Module java.base
Package java.lang.invoke

Class MethodHandles

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

```
public class MethodHandles
extends Object
```

This class consists exclusively of static methods that operate on or return method handles. They fall into several categories:

- Lookup methods which help create method handles for methods and fields.
- Combinator methods, which combine or transform pre-existing method handles into new ones.
- Other factory methods to create method handles that emulate other common JVM operations or control flow patterns.

A lookup, combinator, or factory method will fail and throw an `IllegalArgumentException` if the created method handle's type would have too many parameters.

Since:
1.7

Nested Class Summary

Nested Classes

Modifier and Type	Class	Description
`static final class`	`MethodHandles.Lookup`	A *lookup object* is a factory for creating method handles, when the creation requires access checking.

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`static MethodHandle`	`arrayConstructor(Class<?> arrayClass)`	Produces a method handle constructing arrays of a desired type, as if by the `anewarray` bytecode.
`static MethodHandle`	`arrayElementGetter(Class<?> arrayClass)`	Produces a method handle giving read access to elements of an array, as if by the `aaload` bytecode.
`static MethodHandle`	`arrayElementSetter(Class<?> arrayClass)`	Produces a method handle giving write access to elements of an array, as if by the `astore` bytecode.
`static VarHandle`	`arrayElementVarHandle(Class<?> arrayClass)`	Produces a VarHandle giving access to elements of an array of type `arrayClass`.
`static MethodHandle`	`arrayLength(Class<?> arrayClass)`	Produces a method handle returning the length of an array, as if by the `arraylength` bytecode.
`static VarHandle`	`byteArrayViewVarHandle(Class<?> viewArrayClass, ByteOrder byteOrder)`	Produces a VarHandle giving access to elements of a `byte[]` array viewed as if it were a different primitive array type, such as `int[]` or `long[]`.
`static VarHandle`	`byteBufferViewVarHandle(Class<?> viewArrayClass, ByteOrder byteOrder)`	Produces a VarHandle giving access to elements of a `ByteBuffer` viewed as if it were an array of elements of a different primitive component type to that of `byte`, such as `int[]` or `long[]`.
`static MethodHandle`	`catchException(MethodHandle target, Class<? extends Throwable> exType, MethodHandle handler)`	Makes a method handle which adapts a target method handle, by running it inside an exception handler.
`static <T> T`	`classData(MethodHandles.Lookup caller, String name, Class<T> type)`	Returns the *class data* associated with the lookup class of the given `caller` lookup object, or `null`.
`static <T> T`	`classDataAt(MethodHandles.Lookup caller, String name, Class<T> type, int index)`	Returns the element at the specified index in the class data, if the class data associated with the lookup class of the given `caller` lookup object is a `List`.
`static MethodHandle`	`collectArguments(MethodHandle target, int pos, MethodHandle filter)`	Adapts a target method handle by pre-processing a sub-sequence of its arguments with a filter (another method handle).
`static MethodHandle`	`constant(Class<?> type, Object value)`	Produces a method handle of the requested return type which returns the given constant value every

		time it is invoked.
static MethodHandle	countedLoop(MethodHandle iterations, MethodHandle init, MethodHandle body)	Constructs a loop that runs a given number of iterations.
static MethodHandle	countedLoop(MethodHandle start, MethodHandle end, MethodHandle init, MethodHandle body)	Constructs a loop that counts over a range of numbers.
static MethodHandle	doWhileLoop(MethodHandle init, MethodHandle body, MethodHandle pred)	Constructs a `do-while` loop from an initializer, a body, and a predicate.
static MethodHandle	dropArguments(MethodHandle target, int pos, Class<?>... valueTypes)	Produces a method handle which will discard some dummy arguments before calling some other specified *target* method handle.
static MethodHandle	dropArguments(MethodHandle target, int pos, List<Class<?>> valueTypes)	Produces a method handle which will discard some dummy arguments before calling some other specified *target* method handle.
static MethodHandle	dropArgumentsToMatch(MethodHandle target, int skip, List<Class<?>> newTypes, int pos)	Adapts a target method handle to match the given parameter type list.
static MethodHandle	dropReturn(MethodHandle target)	Drop the return value of the target handle (if any).
static MethodHandle	empty(MethodType type)	Produces a method handle of the requested type which ignores any arguments, does nothing, and returns a suitable default depending on the return type.
static MethodHandle	exactInvoker(MethodType type)	Produces a special *invoker method handle* which can be used to invoke any method handle of the given type, as if by `invokeExact`.
static MethodHandle	explicitCastArguments(MethodHandle target, MethodType newType)	Produces a method handle which adapts the type of the given method handle to a new type by pairwise argument and return type conversion.
static MethodHandle	filterArguments(MethodHandle target, int pos, MethodHandle... filters)	Adapts a target method handle by pre-processing one or more of its arguments, each with its own unary filter function, and then calling the target with each pre-processed argument replaced by the result of its corresponding filter function.
static MethodHandle	filterReturnValue(MethodHandle target, MethodHandle filter)	Adapts a target method handle by post-processing its return value (if any) with a filter (another method handle).
static MethodHandle	foldArguments(MethodHandle target, int pos, MethodHandle combiner)	Adapts a target method handle by pre-processing some of its arguments, starting at a given position, and then calling the target with the result of the pre-processing, inserted into the original sequence of arguments just before the folded arguments.
static MethodHandle	foldArguments(MethodHandle target, MethodHandle combiner)	Adapts a target method handle by pre-processing some of its arguments, and then calling the target with the result of the pre-processing, inserted into the original sequence of arguments.
static MethodHandle	guardWithTest(MethodHandle test, MethodHandle target, MethodHandle fallback)	Makes a method handle which adapts a target method handle, by guarding it with a test, a boolean-valued method handle.
static MethodHandle	identity(Class<?> type)	Produces a method handle which returns its sole argument when invoked.
static MethodHandle	insertArguments(MethodHandle target, int pos, Object... values)	Provides a target method handle with one or more *bound arguments* in advance of the method handle's invocation.
static MethodHandle	invoker(MethodType type)	Produces a special *invoker method handle* which can be used to invoke any method handle compatible with the given type, as if by `invoke`.
static MethodHandle	iteratedLoop(MethodHandle iterator, MethodHandle init, MethodHandle body)	Constructs a loop that ranges over the values produced by an `Iterator<T>`.
static MethodHandles.Lookup	lookup()	Returns a `lookup object` with full capabilities to emulate all supported bytecode behaviors of the caller.
static MethodHandle	loop(MethodHandle[]... clauses)	Constructs a method handle representing a loop with several loop variables that are updated and checked upon each iteration.

`static MethodHandle`	`permuteArguments(MethodHandle target, MethodType newType, int... reorder)`	Produces a method handle which adapts the calling sequence of the given method handle to a new type, by reordering the arguments.
`static MethodHandles.Lookup`	`privateLookupIn(Class<?> targetClass, MethodHandles.Lookup caller)`	Returns a `lookup` object on a target class to emulate all supported bytecode behaviors, including private access.
`static MethodHandles.Lookup`	`publicLookup()`	Returns a `lookup object` which is trusted minimally.
`static <T extends Member> T`	`reflectAs(Class<T> expected, MethodHandle target)`	Performs an unchecked "crack" of a direct method handle.
`static MethodHandle`	`spreadInvoker(MethodType type, int leadingArgCount)`	Produces a method handle which will invoke any method handle of the given `type`, with a given number of trailing arguments replaced by a single trailing `Object[]` array.
`static MethodHandle`	`tableSwitch(MethodHandle fallback, MethodHandle... targets)`	Creates a table switch method handle, which can be used to switch over a set of target method handles, based on a given target index, called selector.
`static MethodHandle`	`throwException(Class<?> returnType, Class<? extends Throwable> exType)`	Produces a method handle which will throw exceptions of the given exType.
`static MethodHandle`	`tryFinally(MethodHandle target, MethodHandle cleanup)`	Makes a method handle that adapts a `target` method handle by wrapping it in a `try-finally` block.
`static MethodHandle`	`varHandleExactInvoker(VarHandle.AccessMode accessMode, MethodType type)`	Produces a special *invoker method handle* which can be used to invoke a signature-polymorphic access mode method on any VarHandle whose associated access mode type is compatible with the given type.
`static MethodHandle`	`varHandleInvoker(VarHandle.AccessMode accessMode, MethodType type)`	Produces a special *invoker method handle* which can be used to invoke a signature-polymorphic access mode method on any VarHandle whose associated access mode type is compatible with the given type.
`static MethodHandle`	`whileLoop(MethodHandle init, MethodHandle pred, MethodHandle body)`	Constructs a `while` loop from an initializer, a body, and a predicate.
`static MethodHandle`	`zero(Class<?> type)`	Produces a constant method handle of the requested return type which returns the default value for that type every time it is invoked.

Methods declared in class java.lang.Object

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

Method Details

lookup

```
public static MethodHandles.Lookup lookup()
```

Returns a `lookup object` with full capabilities to emulate all supported bytecode behaviors of the caller. These capabilities include full privilege access to the caller. Factory methods on the lookup object can create direct method handles for any member that the caller has access to via bytecodes, including protected and private fields and methods. This lookup object is created by the original lookup class and has the `ORIGINAL` bit set. This lookup object is a *capability* which may be delegated to trusted agents. Do not store it in place where untrusted code can access it.

This method is caller sensitive, which means that it may return different values to different callers.

Returns:

a lookup object for the caller of this method, with original and full privilege access.

publicLookup

```
public static MethodHandles.Lookup publicLookup()
```

Returns a `lookup object` which is trusted minimally. The lookup has the `UNCONDITIONAL` mode. It can only be used to create method handles to public members of public classes in packages that are exported unconditionally.

As a matter of pure convention, the lookup class of this lookup object will be `Object`.

API Note:

The use of Object is conventional, and because the lookup modes are limited, there is no special access provided to the internals of Object, its package or its module. This public lookup object or other lookup object with `UNCONDITIONAL` mode assumes readability. Consequently, the lookup class is not used to determine the lookup context.

Discussion: The lookup class can be changed to any other class `C` using an expression of the form `publicLookup().in(C.class)`. A public lookup object is always subject to security manager checks. Also, it cannot access caller sensitive methods.

Returns:

a lookup object which is trusted minimally

privateLookupIn

```
public static MethodHandles.Lookup privateLookupIn(Class<?> targetClass,
                                                    MethodHandles.Lookup caller)
                                             throws IllegalAccessException
```

Returns a `lookup` object on a target class to emulate all supported bytecode behaviors, including private access. The returned lookup object can provide access to classes in modules and packages, and members of those classes, outside the normal rules of Java access control, instead conforming to the more permissive rules for modular *deep reflection*.

A caller, specified as a `Lookup` object, in module `M1` is allowed to do deep reflection on module `M2` and package of the target class if and only if all of the following conditions are `true`:

- If there is a security manager, its `checkPermission` method is called to check `ReflectPermission("suppressAccessChecks")` and that must return normally.
- The caller lookup object must have full privilege access. Specifically:
  - The caller lookup object must have the `MODULE` lookup mode. (This is because otherwise there would be no way to ensure the original lookup creator was a member of any particular module, and so any subsequent checks for readability and qualified exports would become ineffective.)
  - The caller lookup object must have `PRIVATE` access. (This is because an application intending to share intra-module access using `MODULE` alone will inadvertently also share deep reflection to its own module.)
- The target class must be a proper class, not a primitive or array class. (Thus, `M2` is well-defined.)
- If the caller module `M1` differs from the target module `M2` then both of the following must be true:
  - `M1` `reads` `M2`.
  - `M2` `opens` the package containing the target class to at least `M1`.

If any of the above checks is violated, this method fails with an exception.

Otherwise, if `M1` and `M2` are the same module, this method returns a `Lookup` on `targetClass` with full privilege access with `null` previous lookup class.

Otherwise, `M1` and `M2` are two different modules. This method returns a `Lookup` on `targetClass` that records the lookup class of the caller as the new previous lookup class with `PRIVATE` access but no `MODULE` access.

The resulting `Lookup` object has no `ORIGINAL` access.

Parameters:

`targetClass` - the target class

`caller` - the caller lookup object

Returns:

a lookup object for the target class, with private access

Throws:

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

`NullPointerException` - if `targetClass` or `caller` is `null`

`SecurityException` - if denied by the security manager

`IllegalAccessException` - if any of the other access checks specified above fails

Since:

9

See Also:

`MethodHandles.Lookup.dropLookupMode(int)`,
Cross-module lookups

classData

```
public static <T> T classData(MethodHandles.Lookup caller,
                              String name,
                              Class<T> type)
                       throws IllegalAccessException
```

Returns the *class data* associated with the lookup class of the given `caller` lookup object, or `null`.

A hidden class with class data can be created by calling `Lookup::defineHiddenClassWithClassData`. This method will cause the static class initializer of the lookup class of the given `caller` lookup object be executed if it has not been initialized.

A hidden class created by `Lookup::defineHiddenClass` and non-hidden classes have no class data. `null` is returned if this method is called on the lookup object on these classes.

The lookup modes for this lookup must have original access in order to retrieve the class data.

API Note:

This method can be called as a bootstrap method for a dynamically computed constant. A framework can create a hidden class with class data, for example that can be `Class` or `MethodHandle` object. The class data is accessible only to the lookup object created by the original caller but inaccessible to other members in the same nest. If a framework passes security sensitive objects to a hidden class via class data, it is recommended to load the value of class data as a dynamically computed constant instead of storing the class data in private static field(s) which are accessible to other nestmates.

Type Parameters:

`T` - the type to cast the class data object to

Parameters:

`caller` - the lookup context describing the class performing the operation (normally stacked by the JVM)

`name` - must be `ConstantDescs.DEFAULT_NAME` (`"_"`)

`type` - the type of the class data

Returns:

the value of the class data if present in the lookup class; otherwise `null`

Throws:

`IllegalArgumentException` - if name is not `"_"`

`IllegalAccessException` - if the lookup context does not have original access

`ClassCastException` - if the class data cannot be converted to the given `type`

`NullPointerException` - if `caller` or `type` argument is `null`

See *Java Virtual Machine Specification*:

5.5 Initialization

Since:

16

See Also:

`MethodHandles.Lookup.defineHiddenClassWithClassData(byte[], Object, boolean, Lookup.ClassOption...)`, `classDataAt(Lookup, String, Class, int)`

classDataAt

```
public static <T> T classDataAt(MethodHandles.Lookup caller,
                                String name,
                                Class<T> type,
                                int index)
                         throws IllegalAccessException
```

Returns the element at the specified index in the class data, if the class data associated with the lookup class of the given `caller` lookup object is a `List`. If the class data is not present in this lookup class, this method returns `null`.

