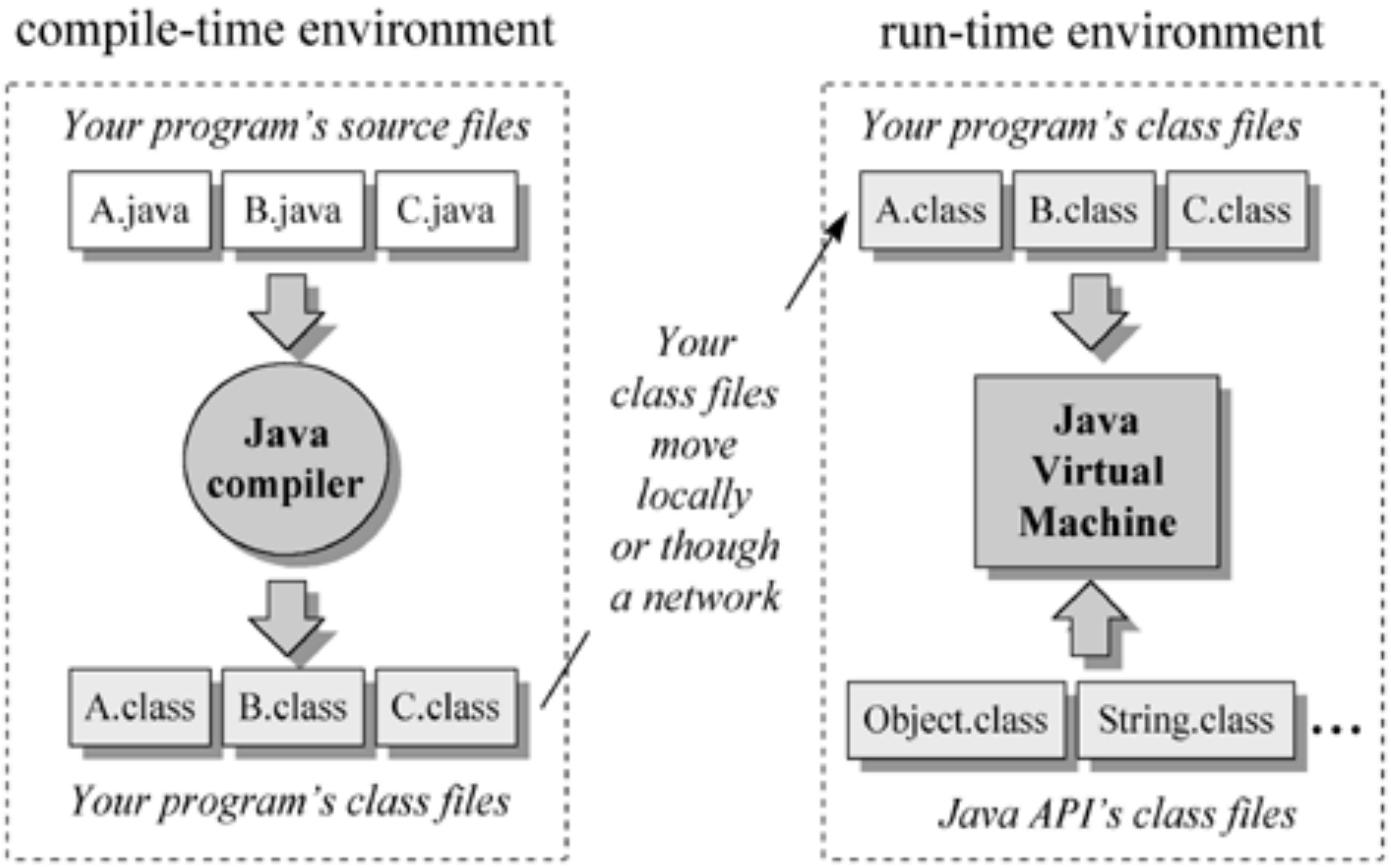
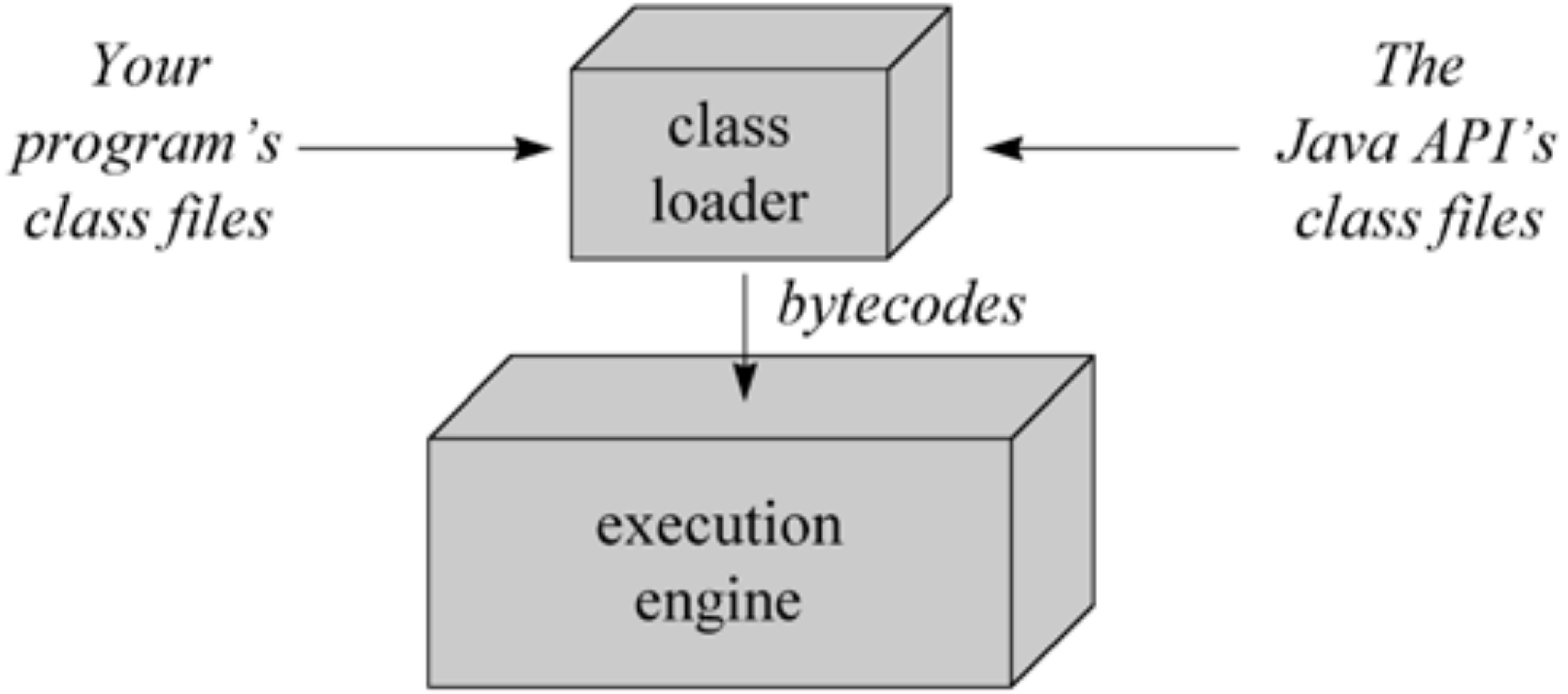
Inside JVM

