Contents

Preface

This document describes version 1.0 of the Java Virtual Machine and its instruction set. We have written this document to act as a specification for both compiler writers, who wish to target the machine, and as a specification for others who may wish to implement a compliant Java Virtual Machine.

The Java Virtual Machine is an imaginary machine that is implemented by emulating it in software on a real machine. Code for the Java Virtual Machine is stored in `.class` files, each of which contains the code for at most one `public` class.

Simple and efficient emulations of the Java Virtual Machine are possible because the machine's format is compact and efficient bytecodes. Implementations whose native code speed approximates that of compiled C are also possible, by translating the bytecodes to machine code, although Sun has not released such implementations at this time.

The rest of this document is structured as follows:

- Chapter 1 describes the architecture of the Java Virtual Machine.
- Chapter 2 describes the `.class` file format.
- Chapter 3 describes the bytecodes.
- Appendix A contains some instructions generated internally by Sun's implementation of the Java Virtual Machine. While not strictly part of the specification we describe these here so that this specification can serve as a reference for our implementation. As more implementations of the Java Virtual Machine become available, we may remove Appendix A from future releases.

Sun will license the Java Virtual Machine trademark and logo for use with compliant implementations of this specification. If you are considering constructing your own implementation of the Java Virtual Machine please contact us, at the email address below, so that we can work together to insure 100% compatiblity of your implementation.

Send comments on this specification or questions about implementing the Java Virtual Machine to our electronic mail address: **java@java.sun.com**.

1 Java Virtual Machine Architecture

1.1 Supported Data Types

The virtual machine data types include the basic data types of the Java language:

```
byte                    // 1-byte signed 2's complement integer
short                   // 2-byte signed 2's complement integer
int                     // 4-byte signed 2's complement integer
long                    // 8-byte signed 2's complement integer
float                   // 4-byte IEEE 754 single-precision float
double                  // 8-byte IEEE 754 double-precision float
char                    // 2-byte unsigned Unicode character
```

Nearly all Java type checking is done at compile time. Data of the primitive types shown above need not be tagged by the hardware to allow execution of Java. Instead, the bytecodes that operate on primitive values indicate the types of the operands so that, for example, the `iadd`, `ladd`, `fadd`, and `dadd` instructions each add two numbers, whose types are `int`, `long`, `float`, and `double`, respectively

The virtual machine doesn't have separate instructions for `boolean` types. Intead, integer instructions, including integer returns, are used to operate on `boolean` values; `byte` arrays are used for arrays of `boolean`.

The virtual machine specifies that floating point be done in IEEE 754 format, with support for gradual underflow. Older computer architectures that do not have support for IEEE format may run Java numeric programs very slowly.

Other virtual machine data types include:

```
object                  // 4-byte reference to a Java object
returnAddress           // 4 bytes, used with jsr/ret/jsr_w/ret_w instructions
```

Note: Java arrays are treated as objects.

This specification does not require any particular internal structure for objects. In our implementation an object reference is to a handle, which is a pair of pointers: one to a method table for the object, and the other to the data allocated for the object. Other implementations may use inline caching, rather than method table dispatch; such methods are likely to be faster on hardware that is emerging between now and the year 2000.

Programs represented by Java Virtual Machine bytecodes are expected to maintain proper type discipline and an implementation may refuse to execute a bytecode program that appears to violate such type discipline.

While the Java Virtual Machines would appear to be limited by the bytecode definition to running on a 32-bit address space machine, it is possible to build a version of the Java Virtual Machine that automatically translates the bytecodes into a 64-bit form. A description of this transformation is beyond the scope of this specification.

1.2 Registers

At any point the virtual machine is executing the code of a single method, and the `pc` register contains the address of the next bytecode to be executed.

Each method has memory space allocated for it to hold:

- a set of local variables, referenced by a `vars` register,
- an operand stack, referenced by an `optop` register, and
- a execution environment structure, referenced by a `frame` register.

All of this space can be allocated at once, since the size of the local variables and operand stack are known at compile time, and the size of the execution environment structure is well-known to the interpreter.

All of these registers are 32 bits wide.

1.3 Local Variables

Each Java method uses a fixed-sized set of local variables. They are addressed as word offsets from the `vars` register. Local variables are all 32 bits wide.

Long integers and double precision floats are considered to take up two local variables but are addressed by the index of the first local variable. (For example, a local variable with index *n* containing a double precision float actually occupies storage at indices *n* and *n*+1.) The virtual machine specification does not require 64-bit values in local variables to be 64-bit aligned. Implementors are free to decide the appropriate way to divide long integers and double precision floats into two words.

Instructions are provided to load the values of local variables onto the operand stack and store values from the operand stack into local variables.

1.4 The Operand Stack

The machine instructions all take operands from an operand stack, operate on them, and return results to the stack. We chose a stack organization so that it would be easy to emulate the machine efficiently on machines with few or irregular registers such as the Intel 486.

The operand stack is 32 bits wide. It is used to pass parameters to methods and receive method results, as well as to supply parameters for operations and save operation results.

For example, the `iadd` instruction adds two integers together. It expects that the integers to be added are the top two words on the operand stack, pushed there by previous instructions. Both integers are popped from the stack, added, and their sum pushed back onto the operand stack. Subcomputations may be nested on the operand stack, and result in a single operand that can be used by the nesting computation.

Each primitive data type has specialized instructions that know how to operate on operands of that type. Each operand requires a single location on the stack, except for `long` and `double`, which require two locations.

Operands must be operated on by operators appropriate to their type. It is illegal, for example, to push two `int`s and then treat them as a `long`. This restriction is enforced, in the Sun implementation, by the bytecode verifier. However, a small number of operations (the `dup` opcodes and `swap`) operate on runtime data areas as raw values of a given width without regard to type.

In our description of the virtual machine instructions below, the effect of an instruction's execution on the operand stack is represented textually, with the stack growing from left to right, and each 32-bit word separately represented. Thus:

Stack: ..., *value1*, *value2* ⇒ ..., *value3*

shows an operation that begins by having *value2* on top of the stack with *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the stack and replaced by *value3*, which has been calculated by the instruction. The remainder of the stack, represented by an ellipsis, is unaffected by the instruction's execution.

The types `long` and `double` take two 32-bit words on the operand stack:

Stack: ... ⇒ ..., *value-word1*, *value-word2*

This specification does not say how the two words are selected from the 64-bit `long` or `double` value; it is only necessary that a particular implementation be internally consistent.

1.5 Execution Environment

The information contained in the execution environment is used to do dynamic linking, normal method returns, and exception propagation.

1.5.1 Dynamic Linking

The execution environment contains references to the interpreter symbol table for the current method and current class, in support of dynamic linking of the method code. The class file code for a method refers to methods to be called and variables to be accessed symbolically. Dynamic linking translates these symbolic method calls into actual method calls, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.

This late binding of the methods and variables makes changes in other classes that a method uses less likely to break this code.

1.5.2 Normal Method Returns

If execution of the current method completes normally, then a value is returned to the calling method. This occurs when the calling method executes a return instruction appropriate to the return type.

The execution environment is used in this case to restore the registers of the caller, with the program counter of the caller appropriately incremented to skip the method call instruction. Execution then continues in the calling method's execution environment.

1.5.3 Exception and Error Propagation

An exceptional condition, known in Java as an `Error` or `Exception`, which are subclasses of `Throwable`, may arise in a program because of:

- a dynamic linkage failure, such as a failure to find a needed class file,
- a run-time error, such as a reference through a null pointer,
- an asynchronous event, such as is thrown by `Thread.stop`, from another thread,
- the program using a `throw` statement.

When an exception occurs:

- A list of *catch clauses* associated with the current method is examined. Each `catch` clause describes the instruction range for which it is active, describes the type of exception that it is to handle, and has the address of the code to handle it.
- An exception matches a `catch` clause if the instruction that caused the exception is in the appropriate instruction range, and the exception type is a subtype of the type of exception that the `catch` clause handles. If a matching `catch` clause is found, the system branches to the specified handler. If no handler is found, the process is repeated until all the nested catch clauses of the current method have been exhausted.
- The order of the `catch` clauses in the list is important. The virtual machine execution continues at the first matching catch clause. Because Java code is structured, it is always possible to sort all the exception handlers for one method into a single list that, for any possible program counter value, can be searched in linear order to find the proper (innermost containing applicable) exception handler for an exception occuring at that program counter value.
- If there is no matching `catch` clause then the current method is said to have as its outcome the uncaught exception. The execution state of the method that called this method is restored from the execution environment, and the propagation of the exception continues, as though the exception had just occurred in this caller.

1.5.4 Additional Information

The execution environment may be extended with additional implementation-specific information, such as debugging information.

1.6 Garbage Collected Heap

The Java heap is the runtime data area from which class instances (objects) are allocated. The Java language is designed to be garbage collected — it does not give the programmer the ability to deallocate objects explicitly. Java does not presuppose any particular kind of garbage collection; various algorithms may be used depending on system requirements.

1.7 Method Area

The method area is analogous to the store for compiled code in conventional languages or the text segment in a UNIX process. It stores method code (compiled Java code) and symbol tables. In the current Java implementation, method code is not part of the garbage-collected heap, although this is planned for a future release.

1.8 The Java Instruction Set

An instruction in the Java instruction set consists of a one-byte *opcode* specifying the operation to be performed, and zero or more *operands* supplying parameters or data that will be used by the operation. Many instructions have no operands and consist only of an opcode.

The inner loop of the virtual machine execution is effectively:

```
do {
    fetch an opcode byte
    execute an action depending on the value of the opcode
} while (there is more to do);
```

The number and size of the additional operands is determined by the opcode. If an additional operand is more than one byte in size, then it is stored in *big-endian* order — high order byte first. For example, a 16-bit parameter is stored as two bytes whose value is:

```
first_byte * 256 + second_byte
```

The bytecode instruction stream is only byte-aligned, with the exception being the `tableswitch` and `lookupswitch` instructions, which force alignment to a 4-byte boundary within their instructions.

These decisions keep the virtual machine code for a compiled Java program compact and reflect a conscious bias in favor of compactness at some possible cost in performance.

1.9 Limitations

The per-class constant pool has a maximum of 65535 entries. This acts as an internal limit on the total complexity of a single class.

The amount of code per method is limited to 65535 bytes by the sizes of the indices in the code in the exception table, the line number table, and the local variable table. This may be fixed for 1.0beta2.

Besides this limit, the only other limitation of note is that the number of words of arguments in a method call is limited to 255.

2 Class File Format

This chapter documents the Java class (`.class`) file format.

Each class file contains the compiled version of either a Java class or a Java interface. Compliant Java interpreters must be capable of dealing with all class files that conform to the following specification.

A Java class file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two or four 8-bit bytes, respectively. The bytes are joined together in network (big-endian) order, where the high bytes come first. This format is supported by the Java `java.io.DataInput` and `java.io.DataOutput` interfaces, and classes such as `java.io.DataInputStream` and `java.io.DataOutputStream`.

