



- Installing
- Contributing
- Sponsoring
- Developers' Guide
- Vulnerabilities
- JDK GA/EA Builds
- Mailing lists
- Wiki · IRC
- Bylaws · Census
- Legal
- Workshop
- JEP Process
- Source code
- Mercurial
- GitHub
- Tools
- Git
- jtreg harness
- Groups
- (overview)
- Adoption
- Build
- Client Libraries
- Compatibility & Specification Review
- Compiler
- Conformance
- Core Libraries
- Governing Board
- HotSpot
- IDE Tooling & Support
- Internationalization
- JMX
- Members
- Networking
- Porters
- Quality
- Security
- Serviceability
- Vulnerability
- Web
- Projects
- (overview, archive)
- Amber
- Audio Engine
- CRaC
- Caciocavallo
- Closures
- Code Tools
- Coin
- Common VM Interface
- Compiler Grammar
- Detroit
- Developers' Guide
- Device I/O
- Duke
- Font Scaler
- Galahad
- Graal
- Graphics Rasterizer
- IcedTea
- JDK 7
- JDK 8
- JDK 8 Updates
- JDK 9
- JDK (... , 21, 22)
- JDK Updates
- JavaDoc.Next
- Jigsaw
- Kona
- Kulla
- Lambda
- Lanai
- Leyden
- Lilliput
- Locale Enhancement
- Loom
- Memory Model Update
- Metropolis
- Mission Control
- Modules
- Multi-Language VM
- Nashorn
- New I/O
- OpenJFX
- Panama
- Penrose
- Port: AArch32
- Port: AArch64
- Port: BSD
- Port: Haiku
- Port: Mac OS X
- Port: MIPS
- Port: Mobile
- Port: PowerPC/AIX
- Port: RISC-V
- Port: s390x
- Portola
- SCTP
- Shenandoah
- Skara
- Sumatra
- Tiered Attribution
- Tsan
- Type Annotations
- Valhalla
- Verona
- VisualVM
- Wakefield
- Zero
- ZGC



JEP 442: Foreign Function & Memory API (Third Preview)

<i>Owner</i>	Maurizio Cimadamore
<i>Type</i>	Feature
<i>Scope</i>	SE
<i>Status</i>	Closed / Delivered
<i>Release</i>	21
<i>Component</i>	core-libs
<i>Discussion</i>	panama dash dev at openjdk dot org
<i>Relates to</i>	JEP 434: Foreign Function & Memory API (Second Preview)
<i>Reviewed by</i>	Alex Buckley, Jorn Vernee
<i>Endorsed by</i>	Mark Reinhold
<i>Created</i>	2023/02/01 14:58
<i>Updated</i>	2023/08/29 22:29
<i>Issue</i>	8301625

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. This is a [preview API](#).

History

The Foreign Function & Memory (FFM) API first previewed in JDK 19 via [JEP 424](#), then previewed again in JDK 20 via [JEP 434](#). This JEP proposes a third preview to incorporate refinements based on feedback. In this version we have:

- Centralized the management of the lifetimes of native segments in the [Arena](#) interface;
- Enhanced layout paths with a new element to dereference address layouts;
- Provided a linker option to optimize calls to functions that are short-lived and will not upcall to Java (e.g., `clock_gettime`);
- Provided a fallback native linker implementation, based on [libffi](#), to facilitate porting; and
- Removed the `Valist` class.

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* — Allow programs to perform unsafe operations on foreign memory, but warn users about such operations by default.

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

Objects created with the new keyword are stored in the JVM's *heap*, where they are subject to garbage collection when no longer needed. However, the cost and unpredictability of garbage collection is unacceptable for performance-critical libraries such as [Tensorflow](#), [Ignite](#), [Lucene](#), and [Netty](#). They need to store data outside the heap, in *off-heap memory* which they allocate and deallocate themselves. Access to off-heap memory also allows data to be serialized and deserialized by mapping files directly into memory via, e.g., [mmap](#).

