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 capabilities 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 `ofAddress(MemoryAddress, long, ResourceScope)`.

Restricted native segments

Sometimes it is necessary to turn a memory address obtained from native code into a memory segment with full spatial, temporal and confinement bounds. To do this, clients can `obtain` a native segment *unsafely* from a give memory address, by providing the segment size, as well as the segment scope. This is a *restricted* operation and should be used with caution: for instance, an incorrect segment size could result in a VM crash when attempting to dereference the memory segment.

Dereference

A memory segment can be read or written using various methods provided in this class (e.g. `get(ValueLayout.OfInt, long)`). Each dereference method takes a value layout, which specifies the size, alignment constraints, byte order as well as the Java type associated with the dereference operation, and an offset. For instance, to read an int from a segment, using `default endianness`, the following code can be used:

```
MemorySegment segment = ...
int value = segment.get(ValueLayout.JAVA_INT, 0);
```

If the value to be read is stored in memory using `big-endian` encoding, the dereference operation can be expressed as follows:

```
MemorySegment segment = ...
int value = segment.get(ValueLayout.JAVA_INT.withOrder(BIG_ENDIAN), 0);
```

For more complex dereference operations (e.g. structured memory access), clients can obtain a *memory access var handle*, that is, a var handle that accepts a segment and, optionally, one or more additional `long` coordinates. Memory access var handles can be obtained from memory layouts by providing a so called *layout path*. Alternatively, clients can obtain raw memory access var handles from a given value layout, and then adapt it using the var handle combinator functions defined in the `MemoryHandles` class.

Alignment

When dereferencing a memory segment using a layout, the runtime must check that the segment address being dereferenced matches the layout's alignment constraints. If the segment being dereferenced is a native segment, then it has a concrete base address, which can be used to perform the alignment check. The pseudo-function below demonstrates this:

```
boolean isAligned(MemorySegment segment, long offset, MemoryLayout layout) {
   return ((segment.address().toRawLongValue() + offset) % layout.byteAlignment()) == 0
}
```

If, however, the segment being dereferenced is a heap segment, the above function will not work: a heap segment's base address is *virtualized* and, as such, cannot be used to construct an alignment check. Instead, heap segments are assumed to produce addresses which are never more aligned than the element size of the Java array from which they have originated from, as shown in the following table:

Array type	Alignment
`boolean[]`	1
`byte[]`	1
`char[]`	2
`short[]`	2
`int[]`	4
`float[]`	4
`long[]`	8
`double[]`	8

Note that the above definition is conservative: it might be possible, for instance, that a heap segment constructed from a `byte[]` might have a subset of addresses `S` which happen to be 8-byte aligned. But determining which segment addresses belong to `S` requires reasoning about details which are ultimately implementation-dependent.

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, scope);
}
segment.get(ValueLayout.JAVA_LONG, 0); // 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 segments associated with it 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, ValueLayout.JAVA_INT);
    MemorySegment segment = MemorySegment.allocateNative(SEQUENCE_LAYOUT, scope);
    int sum = segment.elements(ValueLayout.JAVA_INT).parallel()
                     .mapToInt(s -> s.get(ValueLayout.JAVA_INT, 0))
                     .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()`	Returns the base memory address associated with this native memory segment.
`static MemorySegment`	`allocateNative(long bytesSize, long alignmentBytes, ResourceScope scope)`	Creates a new 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 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 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`	`asOverlappingSlice(MemorySegment other)`	Returns a slice of this segment that is the overlap between this and the provided segment.
`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.
`long`	`byteSize()`	Returns the size (in bytes) of this memory segment.
`static void`	`copy(Object srcArray, int srcIndex, MemorySegment dstSegment, ValueLayout dstLayout, long dstOffset, int elementCount)`	Copies a number of elements from a source array to a destination segment, starting at a given array index, and a given segment offset (expressed in bytes), using the given destination element layout.
`static void`	`copy(MemorySegment srcSegment, long srcOffset, MemorySegment dstSegment, long dstOffset, long bytes)`	Performs a bulk copy from source segment to destination segment.
`static void`	`copy(MemorySegment srcSegment, ValueLayout srcLayout, long srcOffset, Object dstArray, int dstIndex, int elementCount)`	Copies a number of elements from a source segment to a destination array, starting at a given segment offset (expressed in bytes), and a given array index, using the given source element layout.

