Java ByteCode

Manuel Oriol

June 7th, 2007

Chair of
Software Engineering

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Byte Code?

- The Java language is compiled into an intermediary form called byte code
- It ensures portability
- The byte code is a succession of instructions that manipulate a stack
- Each method invocation is having a stack
- Is compiled natively upon use

Java Class Files

- Constant Pool

  (around 60% because is Strings)
- Access rights
- Fields
- Methods

  (around 12%)
- Class Attributes

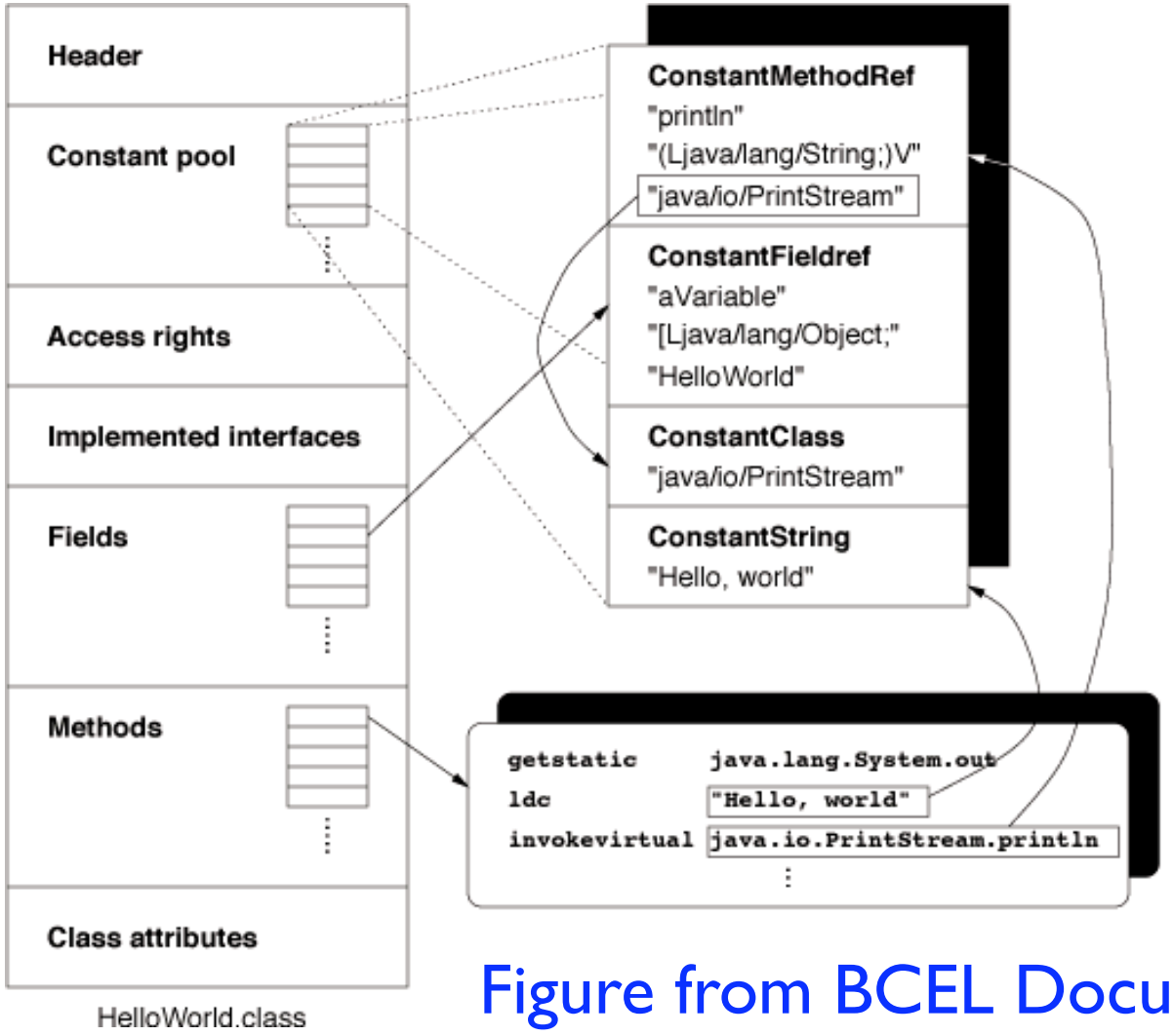
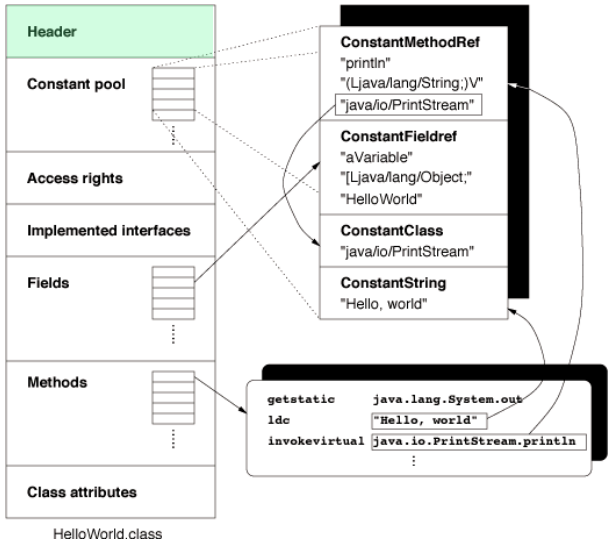


