# JEP 419: Foreign Function & Memory API (Second Incubator)

| | |
|---|---|
| *Owner* | Maurizio Cimadamore |
| *Type* | Feature |
| *Scope* | JDK |
| *Status* | Closed / Delivered |
| *Release* | 18 |
| *Component* | core-libs |
| *Discussion* | panama dash dev at openjdk dot java dot net |
| *Relates to* | JEP 412: Foreign Function & Memory API (Incubator)<br>JEP 424: Foreign Function & Memory API (Preview) |
| *Reviewed by* | Jim Laskey, Paul Sandoz |
| *Created* | 2021/09/21 11:56 |
| *Updated* | 2023/05/12 15:34 |
| *Issue* | 8274073 |

## Summary

Introduce an API by which Java programs can interoperate with code and data outside of the Java runtime. By efficiently invoking foreign functions (i.e., code outside the JVM), and by safely accessing foreign memory (i.e., memory not managed by the JVM), the API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI.

## History

The Foreign Function & Memory API was proposed by JEP 412 and targeted to Java 17 in mid 2021 as an incubating API. It combined two earlier incubating APIs: the Foreign-Memory Access API and the Foreign Linker API. This JEP proposes to incorporate refinements based on feedback, and to re-incubate the API in Java 18. The following changes are included in this refresh:

- Support for more carriers, such as `boolean` and `MemoryAddress`, in memory access var handles;
- A more general dereference API, available in both the `MemorySegment` and `MemoryAddress` interfaces;
- A simpler API to obtain downcall method handles, where passing a `MethodType` parameter is no longer required;
- A simpler API to manage temporal dependencies between resource scopes; and
- A new API to copy Java arrays to and from memory segments.

## Goals

- *Ease of use* — Replace the Java Native Interface (JNI) with a superior, pure-Java development model.
- *Performance* — Provide performance that is comparable to, if not better than, existing APIs such as JNI and `sun.misc.Unsafe`.
- *Generality* — Provide ways to operate on different kinds of foreign memory (e.g., native memory, persistent memory, and managed heap memory) and, over time, to accommodate other platforms (e.g., 32-bit x86) and foreign functions written in languages other than C (e.g., C++, Fortran).
- *Safety* — Disable unsafe operations by default, allowing them only after explicit opt-in from application developers or end users.

## Non-goals

It is not a goal to

- Re-implement JNI on top of this API, or otherwise change JNI in any way;
- Re-implement legacy Java APIs, such as `sun.misc.Unsafe`, on top of this API;
- Provide tooling that mechanically generates Java code from native-code header files; or
- Change how Java applications that interact with native libraries are packaged and deployed (e.g., via multi-platform JAR files).

## Motivation

The Java Platform has always offered a rich foundation to library and application developers who wish to reach beyond the JVM and interact with other platforms. Java APIs expose non-Java resources conveniently and reliably, whether to access remote data (JDBC), invoke web services (HTTP client), serve remote clients (NIO channels), or communicate with local processes (Unix-domain sockets). Unfortunately, Java developers still face significant obstacles in accessing an important kind of non-Java resource: code and data on the same machine as the JVM, but outside the Java runtime.

### *Foreign memory*

Data stored in memory outside the Java runtime is referred to as *off-heap* data. (The *heap* is where Java objects live — *on-heap* data — and where garbage collectors do their work.) Accessing off-heap data is critical for the performance of popular Java libraries such as Tensorflow, Ignite, Lucene, and Netty, primarily because it lets them avoid the cost and unpredictability associated with garbage

collection. It also allows data structures to be serialized and deserialized by mapping files into memory via, e.g., `mmap`. However, the Java Platform does not provide a satisfactory solution for accessing off-heap data.

- The `ByteBuffer` API allows for the creation of *direct* byte buffers that are allocated off-heap, but their maximum size is two gigabytes and they are not deallocated promptly. These and other limitations stem from the fact that the `ByteBuffer` API was designed not only for off-heap memory access but also for producer/consumer exchanges of bulk data in areas such as charset encoding/decoding and partial I/O operations. In that context it has not been possible to satisfy the many requests for off-heap enhancements filed over the years (e.g., 4496703, 6558368, 4837564, and 5029431).
- The `sun.misc.Unsafe` API exposes memory access operations for on-heap data that also work for off-heap data. Using `Unsafe` is efficient because its memory access operations are defined as HotSpot JVM intrinsics and optimized by the JIT compiler. However, using `Unsafe` is dangerous because it allows access to any memory location. This means that a Java program can crash the JVM by accessing an already-freed location; for this and other reasons, the use of `Unsafe` has always been strongly discouraged.
- Using JNI to call a native library which then accesses off-heap data is possible, but the performance overhead seldom makes it applicable: Going from Java to native is several orders of magnitude slower than accessing memory because JNI method calls do not benefit from many common JIT optimizations such as inlining.

