Module java.base
Package java.lang.foreign

Interface MemorySegment

All Superinterfaces:

Addressable[PREVIEW]

```
public sealed interface MemorySegment
extends Addressable[PREVIEW]
```

> **MemorySegment is a preview API of the Java platform.**
> *Programs can only use `MemorySegment` when preview features are enabled.*
> *Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.*

A memory segment models a contiguous region of memory. A memory segment is associated with both spatial and temporal bounds (e.g. a `MemorySession`[PREVIEW]). 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 memory session associated with a memory segment has been closed (see `MemorySession.close()`[PREVIEW]). There are many kinds of memory segments:

- native memory segments, backed by off-heap memory;
- mapped memory segments[PREVIEW], obtained by mapping a file into main memory (`mmap`); the contents of a mapped memory segments can be persisted and loaded to and from the underlying memory-mapped file;
- array segments, wrapping an existing, heap-allocated Java array; and
- buffer segments, wrapping an existing `Buffer` instance; buffer memory segments might be backed by either off-heap memory or on-heap memory, depending on the characteristics of the wrapped buffer instance. For instance, a buffer memory segment obtained from a byte buffer created with the `ByteBuffer.allocateDirect(int)` method will be backed by off-heap memory.

Lifecycle and confinement

Memory segments are associated with a memory session. As for all resources associated with a memory session, a segment cannot be accessed after its underlying session has been closed. For instance, the following code will result in an exception:

```
MemorySegment segment = null;
try (MemorySession session = MemorySession.openConfined()) {
    segment = MemorySegment.allocateNative(8, session);
}
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 memory session; that is, if the segment is associated with a shared session, it can be accessed by multiple threads; if it is associated with a confined session, it can only be accessed by the thread which owns the memory session.

Heap segments are always associated with the global[PREVIEW] memory session. This session cannot be closed, and segments associated with it can be considered as *always alive*. Buffer segments are typically associated with the global memory session, with one exception: buffer segments created from byte buffer instances obtained calling the `asByteBuffer()` method on a memory segment `S` are associated with the same memory session as `S`.

Dereferencing memory segments

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[PREVIEW], 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 segment view var handle[PREVIEW], that is, a var handle that accepts a segment and a `long` offset. More complex access var handles can be obtained by adapting a segment var handle view using the var handle combinator functions defined in the `MethodHandles` class:

```
MemorySegment segment = ...
VarHandle intHandle = MethodHandles.memorySegmentViewVarHandle(ValueLayout.JAVA_INT);
MethodHandle multiplyExact = MethodHandles.lookup()
                                          .findStatic(Math.class, "multiplyExact",
                                                      MethodType.methodType(long.class, long.class
intHandle = MethodHandles.filterCoordinates(intHandle, 1,
```

```
                                                MethodHandles.insertArguments(multiplyExact, 0, 4L));
```

Alternatively, complex access var handles can can be obtained from memory layouts[PREVIEW] by providing a so called *layout path*:

```
MemorySegment segment = ...
VarHandle intHandle = ValueLayout.JAVA_INT.arrayElementVarHandle();
intHandle.get(segment, 3L); // get int element at offset 3 * 4 = 12
```

Memory segments support *slicing*. A memory segment can be used to obtain other segments backed by the same underlying memory region, but with *stricter* spatial bounds than the ones of the original segment:

```
MemorySession session = ...
MemorySegment segment = MemorySegment.allocateNative(100, session);
MemorySegment slice = segment.asSlice(50, 10);
slice.get(ValueLayout.JAVA_INT, 20); // Out of bounds!
session.close();
slice.get(ValueLayout.JAVA_INT, 0); // Already closed!
```

The above code creates a native segment that is 100 bytes long; then, it creates a slice that starts at offset 50 of `segment`, and is 10 bytes long. As a result, attempting to read an int value at offset 20 of the `slice` segment will result in an exception. The temporal bounds[PREVIEW] of the original segment are inherited by its slices; that is, when the memory session associated with `segment` is closed, `slice` will also be become inaccessible.

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 memory session). The following code can be used to sum all int values in a memory segment in parallel:

```
try (MemorySession session = MemorySession.openShared()) {
    SequenceLayout SEQUENCE_LAYOUT = MemoryLayout.sequenceLayout(1024, ValueLayout.JAVA_INT);
    MemorySegment segment = MemorySegment.allocateNative(SEQUENCE_LAYOUT, session);
    int sum = segment.elements(ValueLayout.JAVA_INT).parallel()
                     .mapToInt(s -> s.get(ValueLayout.JAVA_INT, 0))
                     .sum();
}
```

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[PREVIEW]. 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.

Restricted memory 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 session[PREVIEW]. 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.

Clients requiring sophisticated, low-level control over mapped memory segments, might consider writing custom mapped memory segment factories; using `Linker`[PREVIEW], 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)`[PREVIEW] and `ofAddress(MemoryAddress, long, MemorySession)`.

Implementation Requirements:

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

Since:

19

Method Summary

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

Modifier and Type	Method	Description
MemoryAddress[PREVIEW]	address()	Returns the base memory address associated with this native memory segment.
static MemorySegment[PREVIEW]	allocateNative(long bytesSize, long alignmentBytes, MemorySession[PREVIEW] session)	Creates a native memory segment with the given size (in bytes), alignment constraint (in bytes) and memory session.
static MemorySegment[PREVIEW]	allocateNative(long bytesSize, MemorySession[PREVIEW] session)	Creates a native memory segment with the given size (in bytes) and memory session.
static MemorySegment[PREVIEW]	allocateNative (MemoryLayout[PREVIEW] layout, MemorySession[PREVIEW] session)	Creates a native memory segment with the given layout and memory session.
ByteBuffer	asByteBuffer()	Wraps this segment in a ByteBuffer.
Optional<MemorySegment[PREVIEW]>	asOverlappingSlice (MemorySegment[PREVIEW] other)	Returns a slice of this segment that is the overlap between this and the provided segment.
MemorySegment[PREVIEW]	asReadOnly()	Returns a read-only view of this segment.
default MemorySegment[PREVIEW]	asSlice(long offset)	Returns a slice of this memory segment, at the given offset.
MemorySegment[PREVIEW]	asSlice(long offset, long newSize)	Returns a slice of this memory segment, at the given offset.
long	byteSize()	Returns the size (in bytes) of this memory segment.
static void	copy(MemorySegment[PREVIEW] srcSegment, long srcOffset, MemorySegment[PREVIEW] dstSegment, long dstOffset, long bytes)	Performs a bulk copy from source segment to destination segment.
static void	copy(MemorySegment[PREVIEW] srcSegment, ValueLayout[PREVIEW] srcElementLayout, long srcOffset, MemorySegment[PREVIEW] dstSegment, ValueLayout[PREVIEW] dstElementLayout, long dstOffset, long elementCount)	Performs a bulk copy from source segment to destination segment.
static void	copy(MemorySegment[PREVIEW] srcSegment, ValueLayout[PREVIEW] srcLayout, long srcOffset, Object dstArray, int dstIndex, int elementCount)	Copies a number of elements from a source memory segment to a destination array.
static void	copy(Object srcArray, int srcIndex, MemorySegment[PREVIEW] dstSegment, ValueLayout[PREVIEW] dstLayout, long dstOffset, int elementCount)	Copies a number of elements from a source array to a destination memory segment.
default MemorySegment[PREVIEW]	copyFrom(MemorySegment[PREVIEW] src)	Performs a bulk copy from given source segment to this segment.
Stream<MemorySegment[PREVIEW]>	elements (MemoryLayout[PREVIEW] elementLayout)	Returns a sequential Stream over disjoint slices (whose size matches that of the specified layout) in this segment.
boolean	equals(Object that)	Compares the specified object with this memory segment for equality.
MemorySegment[PREVIEW]	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**[PREVIEW]	**get**(**ValueLayout.OfAddress**[PREVIEW] layout, long offset)	Reads an address from this segment at the given offset, with the given layout.
default boolean	**get**(**ValueLayout.OfBoolean**[PREVIEW] layout, long offset)	Reads a boolean from this segment at the given offset, with the given layout.
default byte	**get**(**ValueLayout.OfByte**[PREVIEW] layout, long offset)	Reads a byte from this segment at the given offset, with the given layout.
default char	**get**(**ValueLayout.OfChar**[PREVIEW] layout, long offset)	Reads a char from this segment at the given offset, with the given layout.
default double	**get**(**ValueLayout.OfDouble**[PREVIEW] layout, long offset)	Reads a double from this segment at the given offset, with the given layout.
default float	**get**(**ValueLayout.OfFloat**[PREVIEW] layout, long offset)	Reads a float from this segment at the given offset, with the given layout.
default int	**get**(**ValueLayout.OfInt**[PREVIEW] layout, long offset)	Reads an int from this segment at the given offset, with the given layout.
default long	**get**(**ValueLayout.OfLong**[PREVIEW] layout, long offset)	Reads a long from this segment at the given offset, with the given layout.
default short	**get**(**ValueLayout.OfShort**[PREVIEW] layout, long offset)	Reads a short from this segment at the given offset, with the given layout.
default **MemoryAddress**[PREVIEW]	**getAtIndex**(**ValueLayout.OfAddress**[PREVIEW] layout, long index)	Reads an address from this segment at the given index, scaled by the given layout size.
default char	**getAtIndex**(**ValueLayout.OfChar**[PREVIEW] layout, long index)	Reads a char from this segment at the given index, scaled by the given layout size.