A hidden class with class data can be created by calling `Lookup::defineHiddenClassWithClassData`. This method will cause the static class initializer of the lookup class of the given `caller` lookup object be executed if it has not been initialized.

A hidden class created by `Lookup::defineHiddenClass` and non-hidden classes have no class data. `null` is returned if this method is called on the lookup object on these classes.

The lookup modes for this lookup must have original access in order to retrieve the class data.

API Note:

This method can be called as a bootstrap method for a dynamically computed constant. A framework can create a hidden class with class data, for example that can be `List.of(o1, o2, o3....)` containing more than one object and use this method to load one element at a specific index. The class data is accessible only to the lookup object created by the original caller but inaccessible to other members in the same nest. If a framework passes security sensitive objects to a hidden class via class data, it is recommended to load the value of class data as a dynamically computed constant instead of storing the class data in private static field(s) which are accessible to other nestmates.

Type Parameters:

`T` - the type to cast the result object to

Parameters:

`caller` - the lookup context describing the class performing the operation (normally stacked by the JVM)

`name` - must be `ConstantDescs.DEFAULT_NAME` (`"_"`)

`type` - the type of the element at the given index in the class data

`index` - index of the element in the class data

Returns:

the element at the given index in the class data if the class data is present; otherwise `null`

Throws:

`IllegalArgumentException` - if name is not `"_"`

`IllegalAccessException` - if the lookup context does not have original access

`ClassCastException` - if the class data cannot be converted to `List` or the element at the specified index cannot be converted to the given type

`IndexOutOfBoundsException` - if the index is out of range

`NullPointerException` - if `caller` or `type` argument is `null`; or if unboxing operation fails because the element at the given index is `null`

Since:

16

See Also:

`classData(Lookup, String, Class)`,
`MethodHandles.Lookup.defineHiddenClassWithClassData(byte[], Object, boolean, Lookup.ClassOption...)`

reflectAs

```
public static <T extends Member> T reflectAs(Class<T> expected,
                                             MethodHandle target)
```

Performs an unchecked "crack" of a direct method handle. The result is as if the user had obtained a lookup object capable enough to crack the target method handle, called `Lookup.revealDirect` on the target to obtain its symbolic reference, and then called `MethodHandleInfo.reflectAs` to resolve the symbolic reference to a member.

If there is a security manager, its `checkPermission` method is called with a `ReflectPermission("suppressAccessChecks")` permission.

Type Parameters:

`T` - the desired type of the result, either `Member` or a subtype

Parameters:

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

`expected` - a class object representing the desired result type `T`

Returns:

a reference to the method, constructor, or field object

Throws:

`SecurityException` - if the caller is not privileged to call `setAccessible`

`NullPointerException` - if either argument is `null`

`IllegalArgumentException` - if the target is not a direct method handle

`ClassCastException` - if the member is not of the expected type

Since:

1.8

arrayConstructor

```
public static MethodHandle arrayConstructor(Class<?> arrayClass)
                                     throws IllegalArgumentException
```

Produces a method handle constructing arrays of a desired type, as if by the `anewarray` bytecode. The return type of the method handle will be the array type. The type of its sole argument will be `int`, which specifies the size of the array.

If the returned method handle is invoked with a negative array size, a `NegativeArraySizeException` will be thrown.

Parameters:

`arrayClass` - an array type

Returns:

a method handle which can create arrays of the given type

Throws:

`NullPointerException` - if the argument is `null`

`IllegalArgumentException` - if `arrayClass` is not an array type

See *Java Virtual Machine Specification*:

6.5 `anewarray` Instruction

Since:

9

See Also:

`Array.newInstance(Class, int)`

arrayLength

```
public static MethodHandle arrayLength(Class<?> arrayClass)
                                throws IllegalArgumentException
```

Produces a method handle returning the length of an array, as if by the `arraylength` bytecode. The type of the method handle will have `int` as return type, and its sole argument will be the array type.

If the returned method handle is invoked with a `null` array reference, a `NullPointerException` will be thrown.

Parameters:

`arrayClass` - an array type

Returns:

a method handle which can retrieve the length of an array of the given array type

Throws:

`NullPointerException` - if the argument is `null`

`IllegalArgumentException` - if arrayClass is not an array type

See *Java Virtual Machine Specification*:

6.5 `arraylength` Instruction

Since:

9

arrayElementGetter

```
public static MethodHandle arrayElementGetter(Class<?> arrayClass)
                                       throws IllegalArgumentException
```

Produces a method handle giving read access to elements of an array, as if by the `aaload` bytecode. The type of the method handle will have a return type of the array's element type. Its first argument will be the array type, and the second will be `int`.

When the returned method handle is invoked, the array reference and array index are checked. A `NullPointerException` will be thrown if the array reference is `null` and an `ArrayIndexOutOfBoundsException` will be thrown if the index is negative or if it is greater than or equal to the length of the array.

Parameters:

`arrayClass` - an array type

Returns:

a method handle which can load values from the given array type

Throws:

`NullPointerException` - if the argument is null

`IllegalArgumentException` - if arrayClass is not an array type

See *Java Virtual Machine Specification*:

6.5 `aaload` Instruction

arrayElementSetter

```
public static MethodHandle arrayElementSetter(Class<?> arrayClass)
                                       throws IllegalArgumentException
```

Produces a method handle giving write access to elements of an array, as if by the `astore` bytecode. The type of the method handle will have a void return type. Its last argument will be the array's element type. The first and second arguments will be the array type and int.

When the returned method handle is invoked, the array reference and array index are checked. A `NullPointerException` will be thrown if the array reference is `null` and an `ArrayIndexOutOfBoundsException` will be thrown if the index is negative or if it is greater than or equal to the length of the array.

Parameters:

`arrayClass` - the class of an array

Returns:

a method handle which can store values into the array type

Throws:

`NullPointerException` - if the argument is null

`IllegalArgumentException` - if arrayClass is not an array type

See *Java Virtual Machine Specification*:

6.5 `aastore` Instruction

arrayElementVarHandle

```
public static VarHandle arrayElementVarHandle(Class<?> arrayClass)
                                       throws IllegalArgumentException
```

Produces a VarHandle giving access to elements of an array of type `arrayClass`. The VarHandle's variable type is the component type of `arrayClass` and the list of coordinate types is `(arrayClass, int)`, where the `int` coordinate type corresponds to an argument that is an index into an array.

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

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

If the component 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).

When the returned `VarHandle` is invoked, the array reference and array index are checked. A `NullPointerException` will be thrown if the array reference is `null` and an `ArrayIndexOutOfBoundsException` will be thrown if the index is negative or if it is greater than or equal to the length of the array.

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:

`arrayClass` - the class of an array, of type `T[]`

Returns:

a VarHandle giving access to elements of an array

Throws:

`NullPointerException` - if the arrayClass is null

`IllegalArgumentException` - if arrayClass is not an array type

Since:

9

byteArrayViewVarHandle

```
public static VarHandle byteArrayViewVarHandle(Class<?> viewArrayClass,
                                               ByteOrder byteOrder)
                                        throws IllegalArgumentException
```

Produces a VarHandle giving access to elements of a `byte[]` array viewed as if it were a different primitive array type, such as `int[]` or `long[]`. The VarHandle's variable type is the component type of `viewArrayClass` and the list of coordinate types is `(byte[], int)`, where the `int` coordinate type corresponds to an argument that is an index into a `byte[]` array. The returned VarHandle accesses bytes at an index in a `byte[]` array, composing bytes to or from a value of the component type of `viewArrayClass` according to the given endianness.

The supported component types (variables types) are `short`, `char`, `int`, `long`, `float` and `double`.

Access of bytes at a given index will result in an `ArrayIndexOutOfBoundsException` if the index is less than `0` or greater than the `byte[]` array length minus the size (in bytes) of `T`.

Access of bytes at an index may be aligned or misaligned for `T`, with respect to the underlying memory address, `A` say, associated with the array and index. If access is misaligned then access for anything other than the `get` and `set` access modes will result in an `IllegalStateException`. In such cases atomic access is only guaranteed with respect to the largest power of two that divides the GCD of `A` and the size (in bytes) of `T`. If access is aligned then following access modes are supported and are guaranteed to support atomic access:

- read write access modes for all `T`, with the exception of access modes `get` and `set` for `long` and `double` on 32-bit platforms.
- atomic update access modes for `int`, `long`, `float` or `double`. (Future major platform releases of the JDK may support additional types for certain currently unsupported access modes.)
- numeric atomic update access modes for `int` and `long`. (Future major platform releases of the JDK may support additional numeric types for certain currently unsupported access modes.)
- bitwise atomic update access modes for `int` and `long`. (Future major platform releases of the JDK may support additional numeric types for certain currently unsupported access modes.)

Misaligned access, and therefore atomicity guarantees, may be determined for `byte[]` arrays without operating on a specific array. Given an `index`, `T` and it's corresponding boxed type, `T_BOX`, misalignment may be determined as follows:

```
int sizeOfT = T_BOX.BYTES;  // size in bytes of T
int misalignedAtZeroIndex = ByteBuffer.wrap(new byte[0]).
    alignmentOffset(0, sizeOfT);
int misalignedAtIndex = (misalignedAtZeroIndex + index) % sizeOfT;
boolean isMisaligned = misalignedAtIndex != 0;
```

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

Parameters:

`viewArrayClass` - the view array class, with a component type of type `T`

`byteOrder` - the endianness of the view array elements, as stored in the underlying `byte` array

Returns:

a VarHandle giving access to elements of a `byte[]` array viewed as if elements corresponding to the components type of the view array class

Throws:

`NullPointerException` - if viewArrayClass or byteOrder is null

`IllegalArgumentException` - if viewArrayClass is not an array type

`UnsupportedOperationException` - if the component type of viewArrayClass is not supported as a variable type

Since:

9

byteBufferViewVarHandle

```
public static VarHandle byteBufferViewVarHandle(Class<?> viewArrayClass,
                                                ByteOrder byteOrder)
                                         throws IllegalArgumentException
```

Produces a VarHandle giving access to elements of a `ByteBuffer` viewed as if it were an array of elements of a different primitive component type to that of `byte`, such as `int[]` or `long[]`. The VarHandle's variable type is the component type of `viewArrayClass` and the list of coordinate types is `(ByteBuffer, int)`, where the `int` coordinate type corresponds to an argument that is an index into a `byte[]` array. The returned VarHandle accesses bytes at an index in a `ByteBuffer`, composing bytes to or from a value of the component type of `viewArrayClass` according to the given endianness.

The supported component types (variables types) are `short`, `char`, `int`, `long`, `float` and `double`.

Access will result in a `ReadOnlyBufferException` for anything other than the read access modes if the `ByteBuffer` is read-only.

Access of bytes at a given index will result in an `IndexOutOfBoundsException` if the index is less than `0` or greater than the `ByteBuffer` limit minus the size (in bytes) of `T`.

Access of bytes at an index may be aligned or misaligned for `T`, with respect to the underlying memory address, `A` say, associated with the `ByteBuffer` and index. If access is misaligned then access for anything other than the `get` and `set` access modes will result in an `IllegalStateException`. In such cases atomic access is only guaranteed with respect to the largest power of two that divides the GCD of `A` and the size (in bytes) of `T`. If access is aligned then following access modes are supported and are guaranteed to support atomic access:

- read write access modes for all `T`, with the exception of access modes `get` and `set` for `long` and `double` on 32-bit platforms.
- atomic update access modes for `int`, `long`, `float` or `double`. (Future major platform releases of the JDK may support additional types for certain currently unsupported access modes.)
- numeric atomic update access modes for `int` and `long`. (Future major platform releases of the JDK may support additional numeric types for certain currently unsupported access modes.)
- bitwise atomic update access modes for `int` and `long`. (Future major platform releases of the JDK may support additional numeric types for certain currently unsupported access modes.)

Misaligned access, and therefore atomicity guarantees, may be determined for a `ByteBuffer`, `bb` (direct or otherwise), an `index`, `T` and it's corresponding boxed type, `T_BOX`, as follows:

```
int sizeOfT = T_BOX.BYTES;  // size in bytes of T
ByteBuffer bb = ...
int misalignedAtIndex = bb.alignmentOffset(index, sizeOfT);
boolean isMisaligned = misalignedAtIndex != 0;
```

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

Parameters:

`viewArrayClass` - the view array class, with a component type of type `T`

`byteOrder` - the endianness of the view array elements, as stored in the underlying `ByteBuffer` (Note this overrides the endianness of a `ByteBuffer`)

Returns:

a VarHandle giving access to elements of a `ByteBuffer` viewed as if elements corresponding to the components type of the view array class

Throws:

`NullPointerException` - if viewArrayClass or byteOrder is null

`IllegalArgumentException` - if viewArrayClass is not an array type

`UnsupportedOperationException` - if the component type of viewArrayClass is not supported as a variable type

Since:

9

spreadInvoker

```
public static MethodHandle spreadInvoker(MethodType type,
                                         int leadingArgCount)
```

Produces a method handle which will invoke any method handle of the given `type`, with a given number of trailing arguments replaced by a single trailing `Object[]` array. The resulting invoker will be a method handle with the following arguments:

- a single `MethodHandle` target
- zero or more leading values (counted by `leadingArgCount`)
- an `Object[]` array containing trailing arguments

The invoker will invoke its target like a call to `invoke` with the indicated `type`. That is, if the target is exactly of the given `type`, it will behave like `invokeExact`; otherwise it behave as if `asType` is used to convert the target to the required `type`.

The type of the returned invoker will not be the given `type`, but rather will have all parameters except the first `leadingArgCount` replaced by a single array of type `Object[]`, which will be the final parameter.

Before invoking its target, the invoker will spread the final array, apply reference casts as necessary, and unbox and widen primitive arguments. If, when the invoker is called, the supplied array argument does not have the correct number of elements, the invoker will throw an `IllegalArgumentException` instead of invoking the target.

This method is equivalent to the following code (though it may be more efficient):

```
MethodHandle invoker = MethodHandles.invoker(type);
int spreadArgCount = type.parameterCount() - leadingArgCount;
invoker = invoker.asSpreader(Object[].class, spreadArgCount);
return invoker;
```

This method throws no reflective or security exceptions.

