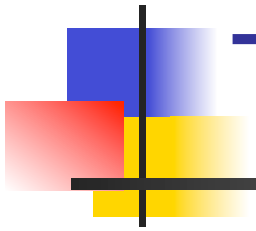
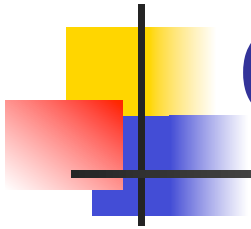


The Java Virtual Machine

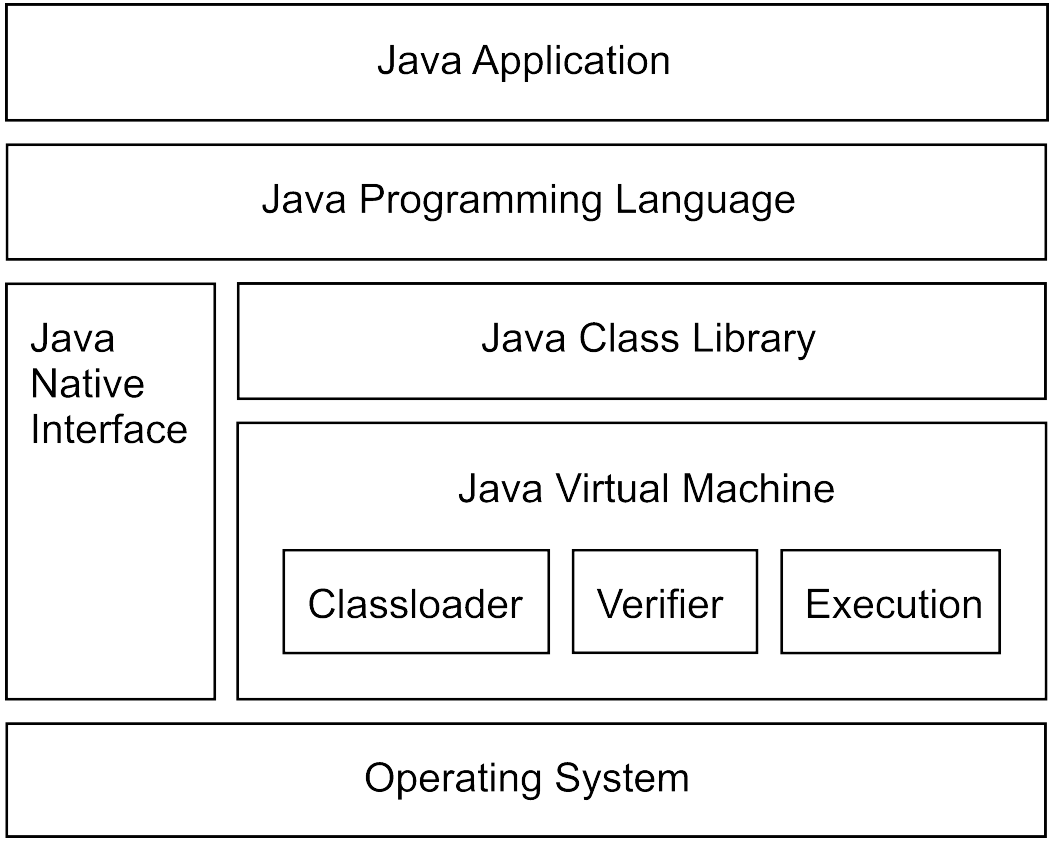
Martin Schöberl

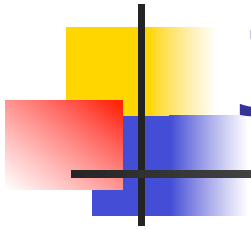


Overview

- Review Java/JVM
- JVM Bytecodes
- Bytecode examples
- Class information
- Parameter passing
- On projects

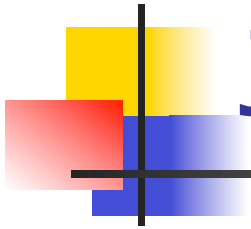
Java System Overview





Java Technology

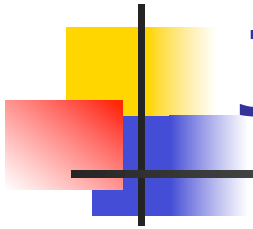
- The Java programming language
- The library (JDK)
- The Java virtual machine (JVM)
  - An instruction set and the meaning of those instructions – the *bytecodes*
  - A binary format – the *class file* format
  - An algorithm to *verify* the class file