static void	copy(MemorySegment srcSegment, ValueLayout srcElementLayout, long srcOffset, MemorySegment dstSegment, ValueLayout dstElementLayout, long dstOffset, long elementCount)	Performs a bulk copy from source segment to destination segment.
default MemorySegment	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.
default MemoryAddress	get(ValueLayout.OfAddress layout, long offset)	Reads an address from this segment and offset with given layout.
default boolean	get(ValueLayout.OfBoolean layout, long offset)	Reads a boolean from this segment and offset with given layout.
default byte	get(ValueLayout.OfByte layout, long offset)	Reads a byte from this segment and offset with given layout.
default char	get(ValueLayout.OfChar layout, long offset)	Reads a char from this segment and offset with given layout.
default double	get(ValueLayout.OfDouble layout, long offset)	Reads a double from this segment and offset with given layout.
default float	get(ValueLayout.OfFloat layout, long offset)	Reads a float from this segment and offset with given layout.
default int	get(ValueLayout.OfInt layout, long offset)	Reads an int from this segment and offset with given layout.
default long	get(ValueLayout.OfLong layout, long offset)	Reads a long from this segment and offset with given layout.
default short	get(ValueLayout.OfShort layout, long offset)	Reads a short from this segment and offset with given layout.
default MemoryAddress	getAtIndex (ValueLayout.OfAddress layout, long index)	Reads an address from this segment and index, scaled by given layout size.
default char	getAtIndex(ValueLayout.OfChar layout, long index)	Reads a char from this segment and index, scaled by given layout size.
default double	getAtIndex(ValueLayout.OfDouble layout, long index)	Reads a double from this segment and index, scaled by given layout size.
default float	getAtIndex(ValueLayout.OfFloat layout, long index)	Reads a float from this segment and index, scaled by given layout size.
default int	getAtIndex(ValueLayout.OfInt layout, long index)	Reads an int from this segment and index, scaled by given layout size.
default long	getAtIndex(ValueLayout.OfLong layout, long index)	Reads a long from this segment and index, scaled by given layout size.
default short	getAtIndex(ValueLayout.OfShort layout, long index)	Reads a short from this segment and index, scaled by given layout size.
default String	getUtf8String(long offset)	Reads a UTF-8 encoded, null-terminated string from this segment at given offset.
boolean	isLoaded()	Determines whether the contents of this mapped segment is resident in physical memory.
boolean	isMapped()	Returns `true` if this segment is a mapped segment.
boolean	isNative()	Returns `true` if this segment is a native segment.
boolean	isReadOnly()	Returns `true`, if this segment is 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	ofAddress(MemoryAddress address, long bytesSize, ResourceScope scope)	Creates a new native memory segment with given size and resource scope, and whose base address is the given address.
static MemorySegment	ofArray(byte[] arr)	Creates a new array memory segment that models the memory associated with a given heap-allocated byte array.
static MemorySegment	ofArray(char[] arr)	Creates a new array memory segment that models the memory associated with a given heap-allocated char array.
static MemorySegment	ofArray(double[] arr)	Creates a new array memory segment that models the memory associated with a given heap-allocated double array.
static MemorySegment	ofArray(float[] arr)	Creates a new array memory segment that models the memory associated with a given heap-allocated float array.
static MemorySegment	ofArray(int[] arr)	Creates a new array memory segment that models the memory associated with a given heap-allocated int array.
static MemorySegment	ofArray(long[] arr)	Creates a new array memory segment that models the memory associated with a given heap-allocated long array.
static MemorySegment	ofArray(short[] arr)	Creates a new array memory segment that models the memory associated with a given heap-allocated short array.
static MemorySegment	ofByteBuffer(ByteBuffer bb)	Creates a new buffer memory segment that models the memory associated with the given byte buffer.
ResourceScope	scope()	Returns the resource scope associated with this memory segment.
long	segmentOffset(MemorySegment other)	Returns the offset, in bytes, of the provided segment, relative to this segment.
default void	set(ValueLayout.OfAddress layout, long offset, Addressable value)	Writes an address to this segment and offset with given layout.
default void	set(ValueLayout.OfBoolean layout, long offset, boolean value)	Writes a boolean to this segment and offset with given layout.
default void	set(ValueLayout.OfByte layout, long offset, byte value)	Writes a byte to this segment and offset with given layout.
default void	set(ValueLayout.OfChar layout, long offset, char value)	Writes a char to this segment and offset with given layout.
default void	set(ValueLayout.OfDouble layout, long offset, double value)	Writes a double to this segment and offset with given layout.
default void	set(ValueLayout.OfFloat layout, long offset, float value)	Writes a float to this segment and offset with given layout.
default void	set(ValueLayout.OfInt layout, long offset, int value)	Writes an int to this segment and offset with given layout.
default void	set(ValueLayout.OfLong layout, long offset, long value)	Writes a long to this segment and offset with given layout.
default void	set(ValueLayout.OfShort layout, long offset, short value)	Writes a short to this segment and offset with given layout.
default void	setAtIndex (ValueLayout.OfAddress layout,	Writes an address to this segment and index, scaled by given layout size.

