Module java.base
Package java.lang.invoke

Class MethodHandles.Lookup

java.lang.Object
    java.lang.invoke.MethodHandles.Lookup

Enclosing class:
MethodHandles

```
public static final class MethodHandles.Lookup
extends Object
```

A *lookup object* is a factory for creating method handles, when the creation requires access checking. Method handles do not perform access checks when they are called, but rather when they are created. Therefore, method handle access restrictions must be enforced when a method handle is created. The caller class against which those restrictions are enforced is known as the lookup class.

A lookup class which needs to create method handles will call `MethodHandles.lookup` to create a factory for itself. When the `Lookup` factory object is created, the identity of the lookup class is determined, and securely stored in the `Lookup` object. The lookup class (or its delegates) may then use factory methods on the `Lookup` object to create method handles for access-checked members. This includes all methods, constructors, and fields which are allowed to the lookup class, even private ones.

Lookup Factory Methods

The factory methods on a `Lookup` object correspond to all major use cases for methods, constructors, and fields. Each method handle created by a factory method is the functional equivalent of a particular *bytecode behavior*. (Bytecode behaviors are described in section 5.4.3.5 of the Java Virtual Machine Specification.) Here is a summary of the correspondence between these factory methods and the behavior of the resulting method handles:

lookup expression	member	bytecode behavior
`lookup.findGetter(C.class,"f",FT.class)`	`FT f;`	`(T) this.f;`
`lookup.findStaticGetter(C.class,"f",FT.class)`	`static FT f;`	`(FT) C.f;`
`lookup.findSetter(C.class,"f",FT.class)`	`FT f;`	`this.f = x;`
`lookup.findStaticSetter(C.class,"f",FT.class)`	`static FT f;`	`C.f = arg;`
`lookup.findVirtual(C.class,"m",MT)`	`T m(A*);`	`(T) this.m(arg*);`
`lookup.findStatic(C.class,"m",MT)`	`static T m(A*);`	`(T) C.m(arg*);`
`lookup.findSpecial(C.class,"m",MT,this.class)`	`T m(A*);`	`(T) super.m(arg*);`
`lookup.findConstructor(C.class,MT)`	`C(A*);`	`new C(arg*);`
`lookup.unreflectGetter(aField)`	`(static)? FT f;`	`(FT) aField.get(thisOrNull);`
`lookup.unreflectSetter(aField)`	`(static)? FT f;`	`aField.set(thisOrNull, arg);`
`lookup.unreflect(aMethod)`	`(static)? T m(A*);`	`(T) aMethod.invoke(thisOrNull, arg*);`

lookup expression	member	bytecode behavior
`lookup.unreflectConstructor(aConstructor)`	`C(A*);`	`(C) aConstructor.newInstance(arg*);`
`lookup.unreflectSpecial(aMethod,this.class)`	`T m(A*);`	`(T) super.m(arg*);`
`lookup.findClass("C")`	`class C { ... }`	`C.class;`

Here, the type `C` is the class or interface being searched for a member, documented as a parameter named `refc` in the lookup methods. The method type `MT` is composed from the return type `T` and the sequence of argument types `A*`. The constructor also has a sequence of argument types `A*` and is deemed to return the newly-created object of type `C`. Both `MT` and the field type `FT` are documented as a parameter named `type`. The formal parameter `this` stands for the self-reference of type `C`; if it is present, it is always the leading argument to the method handle invocation. (In the case of some `protected` members, `this` may be restricted in type to the lookup class; see below.) The name `arg` stands for all the other method handle arguments. In the code examples for the Core Reflection API, the name `thisOrNull` stands for a null reference if the accessed method or field is static, and `this` otherwise. The names `aMethod`, `aField`, and `aConstructor` stand for reflective objects corresponding to the given members declared in type `C`.

The bytecode behavior for a `findClass` operation is a load of a constant class, as if by `ldc CONSTANT_Class`. The behavior is represented, not as a method handle, but directly as a `Class` constant.

In cases where the given member is of variable arity (i.e., a method or constructor) the returned method handle will also be of variable arity. In all other cases, the returned method handle will be of fixed arity.

Discussion: The equivalence between looked-up method handles and underlying class members and bytecode behaviors can break down in a few ways:

- If `C` is not symbolically accessible from the lookup class's loader, the lookup can still succeed, even when there is no equivalent Java expression or bytecoded constant.
- Likewise, if `T` or `MT` is not symbolically accessible from the lookup class's loader, the lookup can still succeed. For example, lookups for `MethodHandle.invokeExact` and `MethodHandle.invoke` will always succeed, regardless of requested type.
- If there is a security manager installed, it can forbid the lookup on various grounds (see below). By contrast, the `ldc` instruction on a `CONSTANT_MethodHandle` constant is not subject to security manager checks.
- If the looked-up method has a very large arity, the method handle creation may fail with an `IllegalArgumentException`, due to the method handle type having too many parameters.

Access checking

Access checks are applied in the factory methods of `Lookup`, when a method handle is created. This is a key difference from the Core Reflection API, since `java.lang.reflect.Method.invoke` performs access checking against every caller, on every call.

All access checks start from a `Lookup` object, which compares its recorded lookup class against all requests to create method handles. A single `Lookup` object can be used to create any number of access-checked method handles, all checked against a single lookup class.

A `Lookup` object can be shared with other trusted code, such as a metaobject protocol. A shared `Lookup` object delegates the capability to create method handles on private members of the lookup class. Even if privileged code uses the `Lookup` object, the access checking is confined to the privileges of the original lookup class.

A lookup can fail, because the containing class is not accessible to the lookup class, or because the desired class member is missing, or because the desired class member is not accessible to the lookup class, or because the lookup object is not trusted enough to access the member. In the case of a field setter function on a `final` field, finality enforcement is treated as a kind of access control,

and the lookup will fail, except in special cases of `Lookup.unreflectSetter`. In any of these cases, a `ReflectiveOperationException` will be thrown from the attempted lookup. The exact class will be one of the following:

- NoSuchMethodException — if a method is requested but does not exist
- NoSuchFieldException — if a field is requested but does not exist
- IllegalAccessException — if the member exists but an access check fails

In general, the conditions under which a method handle may be looked up for a method `M` are no more restrictive than the conditions under which the lookup class could have compiled, verified, and resolved a call to `M`. Where the JVM would raise exceptions like `NoSuchMethodError`, a method handle lookup will generally raise a corresponding checked exception, such as `NoSuchMethodException`. And the effect of invoking the method handle resulting from the lookup is exactly equivalent to executing the compiled, verified, and resolved call to `M`. The same point is true of fields and constructors.

Discussion: Access checks only apply to named and reflected methods, constructors, and fields. Other method handle creation methods, such as `MethodHandle.asType`, do not require any access checks, and are used independently of any `Lookup` object.

If the desired member is `protected`, the usual JVM rules apply, including the requirement that the lookup class must either be in the same package as the desired member, or must inherit that member. (See the Java Virtual Machine Specification, sections 4.9.2, 5.4.3.5, and 6.4.) In addition, if the desired member is a non-static field or method in a different package, the resulting method handle may only be applied to objects of the lookup class or one of its subclasses. This requirement is enforced by narrowing the type of the leading `this` parameter from `C` (which will necessarily be a superclass of the lookup class) to the lookup class itself.

The JVM imposes a similar requirement on `invokespecial` instruction, that the receiver argument must match both the resolved method *and* the current class. Again, this requirement is enforced by narrowing the type of the leading parameter to the resulting method handle. (See the Java Virtual Machine Specification, section 4.10.1.9.)

The JVM represents constructors and static initializer blocks as internal methods with special names (`"<init>"` and `"<clinit>"`). The internal syntax of invocation instructions allows them to refer to such internal methods as if they were normal methods, but the JVM bytecode verifier rejects them. A lookup of such an internal method will produce a `NoSuchMethodException`.

If the relationship between nested types is expressed directly through the `NestHost` and `NestMembers` attributes (see the Java Virtual Machine Specification, sections 4.7.28 and 4.7.29), then the associated `Lookup` object provides direct access to the lookup class and all of its nestmates (see `Class.getNestHost`). Otherwise, access between nested classes is obtained by the Java compiler creating a wrapper method to access a private method of another class in the same nest. For example, a nested class `C.D` can access private members within other related classes such as `C`, `C.D.E`, or `C.B`, but the Java compiler may need to generate wrapper methods in those related classes. In such cases, a `Lookup` object on `C.E` would be unable to access those private members. A workaround for this limitation is the `Lookup.in` method, which can transform a lookup on `C.E` into one on any of those other classes, without special elevation of privilege.

The accesses permitted to a given lookup object may be limited, according to its set of `lookupModes`, to a subset of members normally accessible to the lookup class. For example, the `publicLookup` method produces a lookup object which is only allowed to access public members in public classes of exported packages. The caller sensitive method `lookup` produces a lookup object with full capabilities relative to its caller class, to emulate all supported bytecode behaviors. Also, the `Lookup.in` method may produce a lookup object with fewer access modes than the original lookup object.

Discussion of private and module access: We say that a lookup has *private access* if its lookup modes include the possibility of accessing `private` members (which includes the private members of nestmates). As documented in the

relevant methods elsewhere, only lookups with private access possess the following capabilities:

- access private fields, methods, and constructors of the lookup class and its nestmates
- create method handles which `emulate invokespecial` instructions
- avoid package access checks for classes accessible to the lookup class
- create `delegated lookup objects` which have private access to other classes within the same package member

Similarly, a lookup with module access ensures that the original lookup creator was a member in the same module as the lookup class.

Private and module access are independently determined modes; a lookup may have either or both or neither. A lookup which possesses both access modes is said to possess full privilege access.

A lookup with *original access* ensures that this lookup is created by the original lookup class and the bootstrap method invoked by the VM. Such a lookup with original access also has private and module access which has the following additional capability:

- create method handles which invoke caller sensitive methods, such as `Class.forName`
- obtain the class data associated with the lookup class

Each of these permissions is a consequence of the fact that a lookup object with private access can be securely traced back to an originating class, whose bytecode behaviors and Java language access permissions can be reliably determined and emulated by method handles.