JVM Data Types

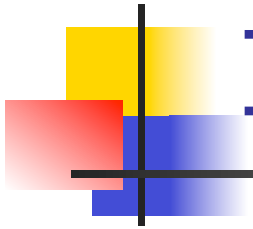
`reference`	Pointer to an object or array
`int`	32-bit integer (signed)
`long`	64-bit integer (signed)
`float`	32-bit floating-point (IEEE 754-1985)
`double`	64-bit floating-point (IEEE 754-1985)

- No `boolean`, `char`, `byte`, and `short` types
  - Stack contains only 32-bit and 64-bit data
  - Conversion instructions



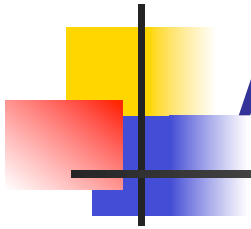
JVM Instruction Set

- The *Bytecodes*
- Operations on the operand stack
- Variable length
- Simple, e.g. `iadd`
- Complex, e.g. `new`
- Symbolic references
- 201 different instructions



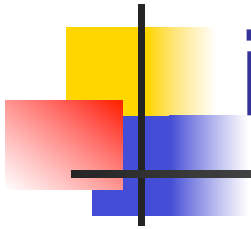
Instruction Types

- Arithmetic
- Load and store
- Type conversion
- Object creation and manipulation
- Operand stack manipulation
- Control transfer
- Method invocation and return



Arithmetic Instructions

- Operate on the values from the stack
- Push the result back onto the stack
- Instructions for `int`, `long`, `float` and `double`
- No direct support for `byte`, `short` or `char` types
  - Handled by `int` operations and type conversion



iadd

Operation Add int

Format *iadd*

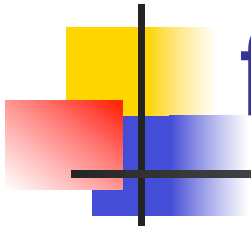
Forms *iadd* = 96 (0x60)

Operand Stack ..., *value1, value2* => ..., *result*

Both *value1* and *value2* must be of type int. The values are popped from the operand stack. The int *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type int. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a runtime exception.



fadd

Operation Add float

Format *fadd*

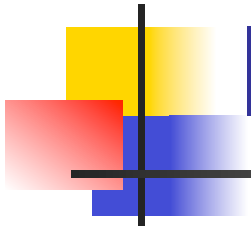
Forms *fadd* = 98 (0x62)

Operand Stack ..., *value1, value2* => ..., *result*

Both *value1* and *value2* must be of type float. The values are popped from the operand stack and undergo value set conversion, resulting in *value1'* and *value2'*. The float *result* is *value1'* + *value2'*. The *result* is pushed onto the operand stack.

The result of an *fadd* instruction is governed by the rules of IEEE arithmetic.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an *fadd* instruction never throws a runtime exception.



ladd

Operation Add long

Format *ladd*

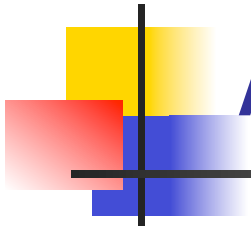
Forms *ladd* = 97 (0x61)

Operand Stack ..., *value1*, *value2* ..., *result*

Both *value1* and *value2* must be of type long. The values are popped from the operand stack. The long *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

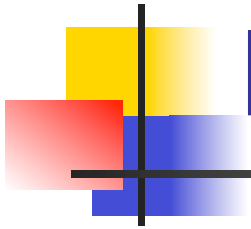
The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type long. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *ladd* instruction never throws a runtime exception.



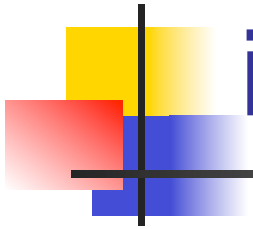
Arithmetic Instructions

- Add: *iadd*, *ladd*, *fadd*, *dadd*
- Subtract: *isub*, *lsub*, *fsub*, *dsub*
- Multiply: *imul*, *lmul*, *fmul*, *dmul*
- Divide: *idiv*, *ldiv*, *fdiv*, *ddiv*
- Remainder: *irem*, *lrem*, *frem*, *drem*
- Negate: *ineg*, *lneg*, *fneg*, *dneg*
- Shift: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*
- Bitwise OR: *ior*, *lor*
- Bitwise AND: *iand*, *land*
- Bitwise exclusive OR: *ixor*, *lxor*
- Local variable increment: *iinc*
- Comparison: *dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*



