**Chapter 5. Loading, Linking, and Initializing**

The Java Virtual Machine dynamically loads, links and initializes classes and interfaces. Loading is the process of finding the binary representation of a class or interface type with a particular name and *creating* a class or interface from that binary representation. Linking is the process of taking a class or interface and combining it into the run-time state of the Java Virtual Machine so that it can be executed. Initialization of a class or interface consists of executing the class or interface initialization method <clinit> ([§2.9](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.9)).

In this chapter, [§5.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.1) describes how the Java Virtual Machine derives symbolic references from the binary representation of a class or interface. [§5.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.2) explains how the processes of loading, linking, and initialization are first initiated by the Java Virtual Machine. [§5.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3) specifies how binary representations of classes and interfaces are loaded by class loaders and how classes and interfaces are created. Linking is described in [§5.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4). [§5.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.5) details how classes and interfaces are initialized. [§5.6](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.6)introduces the notion of binding native methods. Finally, [§5.7](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.7) describes when a Java Virtual Machine exits.

**5.1. The Run-Time Constant Pool**

The Java Virtual Machine maintains a per-type constant pool ([§2.5.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.5.5)), a run-time data structure that serves many of the purposes of the symbol table of a conventional programming language implementation.

The constant\_pool table ([§4.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4)) in the binary representation of a class or interface is used to construct the run-time constant pool upon class or interface creation ([§5.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3)). All references in the run-time constant pool are initially symbolic. The symbolic references in the run-time constant pool are derived from structures in the binary representation of the class or interface as follows:

* A symbolic reference to a class or interface is derived from a CONSTANT\_Class\_info structure ([§4.4.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.1)) in the binary representation of a class or interface. Such a reference gives the name of the class or interface in the form returned by the Class.getName method, that is:
  + For a nonarray class or an interface, the name is the binary name ([§4.2.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.2.1)) of the class or interface.
  + For an array class of *n* dimensions, the name begins with *n* occurrences of the ASCII "[" character followed by a representation of the element type:
    - If the element type is a primitive type, it is represented by the corresponding field descriptor ([§4.3.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.2)).
    - Otherwise, if the element type is a reference type, it is represented by the ASCII "L" character followed by the binary name ([§4.2.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.2.1)) of the element type followed by the ASCII ";" character.

Whenever this chapter refers to the name of a class or interface, it should be understood to be in the form returned by the Class.getName method.

* A symbolic reference to a field of a class or an interface is derived from a CONSTANT\_Fieldref\_info structure ([§4.4.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.2)) in the binary representation of a class or interface. Such a reference 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.
* A symbolic reference to a method of a class is derived from a CONSTANT\_Methodref\_info structure ([§4.4.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.2)) in the binary representation of a class or interface. Such a reference gives the name and descriptor of the method, as well as a symbolic reference to the class in which the method is to be found.
* A symbolic reference to a method of an interface is derived from a CONSTANT\_InterfaceMethodref\_info structure ([§4.4.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.2)) in the binary representation of a class or interface. Such a reference gives the name and descriptor of the interface method, as well as a symbolic reference to the interface in which the method is to be found.
* A symbolic reference to a method handle is derived from a CONSTANT\_MethodHandle\_info structure ([§4.4.8](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.8)) in the binary representation of a class or interface.
* A symbolic reference to a method type is derived from a CONSTANT\_MethodType\_info structure ([§4.4.9](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.9)) in the binary representation of a class or interface.
* A symbolic reference to a *call site specifier* is derived from a CONSTANT\_InvokeDynamic\_info structure ([§4.4.10](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.10)) in the binary representation of a class or interface. Such a reference gives:
  + a symbolic reference to a method handle, which will serve as a bootstrap method for an *invokedynamic* instruction ([§*invokedynamic*](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.invokedynamic));
  + a sequence of symbolic references (to classes, method types, and method handles), string literals, and run-time constant values which will serve as *static arguments* to a bootstrap method;
  + a method name and method descriptor.

In addition, certain run-time values which are not symbolic references are derived from items found in the constant\_pool table:

* A string literal is a reference to an instance of class String, and is derived from a CONSTANT\_String\_info structure ([§4.4.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.3)) in the binary representation of a class or interface. The CONSTANT\_String\_info structure gives the sequence of Unicode code points constituting the string literal.

