The Java(tm) Virtual Machine Specification

Table of Contents

Copyright Information
Preface
1 - Java Virtual Machine Architecture
   1.1 - Supported Data Types
   1.2 - Registers
   1.3 - Local Variables
   1.4 - The Operand Stack
   1.5 - Execution Environment
   1.6 - Garbage Collected Heap
   1.7 - Method Area
   1.8 - The Java Instruction Set
   1.9 - Limitations
2 - Class File Format
   2.1 - Format
   2.2 - Signatures
   2.3 - Constant Pool
   2.4 - Fields
   2.5 - Methods
   2.6 - Attributes
3 - The Virtual Machine Instruction Set
   3.1 - Format for the Instructions
   3.2 - Pushing Constants onto the Stack
   3.3 - Loading Local Variables Onto the Stack
   3.4 - Storing Stack Values into Local Variables
   3.5 - Wider index for Loading, Storing and Incrementing
   3.6 - Managing Arrays
   3.7 - Stack Instructions
Copyright Information

2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This BETA quality release and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this release or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc. and the University of California, respectively. Third-party font software in this release is protected by copyright and licensed from Sun's Font Suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The release described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, Sun Microsystems, Sun Microsystems Computer Corporation, the Sun logo, the Sun Microsystems Computer Corporation logo, WebRunner, Java, FirstPerson and the FirstPerson logo and agent are trademarks or registered trademarks of Sun Microsystems, Inc. The "Duke" character is a trademark of Sun Microsystems, Inc. and Copyright (c) 1992-1995 Sun Microsystems, Inc. All Rights Reserved. UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun® Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing
the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.
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% compatibility 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:

```java
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. Instead, 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
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.

### 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.

**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.

**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 occurring 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.

### 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

2.1 - Format
2.2 - Signatures
2.3 - Constant Pool
2.4 - Fields
2.5 - Methods
2.6 - Attributes

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:

```java
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](http://sunsite.ee/java/vmspec/vmspec-6.html).

**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:

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

**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.

**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.

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT_Class</td>
<td>7</td>
</tr>
<tr>
<td>CONSTANT_Fieldref</td>
<td>9</td>
</tr>
<tr>
<td>CONSTANT_Methodref</td>
<td>10</td>
</tr>
<tr>
<td>CONSTANT_InterfaceMethodref</td>
<td>11</td>
</tr>
<tr>
<td>CONSTANT_String</td>
<td>8</td>
</tr>
<tr>
<td>CONSTANT_Integer</td>
<td>3</td>
</tr>
<tr>
<td>CONSTANT_Float</td>
<td>4</td>
</tr>
<tr>
<td>CONSTANT_Long</td>
<td>5</td>
</tr>
<tr>
<td>CONSTANT_Double</td>
<td>6</td>
</tr>
<tr>
<td>CONSTANT_NameAndType</td>
<td>12</td>
</tr>
<tr>
<td>CONSTANT_Utf8</td>
<td>1</td>
</tr>
<tr>
<td>CONSTANT_Unicode</td>
<td>2</td>
</tr>
</tbody>
</table>

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

**CONSTANT_Class**

CONSTANT_Class is used to represent a class or an interface.

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

```java
int[][]
```

is

```java
[[I
```

The class name for

```java
Thread[]
```

is

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

**CONSTANT_{Fieldref,Methodref,InterfaceMethodref}**

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

```java
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

```java
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.

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.

**CONSTANT_Integer and CONSTANT_Float**

CONSTANT_Integer and CONSTANT_Float represent four-byte constants.

```plaintext
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.

**CONSTANT_Long and CONSTANT_Double**

CONSTANT_Long and CONSTANT_Double represent eight-byte constants.

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

CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
```
All eight-byte constants take up two spots in the constant pool. If this is the nth 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.

**CONSTANT_NameAndType**

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

```c
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.

**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.

```java
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:

```c
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 may 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 may 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.

**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.

**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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant Pool Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>CONSTANT_Long</td>
</tr>
<tr>
<td>float</td>
<td>CONSTANT_Float</td>
</tr>
<tr>
<td>double</td>
<td>CONSTANT_Double</td>
</tr>
<tr>
<td>int, short, char, byte, boolean</td>
<td>CONSTANT_Integer</td>
</tr>
</tbody>
</table>
**Code**

The "Code" attribute has the following format:

```plaintext
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.
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.

