CHAPTER 5

# Loading, Linking, and Initializing

THE Java virtual machine dynamically loads (§2.17.2), links (§2.17.3), and initializes (§2.17.4) 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 runtime 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>` (§3.9).

In this chapter, Section 5.1 describes how the Java virtual machine derives symbolic references from the binary representation of a class or interface. Section 5.2 explains how the processes of loading, linking, and initialization are first initiated by the Java virtual machine. Section 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 Section 5.4. Section 5.5 details how classes and interfaces are initialized. Finally, Section 5.6 introduces the notion of binding native methods.

## 5.1 The Runtime Constant Pool

The Java virtual machine maintains a per-type constant pool (§3.5.5), a runtime data structure that serves many of the purposes of the symbol table of a conventional programming language implementation.

The `constant_pool` table (§4.4) in the binary representation of a class or interface is used to construct the runtime constant pool upon class or interface cre-

ation (§5.3). All references in the runtime constant pool are initially symbolic. The symbolic references in the runtime 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) 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 fully qualified name of the class or interface.
  - For an array class of *M* dimensions, the name begins with *M* 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).
    - Otherwise, if the element type is a reference type, it is represented by the ASCII "L" character followed by the fully qualified name 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 (§2.9) or an interface (§2.13.3.1) is derived from a `CONSTANT_Fieldref_info` structure (§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 (§2.10) is derived from a `CONSTANT_Methodref_info` structure (§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 (§2.13) is derived from a `CONSTANT_InterfaceMethodref_info` structure (§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.

In addition, certain nonreference runtime values are derived from items found in the `constant_pool` table:

- A string literal (§2.3) is derived from a `CONSTANT_String_info` structure (§4.4.3) in the binary representation of a class or interface. The `CONSTANT_String_info` structure gives the sequence of Unicode characters constituting the string literal.
- The Java programming language requires that identical string literals (that is, literals that contain the same sequence of characters) must refer to the same instance of class `String`. 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,

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

  must have the value `true`.
- To derive a string literal, the Java virtual machine examines the sequence of characters 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 characters 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 characters given by the `CONSTANT_String_info` structure; that class instance is the result of string literal derivation. Finally, the `intern` method of the new `String` instance is invoked.
- Runtime constant values are derived from `CONSTANT_Integer_info`, `CONSTANT_Float_info`, `CONSTANT_Long_info`, or `CONSTANT_Double_info` structures (§4.4.4, §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, §4.4.5). The runtime 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` (§4.4.6) and

`CONSTANT_Utf8_info` (§4.4.7) structures are only used indirectly when deriving symbolic references to classes, interfaces, methods, and fields, and when deriving string literals.

## 5.2 Virtual Machine Start-up

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). The Java virtual machine then links the initial class, initializes it, and invokes its `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 some implementations of the Java virtual machine the initial class could be provided as a command line argument, as in JDK releases 1.0 and 1.1. Alternatively, the initial class could be provided by the implementation. In this case the initial class might set up a class loader that would in turn load an application, as in the Java 2 SDK, Standard Edition, v1.2. 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 (§3.5.4) 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 runtime constant pool. Class or interface creation may also be triggered by *D* invoking methods in certain Java class libraries (§3.12) such as reflection.

If *C* is not an array class, it is created by loading a binary representation of *C* (see Chapter 4, "The `class` File Format") using a class loader (§2.17.2). 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 types of class loaders: user-defined class loaders and the bootstrap class loader supplied by the Java virtual machine. Every user-defined class

loader is an instance of a subclass of the abstract class `ClassLoader`. Applications employ 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 fully qualified name and its defining class loader. Each such class or interface belongs to a single *runtime package*. The runtime 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).
  - If *D* was defined by a user-defined class loader, then that same user-defined class loader initiates loading of *C* (§5.3.2).
- Otherwise *N* denotes an array class. An array class is created directly by the Java virtual machine (§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*.

We will sometimes represent a class or interface using the notation $<N, L_d>$, where *N* denotes the name of the class or interface and $L_d$ denotes the defining loader of the class or interface. We will also represent a class or interface using the notation $N^{L_i}$, where *N* denotes the name of the class or interface and $L_i$ 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 performs one of the following two operations in order to load *C*:

1. The Java virtual machine searches for a purported representation of *C* in a platform-dependent manner. Note that there is no guarantee that a purported representation found is valid or is a representation of *C*.

   Typically, a class or interface will be represented using a file in a hierarchical file system. The name of the class or interface will usually be encoded in the pathname of the file.

   This phase of loading must detect the following error:

   - If no purported representation of *C* is found, loading throws an instance of `NoClassDefFoundError` or an instance of one of its subclasses.

   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 Section 5.3.5. That class is *C*.

2. The bootstrap class loader can delegate the loading of *C* to some user-defined class loader *L* by passing *N* to an invocation of a `loadClass` method on *L*. The result of the invocation is *C*. The Java virtual machine then records that the bootstrap loader is an initiating loader of *C* (§5.3.4).

### 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*.[1] 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). 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); it then must invoke the method `defineClass` of class `ClassLoader`. 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 Section 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*.

### 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) 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). If the component type is a reference type, the accessibil-

[1] Since JDK release 1.1 the Java virtual machine invokes the `loadClass` method of a class loader in order to cause it to load a class or interface. The argument to `loadClass` is the name of the class or interface to be loaded. There is also a two-argument version of the `loadClass` method. 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 the Java virtual machine relied on it to link the loaded class or interface. From JDK release 1.1 onward, the Java virtual machine links the class or interface directly, without relying on the class loader.

ity 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 $N^{L1} = N^{L2}$ during preparation (§5.4.2) and resolution (§5.4.3). To enforce these constraints, the Java virtual machine will, at certain prescribed times (see §5.3.1, §5.3.2, §5.3.3, and §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, §5.4.3.2, §5.4.3.3, and §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 $N^L = N^{L'}$.
- $C \neq C'$.