The Java programming language requires that identical string literals (that is, literals that contain the same sequence of code points) must refer to the same instance of class String (JLS §3.10.5). In addition, if the method String.intern is called on any string, the result is a reference to the same class instance that would be returned if that string appeared as a literal. Thus, the following expression must have the value true:

("a" + "b" + "c").intern() == "abc"

To derive a string literal, the Java Virtual Machine examines the sequence of code points given by the CONSTANT\_String\_info structure.

* + If the method String.intern has previously been called on an instance of class String containing a sequence of Unicode code points identical to that given by the CONSTANT\_String\_info structure, then the result of string literal derivation is a reference to that same instance of class String.
  + Otherwise, a new instance of class String is created containing the sequence of Unicode code points given by the CONSTANT\_String\_info structure; areference to that class instance is the result of string literal derivation. Finally, the intern method of the new String instance is invoked.
* Run-time constant values are derived from CONSTANT\_Integer\_info, CONSTANT\_Float\_info, CONSTANT\_Long\_info, or CONSTANT\_Double\_info structures ([§4.4.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.4), [§4.4.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.5)) in the binary representation of a class or interface.

Note that CONSTANT\_Float\_info structures represent values in IEEE 754 single format and CONSTANT\_Double\_info structures represent values in IEEE 754 double format ([§4.4.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.4), [§4.4.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.5)). The run-time constant values derived from these structures must thus be values that can be represented using IEEE 754 single and double formats, respectively.

The remaining structures in the constant\_pool table of the binary representation of a class or interface - the CONSTANT\_NameAndType\_info andCONSTANT\_Utf8\_info structures ([§4.4.6](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.6), [§4.4.7](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.7)) - are only used indirectly when deriving symbolic references to classes, interfaces, methods, fields, method types, and method handles, and when deriving string literals and call site specifiers.

**5.2. Java Virtual Machine Startup**

The Java Virtual Machine starts up by creating an initial class, which is specified in an implementation-dependent manner, using the bootstrap class loader ([§5.3.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.1)). The Java Virtual Machine then links the initial class, initializes it, and invokes the public class method void main(String[]). The invocation of this method drives all further execution. Execution of the Java Virtual Machine instructions constituting the main method may cause linking (and consequently creation) of additional classes and interfaces, as well as invocation of additional methods.

In an implementation of the Java Virtual Machine, the initial class could be provided as a command line argument. Alternatively, the implementation could provide an initial class that sets up a class loader which in turn loads an application. Other choices of the initial class are possible so long as they are consistent with the specification given in the previous paragraph.

**5.3. Creation and Loading**

Creation of a class or interface C denoted by the name N consists of the construction in the method area of the Java Virtual Machine ([§2.5.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.5.4)) of an implementation-specific internal representation of C. Class or interface creation is triggered by another class or interface D, which references C through its run-time constant pool. Class or interface creation may also be triggered by D invoking methods in certain Java SE platform class libraries ([§2.12](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.12)) such as reflection.

If C is not an array class, it is created by loading a binary representation of C ([§4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html)) using a class loader. Array classes do not have an external binary representation; they are created by the Java Virtual Machine rather than by a class loader.

There are two kinds of class loaders: the bootstrap class loader supplied by the Java Virtual Machine, and user-defined class loaders. Every user-defined class loader is an instance of a subclass of the abstract class ClassLoader. Applications employ user-defined class loaders in order to extend the manner in which the Java Virtual Machine dynamically loads and thereby creates classes. User-defined class loaders can be used to create classes that originate from user-defined sources. For example, a class could be downloaded across a network, generated on the fly, or extracted from an encrypted file.

A class loader L may create C by defining it directly or by delegating to another class loader. If L creates C directly, we say that L *defines* C or, equivalently, that L is the *defining loader* of C.

When one class loader delegates to another class loader, the loader that initiates the loading is not necessarily the same loader that completes the loading and defines the class. If L creates C, either by defining it directly or by delegation, we say that L initiates loading of C or, equivalently, that L is an *initiating loader* of C.

At run time, a class or interface is determined not by its name alone, but by a pair: its binary name ([§4.2.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.2.1)) and its defining class loader. Each such class or interface belongs to a single *run-time package*. The run-time package of a class or interface is determined by the package name and defining class loader of the class or interface.

The Java Virtual Machine uses one of three procedures to create class or interface C denoted by N:

* If N denotes a nonarray class or an interface, one of the two following methods is used to load and thereby create C:
  + If D was defined by the bootstrap class loader, then the bootstrap class loader initiates loading of C ([§5.3.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.1)).
  + If D was defined by a user-defined class loader, then that same user-defined class loader initiates loading of C ([§5.3.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.2)).
* Otherwise N denotes an array class. An array class is created directly by the Java Virtual Machine ([§5.3.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.3)), not by a class loader. However, the defining class loader of D is used in the process of creating array class C.