Weng Kai

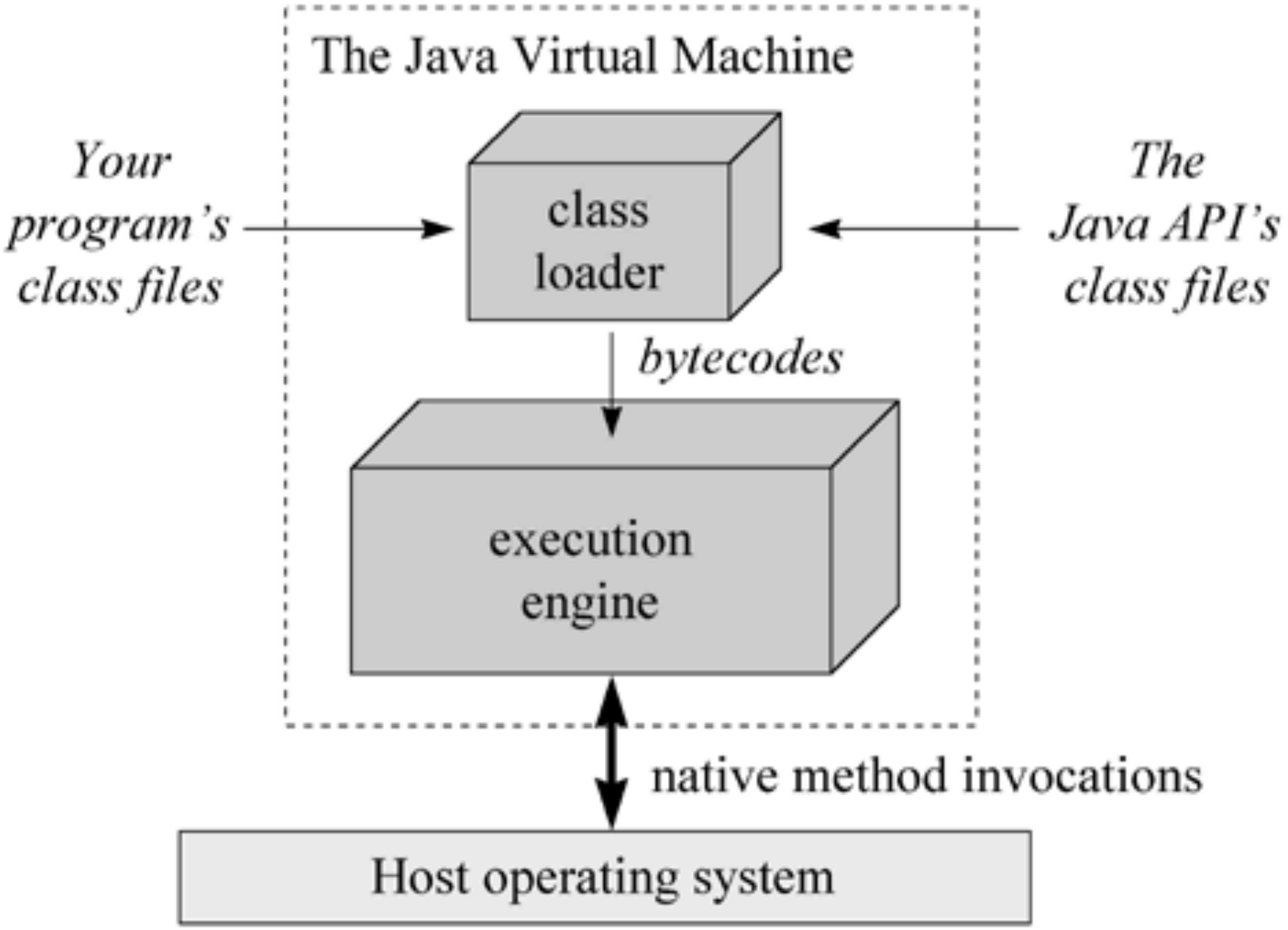
Architecture



Java Virtual Machine



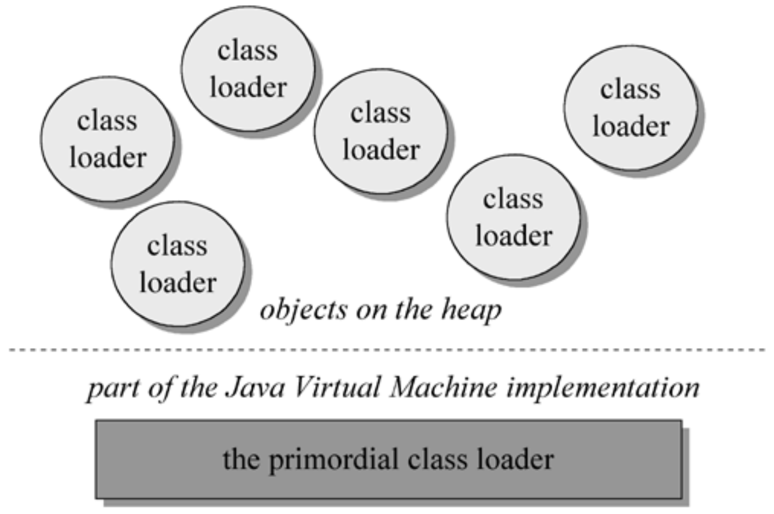
JVM is on OS


The Java Virtual Machine
Your program's class files
class loader
The Java API's class files
bytecodes
execution engine
native method invocations
Host operating system

Class Loader

- The primordial class loader (there is only one of them) is a part of the Java Virtual Machine implementation

- At run-time, a Java application can install class loader objects that load classes in custom ways, such as by downloading class files across a network.



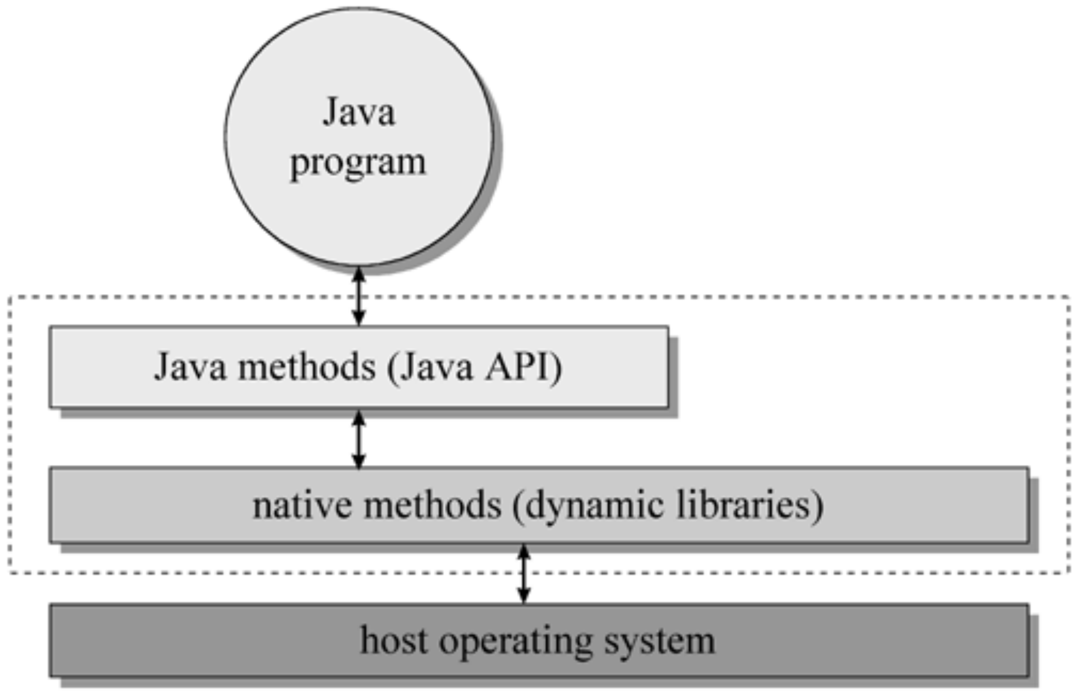
User Class Loader

- For each class it loads, the Java Virtual Machine keeps track of which class loader-- whether primordial or object--loaded the class.
- Classes can by default only see other classes that were loaded by the same class loader.

Java API

- The Java API is set of runtime libraries that give you a standard way to access the system resources of a host computer.

- The class files of the Java API are inherently specific to the host platform.



Architectural Tradeoffs

- Interpreting bytecodes is 10 to 30 times slower than native execution.
- Just-in-time compiling bytecodes can be 7 to 10 times faster than interpreting, but still not quite as fast as native execution.
- Java programs are dynamically linked.
- Array bounds are checked on each array access.
- All objects are created on the heap (no objects are created on the stack).

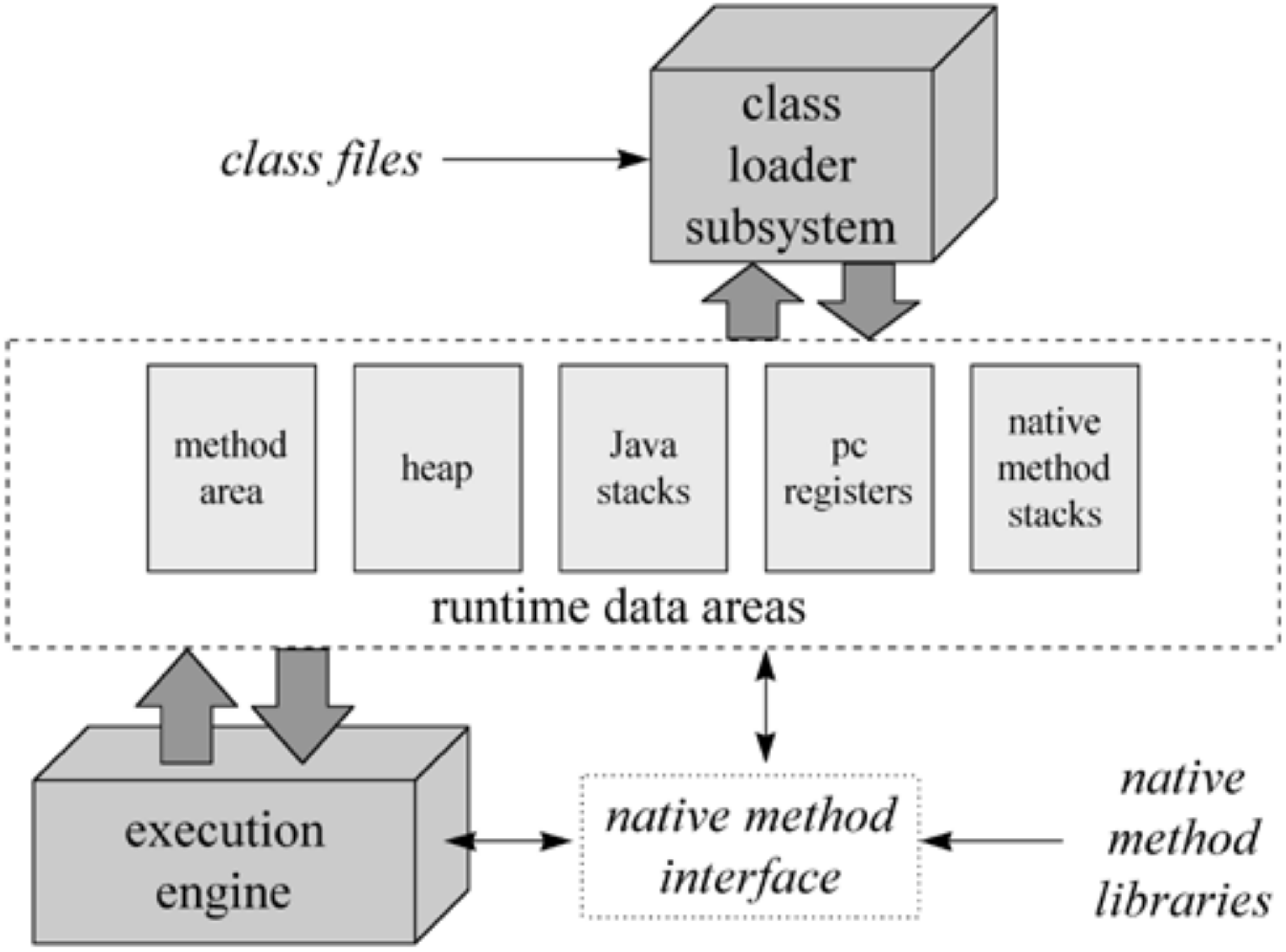
Architectural Tradeoffs

- All uses of object references are checked at run-time for null.
- All reference casts are checked at run-time for type safety.
- The garbage collector is likely less efficient (though often more effective) at managing the heap than you could be if you managed it directly as in C++.
- Primitive types in Java are the same on every platform, rather than adjusting to the most efficient size on each platform as in C++.
- Strings in Java are always UNICODE. When you really need to manipulate just an ASCII string, a Java program will be slightly less efficient than an equivalent C++ program.