Parameters:

`type` - the desired target type

`leadingArgCount` - number of fixed arguments, to be passed unchanged to the target

Returns:

a method handle suitable for invoking any method handle of the given type

Throws:

`NullPointerException` - if `type` is null

`IllegalArgumentException` - if `leadingArgCount` is not in the range from 0 to `type.parameterCount()` inclusive, or if the resulting method handle's type would have too many parameters

exactInvoker

```
public static MethodHandle exactInvoker(MethodType type)
```

Produces a special *invoker method handle* which can be used to invoke any method handle of the given type, as if by `invokeExact`. The resulting invoker will have a type which is exactly equal to the desired type, except that it will accept an additional leading argument of type `MethodHandle`.

This method is equivalent to the following code (though it may be more efficient): `publicLookup().findVirtual(MethodHandle.class, "invokeExact", type)`

Discussion: Invoker method handles can be useful when working with variable method handles of unknown types. For example, to emulate an `invokeExact` call to a variable method handle `M`, extract its type `T`, look up the invoker method `X` for `T`, and call the invoker method, as `X.invoke(T, A...)`. (It would not work to call `X.invokeExact`, since the type `T` is unknown.) If spreading, collecting, or other argument transformations are required, they can be applied once to the invoker `X` and reused on many `M` method handle values, as long as they are compatible with the type of `X`.

(Note: The invoker method is not available via the Core Reflection API. An attempt to call java.lang.reflect.Method.invoke on the declared `invokeExact` or `invoke` method will raise an `UnsupportedOperationException`.)

This method throws no reflective or security exceptions.

Parameters:

`type` - the desired target type

Returns:

a method handle suitable for invoking any method handle of the given type

Throws:

`IllegalArgumentException` - if the resulting method handle's type would have too many parameters

invoker

```
public static MethodHandle invoker(MethodType type)
```

Produces a special *invoker method handle* which can be used to invoke any method handle compatible with the given type, as if by `invoke`. The resulting invoker will have a type which is exactly equal to the desired type, except that it will accept an additional leading argument of type `MethodHandle`.

Before invoking its target, if the target differs from the expected type, the invoker will apply reference casts as necessary and box, unbox, or widen primitive values, as if by `asType`. Similarly, the return value will be converted as necessary. If the target is a variable arity method handle, the required arity conversion will be made, again as if by `asType`.

This method is equivalent to the following code (though it may be more efficient): `publicLookup().findVirtual(MethodHandle.class, "invoke", type)`

Discussion: A general method type is one which mentions only `Object` arguments and return values. An invoker for such a type is capable of calling any method handle of the same arity as the general type.

(Note: The invoker method is not available via the Core Reflection API. An attempt to call java.lang.reflect.Method.invoke on the declared `invokeExact` or `invoke` method will raise an `UnsupportedOperationException`.)

This method throws no reflective or security exceptions.

Parameters:

`type` - the desired target type

Returns:

a method handle suitable for invoking any method handle convertible to the given type

Throws:

`IllegalArgumentException` - if the resulting method handle's type would have too many parameters

varHandleExactInvoker

```
public static MethodHandle varHandleExactInvoker(VarHandle.AccessMode accessMode,
                                                 MethodType type)
```

Produces a special *invoker method handle* which can be used to invoke a signature-polymorphic access mode method on any VarHandle whose associated access mode type is compatible with the given type. The resulting invoker will have a type which is exactly equal to the desired given type, except that it will accept an additional leading argument of type `VarHandle`.

Parameters:

`accessMode` - the VarHandle access mode

`type` - the desired target type

Returns:

a method handle suitable for invoking an access mode method of any VarHandle whose access mode type is of the given type.

Since:

9

varHandleInvoker

```
public static MethodHandle varHandleInvoker(VarHandle.AccessMode accessMode,
                                            MethodType type)
```

Produces a special *invoker method handle* which can be used to invoke a signature-polymorphic access mode method on any VarHandle whose associated access mode type is compatible with the given type. The resulting invoker will have a type which is exactly equal to the desired given type, except that it will accept an additional leading argument of type `VarHandle`.

Before invoking its target, if the access mode type differs from the desired given type, the invoker will apply reference casts as necessary and box, unbox, or widen primitive values, as if by `asType`. Similarly, the return value will be converted as necessary.

This method is equivalent to the following code (though it may be more efficient): `publicLookup().findVirtual(VarHandle.class, accessMode.name(), type)`

Parameters:

`accessMode` - the VarHandle access mode

`type` - the desired target type

Returns:

a method handle suitable for invoking an access mode method of any VarHandle whose access mode type is convertible to the given type.

Since:

9

explicitCastArguments

```
public static MethodHandle explicitCastArguments(MethodHandle target,
                                                 MethodType newType)
```

Produces a method handle which adapts the type of the given method handle to a new type by pairwise argument and return type conversion. The original type and new type must have the same number of arguments. The resulting method handle is guaranteed to report a type which is equal to the desired new type.

If the original type and new type are equal, returns target.

The same conversions are allowed as for `MethodHandle.asType`, and some additional conversions are also applied if those conversions fail. Given types *T0*, *T1*, one of the following conversions is applied if possible, before or instead of any conversions done by `asType`:

- If *T0* and *T1* are references, and *T1* is an interface type, then the value of type *T0* is passed as a *T1* without a cast. (This treatment of interfaces follows the usage of the bytecode verifier.)
- If *T0* is boolean and *T1* is another primitive, the boolean is converted to a byte value, 1 for true, 0 for false. (This treatment follows the usage of the bytecode verifier.)
- If *T1* is boolean and *T0* is another primitive, *T0* is converted to byte via Java casting conversion (JLS 5.5), and the low order bit of the result is tested, as if by `(x & 1) != 0`.
- If *T0* and *T1* are primitives other than boolean, then a Java casting conversion (JLS 5.5) is applied. (Specifically, *T0* will convert to *T1* by widening and/or narrowing.)
- If *T0* is a reference and *T1* a primitive, an unboxing conversion will be applied at runtime, possibly followed by a Java casting conversion (JLS 5.5) on the primitive value, possibly followed by a conversion from byte to boolean by testing the low-order bit.
- If *T0* is a reference and *T1* a primitive, and if the reference is null at runtime, a zero value is introduced.

Parameters:

`target` - the method handle to invoke after arguments are retyped

`newType` - the expected type of the new method handle

Returns:

a method handle which delegates to the target after performing any necessary argument conversions, and arranges for any necessary return value conversions

Throws:

`NullPointerException` - if either argument is null

`WrongMethodTypeException` - if the conversion cannot be made

See Also:

`MethodHandle.asType(java.lang.invoke.MethodType)`

permuteArguments

```
public static MethodHandle permuteArguments(MethodHandle target,
                                            MethodType newType,
                                            int... reorder)
```

Produces a method handle which adapts the calling sequence of the given method handle to a new type, by reordering the arguments. The resulting method handle is guaranteed to report a type which is equal to the desired new type.

The given array controls the reordering. Call `#I` the number of incoming parameters (the value `newType.parameterCount()`, and call `#O` the number of outgoing parameters (the value `target.type().parameterCount()`). Then the length of the reordering array must be `#O`, and each element must be a non-negative number less than `#I`. For every `N` less than `#O`, the `N`-th outgoing argument will be taken from the `I`-th incoming argument, where `I` is `reorder[N]`.

No argument or return value conversions are applied. The type of each incoming argument, as determined by `newType`, must be identical to the type of the corresponding outgoing parameter or parameters in the target method handle. The return type of `newType` must be identical to the return type of the original target.

The reordering array need not specify an actual permutation. An incoming argument will be duplicated if its index appears more than once in the array, and an incoming argument will be dropped if its index does not appear in the array. As in the case of `dropArguments`, incoming arguments which are not mentioned in the reordering array may be of any type, as determined only by `newType`.

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodType intfn1 = methodType(int.class, int.class);
MethodType intfn2 = methodType(int.class, int.class, int.class);
MethodHandle sub = ... (int x, int y) -> (x-y) ...;
assert(sub.type().equals(intfn2));
MethodHandle sub1 = permuteArguments(sub, intfn2, 0, 1);
MethodHandle rsub = permuteArguments(sub, intfn2, 1, 0);
assert((int)rsub.invokeExact(1, 100) == 99);
MethodHandle add = ... (int x, int y) -> (x+y) ...;
assert(add.type().equals(intfn2));
MethodHandle twice = permuteArguments(add, intfn1, 0, 0);
assert(twice.type().equals(intfn1));
assert((int)twice.invokeExact(21) == 42);
```

Note: The resulting adapter is never a variable-arity method handle, even if the original target method handle was.

Parameters:

`target` - the method handle to invoke after arguments are reordered

`newType` - the expected type of the new method handle

`reorder` - an index array which controls the reordering

Returns:

a method handle which delegates to the target after it drops unused arguments and moves and/or duplicates the other arguments

Throws:

`NullPointerException` - if any argument is null

IllegalArgumentException - if the index array length is not equal to the arity of the target, or if any index array element not a valid index for a parameter of newType, or if two corresponding parameter types in target.type() and newType are not identical,

constant

```
public static MethodHandle constant(Class<?> type,
                                    Object value)
```

Produces a method handle of the requested return type which returns the given constant value every time it is invoked.

Before the method handle is returned, the passed-in value is converted to the requested type. If the requested type is primitive, widening primitive conversions are attempted, else reference conversions are attempted.

The returned method handle is equivalent to `identity(type).bindTo(value)`.

Parameters:

`type` - the return type of the desired method handle

`value` - the value to return

Returns:

a method handle of the given return type and no arguments, which always returns the given value

Throws:

`NullPointerException` - if the `type` argument is null

`ClassCastException` - if the value cannot be converted to the required return type

`IllegalArgumentException` - if the given type is `void.class`

identity

```
public static MethodHandle identity(Class<?> type)
```

Produces a method handle which returns its sole argument when invoked.

Parameters:

`type` - the type of the sole parameter and return value of the desired method handle

Returns:

a unary method handle which accepts and returns the given type

Throws:

`NullPointerException` - if the argument is null

`IllegalArgumentException` - if the given type is `void.class`

zero

```
public static MethodHandle zero(Class<?> type)
```

Produces a constant method handle of the requested return type which returns the default value for that type every time it is invoked. The resulting constant method handle will have no side effects.

The returned method handle is equivalent to `empty(methodType(type))`. It is also equivalent to `explicitCastArguments(constant(Object.class, null), methodType(type))`, since `explicitCastArguments` converts `null` to default values.

Parameters:

`type` - the expected return type of the desired method handle

Returns:

a constant method handle that takes no arguments and returns the default value of the given type (or void, if the type is void)

Throws:

`NullPointerException` - if the argument is null

Since:

9

See Also:

`constant(java.lang.Class<?>, java.lang.Object)`,
`empty(java.lang.invoke.MethodType)`,
`explicitCastArguments(java.lang.invoke.MethodHandle, java.lang.invoke.MethodType)`

empty

```
public static MethodHandle empty(MethodType type)
```

Produces a method handle of the requested type which ignores any arguments, does nothing, and returns a suitable default depending on the return type. That is, it returns a zero primitive value, a `null`, or `void`.

The returned method handle is equivalent to `dropArguments(zero(type.returnType()), 0, type.parameterList())`.

API Note:

Given a predicate and target, a useful "if-then" construct can be produced as `guardWithTest(pred, target, empty(target.type())`.

Parameters:

`type` - the type of the desired method handle

Returns:

a constant method handle of the given type, which returns a default value of the given return type

Throws:

`NullPointerException` - if the argument is null

Since:

9

See Also:

`zero(java.lang.Class<?>)`,
`constant(java.lang.Class<?>, java.lang.Object)`

insertArguments

```
public static MethodHandle insertArguments(MethodHandle target,
                                           int pos,
                                           Object... values)
```

Provides a target method handle with one or more *bound arguments* in advance of the method handle's invocation. The formal parameters to the target corresponding to the bound arguments are called *bound parameters*. Returns a new method handle which saves away the bound arguments. When it is invoked, it receives arguments for any non-bound parameters, binds the saved arguments to their corresponding parameters, and calls the original target.

The type of the new method handle will drop the types for the bound parameters from the original target type, since the new method handle will no longer require those arguments to be supplied by its callers.

Each given argument object must match the corresponding bound parameter type. If a bound parameter type is a primitive, the argument object must be a wrapper, and will be unboxed to produce the primitive value.

The `pos` argument selects which parameters are to be bound. It may range between zero and *N-L* (inclusively), where *N* is the arity of the target method handle and *L* is the length of the values array.

Note: The resulting adapter is never a variable-arity method handle, even if the original target method handle was.

Parameters:

`target` - the method handle to invoke after the argument is inserted

`pos` - where to insert the argument (zero for the first)

`values` - the series of arguments to insert

Returns:

a method handle which inserts an additional argument, before calling the original method handle

Throws:

`NullPointerException` - if the target or the `values` array is null

`IllegalArgumentException` - if (@code pos) is less than `0` or greater than `N - L` where `N` is the arity of the target method handle and `L` is the length of the values array.

`ClassCastException` - if an argument does not match the corresponding bound parameter type.

See Also:

`MethodHandle.bindTo(java.lang.Object)`

dropArguments

```
public static MethodHandle dropArguments(MethodHandle target,
                                         int pos,
                                         List<Class<?>> valueTypes)
```

Produces a method handle which will discard some dummy arguments before calling some other specified *target* method handle. The type of the new method handle will be the same as the target's type, except it will also include the dummy argument types, at some given position.

The `pos` argument may range between zero and *N*, where *N* is the arity of the target. If `pos` is zero, the dummy arguments will precede the target's real arguments; if `pos` is *N* they will come after.

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle cat = lookup().findVirtual(String.class,
  "concat", methodType(String.class, String.class));
```

```
     assertEquals("xy", (String) cat.invokeExact("x", "y"));
     MethodType bigType = cat.type().insertParameterTypes(0, int.class, String.class);
     MethodHandle d0 = dropArguments(cat, 0, bigType.parameterList().subList(0,2));
     assertEquals(bigType, d0.type());
     assertEquals("yz", (String) d0.invokeExact(123, "x", "y", "z"));
```

This method is also equivalent to the following code:

```
      dropArguments(target, pos, valueTypes.toArray(new Class[0]))
```

Parameters:

`target` - the method handle to invoke after the arguments are dropped

`pos` - position of first argument to drop (zero for the leftmost)

`valueTypes` - the type(s) of the argument(s) to drop

Returns:

a method handle which drops arguments of the given types, before calling the original method handle

Throws:

`NullPointerException` - if the target is null, or if the `valueTypes` list or any of its elements is null

`IllegalArgumentException` - if any element of `valueTypes` is `void.class`, or if `pos` is negative or greater than the arity of the target, or if the new method handle's type would have too many parameters

dropArguments

```
public static MethodHandle dropArguments(MethodHandle target,
                                         int pos,
                                         Class<?>... valueTypes)
```

Produces a method handle which will discard some dummy arguments before calling some other specified *target* method handle. The type of the new method handle will be the same as the target's type, except it will also include the dummy argument types, at some given position.