**If an error occurs during class loading, then an instance of a subclass of LinkageError must be thrown at a point in the program that (directly or indirectly) uses the class or interface being loaded.**

**If the Java Virtual Machine ever attempts to load a class C during verification (**[**§5.4.1**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.1)**) or resolution (**[**§5.4.3**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3)**) (but not initialization (**[**§5.5**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.5)**)), and the class loader that is used to initiate loading of C throws an instance of ClassNotFoundException, then the Java Virtual Machine must throw an instance ofNoClassDefFoundError whose cause is the instance of ClassNotFoundException.**

**(A subtlety here is that recursive class loading to load superclasses is performed as part of resolution (**[**§5.3.5**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.5)**, step 3). Therefore, a ClassNotFoundExceptionthat results from a class loader failing to load a superclass must be wrapped in a NoClassDefFoundError.)**

*A well-behaved class loader should maintain three properties:*

* *Given the same name, a good class loader should always return the same Class object.*
* *If a class loader L1 delegates loading of a class C to another loader L2, then for any type T that occurs as the direct superclass or a direct superinterface of C, or as the type of a field in C, or as the type of a formal parameter of a method or constructor in C, or as a return type of a method in C, L1 and L2 should return the same Class object.*
* *If a user-defined classloader prefetches binary representations of classes and interfaces, or loads a group of related classes together, then it must reflect loading errors only at points in the program where they could have arisen without prefetching or group loading.*

We will sometimes represent a class or interface using the notation <N, Ld>, where N denotes the name of the class or interface and Ld denotes the defining loader of the class or interface.

We will also represent a class or interface using the notation NLi, where N denotes the name of the class or interface and Li denotes an initiating loader of the class or interface.

**5.3.1. Loading Using the Bootstrap Class Loader**

The following steps are used to load and thereby create the nonarray class or interface C denoted by N using the bootstrap class loader.

First, the Java Virtual Machine determines whether the bootstrap class loader has already been recorded as an initiating loader of a class or interface denoted by N. If so, this class or interface is C, and no class creation is necessary.

Otherwise, the Java Virtual Machine passes the argument N to an invocation of a method on the bootstrap class loader to search for a purported representation of Cin a platform-dependent manner. Typically, a class or interface will be represented using a file in a hierarchical file system, and the name of the class or interface will be encoded in the pathname of the file.

Note that there is no guarantee that a purported representation found is valid or is a representation of C. This phase of loading must detect the following error:

* **If no purported representation of C is found, loading throws an instance of ClassNotFoundException.**

Then the Java Virtual Machine attempts to derive a class denoted by N using the bootstrap class loader from the purported representation using the algorithm found in [§5.3.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.5). That class is C.

**5.3.2. Loading Using a User-defined Class Loader**

The following steps are used to load and thereby create the nonarray class or interface C denoted by N using a user-defined class loader L.

First, the Java Virtual Machine determines whether L has already been recorded as an initiating loader of a class or interface denoted by N. If so, this class or interface is C, and no class creation is necessary.

Otherwise, the Java Virtual Machine invokes loadClass(N) on L. The value returned by the invocation is the created class or interface C. The Java Virtual Machine then records that L is an initiating loader of C ([§5.3.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.4)). The remainder of this section describes this process in more detail.

When the loadClass method of the class loader L is invoked with the name N of a class or interface C to be loaded, L must perform one of the following two operations in order to load C:

1. The class loader L can create an array of bytes representing C as the bytes of a ClassFile structure ([§4.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.1)); it then must invoke the method defineClass of classClassLoader. Invoking defineClass causes the Java Virtual Machine to derive a class or interface denoted by N using L from the array of bytes using the algorithm found in [§5.3.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.5).
2. The class loader L can delegate the loading of C to some other class loader L'. This is accomplished by passing the argument N directly or indirectly to an invocation of a method on L' (typically the loadClass method). The result of the invocation is C.