Cross-module lookups

When a lookup class in one module `M1` accesses a class in another module `M2`, extra access checking is performed beyond the access mode bits. A `Lookup` with `PUBLIC` mode and a lookup class in `M1` can access public types in `M2` when `M2` is readable to `M1` and when the type is in a package of `M2` that is exported to at least `M1`.

A `Lookup` on `C` can also *teleport* to a target class via `Lookup.in` and `MethodHandles.privateLookupIn` methods. Teleporting across modules will always record the original lookup class as the *previous lookup class* and drops `MODULE` access. If the target class is in the same module as the lookup class `C`, then the target class becomes the new lookup class and there is no change to the previous lookup class. If the target class is in a different module from `M1` (`C`'s module), `C` becomes the new previous lookup class and the target class becomes the new lookup class. In that case, if there was already a previous lookup class in `M0`, and it differs from `M1` and `M2`, then the resulting lookup drops all privileges. For example,

```
Lookup lookup = MethodHandles.lookup();   // in class C
Lookup lookup2 = lookup.in(D.class);
MethodHandle mh = lookup2.findStatic(E.class, "m", MT);
```

The `MethodHandles.lookup()` factory method produces a `Lookup` object with `null` previous lookup class. `lookup.in(D.class)` transforms the `lookup` on class `C` to class `D` without elevation of privileges. If `C` and `D` are in the same module, `lookup2` records `D` as the new lookup class and keeps the same previous lookup class as the original `lookup`, or `null` if not present.

When a `Lookup` teleports from a class in one nest to another nest, `PRIVATE` access is dropped. When a `Lookup` teleports from a class in one package to another package, `PACKAGE` access is dropped. When a `Lookup` teleports from a class in one module to another module, `MODULE` access is dropped. Teleporting across modules drops the ability to access non-exported classes in both the module of the new lookup class and the module of the old lookup class and the resulting `Lookup` remains only `PUBLIC` access. A `Lookup` can teleport back and forth to a class in the module of the lookup class and the module of the previous class lookup. Teleporting across modules can only decrease access but cannot increase it. Teleporting to some third module drops all accesses.

In the above example, if `C` and `D` are in different modules, `lookup2` records `D` as its lookup class and `C` as its previous lookup class and `lookup2` has only `PUBLIC` access. `lookup2` can teleport to other

class in C's module and D's module. If class E is in a third module, `lookup2.in(E.class)` creates a `Lookup` on E with no access and `lookup2`'s lookup class D is recorded as its previous lookup class.

Teleporting across modules restricts access to the public types that both the lookup class and the previous lookup class can equally access (see below).

`MethodHandles.privateLookupIn(T.class, lookup)` can be used to teleport a `lookup` from class C to class T and create a new `Lookup` with private access if the lookup class is allowed to do *deep reflection* on T. The `lookup` must have `MODULE` and `PRIVATE` access to call `privateLookupIn`. A `lookup` on C in module M1 is allowed to do deep reflection on all classes in M1. If T is in M1, `privateLookupIn` produces a new `Lookup` on T with full capabilities. A `lookup` on C is also allowed to do deep reflection on T in another module M2 if M1 reads M2 and M2 `opens` the package containing T to at least M1. T becomes the new lookup class and C becomes the new previous lookup class and `MODULE` access is dropped from the resulting `Lookup`. The resulting `Lookup` can be used to do member lookup or teleport to another lookup class by calling `Lookup::in`. But it cannot be used to obtain another private `Lookup` by calling `privateLookupIn` because it has no `MODULE` access.

Cross-module access checks

A `Lookup` with `PUBLIC` or with `UNCONDITIONAL` mode allows cross-module access. The access checking is performed with respect to both the lookup class and the previous lookup class if present.

A `Lookup` with `UNCONDITIONAL` mode can access public type in all modules when the type is in a package that is exported unconditionally.

If a `Lookup` on `LC` in M1 has no previous lookup class, the lookup with `PUBLIC` mode can access all public types in modules that are readable to M1 and the type is in a package that is exported at least to M1.

If a `Lookup` on `LC` in M1 has a previous lookup class `PLC` on M0, the lookup with `PUBLIC` mode can access the intersection of all public types that are accessible to M1 with all public types that are accessible to M0. M0 reads M1 and hence the set of accessible types includes:

Equally accessible types to M0 and M1
unconditional-exported packages from M1
unconditional-exported packages from M0 if M1 reads M0
unconditional-exported packages from a third module M2 if both M0 and M1 read M2
qualified-exported packages from M1 to M0
qualified-exported packages from M0 to M1 if M1 reads M0
qualified-exported packages from a third module M2 to both M0 and M1 if both M0 and M1 read M2

Access modes

The table below shows the access modes of a `Lookup` produced by any of the following factory or transformation methods:

- `MethodHandles::lookup`
- `MethodHandles::publicLookup`
- `MethodHandles::privateLookupIn`
- `Lookup::in`
- `Lookup::dropLookupMode`

Lookup object	original	protected	private	package	module	public
`CL = MethodHandles.lookup()` in C	ORI	PRO	PRI	PAC	MOD	1R
`CL.in(C1)` same package				PAC	MOD	1R

Lookup object	original	protected	private	package	module	public
`CL.in(C1)` same module					MOD	1R
`CL.in(D)` different module						2R
`CL.in(D).in(C)` hop back to module						2R
`PRI1 = privateLookupIn(C1,CL)`		PRO	PRI	PAC	MOD	1R
`PRI1a = privateLookupIn(C,PRI1)`		PRO	PRI	PAC	MOD	1R
`PRI1.in(C1)` same package				PAC	MOD	1R
`PRI1.in(C1)` different package					MOD	1R
`PRI1.in(D)` different module						2R
`PRI1.dropLookupMode(PROTECTED)`			PRI	PAC	MOD	1R
`PRI1.dropLookupMode(PRIVATE)`				PAC	MOD	1R
`PRI1.dropLookupMode(PACKAGE)`					MOD	1R
`PRI1.dropLookupMode(MODULE)`						1R
`PRI1.dropLookupMode(PUBLIC)`						none
`PRI2 = privateLookupIn(D,CL)`		PRO	PRI	PAC		2R
`privateLookupIn(D,PRI1)`		PRO	PRI	PAC		2R
`privateLookupIn(C,PRI2)` fails						IAE
`PRI2.in(D2)` same package				PAC		2R
`PRI2.in(D2)` different package						2R
`PRI2.in(C1)` hop back to module						2R
`PRI2.in(E)` hop to third module						none
`PRI2.dropLookupMode(PROTECTED)`			PRI	PAC		2R
`PRI2.dropLookupMode(PRIVATE)`				PAC		2R
`PRI2.dropLookupMode(PACKAGE)`						2R
`PRI2.dropLookupMode(MODULE)`						2R
`PRI2.dropLookupMode(PUBLIC)`						none
`CL.dropLookupMode(PROTECTED)`			PRI	PAC	MOD	1R
`CL.dropLookupMode(PRIVATE)`				PAC	MOD	1R
`CL.dropLookupMode(PACKAGE)`					MOD	1R
`CL.dropLookupMode(MODULE)`						1R
`CL.dropLookupMode(PUBLIC)`						none
`PUB = publicLookup()`						U
`PUB.in(D)` different module						U
`PUB.in(D).in(E)` third module						U
`PUB.dropLookupMode(UNCONDITIONAL)`						none
`privateLookupIn(C1,PUB)` fails						IAE
`ANY.in(X)`, for inaccessible `X`						none

Notes:

- Class `C` and class `C1` are in module `M1`, but `D` and `D2` are in module `M2`, and `E` is in module `M3`. `X` stands for class which is inaccessible to the lookup. `ANY` stands for any of the example lookups.
- `ORI` indicates `ORIGINAL` bit set, `PRO` indicates `PROTECTED` bit set, `PRI` indicates `PRIVATE` bit set, `PAC` indicates `PACKAGE` bit set, `MOD` indicates `MODULE` bit set, `1R` and `2R` indicate `PUBLIC` bit set, `U` indicates `UNCONDITIONAL` bit set, `IAE` indicates `IllegalAccessException` thrown.
- Public access comes in three kinds:
  - unconditional (`U`): the lookup assumes readability. The lookup has `null` previous lookup class.

    - one-module-reads (`1R`): the module access checking is performed with respect to the lookup class. The lookup has `null` previous lookup class.
    - two-module-reads (`2R`): the module access checking is performed with respect to the lookup class and the previous lookup class. The lookup has a non-null previous lookup class which is in a different module from the current lookup class.
- Any attempt to reach a third module loses all access.
- If a target class `X` is not accessible to `Lookup::in` all access modes are dropped.

Security manager interactions

Although bytecode instructions can only refer to classes in a related class loader, this API can search for methods in any class, as long as a reference to its `Class` object is available. Such cross-loader references are also possible with the Core Reflection API, and are impossible to bytecode instructions such as `invokestatic` or `getfield`. There is a security manager API to allow applications to check such cross-loader references. These checks apply to both the `MethodHandles.Lookup` API and the Core Reflection API (as found on `Class`).

If a security manager is present, member and class lookups are subject to additional checks. From one to three calls are made to the security manager. Any of these calls can refuse access by throwing a `SecurityException`. Define `smgr` as the security manager, `lookc` as the lookup class of the current lookup object, `refc` as the containing class in which the member is being sought, and `defc` as the class in which the member is actually defined. (If a class or other type is being accessed, the `refc` and `defc` values are the class itself.) The value `lookc` is defined as *not present* if the current lookup object does not have full privilege access. The calls are made according to the following rules:

- **Step 1:** If `lookc` is not present, or if its class loader is not the same as or an ancestor of the class loader of `refc`, then `smgr.checkPackageAccess(refcPkg)` is called, where `refcPkg` is the package of `refc`.
- **Step 2a:** If the retrieved member is not public and `lookc` is not present, then `smgr.checkPermission` with `RuntimePermission("accessDeclaredMembers")` is called.
- **Step 2b:** If the retrieved class has a `null` class loader, and `lookc` is not present, then `smgr.checkPermission` with `RuntimePermission("getClassLoader")` is called.
- **Step 3:** If the retrieved member is not public, and if `lookc` is not present, and if `defc` and `refc` are different, then `smgr.checkPackageAccess(defcPkg)` is called, where `defcPkg` is the package of `defc`.