	long index, Addressable value)	
default void	setAtIndex(ValueLayout.OfChar layout, long index, char value)	Writes a char to this segment and index, scaled by given layout size.
default void	setAtIndex(ValueLayout.OfDouble layout, long index, double value)	Writes a double to this segment and index, scaled by given layout size.
default void	setAtIndex(ValueLayout.OfFloat layout, long index, float value)	Writes a float to this segment and index, scaled by given layout size.
default void	setAtIndex(ValueLayout.OfInt layout, long index, int value)	Writes an int to this segment and index, scaled by given layout size.
default void	setAtIndex(ValueLayout.OfLong layout, long index, long value)	Writes a long to this segment and index, scaled by given layout size.
default void	setAtIndex(ValueLayout.OfShort layout, long index, short value)	Writes a short to this segment and index, scaled by given layout size.
default void	setUtf8String(long offset, String str)	Writes the given string into this segment at given offset, converting it to a null-terminated byte sequence using UTF-8 encoding.
Spliterator<MemorySegment>	spliterator(MemoryLayout elementLayout)	Returns a spliterator for this memory segment.
byte[]	toArray (ValueLayout.OfByte elementLayout)	Copy the contents of this memory segment into a fresh byte array.
char[]	toArray (ValueLayout.OfChar elementLayout)	Copy the contents of this memory segment into a fresh char array.
double[]	toArray (ValueLayout.OfDouble elementLayout)	Copy the contents of this memory segment into a fresh double array.
float[]	toArray (ValueLayout.OfFloat elementLayout)	Copy the contents of this memory segment into a fresh float array.
int[]	toArray (ValueLayout.OfInt elementLayout)	Copy the contents of this memory segment into a fresh int array.
long[]	toArray (ValueLayout.OfLong elementLayout)	Copy the contents of this memory segment into a fresh long array.
short[]	toArray (ValueLayout.OfShort elementLayout)	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()`

Returns the base memory address associated with this native memory segment.

Specified by:

`address` in interface `Addressable`

Returns:

the base memory address associated with this native memory segment

Throws:

`UnsupportedOperationException` - if this segment is not a `native` segment.

`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.

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` 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 are associated with the same scope as this segment.

The returned spliterator effectively allows to slice this segment into disjoint slices, 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, if this segment is incompatible with the alignment constraints in the provided layout, or if the `elementLayout` alignment is greater than its size.

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, if this segment is incompatible with the alignment constraints in the provided layout, or if the `elementLayout` alignment is greater than its size.

scope

`ResourceScope scope()`

Returns the resource scope associated with this memory segment.

Returns:

the resource scope associated with this memory segment

byteSize

`long byteSize()`

Returns 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

`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(long, long)`

isReadOnly

`boolean isReadOnly()`

Returns `true`, if this segment is 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()`

Returns `true` if this segment is a native segment. A native memory segment is 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()`

Returns `true` if this segment is a mapped segment. A mapped memory segment is 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.

asOverlappingSlice

`MemorySegment asOverlappingSlice(MemorySegment other)`

Returns a slice of this segment that is the overlap between this and the provided segment.