**In either (1) or (2), if the class loader L is unable to load a class or interface denoted by N for any reason, it must throw an instance ofClassNotFoundException.**

*Since JDK release 1.1, Oracle’s Java Virtual Machine implementation has invoked the loadClass method of a class loader in order to cause it to load a class or interface. The argument toloadClass is the name of the class or interface to be loaded. There is also a two-argument version of the loadClass method, where the second argument is a boolean that indicates whether the class or interface is to be linked or not. Only the two-argument version was supplied in JDK release 1.0.2, and Oracle’s Java Virtual Machine implementation relied on it to link the loaded class or interface. From JDK release 1.1 onward, Oracle’s Java Virtual Machine implementation links the class or interface directly, without relying on the class loader.*

**5.3.3. Creating Array Classes**

The following steps are used to create the array class C denoted by N using class loader L. Class loader L may be either the bootstrap class loader or a user-defined class loader.

If L has already been recorded as an initiating loader of an array class with the same component type as N, that class is C, and no array class creation is necessary.

Otherwise, the following steps are performed to create C:

1. If the component type is a reference type, the algorithm of this section ([§5.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3)) is applied recursively using class loader L in order to load and thereby create the component type of C.
2. The Java Virtual Machine creates a new array class with the indicated component type and number of dimensions.

If the component type is a reference type, C is marked as having been defined by the defining class loader of the component type. Otherwise, C is marked as having been defined by the bootstrap class loader.

In any case, the Java Virtual Machine then records that L is an initiating loader for C ([§5.3.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.4)).

If the component type is a reference type, the accessibility of the array class is determined by the accessibility of its component type. Otherwise, the accessibility of the array class is public.

**5.3.4. Loading Constraints**

Ensuring type safe linkage in the presence of class loaders requires special care. It is possible that when two different class loaders initiate loading of a class or interface denoted by N, the name N may denote a different class or interface in each loader.

When a class or interface C = <N1, L1> makes a symbolic reference to a field or method of another class or interface D = <N2, L2>, the symbolic reference includes a descriptor specifying the type of the field, or the return and argument types of the method. It is essential that any type name N mentioned in the field or method descriptor denote the same class or interface when loaded by L1 and when loaded by L2.

**To ensure this, the Java Virtual Machine imposes *loading constraints* of the form NL1 = NL2 during preparation (**[**§5.4.2**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.2)**) and resolution (**[**§5.4.3**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3)**). To enforce these constraints, the Java Virtual Machine will, at certain prescribed times (see**[**§5.3.1**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.1)**,**[**§5.3.2**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.2)**,**[**§5.3.3**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.3)**, and**[**§5.3.5**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.5)**), record that a particular loader is an initiating loader of a particular class. After recording that a loader is an initiating loader of a class, the Java Virtual Machine must immediately check to see if any loading constraints are violated. If so, the record is retracted, the Java Virtual Machine throws a LinkageError, and the loading operation that caused the recording to take place fails.**

**Similarly, after imposing a loading constraint (see**[**§5.4.2**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.2)**,**[**§5.4.3.2**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.2)**,**[**§5.4.3.3**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.3)**, and**[**§5.4.3.4**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.4)**), the Java Virtual Machine must immediately check to see if any loading constraints are violated. If so, the newly imposed loading constraint is retracted, the Java Virtual Machine throws a LinkageError, and the operation that caused the constraint to be imposed (either resolution or preparation, as the case may be) fails.**

The situations described here are the only times at which the Java Virtual Machine checks whether any loading constraints have been violated. A loading constraint is violated if, and only if, all the following four conditions hold:

* There exists a loader L such that L has been recorded by the Java Virtual Machine as an initiating loader of a class C named N.
* There exists a loader L' such that L' has been recorded by the Java Virtual Machine as an initiating loader of a class C ' named N.
* The equivalence relation defined by the (transitive closure of the) set of imposed constraints implies NL = NL'.
* C ≠ C '.

*A full discussion of class loaders and type safety is beyond the scope of this specification. For a more comprehensive discussion, readers are referred to Dynamic Class Loading in the Java Virtual Machine by Sheng Liang and Gilad Bracha (Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications).*

**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**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.1)**,**[**§4.8**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.8)**), loading throws an instance of ClassFormatError.**
  + **Otherwise, if the purported representation is not of a supported major or minor version (**[**§4.1**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.1)**), loading throws an instance ofUnsupportedClassVersionError.**

*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.**

1. 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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.1). Note that if C is an interface 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, loading throws an IncompatibleClassChangeError.**
  + **Otherwise, if any of the superclasses of C is C itself, loading throws a ClassCircularityError.**

1. 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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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.**

1. 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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.4)).

**5.4. Linking**

Linking a class or interface involves verifying and preparing that class or interface, its direct superclass, its direct superinterfaces, and its element type (if it is an array type), if necessary. Resolution of symbolic references in the class or interface is an optional part of linking.

