

---

# The class File Format

**T**HIS chapter describes the Java virtual machine class file format. Each class file contains the definition of a single class or interface. Although a class or interface need not have an external representation literally contained in a file (for instance, because the class is generated by a class loader), we will colloquially refer to any valid representation of a class or interface as being in the class file format.

A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first. In the Java and Java 2 platforms, this format is supported by interfaces `java.io.DataInput` and `java.io.DataOutput` and classes such as `java.io.DataInputStream` and `java.io.DataOutputStream`.

This chapter defines its own set of data types representing class file data: The types `u1`, `u2`, and `u4` represent an unsigned one-, two-, or four-byte quantity, respectively. In the Java and Java 2 platforms, these types may be read by methods such as `readUnsignedByte`, `readUnsignedShort`, and `readInt` of the interface `java.io.DataInput`.

This chapter presents the class file format using pseudostructures written in a C-like structure notation. To avoid confusion with the fields of classes and class instances, etc., the contents of the structures describing the class file format are referred to as *items*. Successive items are stored in the class file sequentially, without padding or alignment.

*Tables*, consisting of zero or more variable-sized items, are used in several class file structures. Although we use C-like array syntax to refer to table items, the fact that tables are streams of varying-sized structures means that it is not possible to translate a table index directly to a byte offset into the table.

Where we refer to a data structure as an array, it consists of zero or more contiguous fixed-sized items and can be indexed like an array.

## 4.1 Notation and Terminology

We use `this` font for Java virtual machine instructions and for class file structures.

Commentary, designed to clarify the specification, is given as indented text between horizontal lines:

---

Commentary provides intuition, motivation, rationale, examples etc.

---

## 4.2 The ClassFile Structure

A class file consists of a single `ClassFile` structure:

```

ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

The items in the `ClassFile` structure are as follows:

`magic`

The magic item supplies the magic number identifying the class file format; it has the value `0xCAFEBABE`.

`minor_version, major_version`

The values of the `minor_version` and `major_version` items are the minor and major version numbers of this `class` file. Together, a major and a minor version number determine the version of the `class` file format. If a `class` file has major version number  $M$  and minor version number  $m$ , we denote the version of its `class` file format as  $M.m$ . Thus, `class` file format versions may be ordered lexicographically, for example,  $1.5 < 2.0 < 2.1$ .

A Java virtual machine implementation can support a `class` file format of version  $v$  if and only if  $v$  lies in some contiguous range  $M_i.0 \leq v \leq M_j.m$ . Only Sun can specify what range of versions a Java virtual machine implementation conforming to a certain release level of the Java platform may support.<sup>1</sup>

`constant_pool_count`

The value of the `constant_pool_count` item is equal to the number of entries in the `constant_pool` table plus one. A `constant_pool` index is considered valid if it is greater than zero and less than `constant_pool_count`, with the exception for constants of type `long` and `double` noted in §4.5.5.

`constant_pool []`

The `constant_pool` is a table of structures (§4.5) representing various string constants, class and interface names, field names, and other constants that are referred to within the `ClassFile` structure and its substructures. The format of each `constant_pool` table entry is indicated by its first “tag” byte.

The `constant_pool` table is indexed from 1 to `constant_pool_count-1`.

---

<sup>1</sup> The Java virtual machine implementation of Sun’s JDK release 1.0.2 supports `class` file format versions 45.0 through 45.3 inclusive. Sun’s JDK releases 1.1.X can support `class` file formats of versions in the range 45.0 through 45.65535 inclusive. For  $k \geq 2$  implementations of version 1.k of the Java 2 platform can support `class` file formats of versions in the range 45.0 through  $44+k.0$  inclusive.

**access\_flags**

The value of the `access_flags` item is a mask of flags used to denote access permissions to and properties of this class or interface. The interpretation of each flag, when set, is as shown in Table 4.1.

**Table 4.1 Class access and property modifiers**

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared <code>public</code> ; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared <code>final</code> ; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the <i>invokespecial</i> instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared <code>abstract</code> ; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared <code>synthetic</code> ; Not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.

A class may be marked with the ACC\_SYNTHETIC flag to indicate that it was generated by the compiler and does not appear in the source code.

The ACC\_ENUM bit indicates that this class or its superclass is declared as an enumerated type.

An interface is distinguished by its ACC\_INTERFACE flag being set. If its ACC\_INTERFACE flag is not set, this class file defines a class, not an interface.

If the ACC\_INTERFACE flag of this class file is set, its ACC\_ABSTRACT flag must also be set (§2.13.1) and its ACC\_PUBLIC flag may be set. Such a class file must not have any of the other flags in Table 4.1 set.

An annotation type must have its ACC\_ANNOTATION flag set. If the ACC\_ANNOTATION flag is set, the ACC\_INTERFACE flag must be set as well.

If the ACC\_INTERFACE flag of this class file is not set, it may have any of the other flags in Table 4.1 set, **except the**

**ACC\_ANNOTATION flag.** However, such a class file cannot have both its ACC\_FINAL and ACC\_ABSTRACT flags set (§2.8.2).

The setting of the ACC\_SUPER flag indicates which of two alternative semantics for its *invokespecial* instruction the Java virtual machine is to express; the ACC\_SUPER flag exists for backward compatibility for code compiled by Sun's older compilers for the Java programming language. All new implementations of the Java virtual machine should implement the semantics for *invokespecial* documented in this specification. All new compilers to the instruction set of the Java virtual machine should set the ACC\_SUPER flag. Sun's older compilers generated ClassFile flags with ACC\_SUPER unset. Sun's older Java virtual machine implementations ignore the flag if it is set.

All bits of the access\_flags item not assigned in Table 4.1 are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

#### `this_class`

The value of the `this_class` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.5.1) structure representing the class or interface defined by this class file.

#### `super_class`

For a class, the value of the `super_class` item either must be zero or must be a valid index into the `constant_pool` table. If the value of the `super_class` item is nonzero, the `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.5.1) structure representing the direct superclass of the class defined by this class file. Neither the direct superclass nor any of its superclasses may be a final class.

If the value of the `super_class` item is zero, then this class file must represent the class `Object`, the only class or interface without a direct superclass.

For an interface, the value of the `super_class` item must always be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` structure representing the class `Object`.

**interfaces\_count**

The value of the `interfaces_count` item gives the number of direct superinterfaces of this class or interface type.

**interfaces[]**

Each value in the `interfaces` array must be a valid index into the `constant_pool` table. The `constant_pool` entry at each value of `interfaces[i]`, where  $0 \leq i < \text{interfaces\_count}$ , must be a `CONSTANT_Class_info` (§4.5.1) structure representing an interface that is a direct superinterface of this class or interface type, in the left-to-right order given in the source for the type.

**fields\_count**

The value of the `fields_count` item gives the number of `field_info` structures in the `fields` table. The `field_info` (§4.6) structures represent all fields, both class variables and instance variables, declared by this class or interface type.

**fields[]**

Each value in the `fields` table must be a `field_info` (§4.6) structure giving a complete description of a field in this class or interface. The `fields` table includes only those fields that are declared by this class or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

**methods\_count**

The value of the `methods_count` item gives the number of `method_info` structures in the `methods` table.

**methods[]**

Each value in the `methods` table must be a `method_info` (§4.7) structure giving a complete description of a method in this class or interface. If the method is not `native` or `abstract`, the Java virtual machine instructions implementing the method are also supplied.

The `method_info` structures represent all methods declared by this class or interface type, including instance methods, class (`static`) methods, instance initialization methods (§3.9), and any class or interface initialization method (§3.9). The `methods`

table does not include items representing methods that are inherited from superclasses or superinterfaces.

#### `attributes_count`

The value of the `attributes_count` item gives the number of attributes (§4.8) in the `attributes` table of this class.

#### `attributes[]`

Each value of the `attributes` table must be an attribute structure (§4.8).

The only attributes defined by this specification as appearing in the `attributes` table of a `ClassFile` structure are the `InnerClasses` (§4.8.5), `EnclosingMethod` (§4.8.6), `Synthetic` (§4.8.7), `SourceFile` (§4.8.9), `Signature`, and `Deprecated` (§4.8.13) attributes.

A Java virtual machine implementation is required to silently ignore any or all attributes in the `attributes` table of a `ClassFile` structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the `class` file, but only to provide additional descriptive information (§4.8.1).

## 4.3 The Internal Form of Names

### 4.3.1 Fully Qualified Class and Interface Names

Class and interface names that appear in `class` file structures are always represented in a fully qualified form (§2.7.5). Such names are always represented as `CONSTANT_Utf8_info` (§4.5.7) structures and thus may be drawn, where not further constrained, from the entire Unicode character set. Class names and interfaces are referenced both from those `CONSTANT_NameAndType_info` (§4.5.6) structures that have such names as part of their descriptor (§4.4) and from all `CONSTANT_Class_info` (§4.5.1) structures.

For historical reasons the syntax of fully qualified class and interface names that appear in `class` file structures differs from the familiar syntax of fully qualified names documented in §2.7.5. In this internal form, the ASCII periods ('.') that normally separate the identifiers that make up the fully qualified name are replaced by ASCII forward slashes ('/'). The identifiers themselves must be unqualified names as discussed in section (§4.3.2) below. For example, the normal



fully qualified name of class Thread is `java.lang.Thread`. In the form used in descriptors in the `class` file format, a reference to the name of class Thread is implemented using a `CONSTANT_Utf8_info` structure representing the string `"java/lang/Thread"`.

### 4.3.2 Unqualified Names

Names of methods, fields and local variables are stored as *unqualified names*. Unqualified names must not contain the characters `'.'`, `','`, `'['` or `('/')`. Method names are further constrained so that, with the exception of the special method names (§3.9) `<init>` and `<clinit>`, they must not contain the characters `'<` or `'>`.

## 4.4 Descriptors and Signatures

A *descriptor* is a string representing the type of a field or method. Descriptors are represented in the `class` file format using modified UTF-8 strings (§4.5.7) and thus may be drawn, where not further constrained, from the entire Unicode character set.

A *signature* is a string representing the generic type of a field or method, or generic type information for a class declaration.

### 4.4.1 Grammar Notation

Descriptors and signatures are specified using a grammar. This grammar is a set of productions that describe how sequences of characters can form syntactically correct descriptors of various types. Terminal symbols of the grammar are shown in **bold fixed-width** font. Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the production:

*FieldType*:

*BaseType*

*ObjectType*

*ArrayType*

states that a *FieldType* may represent either a *BaseType*, an *ObjectType*, or an *ArrayType*.

A nonterminal symbol on the right-hand side of a production that is followed by an asterisk (\*) represents zero or more possibly different values produced from that nonterminal, appended without any intervening space. Similarly, a nonterminal symbol on the right-hand side of a production that is followed by a plus sign (+) represents one or more possibly different values produced from that nonterminal, appended without any intervening space. The production:

*MethodDescriptor*:

( *ParameterDescriptor*\* ) *ReturnDescriptor*

states that a *MethodDescriptor* represents a left parenthesis, followed by zero or more *ParameterDescriptor* values, followed by a right parenthesis, followed by a *ReturnDescriptor*.

#### 4.4.2 Field Descriptors

A *field descriptor* represents the type of a class, instance, or local variable. It is a series of characters generated by the grammar:

*FieldDescriptor*:

*FieldType*

*ComponentType*:

*FieldType*

*FieldType*:

*BaseType*

*ObjectType*

*ArrayType*

*BaseType*:

B

C

D

F

I

J

S

Z

*ObjectType*:**L Classname;***ArrayType*:[*ComponentType*

The characters of *BaseType*, the **L** and **;** of *ObjectType*, and the **[** of *ArrayType* are all ASCII characters. The **Classname** represents a fully qualified class or interface name. For historical reasons it is encoded in internal form (§4.2). A type descriptor representing an array type is valid only if it represents a type with 255 or fewer dimensions.

The interpretation of the field types is as shown in Table 4.2.

**Table 4.2 Interpretation of *BaseType* characters**

<i>BaseType</i> Character	Type	Interpretation
<b>B</b>	byte	signed byte
<b>C</b>	char	Unicode character
<b>D</b>	double	double-precision floating-point value
<b>F</b>	float	single-precision floating-point value
<b>I</b>	int	integer
<b>J</b>	long	long integer
<b>L Classname;</b>	reference	an instance of class <classname>
<b>S</b>	short	signed short
<b>Z</b>	boolean	true or false
<b>[</b>	reference	one array dimension

For example, the descriptor of an instance variable of type `int` is simply **I**. The descriptor of an instance variable of type `Object` is **Ljava/lang/Object;**. Note that the internal form of the fully qualified name for class `Object` is used. The descriptor of an instance variable that is a multidimensional `double` array,