Two segments S1 and S2 are said to overlap if it is possible to find at least two slices L1 (from S1) and L2 (from S2) that are backed by the same memory region. As such, it is not possible for a `native` segment to overlap with a heap segment; in this case, or when no overlap occurs, `null` is returned.

Parameters:

`other` - the segment to test for an overlap with this segment.

Returns:

a slice of this segment, or `null` if no overlap occurs.

segmentOffset

`long segmentOffset(MemorySegment other)`

Returns the offset, in bytes, of the provided segment, relative to this segment.

The offset is relative to the base address of this segment and can be a negative or positive value. For instance, if both segments are native segments, the resulting offset can be computed as follows:

```
other.baseAddress().toRawLongValue() - segment.baseAddress().toRawLongValue()
```

If the segments share the same base address, `0` is returned. If `other` is a slice of this segment, the offset is always `0 <= x < this.byteSize()`.

Parameters:

`other` - the segment to retrieve an offset to.

Returns:

the relative offset, in bytes, of the provided 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(ValueLayout.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

`default MemorySegment 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`.

Calling this method is equivalent to the following code:

```
MemorySegment.copy(src, 0, this, 0, src.byteSize);
```

Parameters:

`src` - the source segment.

Returns:

this 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 smallest 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()
```

Determines whether the contents of this mapped segment is resident in physical memory.

A return value of `true` implies that it is highly likely that all 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 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 a heap-based segment that is not based on a `byte[]`), or if its size is greater than `Integer.MAX_VALUE`.

toArray

```
byte[] toArray(ValueLayout.OfByte elementLayout)
```

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

Parameters:

`elementLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

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`.

toArray

```
short[] toArray(ValueLayout.OfShort elementLayout)
```

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

Parameters:

`elementLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

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`

toArray

```
char[] toArray(ValueLayout.OfChar elementLayout)
```

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

Parameters:

`elementLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

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`.

toArray

```
int[] toArray(ValueLayout.OfInt elementLayout)
```

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

Parameters:

`elementLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

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`.

toArray

```
float[] toArray(ValueLayout.OfFloat elementLayout)
```

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

Parameters:

`elementLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

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`.

toArray

```
long[] toArray(ValueLayout.OfLong elementLayout)
```

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

Parameters:

`elementLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

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`.

toArray

```
double[] toArray(ValueLayout.OfDouble elementLayout)
```

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

Parameters:

`elementLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

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`.

getUtf8String

```
default String getUtf8String(long offset)
```

Reads a UTF-8 encoded, null-terminated string from this segment at given offset.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

a Java string constructed from the bytes read from the given starting address up to (but not including) the first `'\0'` terminator character (assuming one is found).

Throws:

`IllegalArgumentException` - if the size of the native string is greater than the largest string supported by the platform.

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

setUtf8String

```
default void setUtf8String(long offset,
                           String str)
```

Writes the given string into this segment at given offset, converting it to a null-terminated byte sequence using UTF-8 encoding.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`str` - the Java string to be written into this segment.

Throws:

`IllegalArgumentException` - if the size of the native string is greater than the largest string supported by the platform.

`IllegalStateException` - if the size of the native string is greater than the size of this segment, or 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.

ofByteBuffer

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

Creates a new 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 some other 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 array memory segment that models the memory associated with a given heap-allocated byte array. The returned segment is associated with 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 array memory segment that models the memory associated with a given heap-allocated char array. The returned segment is associated with 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 array memory segment that models the memory associated with a given heap-allocated short array. The returned segment is associated with 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 array memory segment that models the memory associated with a given heap-allocated int array. The returned segment is associated with 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 array memory segment that models the memory associated with a given heap-allocated float array. The returned segment is associated with 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 array memory segment that models the memory associated with a given heap-allocated long array. The returned segment is associated with 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 array memory segment that models the memory associated with a given heap-allocated double array. The returned segment is associated with the global resource scope.

Parameters:

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

Returns:

a new array memory segment.

ofAddress