The `pos` argument may range between zero and *N*, where *N* is the arity of the target. If `pos` is zero, the dummy arguments will precede the target's real arguments; if `pos` is *N* they will come after.

API Note:

```
     import static java.lang.invoke.MethodHandles.*;
     import static java.lang.invoke.MethodType.*;
     ...
     MethodHandle cat = lookup().findVirtual(String.class,
       "concat", methodType(String.class, String.class));
     assertEquals("xy", (String) cat.invokeExact("x", "y"));
     MethodHandle d0 = dropArguments(cat, 0, String.class);
     assertEquals("yz", (String) d0.invokeExact("x", "y", "z"));
     MethodHandle d1 = dropArguments(cat, 1, String.class);
     assertEquals("xz", (String) d1.invokeExact("x", "y", "z"));
     MethodHandle d2 = dropArguments(cat, 2, String.class);
     assertEquals("xy", (String) d2.invokeExact("x", "y", "z"));
     MethodHandle d12 = dropArguments(cat, 1, int.class, boolean.class);
     assertEquals("xz", (String) d12.invokeExact("x", 12, true, "z"));
```

This method is also equivalent to the following code:

```
      dropArguments(target, pos, Arrays.asList(valueTypes))
```

Parameters:

`target` - the method handle to invoke after the arguments are dropped

`pos` - position of first argument to drop (zero for the leftmost)

`valueTypes` - the type(s) of the argument(s) to drop

Returns:

a method handle which drops arguments of the given types, before calling the original method handle

Throws:

`NullPointerException` - if the target is null, or if the `valueTypes` array or any of its elements is null

`IllegalArgumentException` - if any element of `valueTypes` is `void.class`, or if `pos` is negative or greater than the arity of the target, or if the new method handle's type would have too many parameters

dropArgumentsToMatch

```
public static MethodHandle dropArgumentsToMatch(MethodHandle target,
                                                int skip,
                                                List<Class<?>> newTypes,
                                                int pos)
```

Adapts a target method handle to match the given parameter type list. If necessary, adds dummy arguments. Some leading parameters can be skipped before matching begins. The remaining types in the `target`'s parameter type list must be a sub-list of the `newTypes` type list at the starting position `pos`. The resulting handle will have the target handle's parameter type list, with any non-matching parameter types (before or after the matching sub-list) inserted in corresponding positions of the target's original parameters, as if by `dropArguments(MethodHandle, int, Class[])`.

The resulting handle will have the same return type as the target handle.

In more formal terms, assume these two type lists:

- The target handle has the parameter type list `S..., M...`, with as many types in `S` as indicated by `skip`. The `M` types are those that are supposed to match part of the given type list, `newTypes`.
- The `newTypes` list contains types `P..., M..., A...`, with as many types in `P` as indicated by `pos`. The `M` types are precisely those that the `M` types in the target handle's parameter type list are supposed to match. The types in `A` are additional types found after the matching sub-list.

Given these assumptions, the result of an invocation of `dropArgumentsToMatch` will have the parameter type list `S..., P..., M..., A...`, with the `P` and `A` types inserted as if by `dropArguments(MethodHandle, int, Class[])`.

API Note:

Two method handles whose argument lists are "effectively identical" (i.e., identical in a common prefix) may be mutually converted to a common type by two calls to `dropArgumentsToMatch`, as follows:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
...
MethodHandle h0 = constant(boolean.class, true);
MethodHandle h1 = lookup().findVirtual(String.class, "concat", methodType(String.class, String.class));
MethodType bigType = h1.type().insertParameterTypes(1, String.class, int.class);
MethodHandle h2 = dropArguments(h1, 0, bigType.parameterList());
if (h1.type().parameterCount() < h2.type().parameterCount())
    h1 = dropArgumentsToMatch(h1, 0, h2.type().parameterList(), 0);  // lengthen h1
else
    h2 = dropArgumentsToMatch(h2, 0, h1.type().parameterList(), 0);    // lengthen h2
MethodHandle h3 = guardWithTest(h0, h1, h2);
assertEquals("xy", h3.invoke("x", "y", 1, "a", "b", "c"));
```

Parameters:

`target` - the method handle to adapt

`skip` - number of targets parameters to disregard (they will be unchanged)

`newTypes` - the list of types to match `target`'s parameter type list to

`pos` - place in `newTypes` where the non-skipped target parameters must occur

Returns:

a possibly adapted method handle

Throws:

`NullPointerException` - if either argument is null

`IllegalArgumentException` - if any element of `newTypes` is `void.class`, or if `skip` is negative or greater than the arity of the target, or if `pos` is negative or greater than the newTypes list size, or if `newTypes` does not contain the `target`'s non-skipped parameter types at position `pos`.

Since:

9

dropReturn

```
public static MethodHandle dropReturn(MethodHandle target)
```

Drop the return value of the target handle (if any). The returned method handle will have a `void` return type.

Parameters:

`target` - the method handle to adapt

Returns:

a possibly adapted method handle

Throws:

`NullPointerException` - if `target` is null

Since:

16

filterArguments

```
public static MethodHandle filterArguments(MethodHandle target,
                                           int pos,
                                           MethodHandle... filters)
```

Adapts a target method handle by pre-processing one or more of its arguments, each with its own unary filter function, and then calling the target with each pre-processed argument replaced by the result of its corresponding filter function.

The pre-processing is performed by one or more method handles, specified in the elements of the `filters` array. The first element of the filter array corresponds to the `pos` argument of the target, and so on in sequence. The filter functions are invoked in left to right order.

Null arguments in the array are treated as identity functions, and the corresponding arguments left unchanged. (If there are no non-null elements in the array, the original target is returned.) Each filter is applied to the corresponding argument of the adapter.

If a filter `F` applies to the `N`th argument of the target, then `F` must be a method handle which takes exactly one argument. The type of `F`'s sole argument replaces the corresponding argument type of the target in the resulting adapted method handle. The return type of `F` must be identical to the corresponding parameter type of the target.

It is an error if there are elements of `filters` (null or not) which do not correspond to argument positions in the target.

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle cat = lookup().findVirtual(String.class,
  "concat", methodType(String.class, String.class));
MethodHandle upcase = lookup().findVirtual(String.class,
  "toUpperCase", methodType(String.class));
assertEquals("xy", (String) cat.invokeExact("x", "y"));
MethodHandle f0 = filterArguments(cat, 0, upcase);
assertEquals("Xy", (String) f0.invokeExact("x", "y")); // Xy
MethodHandle f1 = filterArguments(cat, 1, upcase);
assertEquals("xY", (String) f1.invokeExact("x", "y")); // xY
MethodHandle f2 = filterArguments(cat, 0, upcase, upcase);
assertEquals("XY", (String) f2.invokeExact("x", "y")); // XY
```

Here is pseudocode for the resulting adapter. In the code, `T` denotes the return type of both the `target` and resulting adapter. `P`/`p` and `B`/`b` represent the types and values of the parameters and arguments that precede and follow the filter position `pos`, respectively. `A[i]`/`a[i]` stand for the types and values of the filtered parameters and arguments; they also represent the return types of the `filter[i]` handles. The latter accept arguments `v[i]` of type `V[i]`, which also appear in the signature of the resulting adapter.

```
T target(P... p, A[i]... a[i], B... b);
A[i] filter[i](V[i]);
T adapter(P... p, V[i]... v[i], B... b) {
  return target(p..., filter[i](v[i])..., b...);
}
```

Note: The resulting adapter is never a variable-arity method handle, even if the original target method handle was.

Parameters:

`target` - the method handle to invoke after arguments are filtered

`pos` - the position of the first argument to filter

`filters` - method handles to call initially on filtered arguments

Returns:

method handle which incorporates the specified argument filtering logic

Throws:

`NullPointerException` - if the target is null or if the `filters` array is null

`IllegalArgumentException` - if a non-null element of `filters` does not match a corresponding argument type of target as described above, or if the `pos+filters.length` is greater than `target.type().parameterCount()`, or if the resulting method handle's type would have too many parameters

collectArguments

```
public static MethodHandle collectArguments(MethodHandle target,
                                            int pos,
                                            MethodHandle filter)
```

Adapts a target method handle by pre-processing a sub-sequence of its arguments with a filter (another method handle). The pre-processed arguments are replaced by the result (if any) of the filter function. The target is then called on the modified (usually shortened) argument list.

If the filter returns a value, the target must accept that value as its argument in position pos, preceded and/or followed by any arguments not passed to the filter. If the filter returns void, the target must accept all arguments not passed to the filter. No arguments are reordered, and a result returned from the filter replaces (in order) the whole subsequence of arguments originally passed to the adapter.

The argument types (if any) of the filter replace zero or one argument types of the target, at position pos, in the resulting adapted method handle. The return type of the filter (if any) must be identical to the argument type of the target at position pos, and that target argument is supplied by the return value of the filter.

In all cases, pos must be greater than or equal to zero, and pos must also be less than or equal to the target's arity.

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle deepToString = publicLookup()
  .findStatic(Arrays.class, "deepToString", methodType(String.class, Object[].class));

MethodHandle ts1 = deepToString.asCollector(String[].class, 1);
assertEquals("[strange]", (String) ts1.invokeExact("strange"));

MethodHandle ts2 = deepToString.asCollector(String[].class, 2);
assertEquals("[up, down]", (String) ts2.invokeExact("up", "down"));

MethodHandle ts3 = deepToString.asCollector(String[].class, 3);
MethodHandle ts3_ts2 = collectArguments(ts3, 1, ts2);
assertEquals("[top, [up, down], strange]",
             (String) ts3_ts2.invokeExact("top", "up", "down", "strange"));

MethodHandle ts3_ts2_ts1 = collectArguments(ts3_ts2, 3, ts1);
assertEquals("[top, [up, down], [strange]]",
             (String) ts3_ts2_ts1.invokeExact("top", "up", "down", "strange"));

MethodHandle ts3_ts2_ts3 = collectArguments(ts3_ts2, 1, ts3);
assertEquals("[top, [[up, down, strange], charm], bottom]",
             (String) ts3_ts2_ts3.invokeExact("top", "up", "down", "strange", "charm", "bottom"));
```

Here is pseudocode for the resulting adapter. In the code, T represents the return type of the target and resulting adapter. V/v stand for the return type and value of the filter, which are also found in the signature and arguments of the target, respectively, unless V is void. A/a and C/c represent the parameter types and values preceding and following the collection position, pos, in the target's signature. They also turn up in the resulting adapter's signature and arguments, where they surround B/b, which represent the parameter types and arguments to the filter (if any).

```
T target(A...,V,C...);
V filter(B...);
T adapter(A... a,B... b,C... c) {
  V v = filter(b...);
  return target(a...,v,c...);
}
// and if the filter has no arguments:
T target2(A...,V,C...);
V filter2();
T adapter2(A... a,C... c) {
  V v = filter2();
  return target2(a...,v,c...);
}
// and if the filter has a void return:
T target3(A...,C...);
void filter3(B...);
T adapter3(A... a,B... b,C... c) {
  filter3(b...);
  return target3(a...,c...);
}
```

A collection adapter collectArguments(mh, 0, coll) is equivalent to one which first "folds" the affected arguments, and then drops them, in separate steps as follows:

```
mh = MethodHandles.dropArguments(mh, 1, coll.type().parameterList()); //step 2
mh = MethodHandles.foldArguments(mh, coll); //step 1
```

If the target method handle consumes no arguments besides than the result (if any) of the filter coll, then collectArguments(mh, 0, coll) is equivalent to filterReturnValue(coll, mh). If the filter method handle coll consumes one argument and produces a non-void result, then collectArguments(mh, N, coll) is equivalent to filterArguments(mh, N, coll). Other equivalences are possible but would require argument permutation.

Note: The resulting adapter is never a variable-arity method handle, even if the original target method handle was.

Parameters:

`target` - the method handle to invoke after filtering the subsequence of arguments

`pos` - the position of the first adapter argument to pass to the filter, and/or the target argument which receives the result of the filter

`filter` - method handle to call on the subsequence of arguments

Returns:

method handle which incorporates the specified argument subsequence filtering logic

Throws:

`NullPointerException` - if either argument is null

`IllegalArgumentException` - if the return type of `filter` is non-void and is not the same as the `pos` argument of the target, or if `pos` is not between 0 and the target's arity, inclusive, or if the resulting method handle's type would have too many parameters

See Also:

`foldArguments(java.lang.invoke.MethodHandle, java.lang.invoke.MethodHandle)`, `filterArguments(java.lang.invoke.MethodHandle, int, java.lang.invoke.MethodHandle...)`, `filterReturnValue(java.lang.invoke.MethodHandle, java.lang.invoke.MethodHandle)`

filterReturnValue

```
public static MethodHandle filterReturnValue(MethodHandle target,
                                             MethodHandle filter)
```

Adapts a target method handle by post-processing its return value (if any) with a filter (another method handle). The result of the filter is returned from the adapter.

If the target returns a value, the filter must accept that value as its only argument. If the target returns void, the filter must accept no arguments.

The return type of the filter replaces the return type of the target in the resulting adapted method handle. The argument type of the filter (if any) must be identical to the return type of the target.

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle cat = lookup().findVirtual(String.class,
  "concat", methodType(String.class, String.class));
MethodHandle length = lookup().findVirtual(String.class,
  "length", methodType(int.class));
System.out.println((String) cat.invokeExact("x", "y")); // xy
MethodHandle f0 = filterReturnValue(cat, length);
System.out.println((int) f0.invokeExact("x", "y")); // 2
```

Here is pseudocode for the resulting adapter. In the code, `T`/`t` represent the result type and value of the `target`; `V`, the result type of the `filter`; and `A`/`a`, the types and values of the parameters and arguments of the `target` as well as the resulting adapter.

```
T target(A...);
V filter(T);
V adapter(A... a) {
  T t = target(a...);
  return filter(t);
}
// and if the target has a void return:
void target2(A...);
V filter2();
V adapter2(A... a) {
  target2(a...);
  return filter2();
}
// and if the filter has a void return:
T target3(A...);
void filter3(V);
void adapter3(A... a) {
  T t = target3(a...);
  filter3(t);
}
```

Note: The resulting adapter is never a variable-arity method handle, even if the original target method handle was.