The Java Platform has historically provided two APIs for accessing off-heap memory:

- The [ByteBuffer API](#) provides *direct* byte buffers, which are Java objects backed by fixed-size regions of off-heap memory. However, the maximum size of a region is limited to two gigabytes and the methods for reading and writing memory are rudimentary and error-prone, providing little more than indexed access to primitive values. More seriously, the memory which backs a direct byte buffer is deallocated only when the buffer object is garbage collected, which the developer cannot control. The lack of support for timely deallocation makes the ByteBuffer API a bad fit for systems programming in Java.
- The [sun.misc.Unsafe API](#) provides low-level access to on-heap memory that also works for off-heap memory. Using Unsafe is fast (because its memory access operations are intrinsified by the JVM), allows huge off-heap regions (theoretically up to 16 exabytes), and offers fine-grained control over deallocation (because `Unsafe::freeMemory` can be called at any time). However, this programming model is weak because it gives the developer too much control. A library in a long-running server application will allocate and interact with multiple regions of off-heap memory over time; data in one region will point to data in another region, and regions must be deallocated in the correct order or else dangling pointers will cause use-after-free bugs. The lack of support for safe deallocation makes the Unsafe API a bad fit for systems programming in Java.

(The same criticism applies to APIs outside the JDK that offer fine-grained allocation and deallocation by wrapping native code which calls `malloc` and `free`.)

In summary, sophisticated clients deserve an API that can allocate, manipulate, and share off-heap memory with the same fluidity and safety as on-heap memory. Such an API should balance the need for predictable deallocation with the need to prevent untimely deallocation that can lead to JVM crashes or, worse, to silent memory corruption.

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](#). These frameworks are often a marked improvement over JNI but 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

- Control the allocation and deallocation of foreign memory ([MemorySegment](#), [Arena](#), and [SegmentAllocator](#)),
- Manipulate and access structured foreign memory ([MemoryLayout](#) and [VarHandle](#)), and
- Call foreign functions ([Linker](#), [FunctionDescriptor](#), and [SymbolLookup](#)).

The FFM API resides in the `java.lang.foreign` package of the `java.base` 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).

Because the FFM API is a [preview API](#), you must compile and run the code with preview features enabled, i.e., `javac --release 21 --enable-preview ...` and `java --enable-preview`

```
// 1. Find foreign function on the C library path
Linker linker          = Linker.nativeLinker();
SymbolLookup stdlib    = linker.defaultLookup();
MethodHandle radixsort = linker.downcallHandle(stdlib.find("radixsort"), ...);
// 2. Allocate on-heap memory to store four strings
String[] javaStrings = { "mouse", "cat", "dog", "car" };
// 3. Use try-with-resources to manage the lifetime of off-heap memory
try (Arena offHeap = Arena.ofConfined()) {
    // 4. Allocate a region of off-heap memory to store four pointers
    MemorySegment pointers
        = offHeap.allocateArray(ValueLayout.ADDRESS, javaStrings.length);
    // 5. Copy the strings from on-heap to off-heap
    for (int i = 0; i < javaStrings.length; i++) {
        MemorySegment cString = offHeap.allocateUtf8String(javaStrings[i]);
        pointers.setAtIndex(ValueLayout.ADDRESS, i, cString);
    }
    // 6. Sort the off-heap data by calling the foreign function
    radixsort.invoke(pointers, javaStrings.length, MemorySegment.NULL, '\0');
    // 7. Copy the (reordered) strings from off-heap to on-heap
    for (int i = 0; i < javaStrings.length; i++) {
        MemorySegment cString = pointers.getAtIndex(ValueLayout.ADDRESS, i);
        javaStrings[i] = cString.getUtf8String(0);
    }
} // 8. All off-heap memory is deallocated here
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 access 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 multiple threads to copy data between on-heap and off-heap memory in parallel.

Memory segments and arenas

A [memory segment](#) is an abstraction backed by a contiguous region of memory, located either off-heap or on-heap. A memory segment can be

- A *native* segment, allocated from scratch in off-heap memory (as if via `malloc`),
- A *mapped* segment, wrapped around a region of mapped off-heap memory (as if via `mmap`), or
- An *array* or *buffer* segment, wrapped around a region of on-heap memory associated with an existing Java array or byte buffer, respectively.

All memory segments provide spatial and temporal bounds which ensure that memory access operations are safe. In a nutshell, the bounds guarantee no use of unallocated memory and no use after free.

The *spatial bounds* of a segment determine the range of memory addresses associated with the segment. For example, the code below allocates a native segment of 100 bytes, so the associated range of addresses is from some *base address* `b` to `b + 99` inclusive.

