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

Interface MemoryLayout

All Superinterfaces:

`Constable`

All Known Implementing Classes:

`GroupLayout`, `SequenceLayout`, `ValueLayout`

```
public interface MemoryLayout
extends Constable
```

A memory layout can be used to describe the contents of a memory segment in a *language neutral* fashion. There are two leaves in the layout hierarchy, *value layouts*, which are used to represent values of given size and kind (see `ValueLayout`) and *padding layouts* which are used, as the name suggests, to represent a portion of a memory segment whose contents should be ignored, and which are primarily present for alignment reasons (see `ofPaddingBits(long)`). Some common value layout constants are defined in the `MemoryLayouts` class.

More complex layouts can be derived from simpler ones: a *sequence layout* denotes a repetition of one or more element layout (see `SequenceLayout`); a *group layout* denotes an aggregation of (typically) heterogeneous member layouts (see `GroupLayout`).

For instance, consider the following struct declaration in C:

```
 typedef struct {
     char kind;
     int value;
 } TaggedValues[5];
```

The above declaration can be modelled using a layout object, as follows:

```
SequenceLayout taggedValues = MemoryLayout.ofSequence(5,
    MemoryLayout.ofStruct(
        MemoryLayout.ofValueBits(8, ByteOrder.nativeOrder()).withName("kind"),
        MemoryLayout.ofPaddingBits(24),
        MemoryLayout.ofValueBits(32, ByteOrder.nativeOrder()).withName("value")
    )
).withName("TaggedValues");
```

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

Non-platform classes should not implement MemoryLayout directly.

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

Size, alignment and byte order

All layouts have a size; layout size for value and padding layouts is always explicitly denoted; this means that a layout description always has the same size in bits, regardless of the platform in which it is used. For derived layouts, the size is computed as follows:

- for a *finite* sequence layout *S* whose element layout is *E* and size is L, the size of *S* is that of *E*, multiplied by *L*
- the size of an *unbounded* sequence layout is *unknown*
- for a group layout *G* with member layouts *M1*, *M2*, ... *Mn* whose sizes are *S1*, *S2*, ... *Sn*, respectively, the size of *G* is either *S1 + S2 + ... + Sn* or *max(S1, S2, ... Sn)* depending on whether the group is a *struct* or an *union*, respectively

Furthermore, all layouts feature a *natural alignment* which can be inferred as follows:

- for a padding layout *L*, the natural alignment is 1, regardless of its size; that is, in the absence of an explicit alignment constraint, a padding layout should not affect the alignment constraint of the group layout it is nested into
- for a value layout *L* whose size is *N*, the natural alignment of *L* is *N*
- for a sequence layout *S* whose element layout is *E*, the natural alignment of *S* is that of *E*
- for a group layout *G* with member layouts *M1*, *M2*, ... *Mn* whose alignments are *A1*, *A2*, ... *An*, respectively, the natural alignment of *G* is *max(A1, A2 ... An)*

A layout's natural alignment can be overridden if needed (see `withBitAlignment(long)`), which can be useful to describe hyper-aligned layouts.

All value layouts have an *explicit* byte order (see `ByteOrder`) which is set when the layout is created.

Layout paths

A *layout path* originates from a *root* layout (typically a group or a sequence layout) and terminates at a layout nested within the root layout - this is the layout *selected* by the layout path. Layout paths are typically expressed as a sequence of one or more `MemoryLayout.PathElement` instances.

Layout paths are for example useful in order to obtain offsets of arbitrarily nested layouts inside another layout (see `bitOffset(PathElement...)`), to quickly obtain a memory access handle corresponding to the selected layout (see `varHandle(Class, PathElement...)`), to select an arbitrarily nested layout inside another layout (see `select(PathElement...)`, or to transform a nested layout element inside another layout (see `map(UnaryOperator, PathElement...)`).

