Hidden Classes

JVMS 5.4.4 Access Control

...

- R is either `protected` or has default access ...
- R is `private` and is declared by a class or interface C that belongs to the same nest as D, according to the nestmate test below.

If R is not accessible to D, then **access control throws an `IllegalAccessError`.**

- ~~If R is `public`, `protected`, or has default access, then access control throws an `IllegalAccessError`.~~
- ~~If R is `private`, then the nestmate test failed, and access control fails for the same reason.~~

Otherwise, access control succeeds.

A *nest* is a set of classes and interfaces that allow mutual access to their `private` members. One of the classes or interfaces is the *nest host*. It enumerates the classes and interfaces which belong to the nest, using the `NestMembers` attribute (§4.7.29). Each of them in turn designates it as the nest host, using the `NestHost` attribute (§4.7.28). A class or interface which lacks a `NestHost` attribute belongs to the nest hosted by itself; if it also lacks a `NestMembers` attribute, this nest is a singleton consisting only of the class or interface itself. **The nest host for a given class or interface (that is, the nest to which the class or interface belongs) is *determined* by the Java Virtual Machine as part of access control, rather than when the class or interface is loaded. Certain methods of the Java SE Platform API may determine the nest host for a given class or interface prior to access control, in which case access control respects the prior determination.**

To determine whether a class or interface C belongs to the same nest as a class or interface D, the *nestmate test* is applied. C and D belong to the same nest if and only if the nestmate test succeeds. The nestmate test is as follows:

- If C and D are the same class or interface, then the nestmate test succeeds.
- Otherwise, the following steps are performed, in order:

    1. **Let H be the nest host of D, if the nest host of D has previously been determined.**
       If the nest host of D **has not previously been determined, then it is determined using the algorithm below, yielding H.** ~~The nest host of D, H, is determined (below). If an exception is thrown, then the nestmate test fails for the same reason.~~
    2. **Let H' be the nest host of C, if the nest host of C has previously been determined.**
       If the nest host of C **has not previously been determined, then it is determined using the algorithm below, yielding H'.** ~~The nest host of C, H', is determined (below). If an exception is thrown, then the nestmate test fails for the same reason.~~
    3. H and H' are compared. If H and H' are the same class or interface, then the nestmate test succeeds. Otherwise, the nestmate test fails ~~by throwing an `IllegalAccessError`~~.

The nest host of a class or interface `M` is determined as follows:

- If `M` lacks a `NestHost` attribute, then `M` is its own nest host.
- Otherwise, `M` has a `NestHost` attribute, and its `host_class_index` item is used as an index into the run-time constant pool of `M`. The symbolic reference at that index is resolved to a class or interface H (§5.4.3.1). **Then:**

  ~~During resolution of this symbolic reference, any of the exceptions pertaining to class or interface resolution can be thrown. Otherwise, resolution of H succeeds.~~

  ~~If any of the following is true, an `IncompatibleClassChangeError` is thrown:~~

  If resolution of this symbolic reference fails, then M is its own nest host. Any exception thrown as a result of failure of class or interface resolution is not re-thrown by access control.

  Otherwise, if resolution succeeds but any of the following is true, then M is its own nest host:

  - H is not in the same run-time package as `M`.
  - H lacks a `NestMembers` attribute.
  - H has a `NestMembers` attribute, but there is no entry in its `classes` array that refers to a class or interface with the name `N`, where `N` is the name of `M`.

  Otherwise, H is the nest host of `M`.

