JEFF specification with notes on the ISO comments

Notations: The modifications corresponding to the answer to each Member Body are preceded by the reference to the Member Body and the comment number.

Example:

| NSI 3, JISC 10, SIS 5 | Original Text | Modified Text |
Copyright 2000, 2001, J Consortium, All rights reserved

Permission is granted by the J Consortium to reproduce this International Specification for the purpose of review and comment, provided this notice is included. All other rights are reserved.

THIS SPECIFICATION IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY IMPLEMENTATION OF THIS SPECIFICATION SHALL BE MADE ENTIRELY AT THE IMPLEMENTER’S OWN RISK, AND NEITHER THE J CONSORTIUM, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER DIRECTLY OR INDIRECTLY ARISING FROM THE IMPLEMENTATION OF THIS SPECIFICATION.

J Consortium and the J Consortium logo are trademarks or service marks, or registered trademarks and service marks, of J Consortium, Inc. in the U.S. and a trademark in other countries.

JEFF is a trademark of J Consortium, Inc.

Java is a registered trademark of Sun Microsystems, Inc. in the United States and in other countries.

J Consortium Specification No. 2000-02.1
JEFF File Format

1 Introduction.................................................................................................................. 5
  1.1 What is JEFF........................................................................................................... 5
      1.1.1 Benefits ..................................................................................................... 5
  1.2 Scope ..................................................................................................................... 6
  1.3 References ............................................................................................................ 6

2 Data Types .................................................................................................................... 7
  2.1 Basic Types .......................................................................................................... 8
  2.2 Language Types .................................................................................................. 8
      2.2.1 Definition ................................................................................................. 10
      2.2.2 Comparison .............................................................................................. 10
      2.2.3 Representation ......................................................................................... 10
  2.3 Specific Types ..................................................................................................... 12
      2.3.1 Access flags .............................................................................................. 12
      2.3.2 Type Descriptor ......................................................................................... 14
      2.3.3 Offsets .................................................................................................... 18
      2.3.4 Index Values ............................................................................................. 19

3 File Structure .............................................................................................................. 20
  3.1 Definitions ............................................................................................................ 20
      3.1.1 Fully Qualified Names ............................................................................. 21
      3.1.2 Symbolic Names .................................................................................... 21
      3.1.3 Internal Classes and External Classes ..................................................... 23
      3.1.4 Fields and Methods ................................................................................ 24
      3.1.5 Field Position .......................................................................................... 24
  3.2 Conventions ......................................................................................................... 25
      3.2.1 Notations ................................................................................................ 26
      3.2.2 Byte Order .............................................................................................. 26
      3.2.3 Alignment and Padding ........................................................................ 27
  3.3 Definition of the File Structures .......................................................................... 28
      3.3.1 File Header ............................................................................................... 29
      3.3.2 Class Section .......................................................................................... 33
          3.3.2.1 Class Header .................................................................................... 34
          3.3.2.2 Interface Table ............................................................................... 38
          3.3.2.3 Referenced Class Table ................................................................. 38
          3.3.2.4 Internal Field Table ........................................................................ 38
          3.3.2.5 Internal Method Table .................................................................. 39
          3.3.2.6 Referenced Field Table ................................................................. 41
          3.3.2.7 Referenced Method Table ............................................................. 41
          3.3.2.8 Bytecode Block Structure .............................................................. 42
          3.3.2.9 Exception Table List .................................................................... 44
          3.3.2.10 Constant Data Section ................................................................. 45
      3.3.3 Attributes Section ...................................................................................... 47
          3.3.3.1 Attribute Type ............................................................................... 48
          3.3.3.2 Class Attributes ............................................................................ 49
          3.3.3.3 Attribute Table ............................................................................... 49
      3.3.4 Symbolic Data Section .............................................................................. 50
1 Introduction

1.1 What is JEFF

This document describes the JEFF File Format. This format is designed to download and store on a platform object oriented programs written in portable code. The distribution of applications is not the target of this specification.

The goal of JEFF is to provide a ready-for-execution format allowing programs to be executed directly from static memory, thus avoiding the necessity to recopy classes into dynamic runtime memory for execution.

The constraints put on the design of JEFF are the following:

- Any set of class files must be translatable into a single JEFF file.
- JEFF must be a ready-for-execution format. A virtual machine can use it efficiently, directly from static memory (ROM, flash memory…). No copy in dynamic runtime memory or extra data modification shall be needed.
- All the standard behaviors and features of a virtual machine such as Java™ virtual machine must be reproducible using JEFF.
- In particular, JEFF must facilitate “symbolic linking” of classes. The replacement of a class definition by another class definition having a compatible signature (same class name, same fields and same method signatures) must not require any modifications in the other class definitions.

The main consequences of these choices are:

- A JEFF file can contain several classes from several packages. The content can be a complete application, parts of it, or only one class.
- To allow “symbolic linking” of classes, the references between classes must be kept at the symbolic level, even within a single JEFF file.
- The binary content of a JEFF file is adapted to be efficiently read by a wide range of processors (with different byte orders, alignments, etc.).
- JEFF is also a highly efficient format for the dynamic downloading of class definitions to dynamic memory (RAM).

The limitations introduced by the use of JEFF are described in chapter 5 Restrictions.

1.1.1 Benefits

JEFF is a file format standard, which allows storing on-platform non pre-linked classes in a form that does not require any modification for efficient execution. JEFF exhibits a large range of benefits:

- The first of these benefits is that classes represented with JEFF can be executed directly from storage memory, without requiring any loading into runtime memory in order to be translated in a format adequate for execution. This results in a dramatic economy of runtime memory: programs with a size of several hundreds of kilobytes may then be executed with only a few kilobytes of dynamic runtime memory thanks to JEFF.
- The second benefit of JEFF is the saving of the processing time usually needed at the start of an execution to load into dynamic memory the stored classes.
- The third benefit is that JEFF does not require the classes to be pre-linked, hence fully preserving the flexibility of portable code technologies. With JEFF, programs can be
updated on-platform by the mere replacement of some individual classes without requiring to replace the complete program. This provides a decisive advantage over previously proposed "ready-for-execution" formats providing only pre-linked programs.

- A last benefit of JEFF is that it allows a compact storage of programs, twice smaller than usual class file format, and this without any compression.

1.2 Scope

JEFF can be used with benefits on all kinds of platform.

JEFF’s most immediate interest is for deploying portable applications on small footprint devices. JEFF provides dramatic savings of dynamic memory and execution time without sacrificing any of the flexibility usually attached to the use of non-pre-linked portable code.

JEFF is especially important to provide a complete solution to execute portable programs of which code size is bigger than the available dynamic memory.

JEFF is also very important when fast reactivity of programs is important. By avoiding the extra-processing related to loading into dynamic memory and formatting classes at runtime, JEFF provides a complete answer to the problem of class-loading slow-down.

These benefits are particularly interesting for small devices supporting financial applications. Such applications are often complex and relying on code of significant size, while the pressure of the market often imposes to these devices to be of a low price and, consequently, to be very small footprint platforms. In addition, to not impose unacceptable delays to customers, it is important these applications to not waste time in loading classes into dynamic memory when they are launched but, on the contrary, to be immediately actively processing the transaction with no delay. When using smart cards, there are also some loose real-time constraints that are better handled if it can be granted that no temporary freezing of processing can occur due to class loading.

JEFF can also be of great benefit for devices dealing with real-time applications. In this case, avoiding the delays due to class loading can play an important role to satisfy real-time constraints.

1.3 References

This document is a self-contained specification of the JEFF format standard. However, to ease the understanding of this specification, the reading of the following document is recommended as informative reference:

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>

Copyright 2000, 2001, J Consortium, All Rights Reserved
1.4 Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Logical entity that provides a set of related fields and methods. The class is a basic element for object-oriented languages.</td>
</tr>
<tr>
<td>Package</td>
<td>Set of classes</td>
</tr>
<tr>
<td>bytecode</td>
<td>A bytecode is the binary value of the encoding of a JEFF instruction. By extension, bytecode is used to designate the instruction itself.</td>
</tr>
<tr>
<td>cell</td>
<td>4-octet word used by bytecode interpreters.</td>
</tr>
<tr>
<td>byte</td>
<td>an octet: representation of an unsigned 8-bit value</td>
</tr>
</tbody>
</table>

The next reference is a normative reference:

2 Data Types

This chapter describes the data types used by the JEFF format specification. All the values in a JEFF file are stored on one, two, four or eight bytes. In this document, the expression "null value" is synonym of a value of zero.

2.1 Basic Types

The types TU1, TU2, and TU4 represent an unsigned one-, two-, or four-byte quantity, respectively. The types TS1, TS2, and TS4 represent a signed one-, two-, four-byte quantity, respectively.

2.2 Language Types

The language types like int, short or char are represented internally as follows:

<table>
<thead>
<tr>
<th>Note 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>This chapter describes the data types used by the JEFF format specification. All the values in a JEFF file are stored on one, two, four or eight contiguous bytes. In this document, the expression “null value” is synonym for a value of zero of the appropriate type.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note 3, 4, 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>The types TU1, TU2, and TU4 represent an unsigned one- and four-byte integer, respectively. The types TS1, TS2, and TS4 represent a signed one- and four-byte integer, respectively.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>The language types like int, short or char are represented internally as follows:</td>
</tr>
</tbody>
</table>
### Formats and Types

<table>
<thead>
<tr>
<th>Format Type</th>
<th>Language Type</th>
<th>Format</th>
<th>Min. Value</th>
<th>Max. Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JBYTE</td>
<td>byte</td>
<td>8-bit signed integer</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>JSHORT</td>
<td>short</td>
<td>16-bit signed integer</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>JINT</td>
<td>int</td>
<td>32-bit signed integer</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>JLONG</td>
<td>long</td>
<td>64-bit signed integer</td>
<td>-9.2233e+18</td>
<td>9.2233e+18</td>
</tr>
<tr>
<td>JFLOAT</td>
<td>float</td>
<td>32-bit IEEE 754</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JDOUBLE</td>
<td>double</td>
<td>64-bit IEEE 754</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JCHAR</td>
<td>char</td>
<td>16-bit Unicode char</td>
<td>0</td>
<td>Unicode max.</td>
</tr>
</tbody>
</table>

Note: The floating-point data are always stored in the file using the `JFLOAT` and `JDOUBLE` format corresponding to 32- and 64-bit IEEE 754 specification. The byte order used is the global byte order used for the whole file. If a specific processor does not use this order, the virtual machine is responsible for the data translation during the download or at runtime.
2.3 Strings

2.2.1 Definition
In this specification, a character is a 16-bit value. A string is an array of characters. Strings are encoded in the JEFF files as a VMString type (see below).

2.2.2 Comparison
In this document, comparisons of strings are based on the lexicographic order of the numerical values of their characters.

2.2.3 Representation

The character strings are stored in the following structure:

In the JEFF file, strings are stored according to the following structure:

```
VMConstUtf8 {
    TU2 nStringLength;
    TU1 nStringValue[];
}
```

```
VMString {
    TU2 nStringLength;
    TU1 nStringValue[nStringLength];
}
```

The items of the VMString structure are as follows:

**nStringLength**
The length of the encoded string, in bytes. This value may be different from the number of characters in the string.
The string value encoded with the Utf8 format as defined in the Virtual Machine Specification (see [1]).

This array of byte is an encoding of the value of the string. Each character of the string is encoded separately. The sequences of bytes corresponding to the encoding of each character are stored consecutively in the array following the order of the of the characters in the string.

The encoding used for each character depends on the range of the character value.

**Case 1**
A character C in the range 0x0000 to 0x007F is represented by a single byte X:

\[
X : \begin{array}{c}
7 & 6------------------0 \\
0 & \text{bits 6-0 of } C \\
\end{array}
\]

The 7 bits of data in the byte give the value of the character represented.

**Case 2**
A character C in the range 0x0080 to 0x07FF is represented by two consecutive bytes X and Y:

\[
X : \begin{array}{c}
7 & 6------------------0 \\
110 & \text{bits 10-6 of } C \\
\end{array} \\
Y : \begin{array}{c}
7 & 6------------------0 \\
10 & \text{bits 5-0 of } C \\
\end{array}
\]

The bytes represent the character with the value \((X \& 0x1F) \ll 6 \) + \((Y \& 0x3F)\).

**Case 3**
A character C in the range 0x0800 to 0xFFFF is represented by 3 consecutive bytes X, Y, and Z:

\[
X : \begin{array}{c}
7 & 6------------------0 \\
1110 & \text{bits 15-12 of } C \\
\end{array} \\
Y : \begin{array}{c}
7 & 6------------------0 \\
10 & \text{bits 11-6 of } C \\
\end{array} \\
Z : \begin{array}{c}
7 & 6------------------0 \\
10 & \text{bits 5-0 of } C \\
\end{array}
\]

The bytes represent the character with the value \(((X \& 0x0F) \ll 12) \) + \(((Y \& 0x3F) \ll 6) \) + \((Z \& 0x3F)\).
2.3 Specific Types

These types are used to store values with a specific meaning.