Lifetime of a JVM

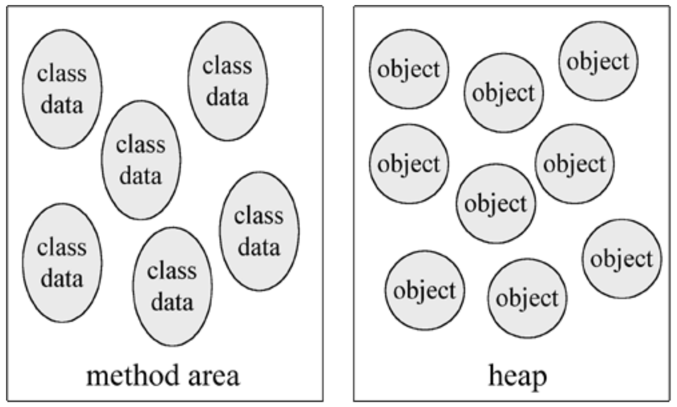
- When a Java application starts, a runtime instance is born. When the application completes, the instance dies.
- A Java Virtual Machine instance starts running its solitary application by invoking the main() method of some initial class.

Architecture of the JVM



Memory Map

- Each instance of the Java Virtual Machine has one method area and one heap. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap.



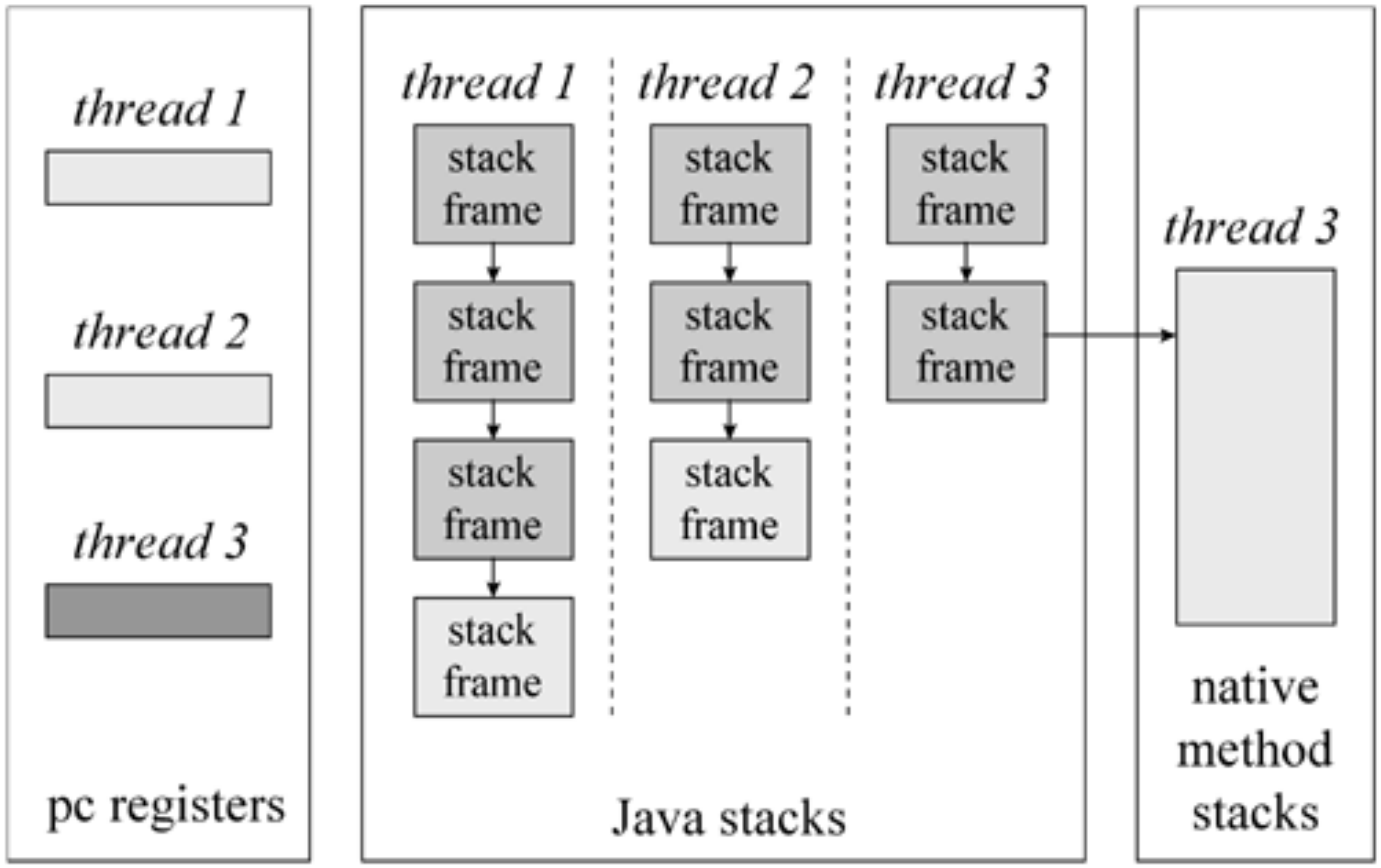
Thread Memory

- As each new thread comes into existence, it gets its own pc register (program counter) and Java stack. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A thread Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in native method stacks, as well as possibly in registers or other implementation-dependent memory areas.

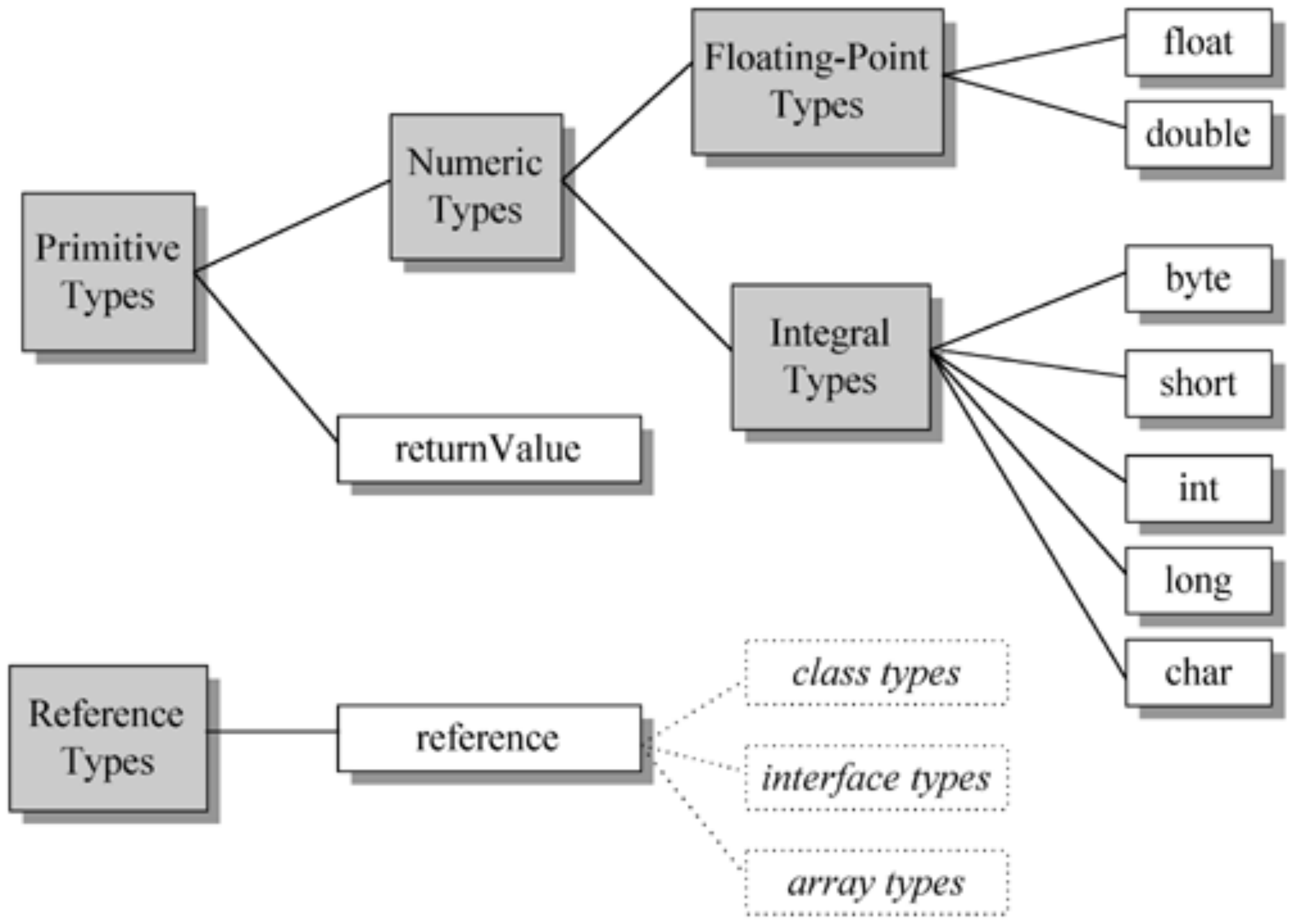
Java Stack

- The Java stack is composed of stack frames (or frames). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java Virtual Machine pushes a new frame onto that thread Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

Java Stacks


thread 1
thread 2
thread 3
pc registers
thread 1
thread 2
thread 3
stack frame
Java stacks
thread 3
native method stacks

Data Types



Primitive Types
Numeric Types
Floating-Point Types
float
double
Integral Types
byte
short
int
long
char
returnValue
Reference Types
reference
class types
interface types
array types

Class Loader Subsystem

- class loader objects are regular Java objects whose class descends from java.lang.ClassLoader. The methods of class ClassLoader allow Java applications to access the virtual machine class loading machinery.