In summary, when it comes to accessing off-heap data, Java developers face a dilemma: Should they choose a safe but inefficient path (`ByteBuffer`) or should they abandon safety in favor of performance (`Unsafe`)? What they in fact require is a supported API for accessing off-heap data (i.e., foreign memory) designed from the ground up to be safe and with JIT optimizations in mind.

***Foreign functions***

JNI has supported the invocation of native code (i.e., foreign functions) since Java 1.1, but it is inadequate for many reasons.

- JNI involves several tedious artifacts: a Java API (`native` methods), a C header file derived from the Java API, and a C implementation that calls the native library of interest. Java developers must work across multiple toolchains to keep platform-dependent artifacts in sync, which is especially burdensome when the native library evolves rapidly.
- JNI can only interoperate with libraries written in languages, typically C and C++, that use the calling convention of the operating system and CPU for which the JVM was built. A `native` method cannot be used to invoke a function written in a language that uses a different convention.
- JNI does not reconcile the Java type system with the C type system. Aggregate data in Java is represented with objects, but aggregate data in C is represented with structs, so any Java object passed to a `native` method must be laboriously unpacked by native code. For example, consider a record class `Person` in Java: Passing a `Person` object to a `native` method will require the native code to use JNI's C API to extract fields (e.g., `firstName` and `lastName`) from the object. As a result, Java developers sometimes flatten their data into a single object (e.g., a byte array or a direct byte buffer) but more often, since passing Java objects via JNI is slow, they use the `Unsafe` API to allocate off-heap memory and pass its address to a `native` method as a `long` — which makes the Java code tragically unsafe!

Over the years, numerous frameworks have emerged to fill the gaps left by JNI, including JNA, JNR and JavaCPP. While these frameworks are often a marked improvement over JNI, the situation is still less than ideal, especially when compared with languages which offer first-class native interoperation. For example, Python's ctypes package can dynamically wrap functions in native libraries without any glue code. Other languages, such as Rust, provide tools which mechanically derive native wrappers from C/C++ header files.

Ultimately, Java developers should have a supported API that lets them straightforwardly consume any native library deemed useful for a particular task, without the tedious glue and clunkiness of JNI. An excellent abstraction to build upon is *method handles*, introduced in Java 7 to support fast dynamic languages on the JVM. Exposing native code via method handles would radically simplify the task of writing, building, and distributing Java libraries which depend upon native libraries. Furthermore, an API capable of modeling foreign functions (i.e., native code) and foreign memory (i.e., off-heap data) would provide a solid foundation for third-party native interoperation frameworks.

## Description

The Foreign Function & Memory API (FFM API) defines classes and interfaces so that client code in libraries and applications can

- Allocate foreign memory (`MemorySegment`, `MemoryAddress`, and `SegmentAllocator`),
- Manipulate and access structured foreign memory (`MemoryLayout`, `VarHandle`),

- Manage the lifecycle of foreign resources (`ResourceScope`), and
- Call foreign functions (`SymbolLookup`, `CLinker`, and `NativeSymbol`).

The FFM API resides in the `jdk.incubator.foreign` package of the `jdk.incubator.foreign` module.

***Example***

As a brief example of using the FFM API, here is Java code that obtains a method handle for a C library function `radixsort` and then uses it to sort four strings which start life in a Java array (a few details are elided):

```
// 1. Find foreign function on the C library path
CLinker linker = CLinker.getInstance();
MethodHandle radixSort = linker.downcallHandle(
                             linker.lookup("radixsort"), ...);
// 2. Allocate on-heap memory to store four strings
String[] javaStrings   = { "mouse", "cat", "dog", "car" };
// 3. Allocate off-heap memory to store four pointers
MemorySegment offHeap  = MemorySegment.allocateNative(
                             MemoryLayout.ofSequence(javaStrings.length,
                                                     ValueLayout.ADDRESS), ...);
// 4. Copy the strings from on-heap to off-heap
for (int i = 0; i < javaStrings.length; i++) {
    // Allocate a string off-heap, then store a pointer to it
    MemorySegment cString = implicitAllocator().allocateUtf8String(javaStrings[i]);
    offHeap.setAtIndex(ValueLayout.ADDRESS, i, cString);
}
// 5. Sort the off-heap data by calling the foreign function
radixSort.invoke(offHeap, javaStrings.length, MemoryAddress.NULL, '\0');
// 6. Copy the (reordered) strings from off-heap to on-heap
for (int i = 0; i < javaStrings.length; i++) {
    MemoryAddress cStringPtr = offHeap.getAtIndex(ValueLayout.ADDRESS, i);
    javaStrings[i] = cStringPtr.getUtf8String(0);
}
assert Arrays.equals(javaStrings, new String[] {"car", "cat", "dog", "mouse"});  // true
```

This code is far clearer than any solution that uses JNI, since implicit conversions and memory dereferences that would have been hidden behind `native` method calls are now expressed directly in Java. Modern Java idioms can also be used; for example, streams can allow for multiple threads to copy data between on-heap and off-heap memory in parallel.