Parameters:

`target` - the method handle to invoke before filtering the return value

`filter` - method handle to call on the return value

Returns:

method handle which incorporates the specified return value filtering logic

Throws:

NullPointerException - if either argument is null

IllegalArgumentException - if the argument list of `filter` does not match the return type of target as described above

foldArguments

```
public static MethodHandle foldArguments(MethodHandle target,
                                         MethodHandle combiner)
```

Adapts a target method handle by pre-processing some of its arguments, and then calling the target with the result of the pre-processing, inserted into the original sequence of arguments.

The pre-processing is performed by `combiner`, a second method handle. Of the arguments passed to the adapter, the first `N` arguments are copied to the combiner, which is then called. (Here, `N` is defined as the parameter count of the combiner.) After this, control passes to the target, with any result from the combiner inserted before the original `N` incoming arguments.

If the combiner returns a value, the first parameter type of the target must be identical with the return type of the combiner, and the next `N` parameter types of the target must exactly match the parameters of the combiner.

If the combiner has a void return, no result will be inserted, and the first `N` parameter types of the target must exactly match the parameters of the combiner.

The resulting adapter is the same type as the target, except that the first parameter type is dropped, if it corresponds to the result of the combiner.

(Note that `dropArguments` can be used to remove any arguments that either the combiner or the target does not wish to receive. If some of the incoming arguments are destined only for the combiner, consider using `asCollector` instead, since those arguments will not need to be live on the stack on entry to the target.)

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle trace = publicLookup().findVirtual(java.io.PrintStream.class,
  "println", methodType(void.class, String.class))
    .bindTo(System.out);
MethodHandle cat = lookup().findVirtual(String.class,
  "concat", methodType(String.class, String.class));
assertEquals("boojum", (String) cat.invokeExact("boo", "jum"));
MethodHandle catTrace = foldArguments(cat, trace);
// also prints "boo":
assertEquals("boojum", (String) catTrace.invokeExact("boo", "jum"));
```

Here is pseudocode for the resulting adapter. In the code, `T` represents the result type of the `target` and resulting adapter. `V`/`v` represent the type and value of the parameter and argument of `target` that precedes the folding position; `V` also is the result type of the `combiner`. `A`/`a` denote the types and values of the `N` parameters and arguments at the folding position. `B`/`b` represent the types and values of the `target` parameters and arguments that follow the folded parameters and arguments.

```
// there are N arguments in A...
T target(V, A[N]..., B...);
V combiner(A...);
T adapter(A... a, B... b) {
  V v = combiner(a...);
  return target(v, a..., b...);
}
// and if the combiner has a void return:
T target2(A[N]..., B...);
void combiner2(A...);
T adapter2(A... a, B... b) {
  combiner2(a...);
  return target2(a..., b...);
}
```

Note: The resulting adapter is never a variable-arity method handle, even if the original target method handle was.

Parameters:

`target` - the method handle to invoke after arguments are combined

`combiner` - method handle to call initially on the incoming arguments

Returns:

method handle which incorporates the specified argument folding logic

Throws:

NullPointerException - if either argument is null

IllegalArgumentException - if combiner's return type is non-void and not the same as the first argument type of the target, or if the initial N argument types of the target (skipping one matching the combiner's return type) are not identical with the argument types of combiner

foldArguments

```
public static MethodHandle foldArguments(MethodHandle target,
                                         int pos,
                                         MethodHandle combiner)
```

Adapts a target method handle by pre-processing some of its arguments, starting at a given position, and then calling the target with the result of the pre-processing, inserted into the original sequence of arguments just before the folded arguments.

This method is closely related to foldArguments(MethodHandle, MethodHandle), but allows to control the position in the parameter list at which folding takes place. The argument controlling this, pos, is a zero-based index. The aforementioned method foldArguments(MethodHandle, MethodHandle) assumes position 0.

API Note:

Example:

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandle trace = publicLookup().findVirtual(java.io.PrintStream.class,
"println", methodType(void.class, String.class))
.bindTo(System.out);
MethodHandle cat = lookup().findVirtual(String.class,
"concat", methodType(String.class, String.class));
assertEquals("boojum", (String) cat.invokeExact("boo", "jum"));
MethodHandle catTrace = foldArguments(cat, 1, trace);
// also prints "jum":
assertEquals("boojum", (String) catTrace.invokeExact("boo", "jum"));
```

Here is pseudocode for the resulting adapter. In the code, T represents the result type of the target and resulting adapter. V/v represent the type and value of the parameter and argument of target that precedes the folding position; V also is the result type of the combiner. A/a denote the types and values of the N parameters and arguments at the folding position. Z/z and B/b represent the types and values of the target parameters and arguments that precede and follow the folded parameters and arguments starting at pos, respectively.

```
// there are N arguments in A...
T target(Z..., V, A[N]..., B...);
V combiner(A...);
T adapter(Z... z, A... a, B... b) {
  V v = combiner(a...);
  return target(z..., v, a..., b...);
}
// and if the combiner has a void return:
T target2(Z..., A[N]..., B...);
void combiner2(A...);
T adapter2(Z... z, A... a, B... b) {
  combiner2(a...);
  return target2(z..., a..., b...);
}
```

Note: The resulting adapter is never a variable-arity method handle, even if the original target method handle was.

Parameters:

target - the method handle to invoke after arguments are combined

pos - the position at which to start folding and at which to insert the folding result; if this is 0, the effect is the same as for foldArguments(MethodHandle, MethodHandle).

combiner - method handle to call initially on the incoming arguments

Returns:

method handle which incorporates the specified argument folding logic

Throws:

NullPointerException - if either argument is null

IllegalArgumentException - if either of the following two conditions holds: (1) combiner's return type is non-void and not the same as the argument type at position pos of the target signature; (2) the N argument types at position pos of the target signature (skipping one matching the combiner's return type) are not identical with the argument types of combiner.

Since:

9

See Also:

foldArguments(MethodHandle, MethodHandle)

guardWithTest

```
public static MethodHandle guardWithTest(MethodHandle test,
                                         MethodHandle target,
                                         MethodHandle fallback)
```

Makes a method handle which adapts a target method handle, by guarding it with a test, a boolean-valued method handle. If the guard fails, a fallback handle is called instead. All three method handles must have the same corresponding argument and return types, except that the return type of the test must be boolean, and the test is allowed to have fewer arguments than the other two method handles.

Here is pseudocode for the resulting adapter. In the code, `T` represents the uniform result type of the three involved handles; `A`/`a`, the types and values of the `target` parameters and arguments that are consumed by the `test`; and `B`/`b`, those types and values of the `target` parameters and arguments that are not consumed by the `test`.

```
boolean test(A...);
T target(A...,B...);
T fallback(A...,B...);
T adapter(A... a,B... b) {
  if (test(a...))
    return target(a..., b...);
  else
    return fallback(a..., b...);
}
```

Note that the test arguments (`a...` in the pseudocode) cannot be modified by execution of the test, and so are passed unchanged from the caller to the target or fallback as appropriate.

Parameters:

`test` - method handle used for test, must return boolean

`target` - method handle to call if test passes

`fallback` - method handle to call if test fails

Returns:

method handle which incorporates the specified if/then/else logic

Throws:

`NullPointerException` - if any argument is null

`IllegalArgumentException` - if `test` does not return boolean, or if all three method types do not match (with the return type of `test` changed to match that of the target).

catchException

```
public static MethodHandle catchException(MethodHandle target,
                                          Class<? extends Throwable> exType,
                                          MethodHandle handler)
```

Makes a method handle which adapts a target method handle, by running it inside an exception handler. If the target returns normally, the adapter returns that value. If an exception matching the specified type is thrown, the fallback handle is called instead on the exception, plus the original arguments.

The target and handler must have the same corresponding argument and return types, except that handler may omit trailing arguments (similarly to the predicate in `guardWithTest`). Also, the handler must have an extra leading parameter of `exType` or a supertype.

Here is pseudocode for the resulting adapter. In the code, `T` represents the return type of the `target` and `handler`, and correspondingly that of the resulting adapter; `A`/`a`, the types and values of arguments to the resulting handle consumed by `handler`; and `B`/`b`, those of arguments to the resulting handle discarded by `handler`.

```
T target(A..., B...);
T handler(ExType, A...);
T adapter(A... a, B... b) {
  try {
    return target(a..., b...);
  } catch (ExType ex) {
    return handler(ex, a...);
  }
}
```

Note that the saved arguments (`a...` in the pseudocode) cannot be modified by execution of the target, and so are passed unchanged from the caller to the handler, if the handler is invoked.

The target and handler must return the same type, even if the handler always throws. (This might happen, for instance, because the handler is simulating a `finally` clause). To create such a throwing handler, compose the handler creation logic with `throwException`, in order to create a method handle of the correct return type.

Parameters:

`target` - method handle to call

`exType` - the type of exception which the handler will catch

`handler` - method handle to call if a matching exception is thrown

Returns:

method handle which incorporates the specified try/catch logic

Throws:

`NullPointerException` - if any argument is null

`IllegalArgumentException` - if `handler` does not accept the given exception type, or if the method handle types do not match in their return types and their corresponding parameters

See Also:

`tryFinally(MethodHandle, MethodHandle)`

throwException

```
public static MethodHandle throwException(Class<?> returnType,
                                          Class<? extends Throwable> exType)
```

Produces a method handle which will throw exceptions of the given `exType`. The method handle will accept a single argument of `exType`, and immediately throw it as an exception. The method type will nominally specify a return of `returnType`. The return type may be anything convenient: It doesn't matter to the method handle's behavior, since it will never return normally.

Parameters:

`returnType` - the return type of the desired method handle

`exType` - the parameter type of the desired method handle

Returns:

method handle which can throw the given exceptions

Throws:

`NullPointerException` - if either argument is null

loop

```
public static MethodHandle loop(MethodHandle[]... clauses)
```

Constructs a method handle representing a loop with several loop variables that are updated and checked upon each iteration. Upon termination of the loop due to one of the predicates, a corresponding finalizer is run and delivers the loop's result, which is the return value of the resulting handle.

Intuitively, every loop is formed by one or more "clauses", each specifying a local *iteration variable* and/or a loop exit. Each iteration of the loop executes each clause in order. A clause can optionally update its iteration variable; it can also optionally perform a test and conditional loop exit. In order to express this logic in terms of method handles, each clause will specify up to four independent actions:

- *init:* Before the loop executes, the initialization of an iteration variable `v` of type `V`.
- *step:* When a clause executes, an update step for the iteration variable `v`.
- *pred:* When a clause executes, a predicate execution to test for loop exit.
- *fini:* If a clause causes a loop exit, a finalizer execution to compute the loop's return value.

The full sequence of all iteration variable types, in clause order, will be notated as `(V...)`. The values themselves will be `(v...)`. When we speak of "parameter lists", we will usually be referring to types, but in some contexts (describing execution) the lists will be of actual values.

Some of these clause parts may be omitted according to certain rules, and useful default behavior is provided in this case. See below for a detailed description.

Parameters optional everywhere: Each clause function is allowed but not required to accept a parameter for each iteration variable `v`. As an exception, the init functions cannot take any `v` parameters, because those values are not yet computed when the init functions are executed. Any clause function may neglect to take any trailing subsequence of parameters it is entitled to take. In fact, any clause function may take no arguments at all.

Loop parameters: A clause function may take all the iteration variable values it is entitled to, in which case it may also take more trailing parameters. Such extra values are called *loop parameters*, with their types and values notated as `(A...)` and `(a...)`. These become the parameters of the resulting loop handle, to be supplied whenever the loop is executed. (Since init functions do not accept iteration variables `v`, any parameter to an init function is automatically a loop parameter `a`.) As with iteration variables, clause functions are allowed but not required to accept loop parameters. These loop parameters act as loop-invariant values visible across the whole loop.

Parameters visible everywhere: Each non-init clause function is permitted to observe the entire loop state, because it can be passed the full list `(v... a...)` of current iteration variable values and incoming loop parameters. The init functions can observe initial pre-loop state, in the form `(a...)`. Most clause functions will not need all of this information, but they will be formally connected to it as if by `dropArguments(java.lang.invoke.MethodHandle, int, java.util.List<java.lang.Class<?>>)`. More specifically, we shall use the notation `(V*)` to express an arbitrary prefix of a full sequence `(V...)` (and likewise for `(v*)`, `(A*)`, `(a*)`). In that notation, the general form of an init function parameter list is `(A*)`, and the general form of a non-init function parameter list is `(V*)` or `(V... A*)`.

Checking clause structure: Given a set of clauses, there is a number of checks and adjustments performed to connect all the parts of the loop. They are spelled out in detail in the steps below. In these steps, every occurrence of the word "must" corresponds to a place where `IllegalArgumentException` will be thrown if the required constraint is not met by the inputs to the loop combinator.

Effectively identical sequences: A parameter list `A` is defined to be *effectively identical* to another parameter list `B` if `A` and `B` are identical, or if `A` is shorter and is identical with a proper prefix of `B`. When speaking of an unordered set of parameter lists, we say they the set is "effectively identical" as a whole if the set contains a longest list, and all members of the set are effectively identical to that longest list. For example, any set of type sequences of the form `(V*)` is effectively identical, and the same is true if more sequences of the form `(V... A*)` are added.

Step 0: Determine clause structure.

a. The clause array (of type `MethodHandle[][]`) must be non-`null` and contain at least one element.
b. The clause array may not contain `null`s or sub-arrays longer than four elements.
c. Clauses shorter than four elements are treated as if they were padded by `null` elements to length four. Padding takes place by appending elements to the array.
d. Clauses with all `null`s are disregarded.
e. Each clause is treated as a four-tuple of functions, called "init", "step", "pred", and "fini".

Step 1A: Determine iteration variable types `(V...)`.

a. The iteration variable type for each clause is determined using the clause's init and step return types.
b. If both functions are omitted, there is no iteration variable for the corresponding clause (`void` is used as the type to indicate that). If one of them is omitted, the other's return type defines the clause's iteration variable type. If both are given, the common return type (they must be identical) defines the clause's iteration variable type.
c. Form the list of return types (in clause order), omitting all occurrences of `void`.
d. This list of types is called the "iteration variable types" (`(V...)`).

Step 1B: Determine loop parameters `(A...)`.

