This document proposes changes to the Java Virtual Machine Specification to introduce a new category of types called value types.

Last updated: 08/07/2018 in file L-World-JVMS-draft-20180807.pdf

Changelog

07/26/2018:

- Cleanup / clarification about static value fields
- Cleanup in loading/initialization between containers and value classes
- Precisions in verifier rules
- Minor editorial changes

06/20/2018:

- Added consistency checks rules
- Removed restriction that ACC_FLATTENABLE cannot be used on static fields

05/23/2018:

- Added ValueTypes attribute specification
- Added consistency rules between ACC_FLATENNABLE and the ValueTypes attribute
- Added pre-loading rules for method argument types and return value types
- Added class initialization requirement for any use of value class' default value

04/11/2018 changes:

- Restored restriction that ACC_FLATTENABLE cannot be used on static fields

02/05/2018 changes:

- Renamed ACC_NON_NULLABLE with ACC_FLATTENABLE
- Removed restriction that ACC_FLATTENABLE cannot be used on static fields
- Added rules about circularity restrictions for fields with the ACC_FLATTENABLE flag set
- Added rules about initialization of classes of flattenable fields

01/31/2018 changes:

- Replaced ACC_VALUE_TYPE flag for field with ACC_FLATTENABLE, with a new semantic associated with this flag, including field initialization, handling of the null reference by putfield/withfield/aastore
- Added definition of a value class' default value.
- Removed areturn, and invoke* (they do not changes anymore)
- Added updated specifications for anewarray and multi anewarray

01/24/2018 changes:

- cleaned up null reference handling for value class references (areturn, invoke*)
- fix runtime error for putfield
- rename vdefault and vwithfield to defaultvalue and withfield respectively, and change opcode value
- Add description of ACC_VALUE_TYPE flag meaning for fields, and handling of such field during the loading phase

01/23/2018 changes:

- removal of Q-descriptors

2.4. Reference Types and Values

There are four kinds of `reference` types: object class types, value class types, array types, and interface types. Their values are references to dynamically created object class instances, value class instances, arrays, or object class instances or value class instances or arrays that implement interfaces, respectively.

An object class type is a class type which is neither a value class type nor an array class type.

A value class type defines a class for which all instances are identity-less and immutable.

Unless specified otherwise, the term class is used to designate either an object class or a value class.

An array type consists of a *component type* with a single dimension (whose length is not given by the type). The component type of an array type may itself be an array type. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; this is called the *element type* of the array type. The element type of an array type is necessarily either a primitive type, or a class type, or an interface type.

A `reference` value may also be the special null reference, a reference to no object, which will be denoted here by `null`. The `null` reference initially has no run-time type, but may be cast to any type. The default value of a `reference` type is `null`.

This specification does not mandate a concrete value encoding `null`.

Value classes have a special value, called the default value, which has all its instance variables set to their default value according to their declaration (§4.5) and the initial default value of each type (§2.3, §2.4). This default value for value classes is a valid, fully initialized value. Any use of a default value of a value type requires class initialization of the value class since a default value is an instance and all instance bytecodes assume pre-initialization. This is subject to the same exception all classes have, which is that during `<clinit>` the initializing thread can create instances (including default values) of themselves.

Note: the JVMS treats references to value class instances as if they were indirection to dynamically created values allocated in or outside of the Java heap. Implementations are free to use a different representation, for instance using an immediate representation rather than an indirection.

2.11.5. Instances Creation and Manipulation

The Java Virtual Machine creates and manipulates object class instances, value class instances and arrays using distinct sets of instructions:

- Create a new object class instance: *new.*
- Create a new value class instance: *defaultvalue, withfield*
- Create a new array: *newarray*, *anewarray*, *multianewarray*.
- Access fields of classes (`static` fields, known as class variables) and fields of class instances (non-`static` fields, known as instance variables): *getstatic*, *putstatic*, *getfield*, *putfield (except for value classes: because of their immutability, putfield cannot be used on a value class instance, withfield must be used instead)*.
- Load an array component onto the operand stack: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Store a value from the operand stack as an array component: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- Get the length of array: *arraylength*.
- Check properties of class instances or arrays: *instanceof*, *checkcast*.

4.1. 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 Mi.0 ≤ v ≤ Mj.m. The release level of the Java SE platform to which a Java Virtual Machine implementation conforms is responsible for determining the range.

Oracle's Java Virtual Machine implementation in JDK release 1.0.2 supports `class` file format versions 45.0 through 45.3 inclusive. JDK releases 1.1. support `class` file format versions in the range 45.0 through 45.65535 inclusive. For k ≥ 2, JDK release 1.k supports `class` file format versions in the range 45.0 through 44+k.0 inclusive.*

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 §4.4.5.

constant_pool[]
The `constant_pool` is a table of structures (§4.4) 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.

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 specified in Table 4.1-A.

Table 4.1-A. Class access and property modifiers

Flag Name	Value	Interpretation
`ACC_PUBLIC`	0x0001	Declared `public`; may be accessed from outside its package.
`ACC_FINAL`	0x0010	Declared `final`; no subclasses allowed.
`ACC_SUPER`	0x0020	Treat superclass methods specially when invoked by the *invokespecial*instruction.
`ACC_VALUE_TYPE`	0x0100	Is a value class, not an object class.
`ACC_INTERFACE`	0x0200	Is an interface, not a class.
`ACC_ABSTRACT`	0x0400	Declared `abstract`; must not be instantiated.
`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.

An interface is distinguished by the `ACC_INTERFACE` flag being set. If the `ACC_INTERFACE` flag is not set, this `class` file defines a class, not an interface.

If the `ACC_INTERFACE` flag is set, the `ACC_ABSTRACT` flag must also be set, and the `ACC_FINAL`, `ACC_SUPER`, and `ACC_ENUM` flags set must not be set.

If the `ACC_INTERFACE` flag is not set, any of the other flags in Table 4.1-A may be set except `ACC_ANNOTATION`. However, such a `class` file must not have both its `ACC_FINAL` and `ACC_ABSTRACT` flags set (JLS §8.1.1.2).

A value class is distinguished by the `ACC_VALUE_TYPE` flag being set. If the `ACC_VALUE_TYPE` flag is not set, this `class` file defines an object class or an interface, not a value class. A `class` file with the `ACC_VALUE_TYPE` flag set must not have either `ACC_INTERFACE` or `ACC_ABSTRACT` flags set. The super class of a value class must be the `java.lang.Object` class.

If the `ACC_VALUE_TYPE` flag is set, the `ACC_FINAL` flag must also be set, and all non-static fields declared in this class file must have the `ACC_FINAL` flag set too.

A value class must not have an `<init>` method.

A value class must not have any synchronized instance methods.

If neither the ACC_VALUE_TYPE or the ACC_INTERFACE flag is set, then this class file defines an object class.

The ACC_SUPER flag indicates which of two alternative semantics is to be expressed by the *invokespecial* instruction (§*invokespecial*) if it appears in this class or interface. Compilers to the instruction set of the Java Virtual Machine should set the ACC_SUPER flag. In Java SE 8 and above, the Java Virtual Machine considers the ACC_SUPER flag to be set in every class file, regardless of the actual value of the flag in the class file and the version of the class file.

The ACC_SUPER flag exists for backward compatibility with code compiled by older compilers for the Java programming language. In JDK releases prior to 1.0.2, the compiler generated access_flags in which the flag now representing ACC_SUPER had no assigned meaning, and Oracle's Java Virtual Machine implementation ignored the flag if it was set.