Load and Store Instructions

- Load
  - Push value from local variable onto stack
  - Push a constant onto the stack
- Store
  - Transfer value from the stack to a local variable
- Typed instructions
- Short versions



iload

Operation Load int from local variable

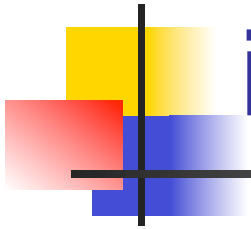
Format *iload*
index

Forms *iload* = 21 (0x15)

Operand Stack ... => ..., *value*

The *index* is an unsigned byte that must be an index into the local variable array of the current frame. The local variable at *index* must contain an int. The *value* of the local variable at *index* is pushed onto the operand stack.

The *iload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.



iload_<n>

Operation Load int from local variable

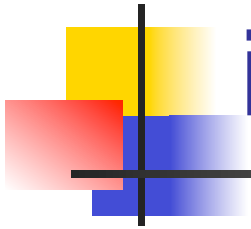
Format *iload_<n>*

Forms *iload_0* = 26 (0x1a)
iload_1 = 27 (0x1b)
iload_2 = 28 (0x1c)
iload_3 = 29 (0x1d)

Operand Stack ... => ..., *value*

The *<n>* must be an index into the local variable array of the current frame. The local variable at *<n>* must contain an int. The *value* of the local variable at *<n>* is pushed onto the operand stack.

Each of the *iload_<n>* instructions is the same as *iload* with an *index* of *<n>*, except that the operand *<n>* is implicit.



istore

Operation Store int into local variable

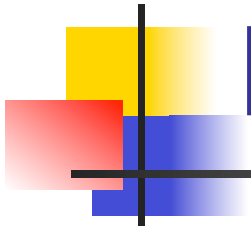
Format *istore*
index

Forms *istore* = 54 (0x36)

Operand Stack ..., *value* => ...

The *index* is an unsigned byte that must be an index into the local variable array of the current frame. The *value* on the top of the operand stack must be of type int. It is popped from the operand stack, and the value of the local variable at *index* is set to *value*.

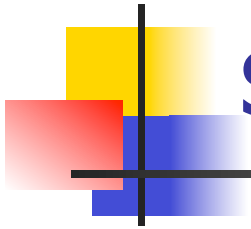
The *istore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.



bipush

Operation	Push byte
Format	*bipush* *byte*
Forms	*bipush* = 16 (0x10)
Operand Stack	... => ..., *value*

The immediate *byte* is sign-extended to an int *value*. That *value* is pushed onto the operand stack.



sipush

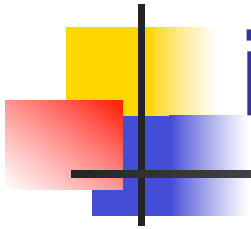
Operation Push short

Format *sipush*
byte1
byte2

Forms *sipush* = 17 (0x11)

Operand Stack ... => ..., *value*

The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate short where the value of the short is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an int *value*. That *value* is pushed onto the operand stack.

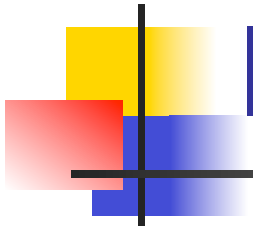


iconst_<i>

Operation	Push int constant
Format	*iconst_<i>*
Forms	*iconst_m1* = 2 (0x2) *iconst_0* = 3 (0x3) *iconst_1* = 4 (0x4) ... *iconst_5* = 8 (0x8)
Operand Stack	... => ..., *<i>*

Push the int constant *<i>* (*-1*, *0*, *1*, *2*, *3*, *4* or *5*) onto the operand stack.

Each of this family of instructions is equivalent to *bipush <i>* for the respective value of *<i>*, except that the operand *<i>* is implicit.



ldc

Operation Push item from runtime constant pool

Format *ldc*
index

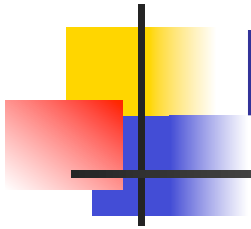
Forms *ldc* = 18 (0x12)

Operand Stack ... => ..., *value*

The *index* is an unsigned byte that must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at *index* either must be a runtime constant of type int or float, or must be a symbolic reference to a string literal.

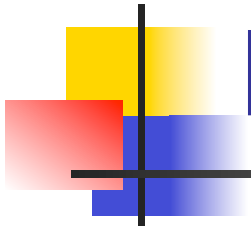
If the runtime constant pool entry is a runtime constant of type int or float, the numeric *value* of that runtime constant is pushed onto the operand stack as an int or float, respectively.