Figure from BCEL Documentation

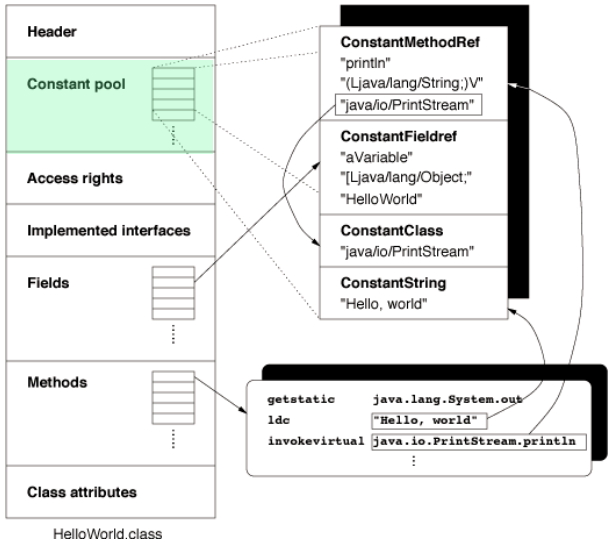
http://jakarta.apache.org/bcel/images/classfile.gif

Header



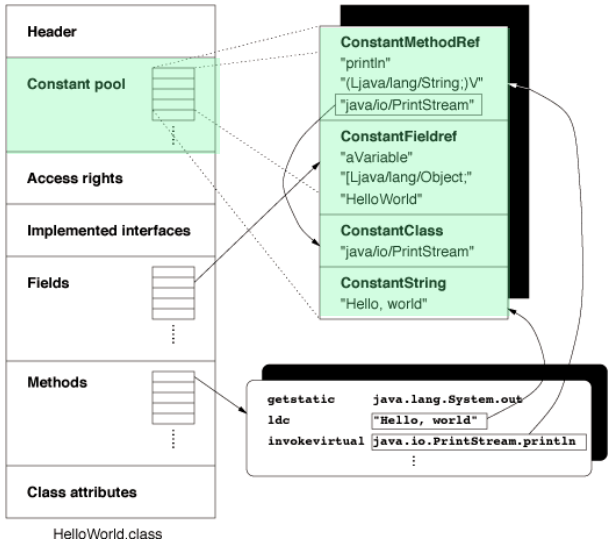
- Magic Number
- minor version number of class file format
- major version number of class file format (49 at the moment!)