```
double d[][][];
```

is

[[[D

### 4.4.3 Method Descriptors

A *method descriptor* represents the parameters that the method takes and the value that it returns:

*MethodDescriptor:*

( *ParameterDescriptor\** ) *ReturnDescriptor*

A *parameter descriptor* represents a parameter passed to a method:

*ParameterDescriptor:*

*FieldType*

A *return descriptor* represents the type of the value returned from a method. It is a series of characters generated by the grammar:

*ReturnDescriptor:*

*FieldType*

*VoidDescriptor*

*VoidDescriptor:*

**V**

The character **V** indicates that the method returns no value (its return type is `void`).

A method descriptor is valid only if it represents method parameters with a total length of 255 or less, where that length includes the contribution for `this` in the case of instance or interface method invocations. The total length is calculated by summing the contributions of the individual parameters, where a parameter of type `long` or `double` contributes two units to the length and a parameter of any other type contributes one unit.

For example, the method descriptor for the method

```
Object mymethod(int i, double d, Thread t)
```

is

**(IDLjava/lang/Thread;)Ljava/lang/Object;**

Note that internal forms of the fully qualified names of `Thread` and `Object` are used in the method descriptor.

The method descriptor for `myMethod` is the same whether `myMethod` is a class or an instance method. Although an instance method is passed `this`, a reference to the current

class instance, in addition to its intended parameters, that fact is not reflected in the method descriptor. (A reference to `this` is not passed to a class method.) The reference to `this` is passed implicitly by the method invocation instructions of the Java virtual machine used to invoke instance methods.

#### 4.4.4 Signatures

Signatures are used to encode Java programming language type information that is not part of the Java virtual machine type system, such as generic type and method declarations and parameterized types. See *The Java Language Specification, Third Edition*, for details about such types.

---

This kind of type information is needed to support reflection and debugging, and by the Java compiler.

---

In the following, the terminal symbol **Identifier** is used to denote an identifier for a type, field, local variable, parameter, method name or type variable, as generated by the Java compiler. Such an identifier may contain characters that must not appear in a legal identifier in the Java programming language.

*ClassSignature:*

*FormalTypeParametersopt SuperclassSignature SuperinterfaceSignature\**

A class signature, defined by the production *ClassSignature* above, is used to encode type information about a class declaration. It gives the fully qualified name of the class, describes any formal type parameters it might have, and lists its (possibly parameterized) direct superclass and direct superinterfaces, if any.

*FormalTypeParameters:*

<*FormalTypeParameter*+>

*FormalTypeParameter:*

**Identifier** *ClassBound InterfaceBound\**

A formal type parameter is described by its name, followed by its class and interface bounds. If the class bound does not specify a type, it is taken to be `Object`.

*ClassBound:*

*: FieldTypeSignatureopt*

*InterfaceBound:*

*: FieldTypeSignature*

*SuperclassSignature:*

*ClassTypeSignature*

*SuperinterfaceSignature:*

*ClassTypeSignature*

*FieldTypeSignature:*

*ClassTypeSignature*

*ArrayTypeSignature*

*TypeVariableSignature*

A field type signature, defined by the production *FieldTypeSignature* above, encodes the (possibly parameterized) type for a field, parameter or local variable.

*ClassTypeSignature:*

**L** *PackageSpecifier\* SimpleClassTypeSignature ClassTypeSignatureSuffix\* ;*

*PackageSpecifier:*

**Identifier** / *PackageSpecifier\**

*SimpleClassTypeSignature:*

**Identifier** *TypeArgumentsopt*

*ClassTypeSignatureSuffix:*

. *SimpleClassTypeSignature*

*TypeVariableSignature:*

**T Identifier ;**

*TypeArguments:*

<*TypeArgument*+>

*TypeArgument:*

*WildcardIndicator*opt *FieldTypeSignature*

\*

*WildcardIndicator:*

+

-

*ArrayTypeSignature:*

[*TypeSignature*

*TypeSignature:*

[*FieldTypeSignature*

[*BaseType*

A class type signature gives complete type information for a class or interface type. The class type signature must be formulated such that it can be reliably mapped to the binary name of the class it denotes by erasing any type arguments and converting ‘.’ characters in the signature to ‘\$’ characters.

*MethodTypeSignature:*

*FormalTypeParameters*opt (*TypeSignature*\*) *ReturnType* *ThrowsSignature*\*

*ReturnType:*

*TypeSignature*

*VoidDescriptor**ThrowsSignature:**^ClassTypeSignature**^TypeVariableSignature*

A method signature, defined by the production *MethodTypeSignature* above, encodes the (possibly parameterized) types of the method's formal arguments and of the exceptions it has declared in its throws clause, its (possibly parameterized) return type, and any formal type parameters in the method declaration.

A Java compiler must output generic signature information for any class, interface, constructor or member whose generic signature would include references to type variables or parameterized types. If the throws clause of a method or constructor does not involve type variables, the *ThrowsSignature* may be elided from the *MethodTypeSignature*.

The Java virtual machine does not check the well formedness of the signatures described in this subsection during loading or linking. Instead, these checks are deferred until the signatures are used by reflective methods, as specified in the API of `Class` and members of `java.lang.reflect`. Future versions of the Java virtual machine may be required to perform some or all of these checks during loading or linking.

## 4.5 The Constant Pool

Java virtual machine instructions do not rely on the runtime layout of classes, interfaces, class instances, or arrays. Instead, instructions refer to symbolic information in the `constant_pool` table.

All `constant_pool` table entries have the following general format:

```
cp_info {
    u1 tag;
    u1 info[];
}
```

Each item in the `constant_pool` table must begin with a 1-byte tag indicating the kind of `cp_info` entry. The contents of the `info` array vary with the value of `tag`. The valid tags and their values are listed in Table 4.3. Each tag byte must be fol-



lowed by two or more bytes giving information about the specific constant. The format of the additional information varies with the tag value.

**Table 4.3 Constant pool tags**

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

#### 4.5.1 The CONSTANT\_Class\_info Structure

The CONSTANT\_Class\_info structure is used to represent a class or an interface:

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

The items of the CONSTANT\_Class\_info structure are the following:

tag

The tag item has the value CONSTANT\_Class (7).

name\_index

The value of the name\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info (§4.5.7) structure representing a

valid fully qualified class or interface name encoded in internal form (§4.3.1).

Because arrays are objects, the opcodes *anewarray* and *multianewarray* can reference array “classes” via `CONSTANT_Class_info` (§4.5.1) structures in the `constant_pool` table. For such array classes, the name of the class is the descriptor of the array type. For example, the class name representing a two-dimensional `int` array type

```
int[][]
```

is

```
[[I
```

The class name representing the type array of class `Thread`

```
Thread[]
```

is

```
[Ljava/lang/Thread;
```

An array type descriptor is valid only if it represents 255 or fewer dimensions.

#### 4.5.2 The `CONSTANT_Fieldref_info`, `CONSTANT_Methodref_info`, and `CONSTANT_InterfaceMethodref_info` Structures

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;
}

```

The items of these structures are as follows:

#### tag

The tag item of a `CONSTANT_Fieldref_info` structure has the value `CONSTANT_Fieldref` (9).

The tag item of a `CONSTANT_Methodref_info` structure has the value `CONSTANT_Methodref` (10).

The tag item of a `CONSTANT_InterfaceMethodref_info` structure has the value `CONSTANT_InterfaceMethodref` (11).

#### class\_index

The value of the `class_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.5.1) structure representing a class or interface type that has the field or method as a member.

The `class_index` item of a `CONSTANT_Methodref_info` structure must be a class type, not an interface type. The `class_index` item of a `CONSTANT_InterfaceMethodref_info` structure must be an interface type. The `class_index` item of a `CONSTANT_Fieldref_info` structure may be either a class type or an interface type.

#### name\_and\_type\_index

The value of the `name_and_type_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_NameAndType_info` (§4.5.6) structure. This `constant_pool` entry indicates the name and descriptor of the field or method. In a `CONSTANT_Fieldref_info` the indicated descriptor must be a field descriptor (§4.4.2). Otherwise, the indicated descriptor must be a method descriptor (§4.4.3).

If the name of the method of a `CONSTANT_Methodref_info` structure begins with a '<' ('\u003c'), then the name must be

the special name `<init>`, representing an instance initialization method (§3.9). The return type of such a method must be `void`.

### 4.5.3 The `CONSTANT_String_info` Structure

The `CONSTANT_String_info` structure is used to represent constant objects of the type `String`:

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

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

#### `tag`

The `tag` item of the `CONSTANT_String_info` structure has the value `CONSTANT_String` (8).

#### `string_index`

The value of the `string_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the sequence of characters to which the `String` object is to be initialized.

### 4.5.4 The `CONSTANT_Integer_info` and `CONSTANT_Float_info` Structures

The `CONSTANT_Integer_info` and `CONSTANT_Float_info` structures represent 4-byte numeric (`int` and `float`) constants:

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

The items of these structures are as follows:

**tag**

The tag item of the `CONSTANT_Integer_info` structure has the value `CONSTANT_Integer` (3).

The tag item of the `CONSTANT_Float_info` structure has the value `CONSTANT_Float` (4).

**bytes**

The bytes item of the `CONSTANT_Integer_info` structure represents the value of the `int` constant. The bytes of the value are stored in big-endian (high byte first) order.

The bytes item of the `CONSTANT_Float_info` structure represents the value of the `float` constant in IEEE 754 floating-point single format (§3.3.2). The bytes of the single format representation are stored in big-endian (high byte first) order.

The value represented by the `CONSTANT_Float_info` structure is determined as follows. The bytes of the value are first converted into an `int` constant *bits*. Then:

- If *bits* is `0x7f800000`, the `float` value will be positive infinity.
- If *bits* is `0xff800000`, the `float` value will be negative infinity.
- If *bits* is in the range `0x7f800001` through `0x7fffffff` or in the range `0xff800001` through `0xffffffff`, the `float` value will be NaN.
- In all other cases, let *s*, *e*, and *m* be three values that might be computed from *bits*:

```
int s = ((bits >> 31) == 0) ? 1 : -1;
int e = ((bits >> 23) & 0xff);
int m = (e == 0) ?
        (bits & 0x7fffffff) << 1 :
        (bits & 0x7fffffff) | 0x800000;
```

Then the `float` value equals the result of the mathematical expression  $s \cdot m \cdot 2^{e-150}$ .

#### 4.5.5 The `CONSTANT_Long_info` and `CONSTANT_Double_info` Structures

The `CONSTANT_Long_info` and `CONSTANT_Double_info` represent 8-byte numeric (`long` and `double`) constants:

```

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

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

```

All 8-byte constants take up two entries in the `constant_pool` table of the `class` file. If a `CONSTANT_Long_info` or `CONSTANT_Double_info` structure is the item in the `constant_pool` table at index `n`, then the next usable item in the pool is located at index `n+2`. The `constant_pool` index `n+1` must be valid but is considered unusable.<sup>2</sup>

The items of these structures are as follows:

#### tag

The `tag` item of the `CONSTANT_Long_info` structure has the value `CONSTANT_Long` (5).

The `tag` item of the `CONSTANT_Double_info` structure has the value `CONSTANT_Double` (6).

#### high\_bytes, low\_bytes

The unsigned `high_bytes` and `low_bytes` items of the `CONSTANT_Long_info` structure together represent the value of the long constant  $((\text{long}) \text{high\_bytes} \ll 32) + \text{low\_bytes}$ , where the bytes of each of `high_bytes` and `low_bytes` are stored in big-endian (high byte first) order.

The `high_bytes` and `low_bytes` items of the `CONSTANT_Double_info` structure together represent the double value in IEEE 754 floating-point double format (§3.3.2). The bytes of each item are stored in big-endian (high byte first) order.

The value represented by the `CONSTANT_Double_info` structure is determined as follows. The `high_bytes` and `low_bytes` items are first converted into the long constant *bits*,

---

<sup>2</sup> In retrospect, making 8-byte constants take two constant pool entries was a poor choice.

which is equal to  $((\text{long}) \text{high\_bytes} \ll 32) + \text{low\_bytes}$ .