The ACC_SYNTHETIC flag indicates that this class or interface was generated by a compiler and does not appear in source code.

An annotation type must have its ACC_ANNOTATION flag set. If the ACC_ANNOTATION flag is set, the ACC_INTERFACE flag must also be set.

The ACC_ENUM flag indicates that this class or its superclass is declared as an enumerated type. A class file must not have both ACC_ENUM and ACC_VALUE_TYPE flags set.

All bits of the access_flags item not assigned in Table 4.1-A 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 structure (§4.4.1) 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 structure representing the direct superclass of the class defined by this class file. Neither the direct superclass nor any of its superclasses may have the ACC_FINAL flag set in the access_flags item of its ClassFile structure.

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`.

For a value class, 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 ≤ *i* < `interfaces_count`, must be a`CONSTANT_Class_info` 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` 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` structure (§4.5) 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` structure (§4.6) giving a complete description of a method in this class or interface. If neither of the `ACC_NATIVE` and `ACC_ABSTRACT` flags are set in the `access_flags` item of a `method_info` structure, 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 methods, instance initialization methods (§2.9), and any class or interface initialization method (§2.9). 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 in the `attributes` table of this class.

attributes[]
Each value of the `attributes` table must be an `attribute_info` structure (§4.7).

The attributes defined by this specification as appearing in the `attributes` table of a `ClassFile` structure are listed in Table 4.7-C.

The rules concerning attributes defined to appear in the `attributes` table of a `ClassFile` structure are given in §4.7.

The rules concerning non-predefined attributes in the `attributes` table of a `ClassFile` structure are given in §4.7.1.

4.3.2. Field Descriptors

A *field descriptor* represents the type of a class, instance, or local variable.

```
FieldDescriptor:
FieldType
FieldType:
BaseType
InstanceType
ArrayType
BaseType:
(one of)
B C D F I J S Z
InstanceType:
L ClassName ;
ArrayType:
[ ComponentType
ComponentType:
FieldType
```

The characters of *BaseType*, the `L` and `;` of *InstanceType*, and the `[` of *ArrayType* are all ASCII characters.

ClassName represents a binary class or interface name encoded in internal form (§4.2.1). The interpretation of field descriptors as types is shown in Table 4.3-A.

A field descriptor representing an array type is valid only if it represents a type with 255 or fewer dimensions.

Table 4.3-A. Interpretation of field descriptors

FieldType term	Type	Interpretation
`B`	`byte`	signed byte
`C`	`char`	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
`D`	`double`	double-precision floating-point value
`F`	`float`	single-precision floating-point value
`I`	`int`	integer
`J`	`long`	long integer
`L` *ClassName* `;`	`reference`	an instance of class *ClassName*
`S`	`short`	signed short
`Z`	`boolean`	`true` or `false`
`[`	`reference`	one array dimension

The field descriptor of an instance variable of type `int` is simply `I`.

The field descriptor of an instance variable of type `Object` is `Ljava/lang/Object;`. Note that the internal form of the binary name for class `Object` is used.

The field descriptor of an instance variable of the multidimensional array type `double[][][]` is `[[[D`.

4.4.1. 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 as follows:

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` structure (§4.4.7) representing a valid binary class or interface name encoded in internal form (§4.2.1).

Because arrays are objects, the opcodes *anewarray* and *multianewarray* - but not the opcode *new* or *defaultvalue* - can reference array "classes" via `CONSTANT_Class_info` structures in the `constant_pool` table. For such array classes, the name of the class is the descriptor of the array type (§4.3.2).

The opcode *defaultvalue* - but not the opcode *new* - can reference value classes via `CONSTANT_Class_info` structures in the `constant_pool` table.

The opcode *new* - but not the opcode *defaultvalue* - can reference object classes via `CONSTANT_Class_info` structures in the `constant_pool` table.

For example, the class name representing the two-dimensional array type `int[][]` is `[[I`, while the class name representing the type `Thread[]` is `[Ljava/lang/Thread;`.

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

4.5. Fields

Each field is described by a `field_info` structure.

No two fields in one `class` file may have the same name and descriptor (§4.3.2).

The structure has the following format:

```
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 specified in Table 4.5-A.

Table 4.5-A. Field access and property flags

Flag Name	Flag Name	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final; never directly assigned to after object construction (JLS §17.5).
ACC_VOLATILE	0x0040	Declared volatile; cannot be cached.
ACC_TRANSIENT	0x0080	Declared transient; not written or read by a persistent object manager.
ACC_FLATTENABLE	0x0100	Declared as a candidate for flattening
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ENUM	0x4000	Declared as an element of an enum.

Fields of classes may set any of the flags in Table 4.5-A. However, each field of a class may have at most one of its ACC_PUBLIC, ACC_PRIVATE, and ACC_PROTECTED flags set (JLS §8.3.1), and must not have both its ACC_FINAL and ACC_VOLATILE flags set (JLS §8.3.1.4).

Fields of value class may have at most one of its ACC_PUBLIC, ACC_PRIVATE, and ACC_PROTECTED flags set (JLS §8.3.1), Each field with the ACC_STATIC flag not set must have its ACC_FINAL flag set.

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 4.5-A set (JLS §9.3).

The ACC_SYNTHETIC flag indicates that this field was generated by a compiler and does not appear in source code.

The ACC_ENUM flag indicates that this field is used to hold an element of an enumerated type.

The ACC_FLATTENABLE flag indicates that this field must never store the null reference. The field signature must be the signature of a class. The class specified in the field's signature is loaded during the loading phase of the class declaring this field. The class of the field must be listed in the ValueTypes attribute of the current class. This field must be initialized with the default value of this value class.

A field declared with a reference type and the `ACC_FLATTENABLE` flag not set must be initialized with the `null` reference.

A field declared with the `ACC_FLATTENABLE` flag set must not have the type of its declaring class, nor refer indirectly to its declaring class through a chain of fields with the `ACC_FLATTENABLE` flag set.

All bits of the `access_flags` item not assigned in Table 4.5-A 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` structure (§4.4.7) which represents a valid unqualified name denoting a field (§4.2.2).

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` structure (§4.4.7) which represents a valid field descriptor (§4.3.2).

attributes_count
The value of the `attributes_count` item indicates the number of additional attributes of this field.

attributes[]
Each value of the `attributes` table must be an `attribute_info` structure (§4.7).

A field can have any number of optional attributes associated with it.

The attributes defined by this specification as appearing in the `attributes` table of a `field_info` structure are listed in Table 4.7-C.

The rules concerning attributes defined to appear in the `attributes` table of a `field_info` structure are given in §4.7.

The rules concerning non-predefined attributes in the `attributes` table of a `field_info` structure are given in §4.7.1.

4.7.25. ValueTypes attribute

The ValueTypes attribute is a variable-length attribute in the attributes table of a ClassFile structure (§4.1). The ValueTypes attribute records types that were known to be value class types at the time the class file was generated.

There may be at most one ValueTypes attribute in the attributes table of a ClassFile structure.

The ValueTypes attribute has the following format:

```
ValueTypes_attribute {
    u2              attribute_name_index;
    u4              attribute_length;
    u2              number_of_value_types;
    {
       u2 value_types_info_index;
    } value_types[number_of_value_types];
}
```

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 (§4.4.7) representing the string “ValueTypes".

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

number_of_value_types
The value of the number_of_value_types item indicates the number of entries in the value_types array.

value_types[]
Each value in the value_types array must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info structure (§4.4.1).

4.9.1. Static Constraints

The *static constraints* on a `class` file are those defining the well-formedness of the file. These constraints have been given in the previous sections, except for static constraints on the code in the `class` file. The static constraints on the 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:

- Only instances of the instructions documented in §6.5 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.
  If the `class` file version number is 51.0 or above, then neither the *jsr* opcode or the *jsr_w* opcode may appear in the `code` array.

- The opcode of the first instruction in the `code` array begins at index `0`.

- 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 and each *ldc_w* instruction must be a valid index into the `constant_pool`table. The constant pool entry referenced by that index must be of type:
    - `CONSTANT_Integer`, `CONSTANT_Float`, or `CONSTANT_String` if the `class` file version number is less than 49.0.
    - `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_String`, or `CONSTANT_Class` if the `class` file version number is 49.0 or 50.0.
    - `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_String`, `CONSTANT_Class`, `CONSTANT_MethodType`, or `CONSTANT_MethodHandle` if the `class` file version number is 51.0 or above.

- 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`.
  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*, *putstatic* and *withfield* 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 constant pool entry referenced by the operand of a *withfield* instruction must be a `CONSTANT_Fieldref` entry that represents a field of a class listed in the ValueTypes attrribute.
- The constant pool entry referenced by the operand of a *putfield* instruction must be a `CONSTANT_Fieldref` entry that represents a field of a class not listed in the ValueTypes attrribute.

- The *indexbyte* operands of each *invokevirtual* 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`.

- The *indexbyte* operands of each *invokespecial* and *invokestatic* instruction must represent a valid index into the `constant_pool` table. If the `class` file version number is less than 52.0, the constant pool entry referenced by that index must be of type `CONSTANT_Methodref`; if the `class` file version number is 52.0 or above, the constant pool entry referenced by that index must be of type `CONSTANT_Methodref` or `CONSTANT_InterfaceMethodref`.

- 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 *indexbyte* operands of each *invokedynamic* instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_InvokeDynamic`.
  The third and fourth operand bytes of each *invokedynamic* instruction must have the value zero.

- Only the *invokespecial* instruction is allowed to invoke an instance initialization method (§2.9).
  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 operands of each *instanceof*, *checkcast*, *new*, *anewarray* and *defaultvalue* 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 *new* instruction may reference a constant pool entry of type `CONSTANT_Class` that represents an array type (§4.3.2). The *new* instruction cannot be used to create an array.

- No *new* instruction may reference a constant pool entry of type `CONSTANT_Class` that represents a class type listed in the ValueTypes attribute. The *new* instruction cannot be used to create an instance of a value class.

- No *defaultvalue* instruction may reference a constant pool entry of type `CONSTANT_Class` that represents an array type (§4.3.2). The *defaultvalue* instruction cannot be used to create an array.

- No *defaultvalue* instruction may reference a constant pool entry of type `CONSTANT_Class` that represents a class type not listed in the ValueTypes attribute. The *defaultvalue* instruction cannot be used to create an instance of a class other than a value class.

- No *anewarray* instruction may be used to create an array of more than 255 dimensions.

- 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 non-negative 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 `max_locals - 1`.

- The *index* operand of each *lload*, *dload*, *lstore*, and *dstore* instruction must be no greater than `max_locals - 2`.
  The implicit index of each *lload_<n>*, *dload_<n>*, *lstore_<n>*,

and *dstore_<n>* instruction must be no greater than `max_locals - 2`.

- The *indexbyte* operands of each *wide* instruction modifying an *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *iinc*, or *ret* instruction must represent a non-negative 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 non-negative integer no greater than `max_locals - 2`.

4.10.1.9. Type Checking Instructions

defaultvalue

A *defaultvalue* instruction with operand `CP` at offset `Offset` is type safe iff `CP` refers to a constant pool entry denoting a value class type, and one can validly push `the class type` onto the incoming operand stack yielding the outgoing type state.

```
    <Prolog code is TBD>
```

withfield

A *withfield* instruction with operand `CP` is type safe iff all of the following are true:

- Its first operand, `CP`, refers to a constant pool entry denoting a field whose declared type is `FieldType`, declared in a class `FieldClassName`. `FieldClassName` must be a value class type.
- One can validly pop types matching `FieldType` and `FieldClassName` off the incoming operand stack yielding the outgoing type state.