Constant Pool



- constant pool count
- table of constants

Content

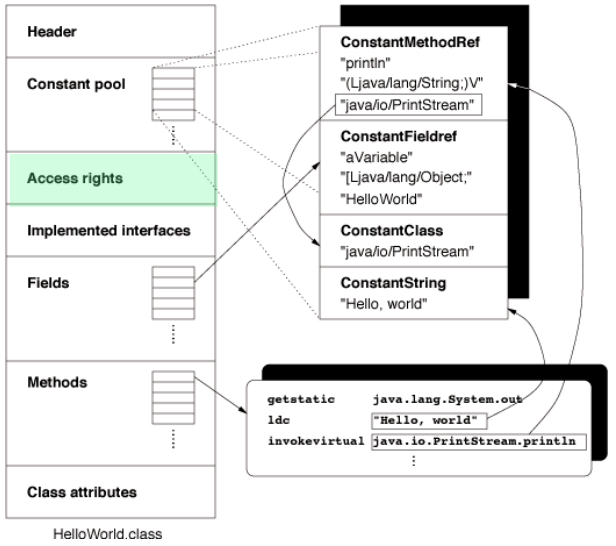


- table of constants of the form:

tag_byte info[]

CONSTANT_Class 7
CONSTANT_Fieldref 9
CONSTANT_Methodref 10
CONSTANT_InterfaceMethodref 11
CONSTANT_String 8
CONSTANT_Integer 3
CONSTANT_Float 4
CONSTANT_Long 5
CONSTANT_Double 6
CONSTANT_NameAndType 12
CONSTANT_Utf8 1

Access Rights



- ACC_PUBLIC 0x0001

  Declared public; may be accessed from outside its package.

- ACC_FINAL 0x0010 Declared final; no subclasses allowed.

- ACC_SUPER 0x0020 Treat superclass methods specially when invoked by the invokespecial instruction.

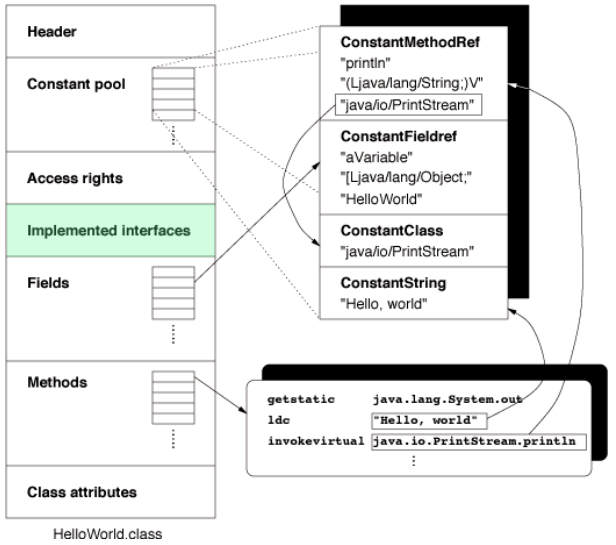
- ACC_INTERFACE 0x0200 Is an interface, not a class.

- ACC_ABSTRACT 0x0400 Declared abstract; may not be instantiated.

this_class, super_class

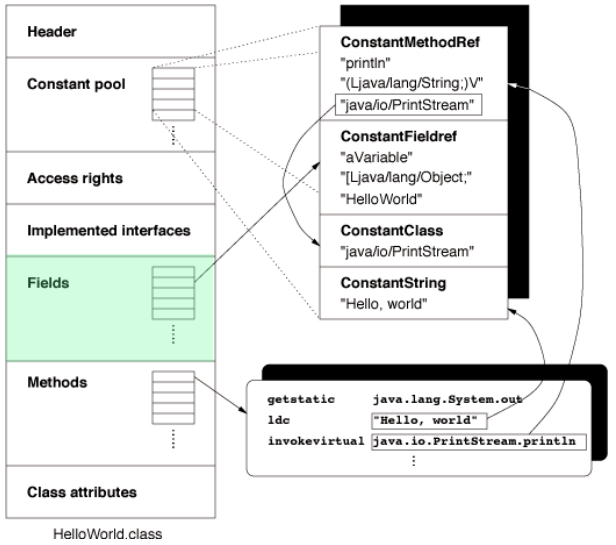
- index in the constant pool to a class
- index in the constant pool to super class