Security checks are performed after other access checks have passed. Therefore, the above rules presuppose a member or class that is public, or else that is being accessed from a lookup class that has rights to access the member or class.

If a security manager is present and the current lookup object does not have full privilege access, then `defineClass`, `defineHiddenClass`, `defineHiddenClassWithClassData` calls `smgr.checkPermission` with `RuntimePermission("defineClass")`.

Caller sensitive methods

A small number of Java methods have a special property called caller sensitivity. A *caller-sensitive* method can behave differently depending on the identity of its immediate caller.

If a method handle for a caller-sensitive method is requested, the general rules for bytecode behaviors apply, but they take account of the lookup class in a special way. The resulting method handle behaves as if it were called from an instruction contained in the lookup class, so that the caller-sensitive method detects the lookup class. (By contrast, the invoker of the method handle is disregarded.) Thus, in the case of caller-sensitive methods, different lookup classes may give rise to differently behaving method handles.

In cases where the lookup object is `publicLookup()`, or some other lookup object without the original access, the lookup class is disregarded. In such cases, no caller-sensitive method handle can be created, access is forbidden, and the lookup fails with an `IllegalAccessException`.

Discussion: For example, the caller-sensitive method `Class.forName(x)` can return varying classes or throw varying exceptions, depending on the class loader of the class that calls it. A public lookup of `Class.forName` will fail, because there is no reasonable way to determine its bytecode behavior.

If an application caches method handles for broad sharing, it should use `publicLookup()` to create them. If there is a lookup of `Class.forName`, it will fail, and the application must take appropriate action in that case. It may be that a later lookup, perhaps during the invocation of a bootstrap method, can incorporate the specific identity of the caller, making the method accessible.

The function `MethodHandles.lookup` is caller sensitive so that there can be a secure foundation for lookups. Nearly all other methods in the JSR 292 API rely on lookup objects to check access requests.

Nested Class Summary

Nested Classes

Modifier and Type	Class	Description
static class	`MethodHandles.Lookup.ClassOption`	The set of class options that specify whether a hidden class created by `Lookup::defineHiddenClass` method is dynamically added as a new member to the nest of a lookup class and/or whether a hidden class has a strong relationship with the class loader marked as its defining loader.

Field Summary

Fields

Modifier and Type	Field	Description
static int	`MODULE`	A single-bit mask representing `module` access, which may contribute to the result of `lookupModes`.
static int	`ORIGINAL`	A single-bit mask representing `original` access which may contribute to the result of `lookupModes`.
static int	`PACKAGE`	A single-bit mask representing `package` access (default access), which may contribute to the result of `lookupModes`.
static int	`PRIVATE`	A single-bit mask representing `private` access, which may contribute to the result of `lookupModes`.
static int	`PROTECTED`	A single-bit mask representing `protected` access, which may contribute to the result of `lookupModes`.
static int	`PUBLIC`	A single-bit mask representing `public` access, which may contribute to the result of `lookupModes`.
static int	`UNCONDITIONAL`	A single-bit mask representing `unconditional` access

which may contribute to the result of `lookupModes`.

Method Summary

All Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method	Description
`Class<?>`	`accessClass(Class<?> targetClass)`	Determines if a class can be accessed from the lookup context defined by this `Lookup` object.
`MethodHandle`	`bind(Object receiver, String name, MethodType type)`	Produces an early-bound method handle for a non-static method.
`Class<?>`	`defineClass(byte[] bytes)`	Creates and links a class or interface from `bytes` with the same class loader and in the same runtime package and protection domain as this lookup's lookup class as if calling `ClassLoader::defineClass`.
`MethodHandles.Lookup`	`defineHiddenClass(byte[] bytes, boolean initialize, MethodHandles.Lookup.ClassOpti`	Creates a *hidden* class or interface from `bytes`, returning a `Lookup` on the newly created class or interface.
`MethodHandles.Lookup`	`defineHiddenClassWithClassData(byte[] bytes, Object classData, boolean initialize, MethodHandles.Lookup.ClassOpti`	Creates a *hidden* class or interface from `bytes` with associated class data, returning a `Lookup` on the newly created class or interface.
`MethodHandles.Lookup`	`dropLookupMode(int modeToDrop)`	Creates a lookup on the same lookup class which this lookup object finds members, but with a lookup mode that has lost the given lookup mode.
`Class<?>`	`ensureInitialized(Class<?> targetClass)`	Ensures that `targetClass` has been initialized.
`Class<?>`	`findClass(String targetName)`	Looks up a class by name from the lookup context defined by this `Lookup` object, as if resolved by an `ldc` instruction.
`MethodHandle`	`findConstructor(Class<?> refc, MethodType type)`	Produces a method handle which creates an object and initializes it, using the constructor of the specified type.
`MethodHandle`	`findGetter(Class<?> refc,`	Produces a method handle giving

	String name, Class<?> type)	read access to a non-static field.
MethodHandle	findSetter(Class<?> refc, String name, Class<?> type)	Produces a method handle giving write access to a non-static field.
MethodHandle	findSpecial(Class<?> refc, String name, MethodType type, Class<?> specialCaller)	Produces an early-bound method handle for a virtual method.
MethodHandle	findStatic(Class<?> refc, String name, MethodType type)	Produces a method handle for a static method.
MethodHandle	findStaticGetter(Class<?> refc, String name, Class<?> type)	Produces a method handle giving read access to a static field.
MethodHandle	findStaticSetter(Class<?> refc, String name, Class<?> type)	Produces a method handle giving write access to a static field.
VarHandle	findStaticVarHandle(Class<?> decl, String name, Class<?> type)	Produces a VarHandle giving access to a static field `name` of type `type` declared in a class of type `decl`.
VarHandle	findVarHandle(Class<?> recv, String name, Class<?> type)	Produces a VarHandle giving access to a non-static field `name` of type `type` declared in a class of type `recv`.
MethodHandle	findVirtual(Class<?> refc, String name, MethodType type)	Produces a method handle for a virtual method.
boolean	hasFullPrivilegeAccess()	Returns `true` if this lookup has *full privilege access*, i.e.
boolean	hasPrivateAccess()	**Deprecated.** This method was originally designed to test `PRIVATE` access that implies full privilege access but `MODULE` access has since become independent of `PRIVATE` access.
MethodHandles.Lookup	in(Class<?> requestedLookupClass)	Creates a lookup on the specified new lookup class.
Class<?>	lookupClass()	Tells which class is performing the lookup.
int	lookupModes()	Tells which access-protection classes of members this lookup object can produce.
Class<?>	previousLookupClass()	Reports a lookup class in another module that this lookup object was previously teleported from, or `null`.
MethodHandleInfo	revealDirect (MethodHandle target)	Cracks a direct method handle created by this lookup object or

	(MethodHandle target)	created by this lookup object or a similar one.
String	toString()	Displays the name of the class from which lookups are to be made, followed by "/" and the name of the previous lookup class if present.
MethodHandle	unreflect(Method m)	Makes a direct method handle to *m*, if the lookup class has permission.
MethodHandle	unreflectConstructor (Constructor<?> c)	Produces a method handle for a reflected constructor.
MethodHandle	unreflectGetter(Field f)	Produces a method handle giving read access to a reflected field.
MethodHandle	unreflectSetter(Field f)	Produces a method handle giving write access to a reflected field.
MethodHandle	unreflectSpecial(Method m, Class<?> specialCaller)	Produces a method handle for a reflected method.
VarHandle	unreflectVarHandle(Field f)	Produces a VarHandle giving access to a reflected field `f` of type `T` declared in a class of type `R`.

Methods declared in class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Details

PUBLIC

```
public static final int PUBLIC
```

A single-bit mask representing `public` access, which may contribute to the result of `lookupModes`. The value, `0x01`, happens to be the same as the value of the `public` modifier bit.

A `Lookup` with this lookup mode performs cross-module access check with respect to the lookup class and previous lookup class if present.

See Also:

Constant Field Values

PRIVATE

```
public static final int PRIVATE
```

A single-bit mask representing `private` access, which may contribute to the result of `lookupModes`. The value, `0x02`, happens to be the same as the value of the `private` modifier

bit.

See Also:

Constant Field Values

PROTECTED

`public static final int PROTECTED`

A single-bit mask representing `protected` access, which may contribute to the result of `lookupModes`. The value, `0x04`, happens to be the same as the value of the `protected` modifier bit.

See Also:

Constant Field Values

PACKAGE

`public static final int PACKAGE`

A single-bit mask representing `package` access (default access), which may contribute to the result of `lookupModes`. The value is `0x08`, which does not correspond meaningfully to any particular modifier bit.

See Also:

Constant Field Values

MODULE

`public static final int MODULE`

A single-bit mask representing `module` access, which may contribute to the result of `lookupModes`. The value is `0x10`, which does not correspond meaningfully to any particular modifier bit. In conjunction with the `PUBLIC` modifier bit, a `Lookup` with this lookup mode can access all public types in the module of the lookup class and public types in packages exported by other modules to the module of the lookup class.

If this lookup mode is set, the previous lookup class is always `null`.

Since:

9

See Also:

Constant Field Values

UNCONDITIONAL

`public static final int UNCONDITIONAL`

A single-bit mask representing `unconditional` access which may contribute to the result of `lookupModes`. The value is `0x20`, which does not correspond meaningfully to any particular modifier bit. A `Lookup` with this lookup mode assumes readability. This lookup mode can

access all public members of public types of all modules when the type is in a package that is `exported unconditionally`.

If this lookup mode is set, the previous lookup class is always `null`.

Since:

9

See Also:

`MethodHandles.publicLookup()`, Constant Field Values

ORIGINAL

```
public static final int ORIGINAL
```

A single-bit mask representing `original` access which may contribute to the result of `lookupModes`. The value is `0x40`, which does not correspond meaningfully to any particular modifier bit.

If this lookup mode is set, the `Lookup` object must be created by the original lookup class by calling `MethodHandles.lookup()` method or by a bootstrap method invoked by the VM. The `Lookup` object with this lookup mode has full privilege access.

Since:

16

See Also:

Constant Field Values

Method Details

lookupClass

```
public Class<?> lookupClass()
```

Tells which class is performing the lookup. It is this class against which checks are performed for visibility and access permissions.

If this lookup object has a previous lookup class, access checks are performed against both the lookup class and the previous lookup class.