Then:

- If *bits* is `0x7ff0000000000000L`, the double value will be positive infinity.
- If *bits* is `0xfff0000000000000L`, the double value will be negative infinity.
- If *bits* is in the range `0x7ff0000000000001L` through `0x7fffffffffffffffffL` or in the range `0xfff0000000000001L` through `0xfffffffffffffffffL`, the double value will be NaN.
- In all other cases, let *s*, *e*, and *m* be three values that might be computed from *bits*:

```
int s = ((bits >> 63) == 0) ? 1 : -1;
int e = (int)((bits >> 52) & 0x7ffL);
long m = (e == 0) ?
    (bits & 0xffffffffffffffffL) << 1 :
    (bits & 0xffffffffffffffffL) | 0x100000000000000L;
```

Then the floating-point value equals the double value of the mathematical expression  $s \cdot m \cdot 2^{e-1075}$ .

#### 4.5.6 The CONSTANT\_NameAndType\_info Structure

The `CONSTANT_NameAndType_info` structure is used to represent a field or method, without indicating which class or interface type it belongs to:

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

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

`tag`

The `tag` item of the `CONSTANT_NameAndType_info` structure has the value `CONSTANT_NameAndType` (12).

**name\_index**

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing either the special method name `<init>` (§3.9) or a valid unqualified name (§4.3.2) denoting a field or method. .

**descriptor\_index**

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing a valid field descriptor (§4.4.2) or method descriptor (§4.4.3).

**4.5.7 The CONSTANT\_Utf8\_info Structure**

The `CONSTANT_Utf8_info` structure is used to represent constant string values. String content is encoded in *modified UTF-8*.

Modified UTF-8 strings are encoded so that character sequences that contain only non-null ASCII characters can be represented using only 1 byte per character, but all Unicode characters can be represented. All characters in the range `'\u0001'` to `'\u007F'` are represented by a single byte:

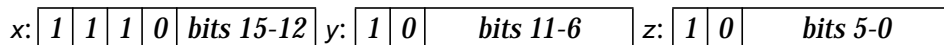


The 7 bits of data in the byte give the value of the character represented. The null character (`'\u0000'`) and characters in the range `'\u0080'` to `'\u07FF'` are represented by a pair of bytes *x* and *y*:



The bytes represent the character with the value  $((x \& 0x1f) \ll 6) + (y \& 0x3f)$ .

Characters in the range `'\u0800'` to `'\uFFFF'` are represented by 3 bytes *x*, *y*, and *z*:



The character with the value  $((x \& 0xf) \ll 12) + ((y \& 0x3f) \ll 6) + (z \& 0x3f)$  is represented by the bytes.



Characters with code points above U+FFFF (so-called *supplementary characters*) are represented by separately encoding the two surrogate code units of their UTF-16 representation. Each of the surrogate code units is represented by three bytes. This means, supplementary characters are represented by six bytes, u, v, w, x, y, and z:

$$u: \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 0 \\ \hline \end{array} \begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 1 \\ \hline \end{array} \quad v: \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \end{array} \begin{array}{|c|} \hline (bits\ 20-16)-1 \\ \hline \end{array} \quad w: \begin{array}{|c|c|} \hline 1 & 0 \\ \hline \end{array} \begin{array}{|c|} \hline bits\ 15-10 \\ \hline \end{array}$$

$$x: \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 0 \\ \hline \end{array} \begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 1 \\ \hline \end{array} \quad y: \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 1 \\ \hline \end{array} \begin{array}{|c|} \hline bits\ 9-6 \\ \hline \end{array} \quad z: \begin{array}{|c|c|} \hline 1 & 0 \\ \hline \end{array} \begin{array}{|c|} \hline bits\ 5-0 \\ \hline \end{array}$$

The character with the value  $0x10000 + ((v \& 0x0f) \ll 16) + ((w \& 0x3f) \ll 10) + (y \& 0x0f) \ll 6 + (z \& 0x3f)$  is represented by the six bytes.

The bytes of multibyte characters are stored in the class file in big-endian (high byte first) order.

There are two differences between this format and the “standard” UTF-8 format. First, the null character (`char`) $\emptyset$  is encoded using the 2-byte format rather than the 1-byte format, so that modified UTF-8 strings never have embedded nulls. Second, only the 1-byte, 2-byte, and 3-byte formats of standard UTF-8 are used. The Java VM does not recognize the four-byte format of standard UTF-8; it uses its own two-times-three-byte format instead.

For more information regarding the standard UTF-8 format, see section 3.9 *Unicode Encoding Forms of The Unicode Standard, Version 4.0*.

The `CONSTANT_Utf8_info` structure is

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

The items of the `CONSTANT_Utf8_info` structure are the following:

**tag**

The `tag` item of the `CONSTANT_Utf8_info` structure has the value `CONSTANT_Utf8 (1)`.

**length**

The value of the `length` item gives the number of bytes in the bytes array (not the length of the resulting string). The strings in the `CONSTANT_Utf8_info` structure are not null-terminated.

**bytes[]**

The bytes array contains the bytes of the string. No byte may have the value (byte)0 or lie in the range (byte)0xf0-(byte)0xff.

## 4.6 Fields

Each field is described by a `field_info` structure. No two fields in one `class` file may have the same name and descriptor (§4.4.2). The format of this structure is

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

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

**access\_flags**

The value of the `access_flags` item is a mask of flags used to denote access permission to and properties of this field. The interpretation of each flag, when set, is as shown in Table 4.4.

**Table 4.4 Field access and property flags**

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared <code>public</code> ; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared <code>private</code> ; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared <code>protected</code> ; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared <code>static</code> .
ACC_FINAL	0x0010	Declared <code>final</code> ; no further assignment after initialization.
ACC_VOLATILE	0x0040	Declared <code>volatile</code> ; cannot be cached.
ACC_TRANSIENT	0x0080	Declared <code>transient</code> ; not written or read by a persistent object manager.
ACC_SYNTHETIC	0x1000	Declared <code>synthetic</code> ; Not present in the source code.
ACC_ENUM	0x4000	Declared as an element of an enum.

The ACC\_ENUM bit indicates that this field is being used to hold an element of an enumerated type.

A field may be marked with the ACC\_SYNTHETIC flag to indicate that it was generated by the compiler and does not appear in the source code.

Fields of classes may set any of the flags in Table 4.4. However, a specific field of a class may have at most one of its ACC\_PRIVATE, ACC\_PROTECTED, and ACC\_PUBLIC flags set (§2.7.4) and must not have both its ACC\_FINAL and ACC\_VOLATILE flags set (§2.9.1).

All fields of interfaces must have their ACC\_PUBLIC, ACC\_STATIC, and ACC\_FINAL flags set; they may have their ACC\_SYNTHETIC flag set and must not have any of the other flags in Table 4.4 set (§2.13.3.1).

All bits of the `access_flags` item not assigned in Table 4.4 are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

### name\_index

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure which must represent a valid unqualified name (§4.3.2) denoting a field.

### descriptor\_index

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure that must represent a valid field descriptor (§4.4.2).

### attributes\_count

The value of the `attributes_count` item indicates the number of additional attributes (§4.8) of this field.

### attributes[]

Each value of the `attributes` table must be an attribute structure (§4.8). A field can have any number of attributes associated with it.

The attributes defined by this specification as appearing in the `attributes` table of a `field_info` structure are the `ConstantValue` (§4.8.2), `Synthetic` (§4.8.7), `Signature` (§4.8.8) and `Deprecated` (§4.8.13) attributes.

A Java virtual machine implementation must recognize and correctly read `ConstantValue` (§4.8.2) attributes found in the `attributes` table of a `field_info` structure. **If a Java virtual machine recognizes class files whose major version is 49.0 or above, it must recognize and correctly read `Signature` (§4.8.8) attributes found in the `attributes` table of a `field_info` structure.** A Java virtual machine implementation is required to silently ignore any or all other attributes in the `attributes` table that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.8.1).

## 4.7 Methods

Each method, including each instance initialization method (§3.9) and the class or interface initialization method (§3.9), is described by a `method_info` structure. No two methods in one class file may have the same name and descriptor (§4.4.3).

The structure has the following format:

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

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

`access_flags`

The value of the `access_flags` item is a mask of flags used to denote access permission to and properties of this method. The interpretation of each flag, when set, is as shown in Table 4.5.

**Table 4.5 Method access and property flags**

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared <code>public</code> ; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared <code>private</code> ; accessible only within the defining class.
ACC_PROTECTED	0x0004	Declared <code>protected</code> ; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared <code>static</code> .
ACC_FINAL	0x0010	Declared <code>final</code> ; must not be overridden.
ACC_SYNCHRONIZED	0x0020	Declared <code>synchronized</code> ; invocation is wrapped in a monitor lock.
ACC_BRIDGE	0x0040	A bridge method, generated by the compiler.
ACC_VARARGS	0x0080	Declared with variable number of arguments.
ACC_NATIVE	0x0100	Declared <code>native</code> ; implemented in a language other than Java.
ACC_ABSTRACT	0x0400	Declared <code>abstract</code> ; no implementation is provided.
ACC_STRICT	0x0800	Declared <code>strictfp</code> ; floating-point mode is FP-strict
ACC_SYNTHETIC	0x1000	Declared <code>synthetic</code> ; Not present in the source code.

The `ACC_VARARGS` flag indicates that this method takes a variable number of arguments at the source code level. A method declared to take a variable number of arguments must be compiled with the `ACC_VARARGS` flag set to 1. All other methods must be compiled with the `ACC_VARARGS` flag set to

0. The `ACC_BRIDGE` method is used to indicate a bridge method generated by the compiler.

A method may be marked with the `ACC_SYNTHETIC` flag to indicate that it was generated by the compiler and does not appear in the source code.

Methods of classes may set any of the flags in Table 4.5. However, a specific method of a class may have at most one of its `ACC_PRIVATE`, `ACC_PROTECTED`, and `ACC_PUBLIC` flags set (§2.7.4). If such a method has its `ACC_ABSTRACT` flag set it must not have any of its `ACC_FINAL`, `ACC_NATIVE`, `ACC_PRIVATE`, `ACC_STATIC`, `ACC_STRICT`, or `ACC_SYNCHRONIZED` flags set (§2.13.3.2).

All interface methods must have their `ACC_ABSTRACT` and `ACC_PUBLIC` flags set; they may have their `ACC_VARARGS`, `ACC_BRIDGE` and `ACC_SYNTHETIC` flags set and must not have any of the other flags in Table 4.5 set (§2.13.3.2).

A specific instance initialization method (§3.9) may have at most one of its `ACC_PRIVATE`, `ACC_PROTECTED`, and `ACC_PUBLIC` flags set and may also have its `ACC_STRICT`, `ACC_VARARGS`, and `ACC_SYNTHETIC` flags set, but must not have any of the other flags in Table 4.5 set.

Class and interface initialization methods (§3.9) are called implicitly by the Java virtual machine; the value of their `access_flags` item is ignored except for the settings of the `ACC_STRICT` flag.

All bits of the `access_flags` item not assigned in Table 4.5 are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

#### `name_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing either one of the special method names (§3.9), `<init>` or `<clinit>`, or a valid unqualified name (§4.3.2) denoting a method.

#### descriptor\_index

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing a valid method descriptor (§4.4.3).

#### attributes\_count

The value of the `attributes_count` item indicates the number of additional attributes (§4.8) of this method.

#### attributes[]

Each value of the `attributes` table must be an attribute structure (§4.8). A method can have any number of optional attributes associated with it.

The only attributes defined by this specification as appearing in the `attributes` table of a `method_info` structure are the `Code` (§4.8.3), `Exceptions` (§4.8.4), `Synthetic` (§4.8.7), `Signature` (§4.8.8) and `Deprecated` (§4.8.13) attributes.

A Java virtual machine implementation must recognize and correctly read `Code` (§4.8.3) and `Exceptions` (§4.8.4) attributes found in the `attributes` table of a `method_info` structure. If a Java virtual machine recognizes class files whose major version is 49.0 or above, it must recognize and correctly read `Signature` (§4.8.8) attributes found in the `attributes` table of a `method_info` structure. A Java virtual machine implementation is required to silently ignore any or all other attributes in the `attributes` table of a `method_info` structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.8.1).