API specification

```
/**
 * Creates a <em>hidden</em> class or interface from {@code bytes},
```

```
* returning a {@code Lookup} on the newly created class or interface.
*
* <p> Ordinarily, a class or interface {@code C} is created by a class loader, which either
* defines {@code C} directly or delegates to another class loader.
* A class loader defines {@code C} directly by invoking
* {@link ClassLoader#defineClass(String, byte[], int, int, ProtectionDomain)
* ClassLoader::defineClass}, which causes the Java Virtual Machine
* to derive {@code C} from a purported representation in {@code class} file format.
* In situations where use of a class loader is undesirable, a class or interface {@code C} can be
* created by this method instead. This method is capable of defining {@code C},
* and thereby creating it, without invoking {@code ClassLoader::defineClass}.
* Instead, this method defines {@code C} as if by arranging for
* the Java Virtual Machine to derive a nonarray class or interface {@code C}
* from a purported representation in {@code class} file format
* using the following rules:
*
* <ol>
* <li> The {@linkplain #lookupModes() lookup modes} for this {@code Lookup}
* must include {@linkplain #hasFullPrivilegeAccess() full privilege} access.
* This level of access is needed to create {@code C} in the module
* of the lookup class of this {@code Lookup}.</li>
*
* <li> The purported representation in {@code bytes} must be a {@code ClassFile}
* structure of a supported major and minor version. The major and minor version
* may differ from the {@code class} file version of the lookup class of this
* {@code Lookup}.</li>
*
* <li> The name given by {@code this_class} in the {@code ClassFile} structure
* must indicate that {@code C} is in the same package as the lookup class.</li>
*
* <li> The name of {@code C} is derived from the name given by {@code this_class}
* as follows. Let {@code N} be the binary name indicated by {@code this_class}
* (that is, take the internal form given by {@code this_class} and decode it
* to a binary name by replacing ASCII forward slashes ({@code /}) with ASCII periods ({@code .}).
* The name of {@code C} is {@code N + '/' + <suffix>},
* where {@code <suffix>} is an unqualified name that is guaranteed to be unique
* during this execution of the JVM. The name of {@code C} is not a binary name
* because it contains an ASCII forward slash.</li>
*
* <li> If {@code C} has a direct superclass, the symbolic reference from {@code C}
* to its direct superclass is resolved using the algorithm of JVMS 5.4.3.1.
* Any exceptions that can be thrown due to class or interface resolution
* can be thrown by this method. In addition:
* <ul>
* <li> The class or interface named as the direct superclass of {@code C}
*  must not in fact be an interface.</li>
* <li> None of the superclasses of {@code C} may be {@code C} itself.</li>
* </ul>
* </li>
*
* <li> If {@code C} has any direct superinterfaces, the symbolic references
* from {@code C} to its direct superinterfaces are resolved using the algorithm
* of JVMS 5.4.3.1.
* Any exceptions that can be thrown due to class or interface resolution
* can be thrown by this method. In addition:
* <ul>
* <li> All of the classes and interfaces named as direct superinterfaces of {@code C}
* must in fact be interfaces.</li>
* <li> None of the superinterfaces of {@code C} may be {@code C} itself.</li>
* </ul>
* </li>
*
* <li> The Java Virtual Machine marks {@code C} as having the same defining class loader,
* runtime package, and {@linkplain java.security.ProtectionDomain protection domain}
* as the lookup class of this {@code Lookup}.
* No class loader is recorded as the initiating class loader for {@code C}.</li>
* </ol>
*
* <p>After {@code C} has been created, it is linked by the Java Virtual Machine.
* The constant pool entry indicated by its {@code this_class} item is resolved.
* If the {@code initialize} parameter is {@code true}, then
* {@code C} is initialized by the Java Virtual Machine.
*
* <p>The newly created class or interface {@code C} is <em>hidden</em>, in the sense that
* no other class or interface can refer to {@code C} via a constant pool entry.
* That is, a hidden class or interface cannot be named as a supertype, a field type,
* a method parameter type, or a method return type by any other class.
* This is because a hidden class or interface does not have a binary name, so
* there is no internal form available to record in any class's constant pool.
```

```
 * (Given the {@code Lookup} object returned this method, its lookup class
 * is a {@code Class} object for which {@link Class#getName()} returns a string
 * that is not a binary name.)
 * A hidden class or interface is not discoverable by {@link Class#forName(String, boolean, ClassLoader)},
 * {@link ClassLoader#loadClass(String, boolean)}, or {@link #findClass(String)}, and
 * is not {@linkplain java.lang.instrument.Instrumentation#isModifiableClass(Class)
 * modifiable} by Java agents or tool agents using the <a href="{@docRoot}/../specs/jvmti.html">