- Examine and collect init function parameter lists (which are of the form `(A*)`).
- Examine and collect the suffixes of the step, pred, and fini parameter lists, after removing the iteration variable types. (They must have the form `(V... A*)`; collect the `(A*)` parts only.)
- Do not collect suffixes from step, pred, and fini parameter lists that do not begin with all the iteration variable types. (These types will be checked in step 2, along with all the clause function types.)
- Omitted clause functions are ignored. (Equivalently, they are deemed to have empty parameter lists.)
- All of the collected parameter lists must be effectively identical.
- The longest parameter list (which is necessarily unique) is called the "external parameter list" (`(A...)`).
- If there is no such parameter list, the external parameter list is taken to be the empty sequence.
- The combined list consisting of iteration variable types followed by the external parameter types is called the "internal parameter list".

Step 1C: Determine loop return type.

a. Examine fini function return types, disregarding omitted fini functions.
b. If there are no fini functions, the loop return type is `void`.
c. Otherwise, the common return type `R` of the fini functions (their return types must be identical) defines the loop return type.

Step 1D: Check other types.

a. There must be at least one non-omitted pred function.
b. Every non-omitted pred function must have a `boolean` return type.

Step 2: Determine parameter lists.

a. The parameter list for the resulting loop handle will be the external parameter list `(A...)`.
b. The parameter list for init functions will be adjusted to the external parameter list. (Note that their parameter lists are already effectively identical to this list.)
c. The parameter list for every non-omitted, non-init (step, pred, and fini) function must be effectively identical to the internal parameter list `(V... A...)`.

Step 3: Fill in omitted functions.

a. If an init function is omitted, use a default value for the clause's iteration variable type.
b. If a step function is omitted, use an identity function of the clause's iteration variable type; insert dropped argument parameters before the identity function parameter for the non-`void` iteration variables of preceding clauses. (This will turn the loop variable into a local loop invariant.)
c. If a pred function is omitted, use a constant `true` function. (This will keep the loop going, as far as this clause is concerned. Note that in such cases the corresponding fini function is unreachable.)
d. If a fini function is omitted, use a default value for the loop return type.

Step 4: Fill in missing parameter types.

a. At this point, every init function parameter list is effectively identical to the external parameter list `(A...)`, but some lists may be shorter. For every init function with a short parameter list, pad out the end of the list.
b. At this point, every non-init function parameter list is effectively identical to the internal parameter list `(V... A...)`, but some lists may be shorter. For every non-init function with a short parameter list, pad out the end of the list.
c. Argument lists are padded out by dropping unused trailing arguments.

Final observations.

a. After these steps, all clauses have been adjusted by supplying omitted functions and arguments.
b. All init functions have a common parameter type list `(A...)`, which the final loop handle will also have.
c. All fini functions have a common return type `R`, which the final loop handle will also have.
d. All non-init functions have a common parameter type list `(V... A...)`, of (non-`void`) iteration variables `V` followed by loop parameters.
e. Each pair of init and step functions agrees in their return type `V`.
f. Each non-init function will be able to observe the current values `(v...)` of all iteration variables.
g. Every function will be able to observe the incoming values `(a...)` of all loop parameters.

Example. As a consequence of step 1A above, the `loop` combinator has the following property:

- Given `N` clauses `Cn = {null, Sn, Pn}` with `n = 1..N`.
- Suppose predicate handles `Pn` are either `null` or have no parameters. (Only one `Pn` has to be non-`null`.)
- Suppose step handles `Sn` have signatures `(B1..BX)Rn`, for some constant `X>=N`.
- Suppose `Q` is the count of non-void types `Rn`, and `(V1...VQ)` is the sequence of those types.
- It must be that `Vn == Bn` for `n = 1..min(X,Q)`.
- The parameter types `Vn` will be interpreted as loop-local state elements `(V...)`.
- Any remaining types `BQ+1..BX` (if `Q<X`) will determine the resulting loop handle's parameter types `(A...)`.

In this example, the loop handle parameters `(A...)` were derived from the step functions, which is natural if most of the loop computation happens in the steps. For some loops, the burden of computation might be heaviest in the pred functions, and so the pred functions might need to accept the loop parameter values. For loops with complex exit logic, the fini functions might need to accept loop parameters, and likewise for loops with complex entry logic, where the init functions will need the extra parameters. For such reasons, the rules for determining these parameters are as symmetric as possible, across all clause parts. In general, the loop parameters function as common invariant values across the whole loop, while the iteration variables function as common variant values, or (if there is no step function) as internal loop invariant temporaries.

Loop execution.

a. When the loop is called, the loop input values are saved in locals, to be passed to every clause function. These locals are loop invariant.
b. Each init function is executed in clause order (passing the external arguments `(a...)`) and the non-`void` values are saved (as the iteration variables `(v...)`) into locals. These locals will be loop varying (unless their steps behave as identity functions, as noted above).
c. All function executions (except init functions) will be passed the internal parameter list, consisting of the non-`void` iteration values `(v...)` (in clause order) and then the loop inputs `(a...)` (in argument order).
d. The step and pred functions are then executed, in clause order (step before pred), until a pred function returns `false`.
e. The non-`void` result from a step function call is used to update the corresponding value in the sequence `(v...)` of loop variables. The updated value is immediately visible to all subsequent function calls.
f. If a pred function returns `false`, the corresponding fini function is called, and the resulting value (of type `R`) is returned from the loop as a whole.
g. If all the pred functions always return true, no fini function is ever invoked, and the loop cannot exit except by throwing an exception.

Usage tips.

- Although each step function will receive the current values of *all* the loop variables, sometimes a step function only needs to observe the current value of its own variable. In that case, the step function may need to explicitly drop all preceding loop variables. This will require mentioning their types, in an expression like `dropArguments(step, 0, V0.class, ...)`.
- Loop variables are not required to vary; they can be loop invariant. A clause can create a loop invariant by a suitable init function with no step, pred, or fini function. This may be useful to "wire" an incoming loop argument into the step or pred function of an adjacent loop variable.
- If some of the clause functions are virtual methods on an instance, the instance itself can be conveniently placed in an initial invariant loop "variable", using an initial clause like `new MethodHandle[]{identity(ObjType.class)}`. In that case, the instance reference will be the first iteration variable value, and it will be easy to use virtual methods as clause parts, since all of them will take a leading instance reference matching that value.

Here is pseudocode for the resulting loop handle. As above, `V` and `v` represent the types and values of loop variables; `A` and `a` represent arguments passed to the whole loop; and `R` is the common result type of all finalizers as well as of the resulting loop.

```
V... init...(A...);
boolean pred...(V..., A...);
V... step...(V..., A...);
R fini...(V..., A...);
R loop(A... a) {
  V... v... = init...(a...);
  for (;;) {
    for ((v, p, s, f) in (v..., pred..., step..., fini...)) {
      v = s(v..., a...);
      if (!p(v..., a...)) {
        return f(v..., a...);
      }
    }
  }
}
```

Note that the parameter type lists `(V...)` and `(A...)` have been expanded to their full length, even though individual clause functions may neglect to take them all. As noted above, missing parameters are filled in as if by `dropArgumentsToMatch(MethodHandle, int, List, int)`.

API Note:

Example:

```
// iterative implementation of the factorial function as a loop handle
static int one(int k) { return 1; }
static int inc(int i, int acc, int k) { return i + 1; }
static int mult(int i, int acc, int k) { return i * acc; }
static boolean pred(int i, int acc, int k) { return i < k; }
static int fin(int i, int acc, int k) { return acc; }
// assume MH_one, MH_inc, MH_mult, MH_pred, and MH_fin are handles to the above methods
// null initializer for counter, should initialize to 0
```

```
    MethodHandle[] counterClause = new MethodHandle[]{null, MH_inc};
    MethodHandle[] accumulatorClause = new MethodHandle[]{MH_one, MH_mult, MH_pred, MH_fin};
    MethodHandle loop = MethodHandles.loop(counterClause, accumulatorClause);
    assertEquals(120, loop.invoke(5));
```

The same example, dropping arguments and using combinators:

```
    // simplified implementation of the factorial function as a loop handle
    static int inc(int i) { return i + 1; } // drop acc, k
    static int mult(int i, int acc) { return i * acc; } //drop k
    static boolean cmp(int i, int k) { return i < k; }
    // assume MH_inc, MH_mult, and MH_cmp are handles to the above methods
    // null initializer for counter, should initialize to 0
    MethodHandle MH_one = MethodHandles.constant(int.class, 1);
    MethodHandle MH_pred = MethodHandles.dropArguments(MH_cmp, 1, int.class); // drop acc
    MethodHandle MH_fin = MethodHandles.dropArguments(MethodHandles.identity(int.class), 0, int.class); // drop i
    MethodHandle[] counterClause = new MethodHandle[]{null, MH_inc};
    MethodHandle[] accumulatorClause = new MethodHandle[]{MH_one, MH_mult, MH_pred, MH_fin};
    MethodHandle loop = MethodHandles.loop(counterClause, accumulatorClause);
    assertEquals(720, loop.invoke(6));
```

A similar example, using a helper object to hold a loop parameter:

```
    // instance-based implementation of the factorial function as a loop handle
    static class FacLoop {
      final int k;
      FacLoop(int k) { this.k = k; }
      int inc(int i) { return i + 1; }
      int mult(int i, int acc) { return i * acc; }
      boolean pred(int i) { return i < k; }
      int fin(int i, int acc) { return acc; }
    }
    // assume MH_FacLoop is a handle to the constructor
    // assume MH_inc, MH_mult, MH_pred, and MH_fin are handles to the above methods
    // null initializer for counter, should initialize to 0
    MethodHandle MH_one = MethodHandles.constant(int.class, 1);
    MethodHandle[] instanceClause = new MethodHandle[]{MH_FacLoop};
    MethodHandle[] counterClause = new MethodHandle[]{null, MH_inc};
    MethodHandle[] accumulatorClause = new MethodHandle[]{MH_one, MH_mult, MH_pred, MH_fin};
    MethodHandle loop = MethodHandles.loop(instanceClause, counterClause, accumulatorClause);
    assertEquals(5040, loop.invoke(7));
```

Parameters:

`clauses` - an array of arrays (4-tuples) of `MethodHandle`s adhering to the rules described above.

Returns:

a method handle embodying the looping behavior as defined by the arguments.

Throws:

`IllegalArgumentException` - in case any of the constraints described above is violated.

Since:

9

See Also:

`whileLoop(MethodHandle, MethodHandle, MethodHandle)`,
`doWhileLoop(MethodHandle, MethodHandle, MethodHandle)`,
`countedLoop(MethodHandle, MethodHandle, MethodHandle)`,
`iteratedLoop(MethodHandle, MethodHandle, MethodHandle)`

whileLoop

```
public static MethodHandle whileLoop(MethodHandle init,
                                     MethodHandle pred,
                                     MethodHandle body)
```

Constructs a `while` loop from an initializer, a body, and a predicate. This is a convenience wrapper for the generic loop combinator.

The `pred` handle describes the loop condition; and `body`, its body. The loop resulting from this method will, in each iteration, first evaluate the predicate and then execute its body (if the predicate evaluates to `true`). The loop will terminate once the predicate evaluates to `false` (the body will not be executed in this case).

The `init` handle describes the initial value of an additional optional loop-local variable. In each iteration, this loop-local variable, if present, will be passed to the `body` and updated with the value returned from its invocation. The result of loop execution will be the final value of the additional loop-local variable (if present).

The following rules hold for these argument handles:

- The body handle must not be null; its type must be of the form (V A...)V, where V is non-void, or else (A...)void. (In the void case, we assign the type void to the name V, and we will write (V A...)V with the understanding that a void type V is quietly dropped from the parameter list, leaving (A...)V.)
- The parameter list (V A...) of the body is called the *internal parameter list*. It will constrain the parameter lists of the other loop parts.
- If the iteration variable type V is dropped from the internal parameter list, the resulting shorter list (A...) is called the *external parameter list*.
- The body return type V, if non-void, determines the type of an additional state variable of the loop. The body must both accept and return a value of this type V.
- If init is non-null, it must have return type V. Its parameter list (of some form (A*)) must be effectively identical to the external parameter list (A...).
- If init is null, the loop variable will be initialized to its default value.
- The pred handle must not be null. It must have boolean as its return type. Its parameter list (either empty or of the form (V A*)) must be effectively identical to the internal parameter list.

The resulting loop handle's result type and parameter signature are determined as follows:

- The loop handle's result type is the result type V of the body.
- The loop handle's parameter types are the types (A...), from the external parameter list.

Here is pseudocode for the resulting loop handle. In the code, V/v represent the type / value of the sole loop variable as well as the result type of the loop; and A/a, that of the argument passed to the loop.

```
V init(A...);
boolean pred(V, A...);
V body(V, A...);
V whileLoop(A... a...) {
  V v = init(a...);
  while (pred(v, a...)) {
    v = body(v, a...);
  }
  return v;
}
```

API Note:

Example:

```
// implement the zip function for lists as a loop handle
static List<String> initZip(Iterator<String> a, Iterator<String> b) { return new ArrayList<>(); }
static boolean zipPred(List<String> zip, Iterator<String> a, Iterator<String> b) { return a.hasNext() && b.hasNext();
static List<String> zipStep(List<String> zip, Iterator<String> a, Iterator<String> b) {
  zip.add(a.next());
  zip.add(b.next());
  return zip;
}
// assume MH_initZip, MH_zipPred, and MH_zipStep are handles to the above methods
MethodHandle loop = MethodHandles.whileLoop(MH_initZip, MH_zipPred, MH_zipStep);
List<String> a = Arrays.asList("a", "b", "c", "d");
List<String> b = Arrays.asList("e", "f", "g", "h");
List<String> zipped = Arrays.asList("a", "e", "b", "f", "c", "g", "d", "h");
assertEquals(zipped, (List<String>) loop.invoke(a.iterator(), b.iterator()));
```

, The implementation of this method can be expressed as follows:

```
MethodHandle whileLoop(MethodHandle init, MethodHandle pred, MethodHandle body) {
    MethodHandle fini = (body.type().returnType() == void.class
                        ? null : identity(body.type().returnType()));
    MethodHandle[]
        checkExit = { null, null, pred, fini },
        varBody   = { init, body };
    return loop(checkExit, varBody);
}
```

Parameters:

init - optional initializer, providing the initial value of the loop variable. May be null, implying a default initial value. See above for other constraints.

pred - condition for the loop, which may not be null. Its result type must be boolean. See above for other constraints.

body - body of the loop, which may not be null. It controls the loop parameters and result type. See above for other constraints.