default double	**getAtIndex**(**ValueLayout.OfDouble**[PREVIEW] layout, long index)	Reads a double from this segment at the given index, scaled by the given layout size.
default float	**getAtIndex**(**ValueLayout.OfFloat**[PREVIEW] layout, long index)	Reads a float from this segment at the given index, scaled by the given layout size.
default int	**getAtIndex**(**ValueLayout.OfInt**[PREVIEW] layout, long index)	Reads an int from this segment at the given index, scaled by the given layout size.
default long	**getAtIndex**(**ValueLayout.OfLong**[PREVIEW] layout, long index)	Reads a long from this segment at the given index, scaled by the given layout size.
default short	**getAtIndex**(**ValueLayout.OfShort**[PREVIEW] layout, long index)	Reads a short from this segment at the given index, scaled by the given layout size.
default **String**	**getUtf8String**(long offset)	Reads a UTF-8 encoded, null-terminated string from this segment at the given offset.
int	**hashCode**()	Returns the hash code value for this memory segment.
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.

long	**mismatch**(**MemorySegment**[PREVIEW] other)	Finds and returns the offset, in bytes, of the first mismatch between this segment and the given other segment.
static **MemorySegment**[PREVIEW]	**ofAddress**(**MemoryAddress**[PREVIEW] address, long bytesSize, **MemorySession**[PREVIEW] session)	Creates a native memory segment with the given size, base address, and memory session.
static **MemorySegment**[PREVIEW]	**ofArray**(byte[] arr)	Creates an array memory segment that models the memory associated with the given heap-allocated byte array.
static **MemorySegment**[PREVIEW]	**ofArray**(char[] arr)	Creates an array memory segment that models the memory associated with the given heap-allocated char array.
static **MemorySegment**[PREVIEW]	**ofArray**(double[] arr)	Creates an array memory segment that models the memory associated with the given heap-allocated double array.
static **MemorySegment**[PREVIEW]	**ofArray**(float[] arr)	Creates an array memory segment that models the memory associated with the given heap-allocated float array.
static **MemorySegment**[PREVIEW]	**ofArray**(int[] arr)	Creates an array memory segment that models the memory associated with the given heap-allocated int array.
static **MemorySegment**[PREVIEW]	**ofArray**(long[] arr)	Creates an array memory segment that models the memory associated with the given heap-allocated long array.
static **MemorySegment**[PREVIEW]	**ofArray**(short[] arr)	Creates an array memory segment that models the memory associated with the given heap-allocated short array.
static **MemorySegment**[PREVIEW]	**ofBuffer**(**Buffer** buffer)	Creates a buffer memory segment that models the memory associated with the given `Buffer` instance.
long	**segmentOffset** (**MemorySegment**[PREVIEW] other)	Returns the offset, in bytes, of the provided segment, relative to this segment.
MemorySession[PREVIEW]	**session**()	Returns the memory session associated with this memory segment.
default void	**set**(**ValueLayout.OfAddress**[PREVIEW] layout, long offset, **Addressable**[PREVIEW] value)	Writes an address into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfBoolean**[PREVIEW] layout, long offset, boolean value)	Writes a boolean into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfByte**[PREVIEW] layout, long offset, byte value)	Writes a byte into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfChar**[PREVIEW] layout, long offset, char value)	Writes a char into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfDouble**[PREVIEW] layout, long offset, double value)	Writes a double into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfFloat**[PREVIEW] layout, long offset, float value)	Writes a float into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfInt**[PREVIEW] layout, long offset, int value)	Writes an int into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfLong**[PREVIEW] layout, long offset, long value)	Writes a long into this segment at the given offset, with the given layout.
default void	**set**(**ValueLayout.OfShort**[PREVIEW] layout, long offset, short value)	Writes a short into this segment at the given offset, with the given layout.
default void	**setAtIndex** (**ValueLayout.OfAddress**[PREVIEW] layout, long index, **Addressable**[PREVIEW] value)	Writes an address into this segment at the given index, scaled by the given layout size.
default void	**setAtIndex** (**ValueLayout.OfChar**[PREVIEW] layout, long index, char value)	Writes a char into this segment at the given index, scaled by the given layout size.

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

Method Details

address

MemoryAddress[PREVIEW] address()

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

Specified by:

address in interface Addressable[PREVIEW]

Returns:

the base memory address associated with this native memory segment

Throws:

UnsupportedOperationException - if this segment is not a native segment.

IllegalStateException - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

spliterator

`Spliterator<MemorySegment`[PREVIEW]`> spliterator(MemoryLayout`[PREVIEW] `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 memory session 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`[PREVIEW]`> elements(MemoryLayout`[PREVIEW] `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.

session

`MemorySession`[PREVIEW] `session()`

Returns the memory session associated with this memory segment.

Returns:

the memory session 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[PREVIEW] asSlice(long offset,
                               long newSize)
```