```
MemorySegment data = Arena.global().allocate(100);
```

The *temporal bounds* of a segment determine its lifetime, that is, the period until the region of memory which backs the segment is deallocated. The FFM API guarantees that a memory segment cannot be accessed after its backing region of memory is deallocated.

The temporal bounds of a segment are determined by the [arena](#) used to allocate the segment. Multiple segments allocated in the same arena have the same temporal bounds, and can safely contain mutual references: Segment A can hold a pointer to an address in segment B, and segment B can hold a pointer to an address in segment A, and both segments will be deallocated at the same time so that neither segment has a dangling pointer.

The simplest arena is the *global* arena, which provides an unbounded lifetime: It is always alive. A segment allocated in the global arena, as in the code above, is always accessible and the region of memory backing the segment is never deallocated.

Most programs, though, require off-heap memory to be deallocated while the program is running, and thus need memory segments with bounded lifetimes.

An *automatic* arena provides a bounded lifetime: A segment allocated by an automatic arena can be accessed until the JVM's garbage collector detects that the memory segment is unreachable, at which point the region of memory backing the segment is deallocated. For example, this method allocates a segment in an automatic arena:

```
void processData() {
    MemorySegment data = Arena.ofAuto().allocateNative(100);
    ... use the 'data' variable ...
    ... use the 'data' variable some more ...
} // the region of memory backing the 'data' segment
// is deallocated here (or later)
```

As long as the data variable does not leak out of the method, the segment will eventually be detected as unreachable and its backing region will be deallocated.

An automatic arena's bounded but non-deterministic lifetime is not always sufficient. For example, an API that maps a memory segment from a file should allow the client to deterministically deallocate the region of memory backing the segment since waiting for the garbage collector to do so could adversely affect performance.

A *confined* arena provides a bounded and deterministic lifetime: It is alive from the time the client opens the arena until the time the client *closes* the arena. A memory segment allocated in a confined arena can be accessed only before the arena is closed, at which point the region of memory backing the segment is deallocated. Attempts to access a memory segment after its arena is closed will fail with an exception. For example, this code opens an arena and uses the arena to allocate two segments:

```
MemorySegment input = null, output = null;
try (Arena processing = Arena.ofConfined()) {
    input = processing.allocate(100);
    ... set up data in 'input' ...
    output = processing.allocate(100);
    ... process data from 'input' to 'output' ...
    ... calculate the ultimate result from 'output' and store it elsewhere ...
} // the regions of memory backing the segments are deallocated here
...
input.get(ValueLayout.JAVA_BYTE, 0); // throws IllegalStateException
// (also for 'output')
```

Exiting the try-with-resources block closes the arena, at which point all segments allocated by the arena are invalidated atomically and the regions of memory backing the segments are deallocated.

A confined arena's deterministic lifetime comes at a price: Only one thread can access the memory segments allocated in a confined arena. If multiple threads need access to a segment then a *shared arena* can be used. The memory segments allocated in a shared arena can be accessed by multiple threads, and any thread — whether it accesses the region or not — can close the arena to deallocate the segments. Closing the arena atomically invalidates the segments, though the deallocation of the regions of memory backing the segments might not occur immediately since an expensive synchronization operation is needed to detect and cancel pending concurrent access operations on the segments.

In summary, an arena controls which threads can access a memory segment, and when, in order to provide both strong temporal safety and a predictable performance model. The FFM API offers a choice of arenas so that a client can trade off breadth of access against timeliness of deallocation.

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 the memory segment, 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, is aligned on four-byte boundaries, 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:

```
MemorySegment segment
    = Arena.ofAuto().allocate(100, // size
                              ValueLayout.JAVA_INT.byteAlignment); // alignment

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 (we assume that `sizeof(int) == 4`):

```
MemorySegment segment
    = Arena.ofAuto().allocate(2 * ValueLayout.JAVA_INT.byteSize() * 10, // size
                              ValueLayout.JAVA_INT.byteAlignment());      // alignment
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 via:

```
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 two *memory-access var handles*, special [var handles](#) which accept 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 = Arena.ofAuto().allocate(ptsLayout);
for (int i = 0; i < ptsLayout.elementCount(); 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 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.

Segment allocators

Memory allocation is often a bottleneck when clients use off-heap memory. The FFM API therefore includes a [SegmentAllocator](#) abstraction to define operations to allocate and initialize memory segments. As a convenience, the [Arena](#) class implements the `SegmentAllocator` interface so that arenas can be used to allocate native segments. In other words, `Arena` is a "one stop shop" for flexible allocation and timely deallocation of off-heap memory:

```
try (Arena offHeap = Arena.ofConfined()) {
    MemorySegment nativeArray = offHeap.allocateArray(ValueLayout.JAVA_INT,
                                                       0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

    MemorySegment nativeString = offHeap.allocateUtf8String("Hello!");
    MemorySegment upcallStub    = linker.upcallStub(handle, desc, offHeap);
    ...
} // memory released here
```

Segment allocators can also be obtained via factories in the [SegmentAllocator](#) interface. For example, one factory creates a *slicing allocator* that responds to allocation requests by returning memory segments which are part of a previously allocated segment; thus, many requests can be satisfied without physically allocating more memory. The following code obtains a slicing allocator over an existing segment and then uses it to allocate a segment initialized from a Java array:

```
MemorySegment segment = ...
SegmentAllocator allocator = SegmentAllocator.slicingAllocator(segment);
```

```
for (int i = 0 ; i < 10 ; i++) {
    MemorySegment s = allocator.allocateArray(JAVA_INT,
                                              1, 2, 3, 4, 5);
    ...
}
```

Segment allocators can be used as building blocks to create arenas that support custom allocation strategies. For example, if a large number of native segments will share the same bounded lifetime then a custom arena could use a slicing allocator to allocate the segments efficiently. This lets clients enjoy both scalable allocation (thanks to slicing) and deterministic deallocation (thanks to the arena).

As an example, the following code defines a *slicing arena* that behaves like a confined arena but internally uses a slicing allocator to respond to allocation requests. When the slicing arena is closed, the underlying confined arena is closed, invalidating all segments allocated in the slicing arena. (Some details are elided.)

```
class SlicingArena implements Arena {
    final Arena arena = Arena.ofConfined();
    final SegmentAllocator slicingAllocator;

    SlicingArena(long size) {
        slicingAllocator = SegmentAllocator.slicingAllocator(arena.allocate(size));
    }

    public void allocate(long byteSize, long byteAlignment) {
        return slicingAllocator.allocate(byteSize, byteAlignment);
    }

    public void close() {
        return arena.close();
    }
}
```

The earlier code which used a slicing allocator directly can now be written more succinctly:

```
try (Arena slicingArena = new SlicingArena(1000)) {
    for (int i = 0 ; i < 10 ; i++) {
        MemorySegment s = slicingArena.allocateArray(JAVA_INT,
                                                    1, 2, 3, 4, 5);
        ...
    }
} // all memory allocated is released here
```

Looking up foreign functions

The first ingredient of any support for foreign functions is a mechanism to find the address of a given symbol in a loaded native library. This capability, represented by a [SymbolLookup](#) object, is crucial for linking Java code to foreign functions (see [below](#)). The FFM API supports three different kinds of symbol lookup objects:

- `SymbolLookup::libraryLookup(String, Arena)` creates a *library lookup*, which locates all the symbols in a user-specified native library. Creating the lookup object causes the library to be loaded (e.g., using `dlopen()`) and associated with a `Arena` object. The library is unloaded (e.g., using `dlclose()`) when the provided arena is closed.
- `SymbolLookup::loaderLookup()` creates a *loader lookup*, which locates all the symbols in all the native libraries that have been loaded by classes in the current class loader using the `System::loadLibrary` and `System::load` methods.
- `Linker::defaultLookup()` creates a *default lookup*, which locates all the symbols in libraries that are commonly used on the OS and processor combination associated with the [Linker](#) instance.

