Module java.base
Package java.lang.invoke

Class MethodHandle

java.lang.Object
    java.lang.invoke.MethodHandle

All Implemented Interfaces:

Constable

```
public abstract sealed class MethodHandle
extends Object
implements Constable
```

A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution.

Method handle contents

Method handles are dynamically and strongly typed according to their parameter and return types. They are not distinguished by the name or the defining class of their underlying methods. A method handle must be invoked using a symbolic type descriptor which matches the method handle's own type descriptor.

Every method handle reports its type descriptor via the `type` accessor. This type descriptor is a `MethodType` object, whose structure is a series of classes, one of which is the return type of the method (or `void.class` if none).

A method handle's type controls the types of invocations it accepts, and the kinds of transformations that apply to it.

A method handle contains a pair of special invoker methods called `invokeExact` and `invoke`. Both invoker methods provide direct access to the method handle's underlying method, constructor, field, or other operation, as modified by transformations of arguments and return values. Both invokers accept calls which exactly match the method handle's own type. The plain, inexact invoker also accepts a range of other call types.

Method handles are immutable and have no visible state. Of course, they can be bound to underlying methods or data which exhibit state. With respect to the Java Memory Model, any method handle will behave as if all of its (internal) fields are final variables. This means that any method handle made visible to the application will always be fully formed. This is true even if the method handle is published through a shared variable in a data race.

Method handles cannot be subclassed by the user. Implementations may (or may not) create internal subclasses of `MethodHandle` which may be visible via the `Object.getClass` operation. The programmer should not draw conclusions about a method handle from its specific class, as the method handle class hierarchy (if any) may change from time to time or across implementations from different vendors.

Method handle compilation

A Java method call expression naming `invokeExact` or `invoke` can invoke a method handle from Java source code. From the viewpoint of source code, these methods can take any arguments and their result can be cast to any return type. Formally this is accomplished by giving the invoker methods `Object` return types and variable arity `Object` arguments, but they have an additional quality called *signature polymorphism* which connects this freedom of invocation directly to the JVM execution stack.

As is usual with virtual methods, source-level calls to `invokeExact` and `invoke` compile to an `invokevirtual` instruction. More unusually, the compiler must record the actual argument types, and may not perform method invocation conversions on the arguments. Instead, it must generate instructions that push them on the stack according to their own unconverted types. The method handle object itself is pushed on the stack before the arguments. The compiler then generates an `invokevirtual` instruction that invokes the method handle with a symbolic type descriptor which describes the argument and return types.

To issue a complete symbolic type descriptor, the compiler must also determine the return type. This is based on a cast on the method invocation expression, if there is one, or else `Object` if the invocation is an expression, or else `void` if the invocation is a statement. The cast may be to a primitive type (but not `void`).

As a corner case, an uncasted `null` argument is given a symbolic type descriptor of `java.lang.Void`. The ambiguity with the type `Void` is harmless, since there are no references of type `Void` except the null reference.

Method handle invocation

The first time an `invokevirtual` instruction is executed it is linked by symbolically resolving the names in the instruction and verifying that the method call is statically legal. This also holds for calls to `invokeExact` and `invoke`. In this case, the symbolic type descriptor emitted by the compiler is checked for correct syntax, and names it contains are resolved. Thus, an `invokevirtual` instruction which invokes a method handle will always link, as long as the symbolic type descriptor is syntactically well-formed and the types exist.

When the `invokevirtual` is executed after linking, the receiving method handle's type is first checked by the JVM to ensure that it matches the symbolic type descriptor. If the type match fails, it means that the method which the caller is invoking is not present on the individual method handle being invoked.

In the case of `invokeExact`, the type descriptor of the invocation (after resolving symbolic type names) must exactly match the method type of the receiving method handle. In the case of plain, inexact `invoke`, the resolved type descriptor must be a valid argument to the receiver's `asType` method. Thus, plain `invoke` is more permissive than `invokeExact`.

After type matching, a call to `invokeExact` directly and immediately invoke the method handle's underlying method (or other behavior, as the case may be).

A call to plain `invoke` works the same as a call to `invokeExact`, if the symbolic type descriptor specified by the caller exactly matches the method handle's own type. If there is a type mismatch, `invoke` attempts to adjust the type of the receiving method handle, as if by a call to `asType`, to obtain an exactly invokable method handle `M2`. This allows a more powerful negotiation of method type between caller and callee.

(*Note:* The adjusted method handle `M2` is not directly observable, and implementations are therefore not required to materialize it.)

Invocation checking

In typical programs, method handle type matching will usually succeed. But if a match fails, the JVM will throw a `WrongMethodTypeException`, either directly (in the case of `invokeExact`) or indirectly as if by a failed call to `asType` (in the case of `invoke`).

Thus, a method type mismatch which might show up as a linkage error in a statically typed program can show up as a dynamic `WrongMethodTypeException` in a program which uses method handles.

Because method types contain "live" `Class` objects, method type matching takes into account both type names and class loaders. Thus, even if a method handle `M` is created in one class loader `L1` and used in another `L2`, method handle calls are type-safe, because the caller's symbolic type descriptor, as resolved in `L2`, is matched against the original callee method's symbolic type descriptor, as resolved in `L1`. The resolution in `L1` happens when `M` is created and its type is assigned, while the resolution in `L2` happens when the `invokevirtual` instruction is linked.

Apart from type descriptor checks, a method handle's capability to call its underlying method is unrestricted. If a method handle is formed on a non-public method by a class that has access to that method, the resulting handle can be used in any place by any caller who receives a reference to it.

Unlike with the Core Reflection API, where access is checked every time a reflective method is invoked, method handle access checking is performed when the method handle is created. In the case of `ldc` (see below), access checking is performed as part of linking the constant pool entry underlying the constant method handle.

Thus, handles to non-public methods, or to methods in non-public classes, should generally be kept secret. They should not be passed to untrusted code unless their use from the untrusted code would be harmless.

Method handle creation

Java code can create a method handle that directly accesses any method, constructor, or field that is accessible to that code. This is done via a reflective, capability-based API called `MethodHandles.Lookup`. For example, a static method handle can be obtained from `Lookup.findStatic`. There are also conversion methods from Core Reflection API objects, such as `Lookup.unreflect`.

Like classes and strings, method handles that correspond to accessible fields, methods, and constructors can also be represented directly in a class file's constant pool as constants to be loaded by `ldc` bytecodes. A new type of constant pool entry, `CONSTANT_MethodHandle`, refers directly to an associated `CONSTANT_Methodref`, `CONSTANT_InterfaceMethodref`, or `CONSTANT_Fieldref` constant pool entry. (For full details on method handle constants, see sections 4.4.8 and 5.4.3.5 of the Java Virtual Machine Specification.)