- For every type a Java Virtual Machine loads, it creates an instance of class java.lang.Class to represent that type. Like all objects, class loader objects and instances of class Class reside on the heap. Data for loaded types resides in the method area.

Loading, Linking and Initialization

- Loading: finding and importing the binary data for a type
- Linking: performing verification, preparation, and (optionally) resolution
    - Verification: ensuring the correctness of the imported type
    - Preparation: allocating memory for class variables and initializing the memory to default values
    - Resolution: transforming symbolic references from the type into direct references.
- Initialization: invoking Java code that initializes class variables to their proper starting values.

The Method Area

- The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area. Memory for class (static) variables declared in the class is also taken from the method area.

Design Issues

- The manner in which a Java Virtual Machine implementation represents type information internally is a decision of the implementation designer.
- The virtual machine will search through and use the type information stored in the method area as it executes the application it is hosting. Designers must attempt to devise data structures that will facilitate speedy execution of the Java application, but must also think of compactness.

Design Issues

- All threads share the same method area, so access to the method area data structures must be designed to be thread-safe.
- The size of the method area need not be fixed.
- Also, the memory of the method area need not be contiguous.
- The method area can also be garbage collected.

Type Information

- The fully qualified name of the type
- The fully qualified name of the type direct superclass (unless the type is an interface or class java.lang.Object, neither of which have a superclass)
- Whether or not the type is a class or an interface
- The type modifiers (some subset of` public, abstract, final)
- An ordered list of the fully qualified names of any direct superinterfaces

Other Information

- In addition to the basic type information listed above, the virtual machine must also store for each loaded type:
  - The constant pool for the type
  - Field information
  - Method information
  - All class (static) variables declared in the type, except constants
  - A reference to class ClassLoader
  - A reference to class Class

Constant Pool

- A constant pool is an ordered set of constants used by the type, including literals (string, integer, and floating point constants) and symbolic references to types, fields, and methods.

- Entries in the constant pool are referenced by index, much like the elements of an array. Because it holds symbolic references to all types, fields, and methods used by a type, the constant pool plays a central role in the dynamic linking of Java programs.

Field Information

- The field name
- The field type
- The field modifiers (some subset of public, private, protected, static, final, volatile, transient)

Method Information

- The method name
- The method return type (or void)
- The number and types (in order) of the method parameters
- The method modifiers (some subset of public, private, protected, static, final, synchronized, native, abstract)
- In addition to the items listed above, the following information must also be stored with each method that is not abstract or native:
  - The method byte codes
  - The sizes of the operand stack and local variables sections of the method stack frame
  - An exception table

Class Variables

- Class variables are shared among all instances of a class and can be accessed even in the absence of any instance. These variables are associated with the class--not with instances of the class--so they are logically part of the class data in the method area. Before a Java Virtual Machine uses a class, it must allocate memory from the method area for each non-final class variable declared in the class.

- Constants (class variables declared final) are not treated in the same way as non-final class variables. Every type that uses a final class variable gets a copy of the constant value in its own constant pool. As part of the constant pool, final class variables are stored in the method area--just like non-final class variables. But whereas non-final class variables are stored as part of the data for the type that declares them, final class variables are stored as part of the data for any type that uses them.

A Reference to Class *ClassLoader*

- For each type it loads, a Java Virtual Machine must keep track of whether or not the type was loaded via the primordial class loader or a class loader object. For those types loaded via a class loader object, the virtual machine must store a reference to the class loader object that loaded the type. This information is stored as part of the type data in the method area.

- The virtual machine uses this information during dynamic linking.

A Reference to Class Class

- An instance of class java.lang.Class is created by the Java Virtual Machine for every type it loads. The virtual machine must in some way associate a reference to the Class instance for a type with the type data in the method area.

Method Tables

- For each non-abstract class a Java Virtual Machine loads, it could generate a method table and include it as part of the class information it stores in the method area. A method table is an array of direct references to all the instance methods that may be invoked on a class instance, including instance methods inherited from superclasses.
- A method table allows a virtual machine to quickly locate an instance method invoked on an object.

Example Code

```
class Lava {
    private int speed = 5;
     void flow() {}
}
class Volcano {
    public static void main(String[] args) {
        Lava lava = new Lava();
        lava.flow();
     }
}
```

Execution

- Given the name Volcano, the virtual machine finds and reads in file Volcano.class. It extracts the definition of class Volcano from the binary data in the imported class file and places the information into the method area. The virtual machine then invokes the main() method, by interpreting the bytecodes stored in the method area. As the virtual machine executes main(), it maintains a pointer to the constant pool (a data structure in the method area) for the current class (class Volcano).

- main()’s first line tells the Java Virtual Machine to allocate enough memory for the class listed in constant pool entry one. The virtual machine uses its pointer into Volcano constant pool to look up entry one and finds a symbolic reference to class Lava. It checks the method area to see if Lava has already been loaded.

- The symbolic reference is just a string giving the class fully qualified name: "Lava".

- When the virtual machine discovers that it has’n yet loaded a class named "Lava," it proceeds to find and read in file Lava.class. It extracts the definition of class Lava from the imported binary data and places the information into the method area.

- The Java Virtual Machine then replaces the symbolic reference in Volcano constant pool entry one, which is just the string "Lava", with a pointer to the class data for Lava.

- This process of replacing symbolic references with direct references (in this case, a native pointer) is called constant pool resolution.

- Finally, the virtual machine is ready to actually allocate memory for a new Lava object.

- A Java Virtual Machine can always determine the amount of memory required to represent an object by looking into the class data stored in the method area. The actual amount of heap space required by a particular object, however, is implementation-dependent.

- Once the Java Virtual Machine has determined the amount of heap space required by a Lava object, it allocates that space on the heap and initializes the instance variable speed to zero, its default initial value.

The Heap

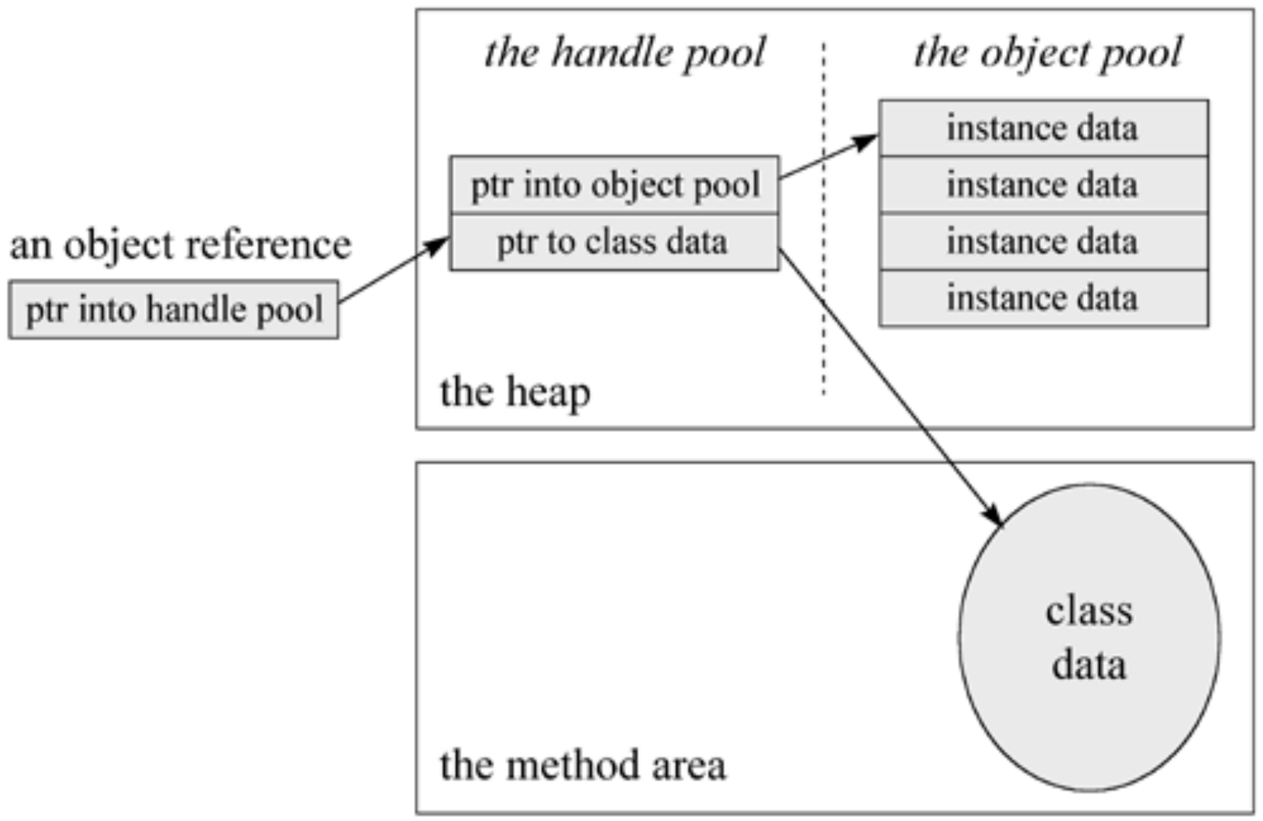
- Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap. As there is only one heap inside a Java Virtual Machine instance, all threads share it.