```
    <Prolog code is TBD>
```

5.3.5. Deriving a Class from a `class` File Representation

The following steps are used to derive a `Class` object for the nonarray class or interface C denoted by `N` using loader `L` from a purported representation in `class` file format.

1. **First, the Java Virtual Machine determines whether it has already recorded that `L` is an initiating loader of a class or interface denoted by `N`. If so, this creation attempt is invalid and loading throws a `LinkageError`.**

2. Otherwise, the Java Virtual Machine attempts to parse the purported representation. However, the purported representation may not in fact be a valid representation of C.
   This phase of loading must detect the following errors:

   - **If the purported representation is not a `ClassFile` structure (§4.1, §4.8), loading throws an instance of `ClassFormatError`.**

   - **Otherwise, if the purported representation is not of a supported major or minor version (§4.1), loading throws an instance of `UnsupportedClassVersionError`.**
     `UnsupportedClassVersionError`, a subclass of `ClassFormatError`, was introduced to enable easy identification of a `ClassFormatError` caused by an attempt to load a class whose representation uses an unsupported version of the `class` file format. In JDK release 1.1 and earlier, an instance of `NoClassDefFoundError` or `ClassFormatError` was thrown in case of an unsupported version, depending on whether the class was being loaded by the system class loader or a user-defined class loader.

   - **Otherwise, if the purported representation does not actually represent a class named `N`, loading throws an instance of `NoClassDefFoundError` or an instance of one of its subclasses.**

3. If C has a direct superclass, the symbolic reference from C to its direct superclass is resolved using the algorithm of §5.4.3.1. Note that if C is an interface or a value class it must have `Object` as its direct superclass, which must already have been loaded. Only `Object` has no direct superclass.
   Any exceptions that can be thrown due to class or interface resolution can be thrown as a result of this phase of loading. In addition, this phase of loading must detect the following errors:

   - **If the class or interface named as the direct superclass of C is in fact an interface or a value class, loading throws an `IncompatibleClassChangeError`.**

- **Otherwise, if any of the superclasses of C is C itself, loading throws a `ClassCircularityError`.**

4. If C has any direct superinterfaces, the symbolic references from C to its direct superinterfaces are resolved using the algorithm of §5.4.3.1.
   Any exceptions that can be thrown due to class or interface resolution can be thrown as a result of this phase of loading. In addition, this phase of loading must detect the following errors:

   - **If any of the classes or interfaces named as direct superinterfaces of C is not in fact an interface, loading throws an `IncompatibleClassChangeError`.**

   - **Otherwise, if any of the superinterfaces of C is C itself, loading throws a `ClassCircularityError`.**

5. For each field declared by C with the `ACC_FLATTENABLE` flag set, the symbolic reference from C to the class of this field is resolved using the algorithm of §5.4.3.1.
   Any exceptions that can be thrown due to class or interface resolution can be thrown as a result of this phase of loading. If the field marked as ACC_FLATTENABLE contains the class C either directly or indirectly, via a chain of fields with the ACC_FLATTENABLE flat set, loading throws a `ClassCircularityError`.
   If the resolved class is a value class, the JVM may use its knowledge of the layout of the resolved class instances to flatten the field.
   If the resolved class is not a value class, an `IncompatibleClassChangeError` is thrown.

6. The Java Virtual Machine marks C as having `L` as its defining class loader and records that `L` is an initiating loader of C (§5.3.4).

5.4.2. Preparation

Preparation involves creating the static fields for a class or interface and initializing such fields to their default values (§2.3, §2.4). This does not require the execution of any Java Virtual Machine code; explicit initializers for static fields are executed as part of initialization (§5.5), not preparation.

During preparation of a class or interface C, the Java Virtual Machine also imposes loading constraints (§5.3.4) and value types consistency checking. Let L_1 be the defining loader of C. For each method `m` declared in C that overrides (§5.4.5) a method

declared in a superclass or superinterface <D, L_2>, the Java Virtual Machine imposes the following loading constraints and value types consistency checking:

- Given that the return type of `m` is T_r, and that the formal parameter types of `m` are T_{f1}, ..., T_{fn}, then:

  If T_r not an array type, let T_0 be T_r; otherwise, let T_0 be the element type (§2.4) of T_r.

  For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.

  - Then $T_i^{L_1} = T_i^{L_2}$ for *i* = 0 to *n,*
  - T_i is either listed in ValueTypes attributes of both <`C`, L_1> and <`D`, L_2>, or is not listed in either of them, otherwise an `IncompatibleClassChangeError` is thrown.

Furthermore, if C implements a method `m` declared in a superinterface <I, L_3> of C, but C does not itself declare the method `m`, then let <D, L_2> be the superclass of C that declares the implementation of method `m` inherited by C. The Java Virtual Machine imposes the following constraints:

- Given that the return type of `m` is T_r, and that the formal parameter types of `m` are T_{f1}, ..., T_{fn}, then:

  If T_r not an array type, let T_0 be T_r; otherwise, let T_0 be the element type (§2.4) of T_r.

  For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.

  - Then $T_i^{L_2} = T_i^{L_3}$ for *i* = 0 to *n,*
  - T_i is either listed in ValueTypes attributes of both <D, L_2> and <`I`, L_3>, or is not listed in either of them, otherwise an `IncompatibleClassChangeError` is thrown.

Preparation may occur at any time following creation but must be completed prior to initialization.

5.4.3. Resolution

The Java Virtual Machine instructions *anewarray*, *checkcast*, *getfield*, *getstatic*, *instanceof*, *invokedynamic*, *invoke interface*, *invokespecial*, *invokestatic*, *invokevirtual*, *ldc*, *ldc_w*, *multianewarray*, *new*, *put field*, and *putstatic* make symbolic references to the run-time constant pool. Execution of any of these instructions requires resolution of its symbolic reference.

Resolution is the process of dynamically determining concrete values from symbolic references in the run-time constant pool.

Resolution of the symbolic reference of one occurrence of an *invokedynamic* instruction *does not* imply that the same symbolic reference is considered resolved for any other *invokedynamic* instruction.

For all other instructions above, resolution of the symbolic reference of one occurrence of an instruction *does* imply that the same symbolic reference is considered resolved for any other non-*invokedynamic* instruction.

(The above text implies that the concrete value determined by resolution for a specific *invokedynamic* instruction is a call site object bound to that specific *invokedynamic* instruction.)

Resolution can be attempted on a symbolic reference that has already been resolved. An attempt to resolve a symbolic reference that has already successfully been resolved always succeeds trivially and always results in the same entity produced by the initial resolution of that reference.

If an error occurs during resolution of a symbolic reference, then an instance of `IncompatibleClassChangeError` (or a subclass) must be thrown at a point in the program that (directly or indirectly) uses the symbolic reference.

If an attempt by the Java Virtual Machine to resolve a symbolic reference fails because an error is thrown that is an instance of `LinkageError` (or a subclass), then subsequent attempts to resolve the reference always fail with the same error that was thrown as a result of the initial resolution attempt.

This means that a class in one module that attempts to access, via resolution of a symbolic reference in its run-time constant pool, an unexported `public` type in a different module will always receive the same error indicating an inaccessible type (§5.4.4), even if the Java SE Platform API is used to dynamically export the `public` type's package at some time after the class's first attempt.

A symbolic reference to a call site specifier by a specific *invokedynamic* instruction must not be resolved prior to execution of that instruction.

In the case of failed resolution of an *invokedynamic* instruction, the bootstrap method is not re-executed on subsequent resolution attempts.

Certain of the instructions above require additional linking checks when resolving symbolic references. For instance, in order for a *getfield* instruction to successfully resolve the symbolic reference to the field on which it operates, it must not only complete the field resolution steps given in §5.4.3.2 but also check that the field is not `static`. If it is a `static` field, a linking exception must be thrown.

Notably, in order for an *invokedynamic* instruction to successfully resolve the symbolic reference to a call site specifier, the bootstrap method specified therein must complete normally and return a suitable call site object. If the bootstrap method completes abruptly or returns an unsuitable call site object, a linking exception must be thrown.

Linking exceptions generated by checks that are specific to the execution of a particular Java Virtual Machine instruction are given in the description of that instruction and are not covered in this general discussion of resolution. Note that such exceptions, although described as part of the execution of Java Virtual Machine instructions rather than resolution, are still properly considered failures of resolution.

The following sections describe the process of resolving a symbolic reference in the run-time constant pool (§5.1) of a class or interface D. Details of resolution differ with the kind of symbolic reference to be resolved.

5.4.3.1. Class and Interface Resolution

To resolve an unresolved symbolic reference from D to a class or interface C denoted by `N`, the following steps are performed:

1. The defining class loader of D is used to create a class or interface denoted by `N`. This class or interface is C. The details of the process are given in §5.3.
   Any exception that can be thrown as a result of failure of class or interface creation can thus be thrown as a result of failure of class and interface resolution.

2. If C is an array class and its element type E is a `reference` type, then a symbolic reference to the class or interface representing E is resolved by invoking the algorithm in §5.4.3.1 recursively.

3. **If C or E is a value class but is not listed in the ValueTypes attribute of D, or if C or E is not a value class but is listed in the ValueTypes attribute of D, the class resolution throws an `IncompatibleClassChangeError`.**

4. Finally, access permissions to C are checked.

   - **If C is not accessible (§5.4.4) to D, class or interface resolution throws an `IllegalAccessError`.**
     This condition can occur, for example, if C is a class that was originally declared to be `public` but was changed to be non-`public` after D was compiled.

If steps 1 and 2 succeed but step 3 or 4 fail, C is still valid and usable. Nevertheless, resolution fails, and D is prohibited from accessing C or E.

5.4.3.2. Field Resolution

To resolve an unresolved symbolic reference from D to a field in a class or interface C, the symbolic reference to C given by the field reference must first be resolved (§5.4.3.1). Therefore, any exception that can be thrown as a result of failure of resolution of a class or interface reference can be thrown as a result of failure of field resolution. If the reference to C can be successfully resolved, an exception relating to the failure of resolution of the field reference itself can be thrown.

When resolving a field reference, field resolution first attempts to look up the referenced field in C and its superclasses:

1. If C declares a field with the name and descriptor specified by the field reference, field lookup succeeds. The declared field is the result of the field lookup.

2. Otherwise, field lookup is applied recursively to the direct superinterfaces of the specified class or interface C.

3. Otherwise, if C has a superclass S, field lookup is applied recursively to S.

4. Otherwise, field lookup fails.

Then:

- **If field lookup fails, field resolution throws a `NoSuchFieldError`.**

- **Otherwise, if field lookup succeeds but the referenced field is not accessible (§5.4.4) to D, field resolution throws an `IllegalAccessError`.**

- Otherwise, let <E, L_1> be the class or interface in which the referenced field is actually declared and let L_2 be the defining loader of D.
  Given that the type of the referenced field is T_f, let T be T_f if T_f is not an array type, and let T be the element type (§2.4) of T_f otherwise.

  - **The Java Virtual Machine must impose the loading constraint that $T_{L_1} = T_{L_2}$ (§5.3.4).**

  - Let VA_D be the ValueTypes attribute of <D, L_2> and VA_E the ValueTypes attribute of <E, L_1>. **If T is listed in VA_D but is not listed in VA_E or if T is listed in VA_E but is not listed in VA_D, field resolution throws an `IncompatibleClassChangeError`.**

5.4.3.3. Method Resolution

To resolve an unresolved symbolic reference from D to a method in a class C, the symbolic reference to C given by the method reference is first resolved (§5.4.3.1). Therefore, any exception that can be thrown as a result of failure of resolution of a class reference can be thrown as a result of failure of method resolution. If the reference to C can be successfully resolved, exceptions relating to the resolution of the method reference itself can be thrown.

When resolving a method reference:

1. If C is an interface, method resolution throws an `IncompatibleClassChangeError`.

2. Otherwise, method resolution attempts to locate the referenced method in C and its superclasses:

    - If C declares exactly one method with the name specified by the method reference, and the declaration is a signature polymorphic method (§2.9.3), then method lookup succeeds. All the class names mentioned in the descriptor are resolved (§5.4.3.1).
      The resolved method is the signature polymorphic method declaration. It is not necessary for C to declare a method with the descriptor specified by the method reference.

    - Otherwise, if C declares a method with the name and descriptor specified by the method reference, method lookup succeeds.

    - Otherwise, if C has a superclass, step 2 of method resolution is recursively invoked on the direct superclass of C.

3. Otherwise, method resolution attempts to locate the referenced method in the superinterfaces of the specified class C:

    - If the *maximally-specific superinterface methods* of C for the name and descriptor specified by the method reference include exactly one method that does not have its `ACC_ABSTRACT` flag set, then this method is chosen and method lookup succeeds.

    - Otherwise, if any superinterface of C declares a method with the name and descriptor specified by the method reference that has neither its `ACC_PRIVATE` flag nor its `ACC_STATIC` flag set, one of these is arbitrarily chosen and method lookup succeeds.

    - Otherwise, method lookup fails.

5. A *maximally-specific superinterface method* of a class or interface C for a particular method name and descriptor is any method for which all of the following are true:

- The method is declared in a superinterface (direct or indirect) of C.
- The method is declared with the specified name and descriptor.
- The method has neither its `ACC_PRIVATE` flag nor its `ACC_STATIC` flag set.
- Where the method is declared in interface I, there exists no other maximally-specific superinterface method of Cwith the specified name and descriptor that is declared in a subinterface of I.

The result of method resolution is determined by whether method lookup succeeds or fails:

- **If method lookup fails, method resolution throws a `NoSuchMethodError`.**
- **Otherwise, if method lookup succeeds and the referenced method is not accessible (§5.4.4) to D, method resolution throws an `IllegalAccessError`.**
- Otherwise, let <E, L_1> be the class or interface in which the referenced method `m` is actually declared, and let L_2 be the defining loader of D.
  Given that the return type of `m` is T_r, and that the formal parameter types of `m` are T_{f1}, ..., T_{fn}, then:
  If T_r is not an array type, let T_0 be T_r; otherwise, let T_0 be the element type (§2.4) of T_r.
  For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.
  - **The Java Virtual Machine must impose the loading constraints $T_i^{L_1} = T_i^{L_2}$ for *i* = 0 to *n* (§5.3.4).**
  - Let VA_D be the ValueTypes attribute of <D, L_2> and VA_E the ValueTypes attribute of <E, L_1>.
    For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.
    If T_i is listed in VA_D but is not listed in VA_E, or if T_i is listed in VA_E but is not listed in VA_D, method resolution throws an `IncompatibleClassChangeError`.

When resolution searches for a method in the class's superinterfaces, the best outcome is to identify a maximally-specific non-`abstract` method. It is possible that this method will be chosen by method selection, so it is desirable to add class loader constraints for it.

Otherwise, the result is nondeterministic. This is not new: The Java® Virtual Machine Specification has never identified exactly which method is chosen, and how "ties" should be broken. Prior to Java SE 8, this was mostly an unobservable distinction. However, beginning with Java SE 8, the set of interface methods is more heterogenous, so care must be taken to avoid problems with nondeterministic behavior. Thus:

- *Superinterface methods that are `private` and `static` are ignored by resolution. This is consistent with the Java programming language, where such interface methods are not inherited.*
- *Any behavior controlled by the resolved method should not depend on whether the method is `abstract` or not.*

Note that if the result of resolution is an `abstract` method, the referenced class C may be non-`abstract`. Requiring C to be `abstract` would conflict with the nondeterministic choice of superinterface methods. Instead, resolution assumes that the run time class of the invoked object has a concrete implementation of the method.

5.4.3.4. Interface Method Resolution

To resolve an unresolved symbolic reference from D to an interface method in an interface C, the symbolic reference to C given by the interface method reference is first resolved (§5.4.3.1). Therefore, any exception that can be thrown as a result of failure of resolution of an interface reference can be thrown as a result of failure of interface method resolution. If the reference to C can be successfully resolved, exceptions relating to the resolution of the interface method reference itself can be thrown.

When resolving an interface method reference:

1. **If C is not an interface, interface method resolution throws an `IncompatibleClassChangeError`.**
2. Otherwise, if C declares a method with the name and descriptor specified by the interface method reference, method lookup succeeds.
3. Otherwise, if the class `Object` declares a method with the name and descriptor specified by the interface method reference, which has its `ACC_PUBLIC` flag set and does not have its `ACC_STATIC` flag set, method lookup succeeds.
4. Otherwise, if the maximally-specific superinterface methods (§5.4.3.3) of C for the name and descriptor specified by the method reference include exactly one method that does not have its `ACC_ABSTRACT` flag set, then this method is chosen and method lookup succeeds.

5. Otherwise, if any superinterface of C declares a method with the name and descriptor specified by the method reference that has neither its `ACC_PRIVATE` flag nor its `ACC_STATIC` flag set, one of these is arbitrarily chosen and method lookup succeeds.

6. Otherwise, method lookup fails.

The result of interface method resolution is determined by whether method lookup succeeds or fails:

- **If method lookup fails, interface method resolution throws a `NoSuchMethodError`.**

- **If method lookup succeeds and the referenced method is not accessible (§5.4.4) to D, interface method resolution throws an `IllegalAccessError`.**

- Otherwise, let <E, L_1> be the class or interface in which the referenced interface method `m` is actually declared, and let L_2 be the defining loader of D.
  Given that the return type of `m` is T_r, and that the formal parameter types of `m` are T_{f1}, ..., T_{fn}, then:
  If T_r is not an array type, let T_0 be T_r; otherwise, let T_0 be the element type (§2.4) of T_r.
  For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.

  - **The Java Virtual Machine must impose the loading constraints $T_i^{L_1} = T_i^{L_2}$ for *i* = 0 to *n* (§5.3.4).**

  - Let VA_D be the ValueTypes attribute of <D, L_2> and VA_E the ValueTypes attribute of <E, L_1>.
    For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.
    If T_i is listed in VA_D but is not listed in VA_E, or if T_i is listed in VA_E but is not listed in VA_D, method resolution throws an `IncompatibleClassChangeError`.

The clause about accessibility is necessary because interface method resolution may pick a `private` method of interface C. (Prior to Java SE 8, the result of interface method resolution could be a non-`public` method of class `Object` or a `static` method of class `Object`; such results were not consistent with the inheritance model of the Java programming language, and are disallowed in Java SE 8 and above.)

5.5. Initialization

Initialization of a class or interface consists of executing its class or interface initialization method (§2.9).

A class or interface C may be initialized only as a result of:

- The execution of any one of the Java Virtual Machine instructions *new*, *defaultvalue*, *getstatic*, *putstatic*, or *invokestatic* that references C (§*new*, §defaultvalue, §*getstatic*, §*putstatic*, §*invokestatic*). These instructions reference a class or interface directly or indirectly through either a field reference or a method reference.
  Upon execution of a *new* or *defaultvalue* instruction, the referenced class is initialized if it has not been initialized already.
  Upon execution of a *getstatic*, *putstatic*, or *invokestatic* instruction, the class or interface that declared the resolved field or method is initialized if it has not been initialized already.
  Upon execution of a *anewarray*, or *multianewarray*, if the element type of the array is a value class type, the class is resolved and initialized if it has not been initialized already.

- The first invocation of a `java.lang.invoke.MethodHandle` instance which was the result of method handle resolution (§5.4.3.5) for a method handle of kind 2 (`REF_getStatic`), 4 (`REF_putStatic`), 6 (`REF_invokeStatic`), or 8 (`REF_newInvokeSpecial`).
  This implies that the class of a bootstrap method is initialized when the bootstrap method is invoked for an invokedynamic instruction (§invokedynamic), as part of the continuing resolution of the call site specifier.

- Invocation of certain reflective methods in the class library (§2.12), for example, in class `Class` or in package `java.lang.reflect`.

- If C is a class, the initialization of one of its subclasses.

- If C is an interface that declares a non-`abstract`, non-`static` method, the initialization of a class that implements C directly or indirectly.

- If C is a reference type and a containing class B declares a field C with the ACC_FLATTENABLE flag set, C is initialized during step 8 of the initialization process of B.

- If C is a class, its designation as the initial class at Java Virtual Machine startup (§5.2).

Prior to initialization, a class or interface must be linked, that is, verified, prepared, and optionally resolved.

Because the Java Virtual Machine is multithreaded, initialization of a class or interface requires careful synchronization, since some other thread may be trying to initialize the same class or interface at the same time. There is also the possibility that initialization of a class or interface may be requested recursively as part of the initialization of that class or interface. The implementation of the Java Virtual Machine is responsible for taking care of synchronization and recursive initialization by using the following procedure. It assumes that the `Class` object has already been verified and prepared, and that the `Class` object contains state that indicates one of four situations:

- This `Class` object is verified and prepared but not initialized.
- This `Class` object is being initialized by some particular thread.
- This `Class` object is fully initialized and ready for use.
- This `Class` object is in an erroneous state, perhaps because initialization was attempted and failed.

For each class or interface C, there is a unique initialization lock `LC`. The mapping from C to `LC` is left to the discretion of the Java Virtual Machine implementation. For example, `LC` could be the `Class` object for C, or the monitor associated with that `Class` object. The procedure for initializing C is then as follows:

1. Synchronize on the initialization lock, `LC`, for C. This involves waiting until the current thread can acquire `LC`.
2. If the `Class` object for C indicates that initialization is in progress for C by some other thread, then release `LC` and block the current thread until informed that the in-progress initialization has completed, at which time repeat this procedure. Thread interrupt status is unaffected by execution of the initialization procedure.
3. If the `Class` object for C indicates that initialization is in progress for C by the current thread, then this must be a recursive request for initialization.

Release `LC` and complete normally.

4. If the `Class` object for C indicates that C has already been initialized, then no further action is required. Release `LC` and complete normally.

5. **If the `Class` object for C is in an erroneous state, then initialization is not possible. Release `LC` and throw a `NoClassDefFoundError`.**

6. Otherwise, record the fact that initialization of the `Class` object for C is in progress by the current thread, and release `LC`.
   Then, initialize each `final static` field of C with the constant value in its `ConstantValue` attribute (§4.7.2), in the order the fields appear in the `ClassFile` structure.

7. Next, if C is a class rather than an interface, and its superclass has not yet been initialized, then let SC be its superclass and let SI_1, ..., SI_n be all superinterfaces of C (whether direct or indirect) that declare at least one non-`abstract`, non-`static` method. The order of superinterfaces is given by a recursive enumeration over the superinterface hierarchy of each interface directly implemented by C. For each interface I directly implemented by C (in the order of the `interfaces` array of C), the enumeration recurs on I's superinterfaces (in the order of the `interfaces` array of I) before returning I.
   For each S in the list [SC, SI_1, ..., SI_n], recursively perform this entire procedure for S. If necessary, verify and prepare S first.
   If the initialization of S completes abruptly because of a thrown exception, then acquire `LC`, label the `Class` object for C as erroneous, notify all waiting threads, release `LC`, and complete abruptly, throwing the same exception that resulted from initializing SC.

8. If C has any field F declared with a reference type FC and the ACC_FLATTENABLE flag set, and FC has not yet been initialized, then this entire procedure is performed for FC, even if F is not flattened. This step might require to resolve, load and prepare FC if FC has not yet been resolved, loaded or prepared.

   If the initialization of FC completes abruptly because of a thrown exception, then acquire `LC`, label the `Class` object for C as erroneous, notify all waiting threads, release `LC`, and complete abruptly, throwing the same exception that resulted from initializing FC.

9. Next, determine whether assertions are enabled for C by querying its defining class loader.

10. Next, execute the class or interface initialization method of C.

11. If the execution of the class or interface initialization method completes normally, then acquire `LC`, label the `Class` object for C as fully initialized, notify all waiting threads, release `LC`, and complete this procedure normally.

12. **Otherwise, the class or interface initialization method must have completed abruptly by throwing some exception E. If the class of E is not `Error` or one of its subclasses, then create a new instance of the class `ExceptionInInitializerError` with E as the argument, and use this object in place of E in the following step. If a new instance of `ExceptionInInitializerError` cannot be created because an`OutOfMemoryError` occurs, then use an `OutOfMemoryError` object in place of E in the following step.**

13. **Acquire `LC`, label the `Class` object for C as erroneous, notify all waiting threads, release `LC`, and complete this procedure abruptly with reason E or its replacement as determined in the previous step.**

A Java Virtual Machine implementation may optimize this procedure by eliding the lock acquisition in step 1 (and release in step 4/5) when it can determine that the initialization of the class has already completed, provided that, in terms of the Java memory model, all *happens-before* orderings (JLS §17.4.5) that would exist if the lock were acquired, still exist when the optimization is performed.

6.5. Instructions

aastore

Operation

Store into `reference` array

Format