Otherwise, the runtime constant pool entry must be a reference to an instance of class String representing a string literal. A reference to that instance, *value*, is pushed onto the operand stack.



Load and Store Instructions

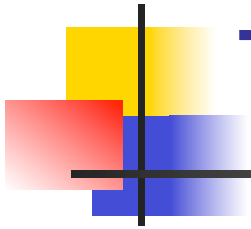
- Load a local variable
  - *iload*, *iload_<n>*, *lload*, *lload_<n>*, *fload*, *fload_<n>*, *dload*, *dload_<n>*, *aload*, *aload_<n>*
- Store a local variable
  - *istore*, *istore_<n>*, *lstore*, *lstore_<n>*, *fstore*, *fstore_<n>*, *dstore*, *dstore_<n>*, *astore*, *astore_<n>*
- Load a constant
  - *bipush*, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *iconst_<i>*, *lconst_<l>*, *fconst_<f>*, *dconst_<d>*
- Wider index, or larger immediate operand
  - *wide*



Load/Add/Store Example

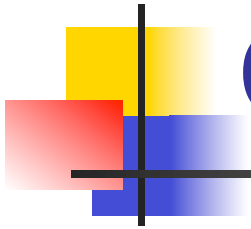
```
int a, b, c;

a = 1;
b = 123;
c = a+b;
```

```
0:  iconst_1
1:  istore_0   // a
2:  bipush 123
4:  istore_1   // b
5:  iload_0    // a
6:  iload_1    // b
7:  iadd
8:  istore_2   // c
```



Type Conversion

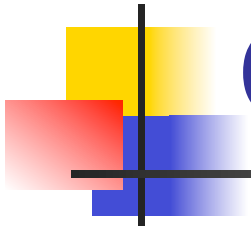
- Widening numeric conversions
  - `int` to `long`, `float`, or `double`
  - `long` to `float` or `double`
  - `float` to `double`
  - *i2l*, *i2f*, *i2d*, *l2f*, *l2d*, and *f2d*
- Narrowing numeric conversions
  - `int` to `byte`, `short`, or `char`
  - `long` to `int`
  - `float` to `int` or `long`
  - `double` to `int`, `long`, or `float`
  - *i2b*, *i2c*, *i2s*, *l2i*, *f2i*, *f2l*, *d2i*, *d2l*, and *d2f*



Conversion Example

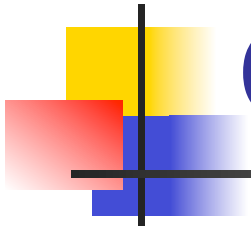
```
short s;
s = 1;
++s;
```

```
0:  iconst_1
1:  istore_0
2:  iload_0
3:  iconst_1
4:  iadd
5:  i2s        // truncate
6:  istore_0
```



Object Instructions

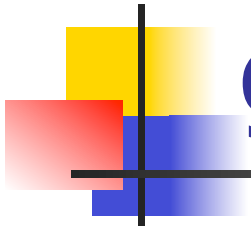
- Create a new class instance or array
  - *new*, *newarray*, *anewarray*, *multianewarray*
- Field access
  - *getfield*, *putfield*, *getstatic*, *putstatic*
- Array load, store
  - *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*
  - *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*
- Length of an array
  - *arraylength*
- Check properties
  - *instanceof*, *checkcast*



Object Creation

```
Object create() {
   return new Object();
}
```

```
0:   new           #2; //class Object
3:   dup
4:   invokespecial #1; //Method java/lang/Object."<init>":()V
7:   areturn
```



getfield

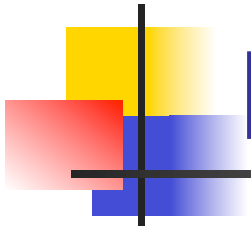
Operation Fetch field from object

Format *getfield*
indexbyte1
indexbyte2

Forms *getfield* = 180 (0xb4)

Operand Stack ..., *objectref* => ..., *value*

The *objectref*, which must be of type reference, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class, where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The runtime constant pool item at that index must be a symbolic reference to a field, which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved. The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.



putfield

Operation Set field in object

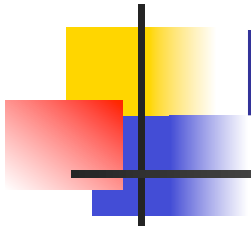
Format *putfield*
indexbyte1
indexbyte2

Forms *putfield* = 181 (0xb5)

Operand Stack ..., *objectref*, *value* => ...