***Memory segments***

A *memory segment* is an abstraction that models a contiguous region of memory, located either off-heap or on-heap. Memory segments can be

- *Native* segments, allocated from scratch in native memory (e.g., via `malloc`),
- *Mapped* segments, wrapped around a region of mapped native memory (e.g., via `mmap`), or
- *Array* or *buffer* segments, wrapped around memory associated with existing Java arrays or byte buffers, respectively.

All memory segments provide strongly enforced spatial, temporal, and thread-confinement guarantees which make memory dereference operations safe. For example, the following code allocates 100 bytes off-heap:

```
MemorySegment segment = MemorySegment.allocateNative(100,
                                                     newImplicitScope());
```

The *spatial bounds* of a segment determine the range of memory addresses associated with the segment. The bounds of the segment in the code above are defined by a *base address* `b`, expressed as a `MemoryAddress` instance, and a size in bytes (100), resulting in a range of addresses from `b` to `b + 99`, inclusive.

The *temporal bounds* of a segment determine the lifetime of the segment, that is, when the segment will be deallocated. A segment's lifetime and thread-confinement state is modeled by a `ResourceScope` abstraction, discussed below. The resource scope in the code above is a new *shared* scope, which ensures that the memory associated with this segment is freed when the `MemorySegment` object is deemed unreachable by the garbage collector. The shared scope also ensures that the memory segment is accessible from multiple threads.

In other words, the code above creates a segment whose behavior closely matches that of a `ByteBuffer` allocated with the `allocateDirect` factory. The FFM API also supports deterministic memory release and other thread-confinement options, discussed below.

***Dereferencing segments***

To dereference some data in a memory segment, we need to take into account several factors:

- The number of bytes to be dereferenced,
- The alignment constraints of the address at which dereference occurs,
- The endianness with which bytes are stored in said memory region, and
- The Java type to be used in the dereference operation (e.g. `int` vs `float`).

All these characteristics are captured in the `ValueLayout` abstraction. For example, the predefined `JAVA_INT` value layout is four bytes wide, has no alignment

constraints, uses the native platform endianness (e.g., little-endian on Linux/x64), and is associated with the Java type `int`.

Memory segments have simple dereference methods to read and write values from and to memory segments. These methods accept a value layout, which uniquely specifies the properties of the dereference operation. For example, we can write 25 `int` values at consecutive offsets in a memory segment using the following code:

```
MemorySegment segment = MemorySegment.allocateNative(100,
                                                     newImplicitScope());
for (int i = 0; i < 25; i++) {
    segment.setAtIndex(ValueLayout.JAVA_INT,
                       /* index */ i,
                       /* value to write */ i);
}
```

***Memory layouts and structured access***

Consider the following C declaration, which defines an array of `Point` structs, where each `Point` struct has two members, namely `Point.x` and `Point.y`:

```
struct Point {
   int x;
   int y;
} pts[10];
```

Using the dereference methods shown in the previous section, to initialize such a native array we would have to write the following code:

```
MemorySegment segment = MemorySegment.allocateNative(2 * 4 * 10,
                                                     newImplicitScope());
for (int i = 0; i < 10; i++) {
    segment.setAtIndex(ValueLayout.JAVA_INT,
                       /* index */ (i * 2),
                       /* value to write */ i); // x
    segment.setAtIndex(ValueLayout.JAVA_INT,
                       /* index */ (i * 2) + 1,
                       /* value to write */ i); // y
}
```

To reduce the need for tedious calculations about memory layout (e.g., `(i * 2) + 1` in the example above), a `MemoryLayout` can be used to describe the content of a memory segment in a more declarative fashion. For example, the desired layout of the native memory segment in the examples above can be described in the following way:

```
SequenceLayout ptsLayout
    = MemoryLayout.sequenceLayout(10,
                                  MemoryLayout.structLayout(
                                      ValueLayout.JAVA_INT.withName("x"),
                                      ValueLayout.JAVA_INT.withName("y")));
```

This creates a *sequence memory layout* containing ten repetitions of a *struct layout* whose elements are two `JAVA_INT` layouts named `x` and `y`, respectively. Given this layout, we can avoid calculating offsets in our code by creating a *memory-access var handle*, a special kind of var handle which accepts a `MemorySegment` parameter (the segment to be dereferenced) followed by one or more `long` coordinates (the indices at which the dereference operation should occur):

```
VarHandle xHandle    // (MemorySegment, long) -> int
    = ptsLayout.varHandle(PathElement.sequenceElement(),
                          PathElement.groupElement("x"));
VarHandle yHandle    // (MemorySegment, long) -> int
    = ptsLayout.varHandle(PathElement.sequenceElement(),
                          PathElement.groupElement("y"));

MemorySegment segment = MemorySegment.allocateNative(ptsLayout,
                                                     newImplicitScope());
for (int i = 0; i < ptsLayout.elementCount().getAsLong(); i++) {
    xHandle.set(segment,
                /* index */ (long) i,
                /* value to write */ i); // x
    yHandle.set(segment,
                /* index */ (long) i,
                /* value to write */ i); // y
}
```

The `ptsLayout` object drives the creation of the memory-access var handle through the creation of a *layout path*, which is used to select a nested layout from a complex layout expression. Since the selected value layout is associated with the Java type `int`, the type of the resulting var handles `xHandle` and `yHandle` will also be `int`. Moreover, since the selected value layout is defined inside a sequence layout, the resulting var handles acquire an extra coordinate of type `long`, namely the index of the `Point` struct whose coordinate is to be read or written. The `ptsLayout` object also drives the allocation of the native memory segment, which is based upon size and alignment information derived from the layout. Offset computations are no longer needed inside the loop since distinct var handles are used to initialize the `Point.x` and `Point.y` elements.

***Resource scopes***

All of the examples above use non-deterministic deallocation: The memory associated with the allocated segments is deallocated by the garbage collector after the memory segment instance becomes unreachable. We say that such segments are *implicitly deallocated*.

There are cases where the client might want to control when memory deallocation occurs. Suppose, e.g., that a large memory segment is mapped from a file using `MemorySegment::map`. The client might prefer to release (i.e., unmap) the memory associated with the segment as soon as the segment is no longer required rather than wait for the garbage collector to do so, since waiting could adversely affect the application's performance.

Memory segments support deterministic deallocation through *resource scopes*. A resource scope models the lifecycle of one or more *resources,* such as memory segments. A newly-created resource scope is in the *alive* state, which means that all the resources it manages can be accessed safely. At the client's request a resource scope can be *closed* so that access to the resources managed by the scope is no longer allowed. The `ResourceScope` class implements the `AutoCloseable` interface so that resource scopes work with the try-with-resources statement:

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    MemorySegment s1 = MemorySegment.map(Path.of("someFile"),
                                         0, 100000,
                                         MapMode.READ_WRITE, scope);
    MemorySegment s2 = MemorySegment.allocateNative(100, scope);
    ...
} // both segments released here
```

This code creates a resource scope and uses it to create two segments: a mapped segment (s1) and a native segment (s2). The lifecycle of the two segments is tied to the lifetime of the resource scope, so accessing the segments (e.g., dereferencing them with memory-access var handles) outside of the try-with-resources statement will cause a runtime exception to be thrown.

In addition to managing a memory segment's lifetime, a resource scope also controls which threads can access the segment. A *confined* resource scope restricts access to the thread which created the scope, whereas a *shared* resource scope allows access from any thread.

Resource scopes, whether confined or shared, can be associated with a `java.lang.ref.Cleaner` object that performs implicit deallocation in case the resource scope becomes unreachable while the scope is still alive, thus preventing accidental memory leaks.

***Segment allocators***

Memory allocation can often be a bottleneck when clients use off-heap memory. The FFM API therefore includes a `SegmentAllocator` abstraction, which defines useful operations to allocate and initialize memory segments. Segment allocators are obtained via factories in the `SegmentAllocator` interface. One such factory returns the *implicit allocator*, that is, an allocator which allocates native segments backed by a fresh implicit scope. Other, more optimized allocators are also provided. For example, the following code creates an arena-based allocator and uses it to allocate a segment whose content is initialized from a Java `int` array:

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    SegmentAllocator allocator = SegmentAllocator.newNativeArena(scope);
    for (int i = 0 ; i < 100 ; i++) {
        MemorySegment s = allocator.allocateArray(JAVA_INT,
                                                  new int[] { 1, 2, 3, 4, 5 });
        ...
    }
    ...
} // all memory allocated is released here
```

This code creates a confined resource scope and then creates an *unbounded arena allocator* associated with that scope. This allocator allocates a segment of memory and responds to allocation requests by returning slices of that pre-allocated segment. If the current segment does not have sufficient space to accommodate an allocation request then a new segment is allocated. All of the memory associated with the segments created by the allocator (i.e., in the body of the `for` loop) is deallocated atomically when the resource scope associated with the arena allocator is closed. This technique combines the advantages of deterministic deallocation, provided by the `ResourceScope` abstraction, with a more flexible and scalable allocation scheme. It can be very useful when writing code which manages a large number of off-heap segments.