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMACCESS</td>
<td>Access Flag (see values below)</td>
<td>16-bit vector</td>
</tr>
<tr>
<td>VMTYPE</td>
<td>Type descriptor (see values below)</td>
<td>8-bit vector</td>
</tr>
<tr>
<td>VMNCELL</td>
<td>Number of virtual machine cells</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>VMOFFSET</td>
<td>Memory offset (in bytes)</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>VMDOFFSET</td>
<td>Memory offset (in bytes)</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>VMCINDEX</td>
<td>Class Index (see 3.1)</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>VMPINDEX</td>
<td>Package Index</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>VMFINDEX</td>
<td>Field Index (see 3.1)</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>VMMINDEX</td>
<td>Method Index (see 3.1)</td>
<td>32-bit unsigned integer</td>
</tr>
</tbody>
</table>

2.3.1 Access flags

The VMACCESS type describes the access privileges for classes, methods and fields. This type is conforming to the access flag type defined in the “Virtual Machine Specification” (see [1]). It's a bit vector with the following values:

The VMACCESS type describes the access privileges for classes, methods and fields. The VMACCESS type is a bit vector with the following values:
<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no subclasses allowed.</td>
</tr>
<tr>
<td>ACC_SUPER</td>
<td>0x0020</td>
<td>Treat superclass methods especially in invokespecial.</td>
</tr>
<tr>
<td>ACC_INTERFACE</td>
<td>0x0200</td>
<td>Is an interface.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Is abstract; may not be instantiated.</td>
</tr>
<tr>
<td><strong>Field</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
<td>Is private; usable only within the defined class.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Is protected; may be accessed within subclasses.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Is static.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no further overriding or assignment after initialization.</td>
</tr>
<tr>
<td>ACC_VOLATILE</td>
<td>0x0040</td>
<td>Is volatile; cannot be cached.</td>
</tr>
<tr>
<td>ACC_TRANSIENT</td>
<td>0x0080</td>
<td>Is transient; not written or read by a persistent object manager.</td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
<td>Is private; usable only within the defined class.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Is protected; may be accessed within subclasses.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Is static.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no overriding is allowed.</td>
</tr>
<tr>
<td>ACC_SYNCHRONIZED</td>
<td>0x0020</td>
<td>Is synchronized; wrap use in monitor lock.</td>
</tr>
<tr>
<td>ACC_NATIVE</td>
<td>0x0100</td>
<td>Is native; implemented in a language other than the source language.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Is abstract; no implementation is provided.</td>
</tr>
<tr>
<td>ACC_STRICT</td>
<td>0x0800</td>
<td>The VM is required to perform strict floating-point operations.</td>
</tr>
</tbody>
</table>
### Flag Name Value Meaning

#### Class

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside of its package.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no subclasses allowed.</td>
</tr>
<tr>
<td>ACC_SUPER</td>
<td>0x0020</td>
<td>Modify the behavior of the jeff_invokespecial bytecodes included in the bytecode area list of this class.</td>
</tr>
<tr>
<td>ACC_INTERFACE</td>
<td>0x0200</td>
<td>Is an interface.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Is abstract; may not be instantiated.</td>
</tr>
</tbody>
</table>

#### Field

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside of its package.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
<td>Is private; usable only within the defined class.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Is protected; may be accessed within subclasses.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Is static.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no further overriding or assignment after initialization.</td>
</tr>
<tr>
<td>ACC_VOLATILE</td>
<td>0x0040</td>
<td>Is volatile; cannot be cached.</td>
</tr>
<tr>
<td>ACC_TRANSIENT</td>
<td>0x0080</td>
<td>Is transient; not written or read by a persistent object manager.</td>
</tr>
</tbody>
</table>

#### Method

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside of its package.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
<td>Is private; usable only within the defined class.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Is protected; may be accessed within subclasses.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Is static.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no overriding is allowed.</td>
</tr>
<tr>
<td>ACC_SYNCHRONIZED</td>
<td>0x0020</td>
<td>Is synchronized; wrap use in monitor lock.</td>
</tr>
<tr>
<td>ACC_NATIVE</td>
<td>0x0100</td>
<td>Is native; implemented in a language other than the source language.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Is abstract; no implementation is provided.</td>
</tr>
<tr>
<td>ACC_STRICT</td>
<td>0x0800</td>
<td>The VM is required to perform strict floating-point operations.</td>
</tr>
</tbody>
</table>

#### 2.3.2 Type Descriptor

A type descriptor is composed of a type value (a VMTYPE), an optional array dimension value (a TU1) and an optional class index (a VMCINDEX).

\[\text{nsi p9 note 3}\]

The presence or the absence of the optional elements of a type descriptor is explicitly specified everywhere a type descriptor is used in the specification.
## Type Value

<table>
<thead>
<tr>
<th>VMTYPE</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_TYPE_VOID</td>
<td>0x00</td>
<td>Used for the return type of a method</td>
</tr>
<tr>
<td>VM_TYPE_SHORT</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_INT</td>
<td>0x02</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_LONG</td>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_BYTE</td>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_CHAR</td>
<td>0x05</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_FLOAT</td>
<td>0x06</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_DOUBLE</td>
<td>0x07</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_BOOLEAN</td>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>VM_TYPE_OBJECT</td>
<td>0x0A</td>
<td></td>
</tr>
</tbody>
</table>

These values can be interpreted as a bit field as follows:

<table>
<thead>
<tr>
<th>7---4 3--2 1--0</th>
<th>YY</th>
<th>XX</th>
<th>YY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Where:

- YY is the type size in bytes. The actual size is: 1 << YY.
- XX serves to differentiate types having the same size.

The following flags are also set:

| VM_TYPE_TWO_CELL | 0x10 | for a type using two virtual machine cells (this flag is not set for an array) |
| VM_TYPE_REF      | 0x20 | for an object or an array                                                    |
| VM_TYPE_MONO     | 0x40 | for a mono-dimensional array                                                 |

The following flags may be set:

| VM_TYPE_MULTI   | 0x80 | for a n-dimensional array, where n >= 2                                    |
| VM_TYPE_MULTI   | 0x80 | for an n-dimensional array, where n >= 2                                    |
### Dimension Value

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8, 9, 12</td>
<td>The dimension value gives the number of dimensions (0-255) of an array type. This value is optional for a non-array type or for a mono-dimensional array. For a multi-dimensional array, the <code>VM_TYPE_MULTI</code> flag is set in the type value and the dimension value is mandatory to know the exact array type. The dimension value gives the number of dimensions (0-255) of an array type. This value is optional for non-array and mono-dimensional array types. This value is not present for a void return type. For a multi-dimensional array, the <code>VM_TYPE_MULTI</code> flag is set in the type value and the dimension value must be present.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>The dimension values are as follows: 0 for a non-array type, 1 for a simple array (ex: int a[2]), 2 for a 2 dimensional array (ex: long array[2][8]), ... 255 for a 255 dimensional array.</td>
</tr>
<tr>
<td>10, 11</td>
<td>The dimension values are as follows: 0 for a non-array type, 1 for a simple array (e.g. int a[2]), 2 for a 2 dimensional array (e.g. long array[2][8]), ... 255 for a 255 dimensional array.</td>
</tr>
</tbody>
</table>

### Class Index

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The optional class index gives the exact type of descriptor of a class or of an array of class. For a scalar type or an array of scalar types, the class index is useless. The optional class index gives the exact type of descriptor of a class or of an array of a class. For a scalar type or an array of scalar types, the class index must not be present.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>The optional class index gives the exact type of descriptor of a class or of an array of class. For a scalar type or an array of scalar types, the class index must not be present.</td>
</tr>
</tbody>
</table>
Summary

Here is a list of the possible code:

<table>
<thead>
<tr>
<th>Type</th>
<th>Type value</th>
<th>Dimension</th>
<th>Class Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>0x00</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>short</td>
<td>0x01</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>int</td>
<td>0x02</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>long</td>
<td>0x13</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>byte</td>
<td>0x04</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>char</td>
<td>0x05</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>float</td>
<td>0x06</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>double</td>
<td>0x17</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>boolean</td>
<td>0x08</td>
<td>0 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>reference</td>
<td>0x0A</td>
<td>0 or absent</td>
<td>index of the class</td>
</tr>
<tr>
<td>short[]</td>
<td>0x61</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>int[]</td>
<td>0x62</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>long[]</td>
<td>0x63</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>byte[]</td>
<td>0x64</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>char[]</td>
<td>0x65</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>float[]</td>
<td>0x66</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>double[]</td>
<td>0x67</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>boolean[]</td>
<td>0x68</td>
<td>1 or absent</td>
<td>absent</td>
</tr>
<tr>
<td>reference[]</td>
<td>0x6A</td>
<td>1 or absent</td>
<td>index of the class</td>
</tr>
<tr>
<td>short[][]</td>
<td>0x81</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>int[][]</td>
<td>0x82</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>long[][]</td>
<td>0x83</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>byte[][]</td>
<td>0x84</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>char[][]</td>
<td>0x85</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>float[][]</td>
<td>0x86</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>double[][]</td>
<td>0x87</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>boolean[][]</td>
<td>0x88</td>
<td>dimension</td>
<td>absent</td>
</tr>
<tr>
<td>reference[][]</td>
<td>0x8A</td>
<td>dimension</td>
<td>index of the class</td>
</tr>
</tbody>
</table>

Examples


\[\text{The examples are not normative. They are just an illustration of the above explanations.}\]
<table>
<thead>
<tr>
<th>Offset Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOFFSET</td>
<td>An unsigned 16-bit value. This value is an offset in bytes from the beginning of a class file header. Depending on where the offset value is located, the corresponding class file header is unambiguous.</td>
</tr>
<tr>
<td>VMDOFFSET</td>
<td>An unsigned 16-bit value located in a class area section (See 3.3.2). This value is an offset in bytes from the beginning of the class header of the class area section.</td>
</tr>
</tbody>
</table>

### 2.3.3 Offsets

There are two types of offset values used in the specification: VMOFFSET and VMDOFFSET.
A **VMDOFFSET** is an unsigned 32-bit value. This value is an offset in bytes from the beginning of the file header.

### 2.3.4 Index Values
See the File Structure.
3 File Structure

This chapter gives the complete structure of the JEFF file format.

3.1 Definitions

This part describes the definitions and rules used in the specification.
### 3.1.1 Fully Qualified Names

Fully qualified name have the following definition:

- The fully qualified name of a named package that is not a sub-package of a named package is its simple name.
- The fully qualified name of a named package that is a sub-package of another named package consists of the fully qualified name of the containing package followed by the character "/" (Unicode 0x002F) followed by the simple (member) name of the sub-package.
- The fully qualified name of a class or interface that is declared in an unnamed package is the simple name of the class or interface.
- The fully qualified name of a class or interface that is declared in a named package consists of the fully qualified name of the package followed by the character "/" (Unicode 0x002F) followed by the simple name of the class or interface.

### 3.1.2 Symbolic Names

The file specification refers to symbolic names for the classes, the packages, the fields and the methods. They are defined as follow:

- The file specification refers to symbolic names for classes, packages, fields and methods. They are defined as follow:
### Class Symbolic Name
A class symbolic name is the fully qualified name of the class (package and class names, e.g. "java/lang/String"). If a class has no package, the class symbolic name is the class name.

### Package Symbolic Name
A package symbolic name is the fully qualified name of the package (e.g. "java/lang").

### Field Symbolic Name
A field symbolic name is the concatenation of the field name, a space character (Unicode 0x0020) and the field descriptor string.

*E.g. for the field `double m_Field[]`, the symbolic name is "m_Field [D]."

### Method Symbolic Name
A method symbolic name is the concatenation of the method name, a space character (Unicode 0x0020) and the method descriptor string.

*E.g. for the method `void append(String)`, the symbolic name is "append (Ljava/lang/String;)V".*
### 3.1.3 Internal Classes and External Classes

A JEFF file contains the definition of one or several classes. For a given file, the classes stored in the file are called “internal classes”. The classes referenced by the internal classes but not included in the same file are called “external classes”.

The packages of the internal and external classes are ordered following the crescent lexicographic order of their fully qualified names. This order defines an index value for each package (a **VMPINDEX** value). The package index range is 0 to **number of packages – 1**. If an internal or an external class has no package, this class is defined in the “default package”, a package with no name. In this case the “default package” must be counted in the **number of packages** and its index is always 0.

The internal classes and the external classes are ordered and identified by an index (a **VMCINDEX** value). The class index range is:

- **0** to **InternalClassCount – 1** for the internal classes
- **InternalClassCount** to **TotalClassCount – 1** for the external classes

The class index values follow the crescent lexicographic order of the classes fully qualified names (separately for the internal classes and for the external classes)

The package index and the class index assignments are local to the file.
### 3.1.4 Fields and Methods

<table>
<thead>
<tr>
<th><strong>Field Symbolic Name</strong></th>
<th>A field symbolic name is the concatenation of the field name, a character 0x0020 and the field descriptor string.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Method Symbolic Name</strong></th>
<th>A method symbolic name is the concatenation of the method name, a character 0x0020 and the method descriptor string.</th>
</tr>
</thead>
</table>

#### Algorithm

The field indexes are computed as follows:

1. The symbolic names of the internal class fields are indexed according to their crescent lexicographic order, with index increment of 1, indexes ranging from zero up to n-1.
2. The symbolic names of the external class fields that are not also symbolic names of internal class fields are indexed according to their crescent lexicographic order, with index increment of 1, starting at n.