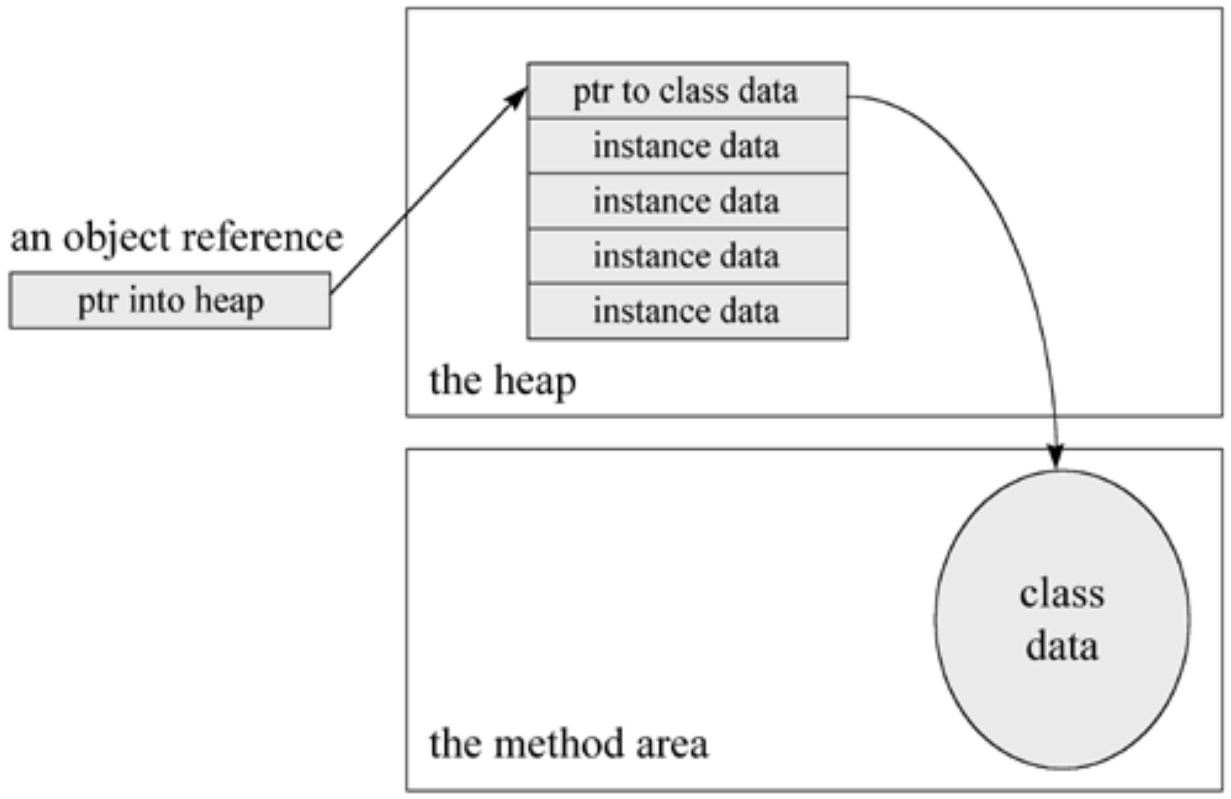
Garbage Collection

- The Java Virtual Machine has an instruction that allocates memory on the heap for a new object, but has no instruction for freeing that memory.
- A garbage collector is not strictly required by the Java Virtual Machine specification.
- The specification only requires that an implementation manage its own heap in some manner.

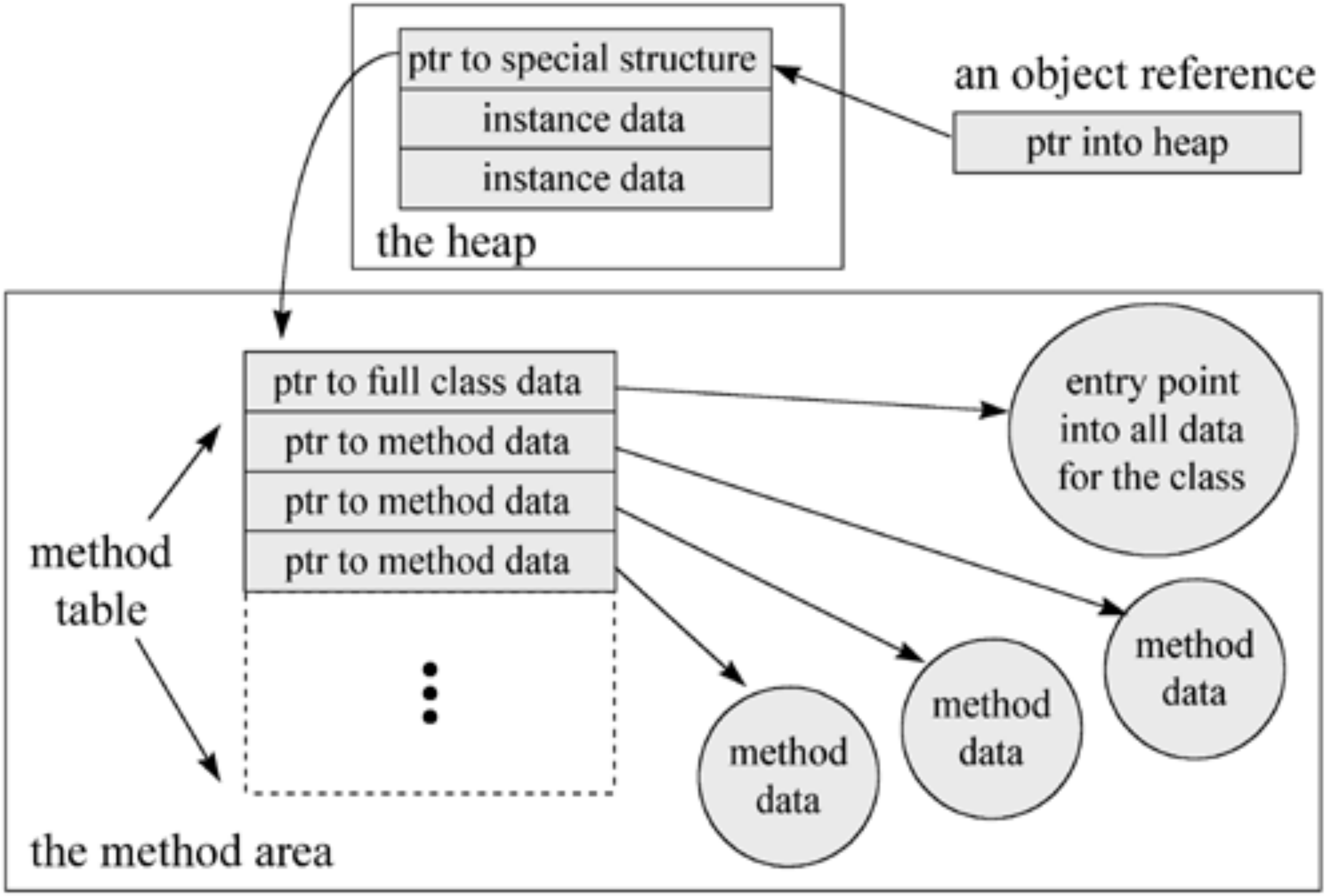
Object Representation

- Object representation--an integral aspect of the overall design of the heap and garbage collector--is a decision of implementation designers