This specification allows an implementation flexibility as to when linking activities (and, because of recursion, loading) take place, provided that all of the following properties are maintained:

* A class or interface is completely loaded before it is linked.
* A class or interface is completely verified and prepared before it is initialized.
* Errors detected during linkage are thrown at a point in the program where some action is taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error.

For example, a Java Virtual Machine implementation may choose to resolve each symbolic reference in a class or interface individually when it is used ("lazy" or "late" resolution), or to resolve them all at once when the class is being verified ("eager" or "static" resolution). This means that the resolution process may continue, in some implementations, after a class or interface has been initialized. Whichever strategy is followed, any error detected during resolution must be thrown at a point in the program that (directly or indirectly) uses a symbolic reference to the class or interface.

**Because linking involves the allocation of new data structures, it may fail with an OutOfMemoryError.**

**5.4.1. Verification**

*Verification* ([§4.10](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10)) ensures that the binary representation of a class or interface is structurally correct ([§4.9](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.9)). Verification may cause additional classes and interfaces to be loaded ([§5.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3)) but need not cause them to be verified or prepared.

**If the binary representation of a class or interface does not satisfy the static or structural constraints listed in**[**§4.9**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.9)**, then a VerifyError must be thrown at the point in the program that caused the class or interface to be verified.**

**If an attempt by the Java Virtual Machine to verify a class or interface fails because an error is thrown that is an instance of LinkageError (or a subclass), then subsequent attempts to verify the class or interface always fail with the same error that was thrown as a result of the initial verification attempt.**

**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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.3), [§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.5)), not preparation.

During preparation of a class or interface C, the Java Virtual Machine also imposes loading constraints ([§5.3.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.4)). Let L1 be the defining loader of C. For each methodm declared in C that overrides ([§5.4.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.5)) a method declared in a superclass or superinterface <D, L2>, the Java Virtual Machine imposes the following loading constraints:

Given that the return type of m is Tr, and that the formal parameter types of m are Tf1, ..., Tfn, then:

If Tr not an array type, let T0 be Tr; otherwise, let T0 be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tr.

For *i* = 1 to *n*: If Tfi is not an array type, let Ti be Tfi; otherwise, let Ti be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tfi.

Then TiL1 = TiL2 for *i* = 0 to *n*.

Furthermore, if C implements a method m declared in a superinterface <I, L3> of C, but C does not itself declare the method m, then let <D, L2> be the superclass of Cthat 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 Tr, and that the formal parameter types of m are Tf1, ..., Tfn, then:

If Tr not an array type, let T0 be Tr; otherwise, let T0 be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tr.

For *i* = 1 to *n*: If Tfi is not an array type, let Ti be Tfi; otherwise, let Ti be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tfi.

Then TiL2 = TiL3 for *i* = 0 to *n*.

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*, *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*,*ldc*, *ldc\_w*, *multianewarray*, *new*, *putfield*, 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.**

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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.2) but also check that the field is notstatic. 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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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.**

1. If C is an array class and its element type is a reference type, then the symbolic reference to the class or interface representing the element type is resolved by invoking the algorithm in [§5.4.3.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.1) recursively.
2. Finally, access permissions to C are checked:
   * **If C is not accessible (**[**§5.4.4**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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 fails, C is still valid and usable. Nevertheless, resolution fails, and D is prohibited from accessing C.

**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**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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 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**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.4)**) to D, field resolution throws an IllegalAccessError.**
* Otherwise, let <E, L1> be the class or interface in which the referenced field is actually declared and let L2 be the defining loader of D.

Given that the type of the referenced field is Tf, let T be Tf if Tf is not an array type, and let T be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tf otherwise.

**The Java Virtual Machine must impose the loading constraint that TL1 = TL2 (**[**§5.3.4**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.4)**).**

**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**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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 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. Method resolution checks whether C is a class or an interface.
   * **If C is an interface, method resolution throws an IncompatibleClassChangeError.**
2. Method resolution attempts to look up 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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.9)), then method lookup succeeds. All the class names mentioned in the descriptor are resolved ([§5.4.3.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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 lookup is recursively invoked on the direct superclass of C.

1. Otherwise, method lookup attempts to locate the referenced method in any of the superinterfaces of the specified class C.
   * If any superinterface of C declares a method with the name and descriptor specified by the method reference, method lookup succeeds.
   * Otherwise, method lookup fails.