```
static MemorySegment ofAddress(MemoryAddress address,
                               long bytesSize,
                               ResourceScope scope)
```

Creates a new native memory segment with given size and resource scope, and whose base address is the given address. This method can be useful when interacting with custom native memory sources (e.g. custom allocators), where an address to some underlying memory region is typically obtained from native code (often as a plain `long` value). The returned segment is not read-only (see `isReadOnly()`), and is associated with the provided resource scope.

Clients should ensure that the address and bounds refer to a valid region of memory that is accessible for reading and, if appropriate, writing; an attempt to access an invalid memory location from Java code will either return an arbitrary value, have no visible effect, or cause an unspecified exception to be thrown.

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.

Parameters:

`address` - the returned segment's base address.

`bytesSize` - the desired size.

`scope` - the native segment scope.

Returns:

a new native memory segment with given base address, size and scope.

Throws:

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

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

`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.

allocateNative

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

Creates a new 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 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 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 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 mapping mode might affect the behavior of the returned memory mapped segment (see `force()`).

`scope` - the segment scope.

Returns:

a new 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.

copy

```
static void copy(MemorySegment srcSegment,
                 long srcOffset,
                 MemorySegment dstSegment,
                 long dstOffset,
                 long bytes)
```

Performs a bulk copy from source segment to destination segment. More specifically, the bytes at offset `srcOffset` through `srcOffset + bytes - 1` in the source segment are copied into the destination segment at offset `dstOffset` through `dstOffset + bytes - 1`.

If the source segment overlaps with this segment, then the copying is performed as if the bytes at offset `srcOffset` through `srcOffset + bytes - 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 the destination segment at offset `dstOffset` through `dstOffset + bytes - 1`.

The result of a bulk copy is unspecified if, in the uncommon case, the source segment and the destination 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.

Calling this method is equivalent to the following code:

```
MemorySegment.copy(srcSegment, ValueLayout.JAVA_BYTE, srcOffset, dstSegment, ValueLayout.JAVA_BYTE, dstOf
```

Parameters:

`srcSegment` - the source segment.

`srcOffset` - the starting offset, in bytes, of the source segment.

`dstSegment` - the destination segment.

`dstOffset` - the starting offset, in bytes, of the destination segment.

`bytes` - the number of bytes to be copied.

Throws:

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

IndexOutOfBoundsException - if srcOffset + bytes > srcSegment.byteSize() or if dstOffset + bytes > dstSegment.byteSize(), or if either srcOffset, dstOffset or bytes are < 0.

UnsupportedOperationException - if the destination segment is read-only (see isReadOnly()).

copy

```
static void copy(MemorySegment srcSegment,
                 ValueLayout srcElementLayout,
                 long srcOffset,
                 MemorySegment dstSegment,
                 ValueLayout dstElementLayout,
                 long dstOffset,
                 long elementCount)
```

Performs a bulk copy from source segment to destination segment. More specifically, if S is the byte size of the element layouts, the bytes at offset srcOffset through srcOffset + (elementCount * S) - 1 in the source segment are copied into the destination segment at offset dstOffset through dstOffset + (elementCount * S) - 1.

The copy occurs in an element-wise fashion: the bytes in the source segment are interpreted as a sequence of elements whose layout is srcElementLayout, whereas the bytes in the destination segment are interpreted as a sequence of elements whose layout is dstElementLayout. Both element layouts must have same size S. If the byte order of the two element layouts differ, the bytes corresponding to each element to be copied are swapped accordingly during the copy operation.

If the source segment overlaps with this segment, then the copying is performed as if the bytes at offset srcOffset through srcOffset + (elementCount * S) - 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 the destination segment at offset dstOffset through dstOffset + (elementCount * S) - 1.

The result of a bulk copy is unspecified if, in the uncommon case, the source segment and the destination 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:

srcSegment - the source segment.

srcElementLayout - the element layout associated with the source segment.

srcOffset - the starting offset, in bytes, of the source segment.

dstSegment - the destination segment.

dstElementLayout - the element layout associated with the destination segment.

dstOffset - the starting offset, in bytes, of the destination segment.

elementCount - the number of elements to be copied.

Throws:

IllegalArgumentException - if the element layouts have different sizes, if the source (resp. destination) segment/offset are incompatible with the alignment constraints in the source (resp. destination) element layout, or if the source (resp. destination) element layout alignment is greater than its size.

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

IndexOutOfBoundsException - if srcOffset + (elementCount * S) > srcSegment.byteSize() or if dstOffset + (elementCount * S) > dstSegment.byteSize(), where S is the byte size of the element layouts, or if either srcOffset, dstOffset or elementCount are < 0.

UnsupportedOperationException - if the destination segment is read-only (see isReadOnly()).

get

