L2/02-042

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:

ANSI 3, JISC 10, SIS 5	Original Text
	Modified Text

INTERNATIONAL J CONSORTIUM™ SPECIFICATION
JEFF™ File Format.

P. O. Box 1565

Cupertino, CA 95015-1565

USA
www.j-consortium.org

J Consortium Specification No. 2000-02.1

JEFF File Format

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 :

ANSI p5 note 1	[1] The Java™ Virtual Machine Specification, Second Edition, by Tim Lindholm and Franck Yellin, 496 pages, Addison Wesley, April 1999, ISBN 0201432943.
	[1] The Java™ Virtual Machine Specification, Second Edition, by Tim Lindholm and **Frank** Yellin, 496 pages, Addison Wesley, April 1999, ISBN 0201432943.

ANSI p6 note 8		[2] The Java™ Language Specification, Second Edition, by Bill Joy, Guy Steele, James Gosling and Gilad Bracha, 544 pages, Addison Wesley, June 5 2000, ISBN 0201310082.
		~~[2] The Java™ Language Specification, Second Edition, by Bill Joy, Guy Steele, James Gosling and Gilad Bracha, 544 pages, Addison Wesley, June 5 2000, ISBN 0201310082.~~ **The next reference is a normative reference:** **[2] Ref IEC 60559:1989, Binary floating point arithmetic for microprocessor systems**

1.4 Definitions

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

2 Data Types

ANSI p6 note 1, 2	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.
	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

ANSI p6 note 3, 4, 5	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.
	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

ANSI p6 note 6	The language types like **int**, **short** or **char** are represented internally as follows:
	The language types ~~like int, short or char~~ are represented internally as follows:

NSI p6
ote 7,
, 9, 10,
1
IS 1
ISC

Format Types	Language Types	Format	Min. Value	Max. Value
JBYTE	**byte**	8-bit signed integer	-128	127
JSHORT	**short**	16-bit signed integer	-32768	32767
JINT	**int**	32-bit signed integer	-2147483648	2147483647
JLONG	**long**	64-bit signed integer	-9.2233e+18	9.2233e+18
JFLOAT	**float**	32-bit IEEE 754	-	-
JDOUBLE	**double**	64-bit IEEE 754	-	-
JCHAR	**char**	16-bit Unicode char	0	Unicode max.

Format Types	Language Types	Format	~~Min. Value~~	~~Max. Value~~
JBYTE	`byte`	8-bit signed integer	~~-128~~	~~127~~
JSHORT	`short`	16-bit signed integer	~~-32768~~	~~32767~~
JINT	`int`	32-bit signed integer	~~-2147483648~~	~~2147483647~~
JLONG	`long`	64-bit signed integer	~~-9.2233e+18~~	~~9.2233e+18~~
JFLOAT	`float`	**IEC 60559 [2] single format**	**-**	**-**
JDOUBLE	`double`	**IEC 60559 [2] double format**	**-**	**-**
~~**JCHAR**~~	~~`char`~~	~~16-bit Unicode char~~	~~0~~	~~Unicode max.~~

NSI p6
ote 12
IS 2B

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.

~~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.~~

ɜIS ꞉omments ɔn Jnicode ꞁISC	-
	## 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

ꓥNSI p6 ıote 13	The character strings are stored in the following structure:
	In the JEFF file, strings are stored according to the following structure:

ꞁISC Ʌinor 2

```
        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.

nStringValue

lISC	The string value encoded with the Utf8 format as defined in the Virtual Machine Specification (see [1]).
;IS :omments on Jnicode	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 :  7 6-------------0` `     0  bits 6-0 of C` 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 :  7---5 4--------------0` `      110   bits 10-6 of C` `Y :  7--6 5---------------0` `      10   bits 5-0 of C` The bytes represent the character with the value ((X & 0x1F) << 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 :  7----4 3---------------0` `      1110   bits 15-12 of C` `Y :  7--6 5-----------------0` `      10   bits 11-6 of C` `Z :  7--6 5-----------------0` `      10   bits 5-0 of C` The bytes represent the character with the value : ((X & 0x0F) << 12) + ((Y & 0x3F) << 6) + (Z & 0x3F).

2.3 Specific Types

ANSI p7 note 1	Theses types are used to store values with a specific meaning.
	These types are used to store values with a specific meaning.

ANSI p7 note 2, 3

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

Types	Description	Format
VMACCESS	Access Flag (see 2.3.1)	16-bit vector
VMTYPE	Type descriptor (see 2.3.2)	8-bit vector
VMNCELL	**Index in an array of U4 values**	16-bit unsigned integer
VMOFFSET	Memory offset (**see 2.3.3**)	16-bit unsigned integer
VMDOFFSET	Memory offset (**see 2.3.3**)	32-bit unsigned integer
VMCINDEX	Class Index **(see 3.1)**	16-bit unsigned integer
VMPINDEX	Package Index **(see 3.1)**	16-bit unsigned integer
VMMINDEX	Method Index **(see 3.1)**	32-bit unsigned integer
VMFINDEX	Field Index **(see 3.1)**	32-bit unsigned integer

2.3.1 Access flags

ANSI p7 note 4	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:

ANSI p7 note 5 MISC			

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

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

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**).

ANSI 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

ANSI p8 note 2	The **VMTYPE** type is a byte built with one of the following values:
	The **VMTYPE** type is a byte **whose low nibble contains one of the following values:**

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

ANSI p8 note 3	These values can be interpreted as a bit field as follows:
	These values **are** interpreted as a bit field as follows:

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

Where:

ANSI p8 note 4, 5	• **YY** is the type size in bytes. The size is: 1 << YY • **XX** is just used to differentiate the types having the same size.
	• **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.

ANSI p8 note 6	The following flags are also set:
	The following flags **may be** 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

ANSI p8 note 7	**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

ANSI p8 note 8, 9, 12	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 **VM_TYPE_MULTI** 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 **VM_TYPE_MULTI** flag is set in the type value and the dimension value **must be present**.

ANSI p8 note 10, 11	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.
	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

ANSI p9 note 1 MISC	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.

IISC	-
	Summary

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

ANSI p9 note 2	-
	The examples are not normative. They are just an illustration of the above explanations.

ANSI p8 note 11	A simple instance of "String": type = 0x2A, optional dimension = 0x00, class index = index of "java.lang.String" 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 "MyClass" (e.g. MyClass[5]) : type = 0x6A, optional dimension = 0x01, class index = index of "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 "MyClass" (e.g. MyClass[5][4][][]): type = 0xAA, dimension = 0x04, class index = index of "MyClass" A "void" return type (for a method): type = 0x00, no dimension, no class index
	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

2.3.3 Offsets

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

ANSI p9 note 4 MISC	A **VMOFFSET** is an unsigned 16-bit value. This value is an offset in bytes from the beginning of a class file header. Depending of where the offset value is located, the corresponding class file header is unambiguous.
	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.

<table>
<tr><td rowspan="2">ANSI p7
note 2
ANSI p9
note 5,
6</td><td>

2.3.4 Index Values

See the File Structure.
</td></tr>
<tr><td>

~~2.3.4 Index Values~~

~~See the File Structure.~~
</td></tr>
</table>

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

ANSI p10 note 1	Fully qualified name have the following definition:
NISC GIS comments on Unicode	Fully qualified **names are string with** the following definition:

NISC GIS comments on Unicode	• 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.
	• 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 **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 **0x002F** followed by the simple name of the class or interface.

ANSI p10 note 2 NISC	**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:
	~~**3.1.2 Symbolic Names**~~ ~~The file specification refers to symbolic names for classes, packages, fields and methods. They are defined as follow:~~

NSI p10
ote 3, 4, 5
IISC
IS
omments
n
nicode

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*”.

~~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 character 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 character 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

<table>
<tr>
<td rowspan="2">ANSI 011 note 1, 2, 3, 4</td>
<td>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 packqage”, 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 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 class 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.</td>
</tr>
<tr>
<td>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:
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.</td>
</tr>
</table>

3.1.4 Fields and Methods

IISC	The field indexes are built as follows: The symbolic name of the internal class fields and the symbolic name of the external class fields are ordered in a table following the crescent lexicographic order. The redundancies are eliminated. All the symbolic names representing the internal class fields are stored at the beginning of the table.
	Field Symbolic Name A field symbolic name is the concatenation of the field name, a character 0x0020 and the field descriptor string. **Method Symbolic Name** A method symbolic name is the concatenation of the method name, a character 0x0020 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 **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

ANSI 011 note 5, 5	JEFF includes some information about the position of the field in memory. These pre-computed values are useful to speedup the download of classes and to have a quick access to the fields at runtime.
	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.

IISC	• When a class A inherits from a class B, the way the instance fields of an instance of A are stored depends of 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.
	• 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.

ANSI 012 note 1	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 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.

ANSI 012 note 2	• 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

ANSI 012 note 3 MISC Minor 2	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.
	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

ANSI 012 note 4	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

ANSI 013 note 1, 2, 3, 4	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 <u>must always</u> 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 <u>must always</u> 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:

SIS 1
MISC

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

Elements	Element size, in bytes	Alignment on memory boundaries of
TU1, TS1, JBYTE, VMTYPE	1	1 byte
TU2, TS2, JSHORT, ~~JCHAR~~, 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**.

ANSI 13 note 5	**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.

ANSI 13 note 6	The file structure is composed of six ordered sections.
	The file structure is composed of six sections ordered as follows:

MISC

Section	Description
File Header	File identification and directory
Class Section	List of class areas
Optional Attributes Section	List of the optional 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

Section	Description
File Header	File identification and directory
Class Section	List of class areas
Attributes Section	List of the ~~optional~~ 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

ANSI 14 note 1	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

ANSI 14 note 2	The class section describes the content of each class (inheritance, fields, methods and code).
	The class section describes the content and the properties of each class.

IISC	**Optional Attributes Section** This optional section contains the optional attributes for the file, the classes, the methods and the fields.
	~~Optional~~ Attributes Section This optional section contains the ~~optional~~ attributes for the file, the classes, the methods and the fields.

Symbolic Data Section

ANSI 14 note 3	In this area are stored all the symbolic information used to identify the classes, the methods and the fields.
	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

ANSI 014 note 4	The format magic word is **nMagicWord1** = 0x4A, **nMagicWord2** = 0x45, **nMagicWord3** = 0x46 and **nMagicWord4** = 0x46 ("JEFF" in Ascii).
	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

MISC	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 height-bytes long values, except the **JFLOAT** and **JDOUBLE** values.
	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.

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

nOptions

ANSI 015 note 1	A set of information on the content of the internal classes.
	A set of information describing some properties of the internal classes.

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

·IS ·omments ·n ·Jnicode			
	VM_USE_LONG_TYPE	0x01	One of the classes uses the "**long**" type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
	VM_USE_UNICODE	0x02	This file contains non-ASCII characters (Unicode).
	VM_USE_FLOAT_TYPE	0x04	One of the classes uses the "**float**" type and/or the "**double**" type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
	VM_USE_STRICT_FLOAT	0x08	One of the classes contains bytecodes with strict floating-point computation (the "**strictfp**" keyword is used in the source file).
	VM_USE_NATIVE_METHOD	0x10	One of the classes contains native methods.
	VM_USE_FINALIZER	0x20	One of the classes has an instance finalizer or a class finalizer.
	VM_USE_MONITOR	0x40	One of the classes uses the flag **ACC_SYNCHRONIZED** or the bytecodes **monitorenter** or **monitorexit** in one of its methods.
	VM_USE_LONG_TYPE	0x01	One of the classes uses the "**long**" type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
	VM_USE_UNICODE	0x02	There is at least one character in the strings encoded in this file which value is not in the range 0x0000 to 0x007F included.
	VM_USE_FLOAT_TYPE	0x04	One of the classes uses the "**float**" type and/or the "**double**" type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
	VM_USE_STRICT_FLOAT	0x08	One of the classes contains bytecodes with strict floating-point computation (the "**strictfp**" keyword is used in the source file).
	VM_USE_NATIVE_METHOD	0x10	One of the classes contains native methods.
	VM_USE_FINALIZER	0x20	One of the classes has an instance finalizer or a class finalizer.
	VM_USE_MONITOR	0x40	One of the classes uses the flag **ACC_SYNCHRONIZED** or the bytecodes **monitorenter** or **monitorexit** 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.

ANSI 16 ote 1	**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:

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.

IISC Minor 2	<pre>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; }</pre>

	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

IISC	Class access flags. The possible values are:
	Class access flags. The possible bit values are:

ACC_PUBLIC	Is public; may be accessed from outside its package.
ACC_FINAL	Is final; no subclasses allowed.

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 **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;
        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:

```
VMMethodInfoTable {
    TU2 nMethodCount;
    TU1 <0-2 byte pad>
    {
        VMMINDEX   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

IISC	The number of method in the class.
	The number of methods in the class.

midMethodIndex
The method index.

ofThisClassIndex
Offset of the current class index, a **VMCINDEX** 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:

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 overriding is allowed.
ACC_SYNCHRONIZED	Is synchronized; wrap use in monitor lock.
ACC_NATIVE	Is native; implemented in a language other than the source language.
ACC_ABSTRACT	Is abstract; no implementation is provided.
ACC_STRICT	The VM is required to perform strict floating-point operations.

ofCode

IISC	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 refer to a separate **nNativeReference** value.
	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

ANSI 21 note 1	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.

```
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.

NSI
23
ote 1
NSI
ote
up. 1

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

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

IISC	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**.
	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

ANSI)32 note 1	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).
	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).

ANSI 024 note 1	**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.

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

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

nCatchCount

IISC	The value of the **nCatchCount** item indicates the number of element in the table.
	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

ANSI 024 note 2	This section contains the constant data values of the class. They are always referred through an offset. Single values of type **JINT**, **JLONG**, **JFLOAT** or **JDOUBLE** can be referred by the bytecodes **ildc**, **lldc**, **fldc** and **dldc**. The **VMConstUtf8** structures are referred by the **sldc** bytecode.
	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.

IISC SIS comments on Unicode	The **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.
	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:

VM_CONST_DOUBLE	0x0001	The section contains values of type **double**
VM_CONST_LONG	0x0002	The section contains values of type **long**
VM_CONST_FLOAT	0x0004	The section contains values of type **float**
VM_CONST_INT	0x0008	The section contains values of type **int**
VM_CONST_SHORT	0x0010	The section contains values of type **short**
VM_CONST_BYTE	0x0020	The section contains values of type **byte**
VM_CONST_STRING	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).

IISC	**nStringValue** The string value encoded with the Utf8 format as defined in the Virtual Machine Specification (see [1]).
	~~**nStringValue**~~ ~~The string value encoded with the Utf8 format as defined in the Virtual Machine Specification (see [1]).~~

ANSI 26 note 1	**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.
	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

IISC	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 iszero if and only if the corresponding class has no attributes.
	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 **VMPINDEX**, **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.

ANSI note sup. 3	**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.
	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 and if the length can be encoded with a TU2 variable.

ANSI note sup. 2	-
	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

ANSI note sup. 4	The structure presented is a generic structure that all the attributes must follow. The **nData** byte array stands for the true attribute data.
	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

ANSI 029 note 1	Offset of a **VMConstUtf8** structure stored in the constant data pool. The string value is the simple (not fully qualified) class name. The index of an entry in this table is the class index (a **VMCINDEX** value).
	Offset of a **VMString** structure stored in the constant data pool. The string value is the simple ~~(not fully qualified)~~ 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

ANSI 029 note 1	Offset of a **VMConstUtf8** structure stored in the constant data pool. The string value is the simple (not fully qualified) field name.
	Offset of a **VMString** structure stored in the constant data pool. The string value is the simple ~~(not fully qualified)~~ 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

ANSI 029 note 1	The value is an offset of a **VMConstUtf8** 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 (not fully qualified) name.
	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 ~~(not fully qualified)~~ 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

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

ANSI p8 note 7	The array dimension associated with the type. This value is only present if the type is a n-dimensional array, where n >= 2.
	The array dimension associated with the type. This value is only present if the type is **an** n-dimensional array, where n >= 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

ANSI p31 note 1 MISC	The number of argument. 0 for a method without argument.
	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.

ANSI p31 note 2	**3.3.6 File Signature**
	3.3.6 Digital Signature

ANSI **031** **note 3** **MISC**	The **VMFileSignature** structure is not defined.
	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.

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.

ANSI 32 note 1	An instruction is an opcode followed by its arguments. An opcode itself is coded on one byte. A <n>-bytes instruction is an instruction of which arguments take <n-1> bytes. A one-byte instruction is an instruction without argument. A two-bytes instruction is an instruction with one argument coded on one byte.
	An instruction is an opcode followed by its operands. An opcode itself is coded on one byte. A <n>-bytes instruction is an instruction of which operands take <n-1> 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

ANSI 32 note 2	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.
	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:

ANSI 32 note 1	• The replacement. A bytecode is replaced by another bytecode with the same behavior but using another syntax for its arguments.
	• 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.

ANSI 32 note 3, 4	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-").
	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

ANSI 32 note 1	The bytecodes and their arguments follow the rules of alignment and padding defined in 3.2.3 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

ANSI 32 note 5	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.
	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:

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


IISC	Where immediately after the padding follow a series of signed 32-bit values: **nDefault**, **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**.
	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.

NSI 32 ote 1	**4.2.4 Opcodes With Class Arguments**
	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:

NSI 35 ote 1	**classfile opcode** **jeff opcode**
	classfile opcode JEFF **opcode**

```
anewarray       jeff_newarray
checkcast       jeff_checkcast
instanceof      jeff_instanceof
```

ISC	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.
	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 **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.
	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 **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:


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

Where **<opcode>** is **getfield**, **getstatic**, **putfield** or **putstatic**. The **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Fieldref**.

The translated structure shall be the following sequence:

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

The opcode translation array is:

ANSI 35 note 1	**classfile opcode**	**jeff opcode**
	classfile opcode	**JEFF opcode**

```
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 **invokespecial**, **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:


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

The opcode translation array is:

ANSI 35 note 1	**classfile opcode**	**jeff opcode**
	classfile opcode	**JEFF opcode**

```
invokespecial   jeff_invokespecial
invokevirtual   jeff_invokevirtual
invokestatic    jeff_invokestatic
invokeinterface jeff_invokeinterface
```

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.

ANSI 335 note 1	classfile opcode	jeff opcode	type of the value pointed to by ofConstant
	classfile opcode	JEFF opcode	type of the value pointed to by ofConstant

```
CONSTANT_String   jeff_sldc    VMString
CONSTANT_Integer  jeff_ildc    JINT
CONSTANT_Float    jeff_fldc    JFLOAT
CONSTANT_Long     jeff_lldc    JLONG
CONSTANT_Double   jeff_dldc    JDOUBLE
```

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
```

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

ANSI 335 note 1	classfile opcode	jeff opcode
	classfile opcode	JEFF opcode

```
wide aload     jeff_aload_w
wide astore    jeff_astore_w
wide dload     jeff_dload_w
wide dstore    jeff_dstore_w
wide fload     jeff_fload_w
wide fstore    jeff_fstore_w
wide iload     jeff_iload_w
wide istore    jeff_istore_w
wide lload     jeff_lload_w
wide lstore    jeff_lstore_w
wide ret       jeff_ret_w
```

4.2.11 The wide iinc Opcode

If the original instruction in class file format is:

```
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:

NSI 35 note 1	classfile opcode	jeff opcode
	classfile opcode	**JEFF opcode**

```
goto              jeff_goto
if_acmpeq         jeff_if_acmpeq
if_acmpne         jeff_if_acmpne
if_icmpeq         jeff_if_icmpeq
if_icmpne         jeff_if_icmpne
if_icmplt         jeff_if_icmplt
if_icmpge         jeff_if_icmpge
if_icmpgt         jeff_if_icmpgt
if_icmple         jeff_if_icmple
ifeq              jeff_ifeq
ifne              jeff_ifne
iflt              jeff_iflt
ifge              jeff_ifge
ifgt              jeff_ifgt
ifle              jeff_ifle
ifnonnull         jeff_ifnonnull
ifnull            jeff_ifnull
jsr               jeff_jsr
```

The **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:

```
TU1 <opcode>
TS4 nOffset
```

Where **<opcode>** is **goto_w** or **jsr_w.** 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:

NSI 35 note 1	classfile opcode	jeff opcode
	classfile opcode	**JEFF opcode**

```
goto_w         jeff_goto
jsr_w          jeff_jsr
```

The **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:

<table>
<tr><td rowspan="2">IISC</td><td><pre>TU1 sipush
TU1 nByte1
TU1 nByte2</pre></td></tr>
<tr><td><pre>TU1 sipush
TS1 nByte1
TU1 nByte2</pre></td></tr>
</table>

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 << 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.

```
TU1      jeff_newconstarray
VMTYPE   tArrayType
TU1      <0-1 byte pad>
TU2      nLength
VMOFFSET 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.

<table>
<tr><td rowspan="2">ANSI note sup. 5</td><td>The nLength value is the length, in elements, of the new array.</td></tr>
<tr><td>The nLength value is the length, in elements, of the new array. This value cannot be zero.</td></tr>
</table>

<table>
<tr><td rowspan="2">IISC</td><td>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.</td></tr>
<tr><td>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.</td></tr>
</table>

IISC SIS comments on Unicode	
	Type of Array / **tArrayType Value** / **Structure pointed to by ofConstData**

Type of Array	tArrayType Value	Structure pointed to by ofConstData
`short[]`	`0x61`	An array of **nLength JSHORT** values.
`int[]`	`0x62`	An array of **nLength JINT** values.
`long[]`	`0x63`	An array of **nLength JLONG** values.
`byte[]`	`0x64`	An array of **nLength JBYTE** values.
`char[]`	`0x65`	An Utf8 string of **nLength** characters (not prefixed by the length)
`float[]`	`0x66`	An array of **nLength JFLOAT** values.
`double[]`	`0x67`	An array of **nLength JDOUBLE** values.
`boolean[]`	`0x68`	An array of **nLength JBYTE** values. Where a zero value means **false** and a non-zero value means **true**.

Type of Array	tArrayType Value	Structure pointed to by ofConstData
`short[]`	`0x61`	An array of **nLength JSHORT** values.
`int[]`	`0x62`	An array of **nLength JINT** values.
`long[]`	`0x63`	An array of **nLength JLONG** values.
`byte[]`	`0x64`	An array of **nLength JBYTE** values.
`char[]`	`0x65`	The first byte of a string of **nLength** characters encoded in a **VMString** structure.
`float[]`	`0x66`	An array of **nLength JFLOAT** values.
`double[]`	`0x67`	An array of **nLength JDOUBLE** values.
`boolean[]`	`0x68`	An array of **nLength JBYTE** values. Where a zero value means **false** and a non-zero value means **true**.

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

ANSI 032 note 1	
	These instructions have no argument. Here is their list (the mnemonic name of the opcode is preceded here by its value):
	These instructions have no operand. Here is their list (the mnemonic name of the opcode is preceded here by its value):

```
(0x00) jeff_nop
(0x01) jeff_aconst_null
(0x02) jeff_iconst_ml
(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_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
(0x1a) jeff_iload_0
(0x1b) jeff_iload_1
(0x1c) jeff_iload_2
(0x1d) jeff_iload_3
(0x1e) jeff_lload_0
(0x1f) jeff_lload_1
(0x20) jeff_lload_2
(0x21) jeff_lload_3
(0x22) jeff_fload_0
(0x23) jeff_fload_1
(0x24) jeff_fload_2
(0x25) jeff_fload_3
(0x26) jeff_dload_0
(0x27) jeff_dload_1
(0x28) jeff_dload_2
(0x29) jeff_dload_3
(0x2a) jeff_aload_0
(0x2b) jeff_aload_1
(0x2c) jeff_aload_2
(0x2d) jeff_aload_3
(0x2e) jeff_iaload
(0x2f) jeff_laload
(0x30) jeff_faload
(0x31) jeff_daload
(0x32) jeff_aaload
(0x33) jeff_baload
(0x34) jeff_caload
(0x35) jeff_saload
(0x3b) jeff_istore_0
(0x3c) jeff_istore_1
(0x3d) jeff_istore_2
(0x3e) jeff_istore_3
(0x3f) jeff_lstore_0
(0x40) jeff_lstore_1
(0x41) jeff_lstore_2
(0x42) jeff_lstore_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_iadd
(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_fmul
(0x6b) jeff_dmul
(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_ior
(0x81) jeff_lor
(0x82) jeff_ixor
(0x83) jeff_lxor
(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_i2b
(0x92) jeff_i2c
(0x93) jeff_i2s
(0x94) jeff_lcmp
(0x95) jeff_fcmpl
(0x96) jeff_fcmpg
(0x97) jeff_dcmpl
(0x98) jeff_dcmpg
```

ANSI note sup. 6	(0xa9) jeff_ret
	~~(0xa9) jeff_ret~~

```
(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
```

4.3.2 Two-bytes Instructions

ANSI 32 note 1	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):