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_index+table[i]] is an Exception that this class is declared to throw.
**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:

```plaintext
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.

**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[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<sup>th</sup> local variable in the method's frame.
3 The Virtual Machine Instruction Set

3.1 - Format for the Instructions
3.2 - Pushing Constants onto the Stack
3.3 - Loading Local Variables Onto the Stack
3.4 - Storing Stack Values into Local Variables
3.5 - Wider index for Loading, Storing and Incrementing
3.6 - Managing Arrays
3.7 - Stack Instructions
3.8 - Arithmetic Instructions
3.9 - Logical Instructions
3.10 - Conversion Operations
3.11 - Control Transfer Instructions
3.12 - Function Return
3.13 - Table Jumping
3.14 - Manipulating Object Fields
3.15 - Method Invocation
3.16 - Exception Handling
3.17 - Miscellaneous Object Operations
3.18 - Monitors
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

| opcode = number |
| operand1      |
| operand2      |
| ...           |

Stack: ¼., 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

\[
\text{bipush} = 16 \\
\text{byte1}
\]

Stack: \( ... \Rightarrow ... , \text{value} \)

`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

\[
\text{sipush} = 17 \\
\text{byte1} \\
\text{byte2}
\]

Stack: \( ... \Rightarrow ... , \text{item} \)

`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

```
| 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

```
| 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
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

\[\text{aconst_null} = 1\]

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

Push the null object reference onto the stack.

**iconst_m1**

Push integer constant -1

\[\text{iconst_m1} = 2\]

Stack: ... => ..., -1

Push the integer -1 onto the stack.
**iconst_<n>**

Push integer constant

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

Forms: icontst_0 = 3, icontst_1 = 4, icontst_2 = 5, icontst_3 = 6, icontst_4 = 7, icontst_5 = 8

Push the integer <n> onto the stack.

**lconst_<l>**

Push long integer constant

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

Stack: ... => ..., <f>
Forms: fconst_0 = 11, fconst_1 = 12, fconst_2 = 13

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

\textbf{dconst_\textless d\textgreater}

Push double float

Stack: \ldots \rightarrow \ldots, \textless d\textgreater\text{-}word1, \textless d\textgreater\text{-}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

| 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

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.
Load long integer from local variable

```
| 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

```
| 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

```
| fload = 23 |
| vindex    |
```

Stack: ... => ..., value
The value of the local variable at vindex in the current Java frame is pushed onto the operand stack.

\textbf{fload}_{<n>}

Load single float from local variable

\begin{center}
\begin{tabular}{|c|}
\hline
fload_{<n>} \\
\hline
\end{tabular}
\end{center}

Stack: $\ldots \Rightarrow \ldots, \text{value}$

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 \textit{vindex} of <n>, except that the operand <n> is implicit.

\textbf{dload}

Load double float from local variable

\begin{center}
\begin{tabular}{|c|}
\hline
dload = 24 \\
vindex \\
\hline
\end{tabular}
\end{center}

Stack: $\ldots \Rightarrow \ldots, \text{value-word1, value-word2}$

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

\textbf{dload}_{<n>}

Load double float from local variable
vmspec: Loading Local Variables Onto the Stack

\(\text{dload}_n\)

Stack: \(\ldots \Rightarrow \ldots, \text{value-word1, value-word2}\)

Forms: \(\text{dload}_0 = 38, \text{dload}_1 = 39, \text{dload}_2 = 40, \text{dload}_3 = 41\)

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 \(\text{dload}\) with a \(vindex\) of \(n\), except that the operand \(n\) is implicit.

\(\text{aload}\)

Load object reference from local variable

\(\text{aload} = 25\)

\(vindex\)

Stack: \(\ldots \Rightarrow \ldots, \text{value}\)

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

\(\text{aload}_n\)

Load object reference from local variable

\(\text{aload}_n\)

Stack: \(\ldots \Rightarrow \ldots, \text{value}\)

Forms: \(\text{aload}_0 = 42, \text{aload}_1 = 43, \text{aload}_2 = 44, \text{aload}_3 = 45\)
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

\[
\text{istore} = 54
\]

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

\[
\text{istore}_n
\]

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.

**Istore**
Store long integer into local variable

\[
\begin{array}{c}
\text{lstore} = 55 \\
vindex
\end{array}
\]

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.

\text{lstore_\text{n}}

Store long integer into local variable

\[
\begin{array}{c}
lstore_\text{n} \\
vindex
\end{array}
\]

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 \text{n} and \text{n}+1 in the current Java frame are set to value.

This instruction is the same as \text{lstore} with a \text{vindex} of \text{n}, except that the operand \text{n} is implicit.

\text{fstore}

Store single float into local variable

\[
\begin{array}{c}
fstore = 56 \\
vindex
\end{array}
\]

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

value must be a long integer. Local variables vindex and vindex+1 in the current Java frame are set to value.
value must be a single-precision floating point number. Local variable vindex in the current Java frame is set to value.

\textbf{fstore}_{<n>}

Store single float into local variable

Stack: \ldots, value \Rightarrow \ldots

Forms: \texttt{fstore\_0} = 67, \texttt{fstore\_1} = 68, \texttt{fstore\_2} = 69, \texttt{fstore\_3} = 70

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

This instruction is the same as \texttt{fstore} with a \texttt{vindex} of \texttt{<n>}, except that the operand \texttt{<n>} is implicit.

\textbf{dstore}

Store double float into local variable

\begin{tabular}{cc}
\texttt{dstore} = & 57 \\
\texttt{vindex} & \\
\end{tabular}

Stack: \ldots, value-word1, value-word2 \Rightarrow \ldots

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

\textbf{dstore}_{<n>}

vmspec: Storing Stack Values into Local Variables

\[
\text{\texttt{dstore}_{n}}
\]

Store double float into local variable

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

Forms: \texttt{dstore}_0 = 71, \texttt{dstore}_1 = 72, \texttt{dstore}_2 = 73, \texttt{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 \texttt{dstore} with a \texttt{vindex} of \(<n>\), except that the operand \(<n>\) is implicit.

\texttt{astore}

Store object reference into local variable

\[
\text{\texttt{astore} = 58}
\]

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

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

\texttt{astore}_{n}

\[
\text{\texttt{astore}_{n}}
\]

Store object reference into local variable

Stack: ..., value => ...
vmspec: Storing Stack Values into Local Variables

Forms: 
\[
\text{astore}_0 = 75, \text{astore}_1 = 76, \text{astore}_2 = 77, \text{astore}_3 = 78
\]

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 \text{astore} with a \textit{vindex} of \(<n>\), except that the operand \(<n>\) is implicit.

\textbf{iinc}

Increment local variable by constant

\[
\begin{array}{c}
\text{iinc} = 132 \\
\text{vindex} \\
\text{const}
\end{array}
\]

Stack: no change

Local variable \textit{vindex} in the current Java frame must contain an integer. Its value is incremented by the value \textit{const}, where \textit{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.

\[
\begin{array}{|c|}
\hline
\text{wide} = 196 \\
\text{vindex2} \\
\hline
\end{array}
\]

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

