

---

---

**Information technology — Programming  
languages, their environments and system  
software interfaces — JEFF file format**

*Technologies de l'information — Langages de programmation, leurs  
environnements et interfaces de logiciel système — Format de fichier JEFF*

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 20970:2002

**PDF disclaimer**

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 20970:2002

© ISO/IEC 2002

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office  
Case postale 56 • CH-1211 Geneva 20  
Tel. + 41 22 749 01 11  
Fax + 41 22 749 09 47  
E-mail [copyright@iso.ch](mailto:copyright@iso.ch)  
Web [www.iso.ch](http://www.iso.ch)

Printed in Switzerland

# Contents

Page

<b>Foreword</b> .....	<b>v</b>
<b>0 Introduction</b> .....	<b>vi</b>
0.1 What is JEFF.....	vi
0.1.1 Benefits.....	vi
<b>1 Scope and normative references</b> .....	<b>1</b>
1.1 Scope .....	1
1.2 Normative references.....	1
1.3 Definitions.....	2
<b>2 Data Types</b> .....	<b>3</b>
2.1 Basic Types.....	3
2.2 Language Types.....	3
2.3 Strings .....	3
2.3.1 Definition .....	3
2.3.2 Comparison.....	3
2.3.3 Representation.....	3
2.4 Specific Types.....	4
2.4.1 Access Flags.....	4
2.4.2 Type Descriptor.....	5
2.4.3 Offsets.....	7
<b>3 File Structure</b> .....	<b>8</b>
3.1 Definitions.....	8
3.1.1 Fully Qualified Names.....	8
3.1.2 Internal Classes and External Classes .....	8
3.1.3 Fields and Methods.....	8
3.1.4 Field Position.....	9
3.2 Conventions.....	10
3.2.1 Notations .....	10
3.2.2 Byte Order.....	10
3.2.3 Alignment and Padding.....	10
3.3 Definition of the File Structures .....	11
3.3.1 File Header.....	11
3.3.2 Class Section .....	14
3.3.2.1 Class Header.....	14
3.3.2.2 Interface Table.....	16
3.3.2.3 Referenced Class Table.....	16
3.3.2.4 Internal Field Table.....	17
3.3.2.5 Internal Method Table.....	17
3.3.2.6 Referenced Field Table.....	19
3.3.2.7 Referenced Method Table .....	19
3.3.2.8 Bytecode Block Structure.....	20
3.3.2.9 Exception Table List.....	21
3.3.2.10 Constant Data Section.....	21
3.3.3 Attributes Section .....	23
3.3.3.1 Attribute Type.....	24
3.3.3.2 Class Attributes.....	24
3.3.3.3 Attribute Table.....	25
3.3.4 Symbolic Data Section .....	25
3.3.5 Constant Data Pool .....	27
3.3.5.1 Constant Data Pool Structure .....	27
3.3.5.2 Descriptor.....	27
3.3.5.3 Method Descriptor.....	28
3.3.6 Digital Signature.....	28

<b>4</b>	<b>Bytecodes</b>	<b>29</b>
4.1	Principles	29
4.2	Translations	29
4.2.1	The tableswitch Opcode	30
4.2.2	The lookupswitch Opcode	30
4.2.3	The new Opcode	31
4.2.4	Opcodes With a Class Operand	31
4.2.5	The newarray Opcode	32
4.2.6	The multianewarray Opcode	32
4.2.7	Field Opcodes	32
4.2.8	Method Opcodes	33
4.2.9	The ldc Opcodes	34
4.2.10	The wide <opcode> Opcodes	34
4.2.11	The wide iinc Opcode	35
4.2.12	Jump Opcodes	35
4.2.13	Long Jump Opcodes	36
4.2.14	The sipush Opcode	36
4.2.15	The newconstarray Opcode	37
4.3	Unchanged Instructions	37
4.3.1	One-Byte Instructions	37
4.3.2	Two-bytes Instructions	39
4.4	Complete Opcode Mnemonics by Opcode	39
<b>5</b>	<b>Restrictions</b>	<b>41</b>

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 20970:2002

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 20970 was prepared by J Consortium and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

J Consortium, Inc. ("J Consortium") has granted permission to ISO and IEC to use the trademark JEFF for products that comply with ISO/IEC 20970. Thus, implementers of ISO/IEC 20970 may use the J Consortium's JEFF trademark in connection with products that fully meet the requirements of ISO/IEC 20970.

## 0 Introduction

### 0.1 What is JEFF

This International Standard 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 this International Standard 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).