***Unsafe memory segments***

So far, we have seen memory segments, memory addresses, and memory layouts. Dereference operations are only possible on memory segments. Since a memory segment has spatial and temporal bounds, the Java runtime ensures that the memory associated with a given segment is dereferenced safely. However, there are situations where clients might only have a `MemoryAddress` instance, as is often the case when interacting with native code. To dereference a memory address, a client has two options:

- First, the client can use one of the dereference methods defined in the `MemoryAddress` class. These methods are unsafe because a memory

address has no spatial or temporal bounds, and thus the FFM API has no way to ensure that the memory location being dereferenced is valid.

- Alternatively, the client can unsafely turn the address into a segment via the `MemorySegment::ofAddressNative` factory. This factory attaches fresh spatial and temporal bounds to an otherwise raw memory address in order to allow dereference operations. The memory segment returned by this factory is *unsafe:* A raw memory address might be associated with a memory region that is 10 bytes long, but the client might accidentally overestimate the size of the region and create an unsafe memory segment that is 100 bytes long. Later, this might result in attempts to dereference memory outside the bounds of the memory region associated with the unsafe segment, which might cause a JVM crash or, worse, result in silent memory corruption.

Both of these options are unsafe and are therefore considered *restricted operations*, which are disabled by default (see more below).

### *Looking up foreign functions*

The first ingredient of any support for foreign functions is a mechanism to load native libraries. JNI accomplishes this with the `System::loadLibrary` and `System::load` methods, which internally map into calls to `dlopen` or its equivalent. Libraries loaded using these methods are always associated with a class loader, namely the loader of the class which called the method. The association between libraries and class loaders is crucial because it governs the lifecycle of loaded libraries: Only when a class loader is no longer reachable can all of its libraries be unloaded safely.

The FFM API does not provide new methods for loading native libraries. Developers use the `System::loadLibrary` and `System::load` methods to load native libraries to be invoked via the FFM API. The association between libraries and class loaders is preserved, so that libraries will be unloaded in the same predictable manner as with JNI.

The FFM API, unlike JNI, provides the ability to find the address of a given symbol in a loaded library. This capability, represented by a `SymbolLookup` object, is crucial for linking Java code to foreign functions (see below). There are two ways to obtain a `SymbolLookup` object:

- By invoking `SymbolLookup::loaderLookup`, which returns a symbol lookup which locates all the symbols in all the libraries loaded by the current class loader, or
- By obtaining a `CLinker` instance, which implements the `SymbolLookup` interface and can be used to look up platform-specific symbols in the standard C library.

Given a symbol lookup, a client can find a foreign function with the `SymbolLookup::lookup(String)` method. If the named function is present among the symbols seen by the symbol lookup then the method returns a `NativeSymbol` that points to the function's entry point. For example, the following code loads the OpenGL library, causing it to be associated with the current class loader, and finds the address of its `glGetString` function:

```
System.loadLibrary("GL");
SymbolLookup loaderLookup  = SymbolLookup.loaderLookup();
NativeSymbol clangVersion = loaderLookup.lookup("glGetString").get();
```

### *Linking Java code to foreign functions*

The `CLinker` interface is the core of how Java code interoperates with native code. While the `CLinker` focuses on providing interoperation between Java and C libraries, the concepts in the interface are general enough to support other non-Java languages in future. The interface enables both *downcalls* (calls from Java code to native code) and *upcalls* (calls from native code back to Java code).

```
interface CLinker {
    MethodHandle downcallHandle(NativeSymbol func,
                                FunctionDescriptor function);
    NativeSymbol upcallStub(MethodHandle target,
                            FunctionDescriptor function,
                            ResourceScope scope);
}
```

For downcalls, the `downcallHandle` method takes the address of a foreign function — typically, a `NativeSymbol` obtained from a library lookup — and exposes the foreign function as a *downcall method handle*. Later, Java code invokes the downcall method handle by calling its `invoke` (or `invokeExact`) method, and the foreign function runs. Any arguments passed to the method handle's `invoke` method are passed on to the foreign function.

For upcalls, the `upcallStub` method takes a method handle — typically, one which refers to a Java method, rather than a downcall method handle — and converts it to a `NativeSymbol` instance. Later, the native symbol is passed as an argument when Java code invokes a downcall method handle. In effect, the native symbol serves as a function pointer. (For more information on upcalls, see below.)