Implemented Interfaces



- interfaces_count
- interfaces[] points to values in the constant pool

Fields



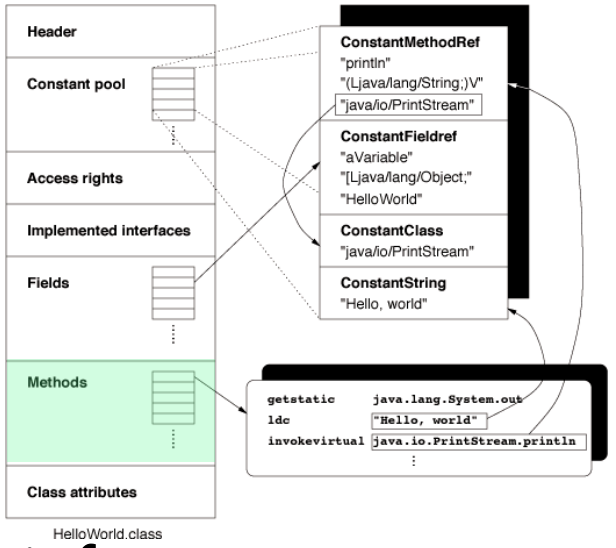
- fields_count
- fields[] points to field_infos

```
field_info {
 u2 access_flags;
 u2 name_index;
 u2 descriptor_index;
 u2 attributes_count;
 attribute_info attributes[attributes_count];
}
```

Constant Pool

synthetic, deprecated, Constant value

Methods



- methods_count
- methods[] stores methods infos

```
method_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Constant Pool

code, exceptions, synthetic, deprecated

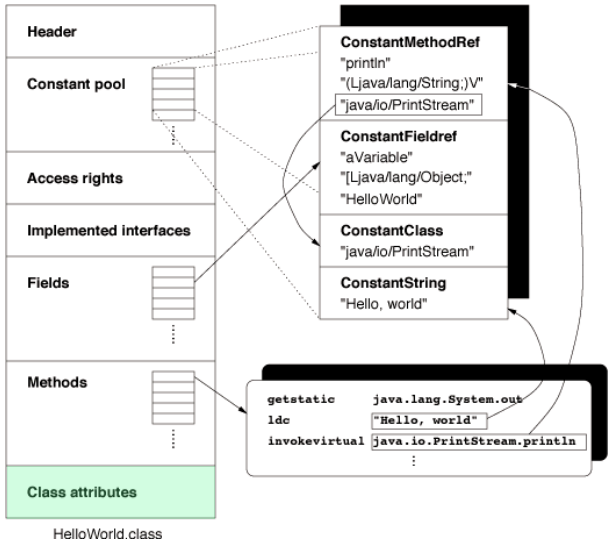
Code attribute

```
Code_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  {    u2 start_pc;
            u2 end_pc;
            u2  handler_pc;
            u2  catch_type;
  }    exception_table[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Points to code

LineNumberTable, LocalVariablesTable

Class Attributes



- attributes_count
- attributes[] may contain only source file or deprecated attributes

Constants in the constant pool

- They are used for almost everything, from fields to external classes to call etc...
- They have a compact encoding

Internal Representation: Field Descriptor

- B byte signed byte
- C char Unicode character
- D double double-precision floating-point value
- F float single-precision floating-point value
- I int integer
- J long long integer
- L<classname>; reference an instance of class <classname> (full path with /)
- S short signed short
- Z boolean true or false
- [reference one array dimension

Internal Representation: Method Descriptor

A method descriptor represents the parameters that the method takes and the value that it returns:

MethodDescriptor:
(ParameterDescriptor*) ReturnDescriptor

A parameter descriptor represents a parameter passed to a method:

ParameterDescriptor:
FieldType

A return descriptor represents the type of the value returned from a method. It is a series of characters generated by the grammar:

ReturnDescriptor:
FieldType
V

Examples

- int[][] -> [[I
- Thread [] -> [java/lang/Thread;
- Object mymethod(int i, double d, Thread t)

  -> (IDLjava/lang/Thread;)Ljava/lang/Object;

Constant pool entries

- There are constants:

CONSTANT_Class 7
CONSTANT_Fieldref 9
CONSTANT_Methodref 10
CONSTANT_InterfaceMethodref 11
CONSTANT_String 8
CONSTANT_Integer 3
CONSTANT_Float 4
CONSTANT_Long 5
CONSTANT_Double 6
CONSTANT_NameAndType 12
CONSTANT_Utf8 1

```
cp_info {
 u1 tag;
 u1 info[];
}
```

Constant Pool info (1/2)

```
CONSTANT_Class_info {
 u1 tag;
 u2 name_index;
}
CONSTANT_Fieldref_info {
 u1 tag;
 u2 class_index;
 u2 name_and_type_index;
}
CONSTANT_Methodref_info {
 u1 tag;
 u2 class_index;
 u2 name_and_type_index;
}
CONSTANT_InterfaceMethodref_info {
 u1 tag;
 u2 class_index;
 u2 name_and_type_index;
}
```

...Field Descriptor

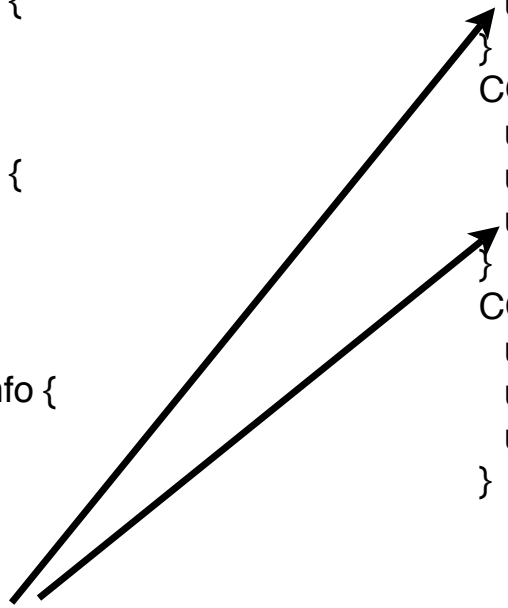
...Method Descriptor

...Method Descriptor

Constant Pool info (2/2)

```
CONSTANT_Integer_info {
  u1 tag;
  u4 bytes;
}
CONSTANT_Float_info {
  u1 tag;
  u4 bytes;
}
CONSTANT_Long_info {
  u1 tag;
  u4 high_bytes;
  u4 low_bytes;
}
CONSTANT_Double_info {
  u1 tag;
  u4 high_bytes;
  u4 low_bytes;
}
```

```
CONSTANT_String_info {
  u1 tag;
  u2 string_index;
}
CONSTANT_NameAndType_info {
  u1 tag;
  u2 name_index;
  u2 descriptor_index;
}
CONSTANT_Utf8_info {
  u1 tag;
  u2 length;
  u1 bytes[length];
}
```



Points to Utf8_info

VM Instruction set

- mnemonic operand1 operand2 ...
- Important: each method call has its own stack

Byte Code Instruction Set: 212 instuctions

- Stack Operations
- Primitive types operations
- Arrays operations
- Object-related instructions
- Control Flow
- Invocations
- Load and Store operations
- Special instructions