Returns a slice of this memory segment, at the given offset. The returned segment's base address is the base address of this segment plus the given offset; its 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 slice of this memory segment.

Throws:

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

See Also:

`asSlice(long)`

asSlice

`default MemorySegment`[PREVIEW] `asSlice(long offset)`

Returns a slice of this memory segment, at the given offset. The returned segment's base address is the base address of this segment plus the given offset; its 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 slice of this memory segment.

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`[PREVIEW] `asReadOnly()`

Returns 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, MemorySession)` (and related) factory, or a buffer segment derived from a direct byte buffer using the `ofBuffer(Buffer)` 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 `FileChannel.map(FileChannel.MapMode, long, long, MemorySession)`[PREVIEW] factory, or a buffer segment derived from a `MappedByteBuffer` using the `ofBuffer(Buffer)` factory.

Returns:

`true` if this segment is a mapped segment.

asOverlappingSlice

`Optional<MemorySegment`[PREVIEW]`> asOverlappingSlice(MemorySegment`[PREVIEW] ` 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 (where overlapping occurs).

segmentOffset

`long segmentOffset(MemorySegment`[PREVIEW] ` 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`[PREVIEW] ` 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 session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

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

copyFrom

`default MemorySegment[PREVIEW] copyFrom(MemorySegment[PREVIEW] 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 the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - if the session associated with `src` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with `src`.