Each entry in the table is identified by a zero-based index (a VMFINDEX value).

By definition of the field symbolic name and the construction of the table, the following properties are deducted:

- Two different field indexes identify two different symbolic names.
- Two different fields, internal or external, share the same index if and only if they have the same name and the same descriptor.

The same construction is used to define the method indexes (VMMINDEX).

By definition of the method symbolic name and the construction of the table, the following properties are deducted:

- Two different method indexes identify two different symbolic names.
- Two different methods, internal or external, share the same index if and only if they have the same name and the same descriptor.

The field index and the method index assignments are local to the file.

### 3.1.5 Field Position

JEFF includes some information about the position of the field in memory. These precomputed values are useful to speedup the download of classes and to have a quick access to the fields at runtime.
The computation must take into account the following constraints:

- Class fields and instance fields are stored in separate memory spaces.
- The field data must be aligned in memory according to their sizes.
- Most of the virtual machines store the field values contiguously for each class.

When a class A inherits from a class B, the way the instance fields of an instance of A are stored depends on the virtual machine. Some virtual machines store the fields of A first and then the fields of B, others use the opposite order and other stores them in non-contiguous memory areas.

The binary compatibility requirement (see Overview) implies that the values computed for a class are independent of the values computed for its super classes, whether or not they are included in the same file.

The consequences of these constraints are the following:

- The pre-computed values are redundant with the field information. They are only included to speedup the virtual machine.
- Some virtual machines may not use these values.
- The values are computed independently for each class.

The same construction process is applied separately for the class fields and the instance fields. The super class fields and the sub-class fields are not taken into account.

The fields are classed in an ordered list. The order used follows the size of each field. The longer fields are stored first (type long or double), the smaller fields are stored at the end of the list (type byte). The order used between fields of the same size is undefined. This ordering allows keeping the alignment between the data.

The fields are ordered in a list. The order used follows the size of each field. The longer fields are stored first (type long or double), the smaller fields are stored at the end of the list (type byte). The order used between fields of the same size is undefined. This ordering allows keeping the alignment between the data.

- The position of a given field is the position of the preceding field in the list plus the size of the preceding field. The first field position is zero.
- The total size of the field area is the sum of the size of each field in the list.

### 3.2 Conventions

The following conventions are use in this chapter.
3.2.1 Notations

The format is presented using pseudo-structures written in a C-like structure notation. Like the fields of a C structure, successive items are stored sequentially, with padding and alignment.

This document contains notations to represent lists and arrays of elements. An array or a list is the representation of a set of several consecutive structures. In an array, the structures are identical with a fix size and there are no padding bytes between them. In a list, the structures may be of variable length and some padding bytes may be added between them. When a list is used, the comments precise the length of each structure and the presence of padding bytes.

3.2.2 Byte Order

All the values are stored using the byte order defined by a set of flags specified in the file header. Floating-point numbers and integer values are treated separately.

All the values are stored using the byte order defined by a set of flags specified in the file header. Floating-point numbers and integer values are treated differently.
### 3.2.3 Alignment and Padding

If a platform requires the alignment of the multi-byte values in memory, JEFF allows an efficient access to all its data without byte-by-byte reading.

When a JEFF file is stored on the platform, the first byte of the file header must always be aligned in memory on a 8-byte boundary.

If a platform requires the alignment of the multi-byte values in memory, JEFF allows an efficient access to all its data without requiring byte-by-byte reading.

When a JEFF file is stored on the platform, the first byte of the file header must always be aligned in memory on an 8-byte boundary.

All the items constituting the file are aligned in memory. The following table gives the memory alignment:

<table>
<thead>
<tr>
<th>Elements</th>
<th>Element size, in bytes</th>
<th>Alignment on memory boundaries of</th>
</tr>
</thead>
<tbody>
<tr>
<td>TU1, TS1, JBYTE, VMTYPE</td>
<td>1</td>
<td>1 byte</td>
</tr>
<tr>
<td>TU2, TS2, JSHORT, JCHAR, VMACCESS, VMNCELL, VMOFFSET, VMCINDEX</td>
<td>2</td>
<td>2 bytes</td>
</tr>
<tr>
<td>TU4, TS4, JINT, JFLOAT, VMOFFSET, VMMINDEX, VMFINDEX</td>
<td>4</td>
<td>4 bytes</td>
</tr>
<tr>
<td>JLONG, JDOUBLE</td>
<td>8</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

When aligning data, some extra bytes may be needed for padding. These bytes must be set to null.

Structures are always aligned following the alignment of their first element.

Example:

```java
VMStructure {
    VMOFFSET ofAnOffset;
    TU1 <0-2 byte pad>
    TU4 nAnyValue;
}
```
The structure is aligned on a 2-byte boundary because `VMOFFSET` is a 2-byte type. The field `nAnyValue` is aligned on a 4-byte boundary. A padding of 2 bytes may be inserted between `ofAnOffset` and `nAnyValue`.

## 3.3 The File Structure

### 3.3 Definition of the File Structures

All the structures defined in this specification are stored in the JEFF file one after the other without overlapping and without any intermediate data other than padding bytes required for alignment. Every unspecified data may be stored in an optional attribute as defined in the Attribute Section.

The file structure is composed of six ordered sections.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Header</td>
<td>File identification and directory</td>
</tr>
<tr>
<td>Class Section</td>
<td>List of class areas</td>
</tr>
<tr>
<td>Optional Attributes</td>
<td>List of the optional attributes</td>
</tr>
<tr>
<td>Section</td>
<td></td>
</tr>
<tr>
<td>Symbolic Data Section</td>
<td>The symbolic information used by the classes</td>
</tr>
<tr>
<td>Constant Data Pool</td>
<td>Set of common constant data</td>
</tr>
<tr>
<td>Digital Signature</td>
<td>Signature of the complete file</td>
</tr>
</tbody>
</table>

The file structure is composed of six sections ordered as follows:

- File Header
- Class Section
- Optional Attributes Section
- Symbolic Data Section
- Constant Data Pool
- Digital Signature

### File Header

The file header contains the information used to identify the file and a directory to access to the other sections content.

The file header contains the information used to identify the file and a directory to access to the other sections' contents.
Class Section

| Note 2 | The class section describes the content of each class (inheritance, fields, methods and code). |
| Note 3 | The class section describes the content and the properties of each class. |

Optional Attributes Section

| Attributes Section | This optional section contains the optional attributes for the file, the classes, the methods and the fields. |

Symbolic Data Section

| Note 3 | In this area are stored all the symbolic information used to identify the classes, the methods and the fields. |
| Note 3 | This section contains the symbolic information used to identify the classes, the methods and the fields. |

Constant Data Pool

The constant strings and the descriptors used by the Optional Attribute Section and the Symbolic Data Section are stored in this structure.

Digital Signature

This part contains the digital signature of the complete file.

3.3.1 File Header

The file header is always located at the beginning of the file. In the file structure, some sections have a variable length. The file header contains a directory providing a quick access to these sections.
VMFileHeader {
    TU1     nMagicWord1;
    TU1     nMagicWord2;
    TU1     nMagicWord3;
    TU1     nMagicWord4;
    TU1     nFormatVersionMajor;
    TU1     nFormatVersionMinor;
    TU1     nByteOrder;
    TU1     nOptions;
    TU4     nFileLength;
    TU2     nFileVersion;
    TU2     nTotalPackageCount;
    TU2     nInternalClassCount;
    TU2     nTotalClassCount;
    TU4     nTotalFieldCount;
    TU4     nTotalMethodCount;
    VMDOFFSET dofAttributeSection;
    VMDOFFSET dofSymbolicData;
    VMDOFFSET dofConstantDataPool;
    VMDOFFSET dofFileSignature;
    VMDOFFSET dofClassHeader[nInternalClassCount];
}

The items of the VMFileHeader structure are as follows:

nMagicWord1, nMagicWord2, nMagicWord3, nMagicWord4

<table>
<thead>
<tr>
<th>INSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>nMagicWord1 = 0x4A, nMagicWord2 = 0x45, nMagicWord3 = 0x46 and nMagicWord4 = 0x46 (&quot;JEFF&quot; in Ascii).</td>
</tr>
</tbody>
</table>

nFormatVersionMajor, nFormatVersionMinor,

Version number of the file format. For this version (1.0), the values are nFormatVersionMajor = 0x01 for the major version number and nFormatVersionMinor = 0x00 for the minor version number.

nByteOrder

This 8-bit vector gives the byte order used by all the values stored in the file, except the magic number. The following set of flags gives the byte order of integer values and the floating-point values separately. In the definitions, the term "integer value" designs all the two-, four- and eight-bytes long values, except the JFLOAT and JDOUBLE values.

| VM_ORDER_INT_BIG  | 0x01 | If this flag is set, integer values are stored using the big-endian convention. Otherwise, they are stored using the little-endian convention. |
| VM_ORDER_INT_64_INV | 0x02 | If this flag is set, the two 32-bit parts of the 64-bit integer values are inverted. |
| VM_ORDER_FLOAT_BIG | 0x04 | If this flag is set, JFLOAT and JDOUBLE values are stored using the big-endian convention. Otherwise, they are stored using the little-endian convention. |
are stored using the big-endian convention. Otherwise, they are stored using the little-endian convention.

**VM_ORDER_FLOAT_64_INV** 0x08 If this flag is set, the two 32-bit parts of the **JDOUBLE** values are inverted.

### nOptions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(\text{NSI} / 15 )ote 1</strong></td>
<td>A set of information on the content of the internal classes.</td>
</tr>
<tr>
<td></td>
<td>A set of information describing some properties of the internal classes.</td>
</tr>
</tbody>
</table>

This item is an 8-bit vector with the following flag values:
<table>
<thead>
<tr>
<th>VM_USE_LONG_TYPE</th>
<th>0x01</th>
<th>One of the classes uses the &quot;long&quot; type (in the fields types, the methods signatures, the constant values or the bytecode instructions).</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_USE_UNICODE</td>
<td>0x02</td>
<td>This file contains non-ASCII characters (Unicode).</td>
</tr>
<tr>
<td>VM_USE_FLOAT_TYPE</td>
<td>0x04</td>
<td>One of the classes uses the &quot;float&quot; type and/or the &quot;double&quot; type (in the fields types, the methods signatures, the constant values or the bytecode instructions).</td>
</tr>
<tr>
<td>VM_USE STRICT_FLOAT</td>
<td>0x08</td>
<td>One of the classes contains bytecodes with strict floating-point computation (the &quot;strictfp&quot; keyword is used in the source file).</td>
</tr>
<tr>
<td>VM_USE_NATIVE_METHOD</td>
<td>0x10</td>
<td>One of the classes contains native methods.</td>
</tr>
<tr>
<td>VM_USE_FINALIZER</td>
<td>0x20</td>
<td>One of the classes has an instance finalizer or a class finalizer.</td>
</tr>
<tr>
<td>VM_USE_MONITOR</td>
<td>0x40</td>
<td>One of the classes uses the flag ACC_SYNCHRONIZED or the bytecodes monitorenter or monitorexit in one of its methods.</td>
</tr>
</tbody>
</table>

nFileLength
Size in bytes of the file (all elements included).

nFileVersion
Version number of the file itself. The most significant byte carries the major version number. The less significant byte carries the minor version number. This specification does not define the interpretation of this field by a virtual machine.

nTotalPackageCount
The total number of unique packages referenced in the file (for the internal classes and the external classes).
nInternalClassCount
The number of classes in the file (internal classes).

nTotalClassCount
The total number of the classes referenced in the file (internal classes and external classes).

nTotalFieldCount
The total number of field symbolic names used in the file.

nTotalMethodCount
The total number of method symbolic names used in the file.

dofAttributeSection
Offset of the Optional Attribute Section, a VMAttributeSection structure. This field is set to null if no optional attributes are stored in the file.

dofSymbolicData
Offset of the symbolic data section, a VMSymbolicDataSection structure.

dofConstantDataPool
Offset of the constant data pool, a VMConstantDataPool structure.

dofFileSignature
Offset of the file signature defined in a VMFileSignature structure. This value is set to null if the file is not signed.

dofClassHeader
Offsets of the VMClassHeader structures for all internal classes. The entries of this table follow the class index order and the class areas are stored in the same order.

3.3.2 Class Area
For each class included in the file, a class area contains the information specific to the class. Within the class area, the references to other elements are given by 16-bit unsigned offsets (VMOFFSET) relative to the beginning of the class header.

3.3.2 Class Section
For each class included in the file, a class area contains the information specific to the class. The Class Section contains these class areas stored consecutively in an ordered list following the crescent order of the corresponding class indexes.