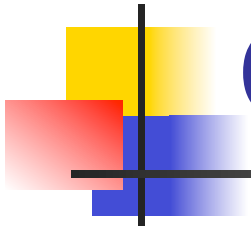
The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class…

The *value* and *objectref* are popped from the operand stack. The *objectref* must be of type reference. The referenced field in *objectref* is set to *value.*



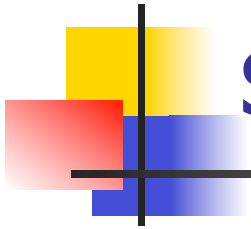
Field Access

```
static int statVal;
private int privVal;

void foo() {

  int i = statVal + privVal;  0:   getstatic  #3; //Field statVal:I
                              3:   aload_0
                              4:   getfield   #4; //Field privVal:I
                              7:   iadd
                              8:   istore_1

  statVal = i;                9:   iload_1
                              10:  putstatic  #3; //Field statVal:I

  privVal = i;                13:  aload_0
                              14:  iload_1
                              15:  putfield   #4; //Field privVal:I

}                             18:  return
```



Operand Stack Manipulation

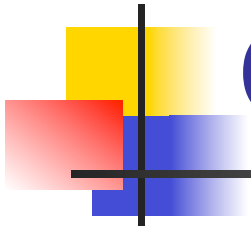
- Direct manipulation of the operand stack
  - *pop*, *pop2*
  - *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2*, *dup2_x2*
  - *swap*



swap

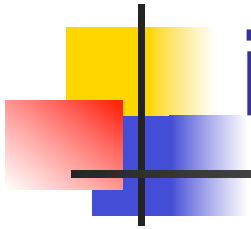
Operation	Swap the top two operand stack values
Format	*swap*
Forms	*swap* = 95 (0x5f)
Operand Stack	..., *value2*, *value1* => ..., *value1*, *value2*

Swap the top two values on the operand stack.



Control Transfer

- Conditional branch
  - *ifeq*, *iflt*, *ifle*, *ifne*, *ifgt*, *ifge*, *ifnull*, *ifnonnull*, *if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmpgt*, *if_icmple*, *if_icmpge*, *if_acmpeq*, *if_acmpne*.
- Switch
  - *tableswitch*, *lookupswitch*.
- Unconditional branch
  - *goto*, *goto_w*, *jsr*, *jsr_w*, *ret*.



if<cond>

Operation Branch if int comparison with zero succeeds

Format
if<cond>
branchbyte1
branchbyte2

Forms
ifeq = 153 (0x99)
...
ifle = 158 (0x9e)

Operand Stack ..., *value* => ...

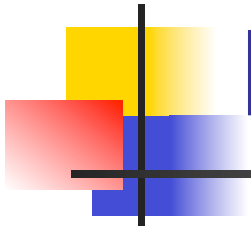
The *value* is popped from the operand stack and compared against zero. All comparisons are signed:

eq succeeds if and only if *value* = 0

...

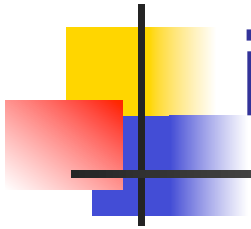
le succeeds if and only if *value* ≤ 0

If the comparison succeeds, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution then proceeds at that offset from the address of the opcode of this *if<cond>* instruction. Otherwise, execution proceeds at the address of the instruction following this *if<cond>* instruction.



Method Invocation, Return

- *invokevirtual*
  - Invokes an instance method of an object, dispatching on the (virtual) type of the object.
  - This is the normal method dispatch in the Java programming language
- *invokeinterface*
  - Invokes a method that is implemented by an interface
- *invokespecial*
  - Invokes an instance method requiring special handling
  - Instance initialization method, a private method, or a superclass method
- *invokestatic*
  - Invokes a class (static) method



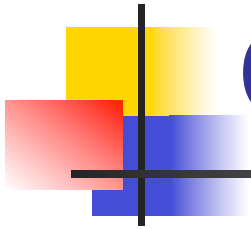
invokevirtual

Operation	Invoke instance method; dispatch based on class
Format	*invokevirtual* *indexbyte1* *indexbyte2*
Forms	*invokevirtual* = 182 (0xb6)
Operand Stack	..., *objectref, [arg1, [arg2 ...]]* => ...