## 4.8 Attributes

Attributes are used in the `ClassFile` (§4.2), `field_info` (§4.6), `method_info` (§4.7), `Code_attribute` (§4.8.3) structures of the class file format. All attributes have the following general format:



```

attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}

```

For all attributes, the `attribute_name_index` must be a valid unsigned 16-bit index into the constant pool of the class. The `constant_pool` entry at `attribute_name_index` must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the name of the attribute. The value of the `attribute_length` item indicates the length of the subsequent information in bytes. The length does not include the initial six bytes that contain the `attribute_name_index` and `attribute_length` items.

Certain attributes are predefined as part of the class file specification. The predefined attributes are the `SourceFile` (§4.8.9), `ConstantValue` (§4.8.2), `Code` (§4.8.3), `Exceptions` (§4.8.4), `InnerClasses` (§4.8.5), `EnclosingMethod` (§4.8.6), `Synthetic` (§4.8.7), `Signature` (§4.8.8), `LineNumberTable` (§4.8.10), `LocalVariableTable` and `Deprecated` (§4.8.13) attributes. Within the context of their use in this specification, that is, in the attributes tables of the class file structures in which they appear, the names of these predefined attributes are reserved.

Of the predefined attributes, the `Code`, `ConstantValue` and `Exceptions` attributes must be recognized and correctly read by a class file reader for correct interpretation of the class file by a Java virtual machine implementation. The `Signature` attribute must be recognized and correctly interpreted by any Java virtual machine implementation that recognizes class files whose major version is 49.0 or above. The `InnerClasses`, `EnclosingMethod` and `Synthetic` attributes must be recognized and correctly read by a class file reader in order to properly implement the Java and Java 2 platform class libraries (§3.12). Use of the remaining predefined attributes is optional; a class file reader may use the information they contain, or otherwise must silently ignore those attributes.

#### 4.8.1 Defining and Naming New Attributes

Compilers are permitted to define and emit class files containing new attributes in the attributes tables of class file structures. Java virtual machine implementations are permitted to recognize and use new attributes found in the attributes tables of class file structures. However, any attribute not defined as part of this Java virtual machine specification must not affect the semantics of class or interface

types. Java virtual machine implementations are required to silently ignore attributes they do not recognize.

For instance, defining a new attribute to support vendor-specific debugging is permitted. Because Java virtual machine implementations are required to ignore attributes they do not recognize, `class` files intended for that particular Java virtual machine implementation will be usable by other implementations even if those implementations cannot make use of the additional debugging information that the `class` files contain.

Java virtual machine implementations are specifically prohibited from throwing an exception or otherwise refusing to use `class` files simply because of the presence of some new attribute. Of course, tools operating on `class` files may not run correctly if given `class` files that do not contain all the attributes they require.

Two attributes that are intended to be distinct, but that happen to use the same attribute name and are of the same length, will conflict on implementations that recognize either attribute. Attributes defined other than by Sun must have names chosen according to the package naming convention defined by *The Java Language Specification*. For instance, a new attribute defined by Netscape might have the name "`com.Netscape.new-attribute`".<sup>3</sup>

Sun may define additional attributes in future versions of this `class` file specification.

#### 4.8.2 The ConstantValue Attribute

The `ConstantValue` attribute is a fixed-length attribute used in the `attributes` table of the `field_info` (§4.6) structures. A `ConstantValue` attribute represents the value of a constant field. There can be no more than one `ConstantValue` attribute in the `attributes` table of a given `field_info` structure. If the field is static (that is, the `ACC_STATIC` bit (Table 4.4) in the `flags` item of the `field_info` structure is set) then the constant field represented by the `field_info` structure is assigned the value referenced by its `ConstantValue` attribute as part of the initialization of the class or interface declaring the constant field (§2.17.4). This occurs immediately prior to the invocation of the class or interface initialization method (§3.9) of that class or interface.

If a `field_info` structure representing a non-static field has a `ConstantValue` attribute, then that attribute must silently be ignored. Every Java virtual machine implementation must recognize `ConstantValue` attributes.

---

<sup>3</sup> The first edition of *The Java Language Specification* required that "`com`" be in uppercase in this example. The second edition reversed that convention and uses lowercase.

The ConstantValue attribute has the following format:

```
ConstantValue_attribute {  
    u2 attribute_name_index;  
    u4 attribute_length;  
    u2 constantvalue_index;  
}
```

The items of the ConstantValue\_attribute structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "ConstantValue".

`attribute_length`

The value of the `attribute_length` item of a `ConstantValue_attribute` structure must be 2.

`constantvalue_index`

The value of the `constantvalue_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index gives the constant value represented by this attribute. The `constant_pool` entry must be of a type appropriate to the field, as shown by Table 4.6.

**Table 4.6** Constant value attribute types

Field Type	Entry Type
long	CONSTANT_Long
float	CONSTANT_Float
double	CONSTANT_Double
int, short, char, byte, boolean	CONSTANT_Integer
String	CONSTANT_String

### 4.8.3 The Code Attribute

The Code attribute is a variable-length attribute used in the attributes table of `method_info` structures. A Code attribute contains the Java virtual machine instructions and auxiliary information for a single method, instance initialization method (§3.9), or class or interface initialization method (§3.9). Every Java virtual machine implementation must recognize Code attributes. If the method is either native or abstract, its `method_info` structure must not have a Code attribute. Otherwise, its `method_info` structure must have exactly one Code attribute.

The Code attribute has the following format:

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

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

**attribute\_name\_index**

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "Code".

**attribute\_length**

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

**max\_stack**

The value of the `max_stack` item gives the maximum depth (§3.6.2) of the operand stack of this method at any point during execution of the method.

**max\_locals**

The value of the `max_locals` item gives the number of local variables in the local variable array allocated upon invocation of this method, including the local variables used to pass parameters to the method on its invocation.

The greatest local variable index for a value of type `long` or `double` is `max_locals-2`. The greatest local variable index for a value of any other type is `max_locals-1`.

**code\_length**

The value of the `code_length` item gives the number of bytes in the code array for this method. The value of `code_length` must be greater than zero; the code array must not be empty.

**code[]**

The code array gives the actual bytes of Java virtual machine code that implement the method.

When the code array is read into memory on a byte-addressable machine, if the first byte of the array is aligned on a 4-byte boundary, the *tableswitch* and *lookupswitch* 32-bit offsets will be 4-byte aligned. (Refer to the descriptions of those instructions for more information on the consequences of code array alignment.)

The detailed constraints on the contents of the code array are extensive and are given in a separate section (§4.10).

### `exception_table_length`

The value of the `exception_table_length` item gives the number of entries in the `exception_table` table.

### `exception_table[]`

Each entry in the `exception_table` array describes one exception handler in the code array. The order of the handlers in the `exception_table` array is significant. See Section 3.10 for more details.

Each `exception_table` entry contains the following four items:

#### `start_pc, end_pc`

The values of the two items `start_pc` and `end_pc` indicate the ranges in the code array at which the exception handler is active. The value of `start_pc` must be a valid index into the code array of the opcode of an instruction. The value of `end_pc` either must be a valid index into the code array of the opcode of an instruction or must be equal to `code_length`, the length of the code array. The value of `start_pc` must be less than the value of `end_pc`.

The `start_pc` is inclusive and `end_pc` is exclusive; that is, the exception handler must be active while the program counter is within the interval `[start_pc, end_pc)`.<sup>4</sup>

#### `handler_pc`

The value of the `handler_pc` item indicates the start of the exception handler. The value of the item must be a valid index into the code array and must be the index of the opcode of an instruction.

#### `catch_type`

If the value of the `catch_type` item is nonzero, it must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.5.1) structure representing a

---

<sup>4</sup> The fact that `end_pc` is exclusive is a historical mistake in the design of the Java virtual machines.

class of exceptions that this exception handler is designated to catch. The exception handler will be called only if the thrown exception is an instance of the given class or one of its subclasses.

If the value of the `catch_type` item is zero, this exception handler is called for all exceptions. This is used to implement `finally` (see Section 7.13, “Compiling `finally`”).

#### `attributes_count`

The value of the `attributes_count` item indicates the number of attributes of the `Code` attribute.

#### `attributes[]`

Each value of the `attributes` table must be an attribute structure (§4.8). A `Code` attribute can have any number of optional attributes associated with it.

Currently, the `LineNumberTable` (§4.8.10) and `LocalVariableTable` (§4.8.11), attributes which contain debugging information, are defined and used with the `Code` attribute.

A Java virtual machine implementation is permitted to silently ignore any or all attributes in the `attributes` table of a `Code` attribute. Attributes not defined in this specification are not allowed to affect the semantics of the `class` file, but only to provide additional descriptive information (§4.8.1).

### 4.8.4 The Exceptions Attribute

The `Exceptions` attribute is a variable-length attribute used in the `attributes` table of a `method_info` (§4.7) structure. The `Exceptions` attribute indicates which checked exceptions a method may throw. There may be at most one `Exceptions` attribute in each `method_info` structure.

The `Exceptions` attribute has the following format:

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

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

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be the `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "Exceptions".

`attribute_length`

The value of the `attribute_length` item indicates the attribute length, excluding the initial six bytes.

`number_of_exceptions`

The value of the `number_of_exceptions` item indicates the number of entries in the `exception_index_table`.

`exception_index_table[]`

Each value in the `exception_index_table` array must be a valid index into the `constant_pool` table. The `constant_pool` entry referenced by each table item must be a `CONSTANT_Class_info` (§4.5.1) structure representing a class type that this method is declared to throw.

A method should throw an exception only if at least one of the following three criteria is met:

- The exception is an instance of `RuntimeException` or one of its subclasses.
- The exception is an instance of `Error` or one of its subclasses.
- The exception is an instance of one of the exception classes specified in the `exception_index_table` just described, or one of their subclasses.

These requirements are not enforced in the Java virtual machine; they are enforced only at compile time.



#### 4.8.5 The InnerClasses Attribute

The InnerClasses attribute<sup>5</sup> is a variable-length attribute in the attributes table of the ClassFile (§4.2) structure. If the constant pool of a class or interface *C* contains a CONSTANT\_Class\_info entry which represents a class or interface that is not a member of a package, then *C*'s ClassFile structure must have exactly one InnerClasses attribute in its attributes table.

The InnerClasses attribute has the following format:

```

InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    { u2 inner_class_info_index;
      u2 outer_class_info_index;
      u2 inner_name_index;
      u2 inner_class_access_flags;
    } classes[number_of_classes];
}

```

The items of the InnerClasses\_attribute structure are as follows:

##### attribute\_name\_index

The value of the attribute\_name\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info (§4.5.7) structure representing the string "InnerClasses".

##### attribute\_length

The value of the attribute\_length item indicates the length of the attribute, excluding the initial six bytes.

##### number\_of\_classes

The value of the number\_of\_classes item indicates the number of entries in the classes array.

---

<sup>5</sup> The InnerClasses attribute was introduced in JDK release 1.1 to support nested classes and interfaces.

`classes[]`

Every `CONSTANT_Class_info` entry in the `constant_pool` table which represents a class or interface `C` that is not a package member must have exactly one corresponding entry in the `classes` array.

If a class has members that are classes or interfaces, its `constant_pool` table (and hence its `InnerClasses` attribute) must refer to each such member, even if that member is not otherwise mentioned by the class. These rules imply that a nested class or interface member will have `InnerClasses` information for each enclosing class and for each immediate member.

Each `classes` array entry contains the following four items:

`inner_class_info_index`

The value of the `inner_class_info_index` item must be zero or a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.5.1) structure representing `C`. The remaining items in the `classes` array entry give information about `C`.

`outer_class_info_index`

If `C` is not a member, the value of the `outer_class_info_index` item must be zero. Otherwise, the value of the `outer_class_info_index` item must be a valid index into the `constant_pool` table, and the entry at that index must be a `CONSTANT_Class_info` (§4.5.1) structure representing the class or interface of which `C` is a member.

`inner_name_index`

If `C` is anonymous, the value of the `inner_name_index` item must be zero. Otherwise, the value of the `inner_name_index` item must be a valid index into the `constant_pool` table, and the entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure that represents the original simple name of `C`, as given in the source code from which this class file was compiled.

`inner_class_access_flags`

The value of the `inner_class_access_flags` item is a mask of flags used to denote access permissions to and properties of class or interface `C` as declared in the source code from which this `class` file was compiled. It is used by compilers to recover the original information when source code is not available. The flags are shown in Table 4.7.