The first element of this area is the class header pointed to from the dofClassHeader array in the file header. The other structures in the class area are stored one after the other without overlapping and without any intermediate data other than padding bytes required for alignment.
The ten sections of the class area must be ordered as follows:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Header</td>
<td>Class identification and directory</td>
</tr>
<tr>
<td>Interface Table</td>
<td>List of the interfaces implemented by the current class</td>
</tr>
<tr>
<td>Referenced Class Table</td>
<td>List of the classes referenced by the current class</td>
</tr>
<tr>
<td>Internal Field Table</td>
<td>List of the fields of the current class</td>
</tr>
<tr>
<td>Internal Method Table</td>
<td>List of the methods of the current class</td>
</tr>
<tr>
<td>Referenced Field Table</td>
<td>List of the fields of other classes used by the current class</td>
</tr>
<tr>
<td>Referenced Method Table</td>
<td>List of the methods of other classes used by the current class</td>
</tr>
<tr>
<td>Bytecode Area List</td>
<td>List of the bytecode areas for the methods of the current class</td>
</tr>
<tr>
<td>Exception Table List</td>
<td>List of the exception handler tables for the methods of the current class</td>
</tr>
<tr>
<td>Constant Data Section</td>
<td>Set of constant data used by the current class</td>
</tr>
</tbody>
</table>

### 3.3.2.1 Class Header

The class header is always located at the beginning of the class representation. In the class file structure, some sections have a variable length. The directory is used as a redirector to have a quick access to these sections.
VMClassHeader {
    VMOFFSET  ofThisClassIndex;
    VMPIINDEX pidPackage;
    VMACCESS  aAccessFlag;
    TU2       nClassData;
    VMOFFSET  ofClassConstructor;
    VMOFFSET  ofInterfaceTable;
    VMOFFSET  ofFieldTable;
    VMOFFSET  ofMethodTable;
    VMOFFSET  ofReferencedFieldTable;
    VMOFFSET  ofReferencedMethodTable;
    VMOFFSET  ofReferencedClassTable;
    VMOFFSET  ofConstantDataSection;
    VMOFFSET  ofSuperClassIndex;
    TU2       nInstanceData;
    VMOFFSET  ofInstanceConstructor;
}
For the classes, the class area has the following structure:

```java
VMClassHeader {
    VMOFFSET ofThisClassIndex;
    VMPINDEX pidPackage;
    VMACCESS aAccessFlag;
    TU2 nClassData;
    VMOFFSET ofClassConstructor;
    VMOFFSET ofInterfaceTable;
    VMOFFSET ofFieldTable;
    VMOFFSET ofMethodTable;
    VMOFFSET ofReferencedFieldTable;
    VMOFFSET ofReferencedMethodTable;
    VMOFFSET ofReferencedClassTable;
    VMOFFSET ofConstantDataSection;
    VMOFFSET ofSuperClassIndex;
    TU2 nInstanceData;
    VMOFFSET ofInstanceConstructor;
}
```

For the interfaces, the class area has the following structure:

```java
VMClassHeader {
    VMOFFSET ofThisClassIndex;
    VMPINDEX pidPackage;
    VMACCESS aAccessFlag;
    TU2 nClassData;
    VMOFFSET ofClassConstructor;
    VMOFFSET ofInterfaceTable;
    VMOFFSET ofFieldTable;
    VMOFFSET ofMethodTable;
    VMOFFSET ofReferencedFieldTable;
    VMOFFSET ofReferencedMethodTable;
    VMOFFSET ofReferencedClassTable;
    VMOFFSET ofConstantDataSection;
}
```

The items of the `VMClassHeader` structure are as follows:

- **ofThisClassIndex**
  Offset of the current class index, a `VMCINDEX` value stored in the “referenced class table” of the current class.

- **pidPackage**
  The current class package index.

- **aAccessFlag**
  Class access flags. The possible values are:

<table>
<thead>
<tr>
<th>Class Access Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>Is public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>Is final; no subclasses allowed.</td>
</tr>
</tbody>
</table>
ACC_SUPER  Treat superclass methods specially in invokespecial.
ACC_INTERFACE  Is an interface.
ACC_ABSTRACT  Is abstract; may not be instantiated.

nClassData
This value is the total size, in bytes, of the class fields. The algorithm used to compute the value is given in 3.1.5 Field Position. The size is null if there is no class field in the class.

OfClassConstructor
Offset of the class constructor "<clinit>". Offset of the corresponding VMMethodInfo structure. Null if there is no class constructor.

ofInterfaceTable
Offset of the interface table, a VMInterfaceTable structure. This value is null if the current class implements no interfaces.

ofFieldTable
Offset of the internal field table, a VMFieldInfoTable structure. This value is null if the current class has no field.

ofMethodTable
Offset of the internal method table, a VMMethodInfoTable structure. This value is null if the current class has no method.

ofReferencedFieldTable
Offset of the referenced field table, a VMReferencedFieldTable structure. This value is null if the bytecode uses no field.

ofReferencedMethodTable
Offset of the referenced method table, a VMReferencedMethodTable structure. This value is null if the bytecode uses no method.

ofReferencedClassTable
Offset of the referenced class table, a VMReferencedClassTable structure.

ofConstantDataSection
Offset of the constant data section, a VMConstantDataSection structure. This value is null if the class does not contain any constants.

ofSuperClassIndex
Offset of the super class index, a VMCINDEX value stored in the “referenced class table” of the current class. If the current class is java.lang.Object, the offset value is zero. This value is not present for an interface.

nInstanceData
This value is the total size, in bytes, of the instance fields. The algorithm used to compute the value is given in 3.1.5 Field Position. The size is null if there is no instance field in the class. This value is not present for an interface.

ofInstanceConstructor
Offset of the default instance constructor "<init>()V". Offset of the corresponding VMMethodInfo structure. The value is null if there is no default instance constructor. This value is not present for an interface.
3.3.2.2 Interface Table

This structure is the list of the interfaces implemented by this class or interface.

```
VMInterfaceTable {
    TU2 nInterfaceCount;
    VMOFFSET ofInterfaceIndex [nInterfaceCount];
}
```

The items of the `VMInterfaceTable` structure are as follows:

- **nInterfaceCount**
The number of interfaces implemented.

- **ofInterfaceIndex**
Offset of a class index, a `VMCINDEX` value stored in the “referenced class table” of the current class. The corresponding class is a super interface implemented by the current class or interface.

3.3.2.3 Referenced Class Table

Every class, internal or external, referenced by the current class is represented in the following table:

```
VMReferencedClassTable {
    TU2 nReferencedClassCount;
    VMCINDEX cidReferencedClass [nReferencedClassCount];
}
```

The current class is also represented in this table.

The items of the `VMReferencedClassTable` structure are as follows:

- **nReferencedClassCount**
The number of referenced classes.

- **cidReferencedClass**
The class index (`VMCINDEX` value) of a class referenced by the current class.

3.3.2.4 Internal Field Table

Every field member of the defined class is described by a field information structure located in a table:

```
VMFieldInfoTable {
    TU2 nFieldCount;
    TU1 <0-2 byte pad>
    {
        VMINDEX fidFieldIndex;
        VMOFFSET ofThisClassIndex;
        VMTYPE tFieldType;
        TU1 nTypeDimension;
        VMACCESS aAccessFlag;
        TU2 nFieldDataOffset;
    } VMFieldInfo [nFieldCount];
}
```
The instance fields are always stored first in the table. The class fields follow them. Instance fields and class fields are stored following the crescent order of their index. The items of the VMFieldInfoTable structure are as follows:

- **nFieldCount**
  The number of fields in the class.

- **fidFieldIndex**
  The field index.

- **ofThisClassIndex**
  Offset of the current class index, a VMCINDEX value stored in the “referenced class table” of the current class.

- **tFieldType**
  The field type. By definition, the field type gives the size of the value stored by the field.

- **nTypeDimension**
  The array dimension associated with the type. This value is always present.

- **aAccessFlag**
  Field access flag. The possible values are:

  - `ACC_PUBLIC` Is public; may be accessed from outside its package.
  - `ACC_PRIVATE` Is private; usable only within the defined class.
  - `ACC_PROTECTED` Is protected; may be accessed within subclasses.
  - `ACC_STATIC` Is static.
  - `ACC_FINAL` Is final; no further overriding or assignment after initialization.
  - `ACC_VOLATILE` Is volatile; cannot be cached.
  - `ACC_TRANSIENT` Is transient; not written or read by a persistent object manager.

- **nFieldDataOffset**
  This value is an offset, in bytes, of the field data in the class field value area or in the instance value area. The algorithm used to compute the value is given in 3.1.5 Field Position. The total size of the instance field data area is given by `nInstanceData`. The total size of the class field data area is given by `nClassData`.

### 3.3.2.5 Internal Method Table

Every method of the defined class, including the special internal methods, `<init>` or `<clinit>`, is described by a method information structure located in a table:

```c
VMMethodInfoTable {
    TU2 nMethodCount;
    TU1 <0-2 byte pad>
    {
        VMMINDEX midMethodIndex;
        VMOFFSET ofThisClassIndex;
        VMCELL ncStackArgument;
        VMACCESS aAccessFlag;
        VMOFFSET ofCode;
    } VMMethodInfo [nMethodCount];
    TU4 nNativeReference[];
}
```
The instance methods are always stored first in the table. The class methods follow them. Instance methods and class methods are stored following the crescent order of their index. The items of the **VMMethodInfoTable** structure are as follows:

<table>
<thead>
<tr>
<th><strong>nMethodCount</strong></th>
<th>The number of method in the class.</th>
</tr>
</thead>
</table>

The number of methods in the class.

<table>
<thead>
<tr>
<th><strong>midMethodIndex</strong></th>
<th>The method index.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>ofThisClassIndex</strong></th>
<th>Offset of the current class index, a <strong>VMINDEX</strong> value stored in the “referenced class table” of the current class.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>ncStackArgument</strong></th>
<th>Size of the method arguments in the stack. The size includes the reference to the instance used for calling an instance method. This size does not include the return value of the method. The bytecode interpreter uses <strong>ncStackArgument</strong> to clean the stack after the method return. The size, in cells, is computed during the class translation.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>aAccessFlag</strong></th>
<th>Method access flag. The possible values are:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>Is public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>Is private; usable only within the defined class.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>Is protected; may be accessed within subclasses.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>Is static.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>Is final; no overriding is allowed.</td>
</tr>
<tr>
<td>ACC_SYNCHRONIZED</td>
<td>Is synchronized; wrap use in monitor lock.</td>
</tr>
<tr>
<td>ACC_NATIVE</td>
<td>Is native; implemented in a language other than the source language.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>Is abstract; no implementation is provided.</td>
</tr>
<tr>
<td>ACC_STRICT</td>
<td>The VM is required to perform strict floating-point operations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ofCode</strong></th>
<th>For a non-native non-abstract method, this value is the offset of the bytecode block, a <strong>VMBytecodeBlock</strong> structure. For an abstract method, the offset value is null. For a native method, the value is the offset of one of the <strong>nNativeReference</strong> values. Each native method must refer to a separate <strong>nNativeReference</strong> value.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ofCode</strong></td>
<td>For a non-native non-abstract method, this value is the offset of the bytecode block, a <strong>VMBytecodeBlock</strong> structure. For an abstract method, the offset value is null. For a native method, the value is the offset of one of the <strong>nNativeReference</strong> values. Each native method must have a different <strong>ofCode</strong> value.</td>
</tr>
</tbody>
</table>
This array of undefined **TU4** values must contain as many elements as the class has native methods. These values are reserved for future use.

This array of **TU4** values contains as many elements as the class has native methods. To each **TU4** value corresponds one and only one native method of the class. The **TU4** values are stored following the order of storage of the corresponding **VMMethodInfo** structure. The **TU4** values are not specified and reserved for future use.

### 3.3.2.6 Referenced Field Table

The referenced field table describes the internal or external class fields that are not members of the current class but are used by this class. If an instruction refers to such a field, the bytecode gives the offset of the corresponding **VMReferencedField** structure.

```java
VMReferencedFieldTable {
    TU2 nFieldCount;
    TU1 <0-2 byte pad>
    {
        VMFINDEX fidFieldIndex;
        VMOFFSET ofClassIndex;
        VMTYPE tFieldType;
        TU1 nTypeDimension;
    } VMReferencedField [nFieldCount];
}
```

The items of the **VMReferencedFieldTable** structure are as follows:

- **nFieldCount**
  The number of fields in the table.

- **fidFieldIndex**
  The field index.

- **ofClassIndex**
  Offset of a class index, a **VMCINDEX** value stored in the "referenced class table" of the current class. This index identifies the class containing the field.

- **tFieldType**
  The field type. By definition, the field type gives the size of the value stored by the field. This information is used to retrieve in the operand stack the reference of the object instance (for an instance field).

- **nTypeDimension**
  The array dimension associated with the type. This value is always present.

### 3.3.2.7 Referenced Method Table

The referenced method table describes the internal or external class methods that are not members of the current class but are used by this class. If an instruction refers to such a method, the bytecode gives the offset of the corresponding **VMReferencedMethod** structure.
VMReferencedMethodTable {
    TU2 nMethodCount;
    TU1 <0-2 byte pad>
    {
        VMMINDEX midMethodIndex;
        VMOFFSET ofClassIndex;
        VMNCELL ncStackArgument;
    } VMReferencedMethod [nMethodCount];
}

The items of the VMReferencedMethodTable structure are as follows:

nMethodCount
The number of methods in the table.

midMethodIndex
The method index.

ofClassIndex
Offset of a class index, a VMCINDEX value stored in the “referenced class table” of the current class. This index identifies the class containing the method.

ncStackArgument
Size of the method arguments in the stack. The size includes the reference to the instance used for calling an instance method. This size does not include the return value of the method. The bytecode interpreter uses ncStackArgument to clean the stack after the method return. The size, in cells, is computed during the class translation.