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 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 in `class` file format (§4.1, pass 1 of §4.9.1), 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`.[2]
   - 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 Section 5.4.3.1. Note that if $C$ is an

[2] `UnsupportedClassVersionError` was introduced in the Java 2 platform, Standard Edition, v1.2. In earlier versions of the platform 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.

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`.

4. If *C* has any direct superinterfaces, the symbolic references from *C* to its direct superinterfaces are resolved using the algorithm of Section 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. 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 Linking

Linking a class or interface (§2.17.3) 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.

### 5.4.1 Verification

The representation of a class or interface is *verified* (§4.9) to ensure that its binary representation is structurally valid (passes 2 and 3 of §4.9.1). Verification may cause additional classes and interfaces to be loaded (§5.3) but need not cause them to be prepared or verified.

Verification must detect the following error:

- If the representation of the class or interface does not satisfy the static or structural constraints listed in Section 4.8, "Constraints on Java Virtual Machine Code," verification throws a `VerifyError`.

A class or interface must be successfully verified before it is initialized. Any attempt to initialize a class or interface that has not been successfully verified must be preceded by verification. Repeated verification of a class or interface that the Java virtual machine has previously unsuccessfully attempted to verify always fails 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 the class or interface and initializing those fields to their standard default values (§2.5.1). Preparation should not be confused with the execution of static initializers (§2.11); unlike execution of static initializers, preparation does not require the execution of any Java virtual machine code.

During preparation of a class or interface *C*, the Java virtual machine also imposes loading constraints (§5.3.4). Let *L1* be the defining loader of *C*. For each method *m* declared in *C* that overrides a method declared in a superclass or superinterface <*D*, *L2*>, the Java virtual machine imposes the following loading constraints: Let *T0* be the name of the type returned by *m*, and let *T1*, ..., *Tn* be the names of the argument types of *m*. Then $Ti^{L1} = Ti^{L2}$ for $i = 0$ to $n$ (§5.3.4).

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 *C* that declares the implementation of method *m* inherited by *C*. The Java virtual machine imposes the following constraints:

Let *T0* be the name of the type returned by *m*, and let *T1*, ..., *Tn* be the names of the argument types of *m*. Then $Ti^{L2} = Ti^{L3}$ for $i = 0$ to $n$ (§5.3.4).

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

### 5.4.3 Resolution

The process of dynamically determining concrete values from symbolic references in the runtime constant pool is known as *resolution*.

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.

Subsequent attempts to resolve a symbolic reference that the Java virtual machine has previously unsuccessfully attempted to resolve always fails with the same error that was thrown as a result of the initial resolution attempt.

Certain Java virtual machine instructions require specific 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 complete the field resolution steps given in Section 5.4.3.2. In addition, it must also check that the field is not `static`. If it is a `static` field, 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 failure of resolution.

The Java virtual machine instructions *anewarray*, *checkcast*, *getfield*, *getstatic*, *instanceof*, *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*, *multianewarray*, *new*, *putfield*, and *putstatic* make symbolic references to the runtime constant pool. Execution of any of these instructions requires resolution of its symbolic reference.

The following sections describe the process of resolving a symbolic reference in the runtime 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*. 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. The details of the process are given in Section 5.3.

2. 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 Section 5.4.3.1 recursively.

3. 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 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). 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.

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*, *L1*> be the class or interface in which the referenced field is actually declared and let *L2* be the defining loader of *D*. Let *T* be the name of the type of the referenced field. The Java virtual machine must impose the loading constraint that $T^{L1} = T^{L2}$ (§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). Therefore, any exceptions that can be thrown due to 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 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*.
3. 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.

If method lookup fails, method resolution throws a `NoSuchMethodError`. If method lookup succeeds and the method is `abstract`, but *C* is not `abstract`, method resolution throws an `AbstractMethodError`. Otherwise, if the referenced method is not accessible (§5.4.4) to *D*, method resolution throws an `IllegalAccessError`.

Otherwise, let <*E*, *L1*> be the class or interface in which the referenced method is actually declared and let *L2* be the defining loader of *D*. Let *T0* be the name of the type returned by the referenced method, and let *T1*, ..., *Tn* be the names of the argument types of the referenced method. The Java virtual machine must impose the loading constraints $Ti^{L1} = Ti^{L2}$ for $i = 0$ to $n$ (§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). Therefore, any exceptions 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:

- 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 class `Object`, interface method resolution throws a `NoSuchMethodError`.

Otherwise, let <*E*, *L1*> be the class or interface in which the referenced interface method is actually declared and let *L2* be the defining loader of *D*. Let *T0* be the name of the type returned by the referenced method, and let *T1*, ..., *Tn* be the names of the argument types of the referenced method. The Java virtual machine must impose the loading constraints $Ti^{L1} = Ti^{L2}$ for $i = 0$ to $n$ (§5.3.4).

### 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 are true:

- *C* is `public`.
- *C* and *D* are members of the same runtime package (§5.3).

A field or method *R* is *accessible* to a class or interface *D* if and only if any of the following conditions is 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 package private (that is, neither `public` nor `protected` nor `private`), and is declared by a class in the same runtime 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); it is not part of link-time access control.

## 5.5 Initialization

*Initialization* of a class or interface consists of invoking its static initializers (§2.11) and the initializers for static fields (§2.9.2) declared in the class. This process is described in more detail in §2.17.4 and §2.17.5.

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. Each of these instructions corresponds to one of the conditions in §2.17.4. All of the previously listed 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.
- Invocation of certain reflective methods in the class library (§3.12), 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).

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

## 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.