**Table 4.7 Nested class access and property flags**

Flag Name	Value	Meaning
ACC_PUBLIC	0x0001	Marked or implicitly public in source.
ACC_PRIVATE	0x0002	Marked private in source.
ACC_PROTECTED	0x0004	Marked protected in source.
ACC_STATIC	0x0008	Marked or implicitly static in source.
ACC_FINAL	0x0010	Marked final in source.
ACC_INTERFACE	0x0200	Was an interface in source.
ACC_ABSTRACT	0x0400	Marked or implicitly abstract in source.
ACC_SYNTHETIC	0x1000	Declared synthetic; Not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.

All bits of the `inner_class_access_flags` item not assigned in Table 4.7 are reserved for future use. They should be set to zero in generated `class` files and should be ignored by Java virtual machine implementations.

The Java virtual machine does not currently check the consistency of the `InnerClasses` attribute with any `class` file actually representing a class or interface referenced by the attribute.

### 4.8.6 The EnclosingMethod Attribute

The `EnclosingMethod` attribute is an optional fixed-length attribute in the `attributes` table of the `ClassFile` (§4.2) structure. A class must have an `EnclosingMethod` attribute if and only if it is a local class or an anonymous class. A class may have no more than one `EnclosingMethod` attribute.

The `EnclosingMethod` attribute has the following format:

```
EnclosingMethod_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 class_index
    u2 method_index;
}
```

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

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "EnclosingMethod".

`attribute_length`

The value of the `attribute_length` item is four.

`class_index`

The value of the `class_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` (§4.5.1) structure representing the innermost class that encloses the declaration of the current class.

`method_index`

If the current class is not immediately enclosed by a method or constructor, then the value of the `method_index` item must be zero. Otherwise, the value of the `method_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_NameAndType_info` (§4.5.6) structure representing a the name and type of a method in

the class referenced by the `class_index` attribute above. It is the responsibility of the Java compiler to ensure that the method identified via the `method_index` is indeed the closest lexically enclosing method of the class that contains this `EnclosingMethod` attribute.

#### 4.8.7 The Synthetic Attribute

The `Synthetic` attribute<sup>6</sup> is a fixed-length attribute in the `attributes` table of `ClassFile` (§4.2), `field_info` (§4.6), and `method_info` (§4.7) structures. A class member that does not appear in the source code must be marked using a `Synthetic` attribute, or else it must have its `ACC_SYNTHETIC` bit set. The only exceptions to this requirement are for default constructors and the class initialization method.

The `Synthetic` attribute has the following format:

```
Synthetic_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

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

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "Synthetic".

`attribute_length`

The value of the `attribute_length` item is zero.

---

<sup>6</sup> The `Synthetic` attribute was introduced in JDK release 1.1 to support nested classes and interfaces.

### 4.8.8 The Signature Attribute

The Signature attribute is an optional fixed-length attribute in the attributes table of the `ClassFile` (§4.2), `field_info`(§4.6) and `method_info` (§4.7) structures.

The Signature attribute has the following format:

```
Signature_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 signature_index;
}
```

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

#### `attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info`(§4.5.7) structure representing the string "Signature".

#### `attribute_length`

The value of the `attribute_length` item of a `Signature_attribute` structure must be 2.

#### `signature_index`

The value of the `signature_index` item must be a valid index into the `constant_pool` table. The constant pool entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing either a class signature, if this signature attribute is an attribute of a `ClassFile` structure, a method type signature, if this signature is an attribute of a `method_info` structure, or a field type signature otherwise.

### 4.8.9 The SourceFile Attribute

The `SourceFile` attribute is an optional fixed-length attribute in the attributes table of the `ClassFile` (§4.2) structure. There can be no more than one `SourceFile` attribute in the attributes table of a given `ClassFile` structure.

The `SourceFile` attribute has the following format:

```

SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}

```

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

#### `attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "SourceFile".

#### `attribute_length`

The value of the `attribute_length` item of a `SourceFile_attribute` structure must be 2.

#### `sourcefile_index`

The value of the `sourcefile_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing a string.

The string referenced by the `sourcefile_index` item will be interpreted as indicating the name of the source file from which this class file was compiled. It will not be interpreted as indicating the name of a directory containing the file or an absolute path name for the file; such platform-specific additional information must be supplied by the runtime interpreter or development tool at the time the file name is actually used.

### 4.8.10 The `SourceDebugExtension` Attribute

The `SourceDebugExtension` attribute is an optional attribute in the attributes table of the `ClassFile` (§4.2) structure. There can be no more than one `SourceDebugExtension` attribute in the attributes table of a given `ClassFile` structure.

The `SourceDebugExtension` attribute has the following format:

```
SourceDebugExtension_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 debug_extension[attribute_length];
}
```

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

#### `attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "SourceDebugExtension".

#### `attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes. The value of the `attribute_length` item is thus the number of bytes in the `debug_extension[]` item.

#### `debug_extension[]`

The `debug_extension` array holds a string, which must be in UTF-8 format. There is no terminating zero byte. The string in the `debug_extension` item will be interpreted as extended debugging information. The content of this string has no semantic effect on the Java Virtual Machine.

### 4.8.11 The `LineNumberTable` Attribute

The `LineNumberTable` attribute is an optional variable-length attribute in the attributes table of a `Code` (§4.8.3) attribute. It may be used by debuggers to determine which part of the Java virtual machine code array corresponds to a given line number in the original source file. If `LineNumberTable` attributes are present in the attributes table of a given `Code` attribute, then they may appear in any order. Furthermore, multiple `LineNumberTable` attributes may together represent a given line of a source file; that is, `LineNumberTable` attributes need not be one-to-one with source lines.



The `LineNumberTable` attribute has the following format:

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    { u2 start_pc;
      u2 line_number;
    } line_number_table[line_number_table_length];
}
```

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

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "LineNumberTable".

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

`line_number_table_length`

The value of the `line_number_table_length` item indicates the number of entries in the `line_number_table` array.

`line_number_table[]`

Each entry in the `line_number_table` array indicates that the line number in the original source file changes at a given point in the code array. Each `line_number_table` entry must contain the following two items:

`start_pc`

The value of the `start_pc` item must indicate the index into the code array at which the code for a new line in the original source file begins. The value of `start_pc` must be less than the value of the `code_length` item of the `Code` attribute of which this `LineNumberTable` is an attribute.

`line_number`

The value of the `line_number` item must give the corresponding line number in the original source file.

#### 4.8.12 The `LocalVariableTable` Attribute

The `LocalVariableTable` attribute is an optional variable-length attribute of a `Code` (§4.8.3) attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method. If `LocalVariableTable` attributes are present in the `attributes` table of a given `Code` attribute, then they may appear in any order. There may be no more than one `LocalVariableTable` attribute per local variable in the `Code` attribute.

The `LocalVariableTable` attribute has the following format:

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    { u2 start_pc;
      u2 length;
      u2 name_index;
      u2 descriptor_index;
      u2 index;
    } local_variable_table[
        local_variable_table_length];
}
```

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

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "`LocalVariableTable`".

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

`local_variable_table_length`

The value of the `local_variable_table_length` item indicates the number of entries in the `local_variable_table` array.

`local_variable_table[]`

Each entry in the `local_variable_table` array indicates a range of code array offsets within which a local variable has a value. It also indicates the index into the local variable array of the current frame at which that local variable can be found. Each entry must contain the following five items:

`start_pc, length`

The given local variable must have a value at indices into the code array in the interval  $[start\_pc, start\_pc + length)$ , that is, between `start_pc` and `start_pc + length` exclusive. The value of `start_pc` must be a valid index into the code array of this Code attribute and must be the index of the opcode of an instruction. The value of `start_pc + length` must either be a valid index into the code array of this Code attribute and be the index of the opcode of an instruction, or it must be the first index beyond the end of that code array.

`name_index, descriptor_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must contain a `CONSTANT_Utf8_info` (§4.5.7) structure representing a valid unqualified name (§4.3.2) denoting a local variable.

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must contain a `CONSTANT_Utf8_info` (§4.5.7) structure. That `CONSTANT_Utf8_info` structure must represent a field descriptor (§4.4.2) encoding the type of a local variable in the source program.

`index`

The given local variable must be at `index` in the local variable array of the current frame. If the local variable at

index is of type `double` or `long`, it occupies both `index` and `index+1`.

#### 4.8.13 The `LocalVariableTypeTable` Attribute

The `LocalVariableTypeTable` attribute is an optional variable-length attribute of a `Code` (§4.8.3) attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method. If `LocalVariableTypeTable` attributes are present in the `attributes` table of a given `Code` attribute, then they may appear in any order. There may be no more than one `LocalVariableTypeTable` attribute per local variable in the `Code` attribute.

The `LocalVariableTypeTable` attribute differs from the `LocalVariableTable` attribute in that it provides signature information rather than descriptor information. This difference is only significant for variables whose type is a generic reference type. Such variables will appear in both tables, while variables of other types will appear only in `LocalVariableTable`.

The `LocalVariableTypeTable` attribute has the following format:

```
LocalVariableTypeTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_type_table_length;
    { u2 start_pc;
      u2 length;
      u2 name_index;
      u2 signature_index;
      u2 index;
    } local_variable_type_table[
        local_variable_type_table_length];
}
```

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

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table `a121`. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) `a122` structure representing the string "`LocalVariableTypeTable`" `a123`.

**attribute\_length**

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

**local\_variable\_table\_length**

The value of the `local_variable_table_length` item indicates the number of entries in the `local_variable_table` array.

**local\_variable\_table[]**

Each entry in the `local_variable_table` array indicates a range of code array offsets within which a local variable has a value. It also indicates the index into the local variable array of the current frame at which that local variable can be found. Each entry must contain the following five items:

**start\_pc, length**

The given local variable must have a value at indices into the code array in the interval  $[start\_pc, start\_pc + length)$ , that is, between `start_pc` and `start_pc + length` exclusive. The value of `start_pc` must be a valid index into the code array of this Code attribute and must be the index of the opcode of an instruction<sup>124</sup>. The value of `start_pc + length` must either be a valid index into the code array of this Code attribute and be the index of the opcode of an instruction, or it must be the first index beyond the end of that code array<sup>125</sup>.

**name\_index, signature\_index**

The value of the `name_index` item must be a valid index into the `constant_pool` table<sup>127</sup>. The `constant_pool` entry at that index must contain a `CONSTANT_Utf8_info` (§4.5.7) structure<sup>128</sup> representing a valid unqualified name (§4.3.2) denoting a local variable<sup>128</sup>. Careful here - do we want any restrictions at all?

The value of the `signature_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must contain a `CONSTANT_Utf8_info` (§4.5.7) structure representing a field type signature (§4.4.4) encoding the type of a local variable in the source program.

**index**

The given local variable must be at `index` in the local variable array of the current frame. If the local variable at `index` is of type `double` or `long`, it occupies both `index` and `index+1`.

**4.8.14 The Deprecated Attribute**

The Deprecated attribute<sup>7</sup> is an optional fixed-length attribute in the attributes table of `ClassFile` (§4.2), `field_info` (§4.6), and `method_info` (§4.7) structures. A class, interface, method, or field may be marked using a Deprecated attribute to indicate that the class, interface, method, or field has been superseded. A runtime interpreter or tool that reads the class file format, such as a compiler, can use this marking to advise the user that a superseded class, interface, method, or field is being referred to. The presence of a Deprecated attribute does not alter the semantics of a class or interface.

The Deprecated attribute has the following format:

```
Deprecated_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

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

**attribute\_name\_index**

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§4.5.7) structure representing the string "Deprecated".

**attribute\_length**

The value of the `attribute_length` item is zero.

---

<sup>7</sup> The Deprecated attribute was introduced in JDK release 1.1 to support the `@deprecated` tag in documentation comments.

#### 4.8.15 The RuntimeVisibleAnnotations attribute

The RuntimeVisibleAnnotations attribute is a variable length attribute in the attributes table of the ClassFile, field\_info, and method\_info structures. The RuntimeVisibleAnnotations attribute records runtime-visible Java programming language annotations on the corresponding class, method, or field. Each ClassFile, field\_info, and method\_info structure may contain at most one RuntimeVisibleAnnotations attribute, which records all the runtime-visible Java programming language annotations on the corresponding program element. The JVM must make these annotations available so they can be returned by the appropriate reflective APIs.

The RuntimeVisibleAnnotations attribute has the following format:

```
RuntimeVisibleAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_annotations;
    annotation annotations[num_annotations];
}
```

The items of the RuntimeVisibleAnnotations structure are as follows:

##### attribute\_name\_index

The value of the attribute\_name\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info structure representing the string "RuntimeVisibleAnnotations".