The class file format is described here using a structure notation. Successive fields in the structure appear in the external representation without padding or alignment. Variable size arrays, often of variable sized elements are called tables and are commonplace in these structures.

The types `u1`, `u2`, and `u4` mean an unsigned one-, two-, or four-byte quantity, respectively, which are read by method such as `readUnsignedByte`, `readUnsignedShort` and `readInt` of the java.io.`DataInput` interface.

2.1 Format

The following pseudo-structure gives a top-level description of the format of a class file:

```
ClassFile {
   u4 magic;
   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[attribute_count];
}
```

`magic`

This field must have the value 0xCAFEBABE.

`minor_version, major_version`

These fields contain the version number of the Java compiler that produced this class file. An implementation of the virtual machine will normally support some range of minor version numbers 0-*n* of a particular major version number. If the minor version number is incremented the new code won't run on the old virtual machines, but it is possible to make a new virtual machine which can run versions up to *n*+1.

A change of the major version number indicates a major incompatible change, one that requires a different virtual machine that may not support the old major version in any way.

The current major version number is 45; the current minor version number is 3.

constant_pool_count

This field indicates the number of entries in the constant pool in the class file.

constant_pool

The constant pool is an table of values. These values are the various string constants, class names, field names, and others that are referred to by the class structure or by the code.

`constant_pool[0]` is always unused by the compiler, and may be used by an implementation for any purpose.

Each of the `constant_pool` entries 1 through `constant_pool_count-1` is a variable-length entry, whose format is given by the first "tag" byte, as described in section 2.3.

access_flags

This field contains a mask of up to sixteen modifiers used with class, method, and field declarations. The same encoding is used on similar fields in `field_info` and `method_info` as described below. Here is the encoding:

Flag Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Visible to everyone	Class, Method, Variable
ACC_PRIVATE	0x0002	Visible only to the defining class	Method, Variable
ACC_PROTECTED	0x0004	Visible to subclasses	Method, Variable
ACC_STATIC	0x0008	Variable or method is static	Method, Variable
ACC_FINAL	0x0010	No further subclassing, overriding, or assignment after initialization	Class, Method, Variable
ACC_SYNCHRONIZED	0x0020	Wrap use in monitor lock	Method
ACC_VOLATILE	0x0040	Can't cache	Variable
ACC_TRANSIENT	0x0080	Not to be written or read by a persistent object manager	Variable
ACC_NATIVE	0x0100	Implemented in a language other than Java	Method
ACC_INTERFACE	0x0200	Is an interface	Class
ACC_ABSTRACT	0x0400	No body provided	Class, Method

this_class

This field is an index into the constant pool; `constant_pool[this_class]` must be a CONSTANT_class.

super_class

This field is an index into the constant pool. If the value of `super_class` is nonzero, then `constant_pool[super_class]` must be a class, and gives the index of this class's superclass in the constant pool.

If the value of `super_class` is zero, then the class being defined must be `java.lang.Object`, and it has no superclass.

interfaces_count

This field gives the number of interfaces that this class implements.

interfaces

Each value in this table is an index into the constant pool. If an table value is nonzero (`interfaces[`*i*`]` != 0, where 0 <= *i* < `interfaces_count`), then `constant_pool[interfaces[`*i*`]]` must be an interface that this class implements.

Question: How could one of these entries ever be 0?

fields_count

This field gives the number of instance variables, both static and dynamic, defined by this class. The `fields` table includes only those variables that are defined explicitly by this class. It does not include those instance variables that are accessible from this class but are inherited from superclasses.

fields

Each value in this table is a more complete description of a field in the class. See section 2.4 for more information on the `field_info` structure.

methods_count

This field indicates the number of methods, both static and dynamic, defined by this class. This table only includes those methods that are explicitly defined by this class. It does not include inherited methods.

methods

Each value in this table is a more complete description of a method in the class. See section 2.5 for more information on the `method_info` structure.

attributes_count

This field indicates the number of additional attributes about this class.

attributes

A class can have any number of optional attributes associated with it. Currently, the only class attribute recognized is the "SourceFile" attribute, which indicates the name of the source file from which this class file was compiled. See section 2.6 for more information on the `attribute_info` structure.

2.2 Signatures

A signature is a string representing a type of a method, field or array.

The field signature represents the value of an argument to a function or the value of a variable. It is a series of bytes generated by the following grammar:

```
<field_signature>    ::=  <field_type>
<field_type>         ::=  <base_type>|<object_type>|<array_type>
<base_type>          ::=  B|C|D|F|I|J|S|Z
<object_type>        ::=  L<fullclassname>;
<array_type>         ::=  [<optional_size><field_type>
<optional_size>      ::=  [0-9]*
```

The meaning of the base types is as follows:

B	`byte`	signed byte
C	`char`	character
D	`double`	double precision IEEE float
F	`float`	single precision IEEE float
I	`int`	integer
J	`long`	long integer
L<fullclassname>**;**	...	an object of the given class
S	`short`	signed short
Z	`boolean`	true or false
[<field sig>	...	array

A return-type signature represents the return value from a method. It is a series of bytes in the following grammar:

```
<return_signature>   ::=  <field_type> | V
```

The character **V** indicates that the method returns no value. Otherwise, the signature indicates the type of the return value.

An argument signature represents an argument passed to a method:

```
<argument_signature> ::=  <field_type>
```

A method signature represents the arguments that the method expects, and the value that it returns.

```
<method_signature>   ::=  (<arguments_signature>) <return_signature>
<arguments_signature>::=  <argument_signature>*
```

2.3 Constant Pool

Each item in the constant pool begins with a 1-byte tag:. The table below lists the valid tags and their values.

Constant Type	Value
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
CONSTANT_Unicode	2

Each tag byte is then followed by one or more bytes giving more information about the specific constant.

2.3.1 CONSTANT_Class

`CONSTANT_Class` is used to represent a class or an interface.

```
CONSTANT_Class_info {
   u1 tag;
   u2 name_index;
}
```

tag

The tag will have the value `CONSTANT_Class`

name_index

constant_pool[name_index] is a `CONSTANT_Utf8` giving the string name of the class.

Because arrays are objects, the opcodes `anewarray` and `multianewarray` can reference array "classes" via `CONSTANT_Class` items in the constant pool. In this case, the name of the class is its signature. For example, the class name for

```
int[][]
```

is

```
[[I
```

The class name for

```
Thread[]
```

is

```
"[Ljava.lang.Thread;"
```

2.3.2 CONSTANT_{Fieldref,Methodref,InterfaceMethodref}

Fields, methods, and interface methods are represented by similar structures.

```
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;
}
```

tag

The tag will have the value `CONSTANT_Fieldref`, `CONSTANT_Methodref`, or `CONSTANT_InterfaceMethodref`.

class_index

`constant_pool[class_index]` will be an entry of type `CONSTANT_Class` giving the name of the class or interface containing the field or method.

For `CONSTANT_Fieldref` and `CONSTANT_Methodref`, the `CONSTANT_Class` item must be an actual class. For `CONSTANT_InterfaceMethodref`, the item must be an interface which purports to implement the given method.

name_and_type_index

`constant_pool[name_and_type_index]` will be an entry of type `CONSTANT_NameAndType`. This constant pool entry indicates the name and signature of the field or method.

2.3.3 CONSTANT_String

CONSTANT_String is used to represent constant objects of the built-in type String.

```
CONSTANT_String_info {
   u1 tag;
   u2 string_index;
}
```

tag

The tag will have the value `CONSTANT_String`

string_index

`constant_pool[string_index]` is a `CONSTANT_Utf8` string giving the value to which the String object is initialized.

2.3.4 CONSTANT_Integer and CONSTANT_Float

`CONSTANT_Integer` and `CONSTANT_Float` represent four-byte constants.

```
CONSTANT_Integer_info {
   u1 tag;
   u4 bytes;
}

CONSTANT_Float_info {
   u1 tag;
   u4 bytes;
}
```

tag

The tag will have the value `CONSTANT_Integer` or `CONSTANT_Float`

bytes

For integers, the four bytes are the integer value. For floats, they are the IEEE 754 standard representation of the floating point value. These bytes are in network (high byte first) order.

2.3.5 CONSTANT_Long and CONSTANT_Double

`CONSTANT_Long` and `CONSTANT_Double` represent eight-byte constants.

```
CONSTANT_Long_info {
   u1 tag;
   u4 high_bytes;
   u4 low_bytes;
}

CONSTANT_Double_info {
   u1 tag;
   u4 high_bytes;
   u4 low_bytes;
}
```

All eight-byte constants take up two spots in the constant pool. If this is the n^{th} item in the constant pool, then the next item will be numbered n+2.

tag

The tag will have the value `CONSTANT_Long` or `CONSTANT_Double`.

high_bytes, low_bytes

For `CONSTANT_Long`, the 64-bit value is (`high_bytes` << 32) + `low_bytes`.

For `CONSTANT_Double`, the 64-bit value, `high_bytes` and `low_bytes` together represent the standard IEEE 754 representation of the double-precision floating point number.

2.3.6 CONSTANT_NameAndType

`CONSTANT_NameAndType` is used to represent a field or method, without indicating which class it belongs to.

```
CONSTANT_NameAndType_info {
   u1 tag;
   u2 name_index;
   u2 signature_index;
}
```

tag

The tag will have the value `CONSTANT_NameAndType`.

name_index

`constant_pool[name_index]` is a `CONSTANT_Utf8` string giving the name of the field or method.

signature_index

`constant_pool[signature_index]` is a `CONSTANT_Utf8` string giving the signature of the field or method.

2.3.7 CONSTANT_Utf8 and CONSTANT_Unicode

`CONSTANT_Utf8` and `CONSTANT_Unicode` are used to represent constant string values.

`CONSTANT_Utf8` strings are "encoded" so that strings containing only non-null ASCII characters, can be represented using only one byte per character, but characters of up to 16 bits can be represented:

All characters in the range 0x0001 to 0x007F are represented by a single byte:

```
+-+-+-+-+-+-+-+-+
|0|7bits of data|
+-+-+-+-+-+-+-+-+
```

The null character (0x0000) and characters in the range 0x0080 to 0x07FF are represented by a pair of two bytes:

```
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
|1|1|0| 5 bits† |   |1|0|   6 bits  |
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
```

Characters in the range 0x0800 to 0xFFFF are represented by three bytes:

```
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
|1|1|1|0|4 bits |   |1|0|   6 bits  |   |1|0|   6 bits  |
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
```

There are two differences between this format and the "standard" UTF-8 format. First, the null byte (0x00) is encoded in two-byte format rather than one-byte, so that our strings never have embedded nulls. Second, only the one-byte, two-byte, and three-byte formats are used. We do not recognize the longer formats.

```
CONSTANT_Utf8_info {
   u1 tag;
   u2 length;
   u1 bytes[length];
}

CONSTANT_Unicode_info {
   u1 tag;
   u2 length;
   u2 bytes[length];
}
```

tag

The tag will have the value `CONSTANT_Utf8` or `CONSTANT_Unicode`.

length

The number of bytes in the string. These strings are not null terminated.

bytes

The actual bytes of the string.