#### 0.1.1 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.

# Information technology — Programming languages, their environments and system software interfaces — JEFF file format

## 1 Scope and normative references

### 1.1 Scope

This International Standard can be used with benefits on all kinds of platform.

This International Standard's most immediate interest is for deploying portable applications on small footprint devices. This International Standard 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.

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

This International Standard 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, this International Standard 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 do 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.

This International Standard 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.

### 1.2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

- [1] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*

- [2] ISO/IEC 10646-1:2000, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*
- [3] ISO/IEC 10646-2:2001, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 2: Supplementary Planes*
- [4] ISO/IEC 10646-1:2000/FDAM 1, *Mathematical symbols and other characters*

NOTE This International Standard 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:

The Java™ Virtual Machine Specification, Second Edition, by Tim Lindholm and Frank Yellin, 496 pages, Addison Wesley, April 1999, ISBN 0201432943.

## 1.3 Definitions

For the purposes of this International Standard, the following terms and definitions apply.

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

STANDARDSISO.COM : Click to view the full text of ISO/IEC 20970:2002

## 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 contiguous bytes. In this document, the expression “null value” is a synonym for a value of zero of the appropriate type.

### 2.1 Basic Types

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

### 2.2 Language Types

The language types are represented internally as follows:

Format Types	Language Types	Format
<b>JBYTE</b>	<b>byte</b>	8-bit signed integer
<b>JSHORT</b>	<b>short</b>	16-bit signed integer
<b>JINT</b>	<b>int</b>	32-bit signed integer
<b>JLONG</b>	<b>long</b>	64-bit signed integer
<b>JFLOAT</b>	<b>float</b>	IEC 60559 [1] single format
<b>JDOUBLE</b>	<b>double</b>	IEC 60559 [1] double format

### 2.3 Strings

#### 2.3.1 Definition

In this International Standard, a *character* is defined in [2], [3], [4]. A *string* is an array of characters. Strings are encoded in the JEFF files as a **VMString** type (see below).

#### 2.3.2 Comparison

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

#### 2.3.3 Representation

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

```
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.

**nStringValue**

This array of byte is an encoding of the value of the string following the UTF-8 encoding algorithm defined in [2], [3], [4].

## 2.4 Specific Types

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

Types	Description	Format
<b>VMACCESS</b>	Access Flag (see 2.4.1)	16-bit vector
<b>VMTYPE</b>	Type descriptor (see 2.4.2)	8-bit vector
<b>VMNCELL</b>	Index in an array of U4 values	16-bit unsigned integer
<b>VMOFFSET</b>	Memory offset (see 2.4.3)	16-bit unsigned integer
<b>VMDOFFSET</b>	Memory offset (see 2.4.3)	32-bit unsigned integer
<b>VMCINDEX</b>	Class Index (see 3.1)	16-bit unsigned integer
<b>VMPINDEX</b>	Package Index (see 3.1)	16-bit unsigned integer
<b>VMMINDEX</b>	Method Index (see 3.1)	32-bit unsigned integer
<b>VMFINDEX</b>	Field Index (see 3.1)	32-bit unsigned integer

### 2.4.1 Access Flags

The **VMACCESS** type describes the access privileges for classes, methods and fields. The **VMACCESS** type is a bit vector with the following values:

Flag Name	Value	Meaning
<b>Class</b>		
<b>ACC_PUBLIC</b>	0x0001	Is public; may be accessed from outside of its package.
<b>ACC_FINAL</b>	0x0010	Is final; no subclasses allowed.
<b>ACC_SUPER</b>	0x0020	Modify the behavior of the jeff_invokespecial bytecodes included in the bytecode area list of this class.
<b>ACC_INTERFACE</b>	0x0200	Is an interface.
<b>ACC_ABSTRACT</b>	0x0400	Is abstract; may not be instantiated.
<b>Field</b>		
<b>ACC_PUBLIC</b>	0x0001	Is public; may be accessed from outside of its package.
<b>ACC_PRIVATE</b>	0x0002	Is private; usable only within the defined class.
<b>ACC_PROTECTED</b>	0x0004	Is protected; may be accessed within subclasses.
<b>ACC_STATIC</b>	0x0008	Is static.
<b>ACC_FINAL</b>	0x0010	Is final; no further overriding or assignment after initialization.
<b>ACC_VOLATILE</b>	0x0040	Is volatile; cannot be cached.
<b>ACC_TRANSIENT</b>	0x0080	Is transient; not written or read by a persistent object manager.
<b>Method</b>		
<b>ACC_PUBLIC</b>	0x0001	Is public; may be accessed from outside of its package.
<b>ACC_PRIVATE</b>	0x0002	Is private; usable only within the defined class.
<b>ACC_PROTECTED</b>	0x0004	Is protected; may be accessed within subclasses.
<b>ACC_STATIC</b>	0x0008	Is static.
<b>ACC_FINAL</b>	0x0010	Is final; no overriding is allowed.
<b>ACC_SYNCHRONIZED</b>	0x0020	Is synchronized; wrap use in monitor lock.
<b>ACC_NATIVE</b>	0x0100	Is native; implemented in a language other than the source language.
<b>ACC_ABSTRACT</b>	0x0400	Is abstract; no implementation is provided.
<b>ACC_STRICT</b>	0x0800	The VM is required to perform strict floating-point operations.

## 2.4.2 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**).

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

The **VMTYPE** type is a byte whose low nibble contains one of the following values:

<b>VM_TYPE_VOID</b>	0x00	Used for the return type of a method
<b>VM_TYPE_SHORT</b>	0x01	
<b>VM_TYPE_INT</b>	0x02	
<b>VM_TYPE_LONG</b>	0x03	
<b>VM_TYPE_BYTE</b>	0x04	
<b>VM_TYPE_CHAR</b>	0x05	
<b>VM_TYPE_FLOAT</b>	0x06	
<b>VM_TYPE_DOUBLE</b>	0x07	
<b>VM_TYPE_BOOLEAN</b>	0x08	
<b>VM_TYPE_OBJECT</b>	0x0A	

These values are interpreted as a bit field as follows:

```
7----4 3--2 1--0
0000 | XX | YY |
```

Where:

- **YY** is an encoded representation of the type size in bytes. The actual type size is:  $1 < YY$ .
- **XX** serves to differentiate types having the same size.

The following flags may be set:

<b>VM_TYPE_TWO_CELL</b>	0x10	for a type using two virtual machine cells (this flag is not set for an array)
<b>VM_TYPE_REF</b>	0x20	for an object or an array
<b>VM_TYPE_MONO</b>	0x40	for a mono-dimensional array
<b>VM_TYPE_MULTI</b>	0x80	for an n-dimensional array, where $n \geq 2$

### Dimension Value

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 **VM\_TYPE\_MULTI** flag is set in the type value and the dimension value must be present.

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.
```

### Class Index

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.

## Summary

Here is a list of the possible code:

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

## Examples

The examples are not normative. They are just an illustration of the above explanations.

A simple instance of the class `mypackage.MyClass`: type = 0x2A, optional dimension = 0x00, class index = index of `mypackage.MyClass`

A primitive type descriptor of a `short`: type = 0x01, optional dimension = 0x00, no class index

A simple array of integers (e.g. `int[5]`): type = 0x62, optional dimension = 0x01, no class index

A simple array of class `mypackage.MyClass` (e.g. `MyClass[5]`): type = 0x6A, optional dimension = 0x01, class index = index of `mypackage.MyClass`

A primitive type descriptor of a `long`: type = 0x13, optional dimension = 0x00, no class index

A 3-dimensional array of `long` (e.g. `long[5][4][]`): type = 0xA3, dimension = 0x03, no class index

A 4-dimensional array of class `mypackage.MyClass` (e.g. `MyClass[5][4][][ ]`): type = 0xAA, dimension = 0x04, class index = index of `mypackage.MyClass`

A `void` return type (for a method): type = 0x00, no dimension, no class index

### 2.4.3 Offsets

There are two types of offset values used in the specification: **VMOFFSET** and **VMDOFFSET**.

A **VMOFFSET** is 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.

A **VMDOFFSET** is an unsigned 32-bit value. This value is an offset in bytes from the beginning of the file header.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 20970:2002

## 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 names are string with 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 “U+ 002E, FULL STOP” 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 “U+ 002E, FULL STOP” followed by the simple name of the class or interface.

#### 3.1.2 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 (of type **VMPINDEX**) for each package. 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 value (of type **VMCINDEX**). The **class** index range is:

<b>0</b>	to	<b>InternalClassCount – 1</b>	for the internal classes
<b>InternalClassCount</b>	to	<b>TotalClassCount – 1</b>	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.

#### 3.1.3 Fields and Methods

##### Field Symbolic Name

A field symbolic name is the concatenation of the field name, a character “U+ 0020, SPACE” and the field descriptor string.