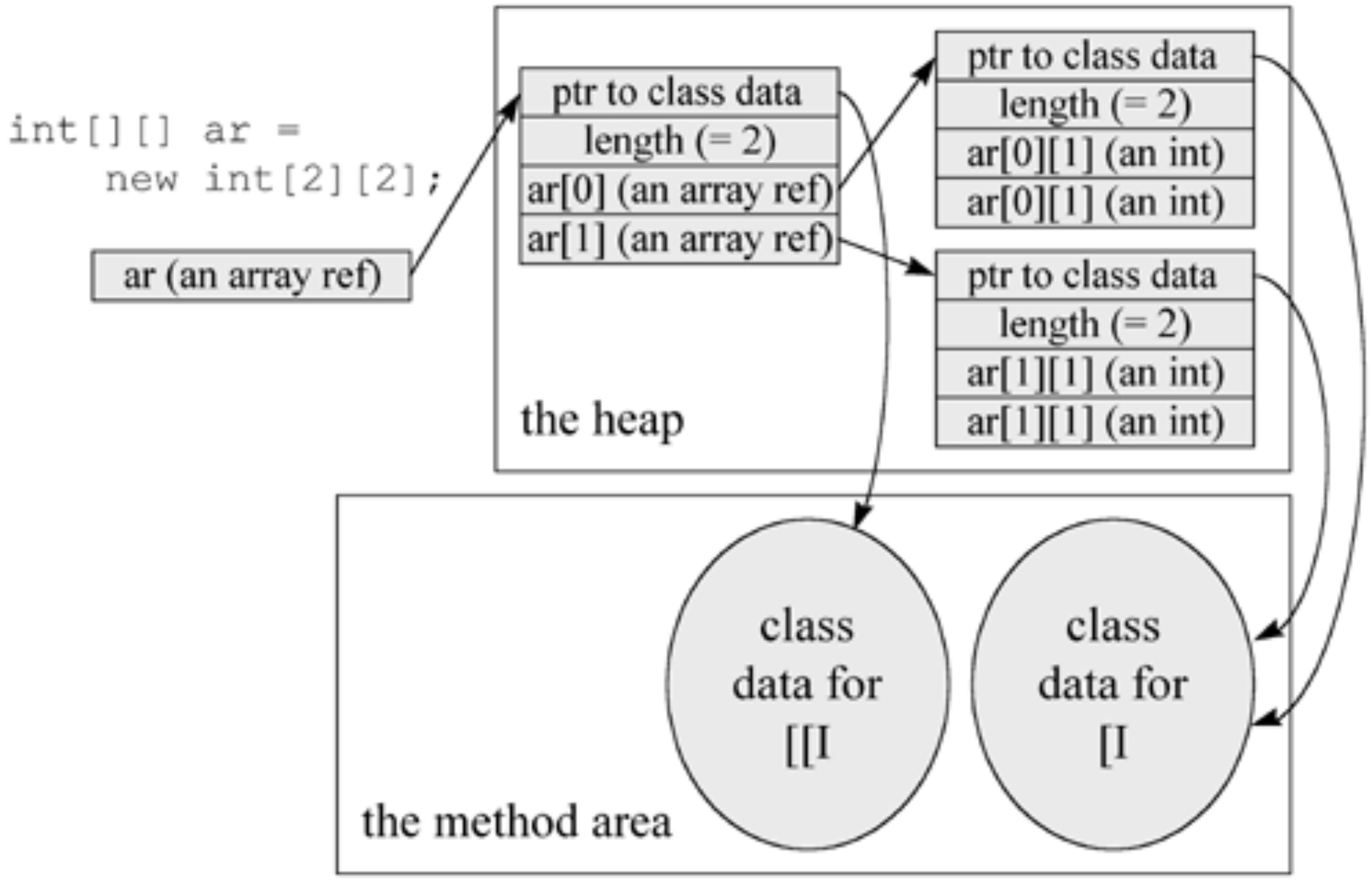




Method Table



Array Representation



Program Counter

- Each thread of a running program has its own pc register, or program counter, which is created when the thread is started. The pc register is one word in size, so it can hold both a native pointer and a returnValue. As a thread executes a Java method, the pc register contains the address of the current instruction being executed by the thread. An "address" can be a native pointer or an offset from the beginning of a method bytecodes. If a thread is executing a native method, the value of the pc register is undefined.

Java Stack

- When a new thread is launched, the Java Virtual Machine creates a new Java stack for the thread. As mentioned earlier, a Java stack stores a thread state in discrete frames. The Java Virtual Machine only performs two operations directly on Java Stacks: it pushes and pops frames.

- The method that is currently being executed by a thread is the thread current method. The stack frame for the current method is the current frame. The class in which the current method is defined is called the current class, and the current class constant pool is the current constant pool. As it executes a method, the Java Virtual Machine keeps track of the current class and current constant pool. When the virtual machine encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame.

Method Stack

- When a thread invokes a Java method, the virtual machine creates and pushes a new frame onto the thread Java stack. This new frame then becomes the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data.

- A method can complete in either of two ways. If a method completes by returning, it is said to have normal completion. If it completes by throwing an exception, it is said to have abrupt completion. When a method completes, whether normally or abruptly, the Java Virtual Machine pops and discards the method stack frame. The frame for the previous method then becomes the current frame.

Method Stacks

- All the data on a thread Java stack is private to that thread. There is no way for a thread to access or alter the Java stack of another thread. Because of this, you need never worry about synchronizing multi-threaded access to local variables in your Java programs. When a thread invokes a method, the method local variables are stored in a frame on the invoking thread Java stack. Only one thread can ever access those local variables: the thread that invoked the method.

Method Stack

- Like the method area and heap, the Java stack and stack frames need not be contiguous in memory. Frames could be allocated on a contiguous stack, or they could be allocated on a heap, or some combination of both. The actual data structures used to represent the Java stack and stack frames is a decision of implementation designers. Implementations may allow users or programmers to specify an initial size for Java stacks, as well as a maximum or minimum size.

Stack Frame

- The stack frame has three parts:
  - local variables;
  - operand stack;
  - frame data.
- When the Java Virtual Machine invokes a Java method, it checks the class data to determine the number of words required by the method in the local variables and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the Java stack.

Local Variables

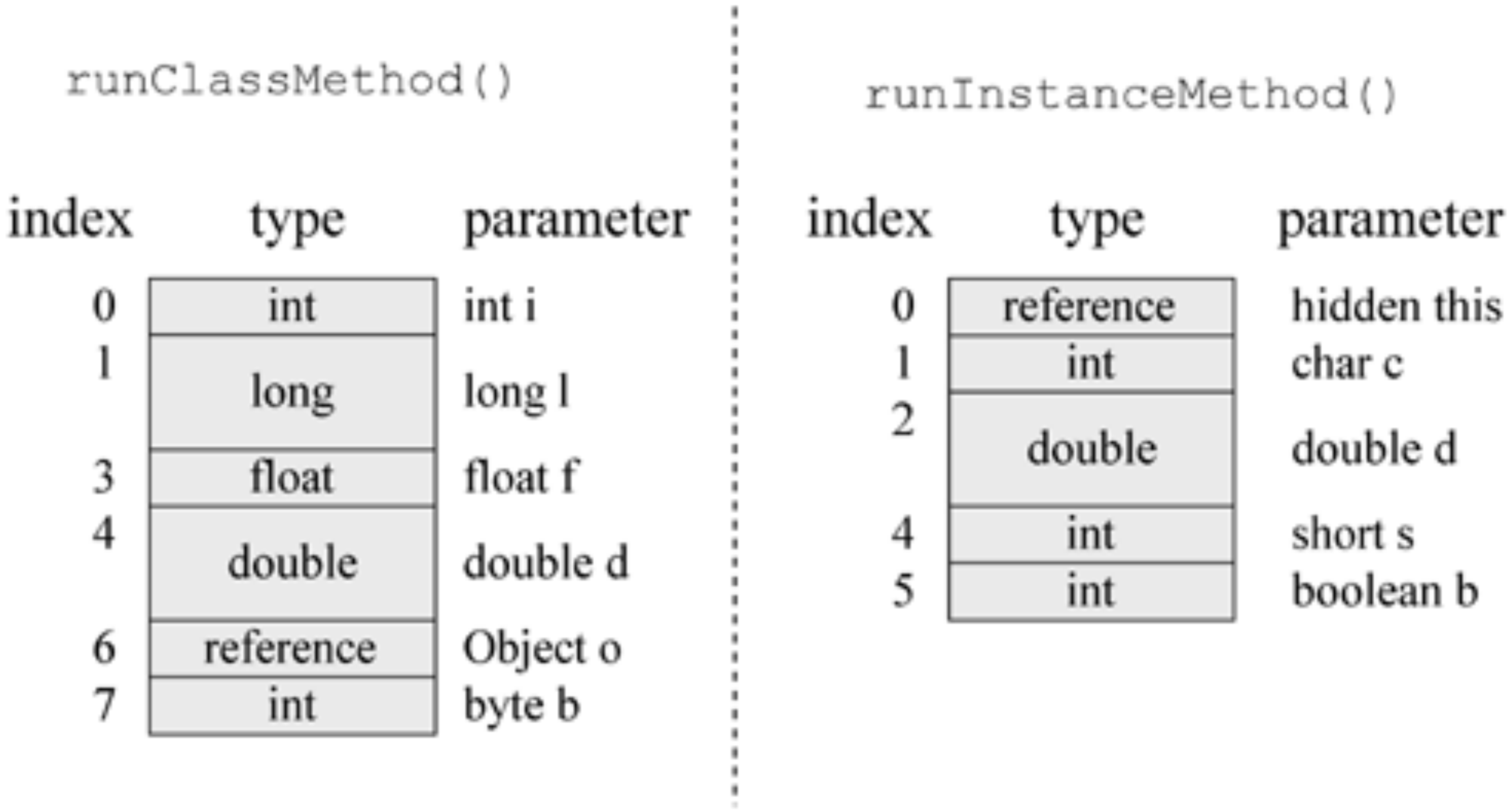
- The local variables section of the Java stack frame is organized as a zero-based array of words. Instructions that use a value from the local variables section provide an index into the zero-based array. Values of type int, float, reference, and returnValue occupy one entry in the local variables array. Values of type byte, short, and char are converted to int before being stored into the local variables. Values of type long and double occupy two consecutive entries in the array.

- All values in the local variables are word-aligned. Dual-entry longs and doubles can start at any index.

- Compilers place the parameters into the local variable array first, in the order in which they are declared.