2.4 Fields

The information for each field immediately follows the field_count field in the class file. Each field is described by a variable length field_info structure. The format of this structure is as follows:

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

access_flags

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table "Access Flags" on page 12 which indicates the meaning of the bits in this field.

The possible fields that can be set for a field are `ACC_PUBLIC`, `ACC_PRIVATE`, `ACC_PROTECTED`, `ACC_STATIC`, `ACC_FINAL`, `ACC_VOLATILE`, and `ACC_TRANSIENT`.

At most one of `ACC_PUBLIC`, `ACC_PROTECTED`, and `ACC_PRIVATE` can be set for any method.

name_index

`constant_pool[name_index]` is a `CONSTANT_Utf8` string which is the name of the field.

signature_index

`constant_pool[signature_index]` is a `CONSTANT_Utf8` string which is the signature of the field. See the section "Signatures" for more information on signatures.

attributes_count

This value indicates the number of additional attributes about this field.

attributes

A field can have any number of optional attributes associated with it. Currently, the only field attribute recognized is the "ConstantValue" attribute, which indicates that this field is a static numeric constant, and indicates the constant value of that field.

Any other attributes are skipped.

2.5 Methods

The information for each method immediately follows the `method_count` field in the class file. Each method is described by a variable length `method_info` structure. The structure has the following format:

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

access_flags

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table “Access Flags” on page 12 which gives the various bits in this field.

The possible fields that can be set for a method are `ACC_PUBLIC`, `ACC_PRIVATE`, `ACC_PROTECTED`, `ACC_STATIC`, `ACC_FINAL`, `ACC_SYNCHRONIZED`, `ACC_NATIVE`, and `ACC_ABSTRACT`.

At most one of `ACC_PUBLIC`, `ACC_PROTECTED`, and `ACC_PRIVATE` can be set for any method.

name_index

`constant_pool[name_index]` is a `CONSTANT_Utf8` string giving the name of the method.

signature_index

`constant_pool[signature_index]`is a `CONSTANT_Utf8` string giving the signature of the field. See the section “Signatures” for more information on signatures.

attributes_count

This value indicates the number of additional attributes about this field.

attributes

A field can have any number of optional attributes associated with it. Each attribute has a name, and other additional information. Currently, the only field attributes recognized are the “Code” and “Exceptions” attributes, which describe the bytecodes that are executed to perform this method, and the Java Exceptions which are declared to result from the execution of the method, respectively.

Any other attributes are skipped.

2.6 Attributes

Attributes are used at several different places in the class format. All attributes have the following format:

```
GenericAttribute_info {
   u2 attribute_name;
   u4 attribute_length;
   u1 info[attribute_length];
}
```

The `attribute_name` is a 16-bit index into the class’s constant pool; the value of `constant_pool[attribute_name]` is a `CONSTANT_Utf8` string giving the name of the attribute. The field `attribute_length` indicates the length of the subsequent information in bytes. This length does not include the six bytes of the `attribute_name` and `attribute_length`.

In the following text, whenever we allow attributes, we give the name of the attributes that are currently understood. In the future, more attributes will be added. Class file readers are expected to skip over and ignore the information in any attribute they do not understand.

2.6.1 SourceFile

The "SourceFile" attribute has the following format:

```
SourceFile_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 sourcefile_index;
}
```

attribute_name_index

`constant_pool[attribute_name_index]` is the `CONSTANT_Utf8` string `"SourceFile"`.

attribute_length

The length of a SourceFile_attribute must be 2.

sourcefile_index

`constant_pool[sourcefile_index]` is a `CONSTANT_Utf8` string giving the source file from which this class file was compiled.

2.6.2 ConstantValue

The "ConstantValue" attribute has the following format:

```
ConstantValue_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 constantvalue_index;
}
```

attribute_name_index

`constant_pool[attribute_name_index]` is the `CONSTANT_Utf8` string `"ConstantValue"`.

attribute_length

The length of a ConstantValue_attribute must be 2.

constantvalue_index

`constant_pool[constantvalue_index]` gives the constant value for this field.

The constant pool entry must be of a type appropriate to the field, as shown by the following table:

long	`CONSTANT_Long`
float	`CONSTANT_Float`
double	`CONSTANT_Double`
int, short, char, byte, boolean	`CONSTANT_Integer`

2.6.3 Code

The "Code" attribute has the following format:

```
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[attribute_count];

}
```

attribute_name_index

`constant_pool[attribute_name_index]` is the `CONSTANT_Utf8` string "`Code`".

attribute_length

This field indicates the total length of the "Code" attribute, excluding the initial six bytes.

max_stack

Maximum number of entries on the operand stack that will be used during execution of this method. See the other chapters in this spec for more information on the operand stack.

max_locals

Number of local variable slots used by this method. See the other chapters in this spec for more information on the local variables.

code_length

The number of bytes in the virtual machine code for this method.

code

These are the actual bytes of the virtual machine code that implement the method. When read into memory, if the first byte of code is aligned onto a multiple-of-four boundary the the `tableswitch` and `tablelookup` opcode entries will be aligned; see their description for more information on alignment requirements.

exception_table_length

The number of entries in the following exception table.

exception_table

Each entry in the exception table describes one exception handler in the code.

start_pc, end_pc

The two fields `start_pc` and `end_pc` indicate the ranges in the code at which the exception handler is active. The values of both fields are offsets from the start of the code. `start_pc` is inclusive. `end_pc` is exclusive.

handler_pc

This field indicates the starting address of the exception handler. The value of the field is an offset from the start of the code.

catch_type

If `catch_type` is nonzero, then `constant_pool[catch_type]` will be the class of exceptions that this exception handler is designated to catch. This exception handler should only be called if the thrown exception is an instance of the given class.

If catch_type is zero, this exception handler should be called for all exceptions.

attributes_count

This field indicates the number of additional attributes about code. The “Code” attribute can itself have attributes.

attributes

A “Code” attribute can have any number of optional attributes associated with it. Each attribute has a name, and other additional information. Currently, the only code attributes defined are the “LineNumberTable” and “LocalVariableTable,” both of which contain debugging information.

2.6.4 Exceptions Table

This table is used by compilers which indicate which Exceptions a method is declared to throw:

```
Exceptions_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 number_of_exceptions;
   u2 exception_index_table[number_of_exceptions];
}
```

attribute_name_index

`constant_pool[attribute_name_index]` will be the `CONSTANT_Utf8` string `"Exceptions"`.

attribute_length

This field indicates the total length of the Exceptions_attribute, excluding the initial six bytes.

number_of_exceptions

This field indicates the number of entries in the following exception index table.

exception_index_table

Each value in this table is an index into the constant pool. For each table element (`exception_index_table[`*i*`]` != 0, where 0 <= *i* < `number_of_exceptions`), then `constant_pool[exception_index+table[`*i*`]]` is a Exception that this class is declared to throw.

2.6.5 LineNumberTable

This attribute is used by debuggers and the exception handler to determine which part of the virtual machine code corresponds to a given location in the source. The LineNumberTable_attribute has the following format:

```
LineNumberTable_attribute {
   u2                  attribute_name_index;
   u4                  attribute_length;
   u2                  line_number_table_length;
   {  u2                  start_pc;
      u2                  line_number;
   }                   line_number_table[line_number_table_length];
}
```

attribute_name_index

`constant_pool[attribute_name_index]` will be the `CONSTANT_Utf8` string "`LineNumberTable`".

attribute_length

This field indicates the total length of the LineNumberTable_attribute, excluding the initial six bytes.

line_number_table_length

This field indicates the number of entries in the following line number table.

line_number_table

Each entry in the line number table indicates that the line number in the source file changes at a given point in the code.

start_pc

This field indicates the place in the code at which the code for a new line in the source begins. `source_pc` <<SHOULD THAT BE `start_pc`?>> is an offset from the beginning of the code.

line_number

The line number that begins at the given location in the file.

2.6.6 LocalVariableTable

This attribute is used by debuggers to determine the value of a given local variable during the dynamic execution of a method. The format of the LocalVariableTable_attribute is as follows:

```
LocalVariableTable_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 local_variable_table_length;
   {  u2    start_pc;
      u2   length;
      u2   name_index;
      u2   signature_index;
      u2   slot;
   }  local_variable_table[local_variable_table_length];
}
```

attribute_name_index

`constant_pool[attribute_name_index]` will be the `CONSTANT_Utf8` string "`LocalVariableTable`".

attribute_length

This field indicates the total length of the LineNumberTable_attribute, excluding the initial six bytes.

local_variable_table_length

This field indicates the number of entries in the following local variable table.

local_variable_table

Each entry in the local variable table indicates a code range during which a local variable has a value. It also indicates where on the stack the value of that variable can be found.

start_pc, length

The given local variable will have a value at the code between `start_pc` and `start_pc + length`. The two values are both offsets from the beginning of the code.

name_index, signature_index

`constant_pool[name_index]` and `constant_pool[signature_index]` are `CONSTANT_Utf8` strings giving the name and signature of the local variable.

slot

The given variable will be the *slot*th local variable in the method's frame.

3 The Virtual Machine Instruction Set

3.1 Format for the Instructions

Java Virtual Machine instructions are represented in this document by an entry of the following form.

instruction name

Short description of the instruction

Syntax:

opcode = number
operand1
operand2
...

Stack: …., *value1*, *value2* ⇒ ..., *value3*

A *longer description* that explains the functions of the instruction and indicates any exceptions that might be thrown during execution.

Each line in the syntax diagram represents a single 8-bit byte.

Operations of the Java Virtual Machine most often take their operands from the stack and put their results back on the stack. As a convention, the descriptions do not usually mention when the stack is the source or destination of an operation, but will always mention when it is not. For instance, the `iload` instruction has the short description "Load integer from local variable." Implicitly, the integer is loaded onto the stack. The `iadd` instruction is described as "Integer add"; both its source and destination are the stack.

Instructions that do not affect the control flow of a computation may be assumed to always advance the virtual machine pc to the opcode of the following instruction. Only instructions that do affect control flow will explicitly mention the effect they have on pc.

3.2 Pushing Constants onto the Stack

bipush

Push one-byte signed integer

Syntax:

bipush = 16
byte1

Stack: ... => ..., *value*

byte1 is interpreted as a signed 8-bit *value*. This *value* is expanded to an integer and pushed onto the operand stack.

sipush

Push two-byte signed integer

Syntax:

sipush = 17
byte1
byte2

Stack: ... => ..., *item*

byte1 and *byte2* are assembled into a signed 16-bit *value*. This *value* is expanded to an integer and pushed onto the operand stack.

ldc1

Push item from constant pool

Syntax:

ldc1 = 18
indexbyte1

Stack: ... => ..., *item*

indexbyte1 is used as an unsigned 8-bit index into the constant pool of the current class. The *item* at that index is resolved and pushed onto the stack. If a `String` is being pushed and there isn't enough memory to allocate space for it then an `OutOfMemoryError` is thrown.