The class implies a maximum level of access permission, but the permissions may be additionally limited by the bitmask `lookupModes`, which controls whether non-public members can be accessed.

Returns:

the lookup class, on behalf of which this lookup object finds members

See Also:

Cross-module lookups

previousLookupClass

```
public Class<?> previousLookupClass()
```

Reports a lookup class in another module that this lookup object was previously teleported from, or `null`.

A `Lookup` object produced by the factory methods, such as the `lookup()` and `publicLookup()` method, has `null` previous lookup class. A `Lookup` object has a non-null previous lookup class when this lookup was teleported from an old lookup class in one module to a new lookup class in another module.

Returns:

the lookup class in another module that this lookup object was previously teleported from, or `null`

Since:

14

See Also:

`in(Class)`, `MethodHandles.privateLookupIn(Class, Lookup)`, Cross-module lookups

lookupModes

```
public int lookupModes()
```

Tells which access-protection classes of members this lookup object can produce. The result is a bit-mask of the bits PUBLIC (0x01), PRIVATE (0x02), PROTECTED (0x04), PACKAGE (0x08), MODULE (0x10), UNCONDITIONAL (0x20), and ORIGINAL (0x40).

A freshly-created lookup object on the caller's class has all possible bits set, except `UNCONDITIONAL`. A lookup object on a new lookup class created from a previous lookup object may have some mode bits set to zero. Mode bits can also be directly cleared. Once cleared, mode bits cannot be restored from the downgraded lookup object. The purpose of this is to restrict access via the new lookup object, so that it can access only names which can be reached by the original lookup object, and also by the new lookup class.

Returns:

the lookup modes, which limit the kinds of access performed by this lookup object

See Also:

`in(java.lang.Class<?>)`, `dropLookupMode(int)`

in

```
public MethodHandles.Lookup in(Class<?> requestedLookupClass)
```

Creates a lookup on the specified new lookup class. The resulting object will report the specified class as its own `lookupClass`.

However, the resulting `Lookup` object is guaranteed to have no more access capabilities than the original. In particular, access capabilities can be lost as follows:

- If the new lookup class is different from the old lookup class, i.e. `ORIGINAL` access is lost.
- If the new lookup class is in a different module from the old one, i.e. `MODULE` access is lost.
- If the new lookup class is in a different package than the old one, protected and default (package) members will not be accessible, i.e. `PROTECTED` and `PACKAGE` access are lost.
- If the new lookup class is not within the same package member as the old one, private members will not be accessible, and protected members will not be accessible by virtue

of inheritance, i.e. PRIVATE access is lost. (Protected members may continue to be accessible because of package sharing.)
- If the new lookup class is not accessible to this lookup, then no members, not even public members, will be accessible i.e. all access modes are lost.
- If the new lookup class, the old lookup class and the previous lookup class are all in different modules i.e. teleporting to a third module, all access modes are lost.

The new previous lookup class is chosen as follows:

- If the new lookup object has UNCONDITIONAL bit, the new previous lookup class is null.
- If the new lookup class is in the same module as the old lookup class, the new previous lookup class is the old previous lookup class.
- If the new lookup class is in a different module from the old lookup class, the new previous lookup class is the old lookup class.

The resulting lookup's capabilities for loading classes (used during findClass(java.lang.String) invocations) are determined by the lookup class' loader, which may change due to this operation.

Parameters:

requestedLookupClass - the desired lookup class for the new lookup object

Returns:

a lookup object which reports the desired lookup class, or the same object if there is no change

Throws:

IllegalArgumentException - if requestedLookupClass is a primitive type or void or array class

NullPointerException - if the argument is null

See Also:

accessClass(Class), Cross-module lookups

dropLookupMode

```
public MethodHandles.Lookup dropLookupMode(int modeToDrop)
```

Creates a lookup on the same lookup class which this lookup object finds members, but with a lookup mode that has lost the given lookup mode. The lookup mode to drop is one of PUBLIC, MODULE, PACKAGE, PROTECTED, PRIVATE, ORIGINAL, or UNCONDITIONAL.

If this lookup is a public lookup, this lookup has UNCONDITIONAL mode set and it has no other mode set. When dropping UNCONDITIONAL on a public lookup then the resulting lookup has no access.

If this lookup is not a public lookup, then the following applies regardless of its lookup modes. PROTECTED and ORIGINAL are always dropped and so the resulting lookup mode will never have these access capabilities. When dropping PACKAGE then the resulting lookup will not have PACKAGE or PRIVATE access. When dropping MODULE then the resulting lookup will not have MODULE, PACKAGE, or PRIVATE access. When dropping PUBLIC then the resulting lookup has no access.

API Note:

A lookup with PACKAGE but not PRIVATE mode can safely delegate non-public access within the package of the lookup class without conferring private access. A lookup with MODULE but not PACKAGE mode can safely delegate PUBLIC access within the module of the lookup class without conferring package access. A lookup with a previous lookup class (and PUBLIC but not MODULE

mode) can safely delegate access to public classes accessible to both the module of the lookup class and the module of the previous lookup class.

Parameters:

`modeToDrop` - the lookup mode to drop

Returns:

a lookup object which lacks the indicated mode, or the same object if there is no change

Throws:

`IllegalArgumentException` - if `modeToDrop` is not one of `PUBLIC`, `MODULE`, `PACKAGE`, `PROTECTED`, `PRIVATE`, `ORIGINAL` or `UNCONDITIONAL`

Since:

9

See Also:

`MethodHandles.privateLookupIn(java.lang.Class<?>, java.lang.invoke.MethodHandles.Lookup)`

defineClass

```
public Class<?> defineClass(byte[] bytes)
                     throws IllegalAccessException
```

Creates and links a class or interface from `bytes` with the same class loader and in the same runtime package and protection domain as this lookup's lookup class as if calling `ClassLoader::defineClass`.

The lookup modes for this lookup must include `PACKAGE` access as default (package) members will be accessible to the class. The `PACKAGE` lookup mode serves to authenticate that the lookup object was created by a caller in the runtime package (or derived from a lookup originally created by suitably privileged code to a target class in the runtime package).

The `bytes` parameter is the class bytes of a valid class file (as defined by the *The Java Virtual Machine Specification*) with a class name in the same package as the lookup class.

This method does not run the class initializer. The class initializer may run at a later time, as detailed in section 12.4 of the *The Java Language Specification*.

If there is a security manager and this lookup does not have full privilege access, its `checkPermission` method is first called to check `RuntimePermission("defineClass")`.

Parameters:

`bytes` - the class bytes

Returns:

the `Class` object for the class

Throws:

`IllegalAccessException` - if this lookup does not have `PACKAGE` access

`ClassFormatError` - if `bytes` is not a `ClassFile` structure

`IllegalArgumentException` - if `bytes` denotes a class in a different package than the lookup class or `bytes` is not a class or interface (`ACC_MODULE` flag is set in the value of the `access_flags` item)

`VerifyError` - if the newly created class cannot be verified

`LinkageError` - if the newly created class cannot be linked for any other reason

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if `bytes` is `null`

Since:

9

See Also:

`MethodHandles.privateLookupIn(java.lang.Class<?>, java.lang.invoke.MethodHandles.Lookup)`, `dropLookupMode(int)`, `ClassLoader.defineClass(String,byte[],int,int,ProtectionDomain)`

defineHiddenClass

```
public MethodHandles.Lookup defineHiddenClass
(byte[] bytes,
 boolean initialize,
 MethodHandles.Lookup.ClassOption... options)
                                      throws IllegalAccessException
```

Creates a *hidden* class or interface from `bytes`, returning a `Lookup` on the newly created class or interface.

Ordinarily, a class or interface `C` is created by a class loader, which either defines `C` directly or delegates to another class loader. A class loader defines `C` directly by invoking `ClassLoader::defineClass`, which causes the Java Virtual Machine to derive `C` from a purported representation in `class` file format. In situations where use of a class loader is undesirable, a class or interface `C` can be created by this method instead. This method is capable of defining `C`, and thereby creating it, without invoking `ClassLoader::defineClass`. Instead, this method defines `C` as if by arranging for the Java Virtual Machine to derive a nonarray class or interface `C` from a purported representation in `class` file format using the following rules:

1. The lookup modes for this `Lookup` must include full privilege access. This level of access is needed to create `C` in the module of the lookup class of this `Lookup`.
2. The purported representation in `bytes` must be a `ClassFile` structure of a supported major and minor version. The major and minor version may differ from the `class` file version of the lookup class of this `Lookup`.
3. The value of `this_class` must be a valid index in the `constant_pool` table, and the entry at that index must be a valid `CONSTANT_Class_info` structure. Let `N` be the binary name encoded in internal form that is specified by this structure. `N` must denote a class or interface in the same package as the lookup class.
4. Let `CN` be the string `N + "." + <suffix>`, where `<suffix>` is an unqualified name.

   Let `newBytes` be the `ClassFile` structure given by `bytes` with an additional entry in the `constant_pool` table, indicating a `CONSTANT_Utf8_info` structure for `CN`, and where the `CONSTANT_Class_info` structure indicated by `this_class` refers to the new `CONSTANT_Utf8_info` structure.

   Let `L` be the defining class loader of the lookup class of this `Lookup`.

   `C` is derived with name `CN`, class loader `L`, and purported representation `newBytes` as if by the rules of JVMS 5.3.5, with the following adjustments:

   - The constant indicated by `this_class` is permitted to specify a name that includes a single `"."` character, even though this is not a valid binary class or interface name in internal form.

- The Java Virtual Machine marks `L` as the defining class loader of `C`, but no class loader is recorded as an initiating class loader of `C`.
- `C` is considered to have the same runtime package, module and protection domain as the lookup class of this `Lookup`.
- Let `GN` be the binary name obtained by taking `N` (a binary name encoded in internal form) and replacing ASCII forward slashes with ASCII periods. For the instance of `Class` representing `C`:
  - `Class.getName()` returns the string `GN + "/" + <suffix>`, even though this is not a valid binary class or interface name.
  - `Class.descriptorString()` returns the string `"L" + N + "." + <suffix> + ";"`, even though this is not a valid type descriptor name.
  - `Class.describeConstable()` returns an empty optional as `C` cannot be described in nominal form.