Then:

* **If method lookup fails, method resolution throws a NoSuchMethodError.**
* **Otherwise, if method lookup succeeds and the method is abstract, but C is not abstract, method resolution throws an AbstractMethodError.**
* **Otherwise, if method lookup succeeds but the referenced method is not accessible (**[**§5.4.4**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.4)**) to D, method resolution throws an IllegalAccessError.**
* Otherwise, let <E, L1> be the class or interface in which the referenced method m is actually declared, and let L2 be the defining loader of D.

Given that the return type of m is Tr, and that the formal parameter types of m are Tf1, ..., Tfn, then:

If Tr is not an array type, let T0 be Tr; otherwise, let T0 be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tr.

For *i* = 1 to *n*: If Tfi is not an array type, let Ti be Tfi; otherwise, let Ti be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tfi.

**The Java Virtual Machine must impose the loading constraints TiL1 = TiL2 for *i* = 0 to *n* (**[**§5.3.4**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.4)**).**

**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**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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 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:

* **If C is not an interface, interface method resolution throws an IncompatibleClassChangeError.**
* **Otherwise, if the referenced method does not have the same name and descriptor as a method in C or in one of the superinterfaces of C, or in classObject, interface method resolution throws a NoSuchMethodError.**
* Otherwise, let <E, L1> be the class or interface in which the referenced interface method m is actually declared, and let L2 be the defining loader of D.

Given that the return type of m is Tr, and that the formal parameter types of m are Tf1, ..., Tfn, then:

If Tr is not an array type, let T0 be Tr; otherwise, let T0 be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tr.

For *i* = 1 to *n*: If Tfi is not an array type, let Ti be Tfi; otherwise, let Ti be the element type ([§2.4](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.4)) of Tfi.

**The Java Virtual Machine must impose the loading constraints TiL1 = TiL2 for *i* = 0 to *n* (**[**§5.3.4**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3.4)**).**

**5.4.3.5. Method Type and Method Handle Resolution**

**To resolve an unresolved symbolic reference to a method type, all symbolic references to classes mentioned in the method descriptor encapsulated by the method type are resolved (**[**§5.4.3.1**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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 method type resolution.**

The result of method type resolution is a reference to an instance of java.lang.invoke.MethodType which represents the method descriptor.

Resolution of an unresolved symbolic reference to a method handle is more complicated. Each method handle resolved by the Java Virtual Machine has an equivalent instruction sequence called its *bytecode behavior*, indicated by the method handle's *kind*. The integer values and descriptions of the nine kinds of method handle are given in [Table 5.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.5-220).

Symbolic references by an instruction sequence to fields or methods are indicated by C.x:T, where x and T are the name and descriptor ([§4.3.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.2), [§4.3.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.3)) of the field or method, and C is the class or interface in which the field or method is to be found.

**Table 5.1. Bytecode Behaviors for Method Handles**

| **Kind** | **Description** | **Interpretation** |
| --- | --- | --- |
| 1 | REF\_getField | getfield C.f:T |
| 2 | REF\_getStatic | getstatic C.f:T |
| 3 | REF\_putField | putfield C.f:T |
| 4 | REF\_putStatic | putstatic C.f:T |
| 5 | REF\_invokeVirtual | invokevirtual C.m:(A\*)T |
| 6 | REF\_invokeStatic | invokestatic C.m:(A\*)T |
| 7 | REF\_invokeSpecial | invokespecial C.m:(A\*)T |
| 8 | REF\_newInvokeSpecial | new C; dup; invokespecial C.<init>:(A\*)void |
| 9 | REF\_invokeInterface | invokeinterface C.m:(A\*)T |

Let MH be the symbolic reference to a method handle ([§5.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.1)) being resolved. Then:

* Let R be the symbolic reference to the field or method contained within MH.

(R is derived from the CONSTANT\_Fieldref, CONSTANT\_Methodref, or CONSTANT\_InterfaceMethodref structure referred to by the reference\_index item of theCONSTANT\_MethodHandle from which MH is derived.)

* Let C be a symbolic reference to the type referenced by R.

(C is derived from the CONSTANT\_Class structure referred to by the class\_index item in the CONSTANT\_Fieldref, CONSTANT\_Methodref, orCONSTANT\_InterfaceMethodref represented by R.)

* Let f or m be the name of the field or method referenced by R.

(f or m is derived from the CONSTANT\_NameAndType structure referred to by the name\_and\_type\_index item in the CONSTANT\_Fieldref, CONSTANT\_Methodref, orCONSTANT\_InterfaceMethodref structure from which R is derived.)