Suppose we wish to downcall from Java to the `strlen` function defined in the standard C library:

```
size_t strlen(const char *s);
```

A downcall method handle that exposes `strlen` can be obtained as follows (the details of `FunctionDescriptor` will be described shortly):

```
CLinker linker = CLinker.systemCLinker();
MethodHandle strlen = linker.downcallHandle(
    linker.lookup("strlen").get(),
    FunctionDescriptor.of(JAVA_LONG, ADDRESS)
);
```

Invoking the downcall method handle will run `strlen` and make its result available in Java. For the argument to `strlen`, we use a helper method to convert a Java string into an off-heap memory segment (using the implicit allocator) which is then passed by-reference:

```
MemorySegment str = implicitAllocator().allocateUtf8String("Hello");
long len          = strlen.invoke(cString);  // 5
```

Method handles work well for exposing foreign functions because the JVM already optimizes the invocation of method handles all the way down to native code. When a method handle refers to a method in a `class` file, invoking the method handle typically causes the target method to be JIT-compiled; subsequently, the JVM interprets the Java bytecode that calls `MethodHandle::invokeExact` by transferring control to the assembly code generated for the target method. Thus, a traditional method handle in Java targets non-Java code behind the scenes; a downcall method handle is a natural extension that lets developers target non-Java code explicitly. Method handles also enjoy a property called *signature polymorphism* which allows box-free invocation with primitive arguments. In sum, method handles let the `CLinker` expose foreign functions in a natural, efficient, and extensible manner.

***Describing C types in Java***

To create a downcall method handle, the FFM API requires the client to provide a `FunctionDescriptor` that describes the C parameter types and C return type of the target C function. C types are described in the FFM API by `MemoryLayout` objects such as `ValueLayout` for scalar C types and `GroupLayout` for C struct types. Clients usually have `MemoryLayout` objects on hand to dereference data in foreign memory, and can reuse them to obtain a `FunctionDescriptor`.

The FFM API also uses the `FunctionDescriptor` to derive the type of the downcall method handle. Every method handle is strongly typed, which means it is stringent about the number and types of the arguments that can be passed to its `invokeExact` method at run time. For example, a method handle created to take one `MemoryAddress` argument cannot be invoked via `invokeExact(<MemoryAddress>, <MemoryAddress>)`, even though `invokeExact` is a varargs method. The type of the downcall method handle describes the Java signature which clients must use when invoking the downcall method handle. It is, effectively, the Java view of the C function.

As an example, suppose a downcall method handle should expose a C function that takes a C `int` and returns a C `long`. On Linux/x64 and macOS/x64, the C types `long` and `int` are associated with the predefined layouts `JAVA_LONG` and `JAVA_INT` respectively, so the required `FunctionDescriptor` can be obtained with `FunctionDescriptor.of(JAVA_LONG, JAVA_INT)`. The `CLinker` will then arrange for the type of the downcall method handle to be the Java signature `int` to `long`.

Clients must be aware of the current platform if they target C functions which use scalar types such as `long`, `int`, and `size_t`. This is because the association of scalar C types with layout constants varies by platform. On Windows/x64, a C long is associated with the `JAVA_INT` layout, so the required `FunctionDescriptor` would be `FunctionDescriptor.of(JAVA_INT, JAVA_INT)` and the type of the downcall method handle would be the Java signature `int` to `int`.

As another example, suppose a downcall method handle should expose a void C function that takes a pointer. On all platforms, a C pointer type is associated with the predefined layout `ADDRESS`, so the required `FunctionDescriptor` can be obtained with `FunctionDescriptor.ofVoid(ADDRESS)`. The `CLinker` will then arrange for the type of the downcall method handle to be the Java signature `Addressable` to `void`. `Addressable` is a common supertype of entities in the FFM API that can be passed by reference, such as `MemorySegment`, `MemoryAddress`, and `NativeSymbol`.

Clients can use C pointers without being aware of the current platform. Clients do not need to know the size of pointers on the current platform, since the size of the `ADDRESS` layout is inferred from the current platform, nor do clients need to distinguish between C pointer types such as `int*` and `char**`.

Finally, unlike JNI, the `CLinker` supports passing structured data to foreign functions. Suppose a downcall method handle should expose a `void` C function that takes a struct, described by the following layout:

```
MemoryLayout SYSTEMTIME  = MemoryLayout.ofStruct(
  JAVA_SHORT.withName("wYear"),      JAVA_SHORT.withName("wMonth"),
  JAVA_SHORT.withName("wDayOfWeek"), JAVA_SHORT.withName("wDay"),
  JAVA_SHORT.withName("wHour"),      JAVA_SHORT.withName("wMinute"),
  JAVA_SHORT.withName("wSecond"),    JAVA_SHORT.withName("wMilliseconds")
);
```

The required `FunctionDescriptor` can be obtained with `FunctionDescriptor.ofVoid(SYSTEMTIME)`. The `CLinker` will arrange for the type of the downcall method handle to be the Java signature `MemorySegment` to `void`.

The memory layout associated with a C struct type must be a composite layout which defines the sub-layouts for all the fields in the C struct, including any platform-dependent padding a native compiler might insert.

If a C function returns a by-value struct (not shown here) then a fresh memory segment must be allocated off-heap and returned to the Java client. To achieve this, the method handle returned by `downcallHandle` requires an additional `SegmentAllocator` argument which the FFM API uses to allocate a memory segment to hold the struct returned by the C function.