Note: A String push results in a reference to an object; what other constants do, and explain this somewhere here.

ldc2

Push item from constant pool

Syntax:

ldc2 = 19
indexbyte1
indexbyte2

Stack: ... => ..., *item*

indexbyte1 and *indexbyte2* are used to construct an unsigned 16-bit index into the constant pool of the current class. The *item* at that index is resolved and pushed onto the stack. If a `String` is being pushed and there isn't enough memory to allocate space for it then an `OutOfMemoryError` is thrown.

Note: A String push results in a reference to an object; what other constants do, and explain this somewhere here.

ldc2w

Push long or double from constant pool

Syntax:

ldc2w = 20
indexbyte1
indexbyte2

Stack: ... => ..., *constant-word1*, *constant-word2*

indexbyte1 and *indexbyte2* are used to construct an unsigned 16-bit index into the constant pool of the current class. The two-word *constant* at that index is resolved and pushed onto the stack.

aconst_null

Push null object reference

Syntax:

aconst_null = 1

Stack: ... => ..., *null*

Push the `null` object reference onto the stack.

iconst_m1

Push integer constant –1

Syntax:

iconst_m1 = 2

Stack: ... => ..., –1

Push the integer –1 onto the stack.

iconst_<n>

Push integer constant

Syntax:

iconst_<n>

Stack: ... => ..., <*n*>

Forms: `iconst_0` = 3, `iconst_1` = 4, `iconst_2` = 5, `iconst_3` = 6, `iconst_4` = 7, `iconst_5` = 8

Push the integer <*n*> onto the stack.

lconst_<l>

Push long integer constant

Syntax:

lconst_<l>

Stack: ... => ..., <*l*>*-word1*, <*l*>*-word2*

Forms: `lconst_0` = 9, `lconst_1` = 10

Push the long integer <*l*> onto the stack.

fconst_<f>

Push single float

Syntax:

fconst_<f>

Stack: ... => ..., <*f*>

Forms: `fconst_0` = 11, `fconst_1` = 12, `fconst_2` = 13

Push the single-precision floating point number <*f*> onto the stack.

dconst_<d>

Push double float

Syntax:

dconst_<d>

Stack: ... => ..., <*d*>*-word1*, <*d*>*-word2*

Forms: `dconst_0` = 14, `dconst_1` = 15

Push the double-precision floating point number <*d*> onto the stack.

3.3 Loading Local Variables Onto the Stack

iload

Load integer from local variable

Syntax:

iload = 21
vindex

Stack: ... => ..., *value*

The *value* of the local variable at *vindex* in the current Java frame is pushed onto the operand stack.

iload_<n>

Load integer from local variable

Syntax:

iload_<n>

Stack: ... => ..., *value*

Forms: `iload_0` = 26, `iload_1` = 27, `iload_2` = 28, `iload_3` = 29

The *value* of the local variable at <*n*> in the current Java frame is pushed onto the operand stack.

This instruction is the same as `iload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

lload

Load long integer from local variable

Syntax:

lload = 22
vindex

Stack: ... => ..., *value-word1*, *value-word2*

The *value* of the local variables at *vindex* and *vindex*+1 in the current Java frame is pushed onto the operand stack.

lload_<n>

Load long integer from local variable

Syntax:

lload_<n>

Stack: ... => ..., *value-word1*, *value-word2*

Forms: `lload_0` = 30, `lload_1` = 31, `lload_2` = 32, `lload_3` = 33

The *value* of the local variables at <*n*> and <*n*>+1 in the current Java frame is pushed onto the operand stack.

This instruction is the same as `lload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

fload

Load single float from local variable

Syntax:

fload = 23
vindex

Stack: ... => ..., *value*

The *value* of the local variable at *vindex* in the current Java frame is pushed onto the operand stack.

fload_<n>

Load single float from local variable

Syntax:

fload_<n>

Stack: ... => ..., *value*

Forms: `fload_0` = 34, `fload_1` = 35, `fload_2` = 36, `fload_3` = 37

The *value* of the local variable at <*n*>in the current Java frame is pushed onto the operand stack.

This instruction is the same as `fload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

dload

Load double float from local variable

Syntax:

dload = 24
vindex

Stack: ... => ..., *value-word1*, *value-word2*

The *value* of the local variables at *vindex* and *vindex*+1 in the current Java frame is pushed onto the operand stack.

dload_<n>

Load double float from local variable

Syntax:

dload_<n>

Stack: ... => ..., *value-word1*, *value-word2*

Forms: `dload_0` = 38, `dload_1` = 39, `dload_2` = 40, `dload_3` = 41

The *value* of the local variables at <*n*> and <*n*>+1 in the current Java frame is pushed onto the operand stack.

This instruction is the same as `dload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

aload

Load object reference from local variable

Syntax:

aload = 25
vindex

Stack: ... => ..., *value*

The *value* of the local variable at *vindex* in the current Java frame is pushed onto the operand stack.

aload_<n>

Load object reference from local variable

Syntax:

aload_<n>

Stack: ... => ..., *value*

Forms: `aload_0` = 42, `aload_1` = 43, `aload_2` = 44, `aload_3` = 45

The *value* of the local variable at <*n*>in the current Java frame is pushed onto the operand stack.

This instruction is the same as `aload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

3.4 Storing Stack Values into Local Variables

istore

Store integer into local variable

Syntax:

istore = 54
vindex

Stack: ..., *value* => ...

value must be an integer. Local variable *vindex* in the current Java frame is set to *value*.

istore_<n>

Store integer into local variable

Syntax:

istore_<n>

Stack: ..., *value* => ...

Forms: `istore_0` = 59, `istore_1` = 60, `istore_2` = 61, `istore_3` = 62

value must be an integer. Local variable <*n*> in the current Java frame is set to *value*.

This instruction is the same as `istore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

lstore

Store long integer into local variable

Syntax:

lstore = 55
vindex

Stack: ..., *value-word1*, *value-word2* => ...

value must be a long integer. Local variables *vindex* and *vindex*+1 in the current Java frame are set to *value*.

lstore_<n>

Store long integer into local variable

Syntax:

lstore_<n>

Stack: ..., *value-word1*, *value-word2* => ...

Forms: `lstore_0` = 63, `lstore_1` = 64, `lstore_2` = 65, `lstore_3` = 66

value must be a long integer. Local variables <*n*> and <*n*>+1 in the current Java frame are set to *value*.

This instruction is the same as `lstore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

fstore

Store single float into local variable

Syntax:

fstore = 56
vindex

Stack: ..., *value* => ...

value must be a single-precision floating point number. Local variable *vindex* in the current Java frame is set to *value*.

fstore_<n>

Store single float into local variable

Syntax:

fstore_<n>

Stack: ..., *value* => ...

Forms: `fstore_0` = 67, `fstore_1` = 68, `fstore_2` = 69, `fstore_3` = 70

value must be a single-precision floating point number. Local variable <*n*> in the current Java frame is set to *value*.

This instruction is the same as `fstore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

dstore

Store double float into local variable

Syntax:

dstore = 57
vindex

Stack: ..., *value-word1*, *value-word2* => ...

value must be a double-precision floating point number. Local variables *vindex* and *vindex*+1 in the current Java frame are set to *value*.

dstore_<n>

Store double float into local variable

Syntax:

dstore_<n>

Stack: ..., *value-word1*, *value-word2* => ...

Forms: `dstore_0` = 71, `dstore_1` = 72, `dstore_2` = 73, `dstore_3` = 74

value must be a double-precision floating point number. Local variables <*n*> and <*n*>+1 in the current Java frame are set to *value*.

This instruction is the same as `dstore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

astore

Store object reference into local variable

Syntax:

astore = 58
vindex

Stack: ..., *value* => ...

value must be a return address or a reference to an object. Local variable *vindex* in the current Java frame is set to *value*.

astore_<n>

Store object reference into local variable

Syntax:

astore_<n>

Stack: ..., *value* => ...

Forms: `astore_0` = 75, `astore_1` = 76, `astore_2` = 77, `astore_3` = 78

value must be a return address or a reference to an object. Local variable <*n*> in the current Java frame is set to *value*.

This instruction is the same as `astore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

iinc

Increment local variable by constant

Syntax:

iinc = 132
vindex
const

Stack: no change

Local variable *vindex* in the current Java frame must contain an integer. Its value is incremented by the value *const*, where *const* is treated as a signed 8-bit quantity.

3.5 Wider index for Loading, Storing and Incrementing

wide

Wider index for accessing local variables in load, store and increment.

Syntax:

wide = 196
vindex2

Stack: no change

This bytecode must precede one of the following bytecodes: `iload`, `lload`, `fload`, `dload`, `aload`, `istore`, `lstore`, `fstore`, `dstore`, `astore`, `iinc`. The *vindex* of the following bytecode and *vindex2* from this bytecode are assembled into an unsigned 16-bit index to a local variable in the current Java frame. The following bytecode operates as normal except for the use of this wider index.

3.6 Managing Arrays

newarray

Allocate new array

Syntax:

newarray = 188
atype

Stack: ..., *size* => *result*

size must be an integer. It represents the number of elements in the new array.

atype is an internal code that indicates the type of array to allocate. Possible values for *atype* are as follows:

T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9
T_INT	10
T_LONG	11

A new array of *atype,* capable of holding *size* elements, is allocated, and *result* is a reference to this new object. Allocation of an array large enough to contain *size* items of *atype* is attempted. All elements of the array are initialized to zero.

If *size* is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

anewarray

Allocate new array of references to objects

Syntax:

anewarray = 189
indexbyte1
indexbyte2

Stack: ..., *size*=> *result*

size must be an integer. It represents the number of elements in the new array.

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry must be a class.

A new array of the indicated class type and capable of holding *size* elements is allocated, and *result* is a reference to this new object. Allocation of an array large enough to contain *size* items of the given class type is attempted. All elements of the array are initialized to `null`.

If *size* is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

`anewarray` is used to create a single dimension of an array of object references. For example, to create

```
new Thread[7]
```

the following code is used:

```
bipush 7
anewarray <Class "java.lang.Thread">
```

`anewarray` can also be used to create the first dimension of a multi-dimensional array. For example, the following array declaration:

```
new int[6][]
```

is created with the following code:

```
bipush 6
anewarray <Class "[I">
```

See `CONSTANT_Class` in the “Class File Format” chapter for information on array class names.

multianewarray

Allocate new multi-dimensional array

Syntax:

multianewarray = 197
indexbyte1
indexbyte2
dimensions

Stack: ..., *size1 size2...sizen* => *result*

Each *size* must be an integer. Each represents the number of elements in a dimension of the array.

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry must be an array class of one or more dimensions.

dimensions has the following aspects:

- It must be an integer ≥ 1.
- It represents the number of dimensions being created. It must be $\leq$ the number of dimensions of the array class.