Method handles produced by lookups or constant loads from methods or constructors with the variable arity modifier bit (`0x0080`) have a corresponding variable arity, as if they were defined with the help of `asVarargsCollector` or `withVarargs`.

A method reference may refer either to a static or non-static method. In the non-static case, the method handle type includes an explicit receiver argument, prepended before any other arguments. In the method handle's type, the initial receiver argument is typed according to the class under which the method was initially requested. (E.g., if a non-static method handle is obtained via `ldc`, the type of the receiver is the class named in the constant pool entry.)

Method handle constants are subject to the same link-time access checks their corresponding bytecode instructions, and the `ldc` instruction will throw corresponding linkage errors if the bytecode behaviors would throw such errors.

As a corollary of this, access to protected members is restricted to receivers only of the accessing class, or one of its subclasses, and the accessing class must in turn be a subclass (or package sibling) of the protected member's defining class. If a method reference refers to a protected non-static method or field of a class outside the current package, the receiver argument will be narrowed to the type of the accessing class.

When a method handle to a virtual method is invoked, the method is always looked up in the receiver (that is, the first argument).

A non-virtual method handle to a specific virtual method implementation can also be created. These do not perform virtual lookup based on receiver type. Such a method handle simulates the effect of an `invokespecial` instruction to the same method. A non-virtual method handle can also be created to simulate the effect of an `invokevirtual` or `invokeinterface` instruction on a private method (as applicable).

Usage examples

Here are some examples of usage:

```
Object x, y; String s; int i;
MethodType mt; MethodHandle mh;
MethodHandles.Lookup lookup = MethodHandles.lookup();
// mt is (char,char)String
mt = MethodType.methodType(String.class, char.class, char.class);
mh = lookup.findVirtual(String.class, "replace", mt);
s = (String) mh.invokeExact("daddy",'d','n');
// invokeExact(Ljava/lang/String;CC)Ljava/lang/String;
assertEquals(s, "nanny");
// weakly typed invocation (using MHs.invoke)
s = (String) mh.invokeWithArguments("sappy", 'p', 'v');
assertEquals(s, "savvy");
// mt is (Object[])List
mt = MethodType.methodType(java.util.List.class, Object[].class);
mh = lookup.findStatic(java.util.Arrays.class, "asList", mt);
assert(mh.isVarargsCollector());
x = mh.invoke("one", "two");
// invoke(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/Object;
assertEquals(x, java.util.Arrays.asList("one","two"));
// mt is (Object,Object,Object)Object
mt = MethodType.genericMethodType(3);
mh = mh.asType(mt);
x = mh.invokeExact((Object)1, (Object)2, (Object)3);
// invokeExact(Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
assertEquals(x, java.util.Arrays.asList(1,2,3));
// mt is ()int
mt = MethodType.methodType(int.class);
mh = lookup.findVirtual(java.util.List.class, "size", mt);
i = (int) mh.invokeExact(java.util.Arrays.asList(1,2,3));
// invokeExact(Ljava/util/List;)I
assert(i == 3);
mt = MethodType.methodType(void.class, String.class);
mh = lookup.findVirtual(java.io.PrintStream.class, "println", mt);
mh.invokeExact(System.out, "Hello, world.");
// invokeExact(Ljava/io/PrintStream;Ljava/lang/String;)V
```

Each of the above calls to `invokeExact` or plain `invoke` generates a single invokevirtual instruction with the symbolic type descriptor indicated in the following comment. In these examples, the helper method `assertEquals` is assumed to be a method which calls `Objects.equals` on its arguments, and asserts that the result is true.

Exceptions

The methods `invokeExact` and `invoke` are declared to throw `Throwable`, which is to say that there is no static restriction on what a method handle can throw. Since the JVM does not distinguish between checked and unchecked exceptions (other than by their class, of course), there is no particular effect on bytecode shape from ascribing checked exceptions to method handle invocations. But in Java source code, methods which perform method handle calls must either explicitly throw `Throwable`, or else must catch all throwables locally, rethrowing only those which are legal in the context, and wrapping ones which are illegal.

Signature polymorphism

The unusual compilation and linkage behavior of `invokeExact` and plain `invoke` is referenced by the term *signature polymorphism*. As defined in the Java Language Specification, a signature polymorphic method is one which can operate with any of a wide range of call signatures and return types.

In source code, a call to a signature polymorphic method will compile, regardless of the requested symbolic type descriptor. As usual, the Java compiler emits an `invokevirtual` instruction with the given symbolic type descriptor against the named method. The unusual part is that the symbolic type descriptor is derived from the actual argument and return types, not from the method declaration.

When the JVM processes bytecode containing signature polymorphic calls, it will successfully link any such call, regardless of its symbolic type descriptor. (In order to retain type safety, the JVM will guard such calls with suitable dynamic type checks, as described elsewhere.)

Bytecode generators, including the compiler back end, are required to emit untransformed symbolic type descriptors for these methods. Tools which determine symbolic linkage are required to accept such untransformed descriptors, without reporting linkage errors.

Interoperation between method handles and the Core Reflection API

Using factory methods in the `Lookup` API, any class member represented by a Core Reflection API object can be converted to a behaviorally equivalent method handle. For example, a reflective `Method` can be converted to a method handle using `Lookup.unreflect`. The resulting method handles generally provide more direct and efficient access to the underlying class members.

As a special case, when the Core Reflection API is used to view the signature polymorphic methods `invokeExact` or plain `invoke` in this class, they appear as ordinary non-polymorphic methods. Their reflective appearance, as viewed by `Class.getDeclaredMethod`, is unaffected by their special status in this API. For example, `Method.getModifiers` will report exactly those modifier bits required for any similarly declared method, including in this case `native` and `varargs` bits.

As with any reflected method, these methods (when reflected) may be invoked via `java.lang.reflect.Method.invoke`. However, such reflective calls do not result in method handle invocations. Such a call, if passed the required argument (a single one, of type `Object[]`), will ignore the argument and will throw an `UnsupportedOperationException`.

Since `invokevirtual` instructions can natively invoke method handles under any symbolic type descriptor, this reflective view conflicts with the normal presentation of these methods via bytecodes. Thus, these two native methods, when reflectively viewed by `Class.getDeclaredMethod`, may be regarded as placeholders only.

In order to obtain an invoker method for a particular type descriptor, use `MethodHandles.exactInvoker`, or `MethodHandles.invoker`. The `Lookup.findVirtual` API is also able to return a method handle to call `invokeExact` or plain `invoke`, for any specified type descriptor .

Interoperation between method handles and Java generics

A method handle can be obtained on a method, constructor, or field which is declared with Java generic types. As with the Core Reflection API, the type of the method handle will be constructed from the erasure of the source-level type. When a method handle is invoked, the types of its arguments or the return value cast type may be generic types or type instances. If this occurs, the compiler will replace those types by their erasures when it constructs the symbolic type descriptor for the `invokevirtual` instruction.

Method handles do not represent their function-like types in terms of Java parameterized (generic) types, because there are three mismatches between function-like types and parameterized Java types.

- Method types range over all possible arities, from no arguments to up to the maximum number of allowed arguments. Generics are not variadic, and so cannot represent this.
- Method types can specify arguments of primitive types, which Java generic types cannot range over.
- Higher order functions over method handles (combinators) are often generic across a wide range of function types, including those of multiple arities. It is impossible to represent such genericity with a Java type parameter.

Arity limits

The JVM imposes on all methods and constructors of any kind an absolute limit of 255 stacked arguments. This limit can appear more restrictive in certain cases:

- A `long` or `double` argument counts (for purposes of arity limits) as two argument slots.
- A non-static method consumes an extra argument for the object on which the method is called.
- A constructor consumes an extra argument for the object which is being constructed.
- Since a method handle's `invoke` method (or other signature-polymorphic method) is non-virtual, it consumes an extra argument for the method handle itself, in addition to any non-virtual receiver object.

These limits imply that certain method handles cannot be created, solely because of the JVM limit on stacked arguments. For example, if a static JVM method accepts exactly 255 arguments, a method handle cannot be created for it. Attempts to create method handles with impossible method types lead to an `IllegalArgumentException`. In particular, a method handle's type must not have an arity of the exact maximum 255.

Since:

1.7

See Also:

`MethodType`, `MethodHandles`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
MethodHandle	**asCollector**(int collectArgPos, **Class**<?> arrayType, int arrayLength)	Makes an *array-collecting* method handle, which accepts a given number of positional arguments starting at a given position, and collects them into an array argument.
MethodHandle	**asCollector**(**Class**<?> arrayType, int arrayLength)	Makes an *array-collecting* method handle, which accepts a given number of trailing positional arguments and collects them into an array argument.
MethodHandle	**asFixedArity**()	Makes a *fixed arity* method handle which is otherwise equivalent to the current method handle.
MethodHandle	**asSpreader**(int spreadArgPos, **Class**<?> arrayType, int arrayLength)	Makes an *array-spreading* method handle, which accepts an array argument at a given position and spreads its elements as positional arguments in place of the array.
MethodHandle	**asSpreader**(**Class**<?> arrayType, int arrayLength)	Makes an *array-spreading* method handle, which accepts a trailing array argument and spreads its elements as positional arguments.
final **MethodHandle**	**asType**(**MethodType** newType)	Produces an adapter method handle which adapts the type of the current method handle to a new type.
MethodHandle	**asVarargsCollector**(**Class**<?> arrayType)	Makes a *variable arity* adapter which is able to accept any number of trailing positional arguments and collect them into an array argument.
MethodHandle	**bindTo**(**Object** x)	Binds a value x to the first argument of a method handle, without invoking it.
Optional<**MethodHandleDesc**>	**describeConstable**()	Return a nominal descriptor for this instance, if one can be constructed, or an empty `Optional` if one cannot be.
final **Object**	**invoke**(**Object**... args)	Invokes the method handle, allowing any caller type descriptor, and optionally performing conversions on arguments and return values.
final **Object**	**invokeExact**(**Object**... args)	Invokes the method handle, allowing any caller type descriptor, but requiring an exact type match.
Object	**invokeWithArguments**(**Object**... arguments)	Performs a variable arity invocation, passing the arguments in the given array to the method handle, as if via an inexact `invoke` from a call site which mentions only the type `Object`, and whose actual argument count is the length of the argument array.
Object	**invokeWithArguments**(**List**<?> arguments)	Performs a variable arity invocation, passing the arguments in the given list to the method handle, as if via an inexact `invoke` from a call site which mentions only the type `Object`, and whose actual argument count is the length of the argument list.
boolean	**isVarargsCollector**()	Determines if this method handle supports variable arity calls.
String	**toString**()	Returns a string representation of the method handle, starting with the string `"MethodHandle"` and ending with the string representation of the method handle's type.
MethodType	**type**()	Reports the type of this method handle.
MethodHandle	**withVarargs**(boolean makeVarargs)	Adapts this method handle to be variable arity if the boolean flag is true, else fixed arity.

Methods declared in class java.lang.Object

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

Method Details

type

```
public MethodType type()
```

Reports the type of this method handle. Every invocation of this method handle via `invokeExact` must exactly match this type.

Returns:

the method handle type

invokeExact

```
public final Object invokeExact(Object... args)
                         throws Throwable
```

Invokes the method handle, allowing any caller type descriptor, but requiring an exact type match. The symbolic type descriptor at the call site of `invokeExact` must exactly match this method handle's `type`. No conversions are allowed on arguments or return values.

When this method is observed via the Core Reflection API, it will appear as a single native method, taking an object array and returning an object. If this native method is invoked directly via `java.lang.reflect.Method.invoke`, via JNI, or indirectly via `Lookup.unreflect`, it will throw an `UnsupportedOperationException`.

Parameters:

`args` - the signature-polymorphic parameter list, statically represented using varargs

Returns:

the signature-polymorphic result, statically represented using `Object`

Throws:

`WrongMethodTypeException` - if the target's type is not identical with the caller's symbolic type descriptor

`Throwable` - anything thrown by the underlying method propagates unchanged through the method handle call

invoke

```
public final Object invoke(Object... args)
                   throws Throwable
```

Invokes the method handle, allowing any caller type descriptor, and optionally performing conversions on arguments and return values.

If the call site's symbolic type descriptor exactly matches this method handle's `type`, the call proceeds as if by `invokeExact`.

Otherwise, the call proceeds as if this method handle were first adjusted by calling `asType` to adjust this method handle to the required type, and then the call proceeds as if by `invokeExact` on the adjusted method handle.

There is no guarantee that the `asType` call is actually made. If the JVM can predict the results of making the call, it may perform adaptations directly on the caller's arguments, and call the target method handle according to its own exact type.

The resolved type descriptor at the call site of `invoke` must be a valid argument to the receivers `asType` method. In particular, the caller must specify the same argument arity as the callee's type, if the callee is not a variable arity collector.

When this method is observed via the Core Reflection API, it will appear as a single native method, taking an object array and returning an object. If this native method is invoked directly via `java.lang.reflect.Method.invoke`, via JNI, or indirectly via `Lookup.unreflect`, it will throw an `UnsupportedOperationException`.

Parameters:

`args` - the signature-polymorphic parameter list, statically represented using varargs

Returns:

the signature-polymorphic result, statically represented using `Object`

Throws:

`WrongMethodTypeException` - if the target's type cannot be adjusted to the caller's symbolic type descriptor

`ClassCastException` - if the target's type can be adjusted to the caller, but a reference cast fails

`Throwable` - anything thrown by the underlying method propagates unchanged through the method handle call

invokeWithArguments

```
public Object invokeWithArguments(Object... arguments)
                           throws Throwable
```

Performs a variable arity invocation, passing the arguments in the given array to the method handle, as if via an inexact `invoke` from a call site which mentions only the type `Object`, and whose actual argument count is the length of the argument array.

Specifically, execution proceeds as if by the following steps, although the methods are not guaranteed to be called if the JVM can predict their effects.

- Determine the length of the argument array as `N`. For a null reference, `N=0`.
- Collect the `N` elements of the array as a logical argument list, each argument statically typed as an `Object`.
- Determine, as `M`, the parameter count of the type of this method handle.
- Determine the general type `TN` of `N` arguments or `M` arguments, if smaller than `N`, as `TN=MethodType.genericMethodType(Math.min(N, M))`.
- If `N` is greater than `M`, perform the following checks and actions to shorten the logical argument list:
  - Check that this method handle has variable arity with a trailing parameter of some array type `A[]`. If not, fail with a `WrongMethodTypeException`.
  - Collect the trailing elements (there are `N-M+1` of them) from the logical argument list into a single array of type `A[]`, using `asType` conversions to convert each trailing argument to type `A`.
  - If any of these conversions proves impossible, fail with either a `ClassCastException` if any trailing element cannot be cast to `A` or a `NullPointerException` if any trailing element is `null` and `A` is not a reference type.
  - Replace the logical arguments gathered into the array of type `A[]` with the array itself, thus shortening the argument list to length `M`. This final argument retains the static type `A[]`.
  - Adjust the type `TN` by changing the `N`th parameter type from `Object` to `A[]`.
- Force the original target method handle `MH0` to the required type, as `MH1 = MH0.asType(TN)`.
- Spread the argument list into `N` separate arguments `A0, ...`.
- Invoke the type-adjusted method handle on the unpacked arguments: MH1.invokeExact(A0, ...).
- Take the return value as an `Object` reference.

If the target method handle has variable arity, and the argument list is longer than that arity, the excess arguments, starting at the position of the trailing array argument, will be gathered (if possible, as if by `asType` conversions) into an array of the appropriate type, and invocation will proceed on the shortened argument list. In this way, *jumbo argument lists* which would spread into more than 254 slots can still be processed uniformly.

Unlike the `generic` invocation mode, which can "recycle" an array argument, passing it directly to the target method, this invocation mode *always* creates a new array parameter, even if the original array passed to `invokeWithArguments` would have been acceptable as a direct argument to the target method. Even if the number `M` of actual arguments is the arity `N`, and the last argument is dynamically a suitable array of type `A[]`, it will still be boxed into a new one-element array, since the call site statically types the argument as `Object`, not an array type. This is not a special rule for this method, but rather a regular effect of the rules for variable-arity invocation.

Because of the action of the `asType` step, the following argument conversions are applied as necessary:

- reference casting
- unboxing
- widening primitive conversions
- variable arity conversion

The result returned by the call is boxed if it is a primitive, or forced to null if the return type is void.

Unlike the signature polymorphic methods `invokeExact` and `invoke`, `invokeWithArguments` can be accessed normally via the Core Reflection API and JNI. It can therefore be used as a bridge between native or reflective code and method handles.

API Note:

This call is approximately equivalent to the following code:

```
// for jumbo argument lists, adapt varargs explicitly:
int N = (arguments == null? 0: arguments.length);
```

```
int M = this.type.parameterCount();
int MAX_SAFE = 127;  // 127 longs require 254 slots, which is OK
if (N > MAX_SAFE && N > M && this.isVarargsCollector()) {
  Class<?> arrayType = this.type().lastParameterType();
  Class<?> elemType = arrayType.getComponentType();
  if (elemType != null) {
    Object args2 = Array.newInstance(elemType, M);
    MethodHandle arraySetter = MethodHandles.arrayElementSetter(arrayType);
    for (int i = 0; i < M; i++) {
      arraySetter.invoke(args2, i, arguments[M-1 + i]);
    }
    arguments = Arrays.copyOf(arguments, M);
    arguments[M-1] = args2;
    return this.asFixedArity().invokeWithArguments(arguments);
  }
} // done with explicit varargs processing

// Handle fixed arity and non-jumbo variable arity invocation.
MethodHandle invoker = MethodHandles.spreadInvoker(this.type(), 0);
Object result = invoker.invokeExact(this, arguments);
```

Parameters:

`arguments` - the arguments to pass to the target

Returns:

the result returned by the target

Throws:

`ClassCastException` - if an argument cannot be converted by reference casting

`WrongMethodTypeException` - if the target's type cannot be adjusted to take the given number of `Object` arguments

`Throwable` - anything thrown by the target method invocation

See Also:

`MethodHandles.spreadInvoker(java.lang.invoke.MethodType, int)`

invokeWithArguments

```
public Object invokeWithArguments(List<?> arguments)
                           throws Throwable
```

Performs a variable arity invocation, passing the arguments in the given list to the method handle, as if via an inexact `invoke` from a call site which mentions only the type `Object`, and whose actual argument count is the length of the argument list.

This method is also equivalent to the following code:

```
invokeWithArguments(arguments.toArray())
```

Jumbo-sized lists are acceptable if this method handle has variable arity. See `invokeWithArguments(Object[])` for details.

Parameters:

`arguments` - the arguments to pass to the target

Returns:

the result returned by the target

Throws:

`NullPointerException` - if `arguments` is a null reference

`ClassCastException` - if an argument cannot be converted by reference casting

`WrongMethodTypeException` - if the target's type cannot be adjusted to take the given number of `Object` arguments

`Throwable` - anything thrown by the target method invocation

asType

```
public final MethodHandle asType(MethodType newType)
```

Produces an adapter method handle which adapts the type of the current method handle to a new type. 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 `this`.

The new method handle, when invoked, will perform the following steps:

- Convert the incoming argument list to match the original method handle's argument list.
- Invoke the original method handle on the converted argument list.
- Convert any result returned by the original method handle to the return type of new method handle.

This method provides the crucial behavioral difference between `invokeExact` and plain, inexact `invoke`. The two methods perform the same steps when the caller's type descriptor exactly matches the callee's, but when the types differ, plain `invoke` also calls `asType` (or some internal equivalent) in order to match up the caller's and callee's types.

If the current method is a variable arity method handle argument list conversion may involve the conversion and collection of several arguments into an array, as described elsewhere. In every other case, all conversions are applied *pairwise*, which means that each argument or return value is converted to exactly one argument or return value (or no return value). The applied conversions are defined by consulting the corresponding component types of the old and new method handle types.

Let *T0* and *T1* be corresponding new and old parameter types, or old and new return types. Specifically, for some valid index `i`, let *T0*=`newType.parameterType(i)` and *T1*=`this.type().parameterType(i)`. Or else, going the other way for return values, let *T0*=`this.type().returnType()` and *T1*=`newType.returnType()`. If the types are the same, the new method handle makes no change to the corresponding argument or return value (if any). Otherwise, one of the following conversions is applied if possible:

- If *T0* and *T1* are references, then a cast to *T1* is applied. (The types do not need to be related in any particular way. This is because a dynamic value of null can convert to any reference type.)
- If *T0* and *T1* are primitives, then a Java method invocation conversion (JLS 5.3) is applied, if one exists. (Specifically, *T0* must convert to *T1* by a widening primitive conversion.)
- If *T0* is a primitive and *T1* a reference, a Java casting conversion (JLS 5.5) is applied if one exists. (Specifically, the value is boxed from *T0* to its wrapper class, which is then widened as needed to *T1*.)
- If *T0* is a reference and *T1* a primitive, an unboxing conversion will be applied at runtime, possibly followed by a Java method invocation conversion (JLS 5.3) on the primitive value. (These are the primitive widening conversions.) *T0* must be a wrapper class or a supertype of one. (In the case where *T0* is Object, these are the conversions allowed by `java.lang.reflect.Method.invoke`.) The unboxing conversion must have a possibility of success, which means that if *T0* is not itself a wrapper class, there must exist at least one wrapper class *TW* which is a subtype of *T0* and whose unboxed primitive value can be widened to *T1*.
- If the return type *T1* is marked as void, any returned value is discarded
- If the return type *T0* is void and *T1* a reference, a null value is introduced.
- If the return type *T0* is void and *T1* a primitive, a zero value is introduced.

(*Note:* Both *T0* and *T1* may be regarded as static types, because neither corresponds specifically to the *dynamic type* of any actual argument or return value.)

The method handle conversion cannot be made if any one of the required pairwise conversions cannot be made.

At runtime, the conversions applied to reference arguments or return values may require additional runtime checks which can fail. An unboxing operation may fail because the original reference is null, causing a `NullPointerException`. An unboxing operation or a reference cast may also fail on a reference to an object of the wrong type, causing a `ClassCastException`. Although an unboxing operation may accept several kinds of wrappers, if none are available, a `ClassCastException` will be thrown.

Parameters:

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

Returns:

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

Throws:

`NullPointerException` - if `newType` is a null reference

`WrongMethodTypeException` - if the conversion cannot be made

See Also:

`MethodHandles.explicitCastArguments(java.lang.invoke.MethodHandle, java.lang.invoke.MethodType)`

asSpreader

```
public MethodHandle asSpreader(Class<?> arrayType,
                               int arrayLength)
```

Makes an *array-spreading* method handle, which accepts a trailing array argument and spreads its elements as positional arguments. The new method handle adapts, as its *target*, the current method handle. The type of the adapter will be the same as the type of the target, except that the final `arrayLength` parameters of the target's type are replaced by a single array parameter of type `arrayType`.

If the array element type differs from any of the corresponding argument types on the original target, the original target is adapted to take the array elements directly, as if by a call to `asType`.

When called, the adapter replaces a trailing array argument by the array's elements, each as its own argument to the target. (The order of the arguments is preserved.) They are converted pairwise by casting and/or unboxing to the types of the trailing parameters of the target. Finally the target is called. What the target eventually returns is returned unchanged by the adapter.

Before calling the target, the adapter verifies that the array contains exactly enough elements to provide a correct argument count to the target method handle. (The array may also be null when zero elements are required.)

When the adapter is called, the length of the supplied `array` argument is queried as if by `array.length` or `arraylength` bytecode. If the adapter accepts a zero-length trailing array argument, the supplied `array` argument can either be a zero-length array or `null`; otherwise, the adapter will throw a `NullPointerException` if the array is `null` and throw an `IllegalArgumentException` if the array does not have the correct number of elements.

Here are some simple examples of array-spreading method handles:

```
MethodHandle equals = publicLookup()
  .findVirtual(String.class, "equals", methodType(boolean.class, Object.class));
assert( (boolean) equals.invokeExact("me", (Object)"me"));
assert(!(boolean) equals.invokeExact("me", (Object)"thee"));
// spread both arguments from a 2-array:
MethodHandle eq2 = equals.asSpreader(Object[].class, 2);
assert( (boolean) eq2.invokeExact(new Object[]{ "me", "me" }));
assert(!(boolean) eq2.invokeExact(new Object[]{ "me", "thee" }));
// try to spread from anything but a 2-array:
for (int n = 0; n <= 10; n++) {
  Object[] badArityArgs = (n == 2 ? new Object[0] : new Object[n]);
  try { assert((boolean) eq2.invokeExact(badArityArgs) && false); }
  catch (IllegalArgumentException ex) { } // OK
}
// spread both arguments from a String array:
MethodHandle eq2s = equals.asSpreader(String[].class, 2);
assert( (boolean) eq2s.invokeExact(new String[]{ "me", "me" }));
assert(!(boolean) eq2s.invokeExact(new String[]{ "me", "thee" }));
// spread second arguments from a 1-array:
MethodHandle eq1 = equals.asSpreader(Object[].class, 1);
assert( (boolean) eq1.invokeExact("me", new Object[]{ "me" }));
assert(!(boolean) eq1.invokeExact("me", new Object[]{ "thee" }));
// spread no arguments from a 0-array or null:
MethodHandle eq0 = equals.asSpreader(Object[].class, 0);
assert( (boolean) eq0.invokeExact("me", (Object)"me", new Object[0]));
assert(!(boolean) eq0.invokeExact("me", (Object)"thee", (Object[])null));
// asSpreader and asCollector are approximate inverses:
for (int n = 0; n <= 2; n++) {
    for (Class<?> a : new Class<?>[]{Object[].class, String[].class, CharSequence[].class}) {
        MethodHandle equals2 = equals.asSpreader(a, n).asCollector(a, n);
        assert( (boolean) equals2.invokeWithArguments("me", "me"));
        assert(!(boolean) equals2.invokeWithArguments("me", "thee"));
    }
}
MethodHandle caToString = publicLookup()
  .findStatic(Arrays.class, "toString", methodType(String.class, char[].class));
assertEquals("[A, B, C]", (String) caToString.invokeExact("ABC".toCharArray()));
MethodHandle caString3 = caToString.asCollector(char[].class, 3);
assertEquals("[A, B, C]", (String) caString3.invokeExact('A', 'B', 'C'));
MethodHandle caToString2 = caString3.asSpreader(char[].class, 2);
assertEquals("[A, B, C]", (String) caToString2.invokeExact('A', "BC".toCharArray()));
```

Parameters:

`arrayType` - usually `Object[]`, the type of the array argument from which to extract the spread arguments

`arrayLength` - the number of arguments to spread from an incoming array argument

Returns:

a new method handle which spreads its final array argument, before calling the original method handle

Throws:

`NullPointerException` - if `arrayType` is a null reference

`IllegalArgumentException` - if `arrayType` is not an array type, or if target does not have at least `arrayLength` parameter types, or if `arrayLength` is negative, or if the resulting method handle's type would have too many parameters

`WrongMethodTypeException` - if the implied `asType` call fails

See Also:

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

asSpreader

```
public MethodHandle asSpreader(int spreadArgPos,
                               Class<?> arrayType,
                               int arrayLength)
```

Makes an *array-spreading* method handle, which accepts an array argument at a given position and spreads its elements as positional arguments in place of the array. The new method handle adapts, as its *target*, the current method handle. The type of the adapter will be the same as the type of the target, except that the `arrayLength` parameters of the target's type, starting at the zero-based position `spreadArgPos`, are replaced by a single array parameter of type `arrayType`.

This method behaves very much like `asSpreader(Class, int)`, but accepts an additional `spreadArgPos` argument to indicate at which position in the parameter list the spreading should take place.

API Note:

Example:

```
MethodHandle compare = LOOKUP.findStatic(Objects.class, "compare", methodType(int.class, Object.class, Object.class, Comparator.class));
MethodHandle compare2FromArray = compare.asSpreader(0, Object[].class, 2);
Object[] ints = new Object[]{3, 9, 7, 7};
Comparator<Integer> cmp = (a, b) -> a - b;
assertTrue((int) compare2FromArray.invoke(Arrays.copyOfRange(ints, 0, 2), cmp) < 0);
assertTrue((int) compare2FromArray.invoke(Arrays.copyOfRange(ints, 1, 3), cmp) > 0);
assertTrue((int) compare2FromArray.invoke(Arrays.copyOfRange(ints, 2, 4), cmp) == 0);
```

Parameters:

`spreadArgPos` - the position (zero-based index) in the argument list at which spreading should start.

`arrayType` - usually `Object[]`, the type of the array argument from which to extract the spread arguments

`arrayLength` - the number of arguments to spread from an incoming array argument

Returns:

a new method handle which spreads an array argument at a given position, before calling the original method handle

Throws:

`NullPointerException` - if `arrayType` is a null reference

`IllegalArgumentException` - if `arrayType` is not an array type, or if target does not have at least `arrayLength` parameter types, or if `arrayLength` is negative, or if `spreadArgPos` has an illegal value (negative, or together with arrayLength exceeding the number of arguments), or if the resulting method handle's type would have too many parameters

`WrongMethodTypeException` - if the implied `asType` call fails

Since:

9

See Also:

`asSpreader(Class, int)`

withVarargs

```
public MethodHandle withVarargs(boolean makeVarargs)
```

Adapts this method handle to be variable arity if the boolean flag is true, else fixed arity. If the method handle is already of the proper arity mode, it is returned unchanged.

API Note:

This method is sometimes useful when adapting a method handle that may be variable arity, to ensure that the resulting adapter is also variable arity if and only if the original handle was. For example, this code changes the first argument of a handle `mh` to `int` without disturbing its variable arity property:
`mh.asType(mh.type().changeParameterType(0,int.class)) .withVarargs(mh.isVarargsCollector())`

This call is approximately equivalent to the following code:

```
if (makeVarargs == isVarargsCollector())
  return this;
else if (makeVarargs)
  return asVarargsCollector(type().lastParameterType());
else
  return asFixedArity();
```

Parameters:

`makeVarargs` - true if the return method handle should have variable arity behavior

Returns:

a method handle of the same type, with possibly adjusted variable arity behavior

Throws:

`IllegalArgumentException` - if `makeVarargs` is true and this method handle does not have a trailing array parameter

Since:

9

See Also:

`asVarargsCollector(java.lang.Class<?>)`,
`asFixedArity()`

asCollector

```
public MethodHandle asCollector(Class<?> arrayType,
                                int arrayLength)
```

Makes an *array-collecting* method handle, which accepts a given number of trailing positional arguments and collects them into an array argument. The new method handle adapts, as its *target*, the current method handle. The type of the adapter will be the same as the type of the target, except that a single trailing parameter (usually of type `arrayType`) is replaced by `arrayLength` parameters whose type is element type of `arrayType`.

If the array type differs from the final argument type on the original target, the original target is adapted to take the array type directly, as if by a call to `asType`.

When called, the adapter replaces its trailing `arrayLength` arguments by a single new array of type `arrayType`, whose elements comprise (in order) the replaced arguments. Finally the target is called. What the target eventually returns is returned unchanged by the adapter.

(The array may also be a shared constant when `arrayLength` is zero.)

(*Note:* The `arrayType` is often identical to the last parameter type of the original target. It is an explicit argument for symmetry with `asSpreader`, and also to allow the target to use a simple `Object` as its last parameter type.)

In order to create a collecting adapter which is not restricted to a particular number of collected arguments, use `asVarargsCollector` or `withVarargs` instead.

Here are some examples of array-collecting method handles:

```
MethodHandle deepToString = publicLookup()
  .findStatic(Arrays.class, "deepToString", methodType(String.class, Object[].class));
```

```
assertEquals("[won]",   (String) deepToString.invokeExact(new Object[]{"won"}));
MethodHandle ts1 = deepToString.asCollector(Object[].class, 1);
assertEquals(methodType(String.class, Object.class), ts1.type());
//assertEquals("[won]", (String) ts1.invokeExact(         new Object[]{"won"})); //FAIL
assertEquals("[[won]]", (String) ts1.invokeExact((Object) new Object[]{"won"}));
// arrayType can be a subtype of Object[]
MethodHandle ts2 = deepToString.asCollector(String[].class, 2);
assertEquals(methodType(String.class, String.class, String.class), ts2.type());
assertEquals("[two, too]", (String) ts2.invokeExact("two", "too"));
MethodHandle ts0 = deepToString.asCollector(Object[].class, 0);
assertEquals("[]", (String) ts0.invokeExact());
// collectors can be nested, Lisp-style
MethodHandle ts22 = deepToString.asCollector(Object[].class, 3).asCollector(String[].class, 2);
assertEquals("[A, B, [C, D]]", ((String) ts22.invokeExact((Object)'A', (Object)"B", "C", "D")));
// arrayType can be any primitive array type
MethodHandle bytesToString = publicLookup()
  .findStatic(Arrays.class, "toString", methodType(String.class, byte[].class))
  .asCollector(byte[].class, 3);
assertEquals("[1, 2, 3]", (String) bytesToString.invokeExact((byte)1, (byte)2, (byte)3));
MethodHandle longsToString = publicLookup()
  .findStatic(Arrays.class, "toString", methodType(String.class, long[].class))
  .asCollector(long[].class, 1);
assertEquals("[123]", (String) longsToString.invokeExact((long)123));
```

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

Parameters:

`arrayType` - often `Object[]`, the type of the array argument which will collect the arguments

`arrayLength` - the number of arguments to collect into a new array argument

Returns:

a new method handle which collects some trailing argument into an array, before calling the original method handle

Throws:

`NullPointerException` - if `arrayType` is a null reference

`IllegalArgumentException` - if `arrayType` is not an array type or `arrayType` is not assignable to this method handle's trailing parameter type, or `arrayLength` is not a legal array size, or the resulting method handle's type would have too many parameters

`WrongMethodTypeException` - if the implied `asType` call fails

See Also:

`asSpreader(java.lang.Class<?>, int)`,
`asVarargsCollector(java.lang.Class<?>)`

asCollector

```
public MethodHandle asCollector(int collectArgPos,
                                Class<?> arrayType,
                                int arrayLength)
```

Makes an *array-collecting* method handle, which accepts a given number of positional arguments starting at a given position, and collects them into an array argument. The new method handle adapts, as its *target*, the current method handle. The type of the adapter will be the same as the type of the target, except that the parameter at the position indicated by `collectArgPos` (usually of type `arrayType`) is replaced by `arrayLength` parameters whose type is element type of `arrayType`.

This method behaves very much like `asCollector(Class, int)`, but differs in that its `collectArgPos` argument indicates at which position in the parameter list arguments should be collected. This index is zero-based.

API Note:

Examples:

```
StringWriter swr = new StringWriter();
MethodHandle swWrite = LOOKUP.findVirtual(StringWriter.class, "write", methodType(void.class, char[].class, int.class, int.class)).bindTo(swr);
MethodHandle swWrite4 = swWrite.asCollector(0, char[].class, 4);
swWrite4.invoke('A', 'B', 'C', 'D', 1, 2);
assertEquals("BC", swr.toString());
swWrite4.invoke('P', 'Q', 'R', 'S', 0, 4);
assertEquals("BCPQRS", swr.toString());
swWrite4.invoke('W', 'X', 'Y', 'Z', 3, 1);
assertEquals("BCPQRSZ", swr.toString());
```

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

Parameters:

`collectArgPos` - the zero-based position in the parameter list at which to start collecting.

`arrayType` - often `Object[]`, the type of the array argument which will collect the arguments

`arrayLength` - the number of arguments to collect into a new array argument

Returns:

a new method handle which collects some arguments into an array, before calling the original method handle

Throws:

`NullPointerException` - if `arrayType` is a null reference

`IllegalArgumentException` - if `arrayType` is not an array type or `arrayType` is not assignable to this method handle's array parameter type, or `arrayLength` is not a legal array size, or `collectArgPos` has an illegal value (negative, or greater than the number of arguments), or the resulting method handle's type would have too many parameters

`WrongMethodTypeException` - if the implied `asType` call fails

Since:

9

See Also:

`asCollector(Class, int)`

asVarargsCollector

```
public MethodHandle asVarargsCollector(Class<?> arrayType)
```

Makes a *variable arity* adapter which is able to accept any number of trailing positional arguments and collect them into an array argument.

The type and behavior of the adapter will be the same as the type and behavior of the target, except that certain `invoke` and `asType` requests can lead to trailing positional arguments being collected into target's trailing parameter. Also, the last parameter type of the adapter will be `arrayType`, even if the target has a different last parameter type.

This transformation may return `this` if the method handle is already of variable arity and its trailing parameter type is identical to `arrayType`.

When called with `invokeExact`, the adapter invokes the target with no argument changes. (*Note:* This behavior is different from a fixed arity collector, since it accepts a whole array of indeterminate length, rather than a fixed number of arguments.)

When called with plain, inexact `invoke`, if the caller type is the same as the adapter, the adapter invokes the target as with `invokeExact`. (This is the normal behavior for `invoke` when types match.)

Otherwise, if the caller and adapter arity are the same, and the trailing parameter type of the caller is a reference type identical to or assignable to the trailing parameter type of the adapter, the arguments and return values are converted pairwise, as if by `asType` on a fixed arity method handle.

Otherwise, the arities differ, or the adapter's trailing parameter type is not assignable from the corresponding caller type. In this case, the adapter replaces all trailing arguments from the original trailing argument position onward, by a new array of type `arrayType`, whose elements comprise (in order) the replaced arguments.

The caller type must provide at least enough arguments, and of the correct type, to satisfy the target's requirement for positional arguments before the trailing array argument. Thus, the caller must supply, at a minimum, `N-1` arguments, where `N` is the arity of the target. Also, there must exist conversions from the incoming arguments to the target's arguments. As with other uses of plain `invoke`, if these basic requirements are not fulfilled, a `WrongMethodTypeException` may be thrown.

In all cases, what the target eventually returns is returned unchanged by the adapter.

In the final case, it is exactly as if the target method handle were temporarily adapted with a fixed arity collector to the arity required by the caller type. (As with `asCollector`, if the array length is zero, a shared constant may be used instead of a new array. If the implied call to `asCollector` would throw an `IllegalArgumentException` or `WrongMethodTypeException`, the call to the variable arity adapter must throw `WrongMethodTypeException`.)

The behavior of `asType` is also specialized for variable arity adapters, to maintain the invariant that plain, inexact `invoke` is always equivalent to an `asType` call to adjust the target type, followed by `invokeExact`. Therefore, a variable arity adapter responds to an `asType` request by building a fixed arity collector, if and only if the adapter and requested type differ either in arity or trailing argument type. The resulting fixed arity collector has its type further adjusted (if necessary) to the requested type by pairwise conversion, as if by another application of `asType`.

When a method handle is obtained by executing an `ldc` instruction of a `CONSTANT_MethodHandle` constant, and the target method is marked as a variable arity method (with the modifier bit `0x0080`), the method handle will accept multiple arities, as if the method handle constant were created by means of a call to `asVarargsCollector`.

In order to create a collecting adapter which collects a predetermined number of arguments, and whose type reflects this predetermined number, use `asCollector` instead.

No method handle transformations produce new method handles with variable arity, unless they are documented as doing so. Therefore, besides `asVarargsCollector` and `withVarargs`, all methods in `MethodHandle` and `MethodHandles` will return a method handle with fixed arity, except in the cases where they are specified to return their original operand (e.g., `asType` of the method handle's own type).

Calling `asVarargsCollector` on a method handle which is already of variable arity will produce a method handle with the same type and behavior. It may (or may not) return the original variable arity method handle.