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool…

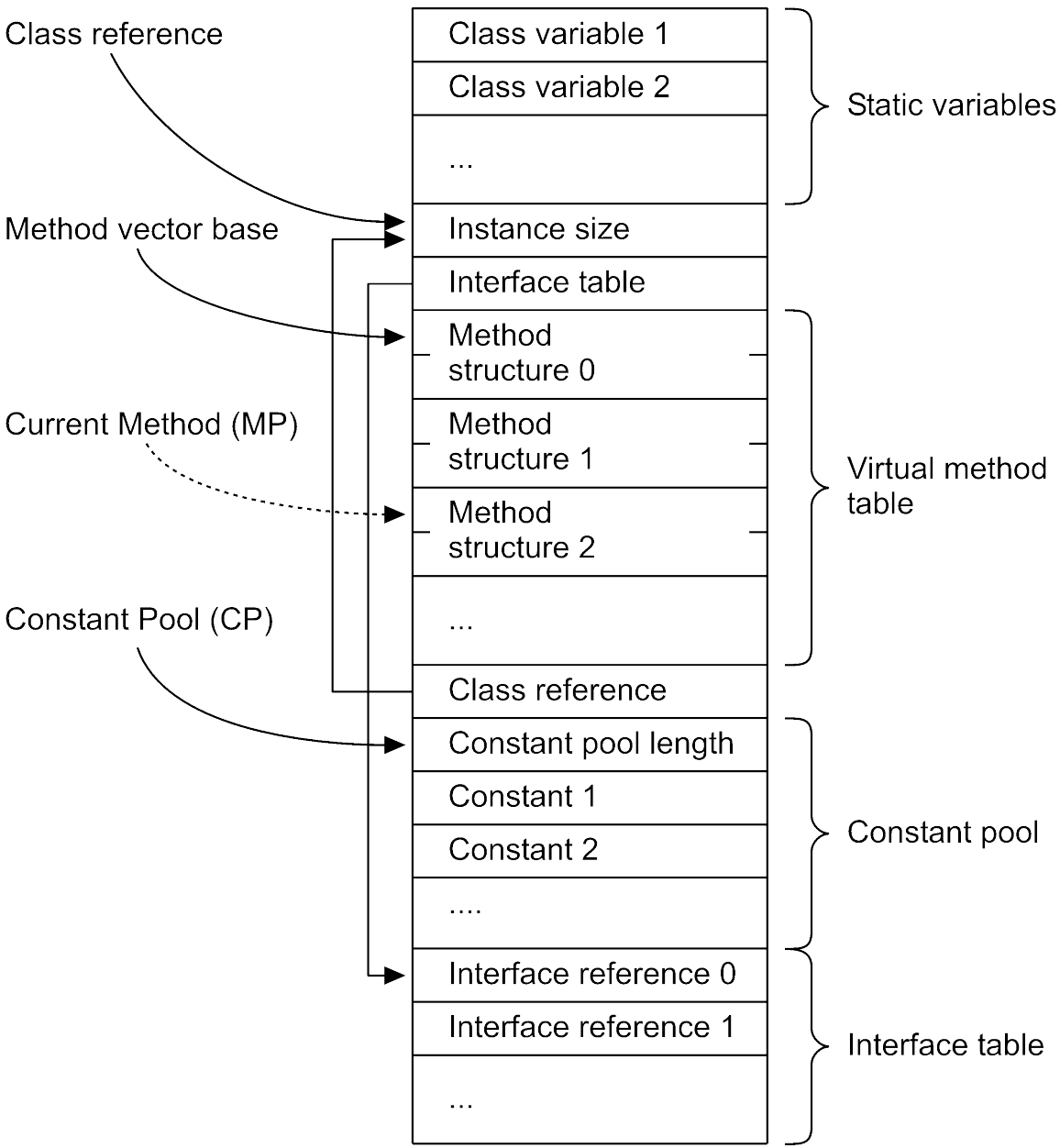
The *objectref* must be followed on the operand stack by *nargs* argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

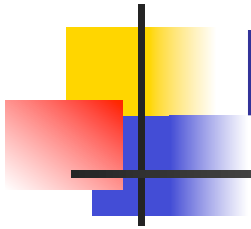
If the method is synchronized, the monitor associated with *objectref* is acquired or reentered.