- It represents the number of elements that are popped off the stack. All must be integers greater than or equal to zero. These are used as the sizes of the dimension. For example, to create

  ```
  new int[6][3][]
  ```

  the following code is used:

  ```
  bipush 6
  bipush 3
  multianewarray <Class "[[[I"> 2
  ```

If any of the *size* arguments on the stack is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

The *result* is a reference to the new array object.

Note: More explanation needed about how this is an array of arrays.

Note: It is more efficient to use `newarray` or `anewarray` when creating a single dimension.

See `CONSTANT_Class` in the “Class File Format” chapter for information on array class names.

arraylength

Get length of array

Syntax:

arraylength = 190

Stack: ..., *objectref* => ..., *length*

objectref must be a reference to an array object. The length of the array is determined and replaces *objectref* on the top of the stack.

If the *objectref* is `null`, a `NullPointerException` is thrown.

iaload

Load integer from array

Syntax:

iaload = 46

Stack: ..., *arrayref*, *index* => ..., *value*

arrayref must be a reference to an array of integers. *index* must be an integer. The integer *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

laload

Load long integer from array

Syntax:

laload = 47

Stack: ..., *arrayref*, *index* => ..., *value-word1*, *value-word2*

arrayref must be a reference to an array of long integers. *index* must be an integer. The long integer *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

faload

Load single float from array

Syntax:

faload = 48

Stack: ..., *arrayref*, *index* => ..., *value*

arrayref must be a reference to an array of single-precision floating point numbers. *index* must be an integer. The single-precision floating point number *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

daload

Load double float from array

Syntax:

daload = 49

Stack: ..., *arrayref*, *index* => ..., *value-word1*, *value-word2*

arrayref must be a reference to an array of double-precision floating point numbers. *index* must be an integer. The double-precision floating point number *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

aaload

Load object reference from array

Syntax:

aaload = 50

Stack: ..., *arrayref*, *index* => ..., *value*

arrayref must be a reference to an array of references to objects. *index* must be an integer. The object reference at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

baload

Load signed byte from array.

Syntax:

baload = 51

Stack: ..., *arrayref*, *index* => ..., *value*

arrayref must be a reference to an array of signed bytes. *index* must be an integer. The signed byte value at position number *index* in the array is retrieved, expanded to an integer, and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

caload

Load character from array

Syntax:

caload = 52

Stack: ..., *arrayref*, *index* => ..., *value*

arrayref must be a reference to an array of characters. *index* must be an integer. The character value at position number *index* in the array is retrieved, zero-extended to an integer, and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

saload

Load short from array

Syntax:

saload = 53

Stack: ..., *arrayref*, *index* => ..., *value*

arrayref must be a reference to an array of short integers. *index* must be an integer. The ;signed short integer value at position number *index* in the array is retrieved, expanded to an integer, and pushed onto the top of the stack.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

iastore

Store into integer array

Syntax:

iastore = 79

Stack: ..., *arrayref*, *index*, *value* => ...

arrayref must be a reference to an array of integers, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

lastore

Store into long integer array

Syntax:

lastore = 80

Stack: ..., *arrayref*, *index*, *value-word1*, *value-word2* => ...

arrayref must be a reference to an array of long integers, *index* must be an integer, and *value* a long integer. The long integer *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array, an `ArrayIndexOutOfBoundsException` is thrown.

fastore

Store into single float array

Syntax:

fastore = 81

Stack: ..., *arrayref*, *index*, *value* => ...

arrayref must be an array of single-precision floating point numbers, *index* must be an integer, and *value* a single-precision floating point number. The single float *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

dastore

Store into double float array

Syntax:

dastore = 82

Stack: ..., *arrayref*, *index*, *value-word1*, *value-word2* => ...

arrayref must be a reference to an array of double-precision floating point numbers, *index* must be an integer, and *value* a double-precision floating point number. The double float *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

aastore

Store into object reference array

Syntax:

aastore = 83

Stack: ..., *arrayref*, *index*, *value* => ...

arrayref must be a reference to an array of references to objects, *index* must be an integer, and *value* a reference to an object. The object reference *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array, an `ArrayIndexOutOfBoundsException` is thrown.

The actual type of *value* must be conformable with the actual type of the elements of the array. For example, it is legal to store an instance of class `Thread` in an array of class `Object`, but not vice versa. (See the *Java Language Specification* for information on how to determine whether a object reference is an instance of a class.) An `ArrayStoreException` is thrown if an attempt is made to store an incompatible object reference.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

bastore

Store into signed byte array

Syntax:

bastore = 84

Stack: ..., *arrayref*, *index*, *value* => ...

arrayref must be a reference to an array of signed bytes, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array. If *value* is too large to be a signed byte, it is truncated.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

castore

Store into character array

Syntax:

castore = 85

Stack: ..., *arrayref*, *index*, *value* => ...

arrayref must be an array of characters, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array. If *value* is too large to be a character, it is truncated.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of [the array an `ArrayIndexOutOfBoundsException` is thrown.

sastore

Store into short array

Syntax:

sastore = 86

Stack: ..., *array*, *index*, *value* => ...

arrayref must be an array of shorts, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array. If *value* is too large to be an short, it is truncated.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

3.7 Stack Instructions

nop

Do nothing

Syntax:

nop = 0

Stack: no change

Do nothing.

pop

Pop top stack word

Syntax:

pop = 87

Stack: ..., *any* => ...

Pop the top word from the stack.

pop2

Pop top two stack words

Syntax:

pop2 = 88

Stack: ..., *any2*, *any1* => ...

Pop the top two words from the stack.

dup

Duplicate top stack word

Syntax:

dup = 89

Stack: ..., *any* => ..., *any*, *any*

Duplicate the top word on the stack.

dup2

Duplicate top two stack words

Syntax:

dup2 = 92

Stack: ..., *any2*, *any1* => ..., *any2*, *any1*, *any2*, *any1*

Duplicate the top two words on the stack.

dup_x1

Duplicate top stack word and put two down

Syntax:

dup_x1 = 90

Stack: ..., *any2*, *any1* => ..., *any1*, *any2*, *any1*

Duplicate the top word on the stack and insert the copy two words down in the stack.

dup2_x1

Duplicate top two stack words and put two down

Syntax:

dup2_x1 = 93

Stack: ..., *any3*, *any2*, *any1* => ..., *any2*, *any1*, *any3*, *any2*, *any1*

Duplicate the top two words on the stack and insert the copies two words down in the stack.

dup_x2

Duplicate top stack word and put three down

Syntax:

dup_x2 = 91

Stack: ..., *any3*, *any2*, *any1* => ..., *any1*, *any3*, *any2*, *any1*

Duplicate the top word on the stack and insert the copy three words down in the stack.

dup2_x2

Duplicate top two stack words and put three down

Syntax:

dup2_x2 = 94

Stack: ..., *any4*, *any3*, *any2*, *any1* => ..., *any2*, *any1*, *any4*, *any3*, *any2*, *any1*

Duplicate the top two words on the stack and insert the copies three words down in the stack.

swap

Swap top two stack words

Syntax:

swap = 95

Stack: ..., *any2*, *any1* => ..., *any2*, *any1*

Swap the top two elements on the stack.

3.8 Arithmetic Instructions

iadd

Integer add

Syntax:

iadd = 96

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

value1 and *value2* must be integers. The values are added and are replaced on the stack by their integer sum.

ladd

Long integer add

Syntax:

ladd = 97

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be long integers. The values are added and are replaced on the stack by their long integer sum.

fadd

Single floats add

Syntax:

fadd = 98

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

value1 and *value2* must be single-precision floating point numbers. The values are added and are replaced on the stack by their single-precision floating point sum.

dadd

Double floats add

Syntax:

dadd = 99

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be double-precision floating point numbers. The values are added and are replaced on the stack by their double-precision floating point sum.

isub

Integer subtract

Syntax:

isub = 100

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

value1 and *value2* must be integers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their integer difference.

lsub

Long integer subtract

Syntax:

lsub = 101

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be long integers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their long integer difference.

fsub

Single float subtract

Syntax:

fsub = 102

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

value1 and *value2* must be single-precision floating point numbers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their single-precision floating point difference.

dsub

Double float subtract

Syntax:

dsub = 103

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be double-precision floating point numbers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their double-precision floating point difference.

imul

Integer multiply

Syntax:

imul = 104

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

value1 and *value2* must be integers. Both values are replaced on the stack by their integer product.

lmul

Long integer multiply

Syntax:

imul = 105

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be long integers. Both values are replaced on the stack by their long integer product.

fmul

Single float multiply

Syntax:

fmul = 106

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

value1 and *value2* must be single-precision floating point numbers. Both values are replaced on the stack by their single-precision floating point product.

dmul

Double float multiply

Syntax:

dmul = 107

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be double-precision floating point numbers. Both values are replaced on the stack by their double-precision floating point product.

idiv

Integer divide

Syntax:

idiv = 108

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

value1 and *value2* must be integers. *value1* is divided by *value2*, and both values are replaced on the stack by their integer quotient.

The result is truncated to the nearest integer that is between it and 0. An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown.

ldiv

Long integer divide

Syntax:

ldiv = 109

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be long integers. *value1* is divided by *value2*, and both values are replaced on the stack by their long integer quotient.

The result is truncated to the nearest integer that is between it and 0. An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown.

fdiv

Single float divide

Syntax:

fdiv = 110

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

value1 and *value2* must be single-precision floating point numbers. *value1* is divided by *value2*, and both values are replaced on the stack by their single-precision floating point quotient.

Divide by zero results in the quotient being NaN.

ddiv

Double float divide

Syntax:

ddiv = 111

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must be double-precision floating point numbers. *value1* is divided by *value2,* and both values are replaced on the stack by their double-precision floating point quotient.

Divide by zero results in the quotient being NaN.

irem

Integer remainder

Syntax:

irem = 112

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

value1 and *value2* must both be integers. *value1* is divided by *value2*, and both values are replaced on the stack by their integer remainder.

An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown.

Note: need a description of the integer remainder semantics

lrem

Long integer remainder

Syntax:

lrem = 113

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must both be long integers. *value1* is divided by *value2*, and both values are replaced on the stack by their long integer remainder.

An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown.

Note: need a description of the integer remainder semantics

frem

Single float remainder

Syntax:

frem = 114

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

value1 and *value2* must both be single-precision floating point numbers. *value1* is divided by *value2*, and the quotient is truncated to an integer, and then multiplied by *value2.* The product is subtracted from *value1.*The result, as a single-precision floating point number, replaces both values on the stack. *result* = *value1* - (integral_part(*value1*/*value2*) * *value2*), where integral_part() rounds to the nearest integer, with a tie going to the even number.

An attempt to divide by zero results in NaN.

Note: gls to provide a better definition of the floating remainder semantics

drem

Double float remainder

Syntax:

drem = 115

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must both be double-precision floating point numbers. *value1* is divided by *value2*, and the quotient is truncated to an integer, and then multiplied by *value2*. The product is subtracted from *value1*.The result, as a double-precision floating point number, replaces both values on the stack. *result* = *value1* - (integral_part(*value1*/*value2*) * *value2*), where integral_part() rounds to the nearest integer, with a tie going to the even number.