Returns:

a method handle implementing the while loop as described by the arguments.

Throws:

IllegalArgumentException - if the rules for the arguments are violated.

NullPointerException - if pred or body are null.

Since:

9

See Also:

`loop(MethodHandle[][])`,
`doWhileLoop(MethodHandle, MethodHandle, MethodHandle)`

doWhileLoop

```
public static MethodHandle doWhileLoop(MethodHandle init,
                                       MethodHandle body,
                                       MethodHandle pred)
```

Constructs a `do-while` loop from an initializer, a body, and a predicate. This is a convenience wrapper for the generic loop combinator.

The `pred` handle describes the loop condition; and `body`, its body. The loop resulting from this method will, in each iteration, first execute its body and then evaluate the predicate. The loop will terminate once the predicate evaluates to `false` after an execution of the body.

The `init` handle describes the initial value of an additional optional loop-local variable. In each iteration, this loop-local variable, if present, will be passed to the `body` and updated with the value returned from its invocation. The result of loop execution will be the final value of the additional loop-local variable (if present).

The following rules hold for these argument handles:

- The `body` handle must not be `null`; its type must be of the form `(V A...)V`, where `V` is non-`void`, or else `(A...)void`. (In the `void` case, we assign the type `void` to the name `V`, and we will write `(V A...)V` with the understanding that a `void` type `V` is quietly dropped from the parameter list, leaving `(A...)V`.)
- The parameter list `(V A...)` of the body is called the *internal parameter list*. It will constrain the parameter lists of the other loop parts.
- If the iteration variable type `V` is dropped from the internal parameter list, the resulting shorter list `(A...)` is called the *external parameter list*.
- The body return type `V`, if non-`void`, determines the type of an additional state variable of the loop. The body must both accept and return a value of this type `V`.
- If `init` is non-`null`, it must have return type `V`. Its parameter list (of some form `(A*)`) must be effectively identical to the external parameter list `(A...)`.
- If `init` is `null`, the loop variable will be initialized to its default value.
- The `pred` handle must not be `null`. It must have `boolean` as its return type. Its parameter list (either empty or of the form `(V A*)`) must be effectively identical to the internal parameter list.

The resulting loop handle's result type and parameter signature are determined as follows:

- The loop handle's result type is the result type `V` of the body.
- The loop handle's parameter types are the types `(A...)`, from the external parameter list.

Here is pseudocode for the resulting loop handle. In the code, `V`/`v` represent the type / value of the sole loop variable as well as the result type of the loop; and `A`/`a`, that of the argument passed to the loop.

```
V init(A...);
boolean pred(V, A...);
V body(V, A...);
V doWhileLoop(A... a...) {
  V v = init(a...);
  do {
    v = body(v, a...);
  } while (pred(v, a...));
  return v;
}
```

API Note:

Example:

```
// int i = 0; while (i < limit) { ++i; } return i; => limit
static int zero(int limit) { return 0; }
static int step(int i, int limit) { return i + 1; }
static boolean pred(int i, int limit) { return i < limit; }
// assume MH_zero, MH_step, and MH_pred are handles to the above methods
MethodHandle loop = MethodHandles.doWhileLoop(MH_zero, MH_step, MH_pred);
assertEquals(23, loop.invoke(23));
```

, The implementation of this method can be expressed as follows:

```
MethodHandle doWhileLoop(MethodHandle init, MethodHandle body, MethodHandle pred) {
    MethodHandle fini = (body.type().returnType() == void.class
                        ? null : identity(body.type().returnType()));
    MethodHandle[] clause = { init, body, pred, fini };
    return loop(clause);
}
```

Parameters:

`init` - optional initializer, providing the initial value of the loop variable. May be `null`, implying a default initial value. See above for other constraints.

`body` - body of the loop, which may not be `null`. It controls the loop parameters and result type. See above for other constraints.

`pred` - condition for the loop, which may not be `null`. Its result type must be `boolean`. See above for other constraints.

Returns:

a method handle implementing the `while` loop as described by the arguments.

Throws:

`IllegalArgumentException` - if the rules for the arguments are violated.

`NullPointerException` - if `pred` or `body` are `null`.

Since:

9

See Also:

`loop(MethodHandle[][])`,
`whileLoop(MethodHandle, MethodHandle, MethodHandle)`

countedLoop

```
public static MethodHandle countedLoop(MethodHandle iterations,
                                       MethodHandle init,
                                       MethodHandle body)
```

Constructs a loop that runs a given number of iterations. This is a convenience wrapper for the generic loop combinator.

The number of iterations is determined by the `iterations` handle evaluation result. The loop counter `i` is an extra loop iteration variable of type `int`. It will be initialized to 0 and incremented by 1 in each iteration.

If the `body` handle returns a non-`void` type `V`, a leading loop iteration variable of that type is also present. This variable is initialized using the optional `init` handle, or to the default value of type `V` if that handle is `null`.

In each iteration, the iteration variables are passed to an invocation of the `body` handle. A non-`void` value returned from the body (of type `V`) updates the leading iteration variable. The result of the loop handle execution will be the final `V` value of that variable (or `void` if there is no `V` variable).

The following rules hold for the argument handles:

- The `iterations` handle must not be `null`, and must return the type `int`, referred to here as `I` in parameter type lists.
- The `body` handle must not be `null`; its type must be of the form `(V I A...)V`, where `V` is non-`void`, or else `(I A...)void`. (In the `void` case, we assign the type `void` to the name `V`, and we will write `(V I A...)V` with the understanding that a `void` type `V` is quietly dropped from the parameter list, leaving `(I A...)V`.)
- The parameter list `(V I A...)` of the body contributes to a list of types called the *internal parameter list*. It will constrain the parameter lists of the other loop parts.
- As a special case, if the body contributes only `V` and `I` types, with no additional `A` types, then the internal parameter list is extended by the argument types `A...` of the `iterations` handle.
- If the iteration variable types `(V I)` are dropped from the internal parameter list, the resulting shorter list `(A...)` is called the *external parameter list*.
- The body return type `V`, if non-`void`, determines the type of an additional state variable of the loop. The body must both accept a leading parameter and return a value of this type `V`.
- If `init` is non-`null`, it must have return type `V`. Its parameter list (of some form `(A*)`) must be effectively identical to the external parameter list `(A...)`.
- If `init` is `null`, the loop variable will be initialized to its default value.
- The parameter list of `iterations` (of some form `(A*)`) must be effectively identical to the external parameter list `(A...)`.

The resulting loop handle's result type and parameter signature are determined as follows:

- The loop handle's result type is the result type `V` of the body.
- The loop handle's parameter types are the types `(A...)`, from the external parameter list.

Here is pseudocode for the resulting loop handle. In the code, `V`/`v` represent the type / value of the second loop variable as well as the result type of the loop; and `A...`/`a...` represent arguments passed to the loop.

```
int iterations(A...);
V init(A...);
V body(V, int, A...);
V countedLoop(A... a...) {
  int end = iterations(a...);
  V v = init(a...);
  for (int i = 0; i < end; ++i) {
    v = body(v, i, a...);
  }
  return v;
}
```

API Note:

Example with a fully conformant body method:

```
// String s = "Lambdaman!"; for (int i = 0; i < 13; ++i) { s = "na " + s; } return s;
// => a variation on a well known theme
static String step(String v, int counter, String init) { return "na " + v; }
// assume MH_step is a handle to the method above
MethodHandle fit13 = MethodHandles.constant(int.class, 13);
MethodHandle start = MethodHandles.identity(String.class);
MethodHandle loop = MethodHandles.countedLoop(fit13, start, MH_step);
assertEquals("na na na na na na na na na na na na na Lambdaman!", loop.invoke("Lambdaman!"));
```

, Example with the simplest possible body method type, and passing the number of iterations to the loop invocation:

```
// String s = "Lambdaman!"; for (int i = 0; i < 13; ++i) { s = "na " + s; } return s;
// => a variation on a well known theme
static String step(String v, int counter ) { return "na " + v; }
// assume MH_step is a handle to the method above
MethodHandle count = MethodHandles.dropArguments(MethodHandles.identity(int.class), 1, String.class);
MethodHandle start = MethodHandles.dropArguments(MethodHandles.identity(String.class), 0, int.class);
MethodHandle loop = MethodHandles.countedLoop(count, start, MH_step);  // (v, i) -> "na " + v
assertEquals("na na na na na na na na na na na na na Lambdaman!", loop.invoke(13, "Lambdaman!"));
```

, Example that treats the number of iterations, string to append to, and string to append as loop parameters:

```
// String s = "Lambdaman!", t = "na"; for (int i = 0; i < 13; ++i) { s = t + " " + s; } return s;
// => a variation on a well known theme
static String step(String v, int counter, int iterations_, String pre, String start_) { return pre + " " + v; }
// assume MH_step is a handle to the method above
MethodHandle count = MethodHandles.identity(int.class);
MethodHandle start = MethodHandles.dropArguments(MethodHandles.identity(String.class), 0, int.class, String.class);
MethodHandle loop = MethodHandles.countedLoop(count, start, MH_step);  // (v, i, _, pre, _) -> pre + " " + v
assertEquals("na na na na na na na na na na na na na Lambdaman!", loop.invoke(13, "na", "Lambdaman!"));
```

, Example that illustrates the usage of `dropArgumentsToMatch(MethodHandle, int, List, int)` to enforce a loop type:

```
// String s = "Lambdaman!", t = "na"; for (int i = 0; i < 13; ++i) { s = t + " " + s; } return s;
// => a variation on a well known theme
static String step(String v, int counter, String pre) { return pre + " " + v; }
// assume MH_step is a handle to the method above
MethodType loopType = methodType(String.class, String.class, int.class, String.class);
MethodHandle count = MethodHandles.dropArgumentsToMatch(MethodHandles.identity(int.class),    0, loopType.parameterLis
MethodHandle start = MethodHandles.dropArgumentsToMatch(MethodHandles.identity(String.class), 0, loopType.parameterLis
MethodHandle body  = MethodHandles.dropArgumentsToMatch(MH_step,                              2, loopType.parameterLis
MethodHandle loop = MethodHandles.countedLoop(count, start, body);  // (v, i, pre, _, _) -> pre + " " + v
assertEquals("na na na na na na na na na na na na na Lambdaman!", loop.invoke("na", 13, "Lambdaman!"));
```

, The implementation of this method can be expressed as follows:

```
MethodHandle countedLoop(MethodHandle iterations, MethodHandle init, MethodHandle body) {
    return countedLoop(empty(iterations.type()), iterations, init, body);
}
```

Parameters:

`iterations` - a non-`null` handle to return the number of iterations this loop should run. The handle's result type must be `int`. See above for other constraints.

`init` - optional initializer, providing the initial value of the loop variable. May be `null`, implying a default initial value. See above for other constraints.

`body` - body of the loop, which may not be `null`. It controls the loop parameters and result type in the standard case (see above for details). It must accept its own return type (if non-void) plus an `int` parameter (for the counter), and may accept any number of additional types. See above for other constraints.

Returns:

a method handle representing the loop.

Throws:

`NullPointerException` - if either of the `iterations` or `body` handles is `null`.

`IllegalArgumentException` - if any argument violates the rules formulated above.

Since:

9

See Also:

countedLoop(MethodHandle, MethodHandle, MethodHandle, MethodHandle)

countedLoop

```
public static MethodHandle countedLoop(MethodHandle start,
                                       MethodHandle end,
                                       MethodHandle init,
                                       MethodHandle body)
```

Constructs a loop that counts over a range of numbers. This is a convenience wrapper for the generic loop combinator.

The loop counter `i` is a loop iteration variable of type `int`. The `start` and `end` handles determine the start (inclusive) and end (exclusive) values of the loop counter. The loop counter will be initialized to the `int` value returned from the evaluation of the `start` handle and run to the value returned from `end` (exclusively) with a step width of 1.

If the `body` handle returns a non-`void` type `V`, a leading loop iteration variable of that type is also present. This variable is initialized using the optional `init` handle, or to the default value of type `V` if that handle is `null`.

In each iteration, the iteration variables are passed to an invocation of the `body` handle. A non-`void` value returned from the body (of type `V`) updates the leading iteration variable. The result of the loop handle execution will be the final `V` value of that variable (or `void` if there is no `V` variable).

The following rules hold for the argument handles:

- The `start` and `end` handles must not be `null`, and must both return the common type `int`, referred to here as `I` in parameter type lists.
- The `body` handle must not be `null`; its type must be of the form `(V I A...)V`, where `V` is non-`void`, or else `(I A...)void`. (In the `void` case, we assign the type `void` to the name `V`, and we will write `(V I A...)V` with the understanding that a `void` type `V` is quietly dropped from the parameter list, leaving `(I A...)V`.)
- The parameter list `(V I A...)` of the body contributes to a list of types called the *internal parameter list*. It will constrain the parameter lists of the other loop parts.
- As a special case, if the body contributes only `V` and `I` types, with no additional `A` types, then the internal parameter list is extended by the argument types `A...` of the `end` handle.
- If the iteration variable types `(V I)` are dropped from the internal parameter list, the resulting shorter list `(A...)` is called the *external parameter list*.
- The body return type `V`, if non-`void`, determines the type of an additional state variable of the loop. The body must both accept a leading parameter and return a value of this type `V`.
- If `init` is non-`null`, it must have return type `V`. Its parameter list (of some form `(A*)`) must be effectively identical to the external parameter list `(A...)`.
- If `init` is `null`, the loop variable will be initialized to its default value.
- The parameter list of `start` (of some form `(A*)`) must be effectively identical to the external parameter list `(A...)`.
- Likewise, the parameter list of `end` must be effectively identical to the external parameter list.

The resulting loop handle's result type and parameter signature are determined as follows:

- The loop handle's result type is the result type `V` of the body.
- The loop handle's parameter types are the types `(A...)`, from the external parameter list.