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

mismatch

`long mismatch(MemorySegment[PREVIEW] other)`

Finds and returns the offset, in bytes, of the first mismatch between this segment and the 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 the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - if the session associated with `other` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with `other`.

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 session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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 session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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 session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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 session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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 memory session associated with this segment has been closed (see `MemorySession.close()`[PREVIEW]), will throw an `IllegalStateException`. Similarly, accessing the returned buffer from a thread other than the thread owning[PREVIEW] this segment's memory session will throw a `WrongThreadException`.

If this segment is associated with a confined memory session, 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`[PREVIEW] ` elementLayout)`

Copy the contents of this memory segment into a new 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 new byte array whose contents are copied from this memory segment.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - 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`[PREVIEW] ` elementLayout)`

Copy the contents of this memory segment into a new 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 new short array whose contents are copied from this memory segment.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - 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`[PREVIEW] ` elementLayout)`

Copy the contents of this memory segment into a new 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 new char array whose contents are copied from this memory segment.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - 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`[PREVIEW] ` elementLayout)`

Copy the contents of this memory segment into a new 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 new int array whose contents are copied from this memory segment.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - 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`[PREVIEW] ` elementLayout)`

Copy the contents of this memory segment into a new 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 new float array whose contents are copied from this memory segment.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - 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`[PREVIEW] ` elementLayout)`

Copy the contents of this memory segment into a new 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 new long array whose contents are copied from this memory segment.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - 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`[PREVIEW] ` elementLayout)`

Copy the contents of this memory segment into a new 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 new double array whose contents are copied from this memory segment.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`IllegalStateException` - 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 the 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().toRawLongValue() + 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 UTF-8 string is greater than the largest string supported by the platform.

`IndexOutOfBoundsException` - if `S + offset > byteSize()`, where `S` is the size of the UTF-8 string (including the terminator character).

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

setUtf8String

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

Writes the given string into this segment at the 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.

If the given string contains any `'\0'` characters, they will be copied as well. This means that, depending on the method used to read the string, such as `getUtf8String(long)`, the string will appear truncated when read again.

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().toRawLongValue() + offset`.

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

Throws:

`IndexOutOfBoundsException` - if `str.getBytes().length() + offset >= byteSize()`.

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

ofBuffer

```
static MemorySegment[PREVIEW] ofBuffer(Buffer buffer)
```

Creates a buffer memory segment that models the memory associated with the given `Buffer` instance. 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 memory session associated with this segment can either be the global[PREVIEW] memory session, in case the buffer has been created independently, or some other memory session, in case the buffer has been obtained using `asByteBuffer()`.

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

Parameters:

`buffer` - the buffer instance backing the buffer memory segment.

Returns:

a buffer memory segment.

ofArray

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

Creates an array memory segment that models the memory associated with the given heap-allocated byte array. The returned segment is associated with the global[PREVIEW] memory session.

Parameters:

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

Returns:

an array memory segment.

ofArray

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

Creates an array memory segment that models the memory associated with the given heap-allocated char array. The returned segment is associated with the global[PREVIEW] memory session.

Parameters:

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

Returns:

an array memory segment.

ofArray

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

Creates an array memory segment that models the memory associated with the given heap-allocated short array. The returned segment is associated with the global[PREVIEW] memory session.

Parameters:

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

Returns:

an array memory segment.

ofArray

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

Creates an array memory segment that models the memory associated with the given heap-allocated int array. The returned segment is associated with the global[PREVIEW] memory session.

Parameters:

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

Returns:

an array memory segment.

ofArray

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

Creates an array memory segment that models the memory associated with the given heap-allocated float array. The returned segment is associated with the global[PREVIEW] memory session.