Given a symbol lookup, a client can find a foreign function with the `SymbolLookup::find(String)` method. If the named function is present among the symbols seen by the symbol lookup then the method returns a zero-length memory segment (see [below](#)) whose base address points to the function's entry point. For example, the following code uses a loader lookup to load the OpenGL library and find the address of its `glGetString` function:

```
try (Arena arena = Arena.ofConfined()) {
    SymbolLookup opengl = SymbolLookup.libraryLookup("libGL.so", arena);
    MemorySegment glVersion = opengl.find("glGetString").get();
    ...
} // libGL.so unloaded here
```

`SymbolLookup::libraryLookup(String, Arena)` differs from JNI's library loading mechanism (i.e., `System::loadLibrary`) in an important way. Native libraries designed to work with JNI can use JNI functions to perform Java operations, such as object allocation or method access, which might trigger [class loading](#). Therefore such JNI-affiliated libraries must be associated with a class loader when they are loaded by the JVM. Then, to preserve [class loader integrity](#), the same JNI-affiliated library cannot be loaded from classes defined in different class loaders.

In contrast, the FFM API does not offer functions for native code to access the Java environment, and does not assume that native libraries are designed to work with the FFM API. The native libraries loaded via

`SymbolLookup::libraryLookup(String, Arena)` are unaware that they are accessed from code running in a JVM, and make no attempt to perform Java operations. As such, they are not tied to a particular class loader and can be (re)loaded as many times as needed by FFM API clients in different loaders.

Linking Java code to foreign functions

The `Linker` interface is the core of how Java code interoperates with foreign code. While in this document we often refer to interoperation between Java and C libraries, the concepts in this interface are general enough to support other non-Java languages in future. The `Linker` interface enables both *downcalls* (calls from Java code to native code) and *upcalls* (calls from native code back to Java code).

```
interface Linker {
    MethodHandle downcallHandle(MemorySegment address,
                               FunctionDescriptor function);
    MemorySegment upcallStub(MethodHandle target,
                             FunctionDescriptor function,
                             Arena arena);
}
```

For downcalls, the `downcallHandle` method takes the address of a foreign function — typically, a `MemorySegment` 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 `MemorySegment` instance. Later, the memory segment is passed as an argument when Java code invokes a downcall method handle. In effect, the memory segment 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);
```

Clients can link C functions using the *native linker* (see `Linker::nativeLinker`), a `Linker` implementation that conforms to the ABI determined by the OS and CPU on which the JVM is running. A downcall method handle that exposes `strlen` can be obtained as follows (the details of `FunctionDescriptor` will be described shortly):

