Module java.base
Package java.lang.foreign

Interface MemorySegment

```
public sealed interface MemorySegment
```

`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 provides access to a contiguous region of memory.

There are two kinds of memory segments:

- A *heap segment* is backed by, and provides access to, a region of memory inside the Java heap (an "on-heap" region).
- A *native segment* is backed by, and provides access to, a region of memory outside the Java heap (an "off-heap" region).

Heap segments can be obtained by calling one of the `ofArray(int[])` factory methods. These methods return a memory segment backed by the on-heap region that holds the specified Java array.

Native segments can be obtained by calling one of the `Arena.allocate(long, long)`[PREVIEW] factory methods, which return a memory segment backed by a newly allocated off-heap region with the given size and aligned to the given alignment constraint. Alternatively, native segments can be obtained by `mapping`[PREVIEW] a file into a new off-heap region (in some systems, this operation is sometimes referred to as `mmap`). Segments obtained in this way are called *mapped* segments, and their contents can be persisted and loaded to and from the underlying memory-mapped file.

Both kinds of segments are read and written using the same methods, known as access operations. An access operation on a memory segment always and only provides access to the region for which the segment was obtained.

Characteristics of memory segments

Every memory segment has an address, expressed as a `long` value. The nature of a segment's address depends on the kind of the segment:

- The address of a heap segment is not a physical address, but rather an offset within the region of memory which backs the segment. The region is inside the Java heap, so garbage collection might cause the region to be relocated in physical memory over time, but this is not exposed to clients of the `MemorySegment` API who see a stable *virtualized* address for a heap segment backed by the region. A heap segment obtained from one of the `ofArray(int[])` factory methods has an address of zero.
- The address of a native segment (including mapped segments) denotes the physical address of the region of memory which backs the segment.

Every memory segment has a size. The size of a heap segment is derived from the Java array from which it is obtained. This size is predictable across Java runtimes. The size of a native segment is either passed explicitly (as in `Arena.allocate(long, long)`[PREVIEW]) or derived from a `MemoryLayout`[PREVIEW] (as in `SegmentAllocator.allocate(MemoryLayout)`[PREVIEW]). The size of a memory segment is typically a positive number but may be zero, but never negative.

The address and size of a memory segment jointly ensure that access operations on the segment cannot fall *outside* the boundaries of the region of memory which backs the segment. That is, a memory segment has *spatial bounds*.

Every memory segment is associated with a scope[PREVIEW]. This ensures that access operations on a memory segment cannot occur when the region of memory which backs the memory segment is no longer available (e.g., after the scope associated with the accessed memory segment is no longer alive[PREVIEW]). That is, a memory segment has *temporal bounds*.

Finally, access operations on a memory segment can be subject to additional thread-confinement checks. Heap segments can be accessed from any thread. Conversely, native segments can only be accessed compatibly with the confinement characteristics of the arena used to obtain them.

Accessing memory segments

A memory segment can be read or written using various access operations provided in this class (e.g. `get(ValueLayout.OfInt, long)`). Each access operation takes a value layout[PREVIEW], which specifies the size and shape of the value, and an offset, expressed in bytes. 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 access operation can be expressed as follows:

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

For more complex access operations (e.g. structured memory access), clients can obtain a var handle[PREVIEW] that accepts a segment and a `long` offset. More complex 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, ValueLayout.JAVA_I
```

Alternatively, complex 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();
int value = (int) intHandle.get(segment, 3L); // get int element at offset 3 * 4 = 12
```

Slicing memory segments

Memory segments support slicing. Slicing a memory segment returns a new memory segment that is backed by the same region of memory as the original. The address of the sliced segment is derived from the address of the original segment, by adding an offset (expressed in bytes). The size of the sliced segment is either derived implicitly (by subtracting the specified offset from the size of the original segment), or provided explicitly. In other words, a sliced segment has *stricter* spatial bounds than those of the original segment:

```
Arena arena = ...
MemorySegment segment = arena.allocate(100);
MemorySegment slice = segment.asSlice(50, 10);
slice.get(ValueLayout.JAVA_INT, 20); // Out of bounds!
arena.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. That is, the address of the `slice` is `segment.address() + 50`, and its size is 10. 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 is inherited by its slices; that is, when the scope associated with `segment` is no longer alive[PREVIEW], `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 accessible from multiple threads). The following code can be used to sum all int values in a memory segment in parallel:

```
try (Arena arena = Arena.ofShared()) {
    SequenceLayout SEQUENCE_LAYOUT = MemoryLayout.sequenceLayout(1024, ValueLayout.JAVA_INT);
    MemorySegment segment = arena.allocate(SEQUENCE_LAYOUT);
    int sum = segment.elements(ValueLayout.JAVA_INT).parallel()
                     .mapToInt(s -> s.get(ValueLayout.JAVA_INT, 0))
                     .sum();
}
```

Alignment

Access operations on a memory segment are constrained not only by the spatial and temporal bounds of the segment, but also by the *alignment constraint* of the value layout specified to the operation. An access operation can access only those offsets in the segment that denote addresses in physical memory which are *aligned* according to the layout. An address in physical memory is *aligned* according to a layout if the address is an integer multiple of the layout's alignment constraint. For example, the address 1000 is aligned according to an 8-byte alignment constraint (because 1000 is an integer multiple of 8), and to a 4-byte alignment constraint, and to a 2-byte alignment constraint; in contrast, the address 1004 is aligned according to a 4-byte alignment constraint, and to a 2-byte alignment constraint, but not to an 8-byte alignment constraint. Access operations are required to respect alignment because it can impact the performance of access operations, and can also determine which access operations are available at a given physical address. For instance, atomic access operations operations using `VarHandle` are only permitted at aligned addresses. In addition, alignment applies to an access operation whether the segment being accessed is a native segment or a heap segment.

If the segment being accessed is a native segment, then its address in physical memory can be combined with the offset to obtain the *target address* in physical memory. The pseudo-function below demonstrates this:

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

For example:

- A native segment with address 1000 can be accessed at offsets 0, 8, 16, 24, etc under an 8-byte alignment constraint, because the target addresses (1000, 1008, 1016, 1024) are 8-byte aligned. Access at offsets 1-7 or 9-15 or 17-23 is disallowed because the target addresses would not be 8-byte aligned.
- A native segment with address 1000 can be accessed at offsets 0, 4, 8, 12, etc under a 4-byte alignment constraint, because the target addresses (1000, 1004, 1008, 1012) are 4-byte aligned. Access at offsets 1-3 or 5-7 or 9-11 is disallowed because the target addresses would not be 4-byte aligned.
- A native segment with address 1000 can be accessed at offsets 0, 2, 4, 6, etc under a 2-byte alignment constraint, because the target addresses (1000, 1002, 1004, 1006) are 2-byte aligned. Access at offsets 1 or 3 or 5 is disallowed because the target addresses would not be 2-byte aligned.

- A native segment with address 1004 can be accessed at offsets 0, 4, 8, 12, etc under a 4-byte alignment constraint, and at offsets 0, 2, 4, 6, etc under a 2-byte alignment constraint. Under an 8-byte alignment constraint, it can be accessed at offsets 4, 12, 20, 28, etc.
- A native segment with address 1006 can be accessed at offsets 0, 2, 4, 6, etc under a 2-byte alignment constraint. Under a 4-byte alignment constraint, it can be accessed at offsets 2, 6, 10, 14, etc. Under an 8-byte alignment constraint, it can be accessed at offsets 2, 10, 18, 26, etc.
- A native segment with address 1007 can be accessed at offsets 0, 1, 2, 3, etc under a 1-byte alignment constraint. Under a 2-byte alignment constraint, it can be accessed at offsets 1, 3, 5, 7, etc. Under a 4-byte alignment constraint, it can be accessed at offsets 1, 5, 9, 13, etc. Under an 8-byte alignment constraint, it can be accessed at offsets 1, 9, 17, 25, etc.

The alignment constraint used to access a segment is typically dictated by the shape of the data structure stored in the segment. For example, if the programmer wishes to store a sequence of 8-byte values in a native segment, then the segment should be allocated by specifying a 8-byte alignment constraint, either via `Arena.allocate(long, long)`[PREVIEW] or `SegmentAllocator.allocate(MemoryLayout)`[PREVIEW]. These factories ensure that the off-heap region of memory backing the returned segment has a starting address that is 8-byte aligned. Subsequently, the programmer can access the segment at the offsets of interest -- 0, 8, 16, 24, etc -- in the knowledge that every such access is aligned.

If the segment being accessed is a heap segment, then determining whether access is aligned is more complex. The address of the segment in physical memory is not known, and is not even fixed (it may change when the segment is relocated during garbage collection). This means that the address cannot be combined with the specified offset to determine a target address in physical memory. Since the alignment constraint *always* refers to alignment of addresses in physical memory, it is not possible in principle to determine if any offset in a heap segment is aligned. For example, suppose the programmer chooses a 8-byte alignment constraint and tries to access offset 16 in a heap segment. If the heap segment's address 0 corresponds to physical address 1000, then the target address (1016) would be aligned, but if address 0 corresponds to physical address 1004, then the target address (1020) would not be aligned. It is undesirable to allow access to target addresses that are aligned according to the programmer's chosen alignment constraint, but might not be predictably aligned in physical memory (e.g. because of platform considerations and/or garbage collection behavior).

In practice, the Java runtime lays out arrays in memory so that each n-byte element occurs at an n-byte aligned physical address (except for `long[]` and `double[]`, where alignment is platform-dependent, as explained below). The runtime preserves this invariant even if the array is relocated during garbage collection. Access operations rely on this invariant to determine if the specified offset in a heap segment refers to an aligned address in physical memory. For example:

- The starting physical address of a `short[]` array will be 2-byte aligned (e.g. 1006) so that successive short elements occur at 2-byte aligned addresses (e.g. 1006, 1008, 1010, 1012, etc). A heap segment backed by a `short[]` array can be accessed at offsets 0, 2, 4, 6, etc under a 2-byte alignment constraint. The segment cannot be accessed at *any* offset under a 4-byte alignment constraint, because there is no guarantee that the target address would be 4-byte aligned, e.g., offset 0 would correspond to physical address 1006 while offset 1 would correspond to physical address 1007. Similarly, the segment cannot be accessed at any offset under an 8-byte alignment constraint, because because there is no guarantee that the target address would be 8-byte aligned, e.g., offset 2 would correspond to physical address 1008 but offset 4 would correspond to physical address 1010.
- The starting physical address of a `long[]` array will be 8-byte aligned (e.g. 1000) on 64-bit platforms, so that successive long elements occur at 8-byte aligned addresses (e.g., 1000, 1008, 1016, 1024, etc.) On 64-bit platforms, a heap segment backed by a `long[]` array can be accessed at offsets 0, 8, 16, 24, etc under an 8-byte alignment constraint. In addition, the segment can be accessed at offsets 0, 4, 8, 12, etc under a 4-byte alignment constraint, because the target addresses (1000, 1004, 1008, 1012) are 4-byte aligned. And, the segment can be accessed at offsets 0, 2, 4, 6, etc under a 2-byte alignment constraint, because the target addresses (e.g. 1000, 1002, 1004, 1006) are 2-byte aligned.
- The starting physical address of a `long[]` array will be 4-byte aligned (e.g. 1004) on 32-bit platforms, so that successive long elements occur at 4-byte aligned addresses (e.g., 1004, 1008, 1012, 1016, etc.) On 32-bit platforms, a heap segment backed by a `long[]` array can be accessed at offsets 0, 4, 8, 12, etc under a 4-byte alignment constraint, because the target addresses (1004, 1008, 1012, 1016) are 4-byte aligned. And, the segment can be accessed at offsets 0, 2, 4, 6, etc under a 2-byte alignment constraint, because the target addresses (e.g. 1000, 1002, 1004, 1006) are 2-byte aligned.

In other words, heap segments feature a (platform-dependent) *maximum* alignment which is derived from the size of the elements of the Java array backing the segment, as shown in the following table:

Array type (of backing region)	Maximum supported alignment (in bytes)
`boolean[]`	`ValueLayout.JAVA_BOOLEAN.byteAlignment()`
`byte[]`	`ValueLayout.JAVA_BYTE.byteAlignment()`
`char[]`	`ValueLayout.JAVA_CHAR.byteAlignment()`
`short[]`	`ValueLayout.JAVA_SHORT.byteAlignment()`
`int[]`	`ValueLayout.JAVA_INT.byteAlignment()`
`float[]`	`ValueLayout.JAVA_FLOAT.byteAlignment()`
`long[]`	`ValueLayout.JAVA_LONG.byteAlignment()`
`double[]`	`ValueLayout.JAVA_DOUBLE.byteAlignment()`

Heap segments can only be accessed using a layout whose alignment is smaller or equal to the maximum alignment associated with the heap segment. Attempting to access a heap segment using a layout whose alignment is greater than the maximum alignment associated with the heap segment will fail, as demonstrated in the following example:

```
MemorySegment byteSegment = MemorySegment.ofArray(new byte[10]);
byteSegment.get(ValueLayout.JAVA_INT, 0); // fails: ValueLayout.JAVA_INT.byteAlignment() > ValueLayout.JAVA_BY
```

In such circumstances, clients have two options. They can use a heap segment backed by a different array type (e.g. `long[]`), capable of supporting greater maximum alignment. More specifically, the maximum alignment associated with `long[]` is set to `ValueLayout.JAVA_LONG.byteAlignment()` which is a platform-dependent value (set to `ValueLayout.ADDRESS.byteSize()`). That is, `long[]`) is guaranteed to provide at least 8-byte alignment in 64-bit platforms, but only 4-byte alignment in 32-bit platforms:

```
MemorySegment longSegment = MemorySegment.ofArray(new long[10]);
longSegment.get(ValueLayout.JAVA_INT, 0); // ok: ValueLayout.JAVA_INT.byteAlignment() <= ValueLayout.JAVA_LONG
```

Alternatively, they can invoke the access operation with an *unaligned layout*. All unaligned layout constants (e.g. `ValueLayout.JAVA_INT_UNALIGNED`[PREVIEW]) have their alignment constraint set to 1:

```
MemorySegment byteSegment = MemorySegment.ofArray(new byte[10]);
byteSegment.get(ValueLayout.JAVA_INT_UNALIGNED, 0); // ok: ValueLayout.JAVA_INT_UNALIGNED.byteAlignment() == 1
```

Zero-length memory segments

When interacting with foreign functions, it is common for those functions to allocate a region of memory and return a pointer to that region. Modeling the region of memory with a memory segment is challenging because the Java runtime has no insight into the size of the region. Only the address of the start of the region, stored in the pointer, is available. For example, a C function with return type `char*` might return a pointer to a region containing a single `char` value, or to a region containing an array of `char` values, where the size of the array might be provided in a separate parameter. The size of the array is not readily apparent to the code calling the foreign function and hoping to use its result. In addition to having no insight into the size of the region of memory backing a pointer returned from a foreign function, it also has no insight into the lifetime intended for said region of memory by the foreign function that allocated it.

The `MemorySegment` API uses *zero-length memory segments* to represent:

- pointers returned from a foreign function;
- pointers passed by a foreign function to an upcall stub; and
- pointers read from a memory segment (more on that below).

The address of the zero-length segment is the address stored in the pointer. The spatial and temporal bounds of the zero-length segment are as follows:

- The size of the segment is zero. any attempt to access these segments will fail with `IndexOutOfBoundsException`. This is a crucial safety feature: as these segments are associated with a region of memory whose size is not known, any access operations involving these segments cannot be validated. In effect, a zero-length memory segment *wraps* an address, and it cannot be used without explicit intent (see below);
- The segment is associated with a fresh scope that is always alive. Thus, while zero-length memory segments cannot be accessed directly, they can be passed, opaquely, to other pointer-accepting foreign functions.

To demonstrate how clients can work with zero-length memory segments, consider the case of a client that wants to read a pointer from some memory segment. This can be done via the get(AddressLayout, long) access method. This method accepts an address layout[PREVIEW] (e.g. `ValueLayout.ADDRESS`[PREVIEW]), the layout of the pointer to be read. For instance on a 64-bit platform, the size of an address layout is 8 bytes. The access operation also accepts an offset, expressed in bytes, which indicates the position (relative to the start of the memory segment) at which the pointer is stored. The access operation returns a zero-length native memory segment, backed by a region of memory whose starting address is the 64-bit value read at the specified offset.

The returned zero-length memory segment cannot be accessed directly by the client: since the size of the segment is zero, any access operation would result in out-of-bounds access. Instead, the client must, *unsafely*, assign new spatial bounds to the zero-length memory segment. This can be done via the `reinterpret(long)` method, as follows:

```
MemorySegment z = segment.get(ValueLayout.ADDRESS, ...);   // size = 0
MemorySegment ptr = z.reinterpret(16);                     // size = 16
int x = ptr.getAtIndex(ValueLayout.JAVA_INT, 3);           // ok
```

In some cases, the client might additionally want to assign new temporal bounds to a zero-length memory segment. This can be done via the `reinterpret(long, Arena, Consumer)` method, which returns a new native segment with the desired size and the same temporal bounds as those of the provided arena:

```
MemorySegment ptr = null;
try (Arena arena = Arena.ofConfined()) {
      MemorySegment z = segment.get(ValueLayout.ADDRESS, ...);    // size = 0, scope = always alive
      ptr = z.reinterpret(16, arena, null);                       // size = 4, scope = arena.scope()
      int x = ptr.getAtIndex(ValueLayout.JAVA_INT, 3);            // ok
}
int x = ptr.getAtIndex(ValueLayout.JAVA_INT, 3);                  // throws IllegalStateException
```

Alternatively, if the size of the region of memory backing the zero-length memory segment is known statically, the client can overlay a target layout[PREVIEW] on the address layout used when reading a pointer. The target layout is then used to dynamically *expand* the size of the native memory segment returned by the access operation, so that the size of the segment is the same as the size of the target layout. In other words, the returned segment is no longer a zero-length memory segment, and the pointer it represents can be dereferenced directly:

```
AddressLayout intArrPtrLayout = ValueLayout.ADDRESS.withTargetLayout(
        MemoryLayout.sequenceLayout(4, ValueLayout.JAVA_INT)); // layout for int (*ptr)[4]
MemorySegment ptr = segment.get(intArrPtrLayout, ...);         // size = 16
int x = ptr.getAtIndex(ValueLayout.JAVA_INT, 3);               // ok
```

All the methods which can be used to manipulate zero-length memory segments (`reinterpret(long)`, `reinterpret(Arena, Consumer)`, `reinterpret(long, Arena, Consumer)` and `AddressLayout.withTargetLayout(MemoryLayout)`[PREVIEW]) are *restricted* methods, and should be used with caution: assigning a segment incorrect spatial and/or temporal bounds could result in a VM crash when attempting to access the memory segment.

Implementation Requirements:

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

Since:

19

Nested Class Summary

Nested Classes

Modifier and Type	Interface	Description
static interface	`MemorySegment.Scope`[PREVIEW]	**Preview.** A scope models the *lifetime* of all the memory segments associated with it.

Field Summary

Fields

Modifier and Type	Field	Description
static final `MemorySegment`[PREVIEW]	`NULL`	A zero-length native segment modelling the NULL address.

Method Summary

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

Modifier and Type	Method	Description
long	`address()`	Returns the address of this memory segment.
`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.
`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.
`MemorySegment`[PREVIEW]	`asSlice(long offset, long newSize, long byteAlignment)`	Returns a slice of this memory segment, at the given offset, with the provided alignment constraint.
default `MemorySegment`[PREVIEW]	`asSlice(long offset, MemoryLayout`[PREVIEW] layout)	Returns a slice of this memory segment with the given layout, 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 the contents of this memory segment with the given value.
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 **MemorySegment**[PREVIEW]	**get**(**AddressLayout**[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 **MemorySegment**[PREVIEW]	**getAtIndex**(**AddressLayout**[PREVIEW] layout, long index)	Reads an address from this segment at the given at the given index, scaled by the given layout size.
default boolean	**getAtIndex**(**ValueLayout.OfBoolean**[PREVIEW] layout, long index)	Reads a boolean from this segment at the given index, scaled by the given layout size.
default byte	**getAtIndex**(**ValueLayout.OfByte**[PREVIEW] layout, long index)	Reads a byte 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.
Optional<**Object**>	**heapBase**()	Returns the Java object stored in the on-heap region of memory backing this memory segment, if any.
boolean	**isAccessibleBy**(**Thread** thread)	Returns `true` if this segment can be accessed from the provided thread.
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.
default 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 long	**mismatch**(**MemorySegment**[PREVIEW] srcSegment, long srcFromOffset, long srcToOffset, **MemorySegment**[PREVIEW] dstSegment, long dstFromOffset, long dstToOffset)	Finds and returns the relative offset, in bytes, of the first mismatch between the source and the destination segments.
static **MemorySegment**[PREVIEW]	**ofAddress**(long address)	Creates a zero-length native segment from the given address value.
static **MemorySegment**[PREVIEW]	**ofArray**(byte[] byteArray)	Creates a heap segment backed by the on-heap region of memory that holds the given byte array.
static **MemorySegment**[PREVIEW]	**ofArray**(char[] charArray)	Creates a heap segment backed by the on-heap region of memory that holds the given char array.
static **MemorySegment**[PREVIEW]	**ofArray**(double[] doubleArray)	Creates a heap segment backed by the on-heap region of memory that holds the given double array.
static **MemorySegment**[PREVIEW]	**ofArray**(float[] floatArray)	Creates a heap segment backed by the on-heap region of memory that holds the given float array.
static **MemorySegment**[PREVIEW]	**ofArray**(int[] intArray)	Creates a heap segment backed by the on-heap region of memory that holds the given int array.
static **MemorySegment**[PREVIEW]	**ofArray**(long[] longArray)	Creates a heap segment backed by the on-heap region of memory that holds the given long array.
static **MemorySegment**[PREVIEW]	**ofArray**(short[] shortArray)	Creates a heap segment backed by the on-heap region of memory that holds the given short array.
static **MemorySegment**[PREVIEW]	**ofBuffer**(**Buffer** buffer)	Creates a memory segment that is backed by the same region of memory that backs the given `Buffer` instance.
MemorySegment[PREVIEW]	**reinterpret**(long newSize)	Returns a new memory segment that has the same address and scope as this segment, but with the provided size.

MemorySegment[PREVIEW]	reinterpret(long newSize, Arena[PREVIEW] arena, Consumer<MemorySegment[PREVIEW]> cleanup)	Returns a new segment with the same address as this segment, but with the provided size and scope.
MemorySegment[PREVIEW]	reinterpret(Arena[PREVIEW] arena, Consumer<MemorySegment[PREVIEW]> cleanup)	Returns a new memory segment with the same address and size as this segment, but with the provided scope.
MemorySegment.Scope[PREVIEW]	scope()	Returns the scope associated with this memory segment.
long	segmentOffset (MemorySegment[PREVIEW] other)	Returns the offset, in bytes, of the provided segment, relative to this segment.
default void	set(AddressLayout[PREVIEW] layout, long offset, MemorySegment[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(AddressLayout[PREVIEW] layout, long index, MemorySegment[PREVIEW] value)	Writes an address into this segment at the given index, scaled by the given layout size.
default void	setAtIndex (ValueLayout.OfBoolean[PREVIEW] layout, long index, boolean value)	Writes a boolean into this segment at the given index, scaled by the given layout size.
default void	setAtIndex (ValueLayout.OfByte[PREVIEW] layout, long index, byte value)	Writes a byte 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.

Field Details

NULL

```
static final MemorySegment[PREVIEW] NULL
```

A zero-length native segment modelling the `NULL` address.

Method Details

address

```
long address()
```

Returns the address of this memory segment.

API Note:

When using this method to pass a segment address to some external operation (e.g. a JNI function), clients must ensure that the segment is kept reachable for the entire duration of the operation. A failure to do so might result in the premature deallocation of the region of memory backing the memory segment, in case the segment has been allocated with an automatic arena[PREVIEW].

Returns:

the address of this memory segment

heapBase