Parameters:

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

Returns:

an array memory segment.

ofArray

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

Creates an array memory segment that models the memory associated with the given heap-allocated long array. The returned segment is associated with the global[PREVIEW] memory session.

Parameters:

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

Returns:

an array memory segment.

ofArray

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

Creates an array memory segment that models the memory associated with the given heap-allocated double array. The returned segment is associated with the global[PREVIEW] memory session.

Parameters:

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

Returns:

an array memory segment.

ofAddress

```
static MemorySegment[PREVIEW] ofAddress(MemoryAddress[PREVIEW] address,
                                        long bytesSize,
                                        MemorySession[PREVIEW] session)
```

Creates a native memory segment with the given size, base address, and memory session. This method can be useful when interacting with custom memory sources (e.g. custom allocators), where an address to some underlying memory region is typically obtained from foreign code (often as a plain `long` value).

The returned segment is not read-only (see `isReadOnly()`), and is associated with the provided memory session.

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.

`session` - the native segment memory session.

Returns:

a native memory segment with the given base address, size and memory session.

Throws:

`IllegalArgumentException` - if `bytesSize < 0`.

`IllegalStateException` - if `session` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning[PREVIEW] `session`.

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

allocateNative

```
static MemorySegment[PREVIEW] allocateNative(MemoryLayout[PREVIEW] layout,
                                             MemorySession[PREVIEW] session)
```

Creates a native memory segment with the given layout and memory session. A client is responsible for ensuring that the memory session 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(), session);
```

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.

`session` - the segment memory session.

Returns:

a new native memory segment.

Throws:

`IllegalStateException` - if `session` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning[PREVIEW] `session`.

allocateNative

```
static MemorySegment[PREVIEW] allocateNative(long bytesSize,
                                             MemorySession[PREVIEW] session)
```

Creates a native memory segment with the given size (in bytes) and memory session. A client is responsible for ensuring that the memory session 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, session);
```

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.

`session` - the segment temporal bounds.

Returns:

a new native memory segment.

Throws:

`IllegalArgumentException` - if `bytesSize < 0`.

`IllegalStateException` - if `session` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning[PREVIEW] `session`.

allocateNative

```
static MemorySegment[PREVIEW] allocateNative(long bytesSize,
                                             long alignmentBytes,
                                             MemorySession[PREVIEW] session)
```

Creates a native memory segment with the given size (in bytes), alignment constraint (in bytes) and memory session. A client is responsible for ensuring that the memory session 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.

`session` - the segment memory session.

Returns:

a new native memory segment.

Throws:

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

`IllegalStateException` - if `session` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning[PREVIEW] `session`.

copy

```
static void copy(MemorySegment[PREVIEW] srcSegment,
                 long srcOffset,
                 MemorySegment[PREVIEW] 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 the session associated with `srcSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with `srcSegment`.

`IllegalStateException` - if the session associated with `dstSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with `dstSegment`.

`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[PREVIEW] srcSegment,
                 ValueLayout[PREVIEW] srcElementLayout,
                 long srcOffset,
                 MemorySegment[PREVIEW] dstSegment,
                 ValueLayout[PREVIEW] 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 the session associated with `srcSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this `srcSegment`.

`IllegalStateException` - if the session associated with `dstSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with `dstSegment`.

`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[PREVIEW] layout,
                 long offset)
```

Reads a byte from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

a byte value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 byte value)
```

Writes a byte into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the byte value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                    long offset)
```

Reads a boolean from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

a boolean value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 boolean value)
```

Writes a boolean into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the boolean value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset)
```

Reads a char from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

a char value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 char value)
```

Writes a char into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the char value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                  long offset)
```

Reads a short from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

a short value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 short value)
```

Writes a short into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the short value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                long offset)
```

Reads an int from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

an int value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 int value)
```

Writes an int into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the int value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                  long offset)
```

Reads a float from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

a float value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 float value)
```

Writes a float into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the float value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset)
```

Reads a long from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

a long value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 long value)
```

Writes a long into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the long value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

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

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

get

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

Reads a double from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

a double value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 double value)
```

Writes a double into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the double value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] get(ValueLayout.OfAddress[PREVIEW] layout,
                                   long offset)
```

Reads an address from this segment at the given offset, with the 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().toRawLongValue() + offset`.

Returns:

an address value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                 long offset,
                 Addressable[PREVIEW] value)
```

Writes an address into this segment at the given offset, with the 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().toRawLongValue() + offset`.

