Module jdk.incubator.foreign
Package jdk.incubator.foreign

Interface MemorySegment

All Superinterfaces:
`Addressable`

```
public sealed interface MemorySegment
extends Addressable
```

A memory segment models a contiguous region of memory. A memory segment is associated with both spatial and temporal bounds (e.g. a `ResourceScope`). Spatial bounds ensure that memory access operations on a memory segment cannot affect a memory location which falls *outside* the boundaries of the memory segment being accessed. Temporal bounds ensure that memory access operations on a segment cannot occur after the resource scope associated with a memory segment has been closed (see `ResourceScope.close()`).

All implementations of this interface must be value-based; programmers should treat instances that are equal as interchangeable and should not use instances for synchronization, or unpredictable behavior may occur. For example, in a future release, synchronization may fail. The `equals` method should be used for comparisons.

Non-platform classes should not implement MemorySegment directly.

Unless otherwise specified, passing a `null` argument, or an array argument containing one or more `null` elements to a method in this class causes a `NullPointerException` to be thrown.

Constructing memory segments

There are multiple ways to obtain a memory segment. First, memory segments backed by off-heap memory can be allocated using one of the many factory methods provided (see `allocateNative(MemoryLayout, ResourceScope)`, `allocateNative(long, ResourceScope)` and `allocateNative(long, long, ResourceScope)`). Memory segments obtained in this way are called *native memory segments*.

It is also possible to obtain a memory segment backed by an existing heap-allocated Java array, using one of the provided factory methods (e.g. `ofArray(int[])`). Memory segments obtained in this way are called *array memory segments*.

It is possible to obtain a memory segment backed by an existing Java byte buffer (see `ByteBuffer`), using the factory method `ofByteBuffer(ByteBuffer)`. Memory segments obtained in this way are called *buffer memory segments*. Note that buffer memory segments might be backed by native memory (as in the case of native memory segments) or heap memory (as in the case of array memory segments), depending on the characteristics of the byte buffer instance the segment is associated with. For instance, a buffer memory segment obtained from a byte buffer created with the `ByteBuffer.allocateDirect(int)` method will be backed by native memory.

Mapping memory segments from files

It is also possible to obtain a native memory segment backed by a memory-mapped file using the factory method `mapFile(Path, long, long, FileChannel.MapMode, ResourceScope)`. Such native memory segments are called *mapped memory segments*; mapped memory segments are associated with an underlying file descriptor.

Contents of mapped memory segments can be persisted and loaded to and from the underlying file; these capabilities are suitable replacements for some of the functionality in the `MappedByteBuffer` class. Note that, while it is possible to map a segment into a byte buffer (see `asByteBuffer()`), and then call e.g. `MappedByteBuffer.force()` that way, this can only be done when the source segment is small enough, due to the size limitation inherent to the ByteBuffer API.

Clients requiring sophisticated, low-level control over mapped memory segments, should consider writing custom mapped memory segment factories; using `CLinker`, e.g. on Linux, it is possible to call `mmap` with the desired parameters; the returned address can be easily wrapped into a memory segment, using `MemoryAddress.ofLong(long)` and `MemoryAddress.asSegment(long, Runnable, ResourceScope)`.

Lifecycle and confinement

Memory segments are associated with a resource scope (see `ResourceScope`), which can be accessed using the `scope()` method. As for all resources associated with a resource scope, a segment cannot be accessed after its corresponding scope has been closed. For instance, the following code will result in an exception:

```
MemorySegment segment = null;
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    segment = MemorySegment.allocateNative(8, 1, scope);
}
MemoryAccess.getLong(segment); // already closed!
```

Additionally, access to a memory segment is subject to the thread-confinement checks enforced by the owning scope; that is, if the segment is associated with a shared scope, it can be accessed by multiple threads; if it is associated with a confined scope, it can only be accessed by the thread which owns the scope.

Heap and buffer segments are always associated with a *global*, shared scope. This scope cannot be closed, and can be considered as *always alive*.

Memory segment views

Memory segments support *views*. For instance, it is possible to create an *immutable* view of a memory segment, as follows:

```
MemorySegment segment = ...
MemorySegment roSegment = segment.asReadOnly();
```

It is also possible to create views whose spatial bounds are stricter than the ones of the original segment (see `asSlice(long, long)`).

Temporal bounds of the original segment are inherited by the views; that is, when the scope associated with a segment is closed, all the views associated with that segment will also be rendered inaccessible.

To allow for interoperability with existing code, a byte buffer view can be obtained from a memory segment (see `asByteBuffer()`). This can be useful, for instance, for those clients that want to keep using the `ByteBuffer` API, but need to operate on large memory segments. Byte buffers obtained in such a way support the same spatial and temporal access restrictions associated with the memory segment from which they originated.

Stream support

A client might obtain a `Stream` from a segment, which can then be used to slice the segment (according to a given element layout) and even allow multiple threads to work in parallel on disjoint segment slices (to do this, the segment has to be associated with a shared scope). The following code can be used to sum all int values in a memory segment in parallel:

```
try (ResourceScope scope = ResourceScope.newSharedScope()) {
    SequenceLayout SEQUENCE_LAYOUT = MemoryLayout.sequenceLayout(1024, MemoryLayouts.JAVA_INT);
    MemorySegment segment = MemorySegment.allocateNative(SEQUENCE_LAYOUT, scope);
    VarHandle VH_int = SEQUENCE_LAYOUT.elementLayout().varHandle(int.class);
    int sum = segment.elements(MemoryLayouts.JAVA_INT).parallel()
                       .mapToInt(s -> (int)VH_int.get(s.address()))
                       .sum();
}
```

Implementation Requirements:

Implementations of this interface are immutable, thread-safe and value-based.

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method	Description
`MemoryAddress`	`address()`	The base memory address associated with this memory segment.
`static MemorySegment`	`allocateNative(long bytesSize, long alignmentBytes, ResourceScope scope)`	Creates a new confined native memory segment that models a newly allocated block of off-heap memory with given size (in bytes), alignment constraint (in bytes) and resource scope.
`static MemorySegment`	`allocateNative(long bytesSize, ResourceScope scope)`	Creates a new confined native memory segment that models a newly allocated block of off-heap memory with given size (in bytes) and resource scope.
`static MemorySegment`	`allocateNative(MemoryLayout layout, ResourceScope scope)`	Creates a new confined native memory segment that models a newly allocated block of off-heap memory with given layout and resource scope.
`ByteBuffer`	`asByteBuffer()`	Wraps this segment in a `ByteBuffer`.
`MemorySegment`	`asReadOnly()`	Obtains a read-only view of this segment.
`default MemorySegment`	`asSlice(long offset)`	Obtains a new memory segment view whose base address is the same as the base address of this segment plus a given offset, and whose new size is computed by subtracting the specified offset from this segment size.
`MemorySegment`	`asSlice(long offset, long newSize)`	Obtains a new memory segment view whose base address is the same as the base address of this segment plus a given offset, and whose new size is specified by the given argument.

default MemorySegment	asSlice(MemoryAddress newBase)	Obtains a new memory segment view whose base address is the given address, and whose new size is computed by subtracting the address offset relative to this segment (see MemoryAddress.segmentOffset(MemorySegment)) from this segment size.
default MemorySegment	asSlice(MemoryAddress newBase, long newSize)	Obtains a new memory segment view whose base address is the given address, and whose new size is specified by the given argument.
long	byteSize()	The size (in bytes) of this memory segment.
void	copyFrom(MemorySegment src)	Performs a bulk copy from given source segment to this segment.
Stream<MemorySegment>	elements(MemoryLayout elementLayout)	Returns a sequential Stream over disjoint slices (whose size matches that of the specified layout) in this segment.
MemorySegment	fill(byte value)	Fills a value into this memory segment.
void	force()	Forces any changes made to the contents of this mapped segment to be written to the storage device described by the mapped segment's file descriptor.
static MemorySegment	globalNativeSegment()	Returns a native memory segment whose base address is MemoryAddress.NULL and whose size is Long.MAX_VALUE.
boolean	isLoaded()	Tells whether or not the contents of this mapped segment is resident in physical memory.
boolean	isMapped()	Is this a mapped segment?
boolean	isNative()	Is this a native segment?
boolean	isReadOnly()	Is this segment read-only?
void	load()	Loads the contents of this mapped segment into physical memory.
static MemorySegment	mapFile(Path path, long bytesOffset, long bytesSize, FileChannel.MapMode mapMode, ResourceScope scope)	Creates a new mapped memory segment that models a memory-mapped region of a file from a given path.
long	mismatch(MemorySegment other)	Finds and returns the offset, in bytes, of the first mismatch between this segment and a given other segment.
static MemorySegment	ofArray(byte[] arr)	Creates a new confined array memory segment that models the memory associated with a given heap-allocated byte array.
static MemorySegment	ofArray(char[] arr)	Creates a new confined array memory segment that models the memory associated with a given heap-allocated char array.
static MemorySegment	ofArray(double[] arr)	Creates a new confined array memory segment that models the memory associated with a given heap-allocated double array.
static MemorySegment	ofArray(float[] arr)	Creates a new confined array memory segment that models the memory associated with a given heap-allocated float array.
static MemorySegment	ofArray(int[] arr)	Creates a new confined array memory segment that models the memory associated with a given heap-allocated int array.
static MemorySegment	ofArray(long[] arr)	Creates a new confined array memory segment that models the memory associated with a given heap-allocated long array.
static MemorySegment	ofArray(short[] arr)	Creates a new confined array memory segment that models the memory associated with a given heap-allocated short array.

static MemorySegment	ofByteBuffer(ByteBuffer bb)	Creates a new confined buffer memory segment that models the memory associated with the given byte buffer.
ResourceScope	scope()	Returns the resource scope associated with this memory segment.
Spliterator<MemorySegment>	spliterator(MemoryLayout elementLayout)	Returns a spliterator for this memory segment.
byte[]	toByteArray()	Copy the contents of this memory segment into a fresh byte array.
char[]	toCharArray()	Copy the contents of this memory segment into a fresh char array.
double[]	toDoubleArray()	Copy the contents of this memory segment into a fresh double array.
float[]	toFloatArray()	Copy the contents of this memory segment into a fresh float array.
int[]	toIntArray()	Copy the contents of this memory segment into a fresh int array.
long[]	toLongArray()	Copy the contents of this memory segment into a fresh long array.
short[]	toShortArray()	Copy the contents of this memory segment into a fresh short array.
void	unload()	Unloads the contents of this mapped segment from physical memory.

Method Details

address

`MemoryAddress address()`

The base memory address associated with this memory segment. The returned memory address is associated with same resource scope as that associated with this segment.

Specified by:

`address` in interface `Addressable`

Returns:

The base memory address.

spliterator

`Spliterator<MemorySegment> spliterator(MemoryLayout elementLayout)`

Returns a spliterator for this memory segment. The returned spliterator reports `Spliterator.SIZED`, `Spliterator.SUBSIZED`, `Spliterator.IMMUTABLE`, `Spliterator.NONNULL` and `Spliterator.ORDERED` characteristics.

The returned spliterator splits this segment according to the specified element layout; that is, if the supplied layout has size N, then calling `Spliterator.trySplit()` will result in a spliterator serving approximately `S/N/2` elements (depending on whether N is even or not), where `S` is the size of this segment. As such, splitting is possible as long as `S/N >= 2`. The spliterator returns segments that feature the same scope as this given segment.

The returned spliterator effectively allows to slice this segment into disjoint sub-segments, which can then be processed in parallel by multiple threads.

Parameters:

`elementLayout` - the layout to be used for splitting.

Returns:

the element spliterator for this segment

Throws:

`IllegalArgumentException` - if the `elementLayout` size is zero, or the segment size modulo the `elementLayout` size is greater than zero.

elements

`Stream<MemorySegment> elements(MemoryLayout elementLayout)`

Returns a sequential `Stream` over disjoint slices (whose size matches that of the specified layout) in this segment. Calling this method is equivalent to the following code:

```
        StreamSupport.stream(segment.spliterator(elementLayout), false);
```

Parameters:

`elementLayout` - the layout to be used for splitting.

Returns:

a sequential `Stream` over disjoint slices in this segment.

Throws:

`IllegalArgumentException` - if the `elementLayout` size is zero, or the segment size modulo the `elementLayout` size is greater than zero.

scope

`ResourceScope scope()`

Returns the resource scope associated with this memory segment.

Returns:

the resource scope associated with this memory segment.

byteSize

`long byteSize()`

The size (in bytes) of this memory segment.

Returns:

The size (in bytes) of this memory segment.

asSlice

```
MemorySegment asSlice(long offset,
                      long newSize)
```

Obtains a new memory segment view whose base address is the same as the base address of this segment plus a given offset, and whose new size is specified by the given argument.

Parameters:

`offset` - The new segment base offset (relative to the current segment base address), specified in bytes.

`newSize` - The new segment size, specified in bytes.

Returns:

a new memory segment view with updated base/limit addresses.

Throws:

`IndexOutOfBoundsException` - if `offset < 0`, `offset > byteSize()`, `newSize < 0`, or `newSize > byteSize() - offset`

See Also:

`asSlice(long)`,
`asSlice(MemoryAddress)`,
`asSlice(MemoryAddress, long)`

asSlice

```
default MemorySegment asSlice(MemoryAddress newBase,
                              long newSize)
```

Obtains a new memory segment view whose base address is the given address, and whose new size is specified by the given argument.

Equivalent to the following code:

```
    asSlice(newBase.segmentOffset(this), newSize);
```

Parameters:

`newBase` - The new segment base address.

`newSize` - The new segment size, specified in bytes.

Returns:

a new memory segment view with updated base/limit addresses.

Throws:

`IndexOutOfBoundsException` - if `offset < 0`, `offset > byteSize()`, `newSize < 0`, or `newSize > byteSize() - offset`

See Also:

`asSlice(long)`,
`asSlice(MemoryAddress)`,
`asSlice(long, long)`

asSlice

`default MemorySegment asSlice(long offset)`

Obtains a new memory segment view whose base address is the same as the base address of this segment plus a given offset, and whose new size is computed by subtracting the specified offset from this segment size.

Equivalent to the following code:

```
    asSlice(offset, byteSize() - offset);
```

Parameters:

`offset` - The new segment base offset (relative to the current segment base address), specified in bytes.

Returns:

a new memory segment view with updated base/limit addresses.

Throws:

`IndexOutOfBoundsException` - if `offset < 0`, or `offset > byteSize()`.

See Also:

`asSlice(MemoryAddress)`,
`asSlice(MemoryAddress, long)`,
`asSlice(long, long)`

asSlice

`default MemorySegment asSlice(MemoryAddress newBase)`

Obtains a new memory segment view whose base address is the given address, and whose new size is computed by subtracting the address offset relative to this segment (see `MemoryAddress.segmentOffset(MemorySegment)`) from this segment size.

Equivalent to the following code:

```
    asSlice(newBase.segmentOffset(this));
```

Parameters:

`newBase` - The new segment base offset (relative to the current segment base address), specified in bytes.

Returns:

a new memory segment view with updated base/limit addresses.

Throws:

`IndexOutOfBoundsException` - if `address.segmentOffset(this) < 0`, or `address.segmentOffset(this) > byteSize()`.

See Also:

`asSlice(long)`,
`asSlice(MemoryAddress, long)`,
`asSlice(long, long)`

isReadOnly

`boolean isReadOnly()`

Is this segment read-only?

Returns:

`true`, if this segment is read-only.

See Also:

`asReadOnly()`

asReadOnly

`MemorySegment asReadOnly()`

Obtains a read-only view of this segment. The resulting segment will be identical to this one, but attempts to overwrite the contents of the returned segment will cause runtime exceptions.

Returns:

a read-only view of this segment

See Also:

`isReadOnly()`

isNative

`boolean isNative()`

Is this a native segment? Returns true if this segment is a native memory segment, created using the `allocateNative(long, ResourceScope)` (and related) factory, or a buffer segment derived from a direct `ByteBuffer` using the `ofByteBuffer(ByteBuffer)` factory, or if this is a mapped segment.

Returns:

`true` if this segment is native segment.

isMapped

`boolean isMapped()`

Is this a mapped segment? Returns true if this segment is a mapped memory segment, created using the `mapFile(Path, long, long, FileChannel.MapMode, ResourceScope)` factory, or a buffer segment derived from a `MappedByteBuffer` using the `ofByteBuffer(ByteBuffer)` factory.

Returns:

`true` if this segment is a mapped segment.

fill

`MemorySegment fill(byte value)`

Fills a value into this memory segment.

More specifically, the given value is filled into each address of this segment. Equivalent to (but likely more efficient than) the following code:

```
byteHandle = MemoryLayout.ofSequence(MemoryLayouts.JAVA_BYTE)
         .varHandle(byte.class, MemoryLayout.PathElement.sequenceElement());
for (long l = 0; l < segment.byteSize(); l++) {
     byteHandle.set(segment.address(), l, value);
}
```

without any regard or guarantees on the ordering of particular memory elements being set.

Fill can be useful to initialize or reset the memory of a segment.

Parameters:

`value` - the value to fill into this segment

Returns:

this memory segment

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope,

`UnsupportedOperationException` - if this segment is read-only (see `isReadOnly()`).

copyFrom

`void copyFrom(MemorySegment src)`

Performs a bulk copy from given source segment to this segment. More specifically, the bytes at offset `0` through `src.byteSize() - 1` in the source segment are copied into this segment at offset `0` through `src.byteSize() - 1`. If the source segment overlaps with this segment, then the copying is performed as if the bytes at offset `0` through `src.byteSize() - 1` in the source segment were first copied into a temporary segment with size `bytes`, and then the contents of the temporary segment were copied into this segment at offset `0` through `src.byteSize() - 1`.

The result of a bulk copy is unspecified if, in the uncommon case, the source segment and this segment do not overlap, but refer to overlapping regions of the same backing storage using different addresses. For example, this may occur if the same file is mapped to two segments.

Parameters:

`src` - the source segment.

Throws:

`IndexOutOfBoundsException` - if `src.byteSize() > this.byteSize()`.

`IllegalStateException` - if either the scope associated with the source segment or the scope associated with this segment have been already closed, or if access occurs from a thread other than the thread owning either scopes.

`UnsupportedOperationException` - if this segment is read-only (see `isReadOnly()`).

mismatch

`long mismatch(MemorySegment other)`

Finds and returns the offset, in bytes, of the first mismatch between this segment and a given other segment. The offset is relative to the base address of each segment and will be in the range of 0 (inclusive) up to the size (in bytes) of the smaller memory segment (exclusive).

If the two segments share a common prefix then the returned offset is the length of the common prefix and it follows that there is a mismatch between the two segments at that offset within the respective segments. If one segment is a proper prefix of the other then the returned offset is the smaller of the segment sizes, and it follows that the offset is only valid for the larger segment. Otherwise, there is no mismatch and `-1` is returned.

Parameters:

`other` - the segment to be tested for a mismatch with this segment

Returns:

the relative offset, in bytes, of the first mismatch between this and the given other segment, otherwise -1 if no mismatch

Throws:

`IllegalStateException` - if either the scope associated with this segment or the scope associated with the `other` segment have been already closed, or if access occurs from a thread other than the thread owning either scopes.

isLoaded

`boolean isLoaded()`

Tells whether or not the contents of this mapped segment is resident in physical memory.

A return value of `true` implies that it is highly likely that all of the data in this segment is resident in physical memory and may therefore be accessed without incurring any virtual-memory page faults or I/O operations. A return value of `false` does not necessarily imply that this segment's content is not resident in physical memory.

The returned value is a hint, rather than a guarantee, because the underlying operating system may have paged out some of this segment's data by the time that an invocation of this method returns.

Returns:

`true` if it is likely that the contents of this segment is resident in physical memory

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope.

`UnsupportedOperationException` - if this segment is not a mapped memory segment, e.g. if `isMapped() == false`.

load

`void load()`

Loads the contents of this mapped segment into physical memory.

This method makes a best effort to ensure that, when it returns, this contents of this segment is resident in physical memory. Invoking this method may cause some number of page faults and I/O operations to occur.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope.

`UnsupportedOperationException` - if this segment is not a mapped memory segment, e.g. if `isMapped() == false`.

unload

`void unload()`

Unloads the contents of this mapped segment from physical memory.

This method makes a best effort to ensure that the contents of this segment are are no longer resident in physical memory. Accessing this segment's contents after invoking this method may cause some number of page faults and I/O operations to occur (as this segment's contents might need to be paged back in).

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope.

`UnsupportedOperationException` - if this segment is not a mapped memory segment, e.g. if `isMapped() == false`.

force

```
void force()
```

Forces any changes made to the contents of this mapped segment to be written to the storage device described by the mapped segment's file descriptor.

If the file descriptor associated with this mapped segment resides on a local storage device then when this method returns it is guaranteed that all changes made to this segment since it was created, or since this method was last invoked, will have been written to that device.

If the file descriptor associated with this mapped segment does not reside on a local device then no such guarantee is made.

If this segment was not mapped in read/write mode (`FileChannel.MapMode.READ_WRITE`) then invoking this method may have no effect. In particular, the method has no effect for segments mapped in read-only or private mapping modes. This method may or may not have an effect for implementation-specific mapping modes.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope.

`UnsupportedOperationException` - if this segment is not a mapped memory segment, e.g. if `isMapped() == false`.

`UncheckedIOException` - if there is an I/O error writing the contents of this segment to the associated storage device

asByteBuffer

```
ByteBuffer asByteBuffer()
```

Wraps this segment in a `ByteBuffer`. Some of the properties of the returned buffer are linked to the properties of this segment. For instance, if this segment is *immutable* (e.g. the segment is a read-only segment, see `isReadOnly()`), then the resulting buffer is *read-only* (see `Buffer.isReadOnly()`. Additionally, if this is a native memory segment, the resulting buffer is *direct* (see `ByteBuffer.isDirect()`).

The returned buffer's position (see `Buffer.position()` is initially set to zero, while the returned buffer's capacity and limit (see `Buffer.capacity()` and `Buffer.limit()`, respectively) are set to this segment' size (see `byteSize()`). For this reason, a byte buffer cannot be returned if this segment' size is greater than `Integer.MAX_VALUE`.

The life-cycle of the returned buffer will be tied to that of this segment. That is, accessing the returned buffer after the scope associated with this segment has been closed (see `ResourceScope.close()`, will throw an `IllegalStateException`.

If this segment is associated with a confined scope, calling read/write I/O operations on the resulting buffer might result in an unspecified exception being thrown. Examples of such problematic operations are `AsynchronousSocketChannel.read(ByteBuffer)` and `AsynchronousSocketChannel.write(ByteBuffer)`.

Finally, the resulting buffer's byte order is `ByteOrder.BIG_ENDIAN`; this can be changed using `ByteBuffer.order(java.nio.ByteOrder)`.

Returns:

a `ByteBuffer` view of this memory segment.

Throws:

`UnsupportedOperationException` - if this segment cannot be mapped onto a `ByteBuffer` instance, e.g. because it models an heap-based segment that is not based on a `byte[]`), or if its size is greater than `Integer.MAX_VALUE`.

toByteArray

```
byte[] toByteArray()
```

Copy the contents of this memory segment into a fresh byte array.

Returns:

a fresh byte array copy of this memory segment.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope, or if this segment's contents cannot be copied into a `byte` instance, e.g. its size is greater than `Integer.MAX_VALUE`.

toShortArray

`short[] toShortArray()`

Copy the contents of this memory segment into a fresh short array.

Returns:

a fresh short array copy of this memory segment.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope, or if this segment's contents cannot be copied into a `short` instance, e.g. because `byteSize() % 2 != 0`, or `byteSize() / 2 > Integer#MAX_VALUE`

toCharArray

`char[] toCharArray()`

Copy the contents of this memory segment into a fresh char array.

Returns:

a fresh char array copy of this memory segment.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope, or if this segment's contents cannot be copied into a `char` instance, e.g. because `byteSize() % 2 != 0`, or `byteSize() / 2 > Integer#MAX_VALUE`.

toIntArray

`int[] toIntArray()`

Copy the contents of this memory segment into a fresh int array.

Returns:

a fresh int array copy of this memory segment.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope, or if this segment's contents cannot be copied into a `int` instance, e.g. because `byteSize() % 4 != 0`, or `byteSize() / 4 > Integer#MAX_VALUE`.

toFloatArray

`float[] toFloatArray()`

Copy the contents of this memory segment into a fresh float array.

Returns:

a fresh float array copy of this memory segment.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope, or if this segment's contents cannot be copied into a `float` instance, e.g. because `byteSize() % 4 != 0`, or `byteSize() / 4 > Integer#MAX_VALUE`.

toLongArray

`long[] toLongArray()`

Copy the contents of this memory segment into a fresh long array.

Returns:

a fresh long array copy of this memory segment.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope, or if this segment's contents cannot be copied into a `long` instance, e.g. because `byteSize() % 8 != 0`, or `byteSize() / 8 > Integer#MAX_VALUE`.

toDoubleArray

`double[] toDoubleArray()`

Copy the contents of this memory segment into a fresh double array.

Returns:

a fresh double array copy of this memory segment.

Throws:

`IllegalStateException` - if the scope associated with this segment has been closed, or if access occurs from a thread other than the thread owning that scope, or if this segment's contents cannot be copied into a `double` instance, e.g. because `byteSize() % 8 != 0`, or `byteSize() / 8 > Integer#MAX_VALUE`.

ofByteBuffer

```
static MemorySegment ofByteBuffer(ByteBuffer bb)
```

Creates a new confined buffer memory segment that models the memory associated with the given byte buffer. The segment starts relative to the buffer's position (inclusive) and ends relative to the buffer's limit (exclusive).

If the buffer is `read-only`, the resulting segment will also be `read-only`. The scope associated with this segment can either be the global resource scope, in case the buffer has been created independently, or to some other (possibly closeable) resource scope, in case the buffer has been obtained using `asByteBuffer()`.

The resulting memory segment keeps a reference to the backing buffer, keeping it *reachable*.

Parameters:

`bb` - the byte buffer backing the buffer memory segment.

Returns:

a new buffer memory segment.

ofArray

```
static MemorySegment ofArray(byte[] arr)
```

Creates a new confined array memory segment that models the memory associated with a given heap-allocated byte array. The returned segment's resource scope is set to the global resource scope.

Parameters:

`arr` - the primitive array backing the array memory segment.

Returns:

a new array memory segment.

ofArray

```
static MemorySegment ofArray(char[] arr)
```

Creates a new confined array memory segment that models the memory associated with a given heap-allocated char array. The returned segment's resource scope is set to the global resource scope.

Parameters:

`arr` - the primitive array backing the array memory segment.

Returns:

a new array memory segment.

ofArray

```
static MemorySegment ofArray(short[] arr)
```

Creates a new confined array memory segment that models the memory associated with a given heap-allocated short array. The returned segment's resource scope is set to the global resource scope.

Parameters:

`arr` - the primitive array backing the array memory segment.

Returns:

a new array memory segment.

ofArray

```
static MemorySegment ofArray(int[] arr)
```

Creates a new confined array memory segment that models the memory associated with a given heap-allocated int array. The returned segment's resource scope is set to the global resource scope.

Parameters:

`arr` - the primitive array backing the array memory segment.

Returns:

a new array memory segment.

ofArray

```
static MemorySegment ofArray(float[] arr)
```

Creates a new confined array memory segment that models the memory associated with a given heap-allocated float array. The returned segment's resource scope is set to the global resource scope.

Parameters:

`arr` - the primitive array backing the array memory segment.

Returns:

a new array memory segment.

ofArray

```
static MemorySegment ofArray(long[] arr)
```

Creates a new confined array memory segment that models the memory associated with a given heap-allocated long array. The returned segment's resource scope is set to the global resource scope.

Parameters:

`arr` - the primitive array backing the array memory segment.

Returns:

a new array memory segment.

ofArray

```
static MemorySegment ofArray(double[] arr)
```

Creates a new confined array memory segment that models the memory associated with a given heap-allocated double array. The returned segment's resource scope is set to the global resource scope.

Parameters:

`arr` - the primitive array backing the array memory segment.

Returns:

a new array memory segment.

allocateNative

```
static MemorySegment allocateNative(MemoryLayout layout,
                                    ResourceScope scope)
```

Creates a new confined native memory segment that models a newly allocated block of off-heap memory with given layout and resource scope. A client is responsible make sure that the resource scope associated with the returned segment is closed when the segment is no longer in use. Failure to do so will result in off-heap memory leaks.

This is equivalent to the following code:

```
        allocateNative(layout.bytesSize(), layout.bytesAlignment(), scope);
```

The block of off-heap memory associated with the returned native memory segment is initialized to zero.

Parameters:

`layout` - the layout of the off-heap memory block backing the native memory segment.

`scope` - the segment scope.

Returns:

a new native memory segment.

Throws:

`IllegalArgumentException` - if the specified layout has illegal size or alignment constraint.

`IllegalStateException` - if `scope` has been already closed, or if access occurs from a thread other than the thread owning `scope`.

allocateNative

```
static MemorySegment allocateNative(long bytesSize,
                                    ResourceScope scope)
```

Creates a new confined native memory segment that models a newly allocated block of off-heap memory with given size (in bytes) and resource scope. A client is responsible make sure that the resource scope associated with the returned segment is closed when the segment is no longer in use. Failure to do so will result in off-heap memory leaks.

This is equivalent to the following code:

```
        allocateNative(bytesSize, 1, scope);
```

The block of off-heap memory associated with the returned native memory segment is initialized to zero.

Parameters:

`bytesSize` - the size (in bytes) of the off-heap memory block backing the native memory segment.

`scope` - the segment scope.

Returns:

a new native memory segment.

Throws:

`IllegalArgumentException` - if `bytesSize <= 0`.

`IllegalStateException` - if `scope` has been already closed, or if access occurs from a thread other than the thread owning `scope`.

allocateNative

```
static MemorySegment allocateNative(long bytesSize,
                                    long alignmentBytes,
                                    ResourceScope scope)
```

Creates a new confined native memory segment that models a newly allocated block of off-heap memory with given size (in bytes), alignment constraint (in bytes) and resource scope. A client is responsible make sure that the resource scope associated with the returned segment is closed when the segment is no longer in use. Failure to do so will result in off-heap memory leaks.

The block of off-heap memory associated with the returned native memory segment is initialized to zero.

Parameters:

`bytesSize` - the size (in bytes) of the off-heap memory block backing the native memory segment.

`alignmentBytes` - the alignment constraint (in bytes) of the off-heap memory block backing the native memory segment.

`scope` - the segment scope.

Returns:

a new native memory segment.

Throws:

`IllegalArgumentException` - if `bytesSize <= 0`, `alignmentBytes <= 0`, or if `alignmentBytes` is not a power of 2.

`IllegalStateException` - if `scope` has been already closed, or if access occurs from a thread other than the thread owning `scope`.

mapFile

```
static MemorySegment mapFile(Path path,
                             long bytesOffset,
                             long bytesSize,
                             FileChannel.MapMode mapMode,
                             ResourceScope scope)
                      throws IOException
```

Creates a new mapped memory segment that models a memory-mapped region of a file from a given path.

If the specified mapping mode is READ_ONLY, the resulting segment will be read-only (see `isReadOnly()`).

The content of a mapped memory segment can change at any time, for example if the content of the corresponding region of the mapped file is changed by this (or another) program. Whether or not such changes occur, and when they occur, is operating-system dependent and therefore unspecified.

All or part of a mapped memory segment may become inaccessible at any time, for example if the backing mapped file is truncated. An attempt to access an inaccessible region of a mapped memory segment will not change the segment's content and will cause an unspecified exception to be thrown either at the time of the access or at some later time. It is therefore strongly recommended that appropriate precautions be taken to avoid the manipulation of a mapped file by this (or another) program, except to read or write the file's content.

Implementation Note:

When obtaining a mapped segment from a newly created file, the initialization state of the contents of the block of mapped memory associated with the returned mapped memory segment is unspecified and should not be relied upon.

Parameters:

`path` - the path to the file to memory map.

`bytesOffset` - the offset (expressed in bytes) within the file at which the mapped segment is to start.

`bytesSize` - the size (in bytes) of the mapped memory backing the memory segment.

`mapMode` - a file mapping mode, see `FileChannel.map(FileChannel.MapMode, long, long)`; the chosen mapping mode might affect the behavior of the returned memory mapped segment (see `force()`).

`scope` - the segment scope.

Returns:

a new confined mapped memory segment.

Throws:

`IllegalArgumentException` - if `bytesOffset < 0`, `bytesSize < 0`, or if `path` is not associated with the default file system.

`IllegalStateException` - if `scope` has been already closed, or if access occurs from a thread other than the thread owning `scope`.

`UnsupportedOperationException` - if an unsupported map mode is specified.

`IOException` - if the specified path does not point to an existing file, or if some other I/O error occurs.

`SecurityException` - If a security manager is installed and it denies an unspecified permission required by the implementation. In the case of the default provider, the `SecurityManager.checkRead(String)` method is invoked to check read access if the file is opened for reading. The `SecurityManager.checkWrite(String)` method is invoked to check write access if the file is opened for writing.

globalNativeSegment

```
static MemorySegment globalNativeSegment()
```

Returns a native memory segment whose base address is `MemoryAddress.NULL` and whose size is `Long.MAX_VALUE`. This method can be very useful when dereferencing memory addresses obtained when interacting with native libraries. The returned segment is associated with the *global* resource scope (see `ResourceScope.globalScope()`). Equivalent to (but likely more efficient than) the following code:

```
    MemoryAddress.NULL.asSegment(Long.MAX_VALUE)
```

This method is *restricted*. Restricted methods are unsafe, and, if used incorrectly, their use might crash the JVM or, worse, silently result in memory corruption. Thus, clients should refrain from depending on restricted methods, and use safe and supported functionalities, where possible.

Returns:

a memory segment whose base address is `MemoryAddress.NULL` and whose size is `Long.MAX_VALUE`.

Throws:

`IllegalCallerException` - if access to this method occurs from a module `M` and the command line option `--enable-native-access` is either absent, or does not mention the module name `M`, or `ALL-UNNAMED` in case `M` is an unnamed module.