After `C` is derived, it is linked by the Java Virtual Machine. Linkage occurs as specified in JVMS 5.4.3, with the following adjustments:

- During verification, whenever it is necessary to load the class named `CN`, the attempt succeeds, producing class `C`. No request is made of any class loader.
- On any attempt to resolve the entry in the run-time constant pool indicated by `this_class`, the symbolic reference is considered to be resolved to `C` and resolution always succeeds immediately.

If the `initialize` parameter is `true`, then `C` is initialized by the Java Virtual Machine.

The newly created class or interface `C` serves as the lookup class of the `Lookup` object returned by this method. `C` is *hidden* in the sense that no other class or interface can refer to `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. A hidden class or interface is not discoverable by `Class.forName(String, boolean, ClassLoader)`, `ClassLoader.loadClass(String, boolean)`, or `findClass(String)`, and is not modifiable by Java agents or tool agents using the JVM Tool Interface.

A class or interface created by a class loader has a strong relationship with that class loader. That is, every `Class` object contains a reference to the `ClassLoader` that defined it. This means that a class created by a class loader may be unloaded if and only if its defining loader is not reachable and thus may be reclaimed by a garbage collector (JLS 12.7). By default, however, a hidden class or interface may be unloaded even if the class loader that is marked as its defining loader is reachable. This behavior is useful when a hidden class or interface serves multiple classes defined by arbitrary class loaders. In other cases, a hidden class or interface may be linked to a single class (or a small number of classes) with the same defining loader as the hidden class or interface. In such cases, where the hidden class or interface must be coterminous with a normal class or interface, the `STRONG` option may be passed in `options`. This arranges for a hidden class to have the same strong relationship with the class loader marked as its defining loader, as a normal class or interface has with its own defining loader. If `STRONG` is not used, then the invoker of `defineHiddenClass` may still prevent a hidden class or interface from being unloaded by ensuring that the `Class` object is reachable.

The unloading characteristics are set for each hidden class when it is defined, and cannot be changed later. An advantage of allowing hidden classes to be unloaded independently of the class loader marked as their defining loader is that a very large number of hidden classes may be created by an application. In contrast, if `STRONG` is used, then the JVM may run out of memory, just as if normal classes were created by class loaders.

Classes and interfaces in a nest are allowed to have mutual access to their private members. The nest relationship is determined by the `NestHost` attribute (JVMS 4.7.28) and the

`NestMembers` attribute (JVMS 4.7.29) in a `class` file. By default, a hidden class belongs to a nest consisting only of itself because a hidden class has no binary name. The `NESTMATE` option can be passed in `options` to create a hidden class or interface `C` as a member of a nest. The nest to which `C` belongs is not based on any `NestHost` attribute in the `ClassFile` structure from which `C` was derived. Instead, the following rules determine the nest host of `C`:

- If the nest host of the lookup class of this `Lookup` has previously been determined, then let `H` be the nest host of the lookup class. Otherwise, the nest host of the lookup class is determined using the algorithm in JVMS 5.4.4, yielding `H`.
- The nest host of `C` is determined to be `H`, the nest host of the lookup class.

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.

Parameters:

`bytes` - the bytes that make up the class data, in the format of a valid `class` file as defined by *The Java Virtual Machine Specification*.

`initialize` - if `true` the class will be initialized.

`options` - class options

Returns:

the `Lookup` object on the hidden class, with original and full privilege access

Throws:

`IllegalAccessException` - if this `Lookup` does not have full privilege access

`SecurityException` - if a security manager is present and it refuses access

`ClassFormatError` - if `bytes` is not a `ClassFile` structure

`UnsupportedClassVersionError` - if `bytes` is not of a supported major or minor version

`IllegalArgumentException` - if `bytes` denotes a class in a different package than the lookup class or `bytes` is not a class or interface (`ACC_MODULE` flag is set in the value of the `access_flags` item)

`IncompatibleClassChangeError` - if the class or interface named as the direct superclass of `C` is in fact an interface, or if any of the classes or interfaces named as direct superinterfaces of `C` are not in fact interfaces

`ClassCircularityError` - if any of the superclasses or superinterfaces of `C` is `C` itself

`VerifyError` - if the newly created class cannot be verified

`LinkageError` - if the newly created class cannot be linked for any other reason

`NullPointerException` - if any parameter is `null`

See *Java Language Specification*:

12.7 Unloading of Classes and Interfaces

See *Java Virtual Machine Specification*:

4.2.1 Binary Class and Interface Names
4.2.2 Unqualified Names
4.7.28 The `NestHost` Attribute
4.7.29 The `NestMembers` Attribute
5.4.3.1 Class and Interface Resolution
5.4.4 Access Control
5.3.5 Deriving a `Class` from a `class` File Representation

5.4 Linking
5.5 Initialization

Since:

15

See Also:

`Class.isHidden()`

defineHiddenClassWithClassData

```
public MethodHandles.Lookup defineHiddenClassWithClassData
(byte[] bytes,
 Object classData,
 boolean initialize,
 MethodHandles.Lookup.ClassOption... options)
                                                    throws IllegalAccessException
```

Creates a *hidden* class or interface from `bytes` with associated class data, returning a `Lookup` on the newly created class or interface.

This method is equivalent to calling `defineHiddenClass(bytes, initialize, options)` as if the hidden class is injected with a private static final *unnamed* field which is initialized with the given `classData` at the first instruction of the class initializer. The newly created class is linked by the Java Virtual Machine.

The `MethodHandles::classData` and `MethodHandles::classDataAt` methods can be used to retrieve the `classData`.

API Note:

A framework can create a hidden class with class data with one or more objects and load the class data as dynamically-computed constant(s) via a bootstrap method. `Class data` is accessible only to the lookup object created by the newly defined hidden class but inaccessible to other members in the same nest (unlike private static fields that are accessible to nestmates). Care should be taken w.r.t. mutability for example when passing an array or other mutable structure through the class data. Changing any value stored in the class data at runtime may lead to unpredictable behavior. If the class data is a `List`, it is good practice to make it unmodifiable for example via `List::of`.

Parameters:

`bytes` - the class bytes

`classData` - pre-initialized class data

`initialize` - if `true` the class will be initialized.

`options` - class options

Returns:

the `Lookup` object on the hidden class, with original and full privilege access

Throws:

`IllegalAccessException` - if this `Lookup` does not have full privilege access

`SecurityException` - if a security manager is present and it refuses access

`ClassFormatError` - if `bytes` is not a `ClassFile` structure

`UnsupportedClassVersionError` - if `bytes` is not of a supported major or minor version

`IllegalArgumentException` - if `bytes` denotes a class in a different package than the lookup class or `bytes` is not a class or interface (`ACC_MODULE` flag is set in the value of the `access_flags` item)

`IncompatibleClassChangeError` - if the class or interface named as the direct superclass of `C` is in fact an interface, or if any of the classes or interfaces named as direct superinterfaces of `C` are not in fact interfaces

`ClassCircularityError` - if any of the superclasses or superinterfaces of `C` is `C` itself

`VerifyError` - if the newly created class cannot be verified

`LinkageError` - if the newly created class cannot be linked for any other reason

`NullPointerException` - if any parameter is `null`

See *Java Language Specification*:

12.7 Unloading of Classes and Interface

See *Java Virtual Machine Specification*:

4.2.1 Binary Class and Interface Names
4.2.2 Unqualified Names
4.7.28 The `NestHost` Attribute
4.7.29 The `NestMembers` Attribute
5.4.3.1 Class and Interface Resolution
5.4.4 Access Control
5.3.5 Deriving a `Class` from a `class` File Representation
5.4 Linking
5.5 Initialization

Since:

16

See Also:

`defineHiddenClass(byte[], boolean, ClassOption...)`, `Class.isHidden()`, `MethodHandles.classData(Lookup, String, Class)`, `MethodHandles.classDataAt(Lookup, String, Class, int)`

toString

```
public String toString()
```

Displays the name of the class from which lookups are to be made, followed by "/" and the name of the previous lookup class if present. (The name is the one reported by `Class.getName`.) If there are restrictions on the access permitted to this lookup, this is indicated by adding a suffix to the class name, consisting of a slash and a keyword. The keyword represents the strongest allowed access, and is chosen as follows:

- If no access is allowed, the suffix is "/noaccess".
- If only unconditional access is allowed, the suffix is "/publicLookup".
- If only public access to types in exported packages is allowed, the suffix is "/public".
- If only public and module access are allowed, the suffix is "/module".
- If public and package access are allowed, the suffix is "/package".
- If public, package, and private access are allowed, the suffix is "/private".

If none of the above cases apply, it is the case that full privilege access (public, module, package, private, and protected) is allowed. In this case, no suffix is added. This is true only of an object obtained originally from `MethodHandles.lookup`. Objects created by `Lookup.in` always have restricted access, and will display a suffix.

(It may seem strange that protected access should be stronger than private access. Viewed independently from package access, protected access is the first to be lost, because it requires a direct subclass relationship between caller and callee.)

Overrides:

`toString` in class `Object`

Returns:

a string representation of the object.

See Also:

`in(java.lang.Class<?>)`

findStatic

```
public MethodHandle findStatic(Class<?> refc,
                               String name,
                               MethodType type)
                        throws NoSuchMethodException,
                               IllegalAccessException
```

Produces a method handle for a static method. The type of the method handle will be that of the method. (Since static methods do not take receivers, there is no additional receiver argument inserted into the method handle type, as there would be with `findVirtual` or `findSpecial`.) The method and all its argument types must be accessible to the lookup object.

The returned method handle will have variable arity if and only if the method's variable arity modifier bit (`0x0080`) is set.

If the returned method handle is invoked, the method's class will be initialized, if it has not already been initialized.