##### Method Symbolic Name

A method symbolic name is the concatenation of the method name, a character “U+ 0020, SPACE” and the method descriptor string.

**Algorithm**

The field indexes are computed as follows:

Let  $n$  be the number of different symbolic names associated to the internal class fields

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 **VMINDEX** value).

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

- 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 deduced:

- 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.

**3.1.4 Field Position**

JEFF includes some information about the position of the field in memory. These pre-computed values are useful to speed up the download of classes and to allow 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 fields of the super-class and the field of the sub-classes are not taken into account.

- 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 members 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 differently.

### 3.2.3 Alignment and Padding

If a platform requires alignment of the multi-byte values in memory, JEFF allows 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:

Elements	Element size, in bytes	Alignment on memory boundaries of
TU1, TS1, JBYTE, VMATYPE	1	1 byte
TU2, TS2, JSHORT, VMACCESS, VMNCELL, VMOFFSET, VMCINDEX, VMPINDEX	2	2 bytes
TU4, TS4, JINT, JFLOAT, VMDOFFSET, VMMINDEX, VMFINDEX	4	4 bytes
JLONG, JDOUBLE	8	8 bytes

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:

```
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 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 sections ordered as follows:

Section	Description
File Header	File identification and directory
Class Section	List of class areas
Attributes Section	List of the attributes
Symbolic Data Section	The symbolic information used by the classes
Constant Data Pool	Set of common constant data
Digital Signature	Signature of the complete file

### File Header

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

### Class Section

The class section describes the content and the properties of each class.

### Attributes Section

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

### Symbolic Data Section

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**

The format magic word is **nMagicWord1** = 0x4A, **nMagicWord2** = 0x45, **nMagicWord3** = 0x46 and **nMagicWord4** = 0x46 ("JEFF" in ASCII).

**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" defines all the two-, four- and eight-bytes long values, except the **JFLOAT** and **JDOUBLE** values.

<b>VM_ORDER_INT_BIG</b>	0x01	If this flag is set, integer values are stored using the big-endian convention. Otherwise, they are stored using the little-endian convention.
<b>VM_ORDER_INT_64_INV</b>	0x02	If this flag is set, the two 32-bit parts of the 64-bit integer values are inverted.
<b>VM_ORDER_FLOAT_BIG</b>	0x04	If this flag is set, <b>JFLOAT</b> and <b>JDOUBLE</b> values are stored using the big-endian convention. Otherwise, they are stored using the little-endian convention.
<b>VM_ORDER_FLOAT_64_INV</b>	0x08	If this flag is set, the two 32-bit parts of the <b>JDOUBLE</b> values are inverted.

**nOptions**

A set of information describing some properties of the internal classes.

This item is an 8-bit vector with the following flag values:

<b>VM_USE_LONG_TYPE</b>	0x01	One of the classes uses the "long" type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
<b>VM_USE_UCS_BMP</b>	0x02	All the characters encoded in the strings of this file are in the "Basic Multilingual Plane" defined in [2], [3], [4], therefore their encoding is in the range U+ 0000 to U+ FFFF included.

<b>VM_USE_FLOAT_TYPE</b>	0x04	One of the classes uses the " <b>float</b> " type and/or the " <b>double</b> " type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
<b>VM_USE_STRICT_FLOAT</b>	0x08	One of the classes contains bytecodes with strict floating-point computation (the " <b>strictfp</b> " keyword is used in the source file).
<b>VM_USE_NATIVE_METHOD</b>	0x10	One of the classes contains native methods.
<b>VM_USE_FINALIZER</b>	0x20	One of the classes has an instance finalizer or a class finalizer.
<b>VM_USE_MONITOR</b>	0x40	One of the classes uses the flag <b>ACC_SYNCHRONIZED</b> or the bytecodes <b>jeff_monitorenter</b> or <b>jeff_monitorexit</b> in one of its methods.

**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 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:

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

#### 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.

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