`value` - the address value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index)
```

Reads a char from this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

Returns:

a char value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index,
                        char value)
```

Writes a char into this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

`value` - the char value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                         long index)
```

Reads a short from this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

Returns:

a short value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index,
                        short value)
```

Writes a short into this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

`value` - the short value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                       long index)
```

Reads an int from this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

Returns:

an int value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index,
                        int value)
```

Writes an int into this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

`value` - the int value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                         long index)
```

Reads a float from this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

Returns:

a float value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index,
                        float value)
```

Writes a float into this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

`value` - the float value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index)
```

Reads a long from this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

Returns:

a long value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index,
                        long value)
```

Writes a long into this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

`value` - the long value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                          long index)
```

Reads a double from this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

Returns:

a double value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index,
                        double value)
```

Writes a double into this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

`value` - the double value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] getAtIndex(ValueLayout.OfAddress[PREVIEW] layout,
                                          long index)
```

Reads an address from this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

Returns:

an address value read from this address.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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[PREVIEW] layout,
                        long index,
                        Addressable[PREVIEW] value)
```

Writes an address into this segment at the given index, scaled by the 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().toRawLongValue() + (index * layout.byteSize())`.

`value` - the address value to be written.

Throws:

`IllegalStateException` - if the session associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with this segment.

`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.

equals

```
boolean equals(Object that)
```

Compares the specified object with this memory segment for equality. Returns `true` if and only if the specified object is also a memory segment, and if that segment refers to the same memory region as this segment. More specifically, for two segments to be considered equals, all the following must be true:

- the two segments must be of the same kind; either both are native segments, backed by off-heap memory, or both are backed by on-heap memory;
- if the two segments are native segments, their `base address` must be equal[PREVIEW]. Otherwise, the two segments must wrap the same Java array instance, at the same starting offset;
- the two segments must have the same size; and
- the two segments must have the same[PREVIEW] temporal bounds.

Overrides:

`equals` in class `Object`

API Note:

This method does not perform a structural comparison of the contents of the two memory segments. Clients can compare memory segments structurally by using the `mismatch(MemorySegment)` method instead.

Parameters:

`that` - the object to be compared for equality with this memory segment.

Returns:

`true` if the specified object is equal to this memory segment.

See Also:

`mismatch(MemorySegment)`,
`asOverlappingSlice(MemorySegment)`

hashCode

```
int hashCode()
```

Returns the hash code value for this memory segment.

Overrides:

`hashCode` in class `Object`

Returns:

the hash code value for this memory segment

See Also:

`Object.equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

copy

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

Copies a number of elements from a source memory segment to a destination array. The elements, whose size and alignment constraints are specified by the given layout, are read from the source segment, starting at the given offset (expressed in bytes), and are copied into the destination array, at the given index. 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:

`IllegalStateException` - if the session associated with `srcSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with `srcSegment`.

`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[PREVIEW] dstSegment,
                 ValueLayout[PREVIEW] dstLayout,
                 long dstOffset,
                 int elementCount)
```

Copies a number of elements from a source array to a destination memory segment. The elements, whose size and alignment constraints are specified by the given layout, are read from the source array, starting at the given index, and are copied into the destination segment, at the given offset (expressed in bytes). 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:

`IllegalStateException` - if the session associated with `dstSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread other than the thread owning the session associated with `dstSegment`.

`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.