An attempt to divide by zero results in NaN.

Note: gls to provide a better definition of the floating remainder semantics

ineg

Integer negate

Syntax:

ineg = 116

Stack: ..., *value* => ..., *result*

value must be an integer. It is replaced on the stack by its arithmetic negation.

lneg

Long integer negate

Syntax:

lneg = 117

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

value must be a long integer. It is replaced on the stack by its arithmetic negation.

fneg

Single float negate

Syntax:

fneg = 118

Stack: ..., *value* => ..., *result*

value must be a single-precision floating point number. It is replaced on the stack by its arithmetic negation.

dneg

Double float negate

Syntax:

dneg = 119

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

value must be a double-precision floating point number. It is replaced on the stack by its arithmetic negation.

3.9 Logical Instructions

ishl

Integer shift left

Syntax:

ishl = 120

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

value1 and *value2* must be integers. *value1* is shifted left by the amount indicated by the low five bits of *value2.* The integer result replaces both values on the stack.

ishr

Integer arithmetic shift right

Syntax:

ishr = 122

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

value1 and *value2* must be integers. *value1* is shifted right arithmetically (with sign extension) by the amount indicated by the low five bits of *value2.* The integer result replaces both values on the stack.

iushr

Integer logical shift right

Syntax:

iushr = 124

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

value1 and *value2* must be integers. *value1* is shifted right logically (with no sign extension) by the amount indicated by the low five bits of *value2.* The integer result replaces both values on the stack.

lshl

Long integer shift left

Syntax:

lshl = 121

Stack: ..., *value1-word1*, *value1-word2*, *value2* => ..., *result-word1*, *result-word2*

value1 must be a long integer and *value2* must be an integer. *value1* is shifted left by the amount indicated by the low six bits of *value2.* The long integer result replaces both values on the stack.

lshr

Long integer arithmetic shift right

Syntax:

lshr = 123

Stack: ..., *value1-word1*, *value1-word2*, *value2* => ..., *result-word1*, *result-word2*

value1 must be a long integer and *value2* must be an integer. *value1* is shifted right arithmetically (with sign extension) by the amount indicated by the low six bits of *value2.* The long integer result replaces both values on the stack.

lushr

Long integer logical shift right

Syntax:

lushr = 125

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 must be a long integer and *value2* must be an integer. *value1* is shifted right logically (with no sign extension) by the amount indicated by the low six bits of *value2*. The long integer result replaces both values on the stack.

iand

Integer boolean AND

Syntax:

iand = 126

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

value1 and *value2* must both be integers. They are replaced on the stack by their bitwise logical and (conjunction).

land

Long integer boolean AND

Syntax:

land = 127

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must both be long integers. They are replaced on the stack by their bitwise logical and (conjunction).

ior

Integer boolean OR

Syntax:

ior = 128

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

value1 and *value2* must both be integers. They are replaced on the stack by their bitwise logical or (disjunction).

lor

Long integer boolean OR

Syntax:

lor = 129

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must both be long integers. They are replaced on the stack by their bitwise logical or (disjunction).

ixor

Integer boolean XOR

Syntax:

ixor = 130

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

value1 and *value2* must both be integers. They are replaced on the stack by their bitwise exclusive or (exclusive disjunction).

lxor

Long integer boolean XOR

Syntax:

lxor = 131

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

value1 and *value2* must both be long integers. They are replaced on the stack by their bitwise exclusive or (exclusive disjunction).

3.10 Conversion Operations

i2l

Integer to long integer conversion

Syntax:

i2l = 133

Stack: ..., *value* => ..., *result-word1*, *result-word2*

value must be an integer. It is converted to a long integer. The result replaces *value* on the stack.

i2f

Integer to single float

Syntax:

i2f = 134

Stack: ..., *value* => ..., *result*

value must be an integer. It is converted to a single-precision floating point number. The result replaces *value* on the stack.

i2d

Integer to double float

Syntax:

i2d = 135

Stack: ..., *value* => ..., *result-word1*, *result-word2*

value must be an integer. It is converted to a double-precision floating point number. The result replaces *value* on the stack.

l2i

Long integer to integer

Syntax:

l2i = 136

Stack: ..., *value-word1*, *value-word2* => ..., *result*

value must be a long integer. It is converted to an integer by taking the low-order 32 bits. The result replaces *value* on the stack.

l2f

Long integer to single float

Syntax:

l2f = 137

Stack: ..., *value-word1*, *value-word2* => ..., *result*

value must be a long integer. It is converted to a single-precision floating point number. The result replaces *value* on the stack.

l2d

Long integer to double float

Syntax:

l2d = 138

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

value must be a long integer. It is converted to a double-precision floating point number. The result replaces *value* on the stack.

f2i

Single float to integer

Syntax:

f2i = 139

Stack: ..., *value* => ..., *result*

value must be a single-precision floating point number. It is converted to an integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

f2l

Single float to long integer

Syntax:

f2l = 140

Stack: ..., *value* => ..., *result-word1*, *result-word2*

value must be a single-precision floating point number. It is converted to a long integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

f2d

Single float to double float

Syntax:

f2d = 141

Stack: ..., *value* => ..., *result-word1*, *result-word2*

value must be a single-precision floating point number. It is converted to a double-precision floating point number. The result replaces *value* on the stack.

d2i

Double float to integer

Syntax:

d2i = 142

Stack: ..., *value-word1*, *value-word2* => ..., *result*

value must be a double-precision floating point number. It is converted to an integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

d2l

Double float to long integer

Syntax:

d2l = 143

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

value must be a double-precision floating point number. It is converted to a long integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

d2f

Double float to single float

Syntax:

$d2f$ = 144

Stack: ..., *value-word1*, *value-word2* => ..., *result*

value must be a double-precision floating point number. It is converted to a single-precision floating point number. If overflow occurs, the result must be infinity with the same sign as *value*. The result replaces *value* on the stack.

int2byte

Integer to signed byte

Syntax:

int2byte = 145

Stack: ..., *value* => ..., *result*

value must be an integer. It is truncated to a signed 8-bit result, then sign extended to an integer. The result replaces *value* on the stack.

int2char

Integer to char

Syntax:

int2char = 146

Stack: ..., *value* => ..., *result*

value must be an integer. It is truncated to an unsigned 16-bit result, then zero extended to an integer. The result replaces *value* on the stack.

int2short

Integer to short

Syntax:

int2short = 147

Stack: ..., *value* => ..., *result*

value must be an integer. It is truncated to a signed 16-bit result, then sign extended to an integer. The result replaces *value* on the stack.

3.11 Control Transfer Instructions

ifeq

Branch if equal to 0

Syntax:

ifeq = 153
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be an integer. It is popped from the stack. If *value* is zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifeq`.

ifnull

Branch if null

Syntax:

ifnull = 198
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be a reference to an object. It is popped from the stack. If *value* is `null`, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifnull`.

iflt

Branch if less than 0

Syntax:

iflt = 155
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be an integer. It is popped from the stack. If *value* is less than zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `iflt`.

ifle

Branch if less than or equal to 0

Syntax:

ifle = 158
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be an integer. It is popped from the stack. If *value* is less than or equal to zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifle`.

ifne

Branch if not equal to 0

Syntax:

ifne = 154
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be an integer. It is popped from the stack. If *value* is not equal to zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifne`.

ifnonnull

Branch if not null

Syntax:

ifnonnull = 199
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be a reference to an object. It is popped from the stack. If *value* is not `null`, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifnonnull.`

ifgt

Branch if greater than 0

Syntax:

ifgt = 157
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be an integer. It is popped from the stack. If *value* is greater than zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifgt`.

ifge

Branch if greater than or equal to 0

Syntax:

ifge = 156
branchbyte1
branchbyte2

Stack: ..., *value* => ...

value must be an integer. It is popped from the stack. If *value* is greater than or equal to zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifge`.

if_icmpeq

Branch if integers equal

Syntax:

if_icmpeq = 159
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be integers. They are both popped from the stack. If *value1* is equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpeq`.

if_icmpne

Branch if integers not equal

Syntax:

if_icmpne = 160
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be integers. They are both popped from the stack. If *value1* is not equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpne`.

if_icmplt

Branch if integer less than

Syntax:

if_icmplt = 161
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be integers. They are both popped from the stack. If *value1* is less than *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmplt`.

if_icmpgt

Branch if integer greater than

Syntax:

if_icmpgt = 163
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be integers. They are both popped from the stack. If *value1* is greater than *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpgt`.

if_icmple

Branch if integer less than or equal to

Syntax:

if_icmple = 164
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be integers. They are both popped from the stack. If *value1* is less than or equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmple`.

if_icmpge

Branch if integer greater than or equal to

Syntax:

if_icmpge = 162
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be integers. They are both popped from the stack. If *value1* is greater than or equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpge`.

lcmp

Long integer compare

Syntax:

lcmp = 148

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word1* => ..., *result*

value1 and *value2* must be long integers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value –1 is pushed onto the stack.

fcmpl

Single float compare (–1 on NaN)

Syntax:

fcmpl = 149

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

value1 and *value2* must be single-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value –1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value –1 is pushed onto the stack.

fcmpg

Single float compare (1 on NaN)

Syntax:

fcmpg = 150

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

value1 and *value2* must be single-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value –1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value 1 is pushed onto the stack.

dcmpl

Double float compare (–1 on NaN)

Syntax:

dcmpl = 151

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word1* => ..., *result*

value1 and *value2* must be double-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value –1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value –1 is pushed onto the stack.

dcmpg

Double float compare (1 on NaN)

Syntax:

dcmpg = 152

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word1* => ..., *result*

value1 and *value2* must be double-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value –1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value 1 is pushed onto the stack.

if_acmpeq

Branch if object references are equal

Syntax:

if_acmpeq = 165
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be references to objects. They are both popped from the stack. If the objects refrerenced are not the same, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the Address of this instruction. Otherwise execution proceeds at the instruction following the `if_acmpeq`.

if_acmpne

Branch if object references not equal

Syntax:

if_acmpne = 166
branchbyte1
branchbyte2

Stack: ..., *value1*, *value2* => ...

value1 and *value2* must be references to objects. They are both popped from the stack. If the objecs referenced are not the same, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_acmpne`.

goto

Branch always

Syntax:

goto = 167
branchbyte1
branchbyte2

Stack: no change

branchbyte1 and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction.

goto_w

Branch always (wide index)

Syntax:

goto_w = 200
branchbyte1
branchbyte2
branchbyte3
branchbyte4

Stack: no change

branchbyte1, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit offset. Execution proceeds at that offset from the address of this instruction.

jsr

Jump subroutine

Syntax:

jsr = 168
branchbyte1
branchbyte2

Stack: ... => ..., *return-address*

branchbyte1 and *branchbyte2* are used to construct a signed 16-bit offset. The address of the instruction immediately following the `jsr` is pushed onto the stack. Execution proceeds at the offset from the address of this instruction.