3.3.2.8 Bytecode Block Structure
This part is a block of bytecode corresponding to the method body:

3.3.2.8 Bytecode Block Structure
This section is a list of consecutive bytecode block structures. To each bytecode block structure corresponds one and only one non-native, non-abstract method of the internal method table of this class area. The bytecode block structures are stored following the order of storage of the corresponding methods in the internal method table.

Each bytecode block is represented by the following structure:

VMBytecodeBlock {
    VMNCELL ncMaxStack;
    VMNCELL ncMaxLocals;
    VMOFFSET ofExceptionCatchTable;
    TU2 nByteCodeSize;
    TU1 bytecode[nByteCodeSize];
}

The items of the VMBytecodeBlock structure are as follows:

ncMaxStack
### ncMaxStack

The value of the **ncMaxStack** item gives the maximum number of cells on the operand stack at any point during execution of this method.

### ncMaxLocals

The value of the **ncMaxLocals** item gives the number of local variables used by this method, including the arguments passed to the method on invocation. The index of the first local variable is 0. The greatest local variable index for a one-word value is **ncMaxLocals-1**. The greatest local variable index for a two-word value is **ncMaxLocals-2**.

### ofExceptionCatchTable

Offset of the caught exception table, a **VMExceptionCatchTable** structure. Null if no exception is caught in this method.

### nByteCodeSize

The size of the bytecode block in bytes. The value of **nByteCodeSize** must be greater than zero; the code array must not be empty.

### bytecode

The bytecode area contains the instructions for the method. All branching instructions included in a bytecode area must specify addresses within the same bytecode area. All exception handlers defined for a bytecode area must reference addresses within that bytecode area. The bytecode area may only contain bytecodes defined in this specification, their arguments and padding bytes (if needed for alignment).

### Note for the class initializer

Since the initialization values of the static fields are not included in JEFF, a piece of code must be added at the beginning of the class initializer "<clinit>" to perform the initialization of these fields (if needed).
### 3.3.2.9 Caught Exception Table

This structure gives the exception handling information for a method.

It describes exception handlers semantically equivalent and in the same order as the `exception_table` item of the `Code_attribute` structure defined in the Virtual Machine Specification [1].

### 3.3.2.9 Exception Table List

This section is a list of consecutive exception table structures. To each exception table structure corresponds one and only one method of the internal method table of this class area. Some methods have no corresponding exception table structure. The exception tables are stored following the order of storage of the corresponding methods in the internal method table.

An exception table gives the exception handling information for a method.

```java
VMExceptionCatchTable {
    TU2 nCatchCount;
    {
        VMOFFSET ofStartPc;
        VMOFFSET ofEndPc;
        VMOFFSET ofHandlerPc;
        VMOFFSET ofExceptionIndex;
    } VMExceptionCatch [nCatchCount];
}
```

The items of the `VMExceptionCatchTable` structure are as follows:

- **nCatchCount**
  
  The value of the `nCatchCount` item indicates the number of elements in the table.

- **ofStartPc**
  
  Offset of the first byte of the first bytecode in the range where the exception handler is active.

- **ofEndPc**
  
  Offset of the first byte following the last byte of the last bytecode in the range where the exception handler is active.

- **ofHandlerPc**
  
  Offset of the first byte of the first bytecode of the exception handler.

- **ofExceptionIndex**
  
  Offset of a class index, a `VMCINDEX` value stored in the “referenced class table” of the current class. This index identifies the class of the caught exception. The offset value is null if the exception handler has to be called for any kind of exception.
### 3.3.2.10 Constant Data Section

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This section contains the constant data values of the class. They are always referred through an offset.</td>
<td></td>
</tr>
<tr>
<td>Single values of type JINT, JLONG, JFLOAT or JDOUBLE can be referred by the bytecodes ildc, lldc, fildc and dldc. The VMConstUtf8 structures are referred by the sldc bytecode.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note 2</th>
<th>This section contains the constant data values of the class. They are always referred through offsets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single values of type JINT, JLONG, JFLOAT or JDOUBLE can be referred to by the bytecodes ildc, lldc, fildc and dldc. The VMString structures are referred to by the sldc bytecode.</td>
<td></td>
</tr>
</tbody>
</table>

| Newconstarray bytecode refers contiguous set of values of type JDOUBLE, JLONG, JFLOAT, JINT, JSHORT and JBYTE. This bytecode also uses the Utf8 strings stored in VMConstUtf8 structures to create character arrays. |
|--------|-------------------------------------------------|
| Newconstarray bytecode refers contiguous set of values of type JDOUBLE, JLONG, JFLOAT, JINT, JSHORT and JBYTE. This bytecode also uses the strings encoded in VMString structures to create character arrays. |

```java
VMConstantDataSection {
    TU2    nConstFlags;
    TU2    nDoubleNumber;
    TU2    nLongNumber;
    TU2    nFloatNumber;
    TU2    nIntNumber;
    TU2    nShortNumber;
    TU2    nByteNumber;
    TU2    nStringNumber;
    JDOUBLE nDoubleValue[nDoubleNumber];
    JLONG  nLongValue[nLongNumber];
    JFLOAT nFloatValue[nFloatNumber];
    JINT   nIntValue[nIntNumber];
    JSHORT nShortValue[nShortNumber];
    JBYTE  nByteValue[nByteNumber];
    TU1 <0-1 byte pad>
    VMString strConstString[nStringNumber];
}
```

The items of the VMConstantDataSection structure are as follows:

**nConstFlags**

The `nConstFlags` value is a set of flags giving the content of the section as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_CONST_DOUBLE</td>
<td>0x0001</td>
<td>The section contains values of type <code>double</code></td>
</tr>
<tr>
<td>VM_CONST_LONG</td>
<td>0x0002</td>
<td>The section contains values of type <code>long</code></td>
</tr>
<tr>
<td>VM_CONST_FLOAT</td>
<td>0x0004</td>
<td>The section contains values of type <code>float</code></td>
</tr>
<tr>
<td>VM_CONST_INT</td>
<td>0x0008</td>
<td>The section contains values of type <code>int</code></td>
</tr>
<tr>
<td>VM_CONST_SHORT</td>
<td>0x0010</td>
<td>The section contains values of type <code>short</code></td>
</tr>
<tr>
<td>VM_CONST_BYTE</td>
<td>0x0020</td>
<td>The section contains values of type <code>byte</code></td>
</tr>
<tr>
<td>VM_CONST_STRING</td>
<td>0x0040</td>
<td>The section contains constant strings</td>
</tr>
</tbody>
</table>
nDoubleNumber
The number of JDOUBLE values. This non-null value is only present if the VM_CONST_DOUBLE flag is set in nConstFlags.

nLongNumber
The number of JLONG values. This non-null value is only present if the VM_CONST_LONG flag is set in nConstFlags.

nFloatNumber
The number of JFLOAT values. This non-null value is only present if the VM_CONST_FLOAT flag is set in nConstFlags.

nIntNumber
The number of JINT values. This non-null value is only present if the VM_CONST_INT flag is set in nConstFlags.

nShortNumber
The number of JSHORT values. This non-null value is only present if the VM_CONST_SHORT flag is set in nConstFlags.

nByteNumber
The number of JBYTE values. This non-null value is only present if the VM_CONST_BYTE flag is set in nConstFlags.

nStringNumber
The number of VMString structures. This non-null value is only present if the VM_CONST_STRING flag is set in nConstFlags.

nDoubleValue
A value of type double.

nLongValue
A value of type long.

n_FloatValue
A value of type float.

nIntValue
A value of type int.

nShortValue
A value of type short.

nByteValue
A value of type byte.

strConstString
A constant string value (See the definition of the VMString structure).
3.3.3  Attribute Section
This optional section contains the optional attributes for the file, the classes, the methods and the fields. The format for the translation of the attributes described in the Virtual Machine Specification (see [1]) will be included in an Annex of the JEFF specification.

VMAttributeSection {
    VMDOFFSET dofFileAttributeList;
    VMDOFFSET dofClassAttributes[nInternalClassCount];
    TU2 nAttributeTypeCount;
    TU2 nClassAttributeCount;
    VMAttributeType sAttributeType[nAttributeTypeCount];
    VMClassAttributes sClassAttributes[nClassAttributeCount]
    TU2 nAttributeTableCount;
    VMAttributeTable sAttributeTable[nAttributeTableCount];
}

The nInternalClassCount value is defined in the file header.

The items of the VMAttributeSection structure are as follows:

dofFileAttributeList
This value is the offset of a VMAttributeTable structure. This structure defines the attribute list of the file. The offset value is zero if and only if the JEFF file has no file attributes.

dofClassAttributes
The index in this table is the class index. Each entry value is the offset of a VMClassAttributes structure. This structure defines the attributes for the internal class of same index. The offset value is zero if and only if the corresponding class has no attributes.

nAttributeTypeCount
This value is the number of attribute types used in the file.

nClassAttributeCount
This value is the number of VMClassAttributes structures used in the file.
nAttributeTableCount
This value is the number of attribute lists (VMAttributeTable structures) used in the file.

3.3.3.1 Attribute Type
This structure defines an attribute type.

VMAttributeType {
    VMDOFFSET  dofTypeName;
    TU2        nTypeFlags;
    TU2        nTypeLength;
}

The items of the VMAttributeType structure are as follows:

dofTypeName
Offset of a VMString structure stored in the constant data pool. The string value is the attribute type name.

nTypeFlags
This value is a set of flags defining the attribute type. The flag values are the following:

VM_ATTR_INDEXES 0x0001 The attribute contains some index values of type VMINDEX, VMCINDEX, VMMINDEX or VMFINDEX.
VM_ATTR_VMOFFSETS 0x0002 The attribute contains some values of type VMOFFSET.
VM_ATTR_VMDOFFSETS 0x0004 The attribute contains some values of type VMDOFFSET.
VM_ATTR_BYTE_ORDER 0x0008 The elements stored in nData (See the VMAttributeTable structure) contain byte ordered values.
VM_ATTR_CST_LENGTH 0x0010 The length of the attribute is constant and given by the nTypeLength item. This flag can only be used if the length of the attribute structure is not subject to variations caused by the type alignment.

The length of the attribute is constant and given by the nTypeLength item. This flag can only be used if the length of the attribute structure is not subject to variations caused by the type alignment and if the length can be encoded with a TU2 variable.

The VM_ATTR_BYTE_ORDER flag must be set if the VM_ATTR_INDEXES, VM_ATTR_VMOFFSETS, or VM_ATTR_VMDOFFSETS flags are specified.

nTypeLength
This value is the fixed length of the attribute in bytes, not including the type index (See the VMAttributeTable structure). This value is null if the VM_ATTR_CST_LENGTH flag is not set in nTypeFlags.
3.3.3.2 Class Attributes

The attributes used by a class such as the class attributes, the method attribute and the field attributes are defined in this structure.

VMClassAttributes {
    VMDOFFSET dofClassAttributeList;
    VMDOFFSET dofFieldAttributeList[nFieldCount];
    VMDOFFSET dofMethodAttributeList[nMethodCount];
}

The items of the VMClassAttribute structure are as follows:

dofClassAttributeList
This value is the offset of a VMAttributeTable structure. This structure defines the attribute list of the class.

dofFieldAttributeList
This item defines the attribute list of a field. The value is the offset of a VMAttributeTable structure. The position of the offset in the list is equal to the position of the field in the internal field list of the corresponding class. The value of the offset is null if the field has no attributes. The value of nFieldCount is given by the internal field table structure of the corresponding class.

dofMethodAttributeList
This item defines the attribute list of a method. The value is the offset of a VMAttributeTable structure. The position of the offset in the list is equal to the position of the method in the internal method list of the corresponding class. The value of the offset is null if the method has no attributes. The value of nMethodCount is given by the internal method table structure of the corresponding class.

3.3.3.3 Attribute Table

This structure is used to store each attribute list.

VMAttributeTable {
    TU2 nAttributeCount;
    { 
        TU2 nAttributeType;
        TU1 <0-2 byte pad>
        TU4 nTypeLength;
        TU1 nData[nTypeLength];
    } VMAttribute[nAttributeCount]
}

The items of the VMAttributeTable structure are as follows:

nAttributeType
This value is the index of a VMAttributeType structure in the attribute type table. The structure defines the type of the attribute.

nTypeLength
This value is the length, in bytes, of the nData array. This value is only present if the VM_ATTR_CST_LENGTH flag is not set in nTypeFlags item of the VMAttributeType structure pointed to by dofAttributeType. The value must take in account variations of length due to type alignment in the structure of the attribute.
The structure presented is a generic structure that all the attributes must follow. The \texttt{nData}
byte array stands for the true attribute data.

The structure presented is a generic structure that all the attributes must follow. The \texttt{nData}
byte array stands for the true attribute data. These data must follow all the alignment and
padding constraints given in section 3.2.3

### 3.3.4 Symbolic Data Section

This section contains the symbolic information used to identify the elements of the internal and
external classes. The reflection feature also uses this section.

```
VMSymbolicDataSection {
    VMPINDEX pidExtClassPackage[nTotalClassCount-nInternalClassCount];
    TU1 <0-2 byte pad>
    VMDOFFSET dofPackageName[nTotalPackageCount];
    VMDOFFSET dofClassName[nTotalClassCount];

    {
        VMDOFFSET dofFieldName;
        VMDOFFSET dofFieldDescriptor;
    } VMFieldSymbolicInfo[nTotalFieldCount]

    {
        VMDOFFSET dofMethodName;
        VMDOFFSET dofMethodDescriptor;
    } VMMMethodSymbolicInfo[nTotalMethodCount]
}
```

The \texttt{nTotalPackageCount}, \texttt{nTotalClassCount}, \texttt{nInternalClassCount}, \texttt{nTotalFieldCount}
and \texttt{nTotalMethodCount} values are defined in the file header.