Stack Operations

- The usual: pop, pop2, dup, dup2, swap
- a bit specific:

  dup_x1, dup_x2, dup2_x1, dup2_x2

Primitive Types Operations

- each primitive type has a letter:

  b (boolean & byte), c, d, f, i, l, s

- Pushing values:

  sipush, bipush, dconst_0, dconst_1, fconst_0,... fconst_2, iconst_0,..., iconst_5, lconst_0, lconst_1, sipush

- Conversions:

  d2f, d2i, d2l, f2d, f2i, f2l, i2b, i2c, i2d, i2f, i2l, i2s

- Operations: dadd, ddiv, drem, dmul, dneg (same with f, i, l: fadd, fdiv...), dcmpg, dcmpl (f,l) (makes comparisons), iand, ior, ishl, ishr, iashr, ixor (also with l)

iadd

Operation

Add `int`

Format

iadd

Forms

iadd = 96 (0x60)

Operand Stack

..., *value1*, *value2* ⇒ ..., *result*

Description

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.

Arrays Operations

3 main types of operations:

- Load: baload, caload, daload, faload, iaload, laload, saload
- Store: bastore, castore, dastore, fastore, iastore, lastore, sastore
- Utilities: newarray, anewarray, multinewarray, arraylength

iaload

Operation

Load `int` from array

Format

iaload

Forms

iaload = 46 (0x2e)

Operand Stack

..., *arrayref*, *index* ⇒ ..., *value*

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `int`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The `int` *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *iaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *iaload* instruction throws an `ArrayIndexOutOfBoundsException`.

Objects-Related Operations

- Fields Manipulation:

  getfield, putfield, getstatic, putstatic

- Critical Sections:

  monitorenter, monitorexit

- Stack Manipulations:

  new, aconst_null

getfield

Operation

Fetch field from object

Format

getfield
indexbyte1
indexbyte2

Forms

getfield = 180 (0xb4)

Operand Stack

..., *objectref* ⇒ ..., *value*

Description

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 (§3.6), 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 (§5.1), 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 (§5.4.3.2). The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.

The class of *objectref* must not be an array. If the field is `protected` (§4.6), and it is either a member of the current class or a member of a superclass of the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Linking Exceptions

During resolution of the symbolic reference to the field, any of the errors pertaining to field resolution documented in Section 5.4.3.2 can be thrown.

Otherwise, if the resolved field is a `static` field, *getfield* throws an `IncompatibleClassChangeError`.

Runtime Exception

Otherwise, if *objectref* is `null`, the *getfield* instruction throws a `NullPointerException`.

Notes

The *getfield* instruction cannot be used to access the `length` field of an array. The *arraylength* instruction is used instead.

Invocations

- invokestatic: for static methods
- invokeinterface: for interface methods
- invokespecial: instance initialization or private methods
- invokevirtual: regular method invocation
- return: returns void
- dreturn, freturn, ireturn, lreturn, areturn

invokevirtual

Operation

Invoke instance method; dispatch based on class

Format

invokevirtual
indexbyte1
indexbyte2

Forms

invokevirtual = 182 (0xb6)

Operand Stack

..., *objectref*, [*arg1*, [*arg2* ...]] ⇒ ...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the runtime constant pool of the current class (§3.6), 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 method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The method must not be an instance initialization method (§3.9) or the class or interface initialization method (§3.9). Finally, if the resolved method is `protected` (§4.6), and it is either a member of the current class or a member of a superclass of the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Let C be the class of *objectref*. The actual method to be invoked is selected by the following lookup procedure:

- If C contains a declaration for an instance method with the same name and descriptor as the resolved method, and the resolved method is accessible from C, then this is the method to be invoked, and the lookup procedure terminates.
- Otherwise, if C has a superclass, this same lookup procedure is performed recursively using the direct superclass of C ; the method to be invoked is the result of the recursive invocation of this lookup procedure.
- Otherwise, an `AbstractMethodError` is raised.

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.

If the method is not `native`, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable *0*, *arg1* in local variable *1* (or, if *arg1* is of type `long` or `double`, in local variables *1* and *2*), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is `native` and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the `native` method is `synchronized`, the monitor associated with *objectref* is released or exited as if by execution of a *monitorexit* instruction.
- If the `native` method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the `native` method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in Section 5.4.3.3 can be thrown.