Note: The `jsr` instruction is used in the implementation of Java's `finally` keyword.

jsr_w

Jump subroutine (wide index)

Syntax:

jsr_w = 201
branchbyte1
branchbyte2
branchbyte3
branchbyte4

Stack: ... => ..., *return-address*

branchbyte1, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit offset. The address of the instruction immediately following the `jsr_w` is pushed onto the stack. Execution proceeds at the offset from the address of this instruction.

ret

Return from subroutine

Syntax:

ret = 169
vindex

Stack: no change

Local variable *vindex* in the current Java frame must contain a return address. The contents of the local variable are written into the pc.

Note that `jsr` pushes the address onto the stack, and `ret` gets it out of a local variable. This asymmetry is intentional.

Note: The `ret` instruction is used in the implementation of Java's `finally` keyword.

ret_w

Return from subroutine (wide index)

Syntax:

ret_w = 209
vindexbyte1
vindexbyte2

Stack: no change

vindexbyte1 and *vindexbyte2* are assembled into an unsigned 16-bit index to a local variable in the current Java frame. That local variable must contain a return address. The contents of the local variable are written into the pc. See the `ret` instruction for more information.

3.12 Function Return

ireturn

Return integer from function

Syntax:

ireturn = 172

Stack: ..., *value* => [empty]

value must be an integer. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

lreturn

Return long integer from function

Syntax:

lreturn = 173

Stack: ..., *value-word1*, *value-word2* => [empty]

value must be a long integer. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

freturn

Return single float from function

Syntax:

freturn = 174

Stack: ..., *value* => [empty]

value must be a single-precision floating point number. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

dreturn

Return double float from function

Syntax:

dreturn = 175

Stack: ..., *value-word1*, *value-word2* => [empty]

value must be a double-precision floating point number. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

areturn

Return object reference from function

Syntax:

areturn = 176

Stack: ..., *value* => [empty]

value must be a reference to an object. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

return

Return (void) from procedure

Syntax:

return = 177

Stack: ... => [empty]

All values on the operand stack are discarded. The interpreter then returns control to its caller.

breakpoint

Stop and pass control to breakpoint handler

Syntax:

breakpoint = 202

Stack: no change

3.13 Table Jumping

tableswitch

Access jump table by index and jump

Syntax:

tableswitch = 170
...0-3 byte pad...
default-offset1
default-offset2
default-offset3
default-offset4
low1
low2
low3
low4
high1
high2
high3
high4
...jump offsets...

Stack: ..., *index* => ...

`tableswitch` is a variable length instruction. Immediately after the `tableswitch` instruction, between zero and three 0's are inserted as padding so that the next byte begins at an address that is a multiple of four. After the padding follow a series of signed 4-byte quantities: *default-offset*, *low*, *high*, and then *high*-*low*+1 further signed 4-byte offsets. The *high*-*low*+1 signed 4-byte offsets are treated as a 0-based jump table.

The *index* must be an integer. If *index* is less than *low* or *index* is greater than *high*, then *default-offset* is added to the address of this instruction. Otherwise, *low* is subtracted from *index*, and the *index*-*low*'th element of the jump table is extracted, and added to the address of this instruction.

lookupswitch

Access jump table by key match and jump

Syntax:

lookupswitch = 171
...0-3 byte pad...
default-offset1
default-offset2
default-offset3
default-offset4
npairs1
npairs2
npairs3
npairs4
..match-offset pairs..

Stack: ..., *key* => ...

`lookupswitch` is a variable length instruction. Immediately after the `lookupswitch` instruction, between zero and three 0's are inserted as padding so that the next byte begins at an address that is a multiple of four.

Immediately after the padding are a series of pairs of signed 4-byte quantities. The first pair is special. The first item of that pair is the default offset, and the second item of that pair gives the number of pairs that follow. Each subsequent pair consists of a *match* and an *offset*.

The *key* must be an integer. The integer *key* on the stack is compared against each of the *match*es. If it is equal to one of them, the *offset* is added to the address of this instruction. If the *key* does not match any of the *match*es, the default offset is added to the address of this instruction.

3.14 Manipulating Object Fields

putfield

Set field in object

Syntax:

putfield = 181
indexbyte1
indexbyte2

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

OR

Stack: ..., *objectref*, *value-word1*, *value-word2* => ...

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a class name and a field name. The item is resolved to a field block pointer which has both the field width (in bytes) and the field offset (in bytes).

The field at that *offset* from the start of the object referenced by *objectref* will be set to the *value* on the top of the stack.

This instruction deals with both 32-bit and 64-bit wide fields.

If *objectref* is `null`, a `NullPointerException` is generated.

If the specified field is a static field, an `IncompatibleClassChangeError` is thrown.

getfield

Fetch field from object

Syntax:

getfield = 180
indexbyte1
indexbyte2

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

OR

Stack: ..., *objectref* => ..., *value-word1*, *value-word2*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a class name and a field name. The item is resolved to a field block pointer which has both the field width (in bytes) and the field *offset* (in bytes).

objectref must be a reference to an object. The value at *offset* into the object referenced by *objectref* replaces *objectref* on the top of the stack.

This instruction deals with both 32-bit and 64-bit wide fields.

If *objectref* is `null`, a `NullPointerException` is generated.

If the specified field is a static field, an `IncompatibleClassChangeError` is thrown.

putstatic

Set static field in class

Syntax:

putstatic = 179
indexbyte1
indexbyte2

Stack: ..., *value* => ...

OR

Stack: ..., *value-word1*, *value-word2* => ...

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field will be set to have the value on the top of the stack.

This instruction works for both 32-bit and 64-bit wide fields.

If the specified field is a dynamic field, an `IncompatibleClassChangeError` is thrown.

getstatic

Get static field from class

Syntax:

getstatic = 178
indexbyte1
indexbyte2

Stack: ..., => ..., *value*

OR

Stack: ..., => ..., *value-word1*, *value-word2*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class.

This instruction deals with both 32-bit and 64-bit wide fields.

If the specified field is a dynamic field, an `IncompatibleClassChangeError` is generated.

3.15 Method Invocation

There are four instructions that implement method invocation.

`invokevirtual` Invoke an instance method of an object, dispatching based on the runtime (virtual) type of the object. This is the normal method dispatch in Java.

`invokenonvirtual` Invoke an instance method of an object, dispatching based on the compile-time (non-virtual) type of the object. This is used, for example, when the keyword `super` or the name of a superclass is used as a method qualifier.

`invokestatic` Invoke a class (`static`) method in a named class.

`invokeinterface` Invoke a method which is implemented by an interface, searching the methods implemented by the particular run-time object to find the appropriate method.

invokevirtual

Invoke instance method, dispatch based on run-time type

Syntax:

invokevirtual = 182
indexbyte1
indexbyte2

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain a reference to an object and some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object reference. The method signature is looked up in the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is an index into the method table of the named class, which is used with the object's dynamic type to look in the method table of that type, where a pointer to the method block for the matched method is found. The method block indicates the type of method (`native`, `synchronized`, and so on) and the number of arguments expected on the operand stack.

If the method is marked `synchronized` the monitor associated with *objectref* is entered.

The *objectref* and arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If the object reference on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

invokenonvirtual

Invoke instance method, dispatching based on compile-time type

Syntax:

invokenonvirtual = 183
indexbyte1
indexbyte2

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain a reference to an object and some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a complete method signature and class. The method signature is looked up in the method table of the class indicated. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, and so on) and the number of arguments (nargs) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with *objectref* is entered.

The *objectref* and arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If the object reference on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

invokestatic

Invoke a class (static) method

Syntax:

invokestatic = 184
indexbyte1
indexbyte2

Stack: ..., [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature and class. The method signature is looked up in the method table of the class indicated. The method signature is guaranteed to exactly match one of the method signatures in the class's method table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, and so on) and the number of arguments (nargs) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with the class is entered.

The arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

invokeinterface

Invoke interface method

Syntax:

invokeinterface = 185
indexbyte1
indexbyte2
nargs
reserved

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain a reference to an object and *nargs*-1 arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object reference. The method signature is looked up in the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, and so on) but unlike `invokevirtual` and `invokenonvirtual`, the number of available arguments (nargs) is taken from the bytecode.

If the method is marked `synchronized` the monitor associated with *objectref* is entered.

The *objectref* and arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If the *objectref* on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

3.16 Exception Handling

athrow

Throw exception or error

Syntax:

athrow = 191

Stack: ..., *objectref* => [undefined]

objectref must be a reference to an object which is a subclass of `Throwable`, which is thrown. The current Java stack frame is searched for the most recent catch clause that catches this class or a superclass of this class. If a matching catch list entry is found, the pc is reset to the address indicated by the catch-list entry, and execution continues there.

If no appropriate catch clause is found in the current stack frame, that frame is popped and the object is rethrown. If one is found, it contains the location of the code for this exception. The pc is reset to that location and execution continues. If no appropriate catch is found in the current stack frame, that frame is popped and the *objectref* is rethrown.

If *objectref* is `null`, then a `NullPointerException` is thrown instead.

3.17 Miscellaneous Object Operations

new

Create new object

Syntax:

new = 187
indexbyte1
indexbyte2

Stack: ... => ..., *objectref*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index must be a class name that can be resolved to a class pointer, *class*. A new instance of that class is then created and a reference to the object is pushed on the stack.

checkcast

Make sure object is of given type

Syntax:

checkcast = 192
indexbyte1
indexbyte2

Stack: ..., *objectref* => ..., *objectref*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name which can be resolved to a class pointer, *class*. *objectref* must be a reference to an object.

`checkcast` determines whether *objectref* can be cast to be a reference to an object of class *class*. A `null` *objectref* can be cast to any *class*. Otherwise the referenced object must be an instance of *class* or one of its superclasses. (See the *Java Language Specification* for information on how to determine whether a *objectref* is an instance of a class.) If *objectref* can be cast to *class* execution proceeds at the next instruction, and the *objectref* remains on the stack.

If *objectref* cannot be cast to *class*, a `ClassCastException` is thrown.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

instanceof

Determine if an object is of given type

Syntax:

instanceof = 193
indexbyte1
indexbyte2

Stack: ..., *objectref* => ..., *result*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name which can be resolved to a class pointer, *class*. *objectref* must be a reference to an object.

instanceof determines whether *objectref* can be cast to be a reference to an object of the class *class*. This instruction will overwrite *objectref* with 1 if *objectref* is an instance of *class* or one of its superclasses. (See the *Java Language Specification* for information on how to determine whether a object reference is an instance of a class.) Otherwise, *objectref* is overwritten by 0. If *objectref* is `null`, it's overwritten by 0.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

3.18 Monitors

monitorenter

Enter monitored region of code

Syntax:

monitorenter = 194

Stack: ..., *objectref* => ...

objectref must be a reference to an object.