```
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:

```

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 bit values are the following:

<b>ACC_PUBLIC</b>	Is public; may be accessed from outside its package.
<b>ACC_FINAL</b>	Is final; no subclasses allowed.
<b>ACC_SUPER</b>	Treat superclass methods specially in invokespecial.
<b>ACC_INTERFACE</b>	Is an interface.
<b>ACC_ABSTRACT</b>	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 3.1.4 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 3.1.4 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 **VMReferenceClassTable** 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>
    {
        VMFINDEX    fidFieldIndex;
        VMOFFSET    ofThisClassIndex;
        VMATYPE     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:

<b>ACC_PUBLIC</b>	Is public; may be accessed from outside its package.
<b>ACC_PRIVATE</b>	Is private; usable only within the defined class.
<b>ACC_PROTECTED</b>	Is protected; may be accessed within subclasses.
<b>ACC_STATIC</b>	Is static.
<b>ACC_FINAL</b>	Is final; no further overriding or assignment after initialization.
<b>ACC_VOLATILE</b>	Is volatile; cannot be cached.
<b>ACC_TRANSIENT</b>	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 3.1.4 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:

```

VMMethodInfoTable {
    TU2 nMethodCount;
    TU1 <0-2 byte pad>
    {
        VMINDEX    midMethodIndex;
        VMOFFSET   ofThisClassIndex;
        VMNCELL    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:

**nMethodCount**

The number of methods in the class.

**midMethodIndex**

The method index.

**ofThisClassIndex**

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

**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.

**aAccessFlag**

Method access flag. The possible values are:

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

**ofCode**

For a non-native non-abstract method, this value is the offset of the bytecode block, a **VMBytecodeBlock** structure. For an abstract method, the offset value is null. For a native method, the value is the offset of one of the **nNativeReference** values. Each native method must have a different **ofCode** value.

**nNativeReference**

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.

```
VMReferencedFieldTable {
    TU2 nFieldCount;
    TU1 <0-2 byte pad>
    {
        VMFINDEX fidFieldIndex;
        VMOFFSET ofClassIndex;
        VMATYPE 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 **VMINDEX** 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 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**

The value of the **ncMaxStack** item gives the maximum number of cells on the operand stack at any point during the 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-cell value is **ncMaxLocals-1**. The greatest local variable index for a two-cell 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 offsets within the same bytecode area. All exception handlers defined for a bytecode area must reference offsets within that bytecode area. The bytecode area may only contain bytecodes defined in this specification, their operands 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 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.

```
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

This section contains the constant data values of the class. They are always referred through offsets.

Single values of type **JINT**, **JLONG**, **JFLOAT** or **JDOUBLE** can be referred to by the bytecodes **ildc**, **lldc**, **fldc** and **dldc**. The **VMString** structures are referred to by the **sldc** bytecode.

The **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.

```

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:

<b>VM_CONST_DOUBLE</b>	0x0001	The section contains values of type <b>double</b>
<b>VM_CONST_LONG</b>	0x0002	The section contains values of type <b>long</b>
<b>VM_CONST_FLOAT</b>	0x0004	The section contains values of type <b>float</b>
<b>VM_CONST_INT</b>	0x0008	The section contains values of type <b>int</b>
<b>VM_CONST_SHORT</b>	0x0010	The section contains values of type <b>short</b>
<b>VM_CONST_BYTE</b>	0x0020	The section contains values of type <b>byte</b>
<b>VM_CONST_STRING</b>	0x0040	The section contains constant strings

#### **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**.

**nFloatValue**

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 Attributes Section

This optional section contains the optional attributes for the file, the classes, the methods and the fields. The format of the attributes 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:

<b>VM_ATTR_INDEXES</b>	0x0001	The attribute contains some index values of type <b>VMINDEX</b> , <b>VMCINDEX</b> , <b>VMMINDEX</b> or <b>VMFINDEX</b> .
<b>VM_ATTR_VMOFFSETS</b>	0x0002	The attribute contains some values of type <b>VMOFFSET</b> .
<b>VM_ATTR_VMDOFFSETS</b>	0x0004	The attribute contains some values of type <b>VMDOFFSET</b> .
<b>VM_ATTR_BYTE_ORDER</b>	0x0008	The elements stored in <b>nData</b> (See the <b>VMAttributeTable</b> structure) contain byte ordered values.
<b>VM_ATTR_CST_LENGTH</b>	0x0010	The length of the attribute is constant and given by the <b>nTypeLength</b> 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.

**nData**

The structure presented is a generic structure that all the attributes must follow. The **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;
    } VMMethodSymbolicInfo [nTotalMethodCount]
}

```

The **nTotalPackageCount**, **nTotalClassCount**, **nInternalClassCount**, **nTotalFieldCount** and **nTotalMethodCount** values are defined in the file header.

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

#### **pidExtClassPackage**

This table gives the package of the corresponding external class. If **n** is a zero-based index in this table, the corresponding entry **pidExtClassPackage[n]**, gives the package index for the external class with a class index value of **n + nInternalClassCount**.