The items of the \texttt{VMSymbolicDataSection} structure are as follows:

- **\texttt{pidExtClassPackage}**
  
  This table gives the package of the corresponding external class. If \( n \) is a zero-based index in
  this table, the corresponding entry \texttt{pidExtClassPackage[n]}, gives the package index for the
  external class with a class index value of \( n + n \text{InternalClassCount} \).

- **\texttt{dofPackageName}**
  
  Offset of a \texttt{VMString} structure stored in the constant data pool. The string value is the package
  fully qualified name. The index used in this table is the package index (a \texttt{VMPINDEX} value). If
  the JEFF file references the "default package", a package with no name, the corresponding
  \texttt{dofPackageName} value is the offset of a \texttt{VMString} structure with a null length.

- **\texttt{dofClassName}**
## 3.3.5 Constant Data Pool

This structure stores the constant strings and the descriptors used by the Optional Attribute Section and the Symbolic Data Section.
3.3.5.1 Constant Data Pool Structure

VMConstantDataPool {
  TU4   nStringCount;
  TU4   nDescriptorCount;
  TU4   nMethodDescriptorCount;
  VMString  strConstantString[nStringCount];
  VMDescriptor  sDescriptor[nDescriptorCount];
  VMMethodDescriptor  sMethodDescriptor[nMethodDescriptorCount];
}

The items of the VMConstantDataPool structure are as follows:

nStringCount
The number of constant strings stored in the structure.

nDescriptorCount
The number of individual descriptors stored in the structure. This number does not take in
account the descriptors included in the method descriptors.

nMethodDescriptorCount
The number of method descriptors stored in the structure.

strConstantString
A constant string value (See the definition of the VMString structure).

sDescriptor
A descriptor value as defined below.

sMethodDescriptor
A method descriptor value as defined below.

3.3.5.2 Descriptor

VMDescriptor {
  VMTYPE   tDataType;
  TU1      nDataTypeDimension;
  TU1      <0-1 byte pad>
  VMCINDEX cidDataTypeIndex;
}

The items of the VMDescriptor structure are as follows:

tDataType
The data type. It must be associated to the nDataTypeDimension and cidDataTypeIndex
items to have the full field descriptor.
nDataTypeDimension

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nsi p8iot 7</td>
<td>The array dimension associated with the type. This value is only present if the type is a n-dimensional array, where n &gt;= 2.</td>
</tr>
<tr>
<td>Nsi p8iot 7</td>
<td>The array dimension associated with the type. This value is only present if the type is an n-dimensional array, where n &gt;= 2.</td>
</tr>
</tbody>
</table>

cidDataTypeIndex
The class index associated with the data type. This item is present only if the tDataType is not a primitive type or an array of primitive types.

3.3.5.3 Method Descriptor

VMMethodDescriptor {
   TU2 nArgCount;
   VMDescriptor sArgumentType[nArgCount];
   VMDescriptor sReturnType;
}

The items of the VMMethodDescriptor structure are as follows:

nArgCount

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nsi 31 iote 1 ISC</td>
<td>The number of argument. 0 for a method without argument.</td>
</tr>
<tr>
<td>Nsi 31 iote 1 ISC</td>
<td>The number of arguments, which for a method without any arguments is zero.</td>
</tr>
</tbody>
</table>

sArgumentType
The descriptor of an argument type.

sReturnType
The descriptor of the type returned by the method.

3.3.6 File Signature

3.3.6 Digital Signature
The VMFileSignature structure is not defined.

The JEFF specification does not impose any algorithm or any scheme for the signature an JEFF file. The digital signature of the JEFF file is stored in a VMFileSignature structure defined as follows:

```cpp
VMFileSignature {
    TU1 nSignature[];
}
```

Where the byte array nSignature contains the signature data. The length of the array can be deduced from the position of the VMFileSignature structure and the total size of the JEFF.
4 Bytecodes

This chapter describes the instruction set used in JEFF. The operational semantics of the instruction is not provided, as it does not impact the structural description of the JEFF format.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>An instruction is an opcode followed by its arguments. An opcode itself is coded on one byte. A (&lt;n&gt;)-bytes instruction is an instruction of which arguments take (&lt;n-1&gt;) bytes. A one-byte instruction is an instruction without argument. A two-bytes instruction is an instruction with one argument coded on one byte.</td>
<td></td>
</tr>
<tr>
<td>An instruction is an opcode followed by its operands. An opcode itself is coded on one byte. A (&lt;n&gt;)-bytes instruction is an instruction of which operands take (&lt;n-1&gt;) bytes. A one-byte instruction is an instruction without operand. A two-bytes instruction is an instruction with one operand coded on one byte.</td>
<td></td>
</tr>
</tbody>
</table>

4.1 Principles

The section 4.2 describes only the differences between the class file bytecodes and the JEFF bytecodes. The two instruction sets are equivalent in term of functionalities. The main purpose of the bytecode translation is to create an efficient instruction set adapted to the structure of the file.

<table>
<thead>
<tr>
<th>Translation Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Several operations are applied to the bytecode:</td>
</tr>
<tr>
<td>• The replacement. A bytecode is replaced by another bytecode with the same behavior but using another syntax for its arguments.</td>
</tr>
<tr>
<td>• The replacement. A bytecode is replaced by another bytecode with the same behavior but using another syntax for its operands.</td>
</tr>
<tr>
<td>• The bytecode splitting. A single bytecode with a wide set of functionalities is replaced by several bytecodes implementing a part of the original behavior. The choice of the new bytecode depends on the context.</td>
</tr>
<tr>
<td>• The bytecode grouping. A group of bytecodes frequently used is replaced by a new single bytecode performing the same task.</td>
</tr>
</tbody>
</table>
If an instruction is not described in section 4.2, its syntax shall be unchanged with respect to the one assigned to the instruction of same opcode value in class file bytecode (the mnemonic of the opcode is then the mnemonic of the original opcode as found in class file bytecode prefixed by "jeff-").

The instructions of JEFF bytecode that result from a particular translation are completely defined in section 4.2.

All the instructions not described in section 4.2 are one-byte or two-bytes instructions and are defined in section 4.3.

Section 4.4 provides the complete set of opcodes with their mnemonics used in JEFF bytecode.

**Alignment and Padding**

The bytecodes and their operands follow the rules of alignment and padding defined in 3.2.3 Alignment and Padding.

**4.2 Translations**

This chapter defines normatively all the instructions of JEFF bytecode that are not exactly the same than those found in the class file format bytecode. This chapter describes also all the translation operations from which these JEFF instructions result, but this description is not necessary for the intrinsic definition of the JEFF instructions and the references to the instruction set of class file format are here provided only for information purpose.

**4.2.1 The tableswitch Opcode**

If the original structure of class file bytecode contains the following sequence:
TU1 tableswitch
tU1 <0-3 byte pad>
TS4 nDefault
TS4 nLowValue
TS4 nHighValue
TS4 nOffset [nHighValue - nLowValue + 1]

Where immediately after the padding follow a series of signed 32-bit values: nDefault, nLowValue, nHighValue and then nHighValue - nLowValue + 1 further signed 32-bit offsets.

The translated structure shall be the following sequence:

<table>
<thead>
<tr>
<th>JISC</th>
<th>If the nLowValue and nHighValue values can be converted in 16-bit signed value, the translated structure is:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If the nLowValue and nHighValue values can be converted in 16-bit signed values, the</td>
</tr>
<tr>
<td></td>
<td>translated structure is:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TU1</td>
<td>jeff_stableswitch</td>
</tr>
<tr>
<td>TU1</td>
<td>&lt;0-1 byte pad&gt;</td>
</tr>
<tr>
<td>VMOFFSET</td>
<td>ofDefault</td>
</tr>
<tr>
<td>TS2</td>
<td>nLowValue</td>
</tr>
<tr>
<td>TS2</td>
<td>nHighValue</td>
</tr>
<tr>
<td>VMOFFSET</td>
<td>ofJump [nHighValue - nLowValue + 1]</td>
</tr>
</tbody>
</table>

Otherwise, the translated structure is:

<table>
<thead>
<tr>
<th>TU1</th>
<th>jeff_tableswitch</th>
</tr>
</thead>
<tbody>
<tr>
<td>TU1</td>
<td>&lt;0-1 byte pad&gt;</td>
</tr>
<tr>
<td>VMOFFSET</td>
<td>ofDefault</td>
</tr>
<tr>
<td>TU1</td>
<td>&lt;0-2 byte pad&gt;</td>
</tr>
<tr>
<td>TS4</td>
<td>nLowValue</td>
</tr>
<tr>
<td>TS4</td>
<td>nHighValue</td>
</tr>
<tr>
<td>VMOFFSET</td>
<td>ofJump [nHighValue - nLowValue + 1]</td>
</tr>
</tbody>
</table>

The ofDefault and ofJump values are the jump addresses in the current bytecode block (offsets in bytes from the beginning of the class header structure).

4.2.2 The lookupswitch Opcode

If the original instruction in class file format is:

<table>
<thead>
<tr>
<th>TU1</th>
<th>lookupswitch</th>
</tr>
</thead>
<tbody>
<tr>
<td>TU1</td>
<td>&lt;0-3 byte pad&gt;</td>
</tr>
<tr>
<td>TS4</td>
<td>nDefault</td>
</tr>
<tr>
<td>TU4</td>
<td>nPairs</td>
</tr>
<tr>
<td></td>
<td>match-offset pairs...</td>
</tr>
<tr>
<td>TS4</td>
<td>nMatch</td>
</tr>
<tr>
<td>TS4</td>
<td>nOffset</td>
</tr>
</tbody>
</table>
The translated structure shall be the following sequence:

If all of the nMatch values can be converted in 16-bit signed value, the translated structure is:

```
TU1      jeff_slookupswitch
TU1      <0-1 byte pad>
VMOFFSET ofDefault
TU2      nPairs
TS2      nMatch [nPairs]
VMOFFSET ofJump [nPairs]
```

Otherwise, the translated structure is:

```
TU1      jeff_lookupswitch
TU1      <0-2 byte pad>
VMOFFSET ofDefault
TU2      nPairs
TU1      <0-2 byte pad>
TS4      nMatch [nPairs]
VMOFFSET ofJump [nPairs]
```

The ofDefault and ofJump values are the jump addresses in the current bytecode block (offsets in bytes from the beginning of the class header structure).

### 4.2.3 The new Opcode

If the original instruction in class file format is:

```
TU1      new
TU2      nIndex
```

Where the nIndex value is an index into the constant pool of the local class. The constant pool entry at this index is a CONSTANT_Class.

The translated structure shall be the following sequence:

```
TU1      jeff_new
TU1      <0-1 byte pad>
VMOFFSET ofClassIndex
```

Where the ofClassIndex value is the offset of the class index, a VMCINDEX value stored in the “referenced class table” of the current class.
4.2.4 Opcodes With Class Arguments

4.2.4 Opcodes With a Class Operand

If the original instruction in class file format is:

\[
\begin{align*}
&\text{TU1 }<\text{opcode}> \\
&\text{TU2 }n\text{Index}
\end{align*}
\]

Where <opcode> is anewarray, checkcast or instanceof. The nIndex value is an index into the constant pool of the local class. The constant pool entry at this index is a CONSTANT_Class.

The translated structure shall be a variable-length instruction:

\[
\begin{align*}
&\text{TU1 }<\text{jeff_opcode}> \\
&\text{VMTYPE }t\text{Descriptor} \\
&\text{TU1 }n\text{Dimension} \quad \text{(optional)} \\
&\text{TU1 }<0-1 \text{ byte pad}> \\
&\text{VMOFFSET }o\text{fClassIndex} \quad \text{(optional)}
\end{align*}
\]

The opcode translation array is:

<table>
<thead>
<tr>
<th>classfile opcode</th>
<th>jeff opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>anewarray</td>
<td>jeff_newarray</td>
</tr>
<tr>
<td>checkcast</td>
<td>jeff_checkcast</td>
</tr>
<tr>
<td>instanceof</td>
<td>jeff_instanceof</td>
</tr>
</tbody>
</table>

The tDescriptor value reflects the CONSTANT_Class information. The descriptor associated with the jeff_newarray bytecode has an array dimension equal to the array dimension of CONSTANT_Class structure plus one. The nDimension value is the array dimension associated with the descriptor. This value is only present if the VM_TYPE_MULTI is set in the tDescriptor value. The ofClassIndex value is only present if tDescriptor describes a class or an array of classes. It's the offset of the class index, a VMCINDEX value stored in the "referenced class table" of the current class.
4.2.5 The newarray Opcode

If the original instruction in class file format is:

    TU1 newarray
    TU1 nType

Where the **nType** is a code that indicates the type of array to create.