Here is an example, of a list-making variable arity method handle:

```
MethodHandle deepToString = publicLookup()
  .findStatic(Arrays.class, "deepToString", methodType(String.class, Object[].class));
MethodHandle ts1 = deepToString.asVarargsCollector(Object[].class);
assertEquals("[won]",   (String) ts1.invokeExact(    new Object[]{"won"}));
assertEquals("[won]",   (String) ts1.invoke(         new Object[]{"won"}));
assertEquals("[won]",   (String) ts1.invoke(                      "won" ));
assertEquals("[[won]]", (String) ts1.invoke((Object) new Object[]{"won"}));
// findStatic of Arrays.asList(...) produces a variable arity method handle:
MethodHandle asList = publicLookup()
  .findStatic(Arrays.class, "asList", methodType(List.class, Object[].class));
assertEquals(methodType(List.class, Object[].class), asList.type());
assert(asList.isVarargsCollector());
assertEquals("[]", asList.invoke().toString());
assertEquals("[1]", asList.invoke(1).toString());
assertEquals("[two, too]", asList.invoke("two", "too").toString());
String[] argv = { "three", "thee", "tee" };
assertEquals("[three, thee, tee]", asList.invoke(argv).toString());
assertEquals("[three, thee, tee]", asList.invoke((Object[])argv).toString());
List ls = (List) asList.invoke((Object)argv);
assertEquals(1, ls.size());
assertEquals("[three, thee, tee]", Arrays.toString((Object[])ls.get(0)));
```

Discussion: These rules are designed as a dynamically-typed variation of the Java rules for variable arity methods. In both cases, callers to a variable arity method or method handle can either pass zero or more positional arguments, or else pass pre-collected arrays of any length. Users should be aware of the special role of the final argument, and of the effect of a type match on that final argument, which determines whether or not a single trailing argument is interpreted as a whole array or a single element of an array to be collected. Note that the dynamic type of the trailing argument has no effect on this decision, only a comparison between the symbolic type descriptor of the call site and the type descriptor of the method handle.

Parameters:

`arrayType` - often `Object[]`, the type of the array argument which will collect the arguments

Returns:

a new method handle which can collect any number of trailing arguments into an array, before calling the original method handle

Throws:

`NullPointerException` - if `arrayType` is a null reference

`IllegalArgumentException` - if `arrayType` is not an array type or `arrayType` is not assignable to this method handle's trailing parameter type

See Also:

`asCollector(java.lang.Class<?>, int)`,
`isVarargsCollector()`,
`withVarargs(boolean)`,
`asFixedArity()`

isVarargsCollector

```
public boolean isVarargsCollector()
```

Determines if this method handle supports variable arity calls. Such method handles arise from the following sources:

- a call to asVarargsCollector
- a call to a lookup method which resolves to a variable arity Java method or constructor
- an `ldc` instruction of a `CONSTANT_MethodHandle` which resolves to a variable arity Java method or constructor

Returns:

true if this method handle accepts more than one arity of plain, inexact `invoke` calls

See Also:

`asVarargsCollector(java.lang.Class<?>)`,
`asFixedArity()`

asFixedArity

```
public MethodHandle asFixedArity()
```

Makes a *fixed arity* method handle which is otherwise equivalent to the current method handle.

If the current method handle is not of variable arity, the current method handle is returned. This is true even if the current method handle could not be a valid input to `asVarargsCollector`.

Otherwise, the resulting fixed-arity method handle has the same type and behavior of the current method handle, except that `isVarargsCollector` will be false. The fixed-arity method handle may (or may not) be the previous argument to `asVarargsCollector`.

Here is an example, of a list-making variable arity method handle:

```
MethodHandle asListVar = publicLookup()
  .findStatic(Arrays.class, "asList", methodType(List.class, Object[].class))
  .asVarargsCollector(Object[].class);
MethodHandle asListFix = asListVar.asFixedArity();
assertEquals("[1]", asListVar.invoke(1).toString());
Exception caught = null;
try { asListFix.invoke((Object)1); }
catch (Exception ex) { caught = ex; }
assert(caught instanceof ClassCastException);
assertEquals("[two, too]", asListVar.invoke("two", "too").toString());
try { asListFix.invoke("two", "too"); }
catch (Exception ex) { caught = ex; }
assert(caught instanceof WrongMethodTypeException);
Object[] argv = { "three", "thee", "tee" };
assertEquals("[three, thee, tee]", asListVar.invoke(argv).toString());
assertEquals("[three, thee, tee]", asListFix.invoke(argv).toString());
assertEquals(1, ((List) asListVar.invoke((Object)argv)).size());
assertEquals("[three, thee, tee]", asListFix.invoke((Object)argv).toString());
```

Returns:

a new method handle which accepts only a fixed number of arguments

See Also:

`asVarargsCollector(java.lang.Class<?>)`,
`isVarargsCollector()`,
`withVarargs(boolean)`

bindTo

```
public MethodHandle bindTo(Object x)
```

Binds a value `x` to the first argument of a method handle, without invoking it. The new method handle adapts, as its *target*, the current method handle by binding it to the given argument. The type of the bound handle will be the same as the type of the target, except that a single leading reference parameter will be omitted.

When called, the bound handle inserts the given value `x` as a new leading argument to the target. The other arguments are also passed unchanged. What the target eventually returns is returned unchanged by the bound handle.

The reference `x` must be convertible to the first parameter type of the target.

Note: Because method handles are immutable, the target method handle retains its original type and behavior.

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

Parameters:

`x` - the value to bind to the first argument of the target

Returns:

a new method handle which prepends the given value to the incoming argument list, before calling the original method handle

Throws:

`IllegalArgumentException` - if the target does not have a leading parameter type that is a reference type

`ClassCastException` - if `x` cannot be converted to the leading parameter type of the target

See Also:

`MethodHandles.insertArguments(java.lang.invoke.MethodHandle, int, java.lang.Object...)`

describeConstable

```
public Optional<MethodHandleDesc> describeConstable()
```

Return a nominal descriptor for this instance, if one can be constructed, or an empty `Optional` if one cannot be.

Specified by:

`describeConstable` in interface `Constable`

Returns:

An `Optional` containing the resulting nominal descriptor, or an empty `Optional` if one cannot be constructed.

Since:

12

toString

```
public String toString()
```

Returns a string representation of the method handle, starting with the string `"MethodHandle"` and ending with the string representation of the method handle's type. In other words, this method returns a string equal to the value of:

```
"MethodHandle" + type().toString()
```

(*Note:* Future releases of this API may add further information to the string representation. Therefore, the present syntax should not be parsed by applications.)

Overrides:

`toString` in class `Object`

Returns:

a string representation of the method handle