```
Optional<Object> heapBase()
```

Returns the Java object stored in the on-heap region of memory backing this memory segment, if any. For instance, if this memory segment is a heap segment created with the `ofArray(byte[])` factory method, this method will return the `byte[]` object which was used to obtain the segment. This method returns an empty `Optional` value if either this segment is a native segment, or if this segment is read-only.

Returns:

the Java object associated with this memory segment, if any.

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 have the same lifetime as that of 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 `elementLayout.byteSize() == 0`.

`IllegalArgumentException` - if `byteSize() % elementLayout.byteSize() != 0`.

`IllegalArgumentException` - if `elementLayout.byteSize() % elementLayout.byteAlignment() != 0`.

`IllegalArgumentException` - if this segment is incompatible with the alignment constraint in the provided layout.

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 `elementLayout.byteSize() == 0`.

`IllegalArgumentException` - if `byteSize() % elementLayout.byteSize() != 0`.

`IllegalArgumentException` - if `elementLayout.byteSize() % elementLayout.byteAlignment() != 0`.

`IllegalArgumentException` - if this segment is incompatible with the alignment constraint in the provided layout.

scope

`MemorySegment.Scope`[PREVIEW]` scope()`

Returns the scope associated with this memory segment.

Returns:

the scope associated with this memory segment

isAccessibleBy

`boolean isAccessibleBy(Thread thread)`

Returns `true` if this segment can be accessed from the provided thread.

Parameters:

`thread` - the thread to be tested.

Returns:

`true` if this segment can be accessed from the provided thread

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 address is the address of this segment plus the given offset; its size is specified by the given argument.

Equivalent to the following code:

```
asSlice(offset, newSize, 1);
```

Parameters:

`offset` - The new segment base offset (relative to the address of this segment), 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, long, long)`

asSlice

```
MemorySegment[PREVIEW] asSlice(long offset,
                               long newSize,
                               long byteAlignment)
```

Returns a slice of this memory segment, at the given offset, with the provided alignment constraint. The returned segment's address is the 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 address of this segment), specified in bytes.

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

`byteAlignment` - The alignment constraint (in bytes) of the returned slice.

Returns:

a slice of this memory segment.

Throws:

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

`IllegalArgumentException` - if this segment cannot be accessed at `offset` under the provided alignment constraint.

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

asSlice

```
default MemorySegment[PREVIEW] asSlice(long offset,
                                       MemoryLayout[PREVIEW] layout)
```

Returns a slice of this memory segment with the given layout, at the given offset. The returned segment's address is the address of this segment plus the given offset; its size is the same as the size of the provided layout.

Equivalent to the following code:

```
asSlice(offset, layout.byteSize(), layout.byteAlignment());
```

Parameters:

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

`layout` - The layout of the segment slice.

Returns:

a slice of this memory segment.

Throws:

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

`IllegalArgumentException` - if this segment cannot be accessed at `offset` under the alignment constraint specified by `layout`.

See Also:

`asSlice(long, long, long)`

asSlice

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

Returns a slice of this memory segment, at the given offset. The returned segment's address is the 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 address of this segment), specified in bytes.

Returns:

a slice of this memory segment.

Throws:

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

See Also:

`asSlice(long, long)`

reinterpret

`MemorySegment`[PREVIEW] `reinterpret(long newSize)`

Returns a new memory segment that has the same address and scope as this segment, but with the provided size.

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:

`newSize` - the size of the returned segment.

Returns:

a new memory segment that has the same address and scope as this segment, but the new provided size.

Throws:

`IllegalArgumentException` - if `newSize < 0`.

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

`IllegalCallerException` - If the caller is in a module that does not have native access enabled.

reinterpret

```
MemorySegment reinterpret(Arena arena,
                          Consumer<MemorySegment> cleanup)
```

Returns a new memory segment with the same address and size as this segment, but with the provided scope. As such, the returned segment cannot be accessed after the provided arena has been closed. Moreover, the returned segment can be accessed compatibly with the confinement restrictions associated with the provided arena: that is, if the provided arena is a confined arena[PREVIEW], the returned segment can only be accessed by the arena's owner thread, regardless of the confinement restrictions associated with this segment. In other words, this method returns a segment that behaves as if it had been allocated using the provided arena.

Clients can specify an optional cleanup action that should be executed when the provided scope becomes invalid. This cleanup action receives a fresh memory segment that is obtained from this segment as follows:

```
MemorySegment cleanupSegment = MemorySegment.ofAddress(this.address())
                                            .reinterpret(byteSize());
```

That is, the cleanup action receives a segment that is associated with a fresh scope that is always alive, and is accessible from any thread. The size of the segment accepted by the cleanup action is `byteSize()`.

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.

API Note:

The cleanup action (if present) should take care not to leak the received segment to external clients which might access the segment after its backing region of memory is no longer available. Furthermore, if the provided scope is the scope of an automatic arena[PREVIEW], the cleanup action must not prevent the scope from becoming unreachable. A failure to do so will permanently prevent the regions of memory allocated by the automatic arena from being deallocated.

Parameters:

`arena` - the arena to be associated with the returned segment.

`cleanup` - the cleanup action that should be executed when the provided arena is closed (can be `null`).

Returns:

a new memory segment with unbounded size.

Throws:

`IllegalStateException` - if `arena.scope().isAlive() == false`.

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

`IllegalCallerException` - If the caller is in a module that does not have native access enabled.

reinterpret

```
MemorySegment[PREVIEW] reinterpret(long newSize,
                                   Arena[PREVIEW] arena,
                                   Consumer<MemorySegment[PREVIEW]> cleanup)
```

Returns a new segment with the same address as this segment, but with the provided size and scope. As such, the returned segment cannot be accessed after the provided arena has been closed. Moreover, if the returned segment can be accessed compatibly with the confinement restrictions associated with the provided arena: that is, if the provided arena is a confined arena[PREVIEW], the returned segment can only be accessed by the arena's owner thread, regardless of the confinement restrictions associated with this segment. In other words, this method returns a segment that behaves as if it had been allocated using the provided arena.

Clients can specify an optional cleanup action that should be executed when the provided scope becomes invalid. This cleanup action receives a fresh memory segment that is obtained from this segment as follows:

```
MemorySegment cleanupSegment = MemorySegment.ofAddress(this.address())
                                            .reinterpret(newSize);
```

That is, the cleanup action receives a segment that is associated with a fresh scope that is always alive, and is accessible from any thread. The size of the segment accepted by the cleanup action is `newSize`.

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.

API Note:

The cleanup action (if present) should take care not to leak the received segment to external clients which might access the segment after its backing region of memory is no longer available. Furthermore, if the provided scope is the scope of an automatic arena[PREVIEW], the cleanup action must not prevent the scope from becoming unreachable. A failure to do so will permanently prevent the regions of memory allocated by the automatic arena from being deallocated.

Parameters:

`newSize` - the size of the returned segment.

`arena` - the arena to be associated with the returned segment.

`cleanup` - the cleanup action that should be executed when the provided arena is closed (can be `null`).

Returns:

a new segment that has the same address as this segment, but with new size and its scope set to that of the provided arena.

Throws:

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

`IllegalArgumentException` - if `newSize < 0`.

`IllegalStateException` - if `arena.scope().isAlive() == false`.

`IllegalCallerException` - If the caller is in a module that does not have native access enabled.

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 segment is created e.g. using the `Arena.allocate(long, long)`[PREVIEW] (and related) factory, or by wrapping a direct buffer.

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 e.g. using the `FileChannel.map(FileChannel.MapMode, long, long, Arena)`[PREVIEW] factory, or by wrapping a mapped byte buffer.

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 region of memory. As such, it is not possible for a native segment to overlap with a heap segment; in this case, or when no overlap occurs, an empty `Optional` 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 address of this segment and can be a negative or positive value. For instance, if both segments are native segments, or heap segments backed by the same array, the resulting offset can be computed as follows:

```
other.address() - address()
```

If the segments share the same 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.

Throws:

`UnsupportedOperationException` - if the two segments cannot be compared, e.g. because they are of different kinds, or because they are backed by different Java arrays.

fill

```
MemorySegment[PREVIEW] fill(byte value)
```

Fills the contents of this memory segment with the given value.

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

```
for (long offset = 0; offset < segment.byteSize(); offset++) {
    byteHandle.set(ValueLayout.JAVA_BYTE, offset, value);
}
```

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

This method can be useful to initialize or reset the contents of a memory segment.

Parameters:

`value` - the value to write into this segment.

Returns:

this memory segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `src.isAccessibleBy(T) == false`.

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

mismatch

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `other.isAccessibleBy(T) == false`.

isLoaded

```
boolean isLoaded()
```

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

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

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

Returns:

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

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

load

```
void load()
```

Loads the contents of this mapped segment into physical memory.

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

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

unload

```
void unload()
```

Unloads the contents of this mapped segment from physical memory.

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

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

force

```
void force()
```

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

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

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

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

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`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. More specifically, the resulting buffer has the following characteristics:

- It is read-only, if this segment is a read-only segment;
- Its position is set to zero;
- Its capacity and limit are both set to this segment' size. For this reason, a byte buffer cannot be returned if this segment's size is greater than `Integer.MAX_VALUE`;
- It is a direct buffer, if this is a native segment.

The life-cycle of the returned buffer is tied to that of this segment. That is, accessing the returned buffer after the scope associated with this segment is no longer alive[PREVIEW], will throw an `IllegalStateException`. Similarly, accessing the returned buffer from a thread `T` such that `isAccessible(T) == false` will throw a `WrongThreadException`.

If this segment is accessible from a single thread, calling read/write I/O operations on the resulting buffer might result in unspecified exceptions 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. if it is a heap segment backed by an array other than `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 scope associated with this segment is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`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 address) at which this access operation will occur.

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 `offset < 0` or `offset > byteSize() - S`, where `S` is the size of the UTF-8 string (including the terminator character).

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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 address) at which this access operation will occur. the final address of this write operation can be expressed as `address() + offset`.

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

Throws:

`IndexOutOfBoundsException` - if `offset < 0` or `offset > byteSize() - str.getBytes().length() + 1`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

ofBuffer

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

Creates a memory segment that is backed by the same region of memory that backs 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 is also read-only. Moreover, if the buffer is a direct buffer, the returned segment is a native segment; otherwise the returned memory segment is a heap segment.

If the provided buffer has been obtained by calling `asByteBuffer()` on a memory segment whose scope[PREVIEW] is `S`, the returned segment will be associated with the same scope `S`. Otherwise, the scope of the returned segment is a fresh scope that is always alive.

The scope associated with the returned segment keeps the provided buffer reachable. As such, if the provided buffer is a direct buffer, its backing memory region will not be deallocated as long as the returned segment (or any of its slices) are

kept reachable.

Parameters:

`buffer` - the buffer instance to be turned into a new memory segment.

Returns:

a memory segment, derived from the given buffer instance.

Throws:

`IllegalArgumentException` - if the provided `buffer` is a heap buffer but is not backed by an array. For example, buffers directly or indirectly obtained via (`CharBuffer.wrap(CharSequence)` or `CharBuffer.wrap(char[], int, int)` are not backed by an array.

ofArray

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

Creates a heap segment backed by the on-heap region of memory that holds the given byte array. The scope of the returned segment is a fresh scope that is always alive, and keeps the given array reachable. The returned segment is always accessible, from any thread. Its `address()` is set to zero.

Parameters:

`byteArray` - the primitive array backing the heap memory segment.

Returns:

a heap memory segment backed by a byte array.

ofArray

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

Creates a heap segment backed by the on-heap region of memory that holds the given char array. The scope of the returned segment is a fresh scope that is always alive, and keeps the given array reachable. The returned segment is always accessible, from any thread. Its `address()` is set to zero.

Parameters:

`charArray` - the primitive array backing the heap segment.

Returns:

a heap memory segment backed by a char array.

ofArray

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

Creates a heap segment backed by the on-heap region of memory that holds the given short array. The scope of the returned segment is a fresh scope that is always alive, and keeps the given array reachable. The returned segment is always accessible, from any thread. Its `address()` is set to zero.

Parameters:

`shortArray` - the primitive array backing the heap segment.

Returns:

a heap memory segment backed by a short array.

ofArray

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

Creates a heap segment backed by the on-heap region of memory that holds the given int array. The scope of the returned segment is a fresh scope that is always alive, and keeps the given array reachable. The returned segment is always accessible, from any thread. Its `address()` is set to zero.

Parameters:

`intArray` - the primitive array backing the heap segment.

Returns:

a heap memory segment backed by an int array.

ofArray

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

Creates a heap segment backed by the on-heap region of memory that holds the given float array. The scope of the returned segment is a fresh scope that is always alive, and keeps the given array reachable. The returned segment is always accessible, from any thread. Its `address()` is set to zero.

Parameters:

`floatArray` - the primitive array backing the heap segment.

Returns:

a heap memory segment backed by a float array.

ofArray

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

Creates a heap segment backed by the on-heap region of memory that holds the given long array. The scope of the returned segment is a fresh scope that is always alive, and keeps the given array reachable. The returned segment is always accessible, from any thread. Its `address()` is set to zero.

Parameters:

`longArray` - the primitive array backing the heap segment.

Returns:

a heap memory segment backed by a long array.

ofArray

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

Creates a heap segment backed by the on-heap region of memory that holds the given double array. The scope of the returned segment is a fresh scope that is always alive, and keeps the given array reachable. The returned segment is always accessible, from any thread. Its `address()` is set to zero.

Parameters:

`doubleArray` - the primitive array backing the heap segment.

Returns:

a heap memory segment backed by a double array.

ofAddress

`static MemorySegment[PREVIEW] ofAddress(long address)`

Creates a zero-length native segment from the given address value. The returned segment is associated with a scope that is always alive, and is accessible from any thread.

On 32-bit platforms, the given address value will be normalized such that the highest-order ("leftmost") 32 bits of the `address` of the returned memory segment are set to zero.

Parameters:

`address` - the address of the returned native segment.

Returns:

a zero-length native segment with the given address.

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 the destination 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 scope associated with `srcSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread `T`, such that `srcSegment.isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `dstSegment.isAccessibleBy(T) == false`.

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

`UnsupportedOperationException` - if `dstSegment` is read-only.

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 the destination 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 constraint in the source (resp. destination) element layout, or if the source (resp. destination) element layout alignment is greater than its size.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `srcSegment().isAccessibleBy(T) == false`.

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

`WrongThreadException` - if this method is called from a thread `T`, such that `dstSegment().isAccessibleBy(T) == false`.

`UnsupportedOperationException` - if `dstSegment` is read-only.

`IndexOutOfBoundsException` - if `elementCount * srcLayout.byteSize()` or `elementCount * dtsLayout.byteSize()` overflows.

`IndexOutOfBoundsException` - if `dstOffset > dstSegment.byteSize() - (elementCount * dstLayout.byteSize())`.

`IndexOutOfBoundsException` - if either `srcOffset`, `dstOffset` or `elementCount` are `< 0`.

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 region of memory to be read.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a byte value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

`value` - the byte value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a boolean value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

`value` - the boolean value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a char value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

`value` - the char value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a short value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

IndexOutOfBoundsException - if offset > byteSize() - layout.byteSize().

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 region of memory to be written.

offset - offset in bytes (relative to this segment address) at which this access operation will occur.

value - the short value to be written.

Throws:

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

WrongThreadException - if this method is called from a thread T, such that isAccessibleBy(T) == false.

IllegalArgumentException - if the access operation is incompatible with the alignment constraint in the provided layout.

IndexOutOfBoundsException - if offset > byteSize() - layout.byteSize().

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 region of memory to be read.

offset - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

an int value read from this segment.

Throws:

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

WrongThreadException - if this method is called from a thread T, such that isAccessibleBy(T) == false.

IllegalArgumentException - if the access operation is incompatible with the alignment constraint in the provided layout.

IndexOutOfBoundsException - if offset > byteSize() - layout.byteSize().

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 region of memory to be written.

offset - offset in bytes (relative to this segment address) at which this access operation will occur.

value - the int value to be written.

Throws:

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

WrongThreadException - if this method is called from a thread T, such that isAccessibleBy(T) == false.

IllegalArgumentException - if the access operation is incompatible with the alignment constraint in the provided layout.

IndexOutOfBoundsException - if offset > byteSize() - layout.byteSize().

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 region of memory to be read.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a float value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

`value` - the float value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a long value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

`value` - the long value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a double value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

`value` - the double value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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

get