Class Information

- Instance size
- Static (class) variables
- Virtual method table
- Interface table
- Constant pool
- Reference to super class


Class reference
Method vector base
Current Method (MP)
Constant Pool (CP)
Class variable 1
Class variable 2
...
Instance size
Interface table
Method structure 0
Method structure 1
Method structure 2
...
Class reference
Constant pool length
Constant 1
Constant 2
....
Interface reference 0
Interface reference 1
...
Static variables
Virtual method table
Constant pool
Interface table

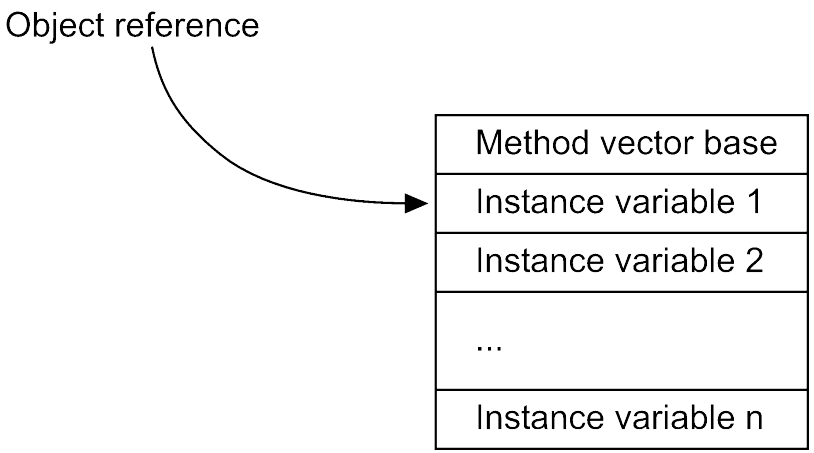


Method Structure

Start address	Method length	
Constant pool	Local count	Arg. count

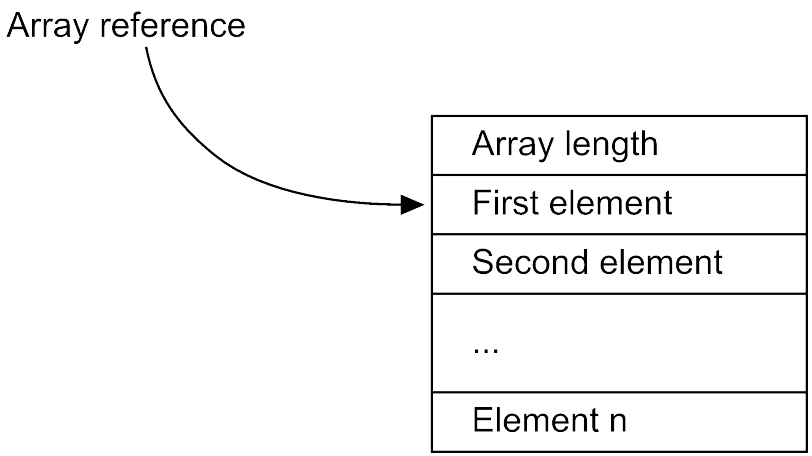
- Information about a method
  - Address
  - Length (for the cache)
  - Pointer to the constant pool of the class
  - Number of arguments and local variables

Object Format

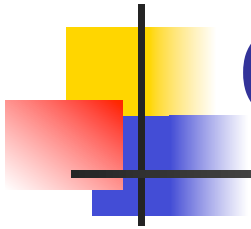


- Direct pointer
- Handle possible
- Return pointer to the class information

Array Format

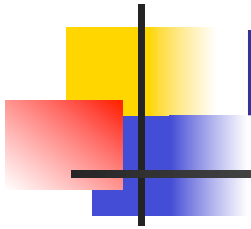


- Direct pointer
- Handle possible
- Length is needed



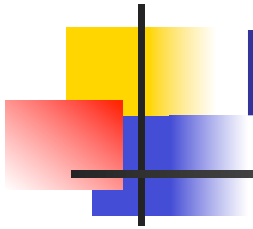
Constant Pool

- Contains:
  - Simple constants (e.g. 123, 0.345)
  - String constants
  - Class references
  - Field references
  - Method references
- All references are symbolic in the class file
- References can and should be converted to direct pointers



