3 Class File Format

Important Note

This chapter documents the Java class file format. An important objective of Java as used in WebRunner is that alternative implementations of Java can exist and interact by sharing class files. For this to be possible, these Java implementations must precisely implement the design given here. Elements of the design not covered by this document are not crucial to class file sharing and may be implemented as you choose.

Please contact us directly with any questions about which design elements are essential to a modified or original Java implementation, or for help validating an Java implementation.

Overview

Class files are used to hold compiled versions of both Java classes and Java Interfaces. Compliant Java interpreters must be capable of dealing with all class files that conform to the following specification.

An Java .class file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two or four 8-bit bytes, respectively. The bytes are joined together in big-endian order.

The class file format is described in terms similar to a C structure. However, unlike a C structure,

- There is no “padding” or “alignment” between pieces of the structure.
- Each field of the structure may be of variable size.
- An array may be of variable size. In this case, some field prior to the array will give the array’s dimension.

We use the types `u1`, `u2`, and `u4` to mean an unsigned one-, two-, or four-byte quantity, respectively.

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

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

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

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

Format

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

```
ClassFile {
   u4 magic;
   u4 version;
   u2 constant_pool_count;
   cp_infoconstant_pool[constant_pool_count - 1];
   u2 access_flags;
   u2 this_class;
   u2 super_class;
   u2 interfaces_count;
   u2 interfaces[interfaces_count];
   u2 fields_count;
   field_infofields[fields_count];
   u2 methods_count;
   method_infomethods[methods_count];
   u2 attributes_count;
   attribute_info attributes[attribute_count];
}
```

magic

This field must have the value 0xCAFEBABE.

version

This field contains the version number of the Java compiler that produced this class file. Different version numbers indicate incompatible changes to either the format of the class file or to the bytecodes.

The current Java version number is 45.

constant_pool_count

This field indicates the number of entries in the constant pool table.

constant_pool

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

`constant_pool[0]` is always unused. The values of `constant_pool` entries 1 through `constant_pool_count-1` are described by the bytes that follow. These bytes are explained more fully in the section "The Constant Pool."

access_flags

This field is a set of sixteen flags used by classes, methods, and fields to describe various properties of the field, method, or class. The flags are also used to show how they can be accessed by methods in other classes. Below is a table of all the access flags. The flags that are used by classes are `ACC_PUBLIC`, `ACC_FINAL`, and `ACC_INTERFACE`.

Flag Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Visible to everyone	Class, Method, Variable
ACC_PRIVATE	0x0002	Visible only to the defining class	Method, Variable
ACC_PROTECTED	0x0004	Visible to subclasses	Method, Variable
ACC_STATIC	0x0008	Variable or method is static	Method, Variable
ACC_FINAL	0x0010	No further subclassing, overriding	Class, Method, Variable
ACC_SYNCHRONIZED	0x0020	Wrap method call in monitor lock	Method
ACC_THREADSAFE	0x0040	Can cache in registers	Variable
ACC_TRANSIENT	0x0080	Not written or read by the persistent object manager	Variable
ACC_NATIVE	0x0100	Implemented in C	Method
ACC_INTERFACE	0x0200	Is an interface	Class
ACC_ABSTRACT	0x0400	No definition provided	Method

Access Flags

this_class

This value is an index into the constant pool. `constant_pool[this_class]` must be a class, and gives the index of this class in the constant pool.

super_class

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

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

interfaces_count

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

interfaces

Each value in the array is an index into the constant pool. If an array value is non-zero, then `constant_pool[interfaces[i]], for 0 <= i < interfaces_count,` must be a class, and gives the index of an interface that this class implements.

fields_count

This value gives the number of instance variables, both static and dynamic, defined by this class. This array only includes those variables that are defined explicitly by this class. It does not include those instance variables that are accessible from this class but are inherited from super classes.

`fields`

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

`methods_count`

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

`methods`

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

`attributes_count`

This value gives the number of additional attributes about this class.

`attributes`

A class can have any number of optional attributes associated with it. Currently, the only class attribute recognized is the "SourceFile" attribute, which gives the name of the source file from which this class file was compiled.

Source File Attribute

The "SourceFile" attribute has the following format:

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

`attribute_name_index`

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

`attribute_length`

The length of a SourceFile_attribute must be 2.

`sourcefile_index`

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

Fields

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

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

access_flags

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table "Access Flags" on page 53 which gives the meaning of the bits in this field.

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

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

name_index

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

signature_index

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

attributes_count

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

attributes

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

Any other attributes are skipped.

Constant Value Attribute

The "ConstantValue" attribute has the following format:

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

attribute_name_index

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

attribute_length

The length of a SourceFile_attribute must be 2.

constantvalue_index

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

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

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

Methods

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

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

access_flags

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table “Access Flags” on page 53 which gives the various bits in this field.

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

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

name_index

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

signature_index

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

attributes_count

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

attributes

A field can have any number of optional attributes associated with it. Each attribute has a name, and other additional information. Currently, the only field attribute recognized is the “Code” attribute, which describes the virtual bytecode that can be executed to perform this method.

Any other attributes are skipped.

Code Attribute

The “Code” attribute has the following format:

```
Code_attribute {
   u2 attribute_name_index;
   u2 attribute_length;
   u1 max_stack;
   u1 max_locals;
   u2 code_length;
   u1 code[code_length];
   u2 exception_table_length;
   {  u2    start_pc;
      u2   end_pc;
      u2   handler_pc;
      u2   catch_type;
   }  exception_table[exception_table_length];
   u2 attributes_count;
   attribute_infoattributes[attribute_count];

}
```

attribute_name_index

`constant_pool[attribute_name_index]` is the string “Code.”

attribute_length

This field gives the total length of the “Code” attribute, excluding the initial four bytes.

max_stack

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

max_locals

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

code_length

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

code

These are the actual bytes of the virtual machine code that implement the method. When read into memory, the first byte of code must be aligned onto a multiple-of-four boundary. See the definitions of the the opcodes “tableswitch” and “tablelookup” for more information on alignment requirements.

exception_table_length

The number of entries in the following exception table.

exception_table

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

start_pc, end_pc

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

handler_pc

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

catch_type

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

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

attributes_count

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

attributes

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

Any other attributes are skipped.

Line Number Table Attribute

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

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

attribute_name_index

`constant_pool[attribute_name_index]` will be the string “LineNumberTable.”

attribute_length

This field gives the total length of the LineNumberTable_attribute, excluding the initial four bytes.

line_number_table_length

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

line_number_table

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

start_pc

This field indicates the place in the code at which the code for a new line in the source begins. `source_pc` is an offset from the beginning of the code.

line_number

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

Local Variable Table Attribute

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

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

attribute_name_index

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

attribute_length

This field gives the total length of the LineNumberTable_attribute, excluding the initial four bytes.

local_variable_table_length

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

line_number_table

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

start_pc, length

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

name_index, signature_index

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

slot

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

Constant Pool

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

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_InterfaceMethodref	11
CONSTANT_NameandType	12
CONSTANT_Asciz	1

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

Strings

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

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

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

tag

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

length

The number of bytes in the string. This length does not include the implicit null termination.

bytes

The actual bytes in the string. The null termination is not included.

Classes and Interfaces

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

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

tag

The tag will have the value `CONSTANT_Class`

name_index

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

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

```
int[][]
```

is

```
[[I
```

The class name for

```
Thread[]
```

is

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

Fields and Methods

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

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

CONSTANT_Methodref_info {
   u1 tag;
   u2 class_index;
   u2 name_and_type_index;
}

CONSTANT_InterfaceMethodref_info {
   u1 tag;
   u2 class_index;
   u2 name_and_type_index;
}
```

tag

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

class_index

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

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

name_and_type_index

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

Abstract Fields and Methods

CONSTANT_NameAndType is used to represent a field or method, detached from any particular class or implementation.

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

tag

The tag will have the value CONSTANT_NameAndType

name_index

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

signature_index

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

String Objects

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

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

tag

The tag will have the value CONSTANT_String

name_index

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

The string at constant_pool[string_index] is "encoded" so that strings containing only ASCIZ characters, can be represented using only one byte per character, but characters of up to 16 bits can be represented. The format we use is a modified UTF[1] format.

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

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

The null character (0x0000) and characters in the range 0x0080 to 0x03FF are represented by a pair of two bytes:

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

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

Characters in the range 0x0400 to 0xFFFF are represented by three bytes:

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

Numeric Constants

Four-Byte Constants

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

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

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

`tag`

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

`bytes`

For integers, the four bytes are in the integer. For floats, the four bytes represent the standard IEEE representation of the floating point number.

Eight-Byte Constants

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

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

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

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

`tag`

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

high_bytes, low_bytes

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

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

Signatures

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

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

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

The meaning of the base types is as follows:

B	signed byte
C	character
D	double precision floating point number
F	single precision floating point number
I	integer
J	long integer
L<fullclassname>**;**	an object of the given class
S	nsigned short
Z	boolean
[<length><field sig>	array

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

```
<return signature>   :=  <field type> | V
```

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

An argument signature represents an argument passed to a method:

```
<argument signature> :=  <field type>
```

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

```
<method_signature>   :=  (<arguments signature>) <return signature>
<arguments signature>:=  <argument signature>*
```