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle MH_asList = publicLookup().findStatic(Arrays.class,
  "asList", methodType(List.class, Object[].class));
assertEquals("[x, y]", MH_asList.invoke("x", "y").toString());
```

Parameters:

`refc` - the class from which the method is accessed

`name` - the name of the method

`type` - the type of the method

Returns:

the desired method handle

Throws:

`NoSuchMethodException` - if the method does not exist

`IllegalAccessException` - if access checking fails, or if the method is not `static`, or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

findVirtual

```
public MethodHandle findVirtual(Class<?> refc,
                                String name,
                                MethodType type)
                         throws NoSuchMethodException,
                                IllegalAccessException
```

Produces a method handle for a virtual method. The type of the method handle will be that of the method, with the receiver type (usually `refc`) prepended. The method and all its argument types must be accessible to the lookup object.

When called, the handle will treat the first argument as a receiver and, for non-private methods, dispatch on the receiver's type to determine which method implementation to enter. For private methods the named method in `refc` will be invoked on the receiver. (The dispatching action is identical with that performed by an `invokevirtual` or `invokeinterface` instruction.)

The first argument will be of type `refc` if the lookup class has full privileges to access the member. Otherwise the member must be `protected` and the first argument will be restricted in type to the lookup class.

The returned method handle will have variable arity if and only if the method's variable arity modifier bit (`0x0080`) is set.

Because of the general equivalence between `invokevirtual` instructions and method handles produced by `findVirtual`, if the class is `MethodHandle` and the name string is `invokeExact` or `invoke`, the resulting method handle is equivalent to one produced by `MethodHandles.exactInvoker` or `MethodHandles.invoker` with the same `type` argument.

If the class is `VarHandle` and the name string corresponds to the name of a signature-polymorphic access mode method, the resulting method handle is equivalent to one produced by `MethodHandles.varHandleInvoker(java.lang.invoke.VarHandle.AccessMode, java.lang.invoke.MethodType)` with the access mode corresponding to the name string and with the same `type` arguments.

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle MH_concat = publicLookup().findVirtual(String.class,
  "concat", methodType(String.class, String.class));
MethodHandle MH_hashCode = publicLookup().findVirtual(Object.class,
  "hashCode", methodType(int.class));
MethodHandle MH_hashCode_String = publicLookup().findVirtual(String.class,
  "hashCode", methodType(int.class));
assertEquals("xy", (String) MH_concat.invokeExact("x", "y"));
assertEquals("xy".hashCode(), (int) MH_hashCode.invokeExact((Object)"xy"));
assertEquals("xy".hashCode(), (int) MH_hashCode_String.invokeExact("xy"));
// interface method:
MethodHandle MH_subSequence = publicLookup().findVirtual(CharSequence.class,
  "subSequence", methodType(CharSequence.class, int.class, int.class));
assertEquals("def", MH_subSequence.invoke("abcdefghi", 3, 6).toString());
```

```
// constructor "internal method" must be accessed differently:
MethodType MT_newString = methodType(void.class); //()V for new String()
try { assertEquals("impossible", lookup()
        .findVirtual(String.class, "<init>", MT_newString));
 } catch (NoSuchMethodException ex) { } // OK
MethodHandle MH_newString = publicLookup()
  .findConstructor(String.class, MT_newString);
assertEquals("", (String) MH_newString.invokeExact());
```

Parameters:

`refc` - the class or interface from which the method is accessed

`name` - the name of the method

`type` - the type of the method, with the receiver argument omitted

Returns:

the desired method handle

Throws:

`NoSuchMethodException` - if the method does not exist

`IllegalAccessException` - if access checking fails, or if the method is `static`, or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

findConstructor

```
public MethodHandle findConstructor(Class<?> refc,
                                    MethodType type)
                             throws NoSuchMethodException,
                                    IllegalAccessException
```

Produces a method handle which creates an object and initializes it, using the constructor of the specified type. The parameter types of the method handle will be those of the constructor, while the return type will be a reference to the constructor's class. The constructor and all its argument types must be accessible to the lookup object.

The requested type must have a return type of `void`. (This is consistent with the JVM's treatment of constructor type descriptors.)

The returned method handle will have variable arity if and only if the constructor's variable arity modifier bit (`0x0080`) is set.

If the returned method handle is invoked, the constructor's class will be initialized, if it has not already been initialized.

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle MH_newArrayList = publicLookup().findConstructor(
  ArrayList.class, methodType(void.class, Collection.class));
Collection orig = Arrays.asList("x", "y");
```

```
Collection copy = (ArrayList) MH_newArrayList.invokeExact(orig);
assert(orig != copy);
assertEquals(orig, copy);
// a variable-arity constructor:
MethodHandle MH_newProcessBuilder = publicLookup().findConstructor(
  ProcessBuilder.class, methodType(void.class, String[].class));
ProcessBuilder pb = (ProcessBuilder)
  MH_newProcessBuilder.invoke("x", "y", "z");
assertEquals("[x, y, z]", pb.command().toString());
```

Parameters:

`refc` - the class or interface from which the method is accessed

`type` - the type of the method, with the receiver argument omitted, and a void return type

Returns:

the desired method handle

Throws:

`NoSuchMethodException` - if the constructor does not exist

`IllegalAccessException` - if access checking fails or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

findClass

```
public Class<?> findClass(String targetName)
                  throws ClassNotFoundException,
                         IllegalAccessException
```

Looks up a class by name from the lookup context defined by this `Lookup` object, as if resolved by an `ldc` instruction. Such a resolution, as specified in JVMS 5.4.3.1 section, attempts to locate and load the class, and then determines whether the class is accessible to this lookup object.

The lookup context here is determined by the lookup class, its class loader, and the lookup modes.

Parameters:

`targetName` - the fully qualified name of the class to be looked up.

Returns:

the requested class.

Throws:

`SecurityException` - if a security manager is present and it refuses access

`LinkageError` - if the linkage fails

`ClassNotFoundException` - if the class cannot be loaded by the lookup class' loader.

`IllegalAccessException` - if the class is not accessible, using the allowed access modes.

See *Java Virtual Machine Specification*:

5.4.3.1 Class and Interface Resolution

Since:

9

ensureInitialized

```
public Class<?> ensureInitialized(Class<?> targetClass)
                           throws IllegalAccessException
```

Ensures that `targetClass` has been initialized. The class to be initialized must be accessible to this `Lookup` object. This method causes `targetClass` to be initialized if it has not been already initialized, as specified in JVMS 5.5.

Parameters:

`targetClass` - the class to be initialized

Returns:

`targetClass` that has been initialized

Throws:

`IllegalArgumentException` - if `targetClass` is a primitive type or `void` or array class

`IllegalAccessException` - if `targetClass` is not accessible to this lookup

`ExceptionInInitializerError` - if the class initialization provoked by this method fails

`SecurityException` - if a security manager is present and it refuses access

See *Java Virtual Machine Specification*:

5.5 Initialization

Since:

15

accessClass

```
public Class<?> accessClass(Class<?> targetClass)
                     throws IllegalAccessException
```

Determines if a class can be accessed from the lookup context defined by this `Lookup` object. The static initializer of the class is not run.

If the `targetClass` is in the same module as the lookup class, the lookup class is `LC` in module `M1` and the previous lookup class is in module `M0` or `null` if not present, `targetClass` is accessible if and only if one of the following is true:

- If this lookup has `PRIVATE` access, `targetClass` is `LC` or other class in the same nest of `LC`.
- If this lookup has `PACKAGE` access, `targetClass` is in the same runtime package of `LC`.
- If this lookup has `MODULE` access, `targetClass` is a public type in `M1`.
- If this lookup has `PUBLIC` access, `targetClass` is a public type in a package exported by `M1` to at least `M0` if the previous lookup class is present; otherwise, `targetClass` is a public type in a package exported by `M1` unconditionally.

Otherwise, if this lookup has `UNCONDITIONAL` access, this lookup can access public types in all modules when the type is in a package that is exported unconditionally.

Otherwise, the target class is in a different module from `lookupClass`, and if this lookup does not have `PUBLIC` access, `lookupClass` is inaccessible.

Otherwise, if this lookup has no previous lookup class, `M1` is the module containing `lookupClass` and `M2` is the module containing `targetClass`, then `targetClass` is accessible if and only if

- `M1` reads `M2`, and
- `targetClass` is public and in a package exported by `M2` at least to `M1`.

Otherwise, if this lookup has a previous lookup class, `M1` and `M2` are as before, and `M0` is the module containing the previous lookup class, then `targetClass` is accessible if and only if one of the following is true:

- `targetClass` is in `M0` and `M1` reads `M0` and the type is in a package that is exported to at least `M1`.
- `targetClass` is in `M1` and `M0` reads `M1` and the type is in a package that is exported to at least `M0`.
- `targetClass` is in a third module `M2` and both `M0` and `M1` reads `M2` and the type is in a package that is exported to at least both `M0` and `M2`.

Otherwise, `targetClass` is not accessible.

Parameters:

`targetClass` - the class to be access-checked

Returns:

the class that has been access-checked

Throws:

`IllegalAccessException` - if the class is not accessible from the lookup class and previous lookup class, if present, using the allowed access modes.

`SecurityException` - if a security manager is present and it refuses access

Since:

9

See Also:

Cross-module lookups

findSpecial

```
public MethodHandle findSpecial(Class<?> refc,
                                String name,
                                MethodType type,
                                Class<?> specialCaller)
                         throws NoSuchMethodException,
                                IllegalAccessException
```

Produces an early-bound method handle for a virtual method. It will bypass checks for overriding methods on the receiver, as if called from an `invokespecial` instruction from within the explicitly specified `specialCaller`. The type of the method handle will be that of the method, with a suitably restricted receiver type prepended. (The receiver type will be `specialCaller` or a subtype.) The method and all its argument types must be accessible to the lookup object.

Before method resolution, if the explicitly specified caller class is not identical with the lookup class, or if this lookup object does not have private access privileges, the access fails.

The returned method handle will have variable arity if and only if the method's variable arity modifier bit (`0x0080`) is set.

(Note: JVM internal methods named "<init>" are not visible to this API, even though the invokespecial instruction can refer to them in special circumstances. Use findConstructor to access instance initialization methods in a safe manner.)

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
static class Listie extends ArrayList {
  public String toString() { return "[wee Listie]"; }
  static Lookup lookup() { return MethodHandles.lookup(); }
}
...
// no access to constructor via invokeSpecial:
MethodHandle MH_newListie = Listie.lookup()
  .findConstructor(Listie.class, methodType(void.class));
Listie l = (Listie) MH_newListie.invokeExact();
try { assertEquals("impossible", Listie.lookup().findSpecial(
        Listie.class, "<init>", methodType(void.class), Listie.class));
 } catch (NoSuchMethodException ex) { } // OK
// access to super and self methods via invokeSpecial:
MethodHandle MH_super = Listie.lookup().findSpecial(
  ArrayList.class, "toString" , methodType(String.class), Listie.class);
MethodHandle MH_this = Listie.lookup().findSpecial(
  Listie.class, "toString" , methodType(String.class), Listie.class);
MethodHandle MH_duper = Listie.lookup().findSpecial(
  Object.class, "toString" , methodType(String.class), Listie.class);
assertEquals("[]", (String) MH_super.invokeExact(l));
assertEquals(""+l, (String) MH_this.invokeExact(l));
assertEquals("[]", (String) MH_duper.invokeExact(l)); // ArrayList method
try { assertEquals("inaccessible", Listie.lookup().findSpecial(
        String.class, "toString", methodType(String.class), Listie.class));
 } catch (IllegalAccessException ex) { } // OK
Listie subl = new Listie() { public String toString() { return "[subclass]"; } };
assertEquals(""+l, (String) MH_this.invokeExact(subl)); // Listie method
```