Runtime Data Structures

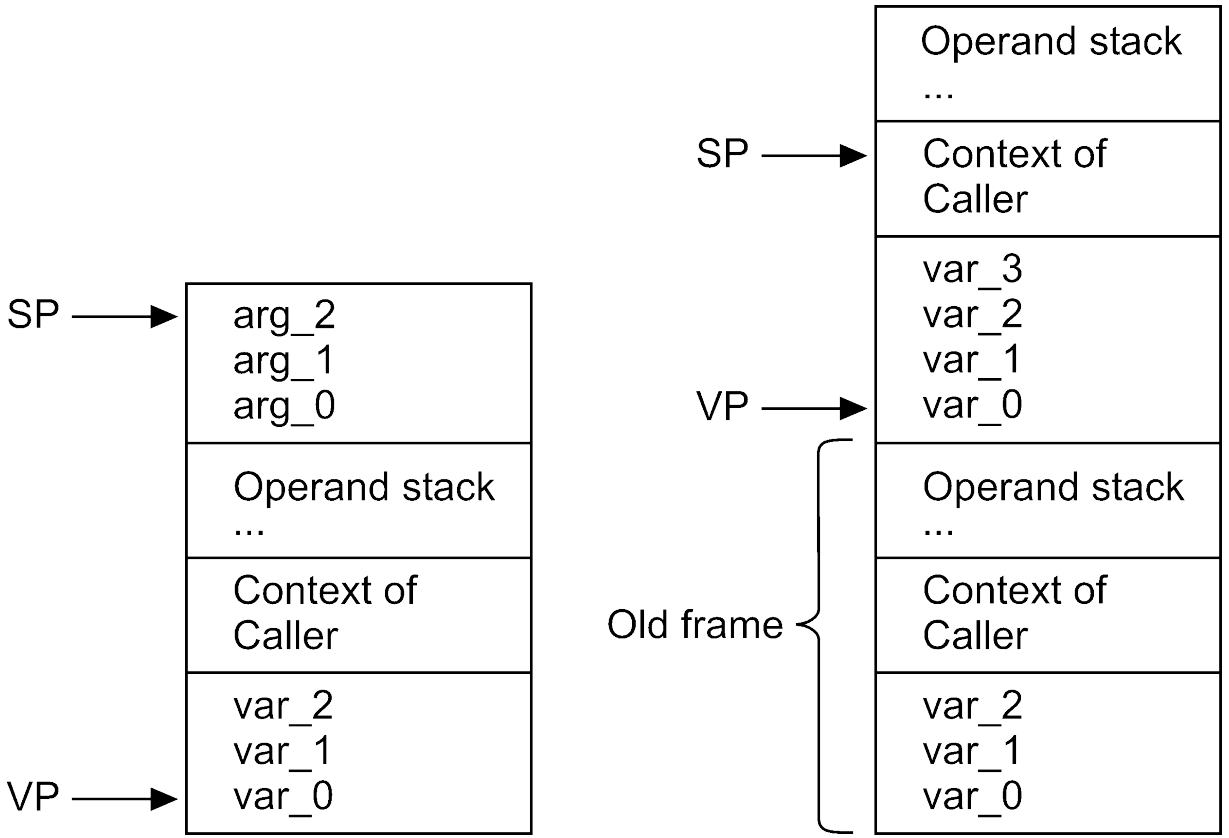
- PC – program counter
- Operand stack
  - SP – stack pointer
  - VP – variable pointer
- MP – method pointer
  - Reference to the method structure
- CP – constant pool
  - Current constant pool

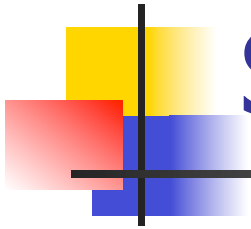


Parameter passing

```
    int val = foo(1, 2);
    ...
    public int foo(int a, int b) {
        int c = 1;
        return a+b+c;
    }

The invocation sequence:
    aload_0             // Push the object reference
    iconst_1            // and the parameter onto the
    iconst_2            // operand stack.
    invokevirtual   #2  // Invoke method foo:(II)I.
    istore_1            // Store the result in val.

public int foo(int,int):
    iconst_1            // The constant is stored in a method
    istore_3            // local variable (at position 3).
    iload_1             // Arguments are accessed as locals
    iload_2             // and pushed onto the operand stack.
    iadd                // Operation on the operand stack.
    iload_3             // Push c onto the operand stack.
    iadd
    ireturn             // Return value is on top of stack.
```

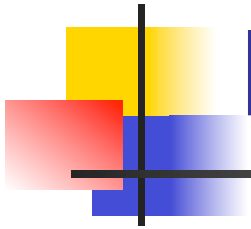
Stack on Method Invocation





Summary

- The JVM defines an instruction set – *Bytecodes*
- Simple, typed stack instructions
- Complex, such as object creation
- Implementation details are not defined
- Method invocation *suggests* a common stack



More Information

- JOP Thesis: p 7-16, p 55-64, p 78-82
  - Or in the Handbook
- Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999, JVMSpec.