The translated structure shall be the following sequence:

    TU1   jeff_newarray
    VMTYPE tDescriptor

The **tDescriptor** value reflects the **nType** information. The **VM_TYPE_MONO** flag is always set in this value.

4.2.6 The multianewarray Opcode

If the original instruction in class file format is:

    TU1 multianewarray
    TU2 nIndex
    TU1 nDimensions

Where the **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Class**. The **nDimensions** value represents the number of dimensions of the array to be created.

The translated structure shall be a variable-length instruction:

    TU1   jeff_multianewarray
    TU1   nDimensions
    VMTYPE tDescriptor
    TU1   nArrayDimension
    TU1   <0-1 byte pad>
    VMOFFSET ofClassIndex  (optional)

**IISC**
The **tDescriptor** value reflects the **CONSTANT_Class** information. The **nArrayDimension** value is the array dimension associated with the descriptor. This value is only present if the **VM_TYPE_MULTI** is set in the **tDescriptor** value. The **ofClassIndex** value is only present if **tDescriptor** describes a class or an array of classes. It's the offset of the class index, a **VMCINDEX** value stored in the “referenced class table” of the current class.

4.2.7 Field Opcodes

If the original instruction in class file format is:
The translated structure shall be the following sequence:

<table>
<thead>
<tr>
<th>n35</th>
<th>classfile opcode</th>
<th>jeff opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>n35</td>
<td>classfile opcode</td>
<td>JEFF opcode</td>
</tr>
</tbody>
</table>

getfield   jeff_getfield
getstatic  jeff_getstatic
putfield   jeff_putfield
putstatic  jeff_putstatic

If the instruction points to a field of the current class, the ofFieldInfo value is the offset of a VMFieldInfo structure in the field list of the current class. If the field belongs to another class, the value of ofFieldInfo is the offset of a VMReferencedField structure in the “referenced field table” of the current class.

4.2.8 Method Opcodes

If the original instruction in class file format is:

```
TU1 <opcode>
TU2 nIndex
```

Where <opcode> is invokevirtual, invokevirtual, or invokestatic. The nIndex value is an index into the constant pool of the local class. The constant pool entry at this index is a CONSTANT_Methodref structure.

or

```
TU1 invokeinterface
TU2 nIndex
TU1 nArgs
TU1 0
```

Where the nIndex value is an index into the constant pool of the local class. The constant pool entry at this index is a CONSTANT_InterfaceMethodref structure. The nArgs value is the size in words of the method's arguments in the stack.

The translated structure shall be the following sequence:
The opcode translation array is:

<table>
<thead>
<tr>
<th>INSI</th>
<th>classfile opcode</th>
<th>jeff opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>p35</td>
<td>invokespecial</td>
<td>jeff_invoke.special</td>
</tr>
<tr>
<td></td>
<td>invokevirtual</td>
<td>jeff_invokevirtual</td>
</tr>
<tr>
<td></td>
<td>invokestatic</td>
<td>jeff_invokestatic</td>
</tr>
<tr>
<td></td>
<td>invokeinterface</td>
<td>jeff_invokeinterface</td>
</tr>
</tbody>
</table>

If the instruction points to a method of the current class, the `ofMethodInfo` value is the offset of a `VMMethodInfo` structure in the method list of the current class. If the method belongs to another class, the value of `ofMethodInfo` is the offset of a `VMReferencedMethod` structure in the "referenced method table" of the current class.

### 4.2.9 The ldc Opcodes

If the original instruction in class file format is:

```
TU1 ldc
TU1 nIndex
```

or

```
TU1 ldc_w
TU2 nIndex
```

Where the `nIndex` value is an index into the constant pool of the local class. The constant pool entry at this index is a `CONSTANT_Integer`, a `CONSTANT_Float`, or a `CONSTANT_String`.

or

```
TU1 ldc2_w
TU2 nIndex
```

Where the `nIndex` value is an index into the constant pool of the local class. The constant pool entry at this index is a `CONSTANT_Long`, or a `CONSTANT_Double`.

The translated structure shall be the following sequence:

```
TU1 <JEFF opcode>
TU1 <0-1 byte pad>
VMOFFSET ofConstant
```

Where `<JEFF opcode>` depends of the constant type. The `ofConstant` value is the offset of a data value stored in the constant data section. The type of the value depends of the constant type.
<table>
<thead>
<tr>
<th>jeff opcode</th>
<th>type of the value pointed to by ofConstant</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeff_sldc</td>
<td>VMString</td>
</tr>
<tr>
<td>jeff_ildc</td>
<td>JINT</td>
</tr>
<tr>
<td>jeff_fldc</td>
<td>JFLOAT</td>
</tr>
<tr>
<td>jeff_lldc</td>
<td>JLONG</td>
</tr>
<tr>
<td>jeff_dldc</td>
<td>JDOUBLE</td>
</tr>
</tbody>
</table>

**4.2.10 The wide <opcode> Opcodes**

If the original instruction in class file format is:

```
TU1 wide
TU1 <opcode>
TU2 nIndex
```

Where `<opcode>` is `aload`, `astore`, `dload`, `dstore`, `fload`, `fstore`, `iload`, `istore`, `lload`, `lstore`, or `ret`. The `nIndex` value is an index to a local variable in the current frame.

The translated structure shall be the following sequence:

```
TU1 <JEFF opcode>
TU1 <0-1 byte pad>
TU2 nIndex
```

**4.2.11 The wide iinc Opcode**

If the original instruction in class file format is:

```
wide iinc
```

<table>
<thead>
<tr>
<th>jeff opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeff_iinc_w</td>
</tr>
</tbody>
</table>

Where the opcode translation array is:

```
Where nIndex is unchanged and the opcode translation array is:
```

<table>
<thead>
<tr>
<th>jeff opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeff_iinc_w</td>
</tr>
</tbody>
</table>

Copyright 2000, 2001, J Consortium, All Rights Reserved
TU1 wide
TU1 iinc
TU2 nIndex
TS2 nConstant

Where the nIndex value is an index to a local variable in the current frame. The nConstant value is a signed 16-bit constant.

The translated structure shall be the following sequence:

TU1 jeff_iinc_w
TU1 <0-1 byte pad>
TU2 nIndex
TS2 nConstant

IISC

Where nIndex and nConstant are unchanged.

4.2.12 Jump Opcodes

If the original instruction in class file format is:

TU1 <opcode>
TS2 nOffset

Where <opcode> is goto, if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, ifeq, ifne, iflt, ifge, ifgt, ifle, ifnonnull, ifnull or jsr. Execution proceeds at the offset nOffset from the address of the opcode of this instruction.

The translated structure shall be the following sequence:

TU1 <JEFF opcode>
TU1 <0-1 byte pad>
VMOFFSET ofJump
Where the opcode translation array is:

<table>
<thead>
<tr>
<th>JNSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>i35</td>
</tr>
<tr>
<td>iote 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>classfile opcode</th>
<th>jeff opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto</td>
<td>jeff_goto</td>
</tr>
<tr>
<td>if_acmpeq</td>
<td>jeff_if_acmpeq</td>
</tr>
<tr>
<td>if_acmpne</td>
<td>jeff_if_acmpne</td>
</tr>
<tr>
<td>if_icmpeq</td>
<td>jeff_if_icmpeq</td>
</tr>
<tr>
<td>if_icmpne</td>
<td>jeff_if_icmpne</td>
</tr>
<tr>
<td>if_icmplt</td>
<td>jeff_if_icmplt</td>
</tr>
<tr>
<td>if_icmpge</td>
<td>jeff_if_icmpge</td>
</tr>
<tr>
<td>if_icmpgt</td>
<td>jeff_if_icmpgt</td>
</tr>
<tr>
<td>if_icmple</td>
<td>jeff_if_icmple</td>
</tr>
<tr>
<td>ifeq</td>
<td>jeff_ifeq</td>
</tr>
<tr>
<td>ifne</td>
<td>jeff_ifne</td>
</tr>
<tr>
<td>iflt</td>
<td>jeff_iflt</td>
</tr>
<tr>
<td>ifge</td>
<td>jeff_ifge</td>
</tr>
<tr>
<td>ifgt</td>
<td>jeff_ifgt</td>
</tr>
<tr>
<td>ifle</td>
<td>jeff_ifle</td>
</tr>
<tr>
<td>ifnonnull</td>
<td>jeff_ifnonnull</td>
</tr>
<tr>
<td>ifnull</td>
<td>jeff_ifnull</td>
</tr>
<tr>
<td>jsr</td>
<td>jeff_jsr</td>
</tr>
</tbody>
</table>

The \texttt{ofJump} value is the address of the jump in the current bytecode block. It's an offset (in bytes) from the beginning of the class header structure.

### 4.2.13 Long Jump Opcodes

If the original instruction in class file format is:

\begin{align*}
TU1 & \ (<\texttt{opcode}> \\
TS4 & \ n\texttt{offset}
\end{align*}

Where \texttt{<opcode>} is \texttt{goto\_w} or \texttt{jsr\_w}. Execution proceeds at the offset \texttt{nOffset} from the address of the opcode of this instruction.

The translated structure shall be the following sequence:

\begin{align*}
TU1 & \ <\texttt{JEFF opcode}> \\
TU1 & \ <0-1\ byte\ pad> \\
\texttt{VMOffset ofJump}
\end{align*}

Where the opcode translation array is:

<table>
<thead>
<tr>
<th>JNSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>i35</td>
</tr>
<tr>
<td>iote 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>classfile opcode</th>
<th>jeff opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{goto_w}</td>
<td>jeff_goto</td>
</tr>
<tr>
<td>\texttt{jsr_w}</td>
<td>jeff_jsr</td>
</tr>
</tbody>
</table>

The \texttt{ofJump} value is the address of the jump in the current bytecode block. It's an offset (in bytes) from the beginning of the class header structure.
4.2.14 The sipush Opcode

If the original instruction in class file format is:

| IISC | TU1 sipush  
|      | TU1 nByte1  
|      | TU1 nByte2  
|      | TU1 sipush  
|      | TS1 nByte1  
|      | TU1 nByte2 |

The translated structure shall be the following sequence:

TU1 jeff_sipush
TU1 <0-1 byte pad>
TS2 nValue

Where nValue is a TS2 with the value \((nByte1 \ll 8) | nByte2\).

4.2.15 The newconstarray Opcode

This bytecode creates a new array with the initial values specified in the constant pool. This instruction replaces a sequence of bytecodes creating an empty array and filling it cell by cell.

| IISC | TU1 jeff_newconstarray  
|      | VMTYPE tArrayType  
|      | TU1 <0-1 byte pad>  
|      | TU2 nLength  
|      | VM_OFFSET ofConstData |

The tArrayType is a code that indicates the type of array to create. It must take one of the following values: char[], byte[], short[], boolean[], int[], long[], float[] or double[]. The VM_TYPE_MONO and VM_TYPE_REF flags are always set in this value.

| NSI  | The nLength value is the length, in elements, of the new array.  
| Note | The nLength value is the length, in elements, of the new array. This value cannot be zero.  

| IISC | The ofConstData value is the offset of an array of values in the constant data section. The type of the array depends of the tArrayType value.  
|      | The ofConstData value is the offset of an array of values in the constant data section. The type of the array depends on the tArrayType value. |
A new mono-dimensional array of `nLength` elements is allocated from the garbage-collected heap. All of the elements of the new array are initialized with the values stored in the constant structure. A reference to this new array object is pushed into the operand stack.

### 4.3 Unchanged Instructions

This section defines all the other instruction of JEFF bytecode not previously described in section 4.2. As already noticed, these instructions are kept unchanged in the translation from class file bytecode. In order for this document to be self-contained, they are defined here.