```
Linker linker = Linker.nativeLinker();
MethodHandle strlen = linker.downcallHandle(
    linker.defaultLookup().find("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 a confined arena, which is then passed by reference:

```
try (Arena arena = Arena.ofConfined()) {
    MemorySegment str = arena.allocateUtf8String("Hello");
    long len          = (long) strlen.invoke(str); // 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 `Linker` 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 `MemorySegment` argument cannot be invoked via `invokeExact(<MemorySegment>, <MemorySegment>)`, 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 native linker 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 native linker will then arrange for the type of the downcall method handle to be the Java signature `MemorySegment to void`. That is, a `MemorySegment` parameter can be passed by reference, or by value, depending on the layout specified in the corresponding function descriptor..

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 native linker supports passing structured data to foreign functions. Suppose a downcall method handle should expose a void C function that takes a struct, described by this 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 Linker 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, while the native linker implementation is focused on providing interoperability between Java and C libraries, the Linker interface 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 interoperability between different languages by specifying how code in one language invokes a function in another language, passes arguments, and receives results. The Linker API is neutral with respect to calling conventions, but the native linker implementation supports several calling conventions out-of-the-box: Linux/x64, Linux/AArch64, Linux/RISC-V, macOS/x64, macOS/AArch64, Windows/x64 and Windows/AArch64. [Additional platforms](#) are supported via the *fallback linker*, which is a native linker implementation based on [Libffi](#). Being written in Java, the Linker API 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 native linker 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 native linker implementation 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 native linker implementation 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 region to the C function. This lowest-level packaging of arguments happens behind the scenes, without any need for supervision by client code.

Zero-length memory segments

Foreign functions often allocate a region of memory and return a pointer to that region. Modeling such a region with a memory segment is challenging because the region's size is not available to the Java runtime. 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 a sequence of `char` values terminated by `'\0'`. The size of the region is not readily apparent to the code calling the foreign function.

The FFM API represents a pointer returned from a foreign function as a *zero-length memory segment*. The address of the segment is the value of the pointer, and the size of the segment is zero. Similarly, when a client reads a pointer from a memory segment then a zero-length memory segment is returned.

A zero-length segment has trivial spatial bounds, so any attempt to access such a segment fails with `IndexOutOfBoundsException`. This is a crucial safety feature: Since these segments are associated with a region of memory whose size is not known, 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.

Clients can turn a zero-length memory segment into a native segment of their chosen size via the method `MemorySegment::reinterpret`. This method attaches fresh spatial and temporal bounds to a zero-length memory segment in order to allow dereference operations. The memory segment returned by this method is unsafe: A zero-length memory segment might be backed by a region of memory that is 10 bytes long, but the client might overestimate the size of the region and use `MemorySegment::reinterpret` to obtain a segment that is 100 bytes long. Later, this might result in attempts to dereference memory outside the bounds of the region, which might cause a JVM crash or — even worse — result in silent memory corruption.

Overriding the spatial and temporal bounds of a zero-length memory segment is unsafe. For this reason, the `MemorySegment::reinterpret` method is *restricted*, and using it in a program causes the Java runtime to issue warnings (see more [below](#)).

Upcalls

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

Consider this function defined in the standard C library:

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

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

```
Linker linker = Linker.nativeLinker();
MethodHandle qsort = linker.downcallHandle(
    linker.defaultLookup().find("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. `Linker::upcallStub` helps us create function pointers by using existing method handles, as follows.

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

```
class Qsort {
    static int qsortCompare(MemorySegment elem1, MemorySegment elem2) {
        return Integer.compare(elem1.get(JAVA_INT, 0), elem2.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,
                                            MemorySegment.class,
                                            MemorySegment.class));
```

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

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

```
ADDRESS.withTargetLayout(JAVA_INT)),
        Arena.ofAuto());
    );
```

We finally have a memory segment, `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:

```
try (Arena arena = Arena.ofConfined()) {
    MemorySegment array
        = arena.allocateArray(ValueLayout.JAVA_INT,
                               0, 9, 3, 4, 6, 5, 1, 8, 2, 7);
    qsort.invoke(array, 10L, ValueLayout.JAVA_INT.byteSize(), 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 native linker. 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 `Linker`, Java code can request a downcall method handle by specifying parameter types that are incompatible with those of the underlying foreign 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::reinterpret`).

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*: Their use is permitted but, by default, every such use causes a warning to be issued at run time. To allow code in a module `M` to use unsafe methods without warnings, specify the `--enable-native-access=M` option on the `java` command line. (Specify multiple modules with a comma-separated list; specify `ALL-UNNAMED` to enable warning-free use for all code on the class path.) When this option is present, any use of unsafe methods from outside the list of specified modules will cause an `IllegalCallerException` to be thrown, rather than a warning to be issued. In a future release, it is likely that this option will be required in order to use unsafe methods.

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.