 * JVM Tool Interface</a>.
 *
 * <p> If {@code options} has the {@link ClassOption#NESTMATE NESTMATE} option, then
 * the newly created class or interface {@code C} is a member of a nest. The nest to which
 * {@code C} belongs is not based on any {@code NestHost} attribute in
 * the {@code ClassFile} structure from which {@code C} was derived. Instead, the following rules
 * determine the nest host of {@code C}:
 * <ul>
 * <li>If the nest host of the lookup class of this {@code Lookup} has previously been determined,
 * then {@code H} be the nest host of the lookup class.</li>
 * <li>Otherwise, it is determined using the algorithm in JVMS 5.4.4, yielding {@code H}.</li>
 * <li>The nest host of {@code C} is determined to be {@code H}, the nest host of the lookup class.</li>
 * </ul>
 *
 * <p> If {@code options} has {@link ClassOption#WEAK WEAK} option, then
 * the newly created class or interface is <em>not strongly referenced</em> from
 * its defining class loader. Therefore, it may be unloaded while
 * its defining class loader is strongly reachable.
 *
 * <p> A hidden class or interface may be serializable, but this requires a custom serialization
 * mechanism in order to ensure that instances are properly serialized
 * and deserialized. The default serialization mechanism supports only
 * classes and interfaces that are discoverable by their class name.
 *
 * @param bytes the bytes that make up the class data,
 * in the format of a valid {@code class} file as defined by The JavaTM Virtual Machine Specification.
 * @param initialize if {@code true} the class will be initialized.
 * @param options {@linkplain ClassOption class options}
 * @return the {@code Lookup} object on the hidden class

 *

 * @throws IllegalAccessException if this {@code Lookup} does not have
 * {@linkplain #hasFullPrivilegeAccess() full privilege} access
 * @throws SecurityException if a security manager is present and it
 *  <a href="MethodHandles.Lookup.html#secmgr">refuses access</a>
 * @throws ClassFormatError if {@code bytes} is not a {@code ClassFile} structure
 * @throws UnsupportedClassVersionError if {@code bytes} is not of a supported major or minor version
 * @throws IllegalArgumentException if {@code bytes} denotes a class in a different package than the
lookup class
 * @throws IncompatibleClassChangeError if the class or interface named as the direct superclass of {@code
C}
 * is in fact an interface, or if any of the classes or interfaces named as direct superinterfaces of
{@code C}
 * are not in fact interfaces
 * @throws ClassCircularityError if any of the superclasses or superinterfaces of {@code C} is {@code C}
itself
 * @throws VerifyError if the newly created class cannot be verified
 * @throws LinkageError if the newly created class cannot be linked for any other reason
 * @throws NullPointerException if any parameter is {@code null}
 *
 * @since 15
 * @see Class#isHiddenClass()
 * @jvms 4.2.1 Binary Class and Interface Names
 * @jvms 4.2.2 Unqualified Names
 * @jvms 5.4.3.1 Class and Interface Resolution
 * @jvms 5.4.4 Access Control
 * @jvms 5.3.5 Deriving a {@code Class} from a {@code class} File Representation
 * @jvms 5.4 Linking
 * @jvms 5.5 Initialization
 */
public Lookup defineHiddenClass(byte[] bytes, boolean initialize, ClassOption... options)
        throws IllegalAccessException
```

```
/**
 * Returns the nest host of the <a href=#nest>nest</a> to which the class
 * or interface represented by this {@code Class} object belongs.
 * Every class and interface is a member of exactly one nest.
 * Often, a class or interface belongs to a nest consisting only of itself,
 * in which case this method returns {@code this} to indicate that the class
 * or interface is the nest host.
 *
 * <p>If this {@code Class} object is an array type or a primitive type or ,
 * {@code void}, then this method returns {@code this} to indicate
 * that the represented entity belongs to the nest consisting only of
 * itself, and is the nest host.
 *
 * <p>Otherwise, if the nest host of this class or interface has previously
 * been determined, then this method returns the nest host of this class
 * or interface. If the nest host of this class or interface has
 * not previously been determined, then this method returns the nest
 * host determined using the algorithm of JVMS 5.4.4.
 *
 * @return the nest host of this class or interface
 *
 * @throws SecurityException
 * If the returned class is not the current class, and
 * if a security manager, <i>s</i>, is present and the caller's
 * class loader is not the same as or an ancestor of the class
 * loader for the returned class and invocation of {@link
 * SecurityManager#checkPackageAccess s.checkPackageAccess()}
 * denies access to the package of the returned class
 * @since 11
 * @jvms 4.7.28 The {@code NestHost} Attribute
 * @jvms 4.7.29 The {@code NestMembers} Attribute
 * @jvms 5.4.4 Access Control
 */
@CallerSensitive
public Class<?> getNestHost()
```