#### 4.3.1 One-Byte Instructions

These instructions have no argument. Here is their list (the mnemonic name of the opcode is preceded here by its value):

- `(0x00)` jeff_nop
- `(0x01)` jeff_acConst_null
- `(0x02)` jeff_iconst_m1
- `(0x03)` jeff_iconst_0
- `(0x04)` jeff_iconst_1
- `(0x05)` jeff_iconst_2

These instructions have no operand. Here is their list (the mnemonic name of the opcode is preceded here by its value):
(0x06) jeff_lconst_3
(0x07) jeff_lconst_4
(0x08) jeff_lconst_5
(0x09) jeff_lconst_0
(0x0a) jeff_lconst_1
(0x0b) jeff_fconst_0
(0x0c) jeff_fconst_1
(0x0d) jeff_fconst_2
(0x0e) jeff_dconst_0
(0x0f) jeff_dconst_1
(0x10) jeff_i1oad_0
(0x11) jeff_i1oad_1
(0x12) jeff_i1oad_2
(0x13) jeff_i1oad_3
(0x14) jeff_f1oad_0
(0x15) jeff_f1oad_1
(0x16) jeff_f1oad_2
(0x17) jeff_f1oad_3
(0x18) jeff_d1oad_0
(0x19) jeff_d1oad_1
(0x1a) jeff_d1oad_2
(0x1b) jeff_d1oad_3
(0x1c) jeff_a1oad_0
(0x1d) jeff_a1oad_1
(0x1e) jeff_a1oad_2
(0x1f) jeff_a1oad_3
(0x20) jeff_iastore_0
(0x21) jeff_iastore_1
(0x22) jeff_iastore_2
(0x23) jeff_iastore_3
(0x24) jeff_lastore_0
(0x25) jeff_lastore_1
(0x26) jeff_lastore_2
(0x27) jeff_lastore_3
(0x28) jeff_fstore_0
(0x29) jeff_fstore_1
(0x2a) jeff_fstore_2
(0x2b) jeff_fstore_3
(0x2c) jeff_dstore_0
(0x2d) jeff_dstore_1
(0x2e) jeff_dstore_2
(0x2f) jeff_dstore_3
(0x30) jeff_istore_0
(0x31) jeff_istore_1
(0x32) jeff_istore_2
(0x33) jeff_istore_3
(0x34) jeff_faload
(0x35) jeff_laload
(0x36) jeff_faload
(0x37) jeff_daload
(0x38) jeff_aaload
(0x39) jeff_baload
(0x3a) jeff_caload
(0x3b) jeff_istore_0
(0x3c) jeff_istore_1
(0x3d) jeff_istore_2
(0x3e) jeff_istore_3
(0x3f) jeff_lastore_0
(0x40) jeff_lastore_1
(0x41) jeff_lastore_2
(0x42) jeff_lastore_3
(0x43) jeff_fstore_0
(0x44) jeff_fstore_1
(0x45) jeff_fstore_2
(0x46) jeff_fstore_3
(0x47) jeff_dstore_0
(0x48) jeff_dstore_1
(0x49) jeff_dstore_2
(0x4a) jeff_dstore_3
(0x4b) jeff_astore_0
(0x4c) jeff_astore_1
(0x4d) jeff_astore_2
(0x4e) jeff_astore_3
(0x4f) jeff_iastore
(0x50) jeff_lastore
(0x51) jeff_fastore
(0x52) jeff_dastore
(0x53) jeff_aastore
(0x54) jeff_bastore
(0x55) jeff_castore
(0x56) jeff_sastore
(0x57) jeff_pop
(0x58) jeff_pop2
(0x59) jeff_dup
(0x5a) jeff_dup_x1
(0x5b) jeff_dup_x2
(0x5c) jeff_dup2
(0x5d) jeff_dup2_x1
(0x5e) jeff_dup2_x2
(0x5f) jeff_swap
(0x60) jeff_ladd
(0x61) jeff_ladd
(0x62) jeff_fadd
(0x63) jeff_dadd
(0x64) jeff_isub
(0x65) jeff_lsub
(0x66) jeff_fsub
(0x67) jeff_dsub
(0x68) jeff_imul
(0x69) jeff_lmul
(0x6a) jeff_fmuli
(0x6b) jeff_dmuli
(0x6c) jeff_idiv
(0x6d) jeff_ldiv
(0x6e) jeff_fdiv
(0x6f) jeff_ddiv
(0x70) jeff_irem
(0x71) jeff_lrem
(0x72) jeff_frem
(0x73) jeff_drem
(0x74) jeff_ineg
(0x75) jeff_lneg
(0x76) jeff_fneg
(0x77) jeff_dneg
(0x78) jeff_ishl
(0x79) jeff_lshl
(0x7a) jeff_ishr
(0x7b) jeff_lshr
(0x7c) jeff_iushr
(0x7d) jeff_lushr
(0x7e) jeff_iand
(0x7f) jeff_land
(0x80) jeff_iorn
(0x81) jeff_lorn
(0x82) jeff_ixorn
(0x83) jeff_lxorn
(0x84) jeff_i2l
(0x85) jeff_i2f
(0x86) jeff_i2d
(0x87) jeff_l2i
(0x88) jeff_l2f
(0x89) jeff_l2d
(0x8a) jeff_f2i
(0x8b) jeff_f2l
(0x8c) jeff_f2d
(0x8d) jeff_f2d
4.3.2 Two-bytes Instructions

These instructions have a one byte argument. Here is their list (the mnemonic name of the opcode is preceded here by its value):

These instructions have a one byte operand. Here is their list (the mnemonic name of the opcode is preceded here by its value):

```java
(0x8e) jeff_d2i
(0x8f) jeff_d2l
(0x90) jeff_d2f
(0x91) jeff_i2b
(0x92) jeff_i2c
(0x93) jeff_i2s
(0x94) jeff_lcmp
(0x95) jeff_fcmpl
(0x96) jeff_fcmpg
(0x97) jeff_dcmpl
(0x98) jeff_dcmpg
```

```java
(0xa9) jeff_ret
```

```java
(0xac) jeff_ireturn
(0xad) jeff_lreturn
(0xae) jeff_freturn
(0xaf) jeff_dreturn
(0xb0) jeff_areturn
(0xb1) jeff_return
(0xbe) jeff_arraylength
(0xbf) jeff_athrow
(0xc2) jeff_monitorenter
(0xc3) jeff_monitorexit
(0xca) jeff_breakpoint
(0xc9) jeff_return
```

```java
(0x32) jeff_bipush
(0x36) jeff_istore
(0x37) jeff_lstore
(0x38) jeff_fstore
(0x39) jeff_dstore
(0x3a) jeff_astore
```

```java
(0x32) jeff_bipush
(0x36) jeff_istore
(0x37) jeff_lstore
(0x38) jeff_fstore
(0x39) jeff_dstore
(0x3a) jeff_astore
```
### 4.4 Complete Opcode Mnemonics by Opcode

This section is the list of all the mnemonics values used in JEFF.

- `(0x00)` `jeff_nop`
- `(0x01)` `jeff_aconst_null`
- `(0x02)` `jeff_iconst_m1`
- `(0x03)` `jeff_iconst_0`
- `(0x04)` `jeff_iconst_1`
- `(0x05)` `jeff_iconst_2`
- `(0x06)` `jeff_iconst_3`
- `(0x07)` `jeff_iconst_4`
- `(0x08)` `jeff_iconst_5`
- `(0x09)` `jeff_iconst_6`
- `(0x0a)` `jeff_iconst_7`
- `(0x0b)` `jeff_lconst_0`
- `(0x0c)` `jeff_lconst_1`
- `(0x0d)` `jeff_fconst_0`
- `(0x0e)` `jeff_fconst_1`
- `(0x0f)` `jeff_fconst_2`
- `(0x10)` `jeff_dconst_0`
- `(0x11)` `jeff_dconst_1`
- `(0x12)` `jeff_bipush`
- `(0x13)` `jeff_sipush`
- `(0x14)` `jeff_unused_0x14`
- `(0x15)` `jeff_iload`
- `(0x16)` `jeff_lload`
- `(0x17)` `jeff_fload`
- `(0x18)` `jeff_dload`
- `(0x19)` `jeff_aload`
- `(0x1a)` `jeff_iaload_0`
- `(0x1b)` `jeff_iaload_1`
- `(0x1c)` `jeff_iaload_2`
- `(0x1d)` `jeff_iaload_3`
- `(0x1e)` `jeff_iaload_4`
- `(0x1f)` `jeff_iaload_5`
- `(0x20)` `jeff_iaload_6`
- `(0x21)` `jeff_iaload_7`
- `(0x22)` `jeff_iaload_8`
- `(0x23)` `jeff_iaload_9`
- `(0x24)` `jeff_iaload_10`
- `(0x25)` `jeff_iaload_11`
- `(0x26)` `jeff_iaload_12`
- `(0x27)` `jeff_iaload_13`
- `(0x28)` `jeff_iaload_14`
- `(0x29)` `jeff_iaload_15`
- `(0x2a)` `jeff_iaload_16`
- `(0x2b)` `jeff_iaload_17`
- `(0x2c)` `jeff_iaload_18`
- `(0x2d)` `jeff_iaload_19`
- `(0x2e)` `jeff_iaload_20`
- `(0x2f)` `jeff_iaload_21`
- `(0x30)` `jeff_faload`
- `(0x31)` `jeff_daload`
- `(0x32)` `jeff_baload`
- `(0x33)` `jeff_caload`
- `(0x34)` `jeff_daload`
- `(0x35)` `jeff_daload`
- `(0x36)` `jeff_iload`
- `(0x37)` `jeff_lload`
- `(0x38)` `jeff_fload`
- `(0x39)` `jeff_dload`
- `(0x3a)` `jeff_daload`
- `(0x3b)` `jeff_daload`
- `(0x3c)` `jeff_daload`
- `(0x3d)` `jeff_daload`
- `(0x3e)` `jeff_daload`
- `(0x3f)` `jeff_daload`
- `(0x40)` `jeff_daload`
- `(0x41)` `jeff_daload`
- `(0x42)` `jeff_daload`
- `(0x43)` `jeff_daload`
- `(0x44)` `jeff_daload`
- `(0x45)` `jeff_daload`
- `(0x46)` `jeff_daload`
- `(0x47)` `jeff_daload`
- `(0x48)` `jeff_daload`
- `(0x49)` `jeff_daload`
- `(0x4a)` `jeff_daload`
- `(0x4b)` `jeff_daload`
- `(0x4c)` `jeff_daload`
- `(0x4d)` `jeff_daload`
- `(0x4e)` `jeff_daload`
- `(0x4f)` `jeff_daload`
- `(0x50)` `jeff_daload`
- `(0x51)` `jeff_daload`
- `(0x52)` `jeff_daload`
- `(0x53)` `jeff_daload`
- `(0x54)` `jeff_daload`
- `(0x55)` `jeff_daload`
- `(0x56)` `jeff_daload`
- `(0x57)` `jeff_daload`
- `(0x58)` `jeff_daload`
- `(0x59)` `jeff_daload`
- `(0x5a)` `jeff_daload`
- `(0x5b)` `jeff_daload`
- `(0x5c)` `jeff_daload`
- `(0x5d)` `jeff_daload`
- `(0x5e)` `jeff_daload`
- `(0x5f)` `jeff_daload`
- `(0x60)` `jeff_daload`
- `(0x61)` `jeff_daload`
- `(0x62)` `jeff_daload`
- `(0x63)` `jeff_daload`
(0x64) jeff_isub
(0x65) jeff_lsub
(0x66) jeff_fsub
(0x67) jeff_dsub
(0x68) jeff_imul
(0x69) jeff_lmul
(0x6a) jeff FMul
(0x6b) jeff_dmul
(0x6c) jeff_idiv
(0x6d) jeff_ldiv
(0x6e) jeff_fdiv
(0x6f) jeff_ddiv
(0x70) jeff_lrem
(0x71) jeff_lrem
(0x72) jeff_frem
(0x73) jeff_drem
(0x74) jeff_ineg
(0x75) jeff_lneg
(0x76) jeff_fneg
(0x77) jeff_dneg
(0x78) jeff_iadd
(0x79) jeff_lshl
(0x7a) jeff_iashr
(0x7b) jeff_lshr
(0x7c) jeff_iuashr
(0x7d) jeff_lushr
(0x7e) jeff_iand
(0x7f) jeff_land
(0x80) jeff_ior
(0x81) jeff_lor
(0x82) jeff_ixor
(0x83) jeff_lxor
(0x84) jeff_iinc
(0x85) jeff_i2l
(0x86) jeff_i2f
(0x87) jeff_i2d
(0x88) jeff_l2i
(0x89) jeff_l2f
(0x8a) jeff_l2d
(0x8b) jeff_f2i
(0x8c) jeff_f2l
(0x8d) jeff_f2d
(0x8e) jeff_d2i
(0x8f) jeff_d2l
(0x90) jeff_d2f
(0x91) jeff_d2l
(0x92) jeff_i2c
(0x93) jeff_l2c
(0x94) jeff_fcmp
(0x95) jeff_fcmp gt
(0x96) jeff_fcmp ge
(0x97) jeff_fcmp eq
(0x98) jeff_fcmp ne
(0x99) jeff_fcmp le
(0x9a) jeff_fcmp lt
(0x9b) jeff_fcmp ltr
(0x9c) jeff_ifne
(0x9d) jeff_iflt
(0x9e) jeff_ifgt
(0x9f) jeff_ifl"
(0xdc) jeff_lstore_w
(0xdd) jeff_aload_w
(0xde) jeff astore_w
## 5 Restrictions

The only restriction of JEFF when compared with class file format is the maximum size of a class area. Within a file, the size of a class area cannot exceed 64Kb. A class area is the block of data included between the `VMClassHeader` structure and the last data specific to the class. The JEFF syntax is very compact and the class area does not include any symbolic information. This means that the corresponding class file can be much bigger than 64Kb.

Otherwise, the following boundaries apply:
- The total size of a file cannot exceed 4Gb.
- The number of classes stored in a file cannot exceed 65,536.
- The number of packages stored in a file cannot exceed 65,536.
- The number of fields in a file cannot exceed 4Giga.
- The number of methods in a file cannot exceed 4Giga.

The only restriction of JEFF when compared with class file format is the maximum size of a class area. Within a file, the size of a class area cannot exceed 65536 bytes. A class area is the block of data included between the `VMClassHeader` structure and the last data specific to the class. The JEFF syntax is very compact and the class area does not include any symbolic information. This means that the corresponding class file can be much bigger than 65536 bytes.

Otherwise, the following limits apply:
- The total size of a file cannot exceed $2^{32}$ bytes.
- The number of classes stored in a file cannot exceed 65,535.
- The number of packages stored in a file cannot exceed 65,534.
- The number of fields in a file cannot exceed $2^{32} - 1$.
- The number of methods in a file cannot exceed $2^{32} - 1$. 