The interpreter attempts to obtain exclusive access via a lock mechanism to *objectref*. If another thread already has *objectref* locked, than the current thread waits until the object is unlocked. If the current thread already has the object locked, then continue execution. If the object is not locked, then obtain an exclusive lock.

If *objectref* is `null`, then a `NullPointerException` is thrown instead.

monitorexit

Exit monitored region of code

Syntax:

monitorexit = 195

Stack: ..., *objectref* => ...

objectref must be a reference to an object.

The lock on the object released. If this is the last lock that this thread has on that object (one thread is allowed to have multiple locks on a single object), then other threads that are waiting for the object to be available are allowed to proceed.

If *objectref* is `null`, then a `NullPointerException` is thrown instead.

Appendix A: An Optimization

The following set of pseudo-instructions suffixed by _quick are variants of Java virtual machine instructions. They are used to improve the speed of interpreting bytecodes. They are not part of the virtual machine specification or instruction set, and are invisible outside of an Java virtual machine implementation. However, inside a virtual machine implementation they have proven to be an effective optimization.

A compiler from Java source code to the Java virtual machine instruction set emits only non-_quick instructions. If the _quick pseudo-instructions are used, each instance of a non-_quick instruction with a _quick variant is overwritten on execution by its _quick variant. Subsequent execution of that instruction instance will be of the _quick variant.

In all cases, if an instruction has an alternative version with the suffix _quick, the instruction references the constant pool. If the _quick optimization is used, each non-_quick instruction with a _quick variant performs the following:

- Resolves the specified item in the constant pool
- Signals an error if the item in the constant pool could not be resolved for some reason
- Turns itself into the _quick version of the instruction. The instructions `putstatic`, `getstatic`, `putfield`, and `getfield` each have two _quick versions.
- Performs its intended operation

This is identical to the action of the instruction without the _quick optimization, except for the additional step in which the instruction overwrites itself with its _quick variant.

The _quick variant of an instruction assumes that the item in the constant pool has already been resolved, and that this resolution did not generate any errors. It simply performs the intended operation on the resolved item.

Note: some of the invoke methods only support a single-byte offset into the method table of the object; for objects with 256 or more methods some invocations cannot be "quicked" with only these bytecodes. We also need to define or change existing `getfield` and `putfield` bytecodes to support more than a byte of *offset*.

This Appendix doesn't give the opcode values of the pseudo-instructions, since they are invisible and subject to change.

A.1 Constant Pool Resolution

When the class is read in, an array `constant_pool[]` of size `nconstants` is created and assigned to a field in the class. `constant_pool[0]` is set to point to a dynamically allocated array which indicates which fields in the constant_pool have already been resolved. `constant_pool[1]` through `constant_pool[nconstants - 1]` are set to point at the "type" field that corresponds to this constant item.

When an instruction is executed that references the constant pool, an index is generated, and `constant_pool[0]` is checked to see if the index has already been resolved. If so, the value of `constant_pool[index]` is returned. If not, the value of `constant_pool[index]` is resolved to be the actual pointer or data, and overwrites whatever value was already in `constant_pool[index]`.

A.2 Pushing Constants onto the Stack (_quick variants)

ldc1_quick

Push item from constant pool onto stack

Syntax:

ldc1_quick
indexbyte1

Stack: ... => ..., *item*

indexbyte1 is used as an unsigned 8-bit index into the constant pool of the current class. The *item* at that index is pushed onto the stack.

ldc2_quick

Push item from constant pool onto stack

Syntax:

ldc2_quick
indexbyte1
indexbyte2

Stack: ... => ..., *item*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The *constant* at that index is resolved and the *item* at that index is pushed onto the stack.

ldc2w_quick

Push long integer or double float from constant pool onto stack

Syntax:

ldc2w_quick
indexbyte1
indexbyte2

Stack: ... => ..., *constant-word1*, *constant-word2*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The *constant* at that index is pushed onto the stack.

A.3 Managing Arrays (_quick variants)

anewarray_quick

Allocate new array of references to objects

Syntax:

anewarray_quick
indexbyte1
indexbyte2

Stack: ..., *size* => *result*

size must be an integer. It represents the number of elements in the new array.

indexbyte1 and *indexbyte2* are are used to construct an index into the constant pool of the current class. The entry must be a class.

A new array of the indicated class type and capable of holding *size* elements is allocated, and *result* is a reference to this new array. Allocation of an array large enough to contain *size* items of the given class type is attempted. All elements of the array are initialized to zero.

If *size* is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

multianewarray_quick

Allocate new multi-dimensional array

Syntax:

multianewarray_quick
indexbyte1
indexbyte2
dimensions

Stack: ..., *size1, size2, ...sizen* => *result*

Each *size* must be an integer. Each represents the number of elements in a dimension of the array.

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The resulting entry must be a class.

dimensions has the following aspects:

- It must be an integer ≥ 1.
- It represents the number of dimensions being created. It must be ≤ the number of dimensions of the array class.
- It represents the number of elements that are popped off the stack. All must be integers greater than or equal to zero. These are used as the sizes of the dimension.

If any of the *size* arguments on the stack is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

The *result* is a reference to the new array object.

Note: More explanation needed about how this is an array of arrays.

A.4 Manipulating Object Fields (_quick variants)

putfield_quick

Set field in object

Syntax:

putfield_quick
offset
unused

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

objectref must be a reference to an object. *value* must be a value of a type appropriate for the specified field. *offset* is the offset for the field in that object. *value* is written at *offset* into the object. Both *objectref* and *value* are popped from the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

putfield2_quick

Set long integer or double float field in object

Syntax:

putfield2_quick
offset
unused

Stack: ..., *objectref*, *value-word1*, *value-word2*=> ...

objectref must be a reference to an object. *value* must be a value of a type appropriate for the specified field. *offset* is the offset for the field in that object. *value* is written at *offset* into the object . Both *objectref* and *value* are popped from the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

getfield_quick

Fetch field from object

Syntax:

getfield_quick
offset
unused

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

objectref must be a handle to an object. The value at *offset* into the object referenced by *objectref* replaces *objectref* on the top of the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

getfield2_quick

Fetch field from object

Syntax:

getfield2_quick
offset
unused

Stack: ..., *objectref* => ..., *value-word1*, *value-word2*

objectref must be a handle to an object. The value at *offset* into the object referenced by *objectref* replaces *objectref* on the top of the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

putstatic_quick

Set static field in class

Syntax:

putstatic_quick
indexbyte1
indexbyte2

Stack: ..., *value* => ...

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. *value* must be the type appropriate to that field. That field will be set to have the value *value.*

putstatic2_quick

Set static field in class

Syntax:

putstatic2_quick
indexbyte1
indexbyte2

Stack: ..., *value-word1*, *value-word2* => ...

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field must either be a long integer or a double precision floating point number. *value* must be the type appropriate to that field. That field will be set to have the value *value.*

getstatic_quick

Get static field from class

Syntax:

getstatic_quick
indexbyte1
indexbyte2

Stack: ..., => ..., *value*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The value of that field will replace *handle* on the stack.

getstatic2_quick

Get static field from class

Syntax:

getstatic2_quick
indexbyte1
indexbyte2

Stack: ..., => ..., *value-word1*, *value-word2*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The field must be a long integer or a double precision floating point number. The value of that field will replace *handle* on the stack

A.5 Method Invocation (_quick variants)

invokevirtual_quick

Invoke instance method, dispatching based on run-time type

Syntax:

invokevirtual_quick
offset
nargs

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object and *nargs-1* arguments. The method block at *offset* in the object's method table, as determined by the object's dynamic type, is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.).

If the method is marked `synchronized` the monitor associated with the object is entered.

The base of the local variables array for the new Java stack frame is set to point to *objectref* on the stack, making *objectref* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

invokevirtualobject_quick

Invoke instance method of class `java.lang.Object`, specifically for benefit of arrays

Syntax:

invokevirtualobject_quick
offset
nargs

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object or to an array and *nargs-1* arguments. The method block at *offset* in `java.lang.Object`'s method table is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.).

If the method is marked `synchronized` the monitor associated with *handle* is entered.

The base of the local variables array for the new Java stack frame is set to point to *objectref* on the stack, making *objectref* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

invokenonvirtual_quick

Invoke instance method, dispatching based on compile-time type

Syntax:

invokenonvirtual_quick
indexbyte1
indexbyte2

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object and some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with the object is entered.

The base of the local variables array for the new Java stack frame is set to point to *objectref* on the stack, making *objectref* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

invokestatic_quick

Invoke a class (static) method

Syntax:

invokestatic_quick
indexbyte1
indexbyte2

Stack: ..., [*arg1*, [*arg2* ...]] => ...

The operand stack must contain some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with the method's class is entered.

The base of the local variables array for the new Java stack frame is set to point to the first argument on the stack, making the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

invokeinterface_quick

Invoke interface method

Syntax:

invokeinterface_quick
idbyte1
idbyte2
nargs
guess

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object, and *nargs*-1 arguments. *idbyte1* and *idbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object *handle*.

The method signature is searched for in the object's method table. As a short-cut, the method signature at slot *guess* is searched first. If that fails, a complete search of the method table is performed. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, etc.) but the number of available arguments (*nargs*) is taken from the bytecode.

If the method is marked `synchronized` the monitor associated with *handle* is entered.

The base of the local variables array for the new Java stack frame is set to point to *handle* on the stack, making *handle* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

guess is the last guess. Each time through, guess is set to the method offset that was used.

A.6 Miscellaneous Object Operations (_quick variants)

new_quick

Create new object

Syntax:

new_quick
indexbyte1
indexbyte2

Stack: ... => ..., *objectref*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index must be a class. A new instance of that class is then created and *objectref*, a reference to that object is pushed on the stack.

checkcast_quick

Make sure object is of given type

Syntax:

checkcast_quick
indexbyte1
indexbyte2

Stack: ..., *objectref* => ..., *objectref*

objectref must be a reference to an object. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The object at that index of the constant pool must have already been resolved.

`checkcast` then determines whether *objectref* can be cast to a reference to an object of class *class*. A `null` reference can be cast to any *class*, and otherwise the superclasses of *objectref*'s type are searched for *class*. If class is determined to be a superclass of *objectref*'s type, or if *objectref* is null, it can be cast to *objectref* cannot be cast to *class*, a `ClassCastException` is thrown.

Note: here (and probably in other places) we assume casts don't change the reference; this is implementation dependent.

instanceof_quick

Determine if object is of given type

Syntax:

instanceof_quick
indexbyte1
indexbyte2

Stack: ..., *objectref* => ..., *result*

objectref must be a reference to an object. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item of class *class* at that index of the constant pool must have already been resolved.

`instanceof` determines whether *objectref* can be cast to an object of the class *class*. A `null` *objectref* can be cast to any *class*, and otherwise the superclasses of *objectref*'s type are searched for *class*. If *class* is determined to be a superclass of *objectref*'s type, *result* is 1 (`true`). Otherwise, *result* is 0 (`false`). If *handle* is `null`, *result* is 0 (`false`).

Index of Instructions