##### attribute\_length

The value of the attribute\_length item indicates the length of the attribute, excluding the initial six bytes. The value of the attribute\_length item is thus dependent on the number of runtime-visible annotations represented by the structure, and their values.

##### num\_annotations

The value of the num\_annotations item gives the number of runtime-visible annotations represented by the structure. Note that a maximum of 65535 runtime-visible Java programming language annotations may be directly attached to a program element.

##### annotations

Each value of the `annotations` table represents a single runtime-visible annotation on a program element.

The annotation structure has the following format:

```

annotation {
    u2    type_index;
    u2    num_element_value_pairs;
    {    u2    element_name_index;
    element_value value;
    }    element_value_pairs[num_element_value_pairs]
}

```

The items of the annotation structure are as follows:

`type_index`

The value of the `type_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing a field descriptor representing the annotation type corresponding to the annotation represented by this annotation structure.

`num_element_value_pairs`

The value of the `num_element_value_pairs` item gives the number of element-value pairs of the annotation represented by this annotation structure. Note that a maximum of 65535 element-value pairs may be contained in a single annotation.

`element_value_pairs`

Each value of the `element_value_pairs` table represents a single element-value pair in the annotation represented by this annotation structure. Each `element_value_pairs` entry contains the following two items:

`element_name_index`

The value of the `element_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the name of the annotation type element represented by this `element_value_pairs` entry.

`value`

The value of the `value` item represents the value of the element-value pair represented by this `element_value_pairs` entry.

#### 4.8.15.1 The `element_value` structure



The `element_value` structure is a discriminated union representing the value of an element-value pair. It is used to represent element values in all attributes that describe annotations (`RuntimeVisibleAnnotations`, `RuntimeInvisibleAnnotations`, `RuntimeVisibleParameterAnnotations`, and `RuntimeInvisibleParameterAnnotations`).

The `element_value` structure has the following format:

```

element_value {
    u1 tag;
    union {
        u2  const_value_index;
    {
        u2 type_name_index;
        u2 const_name_index;
    } enum_const_value;
        u2  class_info_index;
        annotation annotation_value;
    {
        u2  num_values;
        element_value values[num_values];
    } array_value;
    } value;
    }

```

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

**tag**

The `tag` item indicates the type of this annotation element-value pair. The letters 'B', 'C', 'D', 'F', 'I', 'J', 'S', and 'Z' indicate a primitive type. These letters are interpreted as `BaseType` characters (§Table 4.2). The other legal values for `tag` are listed with their interpretations in this table:

**Table 4.8**

tag value	Element Type
s	String
e	enum constant

tag value	Element Type
c	class
@	annotation type
[	array

### value

The `value` item represents the value of this annotation element. This item is a union. The tag item, above, determines which item of the union is to be used:

#### `const_value_index`

The `const_value_index` item is used if the tag item is one of 'B', 'C', 'D', 'F', 'I', 'J', 'S', 'Z', or 's'. The value of the `const_value_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be of the correct entry type for the field type designated by the tag item, as specified in Table 4.8.

#### `enum_const_value`

The `enum_const_value` item is used if the tag item is 'e'. The `enum_const_value` item consists of the following two items:

#### `type_name_index`

The value of the `type_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the binary name (JLS 13.1) of the type of the enum constant represented by this `element_value` structure.

#### `const_name_index`

The value of the `const_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the simple name of the enum constant represented by this `element_value` structure.

#### `class_info_index`

The `class_info_index` item is used if the tag item is 'c'. The `class_info_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the return descriptor (§4.4.3) of the type that is reified by the class represented by this `element_value` structure (e.g., 'V' for `Void.class`, 'Ljava/lang/Object;' for `Object`, etc.)

**annotation\_value**

The `annotation_value` item is used if the tag item is '@'. The `element_value` structure represents a "nested" annotation.

**array\_value**

The `array_value` item is used if the tag item is '['. The `array_value` item consists of the following two items:

**num\_values**

The value of the `num_values` item gives the number of elements in the array-typed value represented by this `element_value` structure. Note that a maximum of 65535 elements are permitted in an array-typed element value.

**values**

Each value of the `values` table gives the value of an element of the array-typed value represented by this `element_value` structure.

**4.8.16 The RuntimeInvisibleAnnotations attribute**

The `RuntimeInvisibleAnnotations` attribute is similar to the `RuntimeVisibleAnnotations` attribute, except that the annotations represented by a `RuntimeInvisibleAnnotations` attribute must not be made available for return by reflective APIs, unless the the JVM has been instructed to retain these annotations via some implementation-specific mechanism such as a command line flag. In the absence of such instructions, the JVM ignores this attribute.

The `RuntimeInvisibleAnnotations` attribute is a variable length attribute in the attributes table of the `ClassFile`, `field_info`, and `method_info` structures. The `RuntimeInvisibleAnnotations` attribute records runtime-invisible Java programming language annotations on the corresponding class, method, or field. Each `ClassFile`, `field_info`, and `method_info` structure may contain at most one `RuntimeInvisibleAnnotations` attribute, which records all the runtime-invisible Java programming language annotations on the corresponding program element.

The `RuntimeInvisibleAnnotations` attribute has the following format:

```
RuntimeInvisibleAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_annotations;
    annotation annotations[num_annotations];
}
```

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

#### `attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "RuntimeInvisibleAnnotations".

#### `attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes. The value of the `attribute_length` item is thus dependent on the number of runtime-invisible annotations represented by the structure, and their values.

#### `num_annotations`

The value of the `num_annotations` item gives the number of runtime-invisible annotations represented by the structure. Note that a maximum of 65535 runtime-invisible Java programming language annotations may be directly attached to a program element.

#### `annotations`

Each value of the `annotations` table represents a single runtime-invisible annotation on a program element.

### 4.8.17 The `RuntimeVisibleParameterAnnotations` attribute

The `RuntimeVisibleParameterAnnotations` attribute is a variable length attribute in the attributes table of the `method_info` structure. The `RuntimeVisibleParameterAnnotations` attribute records runtime-visible Java programming language annotations on the parameters of the corresponding method. Each `method_info` structure may contain at most one `RuntimeVisibleParameterAnnotations` attribute, which records all the runtime-visible Java programming language annotations on the parameters of the corresponding method. The JVM must

make these annotations available so they can be returned by the appropriate reflective APIs.

The `RuntimeVisibleParameterAnnotations` attribute has the following format:

```
RuntimeVisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {
        u2 num_annotations;
        annotation annotations[num_annotations];
    } parameter_annotations[num_parameters];
}
```

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

#### `attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "RuntimeVisibleParameterAnnotations".

#### `attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes. The value of the `attribute_length` item is thus dependent on the number of parameters, the number of runtime-visible annotations on each parameter, and their values.

#### `num_parameters`

The value of the `num_parameters` item gives the number of parameters of the method represented by the `method_info` structure on which the annotation occurs. (This duplicates information that could be extracted from the method descriptor.)

#### `parameter_annotations`

Each value of the `parameter_annotations` table represents all of the runtime-visible annotations on a single parameter. The sequence of values in the table corresponds to the sequence of parameters in the method signature. Each `parameter_annotations` entry contains the following two items:

#### `num_annotations`

The value of the `num_annotations` item indicates the number of runtime-visible annotations on the parameter corresponding to the sequence number of this `parameter_annotations` element.

#### `annotations`

Each value of the `annotations` table represents a single runtime-visible annotation on the parameter corresponding to the sequence number of this `parameter_annotations` element.

### **4.8.18 The `RuntimeInvisibleParameterAnnotations` attribute**

The `RuntimeInvisibleParameterAnnotations` attribute is similar to the `RuntimeVisibleParameterAnnotations` attribute, except that the annotations represented by a `RuntimeInvisibleParameterAnnotations` attribute must not be made available for return by reflective APIs, unless the JVM has specifically been instructed to retain these annotations via some implementation-specific mechanism such as a command line flag. In the absence of such instructions, the JVM ignores this attribute.

The `RuntimeInvisibleParameterAnnotations` attribute is a variable length attribute in the attributes table of the `method_info` structure. The `RuntimeInvisibleParameterAnnotations` attribute records runtime-invisible Java programming language annotations on the parameters of the corresponding method. Each `method_info` structure may contain at most one `RuntimeInvisibleParameterAnnotations` attribute, which records all the runtime-invisible Java programming language annotations on the parameters of the corresponding method.

The `RuntimeInvisibleParameterAnnotations` attribute has the following format:

```

RuntimeInvisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {
        u2 num_annotations;
        annotation annotations[num_annotations];
    } parameter_annotations[num_parameters];
}

```

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

#### `attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "RuntimeInvisibleParameterAnnotations".

#### `attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes. The value of the `attribute_length` item is thus dependent on the number of parameters, the number of runtime-invisible annotations on each parameter, and their values.

#### `num_parameters`

The value of the `num_parameters` item gives the number of parameters of the method represented by the `method_info` structure on which the annotation occurs. (This duplicates information that could be extracted from the method descriptor.)

#### `parameter_annotations`

Each value of the `parameter_annotations` table represents all of the runtime-invisible annotations on a single parameter. The sequence of values in the table corresponds to the sequence of parameters in the method signature. Each `parameter_annotations` entry contains the following two items:

#### `num_annotations`

The value of the `num_annotations` item indicates the number of runtime-invisible annotations on the parameter corresponding to the sequence number of this `parameter_annotations` element.

#### `annotations`

Each value of the annotations table represents a single runtime-invisible annotation on the parameter corresponding to the sequence number of this `parameter_annotations` element.

#### 4.8.19 The AnnotationDefault attribute

The AnnotationDefault attribute is a variable length attribute in the attributes table of certain `method_info` structures, namely those representing elements of annotation types. The AnnotationDefault attribute records the default value for the element represented by the `method_info` structure. Each `method_info` structures representing an element of an annotation types may contain at most one AnnotationDefault attribute. The JVM must make this default value available so it can be applied by appropriate reflective APIs.

The AnnotationDefault attribute has the following format:

```
AnnotationDefault_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    element_value default_value;
}
```

The items of the AnnotationDefault structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "AnnotationDefault".

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes. The value of the `attribute_length` item is thus dependent on the default value.

`default_value`

The `default_value` item represents the default value of the annotation type element whose default value is represented by this AnnotationDefault attribute.



## 4.9 Format Checking

When a prospective `class` file is loaded (§2.17.2) by the Java virtual machine, the Java virtual machine first ensures that the file has the basic format of a `class` file. This process is known as *format checking*. The first four bytes must contain the right magic number. All recognized attributes must be of the proper length. The `class` file must not be truncated or have extra bytes at the end. The constant pool must not contain any superficially unrecognizable information.

---

This check for basic `class` file integrity is necessary for any interpretation of the `class` file contents.

However, format checking is distinct from verification. Historically, the two have been confused, because both are a form of integrity check.

---

## 4.10 Constraints on Java Virtual Machine Code

The Java virtual machine code for a method, instance initialization method (§3.9), or class or interface initialization method (§3.9) is stored in the code array of the `Code` attribute of a `method_info` structure of a `class` file. This section describes the constraints associated with the contents of the `Code_attribute` structure.

### 4.10.1 Static Constraints

The *static constraints* on a `class` file are those defining the well-formedness of the file. With the exception of the static constraints on the Java virtual machine code of the `class` file, these constraints have been given in the previous section. The static constraints on the Java virtual machine code in a `class` file specify how Java virtual machine instructions must be laid out in the code array and what the operands of individual instructions must be.

The static constraints on the instructions in the code array are as follows:

- The code array must not be empty, so the `code_length` item cannot have the value 0.
- The value of the `code_length` item must be less than 65536.
- The opcode of the first instruction in the code array begins at index 0.
- Only instances of the instructions documented in Section 6.4 may appear in the code array. Instances of instructions using the reserved opcodes (§6.2) or any opcodes not documented in this specification must not appear in the code array.
- For each instruction in the code array except the last, the index of the opcode of the next instruction equals the index of the opcode of the current instruction plus the length of that instruction, including all its operands. The *wide* instruction is treated like any other instruction for these purposes; the opcode specifying the operation that a *wide* instruction is to modify is treated as one of the operands of that *wide* instruction. That opcode must never be directly reachable by the computation.
- The last byte of the last instruction in the code array must be the byte at index `code_length-1`.