```
default MemorySegment[PREVIEW] get(AddressLayout[PREVIEW] layout,
                                   long offset)
```

Reads an address from this segment at the given offset, with the given layout. The read address is wrapped in a native segment, associated with a fresh scope that is always alive. Under normal conditions, the size of the returned segment is `0`. However, if the provided address layout has a target layout[PREVIEW] `T`, then the size of the returned segment is set to `T.byteSize()`.

Parameters:

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

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

Returns:

a native segment wrapping an address read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IllegalArgumentException` - if provided address layout has a target layout[PREVIEW] `T`, and the address of the returned segment incompatible with the alignment constraint in `T`.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

set

```
default void set(AddressLayout[PREVIEW] layout,
                 long offset,
                 MemorySegment[PREVIEW] value)
```

Writes an address into this segment at the given offset, with the given layout.

Parameters:

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

`offset` - offset in bytes (relative to this segment address) at which this access operation will occur.

`value` - the address value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

`IllegalArgumentException` - if the access operation is incompatible with the alignment constraint in the provided layout.

`IndexOutOfBoundsException` - if `offset > byteSize() - layout.byteSize()`.

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

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

getAtIndex

```
default byte getAtIndex(ValueLayout.OfByte[PREVIEW] layout,
                        long index)
```

Reads a byte from this segment at the given index, scaled by the given layout size.

Parameters:

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

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a byte value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

getAtIndex

```
default boolean getAtIndex(ValueLayout.OfBoolean[PREVIEW] layout,
                           long index)
```

Reads a boolean from this segment at the given index, scaled by the given layout size.

Parameters:

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

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a boolean value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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 region of memory to be read.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a char value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the char value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a short value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

setAtIndex

```
default void setAtIndex(ValueLayout.OfByte[PREVIEW] layout,
                        long index,
                        byte value)
```

Writes a byte into this segment at the given index, scaled by the given layout size.

Parameters:

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

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the short value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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

setAtIndex

```
default void setAtIndex(ValueLayout.OfBoolean[PREVIEW] layout,
                        long index,
                        boolean value)
```

Writes a boolean into this segment at the given index, scaled by the given layout size.

Parameters:

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

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the short value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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

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 region of memory to be written.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the short value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

an int value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the int value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a float value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the float value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a long value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the long value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

`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 region of memory to be read.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a double value read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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 region of memory to be written.

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

`value` - the double value to be written.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IndexOutOfBoundsException` - if `index * byteSize()` overflows.

`IndexOutOfBoundsException` - if `index * byteSize() > byteSize() - layout.byteSize()`.

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

getAtIndex

```
default MemorySegment[PREVIEW] getAtIndex(AddressLayout[PREVIEW] layout,
                                          long index)
```

Reads an address from this segment at the given at the given index, scaled by the given layout size. The read address is wrapped in a native segment, associated with a fresh scope that is always alive. Under normal conditions, the size of the returned segment is `0`. However, if the provided address layout has a target layout[PREVIEW] `T`, then the size of the returned segment is set to `T.byteSize()`.

Parameters:

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

`index` - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as `(index * layout.byteSize())`.

Returns:

a native segment wrapping an address read from this segment.

Throws:

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

`WrongThreadException` - if this method is called from a thread `T`, such that `isAccessibleBy(T) == false`.

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

`IllegalArgumentException` - if provided address layout has a target layout[PREVIEW] `T`, and the address of the returned segment incompatible with the alignment constraint in `T`.

IndexOutOfBoundsException - if index * byteSize() overflows.

IndexOutOfBoundsException - if index * byteSize() > byteSize() - layout.byteSize().

setAtIndex

```
default void setAtIndex(AddressLayout PREVIEW layout,
                        long index,
                        MemorySegment PREVIEW value)
```

Writes an address into this segment at the given index, scaled by the given layout size.

Parameters:

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

index - a logical index. The offset in bytes (relative to this segment address) at which the access operation will occur can be expressed as (index * layout.byteSize()).

value - the address value to be written.

Throws:

IllegalStateException - if the scope associated with this segment is not alive PREVIEW.

WrongThreadException - if this method is called from a thread T, such that isAccessibleBy(T) == false.

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

IndexOutOfBoundsException - if index * byteSize() overflows.

IndexOutOfBoundsException - if index * byteSize() > byteSize() - layout.byteSize().

UnsupportedOperationException - if this segment is read-only.

UnsupportedOperationException - if value is not a native segment.

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 the two segments refer to the same location, in some region of memory. More specifically, for two segments s1 and s2 to be considered equals, all the following must be true:

- s1.heapBase().equals(s2.heapBase()), that is, the two segments must be of the same kind; either both are native segments, backed by off-heap memory, or both are backed by the same on-heap Java object;
- s1.address() == s2.address(), that is, the address of the two segments should be the same. This means that the two segments either refer to the same location in some off-heap region, or they refer to the same offset inside their associated Java object.

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. Note that this method does *not* compare the temporal and spatial bounds of two segments. As such it is suitable to check whether two segments have the same address.

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)

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 scope associated with `srcSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread `T`, such that `srcSegment().isAccessibleBy(T) == false`.

`IllegalArgumentException` - if `dstArray` is not an array, or if it is an array but whose type is not supported.

`IllegalArgumentException` - if the destination array component type does not match `srcLayout.carrier()`.

`IllegalArgumentException` - if `offset` is incompatible with the alignment constraint in the source element layout.

`IllegalArgumentException` - if `srcLayout.byteAlignment() > srcLayout.byteSize()`.

`IndexOutOfBoundsException` - if `elementCount * srcLayout.byteSize()` overflows.

`IndexOutOfBoundsException` - if `srcOffset > srcSegment.byteSize() - (elementCount * srcLayout.byteSize())`.

`IndexOutOfBoundsException` - if `dstIndex > dstArray.length - elementCount`.

`IndexOutOfBoundsException` - if either `srcOffset`, `dstIndex` or `elementCount` are `< 0`.

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 scope associated with `dstSegment` is not alive[PREVIEW].

`WrongThreadException` - if this method is called from a thread `T`, such that `dstSegment().isAccessibleBy(T) == false`.

`IllegalArgumentException` - if `srcArray` is not an array, or if it is an array but whose type is not supported.

`IllegalArgumentException` - if the source array component type does not match `srcLayout.carrier()`.

`IllegalArgumentException` - if `offset` is incompatible with the alignment constraint in the source element layout.

`IllegalArgumentException` - if `dstLayout.byteAlignment() > dstLayout.byteSize()`.

`UnsupportedOperationException` - if `dstSegment` is read-only.

IndexOutOfBoundsException - if `elementCount * dstLayout.byteSize()` overflows.

IndexOutOfBoundsException - if `dstOffset > dstSegment.byteSize() - (elementCount * dstLayout.byteSize())`.

IndexOutOfBoundsException - if `srcIndex > srcArray.length - elementCount`.

IndexOutOfBoundsException - if either `srcIndex`, `dstOffset` or `elementCount` are `< 0`.

mismatch

```
static long mismatch(MemorySegment[PREVIEW] srcSegment,
                     long srcFromOffset,
                     long srcToOffset,
                     MemorySegment[PREVIEW] dstSegment,
                     long dstFromOffset,
                     long dstToOffset)
```

Finds and returns the relative offset, in bytes, of the first mismatch between the source and the destination segments. More specifically, the bytes at offset `srcFromOffset` through `srcToOffset - 1` in the source segment are compared against the bytes at offset `dstFromOffset` through `dstToOffset - 1` in the destination segment.

If the two segments, over the specified ranges, 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 relative offset within the respective segments. If one segment is a proper prefix of the other, over the specified ranges, then the returned offset is the smallest range, and it follows that the relative offset is only valid for the segment with the larger range. Otherwise, there is no mismatch and `-1` is returned.

Parameters:

`srcSegment` - the source segment.

`srcFromOffset` - the offset (inclusive) of the first byte in the source segment to be tested.

`srcToOffset` - the offset (exclusive) of the last byte in the source segment to be tested.

`dstSegment` - the destination segment.

`dstFromOffset` - the offset (inclusive) of the first byte in the destination segment to be tested.

`dstToOffset` - the offset (exclusive) of the last byte in the destination segment to be tested.

Returns:

the relative offset, in bytes, of the first mismatch between the source and destination segments, otherwise -1 if no mismatch.

Throws:

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

WrongThreadException - if this method is called from a thread `T`, such that `srcSegment.isAccessibleBy(T) == false`.

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

WrongThreadException - if this method is called from a thread `T`, such that `dstSegment.isAccessibleBy(T) == false`.

IndexOutOfBoundsException - if `srcFromOffset < 0`, `srcToOffset < srcFromOffset` or `srcToOffset > srcSegment.byteSize()`

IndexOutOfBoundsException - if `dstFromOffset < 0`, `dstToOffset < dstFromOffset` or `dstToOffset > dstSegment.byteSize()`

See Also:

`mismatch(MemorySegment)`,
`Arrays.mismatch(Object[], int, int, Object[], int, int)`