As mentioned earlier, the `CLinker` is focused on providing interoperation between Java and C libraries, but it is language-neutral: It has no specific knowledge of how C types are defined, so clients are responsible for obtaining suitable layout definitions for C types. This choice is deliberate, since layout definitions for C types — whether simple scalars or complex structs — are ultimately platform-dependent, and can therefore be mechanically generated by a tool that has an intimate understanding of the given target platform.

#### *Packaging Java arguments for C functions*

A *calling convention* enables interoperation between different languages by specifying how code in one language invokes a function in another language, passes arguments, and receives results. The `CLinker` API is neutral with respect to calling conventions, but the `CLinker` implementation implements several calling conventions out-of-the-box: Linux/x64, Linux/AArch64, macOS/x64, and Windows/x64. Being written in Java, it is far easier to maintain and extend than JNI, whose calling conventions are hardwired into HotSpot's C++ code.

Consider the `FunctionDescriptor` obtained above for the `SYSTEMTIME` struct/layout. Given the calling convention of the OS and CPU where the JVM is running, the `CLinker` uses the `FunctionDescriptor` to infer how the struct's fields should be passed to the C function when a downcall method handle is invoked with a `MemorySegment` argument. For one calling convention, the `CLinker` could arrange to decompose the incoming memory segment, pass the first four fields using general CPU registers, and pass the remaining fields on the C stack. For a different calling convention, the `CLinker` could arrange to pass the struct indirectly by allocating a region of memory, bulk-copying the contents of the incoming memory segment into that region, and passing a pointer to that memory region to the C function. This lowest-level packaging of arguments happens behind the scenes, without any need for supervision by client code.

#### *Upcalls*

Sometimes it is useful to pass Java code as a function pointer to some foreign function. We can do that by using the `CLinker` support for upcalls. In this section we build, piece by piece, a more sophisticated example which demonstrates the full power of the `CLinker`, with full bidirectional interoperation of both code and data across the Java/native boundary.

Consider the following function defined in the standard C library:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

To call `qsort` from Java, we first need to create a downcall method handle:

```
CLinker linker = CLinker.systemCLinker();
MethodHandle qsort = linker.downcallHandle(
    linker.lookup("qsort").get(),
    FunctionDescriptor.ofVoid(ADDRESS, JAVA_LONG, JAVA_LONG, ADDRESS)
);
```

As before, we use the `JAVA_LONG` layout to map the C `size_t` type, and we use the `ADDRESS` layout for both the first pointer parameter (the array pointer) and the last parameter (the function pointer).

`qsort` sorts the contents of an array using a custom comparator function, `compar`, passed as a function pointer. Therefore, to invoke the downcall method handle we need a function pointer to pass as the last parameter to the method handle's `invokeExact` method. `CLinker::upcallStub` helps us create function pointers by using existing method handles, as follows.

First, we write a `static` method in Java that compares two `long` values, represented indirectly as `MemoryAddress` objects:

```
class Qsort {
    static int qsortCompare(MemoryAddress addr1, MemoryAddress addr2) {
        return addr1.get(JAVA_INT, 0) - addr2.get(JAVA_INT, 0);
    }
}
```

Second, we create a method handle pointing to the Java comparator method:

```
MethodHandle comparHandle
    = MethodHandles.lookup()
                   .findStatic(Qsort.class, "qsortCompare",
                               MethodType.methodType(int.class,
                                                     MemoryAddress.class,
                                                     MemoryAddress.class));
```

Third, now that we have a method handle for our Java comparator we can create a function pointer using `CLinker::upcallStub`. Just as for downcalls, we describe the signature of the function pointer using a `FunctionDescriptor`:

```
NativeSymbol comparFunc =
  linker.upcallStub(comparHandle,
                    /* A Java description of a C function
                       implemented by a Java method! */
                    FunctionDescriptor.of(JAVA_INT, ADDRESS, ADDRESS),
                    newImplicitScope());
);
```

We finally have a memory address, `comparFunc`, which points to a stub that can be used to invoke our Java comparator function, and so we now have all we need to invoke the `qsort` downcall handle:

```
MemorySegment array = implicitAllocator().allocateArray(
                                          ValueLayout.JAVA_INT,
                                          new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 });
qsort.invoke(array, 10L, 4L, comparFunc);
int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

This code creates an off-heap array, copies the contents of a Java array into it, and then passes the array to the `qsort` handle along with the comparator function we obtained from the `CLinker`. After the invocation, the contents of the off-heap array will be sorted according to our comparator function, written in Java. We then extract a new Java array from the segment, which contains the sorted elements.

### *Safety*

Fundamentally, any interaction between Java code and native code can compromise the integrity of the Java Platform. Linking to a C function in a precompiled library is inherently unreliable because the Java runtime cannot guarantee that the function's signature matches the expectations of the Java code, or even that a symbol in a C library is really a function. Moreover, even if a suitable function is linked, actually calling the function can lead to low-level failures, such as segmentation faults, that end up crashing the VM. Such failures cannot be prevented by the Java runtime or caught by Java code.

Native code that uses JNI functions is especially dangerous. Such code can access JDK internals without command-line flags (e.g., `--add-opens`), by using functions such as `getStaticField` and `callVirtualMethod`. It can also change the values of `final` fields long after they are initialized. Allowing native code to bypass the checks applied to Java code undermines every boundary and assumption in the JDK. In other words, JNI is inherently unsafe.

JNI cannot be disabled, so there is no way to ensure that Java code will not call native code which uses dangerous JNI functions. This is a risk to platform integrity that is almost invisible to application developers and end users because 99% of the use of these functions is typically from third, fourth, and fifth-party libraries sandwiched between the application and the JDK.

Most of the FFM API is safe by design. Many scenarios that required the use of JNI and native code in the past can be accomplished by calling methods in the FFM API, which cannot compromise the Java Platform. For example, a primary use case for JNI, flexible memory allocation, is supported with a simple method, `MemorySegment::allocateNative`, that involves no native code and always returns memory managed by the Java runtime. Generally speaking, Java code that uses the FFM API cannot crash the JVM.

Part of the FFM API, however, is inherently unsafe. When interacting with the `CLinker`, Java code can request a downcall method handle by specifying parameter types that are incompatible with those of the underlying C function. Invoking the downcall method handle in Java will result in the same kind of outcome — a VM crash, or undefined behavior — that can occur when invoking a `native` method in JNI. The FFM API can also produce unsafe segments, that is, memory segments whose spatial and temporal bounds are user-provided and cannot be verified by the Java runtime (see `MemorySegment::ofAddressNative`).

The unsafe methods in the FFM API do not pose the same risks as JNI functions; they cannot, e.g., change the values of `final` fields in Java objects. On the other hand, the unsafe methods in the FFM API are easy to call from Java code. For this reason, the use of unsafe methods in the FFM API is *restricted:* Access to unsafe methods is disabled by default, so that invoking such methods throws an `IllegalAccessException`. To enable access to unsafe methods for code in some module M, specify `java --enable-native-access=M` on the command line. (Specify multiple modules in a comma-separated list; specify `ALL-UNNAMED` to enable access for all code on the class path.) Most methods of the FFM API are safe, and Java code can use those methods regardless of whether `--enable-native-access` is given.

We do not propose here to restrict any aspect of JNI. It will still be possible to call `native` methods in Java, and for native code to call unsafe JNI functions. However, it is likely that we will restrict JNI in some way in a future release. For example, unsafe JNI functions such as `newDirectByteBuffer` may be disabled by default, just like unsafe methods in the FFM API. More broadly, the JNI mechanism is so irredeemably dangerous that we hope libraries will prefer the pure-Java FFM API for both safe and unsafe operations so that, in time, we can disable all of JNI by default. This aligns with the broader Java roadmap of making the platform safe out-of-the-box, requiring end users to opt into unsafe activities such as breaking strong encapsulation or linking to unknown code.

We do not propose here to change `sun.misc.Unsafe` in any way. The FFM API's support for off-heap memory is an excellent alternative to the wrappers around

`malloc` and `free` in `sun.misc.Unsafe`, namely `allocateMemory`, `setMemory`, `copyMemory`, and `freeMemory`. We hope that libraries and applications that require off-heap storage adopt the FFM API so that, in time, we can deprecate and then eventually remove these `sun.misc.Unsafe` methods.

### Alternatives

Keep using `java.nio.ByteBuffer`, `sun.misc.Unsafe`, JNI, and other third-party frameworks.

### Risks and Assumptions

Creating an API to access foreign memory in a way that is both safe and efficient is a daunting task. Since the spatial and temporal checks described in the previous sections need to be performed upon every access, it is crucial that JIT compilers be able to optimize away these checks by, e.g., hoisting them outside of hot loops. The JIT implementations will likely require some work to ensure that uses of the API are as efficient and optimizable as uses of existing APIs such as `ByteBuffer` and `Unsafe`. The JIT implementations will also require work to ensure that uses of the native method handles retrieved from the API are at least as efficient and optimizable as uses of existing JNI native methods.

### Dependencies

- The Foreign Function & Memory API can be used to access non-volatile memory, already possible via JEP 352 (Non-Volatile Mapped Byte Buffers), in a more general and efficient way.
- The work described here will likely enable subsequent work to provide a tool, `jextract`, which, starting from the header files for a given native library, mechanically generates the native method handles required to interoperate with that library. This will further reduce the overhead of using native libraries from Java.