```
(0x10) jeff_bipush
(0x15) jeff_iload
(0x16) jeff_lload
(0x17) jeff_fload
(0x18) jeff_dload
(0x19) jeff_aload
(0x36) jeff_istore
(0x37) jeff_lstore
(0x38) jeff_fstore
(0x39) jeff_dstore
(0x3a) jeff_astore
```


ANSI note sup. 6	-
	(0xa9) jeff_ret

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_ml
(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_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_bipush
(0x11) jeff_sipush
(0x12) jeff_unused_0x12
(0x13) jeff_unused_0x13
(0x14) jeff_unused_0x14
(0x15) jeff_iload
(0x16) jeff_lload
(0x17) jeff_fload
(0x18) jeff_dload
(0x19) jeff_aload
(0x1a) jeff_iload_0
(0x1b) jeff_iload_1
(0x1c) jeff_iload_2
(0x1d) jeff_iload_3
(0x1e) jeff_lload_0
(0x1f) jeff_lload_1
(0x20) jeff_lload_2
(0x21) jeff_lload_3
(0x22) jeff_fload_0
(0x23) jeff_fload_1
(0x24) jeff_fload_2
(0x25) jeff_fload_3
(0x26) jeff_dload_0
(0x27) jeff_dload_1
(0x28) jeff_dload_2
(0x29) jeff_dload_3
(0x2a) jeff_aload_0
(0x2b) jeff_aload_1
(0x2c) jeff_aload_2
(0x2d) jeff_aload_3
(0x2e) jeff_iaload
(0x2f) jeff_laload
(0x30) jeff_faload
(0x31) jeff_daload
(0x32) jeff_aaload
(0x33) jeff_baload
(0x34) jeff_caload
(0x35) jeff_saload
(0x36) jeff_istore
(0x37) jeff_lstore
(0x38) jeff_fstore
(0x39) jeff_dstore
(0x3a) jeff_astore
(0x3b) jeff_istore_0
(0x3c) jeff_istore_1
(0x3d) jeff_istore_2
(0x3e) jeff_istore_3
(0x3f) jeff_lstore_0
(0x40) jeff_lstore_1
(0x41) jeff_lstore_2
(0x42) jeff_lstore_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_iadd
(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_fmul
(0x6b) jeff_dmul
(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_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_i2b
(0x92) jeff_i2c
(0x93) jeff_i2s
(0x94) jeff_lcmp
(0x95) jeff_fcmpl
(0x96) jeff_fcmpg
(0x97) jeff_dcmpl
(0x98) jeff_dcmpg
(0x99) jeff_ifeq
(0x9a) jeff_ifne
(0x9b) jeff_iflt
(0x9c) jeff_ifge
(0x9d) jeff_ifgt
(0x9e) jeff_ifle
(0x9f) jeff_if_icmpeq
(0xa0) jeff_if_icmpne
(0xa1) jeff_if_icmplt
(0xa2) jeff_if_icmpge
(0xa3) jeff_if_icmpgt
(0xa4) jeff_if_icmple
(0xa5) jeff_if_acmpeq
(0xa6) jeff_if_acmpne
(0xa7) jeff_goto
(0xa8) jeff_jsr
(0xa9) jeff_ret
(0xaa) jeff_tableswitch
(0xab) jeff_lookupswitch
(0xac) jeff_ireturn
(0xad) jeff_lreturn
(0xae) jeff_freturn
(0xaf) jeff_dreturn
(0xb0) jeff_areturn
(0xb1) jeff_return
(0xb2) jeff_getstatic
(0xb3) jeff_putstatic
(0xb4) jeff_getfield
(0xb5) jeff_putfield
(0xb6) jeff_invokevirtual
(0xb7) jeff_invokespecial
(0xb8) jeff_invokestatic
(0xb9) jeff_invokeinterface
(0xba) jeff_unused_0xba
(0xbb) jeff_new
(0xbc) jeff_newarray
(0xbd) jeff_unused_0xbd
(0xbe) jeff_arraylength
(0xbf) jeff_athrow
(0xc0) jeff_checkcast
(0xc1) jeff_instanceof
(0xc2) jeff_monitorenter
(0xc3) jeff_monitorexit
(0xc4) jeff_unused_0xc4
(0xc5) jeff_multianewarray
(0xc6) jeff_ifnull
(0xc7) jeff_ifnonnull
(0xc8) jeff_unused_0xc8
(0xc9) jeff_unused_0xc9
(0xca) jeff_breakpoint
(0xcb) jeff_newconstarray
(0xcc) jeff_slookupswitch
(0xcd) jeff_stableswitch
(0xce) jeff_ret_w
(0xcf) jeff_iinc_w
(0xd0) jeff_sldc
(0xd1) jeff_ildc
(0xd2) jeff_lldc
(0xd3) jeff_fldc
(0xd4) jeff_dldc
(0xd5) jeff_dload_w
(0xd6) jeff_dstore_w
(0xd7) jeff_fload_w
(0xd8) jeff_fstore_w
(0xd9) jeff_iload_w
(0xda) jeff_istore_w
(0xdb) jeff_lload_w
```

```
(0xdc) jeff_lstore_w                       (0xde) jeff_astore_w
(0xdd) jeff_aload_w
```

5 Restrictions

ANSI 947 note 1, 2, 3, 4 ANSI note sup. 7	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 $2^{32} - 1$.