```
default byte get(ValueLayout.OfByte layout,
                 long offset)
```

Reads a byte from this segment and offset with given layout.

Parameters:

layout - the layout of the memory region to be read.

offset - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as address().toRowLongValue() + offset.

Returns:

a byte value read from this address.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfByte layout,
                 long offset,
                 byte value)
```

Writes a byte to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the byte value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default boolean get(ValueLayout.OfBoolean layout,
                    long offset)
```

Reads a boolean from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

a boolean value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfBoolean layout,
                 long offset,
                 boolean value)
```

Writes a boolean to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the boolean value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default char get(ValueLayout.OfChar layout,
                 long offset)
```

Reads a char from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

a char value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfChar layout,
                 long offset,
                 char value)
```

Writes a char to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the char value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default short get(ValueLayout.OfShort layout,
                  long offset)
```

Reads a short from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

a short value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfShort layout,
                 long offset,
                 short value)
```

Writes a short to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the short value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default int get(ValueLayout.OfInt layout,
                long offset)
```

Reads an int from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

an int value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfInt layout,
                 long offset,
                 int value)
```

Writes an int to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the int value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default float get(ValueLayout.OfFloat layout,
                  long offset)
```

Reads a float from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

a float value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfFloat layout,
                 long offset,
                 float value)
```

Writes a float to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the float value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default long get(ValueLayout.OfLong layout,
                 long offset)
```

Reads a long from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

a long value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfLong layout,
                 long offset,
                 long value)
```

Writes a long to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the long value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default double get(ValueLayout.OfDouble layout,
                   long offset)
```

Reads a double from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

a double value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfDouble layout,
                 long offset,
                 double value)
```

Writes a double to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the double value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

get

```
default MemoryAddress get(ValueLayout.OfAddress layout,
                          long offset)
```

Reads an address from this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be read.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + offset`.

Returns:

an address value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

set

```
default void set(ValueLayout.OfAddress layout,
                 long offset,
                 Addressable value)
```

Writes an address to this segment and offset with given layout.

Parameters:

`layout` - the layout of the memory region to be written.

`offset` - offset in bytes (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + offset`.

`value` - the address value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

getAtIndex

```
default char getAtIndex(ValueLayout.OfChar layout,
                        long index)
```

Reads a char from this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be read.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

Returns:

a char value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

setAtIndex

```
default void setAtIndex(ValueLayout.OfChar layout,
                        long index,
                        char value)
```

Writes a char to this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be written.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

`value` - the char value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

getAtIndex

```
default short getAtIndex(ValueLayout.OfShort layout,
                         long index)
```

Reads a short from this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be read.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

Returns:

a short value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

setAtIndex

```
default void setAtIndex(ValueLayout.OfShort layout,
                        long index,
                        short value)
```

Writes a short to this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be written.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

`value` - the short value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

getAtIndex

```
default int getAtIndex(ValueLayout.OfInt layout,
                       long index)
```

Reads an int from this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be read.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

Returns:

an int value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

setAtIndex

```
default void setAtIndex(ValueLayout.OfInt layout,
                        long index,
                        int value)
```

Writes an int to this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be written.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

`value` - the int value to be written.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

UnsupportedOperationException - if this segment is read-only.

getAtIndex

```
default float getAtIndex(ValueLayout.OfFloat layout,
                         long index)
```

Reads a float from this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be read.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

Returns:

a float value read from this address.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

setAtIndex

```
default void setAtIndex(ValueLayout.OfFloat layout,
                        long index,
                        float value)
```

Writes a float to this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be written.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