Otherwise, if the resolved method is a class (`static`) method, the *invokevirtual* instruction throws an `IncompatibleClassChangeError`.

Runtime Exceptions

Otherwise, if *objectref* is `null`, the *invokevirtual* instruction throws a `NullPointerException`.

Otherwise, if no method matching the resolved name and descriptor is selected, *invokevirtual* throws an `AbstractMethodError`.

Otherwise, if the selected method is `abstract`, *invokevirtual* throws an `AbstractMethodError`.

Otherwise, if the selected method is `native` and the code that implements the method cannot be bound, *invokevirtual* throws an `UnsatisfiedLinkError`.

Notes

The *nargs* argument values and *objectref* are not one-to-one with the first *nargs* + *1* local variables. Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than *nargs* local variables may be required to pass *nargs* argument values to the invoked method.

Method Frame

- A frame is created for each method invocation and its local variables are stored in an array (size determined at compile-time)
- A frame is destroyed when the method returns

Each frame (§3.6) contains an array of variables known as its *local variables*. The length of the local variable array of a frame is determined at compile time and supplied in the binary representation of a class or interface along with the code for the method associated with the frame (§4.7.3).

A single local variable can hold a value of type `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference`, or `returnAddress`. A pair of local variables can hold a value of type `long` or `double`.

Local variables are addressed by indexing. The index of the first local variable is zero. An integer is be considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array.

A value of type `long` or type `double` occupies two consecutive local variables. Such a value may only be addressed using the lesser index. For example, a value of type `double` stored in the local variable array at index n actually occupies the local variables with indices n and $n+1$; however, the local variable at index $n+1$ cannot be loaded from. It can be stored into. However, doing so invalidates the contents of local variable n.

The Java virtual machine does not require n to be even. In intuitive terms, values of types `double` and `long` need not be 64-bit aligned in the local variables array. Implementors are free to decide the appropriate way to represent such values using the two local variables reserved for the value.

The Java virtual machine uses local variables to pass parameters on method invocation. On class method invocation any parameters are passed in consecutive local variables starting from local variable *0*. On instance method invocation, local variable *0* is always used to pass a reference to the object on which the instance method is being invoked (`this` in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable *1*.

Load and Store

- incrementing an int local variable: iinc
- Loading from a local variable:

  aload, aload_0, ..., aload_3, (same with d, i, f, l)
- Storing in a local variable:

  astore, astore_0, ..., aload_3, (same with d, i, f, l)
- Loading from Constant Pool:

  ldc, ldc_w, ldc2_w

lload

Operation

Load `long` from local variable

Format

lload
index

Forms

lload = 22 (0x16)

Operand Stack

... ⇒ ..., *value*

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be indices into the local variable array of the current frame (§3.6). The local variable at *index* must contain a `long`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes

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

Control Flow

- goto, goto_w: go to an instruction
- jsr, jsr_w: jump to subroutine and pushes return address on the stack
- ret: returns from a subroutine using address in a local variable
- ifeq, ifne, iflt, ifle, ifgt, ifge, if_acmpeq, if_acmpne, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmple, if_icmpgt, if_icmpge (same with d, f, l): if branches
- tableswitch, lookupswitch: switch and hashsets

goto_w

Operation

Branch always (wide index)

Format

goto_w
branchbyte1
branchbyte2
branchbyte3
branchbyte4

Forms

goto_w = 200 (0xc8)

Operand Stack

No change

Description

The unsigned bytes *branchbyte1*, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 24) | (*branchbyte2* << 16) | (*branchbyte3* << 8) | *branchbyte4*. Execution proceeds at that offset from the address of the opcode of this *goto_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto_w* instruction.

Notes

Although the *goto_w* instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes (§4.10). This limit may be raised in a future release of the Java virtual machine.

Special Instructions

- No operation: nop
- throw exception: athrow
- verifying instances: instanceof
- checking a cast operation: checkcast

instanceof

Operation

Determine if object is of given type

Format

instanceof
indexbyte1
indexbyte2

Forms

instanceof = 193 (0xc1)

Operand Stack

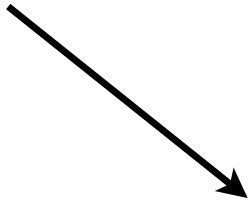
..., *objectref* ⇒ ..., *result*

javap -c: a de-assembler

- javap is a class disassembler, by default it prints only the public interface
- -c prints the code of the methods
- -l prints the code with local variables
- -private show all variables and classes
- -s displays internal type signature

Example 1 (/3)

```
public static int test1(){
 return 2;
}
```



```
public static int test1();
  Signature: ()I
  Code:
   0:   iconst_2
   1:   ireturn
```

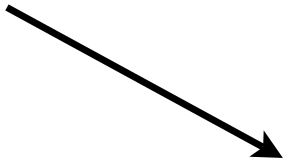
Example 2 (/3)

```
public int test3(int b){
    int j=0;
    for  (int i=0;i<10;i++){
     j=j+i;
    }
    return j;
}
```

```
public int test3(int);
  Signature: (I)I
  Code:
   0:   iconst_0
   1:   istore_2
   2:   iconst_0
   3:   istore_3
   4:   iload_3
   5:   bipush   10
   7:   if_icmpge        20
  10:  iload_2
  11:  iload_3
  12:  iadd
  13:  istore_2
  14:  iinc     3, 1
  17:  goto     4
  20:  iload_2
  21:  ireturn
```

Example 3 (/3)

```
public static void main(String []args){
   Example e=new Example();
   int b;
   test1();
   b=e.test2(2);
   e.test3(b);
}
```



```
public static void main(java.lang.String[]);
  Signature: ([Ljava/lang/String;)V
  Code:
   0:  new     #2; //class Example
   3:  dup
   4:  invokespecial   #3; //Method "<init>":()V
   7:  astore_1
   8:  invokestatic    #4; //Method test1:()I
   11: pop
   12: aload_1
   13: iconst_2
   14: invokevirtual   #5; //Method test2:(I)I
   17: istore_2
   18: aload_1
   19: iload_2
   20: invokevirtual   #6; //Method test3:(I)I
   23: pop
   24: return
```

Try it yourself...

```
public int test2(int a){
   a=a+1;
   return a;
}
```

```
public int test2(int);
  Signature: (I)I
  Code:
   0:   iload_1
   1:   iconst_1
   2:   iadd
   3:   istore_1
   4:   iload_1
   5:   ireturn
```

Optimize it!!!

```
public int test2(int);
  Signature: (I)I
  Code:
    0:   iload_1
    1:   iconst_1
    2:   iadd
    3:   istore_1
    4:   iload_1
    5:   ireturn
```

```
public int test2(int);
  Signature: (I)I
  Code:
    1:   iinc 1,1
    2:   iload_1
    3:   ireturn
```

Decompilers?

- What do we loose, what don't we?

Byte Code Engineering Library (BCEL)

- Reification of everything
- Written in Java
- Visitors and Pattern matching on the code
- Custom class loaders ready to use!

References

- http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
- http://jakarta.apache.org/bcel/

Just-in-Time Compiling

Manuel Oriol

June 7th, 2007

Chair of
Software Engineering

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

What is Just-in-Time Compiling

- It is compilation from bytecode to assembly!
- Already existing in other languages

  (eg Smalltalk VisualWorks)
- optimizations are welcome as long as sound

JIT techniques

- inlining/recursion treatments
- loop optimizations

When Triggered?

- Several strategies exist... but the usual way is that it compiles a method the first time it is called (hence the overhead for the first invocation)
- Possible to pre-compile natively everything as well. That's how one can get better performances with Java than with C++