```
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
TDOUBLE 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

```java
anewarray = 189
indexbyte1
indexbyte2
```

of references to objects

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:

```java
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:

```java
new int[6][]
```
is created with the following code:

```java
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

```
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 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

```java
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

``arraylength = 190``

Get length of array

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

``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

```
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

```
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

\[
\text{daload} = 49
\]

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

\[
\text{aaload} = 50
\]

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.

\[
\text{baload} = 51
\]
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.

\textbf{caload}

Load character from array

\begin{center}
\texttt{caload = 52}
\end{center}

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.

\textbf{saload}

Load short from array

\begin{center}
\texttt{saload = 53}
\end{center}

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

```
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

```
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.
f astore

Store into single float array

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.

d astore

Store into double float array

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.

a astore

Store into object reference array

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

arrayref must be an object array reference, index must be an integer, and value an object reference. The object reference 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.
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

```java
bastore = 84
```

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
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**

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 a 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

\[
\text{nop} = 0
\]

Stack: no change

Do nothing.

**pop**

Pop top stack word

\[
\text{pop} = 87
\]

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

Pop the top word from the stack.

**pop2**

Pop top two stack word

\[
\text{pop2} = 88
\]

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


Pop the top two words from the stack.

**dup**

Duplicate top stack word

\[ \text{dup} = 89 \]

Stack: \(..., \text{any} \Rightarrow \text{...}, \text{any}, \text{any} \)

Duplicate the top word on the stack.

**dup2**

Duplicate top two stack word

\[ \text{dup2} = 92 \]

Stack: \(..., \text{any2, any1} \Rightarrow \text{...}, \text{any2, any1, any2, any1} \)

Duplicate the top two words on the stack.

**dup_x1**

Duplicate top stack word and put two down

\[ \text{dup}_x1 = 90 \]

Stack: \(..., \text{any2, any1} \Rightarrow \text{...}, \text{any1, any2, any1} \)

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

\[ \text{dup2}_x1 = 93 \]

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.

\[ \text{dup}_x2 \]

Duplicate top stack word and put three down

\[ \text{dup}_x2 = 91 \]

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.

\[ \text{dup2}_x2 \]

Duplicate top two stack words and put three down

\[ \text{dup2}_x2 = 94 \]

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.

\[ \text{swap} \]

Swap top two stack words

\[ \text{swap} = 95 \]

Stack: ..., any2, any1 => ..., any2, any1
Swap the top two elements on the stack.