* Let T and (in the case of a method) A\* be the return type and argument type sequence of the field or method referenced by R.

(T and A\* are derived from the CONSTANT\_NameAndType structure referred to by the name\_and\_type\_index item in the CONSTANT\_Fieldref,CONSTANT\_Methodref, or CONSTANT\_InterfaceMethodref structure from which R is derived.)

**To resolve MH, all symbolic references to classes, fields, and methods in MH's bytecode behavior are resolved (**[**§5.4.3.1**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.1)**,**[**§5.4.3.2**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.2)**,**[**§5.4.3.3**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.3)**,**[**§5.4.3.4**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.4)**). That is, C,f, m, T, and A\* are resolved. Therefore, any exception that can be thrown as a result of failure of resolution of a symbolic reference to a class, field, method, or interface method can be thrown as a result of method handle resolution.**

(In general, resolving a method handle can be done in exactly the same circumstances that the Java Virtual Machine would successfully resolve the symbolic references in the bytecode behavior. In particular, method handles to private and protected members can be created in exactly those classes for which the corresponding normal accesses are legal.)

If all such symbolic references can be resolved, then a reference to an instance of java.lang.invoke.MethodType is obtained as if by resolution of a symbolic reference to the method descriptor ([§4.3.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.3)) given for the kind of MH in [Table 5.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.5-270).

**Table 5.2. Method Descriptors for Method Handles**

| **Kind** | **Description** | **Method descriptor** |
| --- | --- | --- |
| 1 | REF\_getField | (C)T |
| 2 | REF\_getStatic | ()T |
| 3 | REF\_putField | (C,T)V |
| 4 | REF\_putStatic | (T)V |
| 5 | REF\_invokeVirtual | (C,A\*)T |
| 6 | REF\_invokeStatic | (A\*)T |
| 7 | REF\_invokeSpecial | (C,A\*)T |
| 8 | REF\_newInvokeSpecial | (A\*)C |
| 9 | REF\_invokeInterface | (C,A\*)T |

The result of method handle resolution is a reference o to an instance of java.lang.invoke.MethodHandle which represents the method handle MH. If the methodm has the ACC\_VARARGS flag set ([§4.6](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.6)), then o is a variable arity method handle; otherwise, o is a fixed arity method handle.

(A variable arity method handle performs argument list boxing (JLS §15.12.4.2) when invoked via invoke, while its behavior with respect to invokeExact is as if theACC\_VARARGS flag were not set.)

**Method handle resolution throws an IncompatibleClassChangeError if m has the ACC\_VARARGS flag set and either m's argument type sequence is empty or the last parameter in m's argument type sequence is not an array type. (That is, creation of a variable arity method handle fails.)**

The type descriptor of the java.lang.invoke.MethodHandle instance referenced by o is the java.lang.invoke.MethodType instance produced by method type resolution mentioned earlier.

(The type descriptor of a method handle is such that a valid call to invokeExact in java.lang.invoke.MethodHandle on the method handle has exactly the same stack effects as the bytecode behavior. Calling this method handle on a valid set of arguments has exactly the same effect and returns the same result (if any) as the corresponding bytecode behavior.)

An implementation of the Java Virtual Machine is not required to intern method types or method handles. That is, two distinct symbolic references to method types or method handles which are structurally identical might not resolve to the same instance of java.lang.invoke.MethodType or java.lang.invoke.MethodHandlerespectively.

*The java.lang.invoke.MethodHandles class in the Java SE platform API allows creation of method handles with no bytecode behavior. Their behavior is defined by the method ofjava.lang.invoke.MethodHandles that creates them. For example, a method handle may, when invoked, first apply transformations to its argument values, then supply the transformed values to the invocation of another method handle, then apply a transformation to the value returned from that invocation, then return the transformed value as its own result.*

**5.4.3.6. Call Site Specifier Resolution**

To resolve an unresolved symbolic reference to a call site specifier involves three steps:

* A call site specifier gives a symbolic reference to a method handle which is to serve as the *bootstrap method* for a dynamic call site. The method handle is resolved ([§5.4.3.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.5)) to obtain a reference to an instance of java.lang.invoke.MethodHandle.
* A call site specifier gives a method descriptor, *TD*. A reference to an instance of java.lang.invoke.MethodType is obtained as if by resolution of a symbolic reference to a method type ([§5.4.3.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.5)) with the same parameter and return types as *TD*.
* A call site specifier gives zero or more *static arguments*, which communicate application-specific metadata to the bootstrap method. Any static arguments which are symbolic references to classes, method handles, or method types are resolved, as if by invocation of the *ldc* instruction ([§*ldc*](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.ldc)), to obtain references to Classobjects, java.lang.invoke.MethodHandle objects, and java.lang.invoke.MethodType objects respectively. Any static arguments that are string literals are used to obtain references to String objects.

The result of call site specifier resolution is a tuple consisting of:

* the reference to an instance of java.lang.invoke.MethodHandle,
* the reference to an instance of java.lang.invoke.MethodType,
* the references to instances of Class, java.lang.invoke.MethodHandle, java.lang.invoke.MethodType, and String.

**During resolution of the symbolic reference to the method handle in the call site specifier, or resolution of the symbolic reference to the method type for the method descriptor in the call site specifier, or resolution of a symbolic reference to any static argument, any of the exceptions pertaining to method type or method handle resolution (**[**§5.4.3.5**](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.5)**) may be thrown.**

**5.4.4. Access Control**

A class or interface C is *accessible* to a class or interface D if and only if either of the following conditions is true:

* C is public.
* C and D are members of the same run-time package ([§5.3](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.3)).

A field or method R is accessible to a class or interface D if and only if any of the following conditions are true:

* R is public.
* R is protected and is declared in a class C, and D is either a subclass of C or C itself. Furthermore, if R is not static, then the symbolic reference to R must contain a symbolic reference to a class T, such that T is either a subclass of D, a superclass of D, or D itself.
* R is either protected or has default access (that is, neither public nor protected nor private), and is declared by a class in the same run-time package as D.
* R is private and is declared in D.

This discussion of access control omits a related restriction on the target of a protected field access or method invocation (the target must be of class D or a subtype of D). That requirement is checked as part of the verification process ([§5.4.1](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.1)); it is not part of link-time access control.

**5.4.5. Method overriding**

An instance method m1 declared in class C overrides another instance method m2 declared in class A iff all of the following are true:

* C is a subclass of A.
* m2 has the same name and descriptor as m1.
* Either:
  + m2 is marked ACC\_PUBLIC; or is marked ACC\_PROTECTED; or is marked neither ACC\_PUBLIC nor ACC\_PROTECTED nor ACC\_PRIVATE and belongs to the same run-time package as C, or
  + m1 overrides a method m3, m3 distinct from m1, m3 distinct from m2, such that m3 overrides m2.

**5.5. Initialization**

*Initialization* of a class or interface consists of executing its class or interface initialization method ([§2.9](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.9)).

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

* The execution of any one of the Java Virtual Machine instructions *new*, *getstatic*, *putstatic*, or *invokestatic* that references the class or interface ([§*new*](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.new), [§*getstatic*](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.getstatic),[§*putstatic*](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.putstatic), [§*invokestatic*](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.invokestatic)). All of these instructions reference a class directly or indirectly through either a field reference or a method reference.

Upon execution of a *new* instruction, the referenced class or interface 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.

* The first invocation of a java.lang.invoke.MethodHandle instance which was the result of resolution of a method handle by the Java Virtual Machine ([§5.4.3.5](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.4.3.5)) and which has a kind of 2 (REF\_getStatic), 4 (REF\_putStatic), or 6 (REF\_invokeStatic).
* Invocation of certain reflective methods in the class library ([§2.12](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.12)), for example, in class Class or in package java.lang.reflect.
* The initialization of one of its subclasses.
* Its designation as the initial class at Java Virtual Machine start-up ([§5.2](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-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.
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 LCand 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](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-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 SC has not yet been initialized, then recursively perform this entire procedure for SC. If necessary, verify and prepare SC first.

**If the initialization of SC 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.**

1. Next, determine whether assertions are enabled for C by querying its defining class loader.
2. Next, execute the class or interface initialization method of C.
3. 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.
4. **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.**

1. 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.

**5.6. Binding Native Method Implementations**

*Binding* is the process by which a function written in a language other than the Java programming language and implementing a native method is integrated into the Java Virtual Machine so that it can be executed. Although this process is traditionally referred to as linking, the term binding is used in the specification to avoid confusion with linking of classes or interfaces by the Java Virtual Machine.

**5.7. Java Virtual Machine Exit**

The Java Virtual Machine exits when some thread invokes the exit method of class Runtime or class System, or the halt method of class Runtime, and the exit orhalt operation is permitted by the security manager.

In addition, the JNI (Java Native Interface) Specification describes termination of the Java Virtual Machine when the JNI Invocation API is used to load and unload the Java Virtual Machine.