```
aastore
```

Forms

aastore = 83 (0x53)

Operand Stack

..., *arrayref*, *index*, *value* →

...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `reference`. The *index* must be of type `int` and value must be of type `reference`. The *arrayref*, *index*, and *value* are popped from the operand stack. The `reference` *value* is stored as the component of the array at *index*.

At run time, the type of *value* must be compatible with the type of the components of the array referenced by *arrayref*. Specifically, assignment of a value of reference type S (source) to an array component of reference type T (target) is allowed only if:

- If S is a class type, then:
    - If T is a class type, then S must be the same class as T, or S must be a subclass of T;
    - If T is an interface type, then S must implement interface T.
- If S is an interface type, then:
    - If T is a class type, then T must be `Object`.
    - If T is an interface type, then T must be the same interface as S or a superinterface of S.
- If S is an array type, namely, the type SC`[]`, that is, an array of components of type SC, then:
    - If T is a class type, then T must be `Object`.
    - If T is an interface type, then T must be one of the interfaces implemented by arrays (JLS §4.10.3).
    - If T is an array type TC`[]`, that is, an array of components of type TC, then one of the following must be true:

- TC and SC are the same primitive type.
- TC and SC are reference types, and type SC is assignable to TC by these run-time rules.

Run-time Exceptions

If *arrayref* is `null`, *aastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aastore* instruction throws an `ArrayIndexOutOfBoundsException`.

Otherwise, if the actual type of the components of the array is a value class type, and *value* is the `null` reference, aastore throws a `NullPointerException`.

Otherwise, if *arrayref* is not `null` and the actual type of *value* is not assignment compatible (JLS §5.2) with the actual type of the components of the array, *aastore* throws an `ArrayStoreException`.

anewarray

Operation

Create new array of `reference`

Format