The static constraints on the operands of instructions in the code array are as follows:

- The target of each jump and branch instruction (*jsr*, *jsr\_w*, *goto*, *goto\_w*, *ifeq*, *ifne*, *ifle*, *iflt*, *ifge*, *ifgt*, *ifnull*, *ifnonnull*, *if\_icmpeq*, *if\_icmpne*, *if\_icmple*, *if\_icmplt*, *if\_icmpge*, *if\_icmpgt*, *if\_acmpeq*, *if\_acmpne*) must be the opcode of an instruction within this method. The target of a jump or branch instruction must never be the opcode used to specify the operation to be modified by a *wide* instruction; a jump or branch target may be the *wide* instruction itself.
- Each target, including the default, of each *tableswitch* instruction must be the opcode of an instruction within this method. Each *tableswitch* instruction must have a number of entries in its jump table that is consistent with the value of its *low* and *high* jump table operands, and its *low* value must be less than or equal to its *high* value. No target of a *tableswitch* instruction may be the opcode used to specify the operation to be modified by a *wide* instruction; a *tableswitch* target may be a *wide* instruction itself.

- Each target, including the default, of each *lookupswitch* instruction must be the opcode of an instruction within this method. Each *lookupswitch* instruction must have a number of *match-offset* pairs that is consistent with the value of its *npairs* operand. The *match-offset* pairs must be sorted in increasing numerical order by signed *match* value. No target of a *lookupswitch* instruction may be the opcode used to specify the operation to be modified by a *wide* instruction; a *lookupswitch* target may be a *wide* instruction itself.
- The operand of each *ldc* instruction must be a valid index into the `constant_pool` table. The operands of each *ldc\_w* instruction must represent a valid index into the `constant_pool` table. In both cases the constant pool entry referenced by that index must be of type `CONSTANT_Class`, `CONSTANT_Integer`, `CONSTANT_Float`, or `CONSTANT_String` if the class file version number is less than 49.0. If the class file version is 49.0 or above, then the constant pool entry referenced by the entry must be of type `CONSTANT_Class`, `CONSTANT_Integer`, `CONSTANT_Float`, or `CONSTANT_String` or `CONSTANT_Class`.
- The operands of each *ldc2\_w* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Long` or `CONSTANT_Double`. In addition, the subsequent constant pool index must also be a valid index into the constant pool, and the constant pool entry at that index must not be used.
- The operands of each *getfield*, *putfield*, *getstatic*, and *putstatic* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Fieldref`.
- The indexbyte operands of each *invokevirtual*, *invokespecial*, and *invokestatic* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Methodref`.
- Only the *invokespecial* instruction is allowed to invoke an instance initialization method (§3.9). No other method whose name begins with the character '<' ('\u003c') may be called by the method invocation instructions. In particular, the class or interface initialization method specially named `<clinit>` is never called explicitly from Java virtual machine instructions, but only implicitly by the Java virtual machine itself.

- The indexbyte operands of each *invokeinterface* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_InterfaceMethodref`. The value of the *count* operand of each *invokeinterface* instruction must reflect the number of local variables necessary to store the arguments to be passed to the interface method, as implied by the descriptor of the `CONSTANT_NameAndType_info` structure referenced by the `CONSTANT_InterfaceMethodref` constant pool entry. The fourth operand byte of each *invokeinterface* instruction must have the value zero.
- The operands of each *instanceof*, *checkcast*, *new*, and *anewarray* instruction and the indexbyte operands of each *multianewarray* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Class`.
- No *anewarray* instruction may be used to create an array of more than 255 dimensions.
- No *new* instruction may reference a `CONSTANT_Class` `constant_pool` table entry representing an array class. The *new* instruction cannot be used to create an array.
- A *multianewarray* instruction must be used only to create an array of a type that has at least as many dimensions as the value of its *dimensions* operand. That is, while a *multianewarray* instruction is not required to create all of the dimensions of the array type referenced by its indexbyte operands, it must not attempt to create more dimensions than are in the array type. The *dimensions* operand of each *multianewarray* instruction must not be zero.
- The *atype* operand of each *newarray* instruction must take one of the values `T_BOOLEAN` (4), `T_CHAR` (5), `T_FLOAT` (6), `T_DOUBLE` (7), `T_BYTE` (8), `T_SHORT` (9), `T_INT` (10), or `T_LONG` (11).
- The index operand of each *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *iinc*, and *ret* instruction must be a nonnegative integer no greater than `max_locals-1`.
- The implicit index of each *iload* `<n>`, *fload* `<n>`, *aload* `<n>`, *istore* `<n>`, *fstore* `<n>`, and *astore* `<n>` instruction must be no greater than the value of `max_locals-1`.
- The index operand of each *lload*, *dload*, *lstore*, and *dstore* instruction must be no greater than the value of `max_locals-2`.

- The implicit index of each *lload*\_*<n>*, *dload*\_*<n>*, *lstore*\_*<n>*, and *dstore*\_*<n>* instruction must be no greater than the value of `max_locals-2`.
- The indexbyte operands of each *wide* instruction modifying an *lload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *ret*, or *iinc* instruction must represent a nonnegative integer no greater than `max_locals-1`. The indexbyte operands of each *wide* instruction modifying an *lload*, *dload*, *lstore*, or *dstore* instruction must represent a nonnegative integer no greater than `max_locals-2`.

#### 4.10.2 Structural Constraints

The structural constraints on the code array specify constraints on relationships between Java virtual machine instructions. The structural constraints are as follows:

- Each instruction must only be executed with the appropriate type and number of arguments in the operand stack and local variable array, regardless of the execution path that leads to its invocation. An instruction operating on values of type `int` is also permitted to operate on values of type `boolean`, `byte`, `char`, and `short`. (As noted in §3.3.4 and §3.11.1, the Java virtual machine internally converts values of types `boolean`, `byte`, `char`, and `short` to type `int`.)
- If an instruction can be executed along several different execution paths, the operand stack must have the same depth (§3.6.2) prior to the execution of the instruction, regardless of the path taken.
- At no point during execution can the order of the local variable pair holding a value of type `long` or `double` be reversed or the pair split up. At no point can the local variables of such a pair be operated on individually.
- No local variable (or local variable pair, in the case of a value of type `long` or `double`) can be accessed before it is assigned a value.
- At no point during execution can the operand stack grow to a depth (§3.6.2) greater than that implied by the `max_stack` item.
- At no point during execution can more values be popped from the operand stack than it contains.
- Each *invokespecial* instruction must name an instance initialization method (§3.9), a method in the current class, or a method in a superclass of the current class.

- When the instance initialization method (§3.9) is invoked, an uninitialized class instance must be in an appropriate position on the operand stack. An instance initialization method must never be invoked on an initialized class instance.
- When any instance method is invoked or when any instance variable is accessed, the class instance that contains the instance method or instance variable must already be initialized.
- There must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken.
- There must never be an uninitialized class instance on the operand stack or in a local variable when a *jsr* or *jsr\_w* instruction is executed.
- Each instance initialization method (§3.9), except for the instance initialization method derived from the constructor of class `Object`, must call either another instance initialization method of `this` or an instance initialization method of its direct superclass `super` before its instance members are accessed. However, instance fields of `this` that are declared in the current class may be assigned before calling any instance initialization method.
- The arguments to each method invocation must be method invocation compatible (§2.6.8) with the method descriptor (§4.4.3).
- The type of every class instance that is the target of a method invocation instruction must be assignment compatible (§2.6.7) with the class or interface type specified in the instruction. In addition, the type of the target of an *invokespecial* instruction must be assignment compatible with the current class, unless an instance initialization method is being invoked. Each return instruction must match its method's return type. If the method returns a `boolean`, `byte`, `char`, `short`, or `int`, only the *ireturn* instruction may be used. If the method returns a `float`, `long`, or `double`, only an *freturn*, *lreturn*, or *dreturn* instruction, respectively, may be used. If the method returns a reference type, it must do so using an *areturn* instruction, and the type of the returned value must be assignment compatible (§2.6.7) with the return descriptor (§4.4.3) of the method. All instance initialization methods, class or interface initialization methods, and methods declared to return `void` must use only the *return* instruction.
- If *getfield* or *putfield* is used to access a protected field of a superclass that is a member of different runtime package than the current class, then the type of

the class instance being accessed must be the same as or a subclass of the current class. If *invokevirtual* or *invokespecial* is used to access a protected method of a superclass that is a member of different runtime package than the current class, then the type of the class instance being accessed must be the same as or a subclass of the current class

- The type of every class instance accessed by a *getfield* instruction or modified by a *putfield* instruction must be assignment compatible (§2.6.7) with the class type specified in the instruction.
- The type of every value stored by a *putfield* or *putstatic* instruction must be compatible with the descriptor of the field (§4.4.2) of the class instance or class being stored into. If the descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the value must be an `int`. If the descriptor type is `float`, `long`, or `double`, then the value must be a `float`, `long`, or `double`, respectively. If the descriptor type is a reference type, then the value must be of a type that is assignment compatible (§2.6.7) with the descriptor type.
- The type of every value stored into an array by an *aastore* instruction must be a reference type. The component type of the array being stored into by the *aastore* instruction must also be a reference type.
- Each *athrow* instruction must throw only values that are instances of class `Throwable` or of subclasses of `Throwable`. Each class mentioned in a `catch_type` item of a method's exception table must be `Throwable` or of subclasses of `Throwable`.
- Execution never falls off the bottom of the code array.
- No return address (a value of type `returnAddress`) may be loaded from a local variable.
- The instruction following each *jsr* or *jsr\_w* instruction may be returned to only by a single *ret* instruction.
- No *jsr* or *jsr\_w* instruction may be used to recursively call a subroutine if that subroutine is already present in the subroutine call chain. (Subroutines can be nested when using `try-finally` constructs from within a `finally` clause. For more information on Java virtual machine subroutines, see §4.11.1.6.)
- Each instance of type `returnAddress` can be returned to at most once. If a *ret* instruction returns to a point in the subroutine call chain above the *ret* instruc-

tion corresponding to a given instance of type `returnAddress`, then that instance can never be used as a return address.

## 4.11 Verification of class Files

Even though any compiler for the Java programming language must only produce class files that satisfy all the static constraints in the previous sections, the Java virtual machine has no guarantee that any file it is asked to load was generated by that compiler or is properly formed. Applications such as web browsers do not download source code, which they then compile; these applications download already-compiled class files. The browser needs to determine whether the class file was produced by a trustworthy compiler or by an adversary attempting to exploit the virtual machine.

---

An additional problem with compile-time checking is version skew. A user may have successfully compiled a class, say `PurchaseStockOptions`, to be a subclass of `TradingClass`. But the definition of `TradingClass` might have changed since the time the class was compiled in a way that is not compatible with preexisting binaries. Methods might have been deleted or had their return types or modifiers changed. Fields might have changed types or changed from instance variables to class variables. The access modifiers of a method or variable may have changed from `public` to `private`. For a discussion of these issues, see Chapter 13, “Binary Compatibility,” in the *The Java™ Language Specification*.

---

Because of these potential problems, the Java virtual machine needs to verify for itself that the desired constraints are satisfied by the class files it attempts to incorporate. A Java virtual machine implementation verifies that each class file satisfies the necessary constraints at linking time (§2.17.3).

Linking-time verification enhances the performance of the interpreter. Expensive checks that would otherwise have to be performed to verify constraints at run time for each interpreted instruction can be eliminated. The Java virtual machine can assume that these checks have already been performed. For example, the Java virtual machine will already know the following:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.



- The arguments to all the Java virtual machine instructions are of valid types.

The verifier also performs verification that can be done without looking at the code array of the `Code` attribute (§4.8.3). The checks performed include the following:

- Ensuring that `final` classes are not subclassed and that `final` methods are not overridden.
- Checking that every class (except `Object`) has a direct superclass.
- Ensuring that the constant pool satisfies the documented static constraints: for example, that each `CONSTANT_Class_info` structure in the constant pool contains in its `name_index` item a valid constant pool index for a `CONSTANT_Utf8_info` structure.
- Checking that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

Note that these checks do not ensure that the given field or method actually exists in the given class, nor does it check that the type descriptors given refer to real classes. They ensure only that these items are well formed. More detailed checking is performed when the byte codes themselves are verified, and during resolution.

Verification by type inference must be supported by all Java virtual machines, except those conforming to the JavaCard and J2ME CLDC profiles.

#### **4.11.1 Verification by Type Inference**

##### **4.11.1.1 The Process of Verification by Type Inference**

During linking, the verifier checks the code array of the `Code` attribute for each method of the `class` file by performing data-flow analysis on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point, the following is true:

- The operand stack is always the same size and contains the same types of values.
- No local variable is accessed unless it is known to contain a value of an appropriate type.
- Methods are invoked with the appropriate arguments.