Parameters:

`refc` - the class or interface from which the method is accessed

`name` - the name of the method (which must not be "<init>")

`type` - the type of the method, with the receiver argument omitted

`specialCaller` - the proposed calling class to perform the `invokespecial`

Returns:

the desired method handle

Throws:

`NoSuchMethodException` - if the method does not exist

`IllegalAccessException` - if access checking fails, or if the method is `static`, or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

findGetter

```
public MethodHandle findGetter(Class<?> refc,
                               String name,
                               Class<?> type)
                        throws NoSuchFieldException,
                               IllegalAccessException
```

Produces a method handle giving read access to a non-static field. The type of the method handle will have a return type of the field's value type. The method handle's single argument will be the instance containing the field. Access checking is performed immediately on behalf of the lookup class.

Parameters:

`refc` - the class or interface from which the method is accessed

`name` - the field's name

`type` - the field's type

Returns:

a method handle which can load values from the field

Throws:

`NoSuchFieldException` - if the field does not exist

`IllegalAccessException` - if access checking fails, or if the field is `static`

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

See Also:

`findVarHandle(Class, String, Class)`

findSetter

```
public MethodHandle findSetter(Class<?> refc,
                               String name,
                               Class<?> type)
                        throws NoSuchFieldException,
                               IllegalAccessException
```

Produces a method handle giving write access to a non-static field. The type of the method handle will have a void return type. The method handle will take two arguments, the instance containing the field, and the value to be stored. The second argument will be of the field's value type. Access checking is performed immediately on behalf of the lookup class.

Parameters:

`refc` - the class or interface from which the method is accessed

`name` - the field's name

`type` - the field's type

Returns:

a method handle which can store values into the field

Throws:

`NoSuchFieldException` - if the field does not exist

`IllegalAccessException` - if access checking fails, or if the field is `static` or `final`

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

See Also:

`findVarHandle(Class, String, Class)`

findVarHandle

```
public VarHandle findVarHandle(Class<?> recv,
                               String name,
                               Class<?> type)
                        throws NoSuchFieldException,
                               IllegalAccessException
```

Produces a VarHandle giving access to a non-static field `name` of type `type` declared in a class of type `recv`. The VarHandle's variable type is `type` and it has one coordinate type, `recv`.

Access checking is performed immediately on behalf of the lookup class.

Certain access modes of the returned VarHandle are unsupported under the following conditions:

- if the field is declared `final`, then the write, atomic update, numeric atomic update, and bitwise atomic update access modes are unsupported.
- if the field type is anything other than `byte`, `short`, `char`, `int`, `long`, `float`, or `double` then numeric atomic update access modes are unsupported.
- if the field type is anything other than `boolean`, `byte`, `short`, `char`, `int` or `long` then bitwise atomic update access modes are unsupported.

If the field is declared `volatile` then the returned VarHandle will override access to the field (effectively ignore the `volatile` declaration) in accordance to its specified access modes.

If the field type is `float` or `double` then numeric and atomic update access modes compare values using their bitwise representation (see `Float.floatToRawIntBits(float)` and `Double.doubleToRawLongBits(double)`, respectively).

API Note:

Bitwise comparison of `float` values or `double` values, as performed by the numeric and atomic update access modes, differ from the primitive == operator and the `Float.equals(java.lang.Object)` and `Double.equals(java.lang.Object)` methods, specifically with respect to comparing NaN values or comparing `-0.0` with `+0.0`. Care should be taken when performing a compare and set or a compare and exchange operation with such values since the operation may unexpectedly fail. There are many possible NaN values that are considered to be `NaN` in Java, although no IEEE 754 floating-point operation provided by Java can distinguish between them. Operation failure can occur if the expected or witness value is a NaN value and it is transformed (perhaps in a platform specific manner) into another NaN value, and thus has a different bitwise representation (see `Float.intBitsToFloat(int)` or `Double.longBitsToDouble(long)` for more details). The values `-0.0` and `+0.0` have different bitwise representations but are considered equal when using the primitive == operator. Operation failure can occur if, for example, a numeric algorithm computes an expected value to be say `-0.0` and previously computed the witness value to be say `+0.0`.

Parameters:

`recv` - the receiver class, of type `R`, that declares the non-static field

`name` - the field's name

`type` - the field's type, of type `T`

Returns:

a VarHandle giving access to non-static fields.

Throws:

`NoSuchFieldException` - if the field does not exist

`IllegalAccessException` - if access checking fails, or if the field is `static`

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

Since:

9

findStaticGetter

```
public MethodHandle findStaticGetter(Class<?> refc,
                                     String name,
                                     Class<?> type)
                              throws NoSuchFieldException,
                                     IllegalAccessException
```

Produces a method handle giving read access to a static field. The type of the method handle will have a return type of the field's value type. The method handle will take no arguments. Access checking is performed immediately on behalf of the lookup class.

If the returned method handle is invoked, the field's class will be initialized, if it has not already been initialized.

Parameters:

`refc` - the class or interface from which the method is accessed

`name` - the field's name

`type` - the field's type

Returns:

a method handle which can load values from the field

Throws:

`NoSuchFieldException` - if the field does not exist

`IllegalAccessException` - if access checking fails, or if the field is not `static`

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

findStaticSetter

```
public MethodHandle findStaticSetter(Class<?> refc,
                                     String name,
                                     Class<?> type)
                              throws NoSuchFieldException,
                                     IllegalAccessException
```

Produces a method handle giving write access to a static field. The type of the method handle will have a void return type. The method handle will take a single argument, of the field's value type, the value to be stored. Access checking is performed immediately on behalf of the lookup class.

If the returned method handle is invoked, the field's class will be initialized, if it has not already been initialized.

Parameters:

`refc` - the class or interface from which the method is accessed

`name` - the field's name

`type` - the field's type

Returns:

a method handle which can store values into the field

Throws:

`NoSuchFieldException` - if the field does not exist

`IllegalAccessException` - if access checking fails, or if the field is not `static` or is `final`

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

findStaticVarHandle

```
public VarHandle findStaticVarHandle(Class<?> decl,
                                     String name,
                                     Class<?> type)
                              throws NoSuchFieldException,
                                     IllegalAccessException
```

Produces a VarHandle giving access to a static field `name` of type `type` declared in a class of type `decl`. The VarHandle's variable type is `type` and it has no coordinate types.

Access checking is performed immediately on behalf of the lookup class.

If the returned VarHandle is operated on, the declaring class will be initialized, if it has not already been initialized.

Certain access modes of the returned VarHandle are unsupported under the following conditions:

- if the field is declared `final`, then the write, atomic update, numeric atomic update, and bitwise atomic update access modes are unsupported.
- if the field type is anything other than `byte`, `short`, `char`, `int`, `long`, `float`, or `double`, then numeric atomic update access modes are unsupported.
- if the field type is anything other than `boolean`, `byte`, `short`, `char`, `int` or `long` then bitwise atomic update access modes are unsupported.

If the field is declared `volatile` then the returned VarHandle will override access to the field (effectively ignore the `volatile` declaration) in accordance to its specified access modes.

If the field type is `float` or `double` then numeric and atomic update access modes compare values using their bitwise representation (see `Float.floatToRawIntBits(float)` and `Double.doubleToRawLongBits(double)`, respectively).

API Note:

Bitwise comparison of `float` values or `double` values, as performed by the numeric and atomic update access modes, differ from the primitive == operator and the `Float.equals(java.lang.Object)` and `Double.equals(java.lang.Object)` methods, specifically with respect to comparing NaN values or comparing `-0.0` with `+0.0`. Care should be taken when performing a compare and set or a compare and exchange operation with such values since the operation may unexpectedly fail. There are many possible NaN values that are considered to be `NaN` in Java, although no IEEE 754 floating-point operation provided by Java can distinguish between them. Operation failure can occur if the expected or witness value is a NaN value and it is transformed (perhaps in a platform specific manner) into another NaN value, and thus has a different bitwise representation (see `Float.intBitsToFloat(int)` or `Double.longBitsToDouble(long)` for more details). The values `-0.0` and `+0.0` have different bitwise representations but are considered equal when using the primitive == operator. Operation failure can occur if, for example, a numeric algorithm computes an expected value to be say `-0.0` and previously computed the witness value to be say `+0.0`.

Parameters:

`decl` - the class that declares the static field

`name` - the field's name

`type` - the field's type, of type `T`

Returns:

a VarHandle giving access to a static field

Throws:

`NoSuchFieldException` - if the field does not exist

`IllegalAccessException` - if access checking fails, or if the field is not `static`

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

Since:

9

bind

```
public MethodHandle bind(Object receiver,
                         String name,
                         MethodType type)
                  throws NoSuchMethodException,
                         IllegalAccessException
```

Produces an early-bound method handle for a non-static method. The receiver must have a supertype `defc` in which a method of the given name and type is accessible to the lookup class. The method and all its argument types must be accessible to the lookup object. The type of the method handle will be that of the method, without any insertion of an additional receiver parameter. The given receiver will be bound into the method handle, so that every call to the method handle will invoke the requested method on the given receiver.

The returned method handle will have variable arity if and only if the method's variable arity modifier bit (`0x0080`) is set *and* the trailing array argument is not the only argument. (If the trailing array argument is the only argument, the given receiver value will be bound to it.)