```
anewarray
indexbyte1
indexbyte2
```

Forms

anewarray = 189 (0xbd)

Operand Stack

..., *count* →

..., *arrayref*

Description

The *count* must be of type `int`. It is popped off the operand stack. The *count* represents the number of components of the array to be created. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). A new array with components of that type, of length *count*, is allocated from the garbage-collected heap, and a `reference` *arrayref* to this new array object is pushed onto the operand stack. If the type of the components is a value class type, all components of the new array are initialized to the default value of this value class (§2.4). Otherwise, all components of the new array are initialized to `null`, the default value for `reference` types (§2.4).

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

Run-time Exceptions

Otherwise, if *count* is less than zero, the *anewarray* instruction throws a `NegativeArraySizeException`.

Notes

The *anewarray* instruction is used to create a single dimension of an array of object references or part of a multidimensional array.

if_acmp<cond>

Operation

Branch if `reference` comparison succeeds

Format

```
if_acmp<cond>
branchbyte1
```

branchbyte2

Forms

if_acmpeq = 165 (0xa5)

if_acmpne = 166 (0xa6)

Operand Stack

..., *value1*, *value2* →

...

Description

Both *value1* and *value2* must be of type `reference`. They are both popped from the operand stack and compared. The results of the comparison are as follows:

- *if_acmpeq* succeeds if and only if *value1* = *value2* and neither is an instance of a value class.
- *if_acmpne* succeeds if and only if *value1* ≠ *value2* or either is an instance of a value class.

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if_acmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_acmp<cond>* instruction.

Otherwise, if the comparison fails, execution proceeds at the address of the instruction following this *if_acmp<cond>*instruction.

monitorenter

Operation

Enter monitor for object

Format

monitorenter

Forms

monitorenter = 194 (0xc2)

Operand Stack

..., *objectref* →

...

Description

The *objectref* must be of type `reference`.

Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes*monitorenter* attempts to gain ownership of the monitor associated with *objectref*, as follows:

- If the entry count of the monitor associated with *objectref* is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.

- If the thread already owns the monitor associated with *objectref*, it reenters the monitor, incrementing its entry count.

- If another thread already owns the monitor associated with *objectref*, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

Run-time Exception

If *objectref* is `null`, *monitorenter* throws a `NullPointerException`.

Otherwise, if *objectref* is an instance of a value class , *monitorenter* throws an `IllegalMonitorStateException`.

Notes

A *monitorenter* instruction may be used with one or more *monitorexit* instructions (§*monitorexit*) to implement `synchronized` statement in the Java programming language (§3.14). The *monitorenter* and *monitorexit* instructions are not used in the

implementation of `synchronized` methods, although they can be used to provide equivalent locking semantics. Monitor entry on invocation of a `synchronized` method, and monitor exit on its return, are handled implicitly by the Java Virtual Machine's method invocation and return instructions, as if *monitorenter* and *monitorexit* were used.

The association of a monitor with an object may be managed in various ways that are beyond the scope of this specification. For instance, the monitor may be allocated and deallocated at the same time as the object. Alternatively, it may be dynamically allocated at the time when a thread attempts to gain exclusive access to the object and freed at some later time when no thread remains in the monitor for the object.

The synchronization constructs of the Java programming language require support for operations on monitors besides entry and exit. These include waiting on a monitor (`Object.wait`) and notifying other threads waiting on a monitor (`Object.notifyAll` and `Object.notify`). These operations are supported in the standard package `java.lang` supplied with the Java Virtual Machine. No explicit support for these operations appears in the instruction set of the Java Virtual Machine.

monitorexit

Operation

Exit monitor for object

Format

`monitorexit`

Forms

monitorexit = 195 (0xc3)

Operand Stack

..., *objectref* →

...

Description

The *objectref* must be of type `reference`.

The thread that executes *monitorexit* must be the owner of the monitor associated with the instance referenced by*objectref*.

The thread decrements the entry count of the monitor associated with *objectref*. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

Run-time Exceptions

If *objectref* is `null`, *monitorexit* throws a `NullPointerException`.

Otherwise, if *objectref* is an instance of a value class , *monitorexit* throws a `IllegalMonitorStateException`.

Otherwise, if the thread that executes *monitorexit* is not the owner of the monitor associated with the instance referenced by *objectref*, *monitorexit* throws an `IllegalMonitorStateException`.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the second of those rules is violated by the execution of this *monitorexit* instruction, then *monitorexit* throws an `IllegalMonitorStateException`.

Notes

One or more *monitorexit* instructions may be used with a *monitorenter* instruction (§*monitorenter*) to implement a `synchronized` statement in the Java programming language (§3.14). The *monitorenter* and *monitorexit* instructions are not used in the implementation of `synchronized` methods, although they can be used to provide equivalent locking semantics.

The Java Virtual Machine supports exceptions thrown within `synchronized` methods and `synchronized` statements differently:

- Monitor exit on normal `synchronized` method completion is handled by the Java Virtual Machine's return instructions. Monitor exit on abrupt `synchronized` method completion is handled implicitly by the Java Virtual Machine's *athrow* instruction.

- When an exception is thrown from within a `synchronized` statement, exit from the monitor entered prior to the execution of the `synchronized` statement is achieved using the Java Virtual Machine's exception handling mechanism (§3.14).

multianewarray

Operation

Create new multidimensional array

Format

```
multianewarray
indexbyte1
indexbyte2
dimensions
```

Forms

multianewarray = 197 (0xc5)

Operand Stack

..., *count1*, [*count2*, ...] →

..., *arrayref*

Description

The *dimensions* operand is an unsigned byte that must be greater than or equal to 1. It represents the number of dimensions of the array to be created. The operand stack must contain *dimensions* values. Each such value represents the number of components in a dimension of the array to be created, must be of type `int`, and must be non-negative. The *count1* is the desired length in the first dimension, *count2* in the second, etc.

All of the *count* values are popped off the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). The resulting entry must be an array class type of dimensionality greater than or equal to *dimensions*.

A new multidimensional array of the array type is allocated from the garbage-collected heap. If any *count* value is zero, no subsequent dimensions are allocated. The components of the array in the first dimension are initialized to subarrays of the type of the second dimension, and so on. If the type of the components of the last allocated dimension of the array is a value class type, all components are initialized to the default

value of this value class. Otherwise, they are initialized to the default initial value (§2.3, §2.4) for the element type of the array type. A `reference` *arrayref* to the new array is pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

Otherwise, if the current class does not have permission to access the element type of the resolved array class, *multianewarray* throws an `IllegalAccessError`.

Run-time Exception

Otherwise, if any of the *dimensions* values on the operand stack are less than zero, the *multianewarray*instruction throws a `NegativeArraySizeException`.

Notes

It may be more efficient to use *newarray* or *anewarray* (§*newarray*, §*anewarray*) when creating an array of a single dimension.

The array class referenced via the run-time constant pool may have more dimensions than the *dimensions* operand of the *multianewarray* instruction. In that case, only the first *dimensions* of the dimensions of the array are created.

new

Operation

Create new object

Format

```
new
indexbyte1
indexbyte2
```

Forms

new = 187 (0xbb)

Operand Stack

... →

..., *objectref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in an object class type. Memory for a new instance of that object class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized according to their declaration (§4.5), using the default initial values (§2.3, §2.4). The *objectref*, a `reference` to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

Otherwise, if the symbolic reference to the class, array, or interface type resolves to an interface or is an `abstract` class or a value class, *new* throws an `InstantiationError`.

Run-time Exception

Otherwise, if execution of this *new* instruction causes initialization of the referenced class, *new* may throw an `Error` as detailed in JLS §15.9.4.

Notes

The *new* instruction does not completely create a new instance; instance creation is not completed until an instance initialization method (§2.9) has been invoked on the uninitialized instance.

putfield

Operation

Set field in object

Format

```
putfield
indexbyte1
indexbyte2
```

Forms

putfield = 181 (0xb5)

Operand Stack

..., *objectref*, *value* →

...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The class of *objectref* must not be an array nor a value class. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

The referenced field is resolved (§5.4.3.2). The type of a *value* stored by a *putfield* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class, and the instruction must occur in an instance initialization method (`<init>`) of the current class (§2.9).

The *value* and *objectref* are popped from the operand stack. The *objectref* must be of type `reference`. The *value* undergoes value set conversion (§2.8.3), resulting in *value*', and the referenced field in *objectref* is set to *value*'.

Linking Exceptions

During resolution of the symbolic reference to the field, any of the exceptions pertaining to field resolution (§5.4.3.2) can be thrown.

Otherwise, if the resolved field is a `static` field, *putfield* throws an `IncompatibleClassChangeError`.

Otherwise, if the resolved field is a field of a value class, *putfield* throws an `IllegalAccessError`.

Otherwise, if the field is `final`, it must be declared in the current class, and the instruction must occur in an instance initialization method (`<init>`) of the current class. Otherwise, an `IllegalAccessError` is thrown.

Run-time Exception

Otherwise, if *objectref* is `null`, the *putfield* instruction throws a `NullPointerException`.

Otherwise, if the field has been declared with `ACC_FLATTENABLE` flag set and the value is the `null` reference, a `NullPointerException` is thrown.

putstatic

Operation

Set static field in class

Format

```
putstatic
indexbyte1
indexbyte2
```

Forms

putstatic = 179 (0xb3)

Operand Stack

..., *value* →

...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.2).

On successful resolution of the field, the class or interface that declared the resolved field is initialized (§5.5) if that class or interface has not already been initialized.

The type of a *value* stored by a *putstatic* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class, and the instruction must occur in the `<clinit>` method of the current class (§2.9).

The *value* is popped from the operand stack and undergoes value set conversion (§2.8.3), resulting in *value*'. The class field is set to *value*'.

Linking Exceptions

During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution (§5.4.3.2) can be thrown.

Otherwise, if the resolved field is not a `static` (class) field or an interface field, *putstatic* throws an `IncompatibleClassChangeError`.

Otherwise, if the field is `final`, it must be declared in the current class, and the instruction must occur in the `<clinit>` method of the current class. Otherwise, an `IllegalAccessError` is thrown.

Run-time Exception

Otherwise, if execution of this *putstatic* instruction causes initialization of the referenced class or interface, *putstatic* may throw an `Error` as detailed in §5.5.

Otherwise, if the field has been declared with `ACC_FLATTENABLE` flag set and the value is the `null` reference, a `NullPointerException` is thrown.

Notes

A *putstatic* instruction may be used only to set the value of an interface field on the initialization of that field. Interface fields may be assigned to only once, on execution of an interface variable initialization expression when the interface is initialized (§5.5, JLS §9.3.1).

defaultvalue

Operation

Provide a value class instance with all its instance variables set to their default value.

Format

```
defaultvalue
indexbyte1
indexbyte2
```

Forms

defaultvalue = 203 (0xcb)

Operand Stack

... →

…, *valueref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a value class. The named value class is resolved (§5.4.3.1) and should result in a value class type. A *valueref*, a `reference` to an instance of the value class, is pushed onto the operand stack. The *valueref* can point to either a newly allocated instance or an existing instance, either stored in or outside of the Java heap. All instance variables of this instance must have their values set according to their declaration (§4.5), and the default initial values (§2.3, §2.4).

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

Otherwise, if the symbolic reference to the class, array, or interface type resolves to an interface or is an `abstract` class, *defaultvalue* throws an `InstantiationError`.

Otherwise, if the symbolic reference to the class, array, or interface type resolves to an object class, *defaultvalue* throws an `IncompatibleClassChangeError`.

Run-time Exception

Otherwise, if execution of this *defaultvalue* instruction causes initialization of the referenced class, *defaultvalue* may throw an `Error` as detailed in JLS §15.9.4.

Notes

The *defaultvalue* instruction does provide a completely initialized instance; known as the default value of the value class.

withfield

Operation

Return an instance of a value class with an updated instance field.

Format

```
withfield
indexbyte1
indexbyte2
```

Forms

withtfield = 204 (0xcc)

Operand Stack

..., *valueref*, *value* →

…, *newvalueref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the value class in which the field is to be found.

The referenced field is resolved (§5.4.3.2). The type of a *value* stored by a *withfield* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type.

The *value* and *valueref* are popped from the operand stack. The *valueref* must be of type `reference`. The *value* undergoes value set conversion (§2.8.3), resulting in *value*', a copy of *valueref* is created, resulting in *newvalueref* and the referenced field in *newvalueref* is set to *value*' before *newvalueref* is pushed on the stack.

Linking Exceptions

During resolution of the symbolic reference to the field, any of the exceptions pertaining to field resolution (§5.4.3.2) can be thrown.

Otherwise, if the resolved field is a `static` field, *withfield* throws an `IncompatibleClassChangeError`.

Otherwise, if the resolved field is a field of an object class, *withfield* throws an `IncompatibleClassChangeError`.

The field must be `final`, it must be declared in the current value class, and the instruction must occur in a method of the current value class. Otherwise, an `IllegalAccessError` is thrown.

Run-time Exception

Otherwise, if *valueref* is `null`, the *withfield* instruction throws a `NullPointerException`.

Otherwise, if the resolved field has been declared with the `ACC_FLATTENABLE` flag set and the *value* is the null reference, a `NullPointerException` is thrown.

Legal Notice

Notice

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Oracle America, Inc. ("Oracle") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Oracle hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Oracle's intellectual property rights to:

1. *Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.*

2. *Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:*

    i. *does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;*

    ii. *is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and*

    iii. *includes the following notice: "This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."*

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Oracle intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Oracle if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof.

Trademarks

No right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

Disclaimer of Warranties

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY ORACLE. ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. ORACLE MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

Limitation of Liability

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ORACLE OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF ORACLE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Oracle (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

Restricted Rights Legend

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

Report

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Oracle with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

General Terms

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.