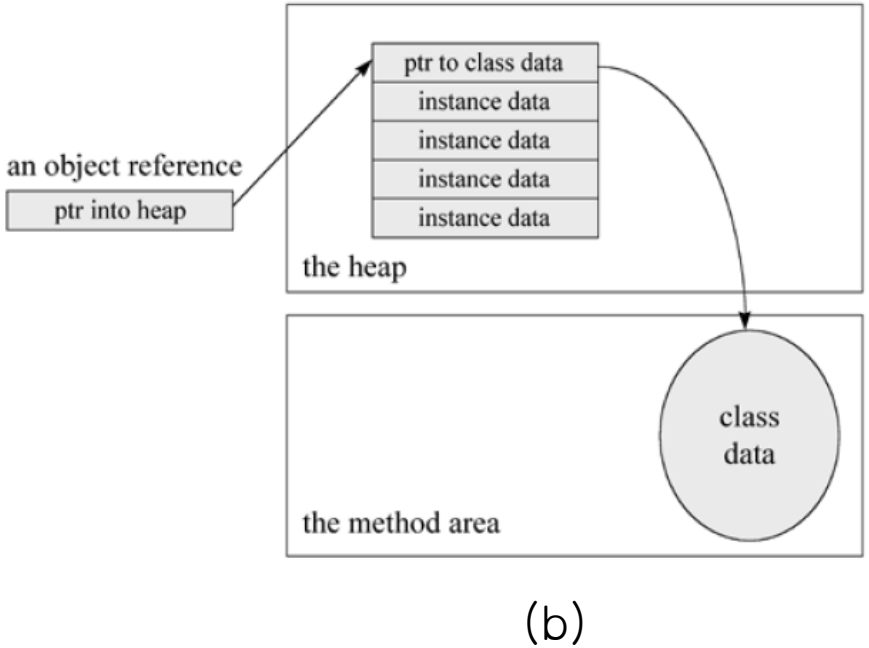
- Heap
  - Memory for objects, arrays, and class instances
  - One heap per one JVM
    - All threads share it
    - Careful synchronization of multi-threaded access is needed
  - Never deallocated
    - GC collects garbage
- Method area
  - One method area for one JVM, created on JVM start-up
  - Analogous to ‘text’ segment of conventional binaries
  - Constant pool
  - Type (class or interface) information and field information
  - Method table, information, and code
  - Class variables

Object Representation in Heap

- Object Representation in the heap: two ways



(a)

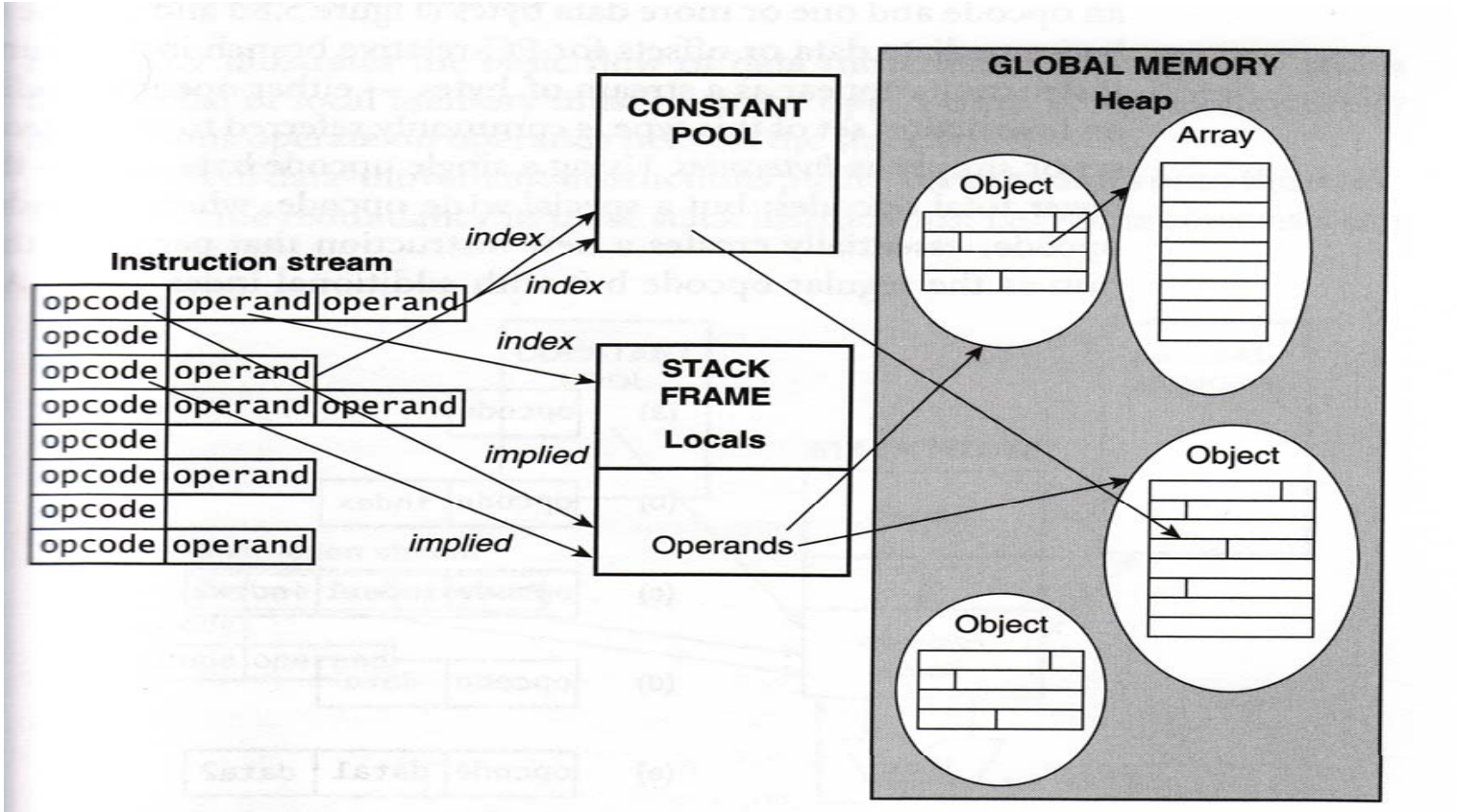


(b)

Data Area

- Constant pool (CP)
  - Analogous to symbol table
  - Collection of all symbolic data need by a class
  - Instructions often need constants (which might be shared)
    - E.g, integer operands, addresses, etc.
  - Instead of storing them within an instruction, they are saved in CP and instructions include indexes to CP
    - E.g., ldc #4, invokespecial #6
    - #4: 3040550
    - #6: java/lang/object/<init>() V
  - Each constant pool is private to its class

Memory Hierarchy



Memory Hierarchy Used by a Java Program.