This is almost equivalent to the following code, with some differences noted below:

```
import static java.lang.invoke.MethodHandles.*;
```

```
import static java.lang.invoke.MethodType.*;
...
MethodHandle mh0 = lookup().findVirtual(defc, name, type);
MethodHandle mh1 = mh0.bindTo(receiver);
mh1 = mh1.withVarargs(mh0.isVarargsCollector());
return mh1;
```

where `defc` is either `receiver.getClass()` or a super type of that class, in which the requested method is accessible to the lookup class. (Unlike `bind`, `bindTo` does not preserve variable arity. Also, `bindTo` may throw a `ClassCastException` in instances where `bind` would throw an `IllegalAccessException`, as in the case where the member is `protected` and the receiver is restricted by `findVirtual` to the lookup class.)

Parameters:

`receiver` - the object from which the method is accessed

`name` - the name of the method

`type` - the type of the method, with the receiver argument omitted

Returns:

the desired method handle

Throws:

`NoSuchMethodException` - if the method does not exist

`IllegalAccessException` - if access checking fails or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`SecurityException` - if a security manager is present and it refuses access

`NullPointerException` - if any argument is null

See Also:

`MethodHandle.bindTo(java.lang.Object)`, `findVirtual(java.lang.Class<?>, java.lang.String, java.lang.invoke.MethodType)`

unreflect

```
public MethodHandle unreflect(Method m)
                       throws IllegalAccessException
```

Makes a direct method handle to *m*, if the lookup class has permission. If *m* is non-static, the receiver argument is treated as an initial argument. If *m* is virtual, overriding is respected on every call. Unlike the Core Reflection API, exceptions are *not* wrapped. The type of the method handle will be that of the method, with the receiver type prepended (but only if it is non-static). If the method's `accessible` flag is not set, access checking is performed immediately on behalf of the lookup class. If *m* is not public, do not share the resulting handle with untrusted parties.

The returned method handle will have variable arity if and only if the method's variable arity modifier bit (`0x0080`) is set.

If *m* is static, and if the returned method handle is invoked, the method's class will be initialized, if it has not already been initialized.

Parameters:

`m` - the reflected method

Returns:

a method handle which can invoke the reflected method

Throws:

`IllegalAccessException` - if access checking fails or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`NullPointerException` - if the argument is null

unreflectSpecial

```
public MethodHandle unreflectSpecial(Method m,
                                     Class<?> specialCaller)
                              throws IllegalAccessException
```

Produces a method handle for a reflected method. It will bypass checks for overriding methods on the receiver, as if called from an `invokespecial` instruction from within the explicitly specified `specialCaller`. The type of the method handle will be that of the method, with a suitably restricted receiver type prepended. (The receiver type will be `specialCaller` or a subtype.) If the method's `accessible` flag is not set, access checking is performed immediately on behalf of the lookup class, as if `invokespecial` instruction were being linked.

Before method resolution, if the explicitly specified caller class is not identical with the lookup class, or if this lookup object does not have private access privileges, the access fails.

The returned method handle will have variable arity if and only if the method's variable arity modifier bit (`0x0080`) is set.

Parameters:

`m` - the reflected method

`specialCaller` - the class nominally calling the method

Returns:

a method handle which can invoke the reflected method

Throws:

`IllegalAccessException` - if access checking fails, or if the method is `static`, or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`NullPointerException` - if any argument is null

unreflectConstructor

```
public MethodHandle unreflectConstructor(Constructor<?> c)
                                 throws IllegalAccessException
```

Produces a method handle for a reflected constructor. The type of the method handle will be that of the constructor, with the return type changed to the declaring class. The method handle will perform a `newInstance` operation, creating a new instance of the constructor's class on the arguments passed to the method handle.

If the constructor's `accessible` flag is not set, access checking is performed immediately on behalf of the lookup class.

The returned method handle will have variable arity if and only if the constructor's variable arity modifier bit (`0x0080`) is set.

If the returned method handle is invoked, the constructor's class will be initialized, if it has not already been initialized.

Parameters:

`c` - the reflected constructor

Returns:

a method handle which can invoke the reflected constructor

Throws:

`IllegalAccessException` - if access checking fails or if the method's variable arity modifier bit is set and `asVarargsCollector` fails

`NullPointerException` - if the argument is null

unreflectGetter

```
public MethodHandle unreflectGetter(Field f)
                            throws IllegalAccessException
```

Produces a method handle giving read access to a reflected field. The type of the method handle will have a return type of the field's value type. If the field is `static`, the method handle will take no arguments. Otherwise, its single argument will be the instance containing the field. If the `Field` object's `accessible` flag is not set, access checking is performed immediately on behalf of the lookup class.

If the field is static, and if the returned method handle is invoked, the field's class will be initialized, if it has not already been initialized.

Parameters:

`f` - the reflected field

Returns:

a method handle which can load values from the reflected field

Throws:

`IllegalAccessException` - if access checking fails

`NullPointerException` - if the argument is null

unreflectSetter

```
public MethodHandle unreflectSetter(Field f)
                            throws IllegalAccessException
```

Produces a method handle giving write access to a reflected field. The type of the method handle will have a void return type. If the field is `static`, the method handle will take a single argument, of the field's value type, the value to be stored. Otherwise, the two arguments will be the instance containing the field, and the value to be stored. If the `Field` object's `accessible` flag is not set, access checking is performed immediately on behalf of the lookup class.

If the field is `final`, write access will not be allowed and access checking will fail, except under certain narrow circumstances documented for `Field.set`. A method handle is returned only if a corresponding call to the `Field` object's `set` method could return normally. In particular, fields which are both `static` and `final` may never be set.

If the field is `static`, and if the returned method handle is invoked, the field's class will be initialized, if it has not already been initialized.

Parameters:

`f` - the reflected field

Returns:

a method handle which can store values into the reflected field

Throws:

`IllegalAccessException` - if access checking fails, or if the field is `final` and write access is not enabled on the `Field` object

`NullPointerException` - if the argument is null

unreflectVarHandle

```
public VarHandle unreflectVarHandle(Field f)
                             throws IllegalAccessException
```

Produces a VarHandle giving access to a reflected field `f` of type `T` declared in a class of type `R`. The VarHandle's variable type is `T`. If the field is non-static the VarHandle has one coordinate type, `R`. Otherwise, the field is static, and the VarHandle has no coordinate types.

Access checking is performed immediately on behalf of the lookup class, regardless of the value of the field's `accessible` flag.

If the field is static, and if the returned VarHandle is operated on, the field's declaring class will be initialized, if it has not already been initialized.

Certain access modes of the returned VarHandle are unsupported under the following conditions:

- if the field is declared `final`, then the write, atomic update, numeric atomic update, and bitwise atomic update access modes are unsupported.
- if the field type is anything other than `byte`, `short`, `char`, `int`, `long`, `float`, or `double` then numeric atomic update access modes are unsupported.
- if the field type is anything other than `boolean`, `byte`, `short`, `char`, `int` or `long` then bitwise atomic update access modes are unsupported.

If the field is declared `volatile` then the returned VarHandle will override access to the field (effectively ignore the `volatile` declaration) in accordance to its specified access modes.

If the field type is `float` or `double` then numeric and atomic update access modes compare values using their bitwise representation (see `Float.floatToRawIntBits(float)` and `Double.doubleToRawLongBits(double)`, respectively).

API Note:

Bitwise comparison of `float` values or `double` values, as performed by the numeric and atomic update access modes, differ from the primitive == operator and the `Float.equals(java.lang.Object)` and `Double.equals(java.lang.Object)` methods, specifically with respect to comparing NaN values or comparing `-0.0` with `+0.0`. Care should be taken when performing a compare and set or a compare and exchange operation with such values since the operation may unexpectedly fail. There are many possible NaN values that are considered to be `NaN` in Java, although no IEEE 754 floating-point operation provided by Java can distinguish between them. Operation failure can occur if the expected or witness value is a NaN value and it is transformed (perhaps in a platform specific manner) into another NaN value, and thus has a different bitwise representation (see `Float.intBitsToFloat(int)` or `Double.longBitsToDouble(long)` for more details). The values `-0.0` and `+0.0` have different

bitwise representations but are considered equal when using the primitive == operator. Operation failure can occur if, for example, a numeric algorithm computes an expected value to be say `-0.0` and previously computed the witness value to be say `+0.0`.

Parameters:

`f` - the reflected field, with a field of type `T`, and a declaring class of type `R`

Returns:

a VarHandle giving access to non-static fields or a static field

Throws:

`IllegalAccessException` - if access checking fails

`NullPointerException` - if the argument is null

Since:

9

revealDirect

```
public MethodHandleInfo revealDirect(MethodHandle target)
```

Cracks a direct method handle created by this lookup object or a similar one. Security and access checks are performed to ensure that this lookup object is capable of reproducing the target method handle. This means that the cracking may fail if target is a direct method handle but was created by an unrelated lookup object. This can happen if the method handle is caller sensitive and was created by a lookup object for a different class.

Parameters:

`target` - a direct method handle to crack into symbolic reference components

Returns:

a symbolic reference which can be used to reconstruct this method handle from this lookup object

Throws:

`SecurityException` - if a security manager is present and it refuses access

`IllegalArgumentException` - if the target is not a direct method handle or if access checking fails

`NullPointerException` - if the target is `null`

Since:

1.8

See Also:

`MethodHandleInfo`

hasPrivateAccess

```
@Deprecated(since="14")
public boolean hasPrivateAccess()
```

Deprecated.
This method was originally designed to test `PRIVATE` access that implies full privilege access but `MODULE` access has since become independent of `PRIVATE` access. It is recommended to call `hasFullPrivilegeAccess()` instead.

Returns `true` if this lookup has `PRIVATE` and `MODULE` access.

Returns:

`true` if this lookup has `PRIVATE` and `MODULE` access.

Since:

9

hasFullPrivilegeAccess

```
public boolean hasFullPrivilegeAccess()
```

Returns `true` if this lookup has *full privilege access*, i.e. `PRIVATE` and `MODULE` access. A `Lookup` object must have full privilege access in order to access all members that are allowed to the lookup class.

Returns:

`true` if this lookup has full privilege access.

Since:

14

See Also:

private and module access