Here is pseudocode for the resulting loop handle. In the code, `V`/`v` represent the type / value of the second loop variable as well as the result type of the loop; and `A...`/`a...` represent arguments passed to the loop.

```
 int start(A...);
 int end(A...);
 V init(A...);
 V body(V, int, A...);
 V countedLoop(A... a...) {
   int e = end(a...);
   int s = start(a...);
   V v = init(a...);
   for (int i = s; i < e; ++i) {
     v = body(v, i, a...);
   }
   return v;
 }
```

API Note:

The implementation of this method can be expressed as follows:

```
 MethodHandle countedLoop(MethodHandle start, MethodHandle end, MethodHandle init, MethodHandle body) {
     MethodHandle returnVar = dropArguments(identity(init.type().returnType()), 0, int.class, int.class);
     // assume MH_increment and MH_predicate are handles to implementation-internal methods with
     // the following semantics:
     // MH_increment: (int limit, int counter) -> counter + 1
     // MH_predicate: (int limit, int counter) -> counter < limit
     Class<?> counterType = start.type().returnType();  // int
     Class<?> returnType = body.type().returnType();
     MethodHandle incr = MH_increment, pred = MH_predicate, retv = null;
     if (returnType != void.class) {  // ignore the V variable
         incr = dropArguments(incr, 1, returnType);  // (limit, v, i) => (limit, i)
```

```
            pred = dropArguments(pred, 1, returnType);  // ditto
            retv = dropArguments(identity(returnType), 0, counterType); // ignore limit
        }
        body = dropArguments(body, 0, counterType);  // ignore the limit variable
        MethodHandle[]
            loopLimit  = { end, null, pred, retv }, // limit = end(); i < limit || return v
            bodyClause = { init, body },            // v = init(); v = body(v, i)
            indexVar   = { start, incr };           // i = start(); i = i + 1
        return loop(loopLimit, bodyClause, indexVar);
    }
```

Parameters:

`start` - a non-`null` handle to return the start value of the loop counter, which must be `int`. See above for other constraints.

`end` - a non-`null` handle to return the end value of the loop counter (the loop will run to `end-1`). The result type must be `int`. See above for other constraints.

`init` - optional initializer, providing the initial value of the loop variable. May be `null`, implying a default initial value. See above for other constraints.

`body` - body of the loop, which may not be `null`. It controls the loop parameters and result type in the standard case (see above for details). It must accept its own return type (if non-void) plus an `int` parameter (for the counter), and may accept any number of additional types. See above for other constraints.

Returns:

a method handle representing the loop.

Throws:

`NullPointerException` - if any of the `start`, `end`, or `body` handles is `null`.

`IllegalArgumentException` - if any argument violates the rules formulated above.

Since:

9

See Also:

`countedLoop(MethodHandle, MethodHandle, MethodHandle)`

iteratedLoop

```
public static MethodHandle iteratedLoop(MethodHandle iterator,
                                        MethodHandle init,
                                        MethodHandle body)
```

Constructs a loop that ranges over the values produced by an `Iterator<T>`. This is a convenience wrapper for the generic loop combinator.

The iterator itself will be determined by the evaluation of the `iterator` handle. Each value it produces will be stored in a loop iteration variable of type `T`.

If the `body` handle returns a non-`void` type `V`, a leading loop iteration variable of that type is also present. This variable is initialized using the optional `init` handle, or to the default value of type `V` if that handle is `null`.

In each iteration, the iteration variables are passed to an invocation of the `body` handle. A non-`void` value returned from the body (of type `V`) updates the leading iteration variable. The result of the loop handle execution will be the final `V` value of that variable (or `void` if there is no `V` variable).

The following rules hold for the argument handles:

- The `body` handle must not be `null`; its type must be of the form `(V T A...)V`, where `V` is non-`void`, or else `(T A...)void`. (In the `void` case, we assign the type `void` to the name `V`, and we will write `(V T A...)V` with the understanding that a `void` type `V` is quietly dropped from the parameter list, leaving `(T A...)V`.)
- The parameter list `(V T A...)` of the body contributes to a list of types called the *internal parameter list*. It will constrain the parameter lists of the other loop parts.
- As a special case, if the body contributes only `V` and `T` types, with no additional `A` types, then the internal parameter list is extended by the argument types `A...` of the `iterator` handle; if it is `null` the single type `Iterable` is added and constitutes the `A...` list.
- If the iteration variable types `(V T)` are dropped from the internal parameter list, the resulting shorter list `(A...)` is called the *external parameter list*.
- The body return type `V`, if non-`void`, determines the type of an additional state variable of the loop. The body must both accept a leading parameter and return a value of this type `V`.
- If `init` is non-`null`, it must have return type `V`. Its parameter list (of some form `(A*)`) must be effectively identical to the external parameter list `(A...)`.
- If `init` is `null`, the loop variable will be initialized to its default value.
- If the `iterator` handle is non-`null`, it must have the return type `java.util.Iterator` or a subtype thereof. The iterator it produces when the loop is executed will be assumed to yield values which can be converted to type `T`.
- The parameter list of an `iterator` that is non-`null` (of some form `(A*)`) must be effectively identical to the external parameter list `(A...)`.
- If `iterator` is `null` it defaults to a method handle which behaves like `Iterable.iterator()`. In that case, the internal parameter list `(V T A...)` must have at least one `A` type, and the default iterator handle parameter is adjusted to accept the leading `A` type, as if by the `asType` conversion method. The leading `A` type must be `Iterable` or a subtype thereof. This conversion step, done at loop construction time, must not throw a `WrongMethodTypeException`.

The type T may be either a primitive or reference. Since type `Iterator<T>` is erased in the method handle representation to the raw type `Iterator`, the `iteratedLoop` combinator adjusts the leading argument type for `body` to `Object` as if by the `asType` conversion method. Therefore, if an iterator of the wrong type appears as the loop is executed, runtime exceptions may occur as the result of dynamic conversions performed by `MethodHandle.asType(MethodType)`.

The resulting loop handle's result type and parameter signature are determined as follows:

- The loop handle's result type is the result type `V` of the body.
- The loop handle's parameter types are the types `(A...)`, from the external parameter list.

Here is pseudocode for the resulting loop handle. In the code, `V`/`v` represent the type / value of the loop variable as well as the result type of the loop; `T`/`t`, that of the elements of the structure the loop iterates over, and `A...`/`a...` represent arguments passed to the loop.

```
Iterator<T> iterator(A...);  // defaults to Iterable::iterator
V init(A...);
V body(V,T,A...);
V iteratedLoop(A... a...) {
  Iterator<T> it = iterator(a...);
  V v = init(a...);
  while (it.hasNext()) {
    T t = it.next();
    v = body(v, t, a...);
  }
  return v;
}
```

API Note:

Example:

```
// get an iterator from a list
static List<String> reverseStep(List<String> r, String e) {
  r.add(0, e);
  return r;
}
static List<String> newArrayList() { return new ArrayList<>(); }
// assume MH_reverseStep and MH_newArrayList are handles to the above methods
MethodHandle loop = MethodHandles.iteratedLoop(null, MH_newArrayList, MH_reverseStep);
List<String> list = Arrays.asList("a", "b", "c", "d", "e");
List<String> reversedList = Arrays.asList("e", "d", "c", "b", "a");
assertEquals(reversedList, (List<String>) loop.invoke(list));
```

, The implementation of this method can be expressed approximately as follows:

```
MethodHandle iteratedLoop(MethodHandle iterator, MethodHandle init, MethodHandle body) {
    // assume MH_next, MH_hasNext, MH_startIter are handles to methods of Iterator/Iterable
    Class<?> returnType = body.type().returnType();
    Class<?> ttype = body.type().parameterType(returnType == void.class ? 0 : 1);
    MethodHandle nextVal = MH_next.asType(MH_next.type().changeReturnType(ttype));
    MethodHandle retv = null, step = body, startIter = iterator;
    if (returnType != void.class) {
        // the simple thing first:  in (I V A...), drop the I to get V
        retv = dropArguments(identity(returnType), 0, Iterator.class);
        // body type signature (V T A...), internal loop types (I V A...)
        step = swapArguments(body, 0, 1);  // swap V <-> T
    }
    if (startIter == null)  startIter = MH_getIter;
    MethodHandle[]
        iterVar    = { startIter, null, MH_hasNext, retv }, // it = iterator; while (it.hasNext())
        bodyClause = { init, filterArguments(step, 0, nextVal) };  // v = body(v, t, a)
    return loop(iterVar, bodyClause);
}
```

Parameters:

`iterator` - an optional handle to return the iterator to start the loop. If non-`null`, the handle must return `Iterator` or a subtype. See above for other constraints.

`init` - optional initializer, providing the initial value of the loop variable. May be `null`, implying a default initial value. See above for other constraints.

`body` - body of the loop, which may not be `null`. It controls the loop parameters and result type in the standard case (see above for details). It must accept its own return type (if non-void) plus a `T` parameter (for the iterated values), and may accept any number of additional types. See above for other constraints.

Returns:

a method handle embodying the iteration loop functionality.

Throws:

`NullPointerException` - if the `body` handle is `null`.

`IllegalArgumentException` - if any argument violates the above requirements.

Since:

9

tryFinally

```
public static MethodHandle tryFinally(MethodHandle target,
                                      MethodHandle cleanup)
```

Makes a method handle that adapts a `target` method handle by wrapping it in a `try-finally` block. Another method handle, `cleanup`, represents the functionality of the `finally` block. Any exception thrown during the execution of the `target` handle will be passed to the `cleanup` handle. The exception will be rethrown, unless `cleanup` handle throws an exception first. The value returned from the `cleanup` handle's execution will be the result of the execution of the `try-finally` handle.

The `cleanup` handle will be passed one or two additional leading arguments. The first is the exception thrown during the execution of the `target` handle, or `null` if no exception was thrown. The second is the result of the execution of the `target` handle, or, if it throws an exception, a `null`, zero, or `false` value of the required type is supplied as a placeholder. The second argument is not present if the `target` handle has a `void` return type. (Note that, except for argument type conversions, combinators represent `void` values in parameter lists by omitting the corresponding paradoxical arguments, not by inserting `null` or zero values.)

The `target` and `cleanup` handles must have the same corresponding argument and return types, except that the `cleanup` handle may omit trailing arguments. Also, the `cleanup` handle must have one or two extra leading parameters:

- a `Throwable`, which will carry the exception thrown by the `target` handle (if any); and
- a parameter of the same type as the return type of both `target` and `cleanup`, which will carry the result from the execution of the `target` handle. This parameter is not present if the `target` returns `void`.

The pseudocode for the resulting adapter looks as follows. In the code, `V` represents the result type of the `try/finally` construct; `A`/`a`, the types and values of arguments to the resulting handle consumed by the cleanup; and `B`/`b`, those of arguments to the resulting handle discarded by the cleanup.

```
V target(A..., B...);
V cleanup(Throwable, V, A...);
V adapter(A... a, B... b) {
  V result = (zero value for V);
  Throwable throwable = null;
  try {
    result = target(a..., b...);
  } catch (Throwable t) {
    throwable = t;
    throw t;
  } finally {
    result = cleanup(throwable, result, a...);
  }
  return result;
}
```

Note that the saved arguments (`a...` in the pseudocode) cannot be modified by execution of the target, and so are passed unchanged from the caller to the cleanup, if it is invoked.

The target and cleanup must return the same type, even if the cleanup always throws. To create such a throwing cleanup, compose the cleanup logic with `throwException`, in order to create a method handle of the correct return type.

Note that `tryFinally` never converts exceptions into normal returns. In rare cases where exceptions must be converted in that way, first wrap the target with `catchException(MethodHandle, Class, MethodHandle)` to capture an outgoing exception, and then wrap with `tryFinally`.

It is recommended that the first parameter type of `cleanup` be declared `Throwable` rather than a narrower subtype. This ensures `cleanup` will always be invoked with whatever exception that `target` throws. Declaring a narrower type may result in a `ClassCastException` being thrown by the `try-finally` handle if the type of the exception thrown by `target` is not assignable to the first parameter type of `cleanup`. Note that various exception types of `VirtualMachineError`, `LinkageError`, and `RuntimeException` can in principle be thrown by almost any kind of Java code, and a finally clause that catches (say) only `IOException` would mask any of the others behind a `ClassCastException`.

Parameters:

`target` - the handle whose execution is to be wrapped in a `try` block.

`cleanup` - the handle that is invoked in the finally block.

Returns:

a method handle embodying the `try-finally` block composed of the two arguments.

Throws:

`NullPointerException` - if any argument is null

`IllegalArgumentException` - if `cleanup` does not accept the required leading arguments, or if the method handle types do not match in their return types and their corresponding trailing parameters

Since:

9

See Also:

`catchException(MethodHandle, Class, MethodHandle)`

tableSwitch

```
public static MethodHandle tableSwitch(MethodHandle fallback,
                                       MethodHandle... targets)
```

Creates a table switch method handle, which can be used to switch over a set of target method handles, based on a given target index, called selector.

For a selector value of `n`, where `n` falls in the range `[0, N)`, and where `N` is the number of target method handles, the table switch method handle will invoke the n-th target method handle from the list of target method handles.

For a selector value that does not fall in the range `[0, N)`, the table switch method handle will invoke the given fallback method handle.

All method handles passed to this method must have the same type, with the additional requirement that the leading parameter be of type `int`. The leading parameter represents the selector.

Any trailing parameters present in the type will appear on the returned table switch method handle as well. Any arguments assigned to these parameters will be forwarded, together with the selector value, to the selected method handle when invoking it.

API Note:

Example: The cases each drop the `selector` value they are given, and take an additional `String` argument, which is concatenated (using `String.concat(String)`) to a specific constant label string for each case:

```
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodHandle caseMh = lookup.findVirtual(String.class, "concat",
        MethodType.methodType(String.class, String.class));
caseMh = MethodHandles.dropArguments(caseMh, 0, int.class);

MethodHandle caseDefault = MethodHandles.insertArguments(caseMh, 1, "default: ");
MethodHandle case0 = MethodHandles.insertArguments(caseMh, 1, "case 0: ");
MethodHandle case1 = MethodHandles.insertArguments(caseMh, 1, "case 1: ");

MethodHandle mhSwitch = MethodHandles.tableSwitch(
    caseDefault,
    case0,
    case1
);

assertEquals("default: data", (String) mhSwitch.invokeExact(-1, "data"));
assertEquals("case 0: data", (String) mhSwitch.invokeExact(0, "data"));
assertEquals("case 1: data", (String) mhSwitch.invokeExact(1, "data"));
assertEquals("default: data", (String) mhSwitch.invokeExact(2, "data"));
```

Parameters:

`fallback` - the fallback method handle that is called when the selector is not within the range `[0, N)`.

`targets` - array of target method handles.

Returns:

the table switch method handle.

Throws:

`NullPointerException` - if `fallback`, the `targets` array, or any any of the elements of the `targets` array are `null`.

`IllegalArgumentException` - if the `targets` array is empty, if the leading parameter of the fallback handle or any of the target handles is not `int`, or if the types of the fallback handle and all of target handles are not the same.