- Fields are assigned only using values of appropriate types.
- All opcodes have appropriate type arguments on the operand stack and in the local variable array.
- There is never an uninitialized class instance in a local variable in code protected by an exception handler. However, an uninitialized class instance may be on the operand stack in code protected by an exception handler. When an exception is thrown, the contents of the operand stack are discarded.

For further information on this pass, see Section 4.11.1.2, “The Bytecode Verifier.”

For efficiency reasons, certain tests that could in principle be performed by the verifier are delayed until the first time the code for the method is actually invoked. In so doing, the verifier avoids loading `class` files unless it has to.

For example, if a method invokes another method that returns an instance of class A, and that instance is assigned only to a field of the same type, the verifier does not bother to check if the class A actually exists. However, if it is assigned to a field of the type B, the definitions of both A and B must be loaded in to ensure that A is a subclass of B.

#### 4.11.1.2 The Bytecode Verifier

This section looks at the verification of Java virtual machine code in more detail.

The code for each method is verified independently. First, the bytes that make up the code are broken up into a sequence of instructions, and the index into the code array of the start of each instruction is placed in an array. The verifier then goes through the code a second time and parses the instructions. During this pass a data structure is built to hold information about each Java virtual machine instruction in the method. The operands, if any, of each instruction are checked to make sure they are valid. For instance:

- Branches must be within the bounds of the code array for the method.
- The targets of all control-flow instructions are each the start of an instruction. In the case of a *wide* instruction, the *wide* opcode is considered the start of the instruction, and the opcode giving the operation modified by that *wide* instruction is not considered to start an instruction. Branches into the middle of an instruction are disallowed.
- No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates.
- All references to the constant pool must be to an entry of the appropriate type. For example: the instruction *getfield* must reference a field.

- The code does not end in the middle of an instruction.
- Execution cannot fall off the end of the code.
- For each exception handler, the starting and ending point of code protected by the handler must be at the beginning of an instruction or, in the case of the ending point, immediately past the end of the code. The starting point must be before the ending point. The exception handler code must start at a valid instruction, and it must not start at an opcode being modified by the *wide* instruction.

For each instruction of the method, the verifier records the contents of the operand stack and the contents of the local variable array prior to the execution of that instruction. For the operand stack, it needs to know the stack height and the type of each value on it. For each local variable, it needs to know either the type of the contents of that local variable or that the local variable contains an unusable or unknown value (it might be uninitialized). The bytecode verifier does not need to distinguish between the integral types (e.g., *byte*, *short*, *char*) when determining the value types on the operand stack.

Next, a data-flow analyzer is initialized. For the first instruction of the method, the local variables that represent parameters initially contain values of the types indicated by the method's type descriptor; the operand stack is empty. All other local variables contain an illegal value. For the other instructions, which have not been examined yet, no information is available regarding the operand stack or local variables.

Finally, the data-flow analyzer is run. For each instruction, a "changed" bit indicates whether this instruction needs to be looked at. Initially, the "changed" bit is set only for the first instruction. The data-flow analyzer executes the following loop:

1. Select a virtual machine instruction whose "changed" bit is set. If no instruction remains whose "changed" bit is set, the method has successfully been verified. Otherwise, turn off the "changed" bit of the selected instruction.
2. Model the effect of the instruction on the operand stack and local variable array by doing the following:
  - If the instruction uses values from the operand stack, ensure that there are a sufficient number of values on the stack and that the top values on the stack are of an appropriate type. Otherwise, verification fails.
  - If the instruction uses a local variable, ensure that the specified local variable contains a value of the appropriate type. Otherwise, verification fails.

- If the instruction pushes values onto the operand stack, ensure that there is sufficient room on the operand stack for the new values. Add the indicated types to the top of the modeled operand stack.
  - If the instruction modifies a local variable, record that the local variable now contains the new type.
3. Determine the instructions that can follow the current instruction. Successor instructions can be one of the following:
    - The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance *goto*, *return*, or *throw*). Verification fails if it is possible to “fall off” the last instruction of the method.
    - The target(s) of a conditional or unconditional branch or switch.
    - Any exception handlers for this instruction.
  4. Merge the state of the operand stack and local variable array at the end of the execution of the current instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information.
    - If this is the first time the successor instruction has been visited, record that the operand stack and local variable values calculated in steps 2 and 3 are the state of the operand stack and local variable array prior to executing the successor instruction. Set the “changed” bit for the successor instruction.
    - If the successor instruction has been seen before, merge the operand stack and local variable values calculated in steps 2 and 3 into the values already there. Set the “changed” bit if there is any modification to the values.
  5. Continue at step 1.

To merge two operand stacks, the number of values on each stack must be identical. The types of values on the stacks must also be identical, except that differently typed reference values may appear at corresponding places on the two stacks. In this case, the merged operand stack contains a reference to an instance of the first common superclass of the two types. Such a reference type always exists because the type `Object` is a

superclass of all class and interface types. If the operand stacks cannot be merged, verification of the method fails.

To merge two local variable array states, corresponding pairs of local variables are compared. If the two types are not identical, then unless both contain reference values, the verifier records that the local variable contains an unusable value. If both of the pair of local variables contain reference values, the merged state contains a reference to an instance of the first common superclass of the two types.

If the data-flow analyzer runs on a method without reporting a verification failure, then the method has been successfully verified by the `class` file verifier.

Certain instructions and data types complicate the data-flow analyzer. We now examine each of these in more detail.

#### 4.11.1.3 Values of Types `long` and `double`

Values of the `long` and `double` types are treated specially by the verification process.

Whenever a value of type `long` or `double` is moved into a local variable at index  $n$ , index  $n + 1$  is specially marked to indicate that it has been reserved by the value at index  $n$  and must not be used as a local variable index. Any value previously at index  $n + 1$  becomes unusable.

Whenever a value is moved to a local variable at index  $n$ , the index  $n - 1$  is examined to see if it is the index of a value of type `long` or `double`. If so, the local variable at index  $n - 1$  is changed to indicate that it now contains an unusable value. Since the local variable at index  $n$  has been overwritten, the local variable at index  $n - 1$  cannot represent a value of type `long` or `double`.

Dealing with values of types `long` or `double` on the operand stack is simpler; the verifier treats them as single values on the stack. For example, the verification code for the *dadd* opcode (add two `double` values) checks that the top two items on the stack are both of type `double`. When calculating operand stack length, values of type `long` and `double` have length two.

Untyped instructions that manipulate the operand stack must treat values of type `double` and `long` as atomic (indivisible). For example, the verifier reports a failure if the top value on the stack is a `double` and it encounters an instruction such as *pop* or *dup*. The instructions *pop2* or *dup2* must be used instead.

#### 4.11.1.4 Instance Initialization Methods and Newly Created Objects

Creating a new class instance is a multistep process. The statement

```

...
new myClass(i, j, k);
...

```

can be implemented by the following:

```

...
new #1           // Allocate uninitialized space for myClass
dup             // Duplicate object on the operand stack
iload_1         // Push i
iload_2         // Push j
iload_3         // Push k
invokespecial #5 // Invoke myClass.<init>
...

```

This instruction sequence leaves the newly created and initialized object on top of the operand stack. (Additional examples of compilation to the instruction set of the Java virtual machine are given in Chapter 7, “Compiling for the Java Virtual Machine.”)

The instance initialization method (§3.9) for class `myClass` sees the new uninitialized object as its `this` argument in local variable `0`. Before that method invokes another instance initialization method of `myClass` or its direct superclass on `this`, the only operation the method can perform on `this` is assigning fields declared within `myClass`.

When doing dataflow analysis on instance methods, the verifier initializes local variable `0` to contain an object of the current class, or, for instance initialization methods, local variable `0` contains a special type indicating an uninitialized object. After an appropriate instance initialization method is invoked (from the current class or the current superclass) on this object, all occurrences of this special type on the verifier’s model of the operand stack and in the local variable array are replaced by the current class type. The verifier rejects code that uses the new object before it has been initialized or that initializes the object more than once. In addition, it ensures that every normal return of the method has invoked an instance initialization method either in the class of this method or in the direct superclass.

Similarly, a special type is created and pushed on the verifier’s model of the operand stack as the result of the Java virtual machine instruction `new`. The special type indicates the instruction by which the class instance was created and the type of the uninitialized class instance created. When an instance initialization method is invoked on that class instance, all occurrences of the special type are replaced by the intended type of the class instance. This change in type may propagate to subsequent instructions as the dataflow analysis proceeds.

The instruction number needs to be stored as part of the special type, as there may be multiple not-yet-initialized instances of a class in existence on the operand stack at one time. For example, the Java virtual machine instruction sequence that implements

```
new InputStream(new Foo(), new InputStream("foo"))
```

may have two uninitialized instances of `InputStream` on the operand stack at once. When an instance initialization method is invoked on a class instance, only those occurrences of the special type on the operand stack or in the local variable array that are the *same object* as the class instance are replaced.

A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a `finally` clause. Otherwise, a devious piece of code might fool the verifier into thinking it had initialized a class instance when it had, in fact, initialized a class instance created in a previous pass through a loop.

#### 4.11.1.5 Exception Handlers

Java virtual machine code produced by Sun's compiler for the Java programming language always generates exception handlers such that:

- Either the ranges of instructions protected by two different exception handlers always are completely disjoint, or else one is a subrange of the other. There is never a partial overlap of ranges.
- The handler for an exception will never be inside the code that is being protected.
- The only entry to an exception handler is through an exception. It is impossible to fall through or "goto" the exception handler.

These restrictions are not enforced by the `class` file verifier since they do not pose a threat to the integrity of the Java virtual machine. As long as every nonexceptional path to the exception handler causes there to be a single object on the operand stack, and as long as all other criteria of the verifier are met, the verifier will pass the code.

## 4.12 Limitations of the Java Virtual Machine

The following limitations of the Java virtual machine are implicit in the `class` file format:

- The per-class or per-interface constant pool is limited to 65535 entries by the 16-bit `constant_pool_count` field of the `ClassFile` structure (§4.2). This acts as an internal limit on the total complexity of a single class or interface.
- The greatest number of local variables in the local variables array of a frame created upon invocation of a method is limited to 65535 by the size of the `max_locals` item of the `Code` attribute (§4.8.3) giving the code of the method, and by the 16-bit local variable indexing of the Java virtual machine instruction set. Note that values of type `long` and `double` are each considered to reserve two local variables and contribute two units toward the `max_locals` value, so use of local variables of those types further reduces this limit.
- The number of fields that may be declared by a class or interface is limited to 65535 by the size of the `fields_count` item of the `ClassFile` structure (§4.2). Note that the value of the `fields_count` item of the `ClassFile` structure does not include fields that are inherited from superclasses or superinterfaces.
- The number of methods that may be declared by a class or interface is limited to 65535 by the size of the `methods_count` item of the `ClassFile` structure (§4.2). Note that the value of the `methods_count` item of the `ClassFile` structure does not include methods that are inherited from superclasses or superinterfaces.
- The number of direct superinterfaces of a class or interface is limited to 65535 by the size of the `interfaces_count` item of the `ClassFile` structure (§4.2).
- The size of an operand stack in a frame (§3.6) is limited to 65535 values by the `max_stack` field of the `Code` attribute (§4.8.3). Note that values of type `long` and `double` are each considered to contribute two units toward the `max_stack` value, so use of values of these types on the operand stack further reduces this limit.
- The number of dimensions in an array is limited to 255 by the size of the *dimensions* opcode of the *multianewarray* instruction and by the constraints imposed on the *multianewarray*, *anewarray*, and *newarray* instructions by §4.10.2.
- The number of method parameters is limited to 255 by the definition of a method descriptor (§4.4.3), where the limit includes one unit for `this` in the case of instance or interface method invocations. Note that a method descriptor is defined in terms of a notion of method parameter length in which a parameter



of type `long` or `double` contributes two units to the length, so parameters of these types further reduce the limit.

- The length of field and method names, field and method descriptors, and other constant string values is limited to 65535 characters by the 16-bit unsigned `length` item of the `CONSTANT_Utf8_info` structure (§4.5.7). Note that the limit is on the number of bytes in the encoding and not on the number of encoded characters. UTF-8 encodes some characters using two or three bytes. Thus, strings incorporating multibyte characters are further constrained.