`value` - the float value to be written.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

UnsupportedOperationException - if this segment is read-only.

getAtIndex

```
default long getAtIndex(ValueLayout.OfLong layout,
                        long index)
```

Reads a long from this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be read.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

Returns:

a long value read from this address.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

setAtIndex

```
default void setAtIndex(ValueLayout.OfLong layout,
                        long index,
                        long value)
```

Writes a long to this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be written.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

`value` - the long value to be written.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

UnsupportedOperationException - if this segment is read-only.

getAtIndex

```
default double getAtIndex(ValueLayout.OfDouble layout,
                          long index)
```

Reads a double from this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be read.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

Returns:

a double value read from this address.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

setAtIndex

```
default void setAtIndex(ValueLayout.OfDouble layout,
                        long index,
                        double value)
```

Writes a double to this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be written.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

`value` - the double value to be written.

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.

IllegalArgumentException - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

IndexOutOfBoundsException - when the dereference operation falls outside the *spatial bounds* of the memory segment.

UnsupportedOperationException - if this segment is read-only.

getAtIndex

```
default MemoryAddress getAtIndex(ValueLayout.OfAddress layout,
                                 long index)
```

Reads an address from this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be read.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this read operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

Returns:

an address value read from this address.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

setAtIndex

```
default void setAtIndex(ValueLayout.OfAddress layout,
                        long index,
                        Addressable value)
```

Writes an address to this segment and index, scaled by given layout size.

Parameters:

`layout` - the layout of the memory region to be written.

`index` - index (relative to this segment). For instance, if this segment is a native segment, the final address of this write operation can be expressed as `address().toRowLongValue() + (index * layout.byteSize())`.

`value` - the address value to be written.

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.

`IllegalArgumentException` - if the dereference operation is incompatible with the alignment constraints in the provided layout, or if the layout alignment is greater than its size.

`IndexOutOfBoundsException` - when the dereference operation falls outside the *spatial bounds* of the memory segment.

`UnsupportedOperationException` - if this segment is read-only.

copy

```
static void copy(MemorySegment srcSegment,
                 ValueLayout srcLayout,
                 long srcOffset,
                 Object dstArray,
                 int dstIndex,
                 int elementCount)
```

Copies a number of elements from a source segment to a destination array, starting at a given segment offset (expressed in bytes), and a given array index, using the given source element layout. Supported array types are `byte[]`, `char[]`, `short[]`, `int[]`, `float[]`, `long[]` and `double[]`.

Parameters:

`srcSegment` - the source segment.

`srcLayout` - the source element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

`srcOffset` - the starting offset, in bytes, of the source segment.

`dstArray` - the destination array.

`dstIndex` - the starting index of the destination array.

`elementCount` - the number of array elements to be copied.

Throws:

`IllegalArgumentException` - if `dstArray` is not an array, or if it is an array but whose type is not supported, if the destination array component type does not match the carrier of the source element layout, if the source segment/offset are incompatible with the alignment constraints in the source element layout, or if the destination element layout alignment is greater than its size.

copy

```
static void copy(Object srcArray,
                 int srcIndex,
                 MemorySegment dstSegment,
                 ValueLayout dstLayout,
                 long dstOffset,
                 int elementCount)
```

Copies a number of elements from a source array to a destination segment, starting at a given array index, and a given segment offset (expressed in bytes), using the given destination element layout. Supported array types are `byte[]`, `char[]`, `short[]`, `int[]`, `float[]`, `long[]` and `double[]`.

Parameters:

`srcArray` - the source array.

`srcIndex` - the starting index of the source array.

`dstSegment` - the destination segment.

`dstLayout` - the destination element layout. If the byte order associated with the layout is different from the native order, a byte swap operation will be performed on each array element.

`dstOffset` - the starting offset, in bytes, of the destination segment.

`elementCount` - the number of array elements to be copied.

Throws:

`IllegalArgumentException` - if `srcArray` is not an array, or if it is an array but whose type is not supported, if the source array component type does not match the carrier of the destination element layout, if the destination segment/offset are incompatible with the alignment constraints in the destination element layout, or if the destination element layout alignment is greater than its size.