Example Code

```
class Example3a {
    public static int runClassMethod(int i, long l, float f,
        double d, Object o, byte b) {
         return 0;
    }
    public int runInstanceMethod(char c, double d, short s,
boolean b) {
    return 0;
    }
}
```


runClassMethod()
index
type
parameter
0
int
int i
1
long
long l
3
float
float f
4
double
double d
6
reference
Object o
7
int
byte b
runInstanceMethod()
index
type
parameter
0
reference
hidden this
1
int
char c
2
double
double d
4
int
short s
5
int
boolean b

Operand Stack

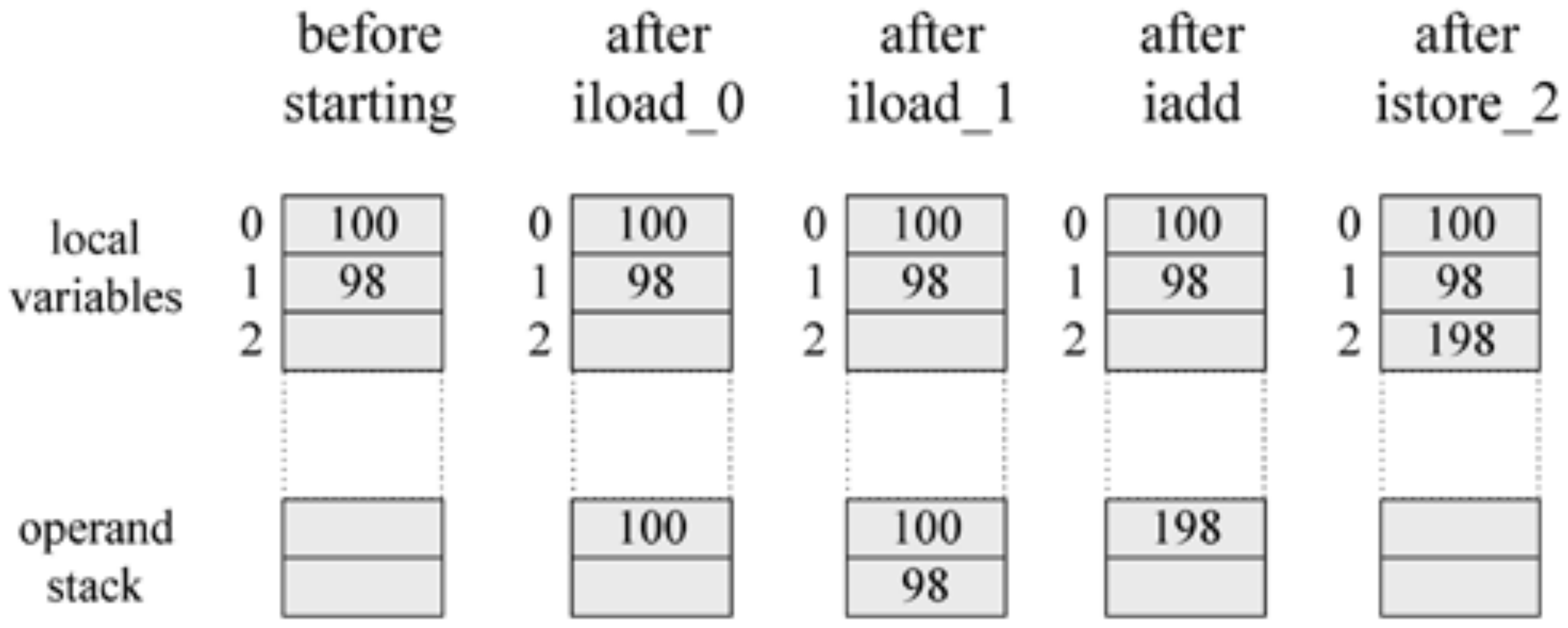
- The operand stack is organized as an array of words. But unlike the local variables, which are accessed via array indices, the operand stack is accessed by pushing and popping values. If an instruction pushes a value onto the operand stack, a later instruction can pop and use that value.

- The Java Virtual Machine is stack-based.

- The Java Virtual Machine uses the operand stack as a work space.

Example Code

```
iload_0    // push the int in local variable 0 onto the stack
iload_1    // push the int in local variable 1 onto the stack
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
```



Frame Data

- In addition to the local variables and operand stack, the Java stack frame includes data to support constant pool resolution, normal method return, and exception dispatch. This data is stored in the frame data portion of the Java stack frame.

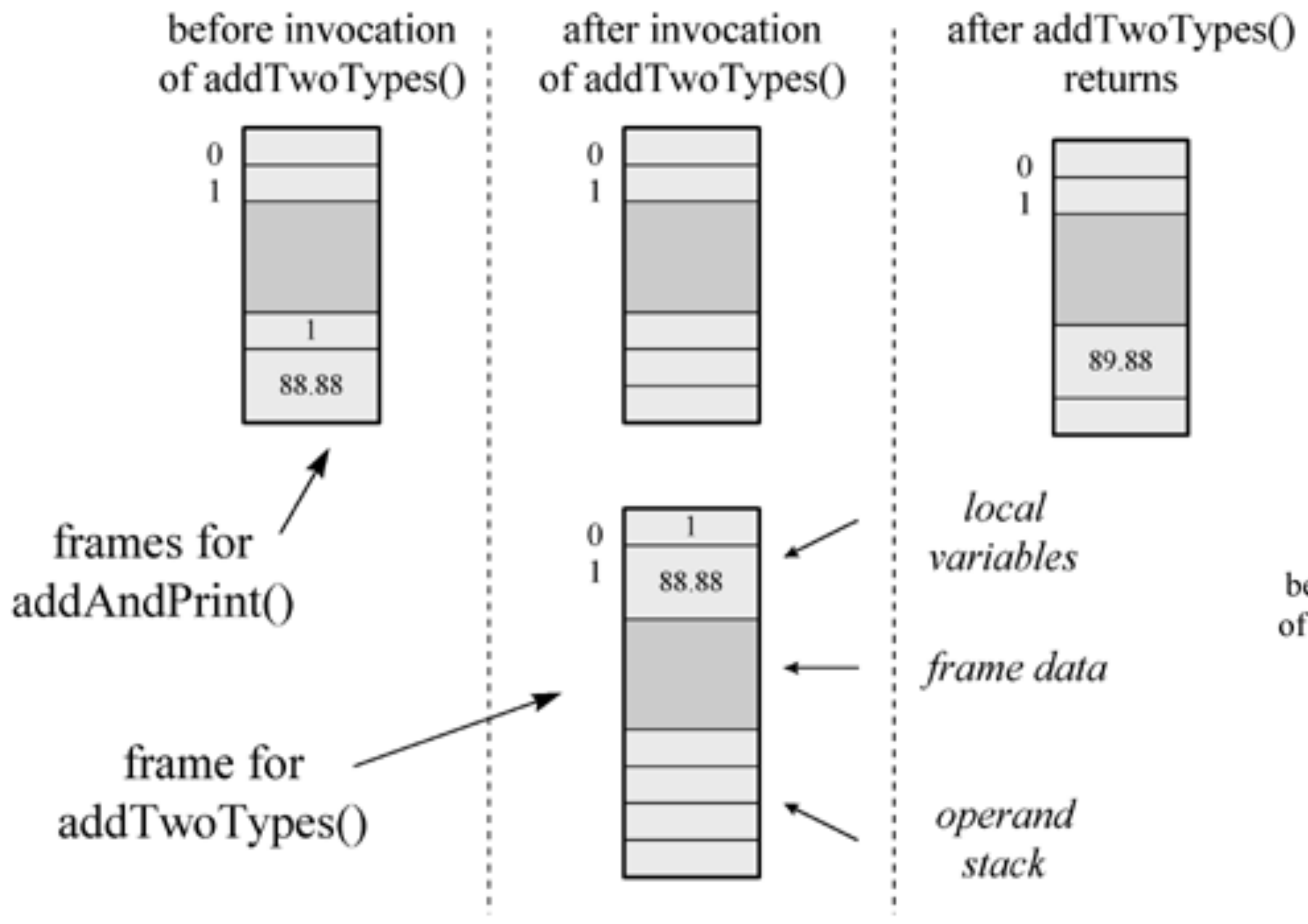
- Whenever the Java Virtual Machine encounters any of the instructions that refer to an entry in the constant pool, it uses the frame data pointer to the constant pool to access that information.