#### **dofPackageName**

Offset of a **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 **VMPINDEX** value). If the JEFF file references the "default package", a package with no name, the corresponding **dofPackageName** value is the offset of a **VMString** structure with a null length.

#### **dofClassName**

Offset of a **VMString** structure stored in the constant data pool. The string value is the simple class name. The index of an entry in this table is the class index (a **VMCINDEX** value).

#### **VMFieldSymbolicInfo**

Table of field symbolic information. The index of an entry in this table is the field index (a **VMFINDEX** value).

#### **dofFieldName**

Offset of a **VMString** structure stored in the constant data pool. The string value is the simple field name.

#### **dofFieldDescriptor**

Offset of a **VMDescriptor** structure stored in the constant data pool. The descriptor value gives the field type.

#### **VMMethodSymbolicInfo**

Table of method symbolic information. The index of an entry in this table is the method index (a **VMMINDEX** value).

#### **dofMethodName**

The value is an offset of a **VMString** structure stored in the constant data pool representing either one of the special internal method names, either **<init>** or **<clinit>**, or a method name, stored as a simple name.

**dofMethodDescriptor**

Offset of a **VMMethodDescriptor** structure stored in the constant data pool. The descriptor gives the type of the method arguments and the type of return value.

**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 the descriptors included in the method descriptors into account.

**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>
    VMINDEX     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**

The array dimension associated with the type. This value is only present if the type is an n-dimensional array, where  $n \geq 2$ .

**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**

The number of arguments, which for a method without any arguments is zero.

**sArgumentType**

The descriptor of an argument type.

**sReturnType**

The descriptor of the type returned by the method.

**3.3.6 Digital Signature**

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

```
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.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC 20970:2002

## 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.

An instruction is an opcode followed by its operands. An opcode itself is coded on one byte. A  $\langle n \rangle$ -bytes instruction is an instruction of which operands take  $\langle n-1 \rangle$  bytes. A one-byte instruction is an instruction without operand. A two-bytes instruction is an instruction with one operand coded on one byte.

### 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 functionality. The main purpose of the bytecode translation is to create an efficient instruction set adapted to the structure of the file.

#### Translation Rules

Several operations are applied to the bytecode:

- The replacement. A bytecode is replaced by another bytecode with the same behavior but using another syntax for its operands.
- 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.
- The bytecode grouping. A group of bytecodes frequently used is replaced by a new single bytecode performing the same task.

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 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:

If the **nLowValue** and **nHighValue** values can be converted in 16-bit signed values, the translated structure is:

```
TU1      jeff_stableswitch
TU1      <0-1 byte pad>
VMOFFSET ofDefault
TS2      nLowValue
TS2      nHighValue
VMOFFSET ofJump [nHighValue - nLowValue + 1]
```

Otherwise, the translated structure is:

```
TU1      jeff_tableswitch
TU1      <0-1 byte pad>
VMOFFSET ofDefault
TU1      <0-2 byte pad>
TS4      nLowValue
TS4      nHighValue
VMOFFSET ofJump [nHighValue - nLowValue + 1]
```

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:

```
TU1 lookupswitch
TU1 <0-3 byte pad>
TS4 nDefault
TU4 nPairs
    match-offset pairs...
TS4 nMatch
TS4 nOffset
```

Where immediately after the padding follow a signed 32-bit values: **nDefault**, an unsigned 32-bit values: **nPairs**, and then **nPairs** pairs of signed 32-bit values. Each of the **nPairs** pairs consists of an **int nMatch** and a signed 32-bit **nOffset**.

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      <0-1 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 a Class Operand

If the original instruction in class file format is:

```

TU1 <opcode>
TU2 nIndex

```

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:

```

TU1      <jeff_opcode>
VMTYPE  tDescriptor
TU1      nDimension      (optional)
TU1      <0-1 byte pad>
VMOFFSET ofClassIndex  (optional)

```

The opcode translation array is:

**classfile opcode**    **JEFF opcode**

```

anewarray      jeff_newarray
checkcast      jeff_checkcast
instanceof     jeff_instanceof

```

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 a class. It's the offset of the class index, a **VMINDEX** 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
VMATYPE 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
VMATYPE tDescriptor
TU1      nArrayDimension
TU1      <0-1 byte pad>
VMOFFSET ofClassIndex (optional)

```

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 a class. It's the offset of the class index, a **VMINDEX** 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: