CSCE 314
Programming Languages

JVM

Dr. Hyunyoung Lee

Java Virtual Machine and Java

- The Java Virtual Machine (JVM) is a stack-based abstract computing machine.
- JVM was designed to support Java -- Some concepts and vocabulary from Java carry to JVM
- A Java program is a collection of class definitions written in Java
- A Java compiler translates a class definition into a format JVM understands: class file.
- A class file contains JVM instructions (or bytecodes) and a symbol table, and some other information. When a JVM reads and executes these instructions, the effect is what the original Java program called for: the class file has the same semantics as the original Java program.

JVM and Java (cont.)

Although JVM was primarily designed for Java, it is theoretically possible to design a translator from any programming language into JVM's world.

Role of Java Virtual Machine

- Loads class files needed and executes bytecodes they contain in a running program
- Organizes memory into structured areas

Refer the most recent edition of the official definition of the JVM:

"The Java Virtual Machine Specification, Java SE 8 Edition"

http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

by Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley

The JVM specification defines:

1. A set of instructions and a definition of the meanings of those instructions called bytecodes.
2. A binary format called the class file format, used to convey bytecodes and related class infrastructures in a platform-independent manner.
3. An algorithm for identifying programs that cannot compromise the integrity of the JVM. This algorithm is called verification.

A Stack-based Architecture

The JVM instruction set is designed around a stack-based architecture with special object-oriented instructions.

Bytecodes stored in a class file are stored in a binary format

- readable by a computer program but unintelligible to humans
- one needs special programs to display the class files in human readable forms
- Human readable forms usually are mnemonics
- the JVM specification suggests some mnemonics

Example:

```
iconst_2  // push integer constant 2
iconst_3  // push integer constant 3
iadd      // add them together
```

Representation of Memory

Typical CPU instruction set views memory as array of bytes

- Construct object: allocate contiguous sequence of bytes
- Access a field: access bytes at a specific offset
- Call a function: jump to a location in memory where function resides

JVM allows no byte-level access

- Direct operations for allocating objects, invoking methods, accessing fields

Example JVM Bytecode

Assume the following method in Factorial.java source code:

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

> javac Factorial.java

> javap -c Factorial

will produce …

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst_1        // push the integer constant 1
1: istore_1        // store it in register 1 (the res variable)
2: iload_0         // push register 0 (the n parameter)
3: ifle    16      // if negative or null, go to PC 16
6: iload_1         // push register 1 (res)
7: iload_0         // push register 0 (n)
8: imul            // multiply the two integers at top of stack
9: istore_1        // pop result and store it in register 1
10: iinc   0, -1   // decrement register 0 (n) by 1
13: goto   2       // go to PC 2
16: iload_1        // load register 1 (res)
17: ireturn        // return its value to caller
```

Bytecode Format

Operation: Instruction Format:

mnemonic
operand1
operand2

(opcode) -- one byte

. . .

Description: A longer description detailing constraints on the operand stack contents or constant pool entries, the operation performed, the type of the result, etc.

Linking Exceptions: If any linking exceptions may be thrown by the execution of this instruction, they are set off one to a line, in the order in which they can be thrown.

Runtime Exceptions: Ditto. Other than the linking and execution exceptions, if any, listed for an instruction, that instruction must not throw any runtime exception except for instances of VirtualMachineError or its subclasses.

Notes: Comments not strictly part of the specification of an instruction are set aside as notes.

The Main Loop of a JVM Interpreter

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

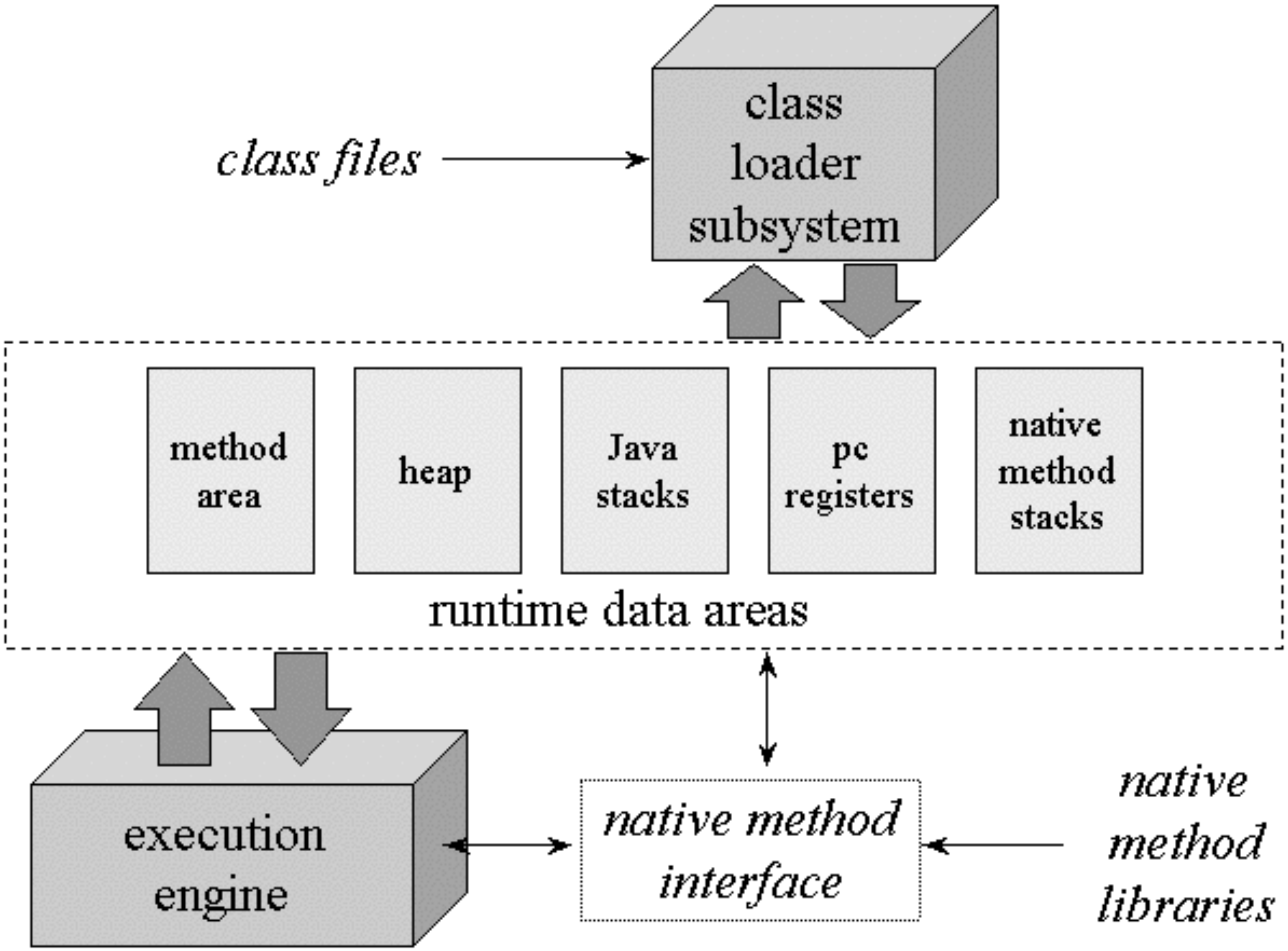
Class Loaders

- Data in class file format do not have to be stored in a file. They can be stored in a database, across the network, as part of Java archive file (JAR), or in variety of other ways.
- Essential component of using class files is the class ClassLoader, part of the Java platform. Many different subclasses of ClassLoaders are available, which load from databases, across the network, from JAR files, and so on. Java-supporting web browsers have a subclass of ClassLoader that can load class file over the Internet.
- If you store your information in some nonstandard format (such as compressed) or in a nonstandard place (such as a database), you can write your own subclass of ClassLoader.

The Verifier

- To ensure that certain parts of the machine are kept safe from tampering, the JVM has a verification algorithm to check every class.
- Programs can try to subvert the security of the JVM in a variety of ways:
  - They might try to overflow the stack, hoping to corrupt memory they are not allowed to access.
  - They might try to cast an object inappropriately, hoping to obtain pointers to forbidden memory.
- The verification algorithm ensures that this does not happen by tracing through the code to check that objects are always used according to their proper types.

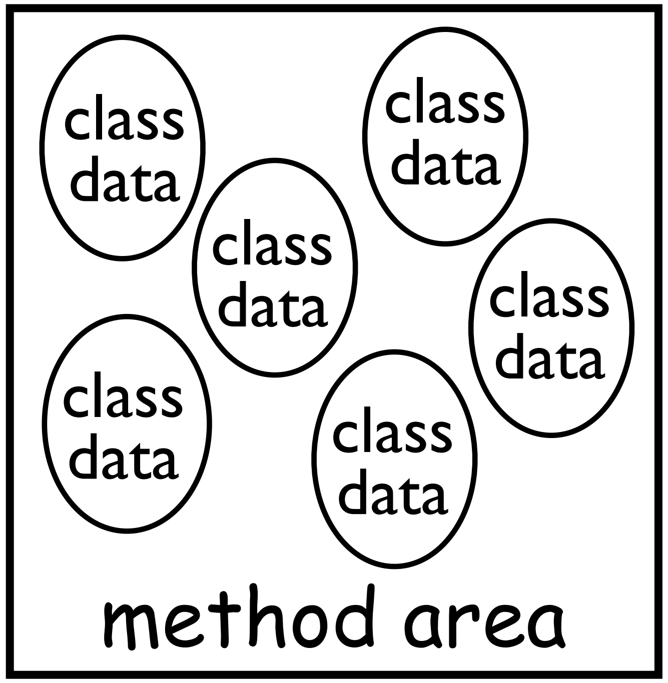
Internal Architecture of JVM



Runtime Data Areas in JVM

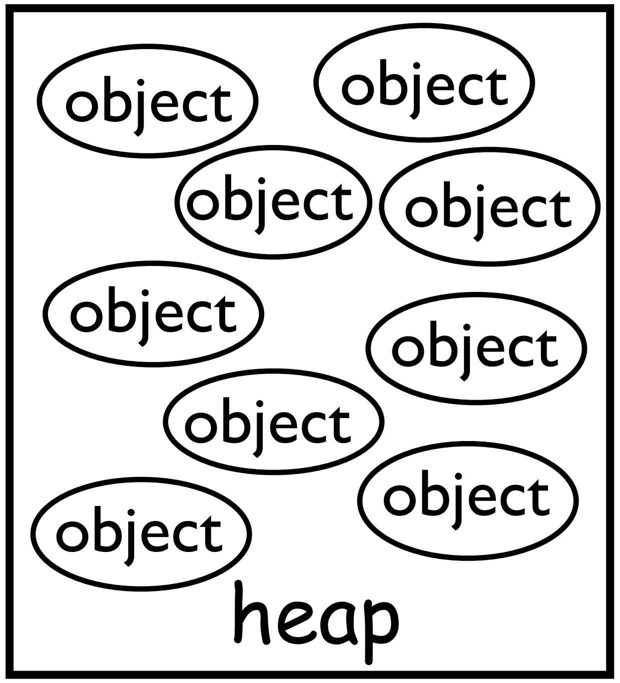
- Method area: contains class information, code and constants
- Heap: memory for class instances and arrays. This is where objects live.
- Java stack (JVM stack): stores "activation records" or "stack frames" - a chunk of computer memory that contains the data needed for the activation of a routine
- PC registers – program counters for each thread
- Native method stacks

Runtime Data Areas



Method Area

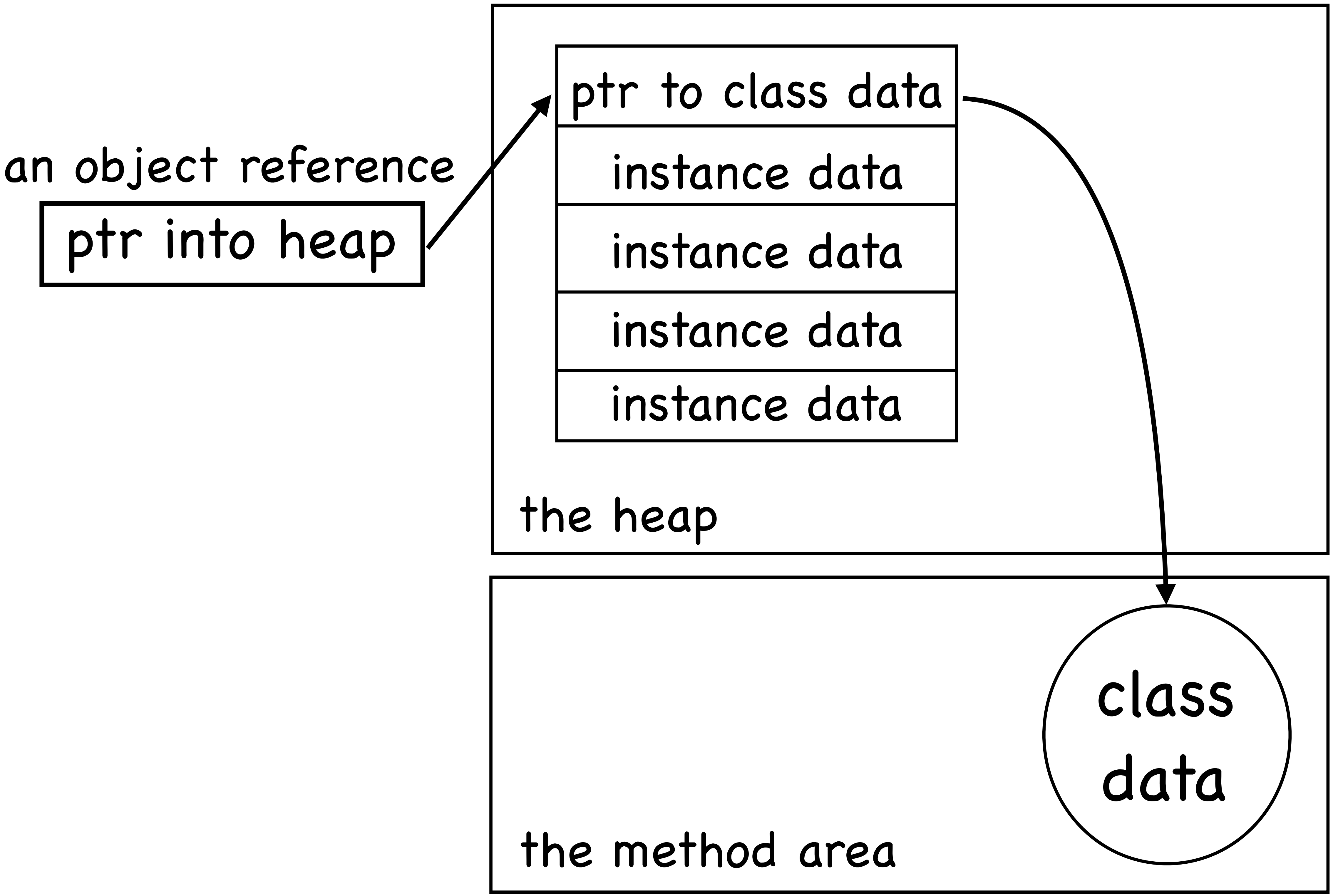
- Contains class information
- One for each JVM instance
- Shared by all threads in JVM
- One thread access at a time



Heap

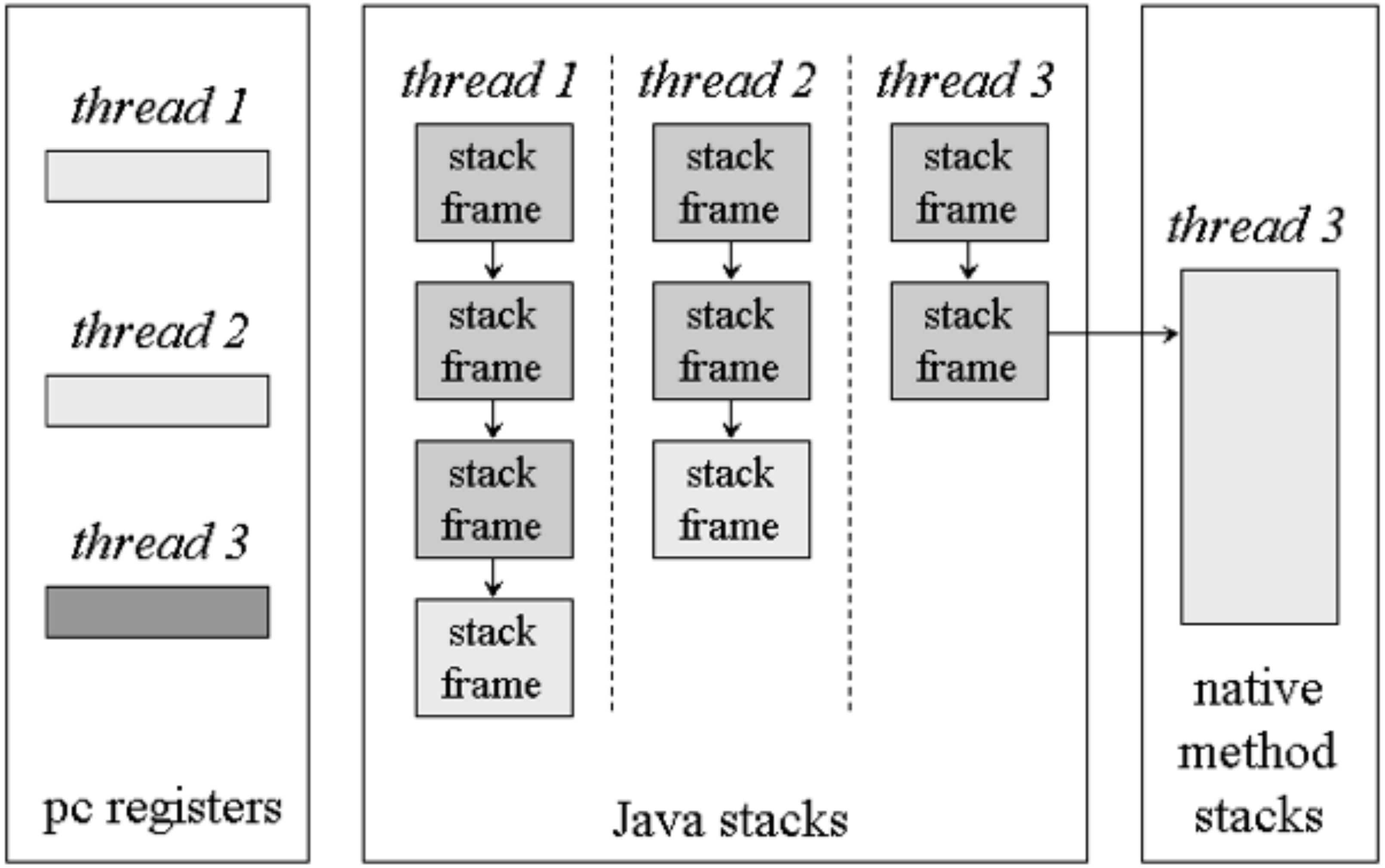
- Contains class instance or array (objects)
- One for each JVM instance
- Facilitates garbage collection
- Expands and contracts as program progresses

Objects Representation in Heap



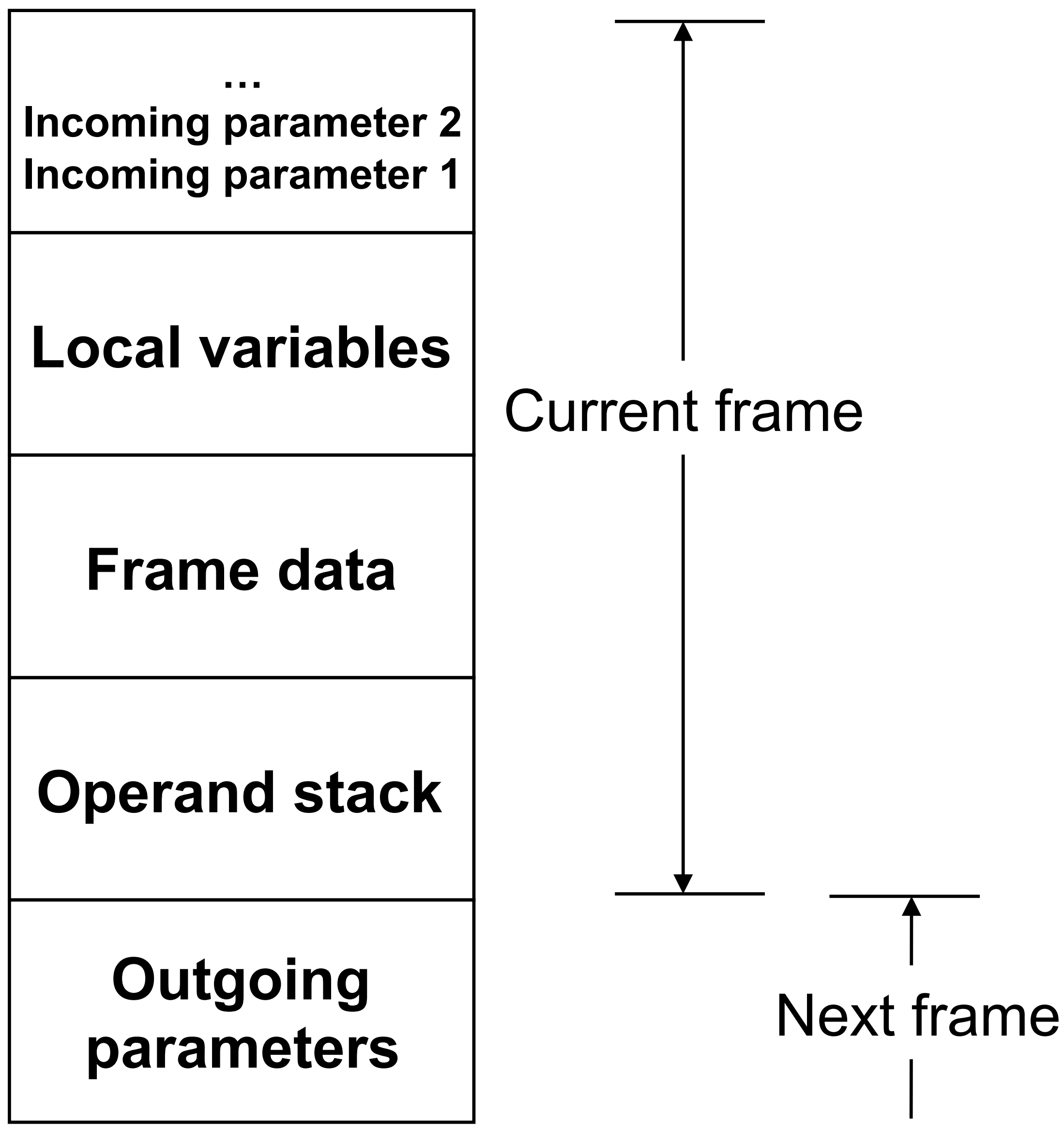
Runtime Data Areas: JVM Stack

- Each thread creates separate JVM stack
- Contains frames: current thread's state
- Pushing and popping of frames



Stack Frame

- Local Variables
    - Organized in an array
    - Accessed via array indices
- Operand Stack
    - Organized in an array
    - Accessed via pushing and popping
    - Always on top of the stack frame
    - Work space for operations
- Frame Data
    - Constant Pool Resolution: Dynamic Binding
    - Normal Method Return
    - No exception thrown
    - Returns a value to the previous frame



Bank Account Example

```
public class BankAccount {
    private double balance;
    public static int totalAccounts = 0;
    public BankAccount() {
        balance = 0;
        totalAccounts++;
    }
    public void deposit( double amount ) {  balance += amount; }
}
public class Driver {
    public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
    }
}
```

See the animated ppt slides for how the Java activation records work with this example code

Principles

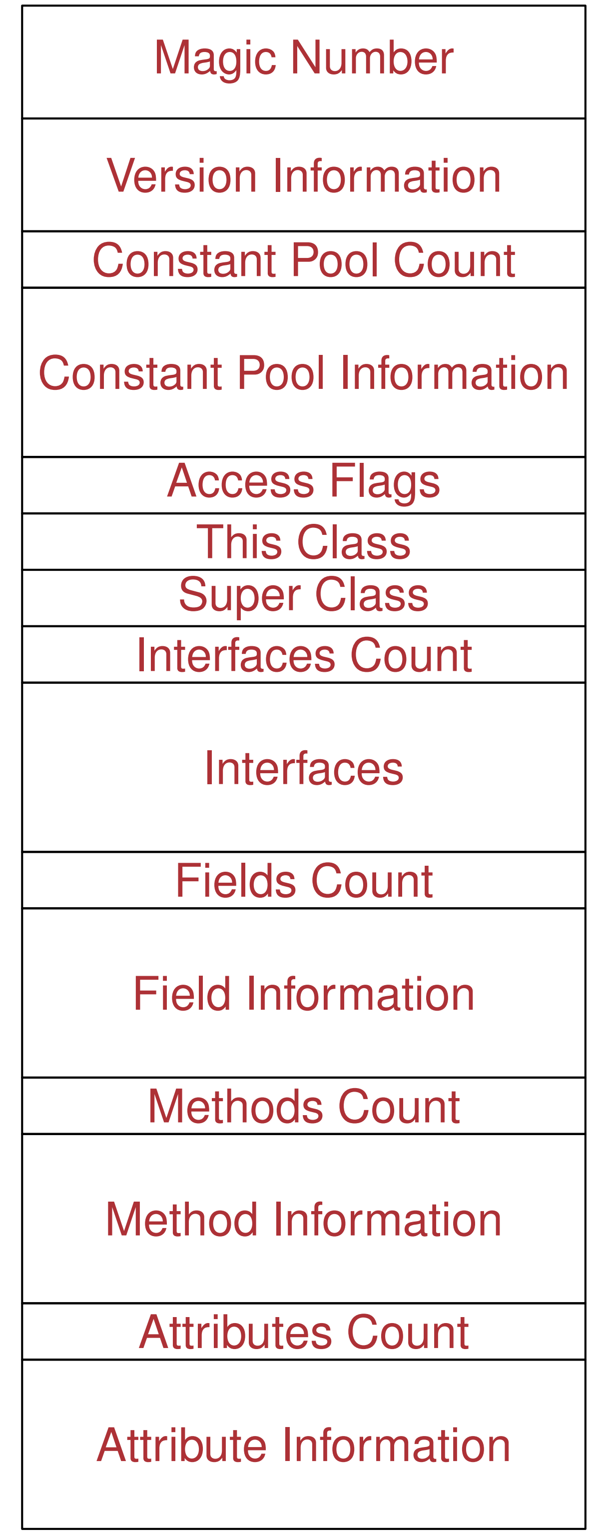
- All the Java binary class files that form a complete program do not have to be loaded when a program is started.
  - Rather, they are loaded on demand, at the time they are needed by the program.
  - For efficiency, the first time a class file is used, it is parsed and placed into method memory
- Each component of a class file is of a fixed size or the size is explicitly given immediately before the component contents.
  - In this manner, the loader can parse the entire class file from beginning to end, with each of the component being easily recognized and delineated. The same principle applies recursively.

Class File Format

- A stream of 8-bit bytes: All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes.
- Multibyte data items are always stored in big-endian order, where the high bytes come first.
  - Format supported by java.io.DataInput, and java.io.DataOutput; java.io.DataInputStream, and java.io.DataOutputStream;
- For illustration, assume C/C++ like structures with items of types
  - u1 : a single 8-bit byte quantity
  - u2 : two 8-bit bytes quantity
  - u4 : four 8-bit bytes quantity
- Successive items are stored in the class file sequentially, without padding or alignment.

Class File Format

```
struct ClassFile {
  u4        magic_number;
  u2        minor_version;
  u2        major_version;
  u2        constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2        access_flags;
  u2        this_class;
  u2        super_class;
  u2        interfaces_count;
  u2        interfaces[interfaces_count];
  u2        fields_count;
  field_info        fields[fields_count];
  u2        methods_count;
  method_info     methods[methods_count];
  u2        attributes_count;
  attribute_info   attributes[attributes_count];
};
```



Class File Header

- Magic Number (magic_number): identifies this block of data as a binary class file. It has the value 0xCAFEBABE. A byte sequence, same for all JVM class files.
- Version Information determine the version of a class file.
  - minor_version
  - major_version
  - Together, the minor and major version items. If major_version has value x and minor_version has value y, then the class file has version x.y, for example 1.6.

The Constant Pool

- constant_pool_count: The value of this item is equal to the number of entries in the constant_pool plus one. A constant_pool index is valid if and only if it is greater than zero and less than constant_pool_count.
- constant_pool[]: a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within a ClassFile structure and its substructures. The constant_pool is indexed from 1 to constant_pool_count-1.

Access Specifiers

access_flags: The value of this item is a mask of flags used to denote access permissions to and properties of this class or interface. The interpretation of each flag is as follows:

Flag Name	Value	Interpretation
`ACC_PUBLIC`	`0x0001`	Declared `public`
`ACC_FINAL`	`0x0010`	Declared `final`
`ACC_SUPER`	`0x0020`	Treat superclass methods specially
`ACC_INTERFACE`	`0x0200`	Is an interface, not a class
`ACC_ABSTRACT`	`0x0400`	Declared `abstract`

Self Description

- this_class: The value of this item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info structure representing the class or interface defined by the class file.
- super_class: For a class this item must have value zero or must be a valid index into constant_pool:
  - if zero, the class file represents the class Object.
  - otherwise, the constant_pool entry at that index must be a CONSTANT_Class_info structure representing the direct superclass of the class defined by the class file.

Interfaces

- interfaces_count: The value of this item gives the number of direct superinterfaces for this class or interface type.
- interfaces[]: The value of this item must be a valid index into the constant_pool table. The entry at each interfaces[i] must be a CONSTANT_Class_info structure representing an interface that is a direct superinterface of this class or interface type, in the left-to-right order given in the source program.

Fields

- fields_count: The value of this item is the number of field_info structures in the fields table. The field_info structures represent all fields, both class variables, and instance variables, declared by this class or interface type.
- fields[]: Each value in the fields table must be a field_info structure giving a complete description of a field in this class or interface. The fields table includes only those fields that are declared by this class or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

Methods

- methods_count: The value of this item gives the number of method_info structures in the methods table.
- methods[]: Each value in this table must be a method_info structure giving a complete description of a method in this class or interface. If the method is not native or abstract, the JVM instructions implementing the method are also supplied. The methods table does not include items representing methods that are inherited from superclasses or superinterfaces.

Attributes

- attributes_counts: The value of this item gives the number of attributes in the attributes table of this class.
- attributes[]: Each value of the attributes table must be an attribute structure. A JVM implementation is required to silently ignore any or all attributes in the attributes table of the ClassFile structure that it does not recognize.

Verification Process

- One of the most distinctive features of the JVM
  - Ensure that class files loaded in memory follow certain rules
  - Guarantee that programs cannot gain access to fields and methods they are not allowed to access, and that they can't otherwise trick the JVM into doing unsafe things
- Verification algorithm is applied to every class as it is loaded into the system, before instances are created or static properties are used
  - Safely download Java applets from the Internet.
  - Allows the JVM implementation to assume that the class has certain safety properties, therefore making certain optimizations possible.

Ideas Behind the Verifier

Given a class file, the verifier asks:

1. Is it a structurally valid class file? (refer slide 21)
2. Are all constant references correct?
3. Are the instructions valid?
4. Will the stack and local variables always contain values of the appropriate types?
5. Do the classes used really exist, and do they have the necessary methods and field?

Exact details are spelled out in the JVMS document

Are All Constant References Correct?

- Do Class and String constants have a reference to another constant that is an UTF8 constant?
- Do Fieldref, Methoref, and InterfaceMethodref constants have a class index that is a Class constant and a name-and-type index that is a NameAndType constant?
- Do NameAndType constants have a name index that points to a UTF8 and type index that points to a UTF8?
- Does this_class index point to a Class constant?
- Does the super_class index point to a Class constant?
- Do the name and descriptor fields of each field and each method entry point to a UTF8 constant?
- Are the type names referred to by NameAndType constants valid method or field descriptors?

Are All the Instructions Valid?

Once a class file is known to be structurally valid, the verifier tries to answer:

- Does each instruction begin with a recognized opcode?
- If the instruction takes a constant pool reference as an argument, does it point to an actual constant pool entry with the correct type?
- If the instruction uses a local variable, is the local variable range within the correct range? (determined by the .limit locals directive)
- If the instruction is a branch, does it point to the beginning of an instruction? (JVM branch instructions use byte offsets)

Will Each Instruction Always Find a Correctly Formed Stack and Local Variable Array?

- Ideals:
  - You want the verifier to prove that your program does what you meant
  - Failing that, you'd like the verifier to reject any programs that could do something illegal, like stack overflow or applying an instruction to a value with wrong type.
- Approximations:
  - The ideals are too strong requirements: undecidability
  - Will the right element always be on top of the stack?
  - Each time an instruction at a particular location is executed, will the stack always be the same size?

Example 1: Summing array of integers

```
.method public static addit([I)V
.limit stack 2
.limit locals 3
  iconst_0    // -- initialize running total: variable 1
  istore_1
  iconst_0    // -- initialize loop counter: variable 2
  istore_2
loop:
  aload_0     // -- if length of array is greater
  arraylength  // -- than the loop counter then exit the loop
  iload_2
  if_icmpge end
body:
  aload_0     // push array a
  iload_2     // push loop counter i
  iaload      // push a[i]
  iload_1     // push the sum computed so far
  iadd        // add them
  istore_1    // store the result back into the running sum
  iinc 2 1    // increment loop counter by 1
  goto loop   // start over again
end:
return
```

Example 2: Code that doesn't verify

```
// loop 5 times. Each time, push local var 0 onto the stack
  iconst_5  // initialize var 0 to 5
  istore0
loop:
  iinc 0 -1    // decrement counter
  iload_0      // push the result on the stack
  dup          // make a copy
  ifeq break // get out of the loop on var value 0
  goto loop  // otherwise keep going
break:
  // more instructions
```

code rejected: Verifier cannot see that it would not cause a stack overflow