Such *layout paths* can be constructed programmatically using the methods in this class. For instance, given the `taggedValues` layout instance constructed as above, we can obtain the offset, in bits, of the member layout named `value` in the *first* sequence element, as follows:

```
long valueOffset = taggedValues.bitOffset(PathElement.sequenceElement(0),
                                          PathElement.groupElement("value")); // yields 32
```

Similarly, we can select the member layout named `value`, as follows:

```
MemoryLayout value = taggedValues.select(PathElement.sequenceElement(),
                                         PathElement.groupElement("value"));
```

And, we can also replace the layout named `value` with another layout, as follows:

```
MemoryLayout taggedValuesWithHole = taggedValues.map(l -> MemoryLayout.ofPadding(32),
                                            PathElement.sequenceElement(), PathElement.groupElement("value"));
```

That is, the above declaration is identical to the following, more verbose one:

```
MemoryLayout taggedValuesWithHole = MemoryLayout.ofSequence(5,
    MemoryLayout.ofStruct(
        MemoryLayout.ofValueBits(8, ByteOrder.nativeOrder()).withName("kind").
        MemoryLayout.ofPaddingBits(32),
        MemoryLayout.ofPaddingBits(32)
));
```

Layout paths can feature one or more *free dimensions*. For instance, a layout path traversing an unspecified sequence element (that is, where one of the path component was obtained with the `MemoryLayout.PathElement.sequenceElement()` method) features an additional free dimension, which will have to be bound at runtime. This is important when obtaining memory access var handle from layouts, as in the following code:

```
VarHandle valueHandle = taggedValues.varHandle(int.class,
                                               PathElement.sequenceElement(),
                                               PathElement.groupElement("value"));
```

Since the layout path constructed in the above example features exactly one free dimension (as it doesn't specify *which* member layout named `value` should be selected from the enclosing sequence layout), it follows that the memory access var handle `valueHandle` will feature an *additional* `long` access coordinate.

A layout path with free dimensions can also be used to create an offset-computing method handle, using the `bitOffset(PathElement...)` or `byteOffsetHandle(PathElement...)` method. Again, free dimensions are translated into `long` parameters of the created method handle. The method handle can be used to compute the offsets of elements of a sequence at different indices, by supplying these indices when invoking the method handle. For instance:

```
MethodHandle offsetHandle = taggedValues.byteOffsetHandle(PathElement.sequenceElement(),
                                                          PathElement.groupElement("kind"));
long offset1 = (long) offsetHandle.invokeExact(1L); // 8
long offset2 = (long) offsetHandle.invokeExact(2L); // 16
```

Layout attributes

Layouts can be optionally associated with one or more *attributes*. A layout attribute forms a *name/value* pair, where the name is a `String` and the value is a `Constable`. The most common form of layout attribute is the *layout name* (see `LAYOUT_NAME`), a custom name that can be associated to memory layouts and that can be referred to when constructing *layout paths*.

API Note:

In the future, if the Java language permits, `MemoryLayout` may become a `sealed` interface, which would prohibit subclassing except by explicitly permitted types.

Implementation Requirements:

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

Nested Class Summary

Nested Classes

Modifier and Type	Interface	Description
`static interface`	`MemoryLayout.PathElement`	Instances of this class are used to form *layout paths*.

Field Summary

Fields

Modifier and Type	Field	Description
`static String`	`LAYOUT_NAME`	Attribute name used to specify the *name* property of a memory layout (see `name()` and `withName(String)`).

Method Summary

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

Modifier and Type	Method	Description
`Optional<Constable>`	`attribute(String name)`	Returns the attribute with the given name (if it exists).
`Stream<String>`	`attributes()`	Returns a stream of the attribute names associated with this layout.
`long`	`bitAlignment()`	Returns the alignment constraint associated with this layout, expressed in bits.
`default long`	`bitOffset (MemoryLayout.PathElement... elements`	Computes the offset, in bits, of the layout selected by a given layout path, where the path is considered rooted in this layout.
`default MethodHandle`	`bitOffsetHandle (MemoryLayout.PathElement... elements`	Creates a method handle that can be used to compute the offset, in bits, of the layout selected by a given layout path, where the path is considered rooted in this layout.
`long`	`bitSize()`	Computes the layout size, in bits.
`default long`	`byteAlignment()`	Returns the alignment constraint associated with this layout, expressed in bytes.
`default long`	`byteOffset (MemoryLayout.PathElement... elements`	Computes the offset, in bytes, of the layout selected by a given layout path, where the path is considered rooted in this layout.
`default MethodHandle`	`byteOffsetHandle (MemoryLayout.PathElement... elements`	Creates a method handle that can be used to compute the offset, in bytes, of the layout selected by a given layout path, where the path is considered rooted in this layout.
`default long`	`byteSize()`	Computes the layout size, in bytes.

`Optional<? extends DynamicConstantDesc<? extends MemoryLayout>>`	`describeConstable()`	Returns an `Optional` containing the nominal descriptor for this layout, if one can be constructed, or an empty `Optional` if one cannot be constructed.
`boolean`	`equals(Object that)`	Compares the specified object with this layout for equality.
`int`	`hashCode()`	Returns the hash code value for this layout.
`boolean`	`hasSize()`	Does this layout have a specified size?
`boolean`	`isPadding()`	Is this a padding layout (e.g. a layout created from `ofPaddingBits(long)`) ?
`default MemoryLayout`	`map(UnaryOperator<MemoryLayout> op, MemoryLayout.PathElement... elements)`	Creates a transformed copy of this layout where a selected layout, from a path rooted in this layout, is replaced with the result of applying the given operation.
`Optional<String>`	`name()`	Return the *name* (if any) associated with this layout.
`static MemoryLayout`	`ofPaddingBits(long size)`	Create a new padding layout with given size.
`static SequenceLayout`	`ofSequence(long elementCount, MemoryLayout elementLayout)`	Create a new sequence layout with given element layout and element count.
`static SequenceLayout`	`ofSequence (MemoryLayout elementLayout)`	Create a new sequence layout, with unbounded element count and given element layout.
`static GroupLayout`	`ofStruct(MemoryLayout... elements)`	Create a new *struct* group layout with given member layouts.
`static GroupLayout`	`ofUnion(MemoryLayout... elements)`	Create a new *union* group layout with given member layouts.
`static ValueLayout`	`ofValueBits(long size, ByteOrder order)`	Create a value layout of given byte order and size.
`default MemoryLayout`	`select (MemoryLayout.PathElement... elements`	Selects the layout from a path rooted in this layout.
`String`	`toString()`	Returns a string representation of this layout.
`default VarHandle`	`varHandle(Class<?> carrier, MemoryLayout.PathElement... elements)`	Creates a memory access var handle that can be used to dereference memory at the layout selected by a given layout path, where the path is considered rooted in this layout.
`MemoryLayout`	`withAttribute(String name, Constable value)`	Returns a new memory layout which features the same attributes as this layout, plus the newly specified attribute.
`MemoryLayout`	`withBitAlignment(long bitAlignment)`	Creates a new layout which features the desired alignment constraint.
`MemoryLayout`	`withName(String name)`	Creates a new layout which features the desired layout *name*.

Field Details

LAYOUT_NAME

```
static final String LAYOUT_NAME
```

Attribute name used to specify the *name* property of a memory layout (see `name()` and `withName(String)`).

See Also:

Constant Field Values

Method Details

describeConstable

`Optional<? extends DynamicConstantDesc<? extends MemoryLayout>> describeConstable()`

Returns an `Optional` containing the nominal descriptor for this layout, if one can be constructed, or an empty `Optional` if one cannot be constructed.

Specified by:

`describeConstable` in interface `Constable`

Returns:

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

hasSize

`boolean hasSize()`

Does this layout have a specified size? A layout does not have a specified size if it is (or contains) a sequence layout whose size is unspecified (see `SequenceLayout.elementCount()`). Value layouts (see `ValueLayout`) and padding layouts (see `ofPaddingBits(long)`) *always* have a specified size, therefore this method always returns `true` in these cases.

Returns:

`true`, if this layout has a specified size.

bitSize

`long bitSize()`

Computes the layout size, in bits.

Returns:

the layout size, in bits.

Throws:

`UnsupportedOperationException` - if the layout is, or contains, a sequence layout with unspecified size (see `SequenceLayout`).

byteSize

`default long byteSize()`

Computes the layout size, in bytes.

Returns:

the layout size, in bytes.

Throws:

`UnsupportedOperationException` - if the layout is, or contains, a sequence layout with unspecified size (see `SequenceLayout`), or if `bitSize()` is not a multiple of 8.

name

`Optional<String> name()`

Return the *name* (if any) associated with this layout.

This is equivalent to the following code:

```
        attribute(LAYOUT_NAME).map(String.class::cast);
```

Returns:

the layout *name* (if any).

See Also:

`withName(String)`

withName

`MemoryLayout withName(String name)`

Creates a new layout which features the desired layout *name*.

This is equivalent to the following code:

```
        withAttribute(LAYOUT_NAME, name);
```

Parameters:

`name` - the layout name.

Returns:

a new layout which is the same as this layout, except for the *name* associated to it.

See Also:

`name()`

bitAlignment

`long bitAlignment()`

Returns the alignment constraint associated with this layout, expressed in bits. Layout alignment defines a power of two `A` which is the bit-wise alignment of the layout. If `A <= 8` then `A/8` is the number of bytes that must be aligned for any pointer that correctly points to this layout. Thus:

- `A=8` means unaligned (in the usual sense), which is common in packets.
- `A=64` means word aligned (on LP64), `A=32` int aligned, `A=16` short aligned, etc.
- `A=512` is the most strict alignment required by the x86/SV ABI (for AVX-512 data).

If no explicit alignment constraint was set on this layout (see `withBitAlignment(long)`), then this method returns the natural alignment constraint (in bits) associated with this layout.

Returns:

the layout alignment constraint, in bits.

byteAlignment

`default long byteAlignment()`

Returns the alignment constraint associated with this layout, expressed in bytes. Layout alignment defines a power of two `A` which is the byte-wise alignment of the layout, where `A` is the number of bytes that must be aligned for any pointer that correctly points to this layout. Thus:

- `A=1` means unaligned (in the usual sense), which is common in packets.
- `A=8` means word aligned (on LP64), `A=4` int aligned, `A=2` short aligned, etc.
- `A=64` is the most strict alignment required by the x86/SV ABI (for AVX-512 data).

If no explicit alignment constraint was set on this layout (see `withBitAlignment(long)`), then this method returns the natural alignment constraint (in bytes) associated with this layout.

Returns:

the layout alignment constraint, in bytes.

Throws:

`UnsupportedOperationException` - if `bitAlignment()` is not a multiple of 8.

withBitAlignment

`MemoryLayout withBitAlignment(long bitAlignment)`

Creates a new layout which features the desired alignment constraint.

Parameters:

`bitAlignment` - the layout alignment constraint, expressed in bits.

Returns:

a new layout which is the same as this layout, except for the alignment constraint associated to it.

Throws:

`IllegalArgumentException` - if `bitAlignment` is not a power of two, or if it's less than than 8.

attribute

```
Optional<Constable> attribute(String name)
```

Returns the attribute with the given name (if it exists).

Parameters:

`name` - the attribute name

Returns:

the attribute with the given name (if it exists).

withAttribute

```
MemoryLayout withAttribute(String name,
                           Constable value)
```

Returns a new memory layout which features the same attributes as this layout, plus the newly specified attribute. If this layout already contains an attribute with the same name, the existing attribute value is overwritten in the returned layout.

Parameters:

`name` - the attribute name.

`value` - the attribute value.

Returns:

a new memory layout which features the same attributes as this layout, plus the newly specified attribute.

attributes

```
Stream<String> attributes()
```

Returns a stream of the attribute names associated with this layout.

Returns:

a stream of the attribute names associated with this layout.

bitOffset

```
default long bitOffset(MemoryLayout.PathElement... elements)
```

Computes the offset, in bits, of the layout selected by a given layout path, where the path is considered rooted in this layout.

Parameters:

`elements` - the layout path elements.

Returns:

The offset, in bits, of the layout selected by the layout path in `elements`.

Throws:

`IllegalArgumentException` - if the layout path does not select any layout nested in this layout, or if the layout path contains one or more path elements that select multiple sequence element indices (see `MemoryLayout.PathElement.sequenceElement()` and `MemoryLayout.PathElement.sequenceElement(long, long)`).

`UnsupportedOperationException` - if one of the layouts traversed by the layout path has unspecified size.

bitOffsetHandle

```
default MethodHandle bitOffsetHandle(MemoryLayout.PathElement... elements)
```

Creates a method handle that can be used to compute the offset, in bits, of the layout selected by a given layout path, where the path is considered rooted in this layout.

The returned method handle has a return type of `long`, and features as many `long` parameter types as there are free dimensions in the provided layout path (see `MemoryLayout.PathElement.sequenceElement()`, where the order of the parameters corresponds to the order of the path elements. The returned method handle can be used to compute a layout offset similar to `bitOffset(PathElement...)`, but where some sequence indices are specified only when invoking the method handle.

The final offset returned by the method handle is computed as follows:

```
        offset = c_1 + c_2 + ... + c_m + (x_1 * s_1) + (x_2 * s_2) + ... + (x_n * s_n)
```

where `x_1`, `x_2`, ... `x_n` are *dynamic* values provided as `long` arguments, whereas `c_1`, `c_2`, ... `c_m` and `s_0`, `s_1`, ... `s_n` are *static* stride constants which are derived from the layout path.

Parameters:

`elements` - the layout path elements.

Returns:

a method handle that can be used to compute the bit offset of the layout element specified by the given layout path elements, when supplied with the missing sequence element indices.

Throws:

`IllegalArgumentException` - if the layout path contains one or more path elements that select multiple sequence element indices (see `MemoryLayout.PathElement.sequenceElement(long, long)`).

`UnsupportedOperationException` - if one of the layouts traversed by the layout path has unspecified size.

byteOffset

```
default long byteOffset(MemoryLayout.PathElement... elements)
```

Computes the offset, in bytes, of the layout selected by a given layout path, where the path is considered rooted in this layout.

Parameters:

`elements` - the layout path elements.

Returns:

The offset, in bytes, of the layout selected by the layout path in `elements`.

Throws:

`IllegalArgumentException` - if the layout path does not select any layout nested in this layout, or if the layout path contains one or more path elements that select multiple sequence element indices (see `MemoryLayout.PathElement.sequenceElement()` and `MemoryLayout.PathElement.sequenceElement(long, long)`).

`UnsupportedOperationException` - if one of the layouts traversed by the layout path has unspecified size, or if `bitOffset(elements)` is not a multiple of 8.

byteOffsetHandle

```
default MethodHandle byteOffsetHandle(MemoryLayout.PathElement... elements)
```

Creates a method handle that can be used to compute the offset, in bytes, of the layout selected by a given layout path, where the path is considered rooted in this layout.

The returned method handle has a return type of `long`, and features as many `long` parameter types as there are free dimensions in the provided layout path (see `MemoryLayout.PathElement.sequenceElement()`, where the order of the parameters corresponds to the order of the path elements. The returned method handle can be used to compute a layout offset similar to `byteOffset(PathElement...)`, but where some sequence indices are specified only when invoking the method handle.

The final offset returned by the method handle is computed as follows:

```
        bitOffset = c_1 + c_2 + ... + c_m + (x_1 * s_1) + (x_2 * s_2) + ... + (x_n * s_n)
        offset = bitOffset / 8
```

where `x_1`, `x_2`, ... `x_n` are *dynamic* values provided as `long` arguments, whereas `c_1`, `c_2`, ... `c_m` and `s_0`, `s_1`, ... `s_n` are *static* stride constants which are derived from the layout path.

The method handle will throw an `UnsupportedOperationException` if the computed offset in bits is not a multiple of 8.

Parameters:

`elements` - the layout path elements.

Returns:

a method handle that can be used to compute the byte offset of the layout element specified by the given layout path elements, when supplied with the missing sequence element indices.

Throws:

`IllegalArgumentException` - if the layout path contains one or more path elements that select multiple sequence element indices (see `MemoryLayout.PathElement.sequenceElement(long, long)`).

`UnsupportedOperationException` - if one of the layouts traversed by the layout path has unspecified size.

varHandle

```
default VarHandle varHandle(Class<?> carrier,
                            MemoryLayout.PathElement... elements)
```

Creates a memory access var handle that can be used to dereference memory at the layout selected by a given layout path, where the path is considered rooted in this layout.

The final memory location accessed by the returned memory access var handle can be computed as follows:

```
        address = base + offset
```

where `base` denotes the base address expressed by the `MemorySegment` access coordinate (see `MemorySegment.address()` and `MemoryAddress.toRawLongValue()`) and `offset` can be expressed in the following form:

```
        offset = c_1 + c_2 + ... + c_m + (x_1 * s_1) + (x_2 * s_2) + ... + (x_n * s_n)
```

where `x_1`, `x_2`, ... `x_n` are *dynamic* values provided as optional `long` access coordinates, whereas `c_1`, `c_2`, ... `c_m` and `s_0`, `s_1`, ... `s_n` are *static* stride constants which are derived from the layout path.

API Note:

the resulting var handle will feature an additional `long` access coordinate for every unspecified sequence access component contained in this layout path. Moreover, the resulting var handle features certain access mode restrictions, which are common to all memory access var handles.

Parameters:

`carrier` - the var handle carrier type.

`elements` - the layout path elements.

Returns:

a var handle which can be used to dereference memory at the (possibly nested) layout selected by the layout path in `elements`.

Throws:

`UnsupportedOperationException` - if the layout path has one or more elements with incompatible alignment constraints, or if one of the layouts traversed by the layout path has unspecified size.

`IllegalArgumentException` - if the carrier does not represent a primitive type, if the carrier is `void`, `boolean`, or if the layout path in `elements` does not select a value layout (see `ValueLayout`), or if the selected value layout has a size that that does not match that of the specified carrier type.

select

```
default MemoryLayout select(MemoryLayout.PathElement... elements)
```

Selects the layout from a path rooted in this layout.

Parameters:

`elements` - the layout path elements.

Returns:

the layout selected by the layout path in `elements`.

Throws:

`IllegalArgumentException` - if the layout path does not select any layout nested in this layout, or if the layout path contains one or more path elements that select one or more sequence element indices (see `MemoryLayout.PathElement.sequenceElement(long)` and `MemoryLayout.PathElement.sequenceElement(long, long)`).

map

```
default MemoryLayout map(UnaryOperator<MemoryLayout> op,
                         MemoryLayout.PathElement... elements)
```

Creates a transformed copy of this layout where a selected layout, from a path rooted in this layout, is replaced with the result of applying the given operation.

Parameters:

`op` - the unary operation to be applied to the selected layout.

`elements` - the layout path elements.

Returns:

a new layout where the layout selected by the layout path in `elements`, has been replaced by the result of applying `op` to the selected layout.

Throws:

`IllegalArgumentException` - if the layout path does not select any layout nested in this layout, or if the layout path contains one or more path elements that select one or more sequence element indices (see `MemoryLayout.PathElement.sequenceElement(long)` and `MemoryLayout.PathElement.sequenceElement(long, long)`).

isPadding

```
boolean isPadding()
```

Is this a padding layout (e.g. a layout created from `ofPaddingBits(long)`) ?

Returns:

true, if this layout is a padding layout.

equals

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

Compares the specified object with this layout for equality. Returns `true` if and only if the specified object is also a layout, and it is equal to this layout. Two layouts are considered equal if they are of the same kind, have the same size, name and alignment constraints. Furthermore, depending on the layout kind, additional conditions must be satisfied:

- two value layouts are considered equal if they have the same byte order (see `ValueLayout.order()`)
- two sequence layouts are considered equal if they have the same element count (see `SequenceLayout.elementCount()`), and if their element layouts (see `SequenceLayout.elementLayout()`) are also equal
- two group layouts are considered equal if they are of the same kind (see `GroupLayout.isStruct()`, `GroupLayout.isUnion()`) and if their member layouts (see `GroupLayout.memberLayouts()`) are also equal

Overrides:

`equals` in class `Object`

Parameters:

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

Returns:

`true` if the specified object is equal to this layout.

See Also:

`Object.hashCode()`, `HashMap`

hashCode

```
int hashCode()
```

Returns the hash code value for this layout.

Overrides:

`hashCode` in class `Object`

Returns:

the hash code value for this layout.

See Also:

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

toString

```
String toString()
```

Returns a string representation of this layout.

Overrides:

`toString` in class `Object`

Returns:

a string representation of this layout.

ofPaddingBits

```
static MemoryLayout ofPaddingBits(long size)
```

Create a new padding layout with given size.

Parameters:

`size` - the padding size in bits.

Returns:

the new selector layout.

Throws:

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

ofValueBits

```
static ValueLayout ofValueBits(long size,
                               ByteOrder order)
```

Create a value layout of given byte order and size.

Parameters:

`size` - the value layout size.

`order` - the value layout's byte order.

Returns:

a new value layout.

Throws:

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

ofSequence

```
static SequenceLayout ofSequence(long elementCount,
                                 MemoryLayout elementLayout)
```

Create a new sequence layout with given element layout and element count.

Parameters:

`elementCount` - the sequence element count.

`elementLayout` - the sequence element layout.

Returns:

the new sequence layout with given element layout and size.

Throws:

`IllegalArgumentException` - if `elementCount < 0`.

ofSequence

```
static SequenceLayout ofSequence(MemoryLayout elementLayout)
```

Create a new sequence layout, with unbounded element count and given element layout.

Parameters:

`elementLayout` - the element layout of the sequence layout.

Returns:

the new sequence layout with given element layout.

ofStruct

```
static GroupLayout ofStruct(MemoryLayout... elements)
```

Create a new *struct* group layout with given member layouts.

Parameters:

`elements` - The member layouts of the *struct* group layout.

Returns:

a new *struct* group layout with given member layouts.

ofUnion

```
static GroupLayout ofUnion(MemoryLayout... elements)
```

Create a new *union* group layout with given member layouts.

Parameters:

`elements` - The member layouts of the *union* layout.

Returns:

a new *union* group layout with given member layouts.