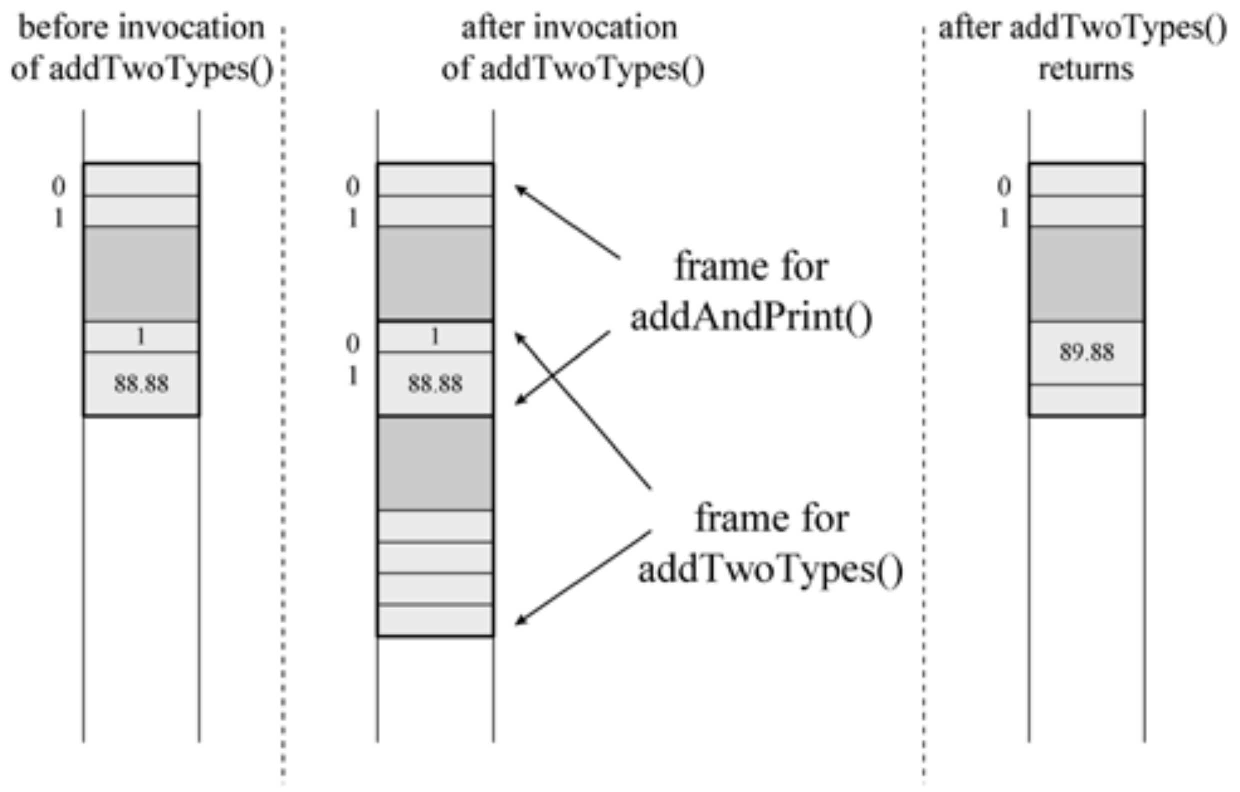
- When the virtual machine looks up a constant pool entry that refers to a class, interface, field, or method, that reference may still be symbolic. If so, the virtual machine must resolve the reference at that time.

Example Code

```
class Example3c {
    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```

Possible Implementations of the Java Stack




before invocation of addTwoTypes()
after invocation of addTwoTypes()
after addTwoTypes() returns
0
1
1
88.88
89.88
frame for addAndPrint()
frame for addTwoTypes()

Instruction Set

- use javap to get bytecode of a .class file
- For many instructions, the virtual machine needs to know the types being operated on to know how to perform the operation. For example, the Java Virtual Machine supports two ways of adding two words together, yielding a one-word result. One addition treats the words as ints, the other as floats. The difference between these two instructions facilitates verification, but also tells the virtual machine whether it should perform integer or floating point arithmetic.

Types

- The majority of the instructions, however, operate on a specific type. The mnemonics for most of these "typed" instructions indicate their type by a single character prefix that starts their mnemonic.

Type	**Code**	**Example**	**Description**
`byte`	`b`	`baload`	load `byte` from array
`short`	`s`	`saload`	load `short` from array
`int`	`i`	`iaload`	load `int` from array
`long`	`l`	`laload`	load `long` from array
`char`	`c`	`caload`	load `char` from array
`float`	`f`	`faload`	load `float` from array
`double`	`d`	`daload`	load `double` from array
`reference`	`a`	`aaload`	load `reference` from array

Class File Header

- magic : 0xCAFEBABE
- minor_version and major_version
- constant_pool_count (u2) and constant_pool

constant_pool_count

- Throughout the class file, constant pool entries are referred to by the integer index that indicates their position in the constant_pool list. The first entry in the list has an index of one.

- Although there is no entry in the constant_pool list that has an index of zero, the missing zeroeth entry is included in the constant_pool_count.

types

- Each constant pool entry starts with a one-byte tag that indicates the type of constant making its home at that position in the list.

Entry Type	**Tag Value**	**Description**
`CONSTANT_Utf8`	1	A UTF-8 encoded Unicode string
`CONSTANT_Integer`	3	An `int` literal value
`CONSTANT_Float`	4	A `float` literal value
`CONSTANT_Long`	5	A `long` literal value
`CONSTANT_Double`	6	A `double` literal value
`CONSTANT_Class`	7	A symbolic reference to a class or interface
`CONSTANT_String`	8	A `String` literal value
`CONSTANT_Fieldref`	9	A symbolic reference to a field
`CONSTANT_Methodref`	10	A symbolic reference to a method declared in a class
`CONSTANT_InterfaceMethodref`	11	A symbolic reference to a method declared in an interface
`CONSTANT_NameAndType`	12	Part of a symbolic reference to a field or method

access_flags

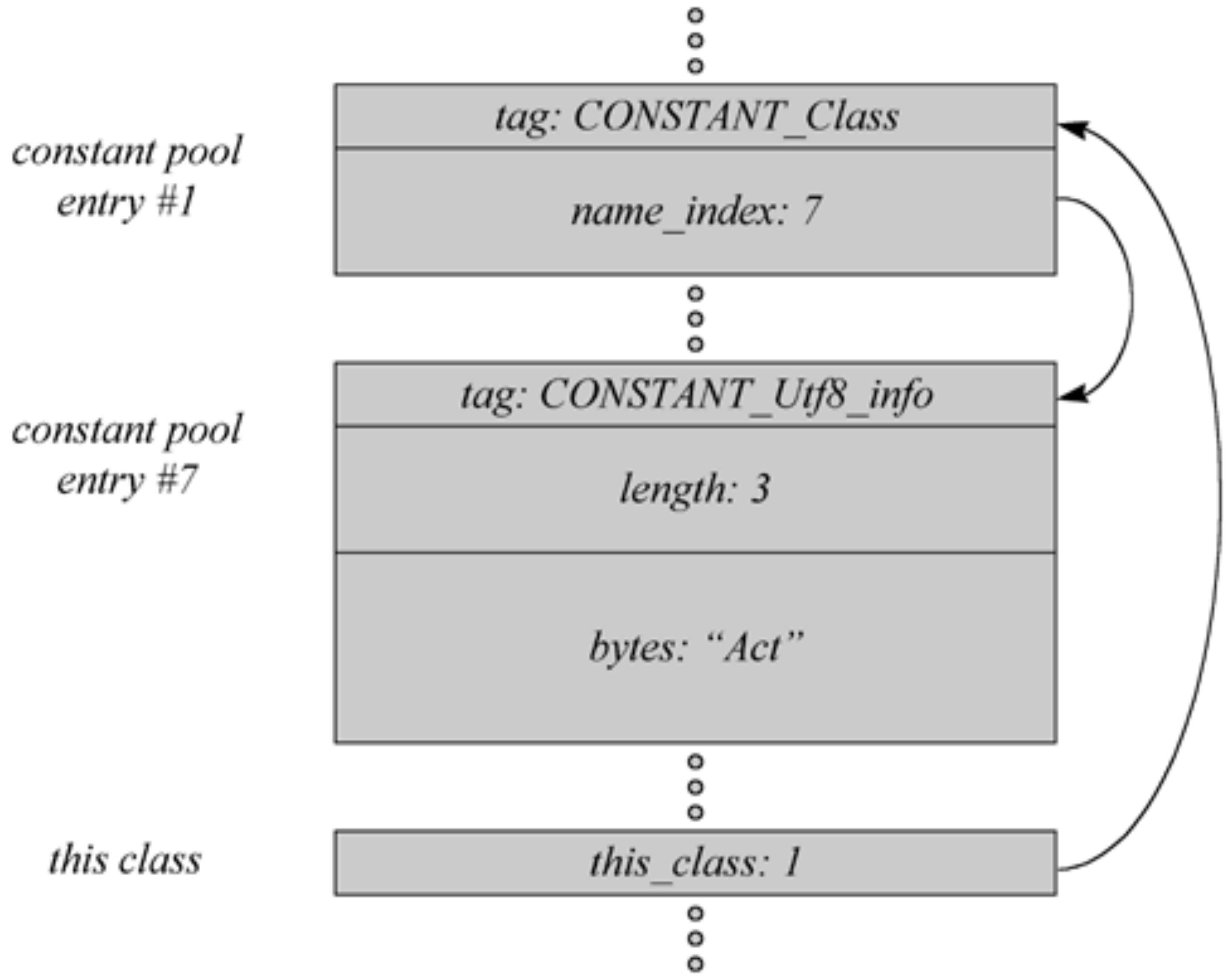
- The first two bytes after the constant pool, the access flags, reveal several pieces of information about the class or interface defined in the file.

Flag Name	Value	Meaning if Set	Set By
`ACC_PUBLIC`	0x0001	Type is public	Classes and interfaces
`ACC_FINAL`	0x0010	Class is final	Classes only
`ACC_SUPER`	0x0020	Use new	`invokespecial` semanticsClasses and interfaces
`ACC_INTERFACE`	0x0200	Type is an interface, not a class	All interfaces, no classes
`ACC_ABSTRACT`	0x0400	Type is abstract	All interfaces, some classes

this_class

- The next two bytes are the this_class item, an index into the constant pool. The constant pool entry at position this_class must be a CONSTANT_Class_info table, which has two parts: a tag and a name_index. The tag will have the value CONSTANT_Class. The constant pool entry at position name_index will be a CONSTANT_Utf8_info table containing the fully qualified name of the class or interface.

this_class


constant pool entry #1
tag: CONSTANT_Class
name_index: 7
constant pool entry #7
tag: CONSTANT_Utf8_info
length: 3
bytes: "Act"
this class
this_class: 1