```
OpenIDK
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds
Mailing lists
Wiki · IRC
Bylaws · Census
Legal
Workshop
JEP Process
Source code
Mercurial
GitHub
Tools
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
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM
 Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
IDK 9
JDK (..., 21, 22, 23)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model
 Update
Metropolis
Mission Control
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: RISC-V Port: s390x

Shenandoah

Tiered Attribution

**Type Annotations** 

ORACLE

Portola

SCTP

Skara

Tsan

Sumatra

Valhalla

Verona VisualVM

Zero ZGC

Wakefield

Port: PowerPC/AIX

```
Type Feature
      Scope JDK
     Status Closed / Delivered
    Release 16
 Component core-libs
 Discussion panama dash dev at openjdk dot java dot net
      Effort L
   Duration L
  Relates to JEP 393: Foreign-Memory Access API (Third Incubator)
             JEP 412: Foreign Function & Memory API (Incubator)
Reviewed by Brian Goetz, Jorn Vernee, Paul Sandoz
    Created 2020/07/20 11:19
    Updated 2022/03/02 17:08
       Issue 8249755
```

JEP 389: Foreign Linker API (Incubator)

Owner Maurizio Cimadamore

# Summary

otherwise error-prone process of binding to a native library. History

and then subsequently refreshed by JEP 383 and JEP 393, which were targeted to Java 15 and 16, respectively. Together, the Foreign-Memory Access API and the Foreign Linker API constitute key deliverables of Project Panama. Goals

Ease of use: Replace JNI with a superior pure-Java development model. C support: The initial scope of this effort aims at providing high quality, fully optimized interoperability with C libraries, on x64 and AArch64

- platforms. Generality: The Foreign Linker API and implementation should be flexible enough to, over time, accommodate support for other platforms (e.g., 32bit x86) and foreign functions written in languages other than C (e.g. C++,
- Performance: The Foreign Linker API should provide performance that is comparable to, or better than, JNI. **Non-Goals**
- It is not a goal to: Drop, re-implement, or improve JNI, Provide a tool to mechanically generate Java code from native-code header

# files, or

Fortran).

- Change or improve the way in which Java applications interacting with native libraries are packaged and deployed (e.g., multi-platform JAR files).
- **Motivation** Java has supported native method calls via the Java Native Interface (JNI) since
- Java 1.1, but this path has always been hard and brittle. Wrapping a native

function with JNI requires developing multiple artifacts: a Java API, a C header file,

### and a C implementation. Even with tooling help, Java developers must work across multiple toolchains to keep multiple platform-dependent artifacts in sync. This is

Symbol lookups

hard enough with stable APIs, but when trying to track APIs in progress, it is a significant maintenance burden to update all of these artifacts each time the API evolves. Finally, JNI is largely about code, but code always exchanges data, and JNI offers little help in accessing native data. For this reason, developers often resort to workarounds (such as direct buffers or sun.misc.Unsafe) which make the application code harder to maintain or even less safe. Over the years, numerous frameworks have emerged to fill the gaps left by JNI, including JNA, JNR and JavaCPP. JNA and JNR generate wrappers dynamically from a user-defined interface declaration; JavaCPP generates wrappers statically driven by annotations on JNI method declarations. While these frameworks are often a marked improvement over the JNI experience, the situation is still less than ideal, especially when compared with languages which offer first-class native

interoperation. For instance, Python's ctypes package can dynamically wrap native

Ultimately, Java developers should be able to (mostly) just use any native library

functions without any glue code. Other languages, such as Rust, provide tools

which mechanically derive native wrappers from C/C++ header files.

that is deemed useful for a particular task — and we have seen how the status quo gets in the way of achieving that. This JEP rectifies this imbalance by introducing an efficient and supported API — the Foreign Linker API — which provides foreignfunction support without the need for any intervening JNI glue code. It does this by exposing foreign functions as method handles which can be declared and invoked in pure Java code. This greatly simplifies the task of writing, building and distributing Java libraries and applications which depend upon foreign libraries. Moreover, the Foreign Linker API, together with the Foreign-Memory Access API, provides a solid and efficient foundation which third-party native interoperation frameworks — both present and future — can reliably build upon. **Description** 

In this section we dive deeper into how native interoperation is achieved using the

provided as an incubator module named jdk.incubator.foreign, in a package of

Foreign Linker API. The various abstractions described in this section will be

the same name, side-by-side with the existing Foreign Memory Access API.

The first ingredient of any foreign-function support is a mechanism to look up

symbols in native libraries. In traditional Java/JNI scenarios, this is done via the System::loadLibrary and System::load methods, which internally map into calls to dlopen. The Foreign Linker API provides a simple library-lookup abstraction via the LibraryLookup class (similar to a method-handle lookup), which provides capabilities to look up named symbols in a given native library. We can obtain a library lookup in three different ways: LibraryLookup::ofDefault — returns the library lookup which can see all the symbols that have been loaded with the VM.

LibraryLookup::ofPath — creates a library lookup associated with the

library found at the given absolute path.

provided by the clang library:

interface CLinker {

CLinker.Win64.C LONG).

library with given name (this might require setting the java.library.path variable appropriately). Once a lookup is obtained, a client can use it to retrieve handles to library symbols,

either global variables or functions, using the lookup(String) method. This

LibraryLookup libclang = LibraryLookup.ofLibrary("clang");

be loaded into more than one class loader. The foreign-function mechanism

The CLinker interface is the foundation of the API's foreign function support.

MemorySegment upcallStub(MethodHandle target,

MethodHandle downcallHandle(LibraryLookup.Symbol func,

MethodType type,

FunctionDescriptor function);

FunctionDescriptor function);

LibraryLookup::ofLibrary — creates a library lookup associated with the

address and a name. For instance, the following code looks up the clang getClangVersion function

method returns a fresh LibraryLookup.Symbol, which is just a proxy for a memory

LibraryLookup.Symbol clangVersion = libclang.lookup("clang getClangVersion"); One crucial distinction between the library loading mechanism of the Foreign Linker API and that of INI is that loaded INI libraries are associated with a class

loader. Furthermore, to preserve class loader integrity, the same JNI library cannot

described here is more primitive: The Foreign Linker API allows clients to target native libraries directly, without any intervening JNI code. Crucially, Java objects are never passed to and from native code by the Foreign Linker API. Because of this, libraries loaded via LibraryLookup are not tied to any class loader and can be (re)loaded as many times as needed. The C linker

MethodHandle objects. Second, for *upcalls* (e.g. calls from native back to Java code), the upcallStub method can be used to convert an existing MethodHandle (which might point to some Java method) into a MemorySegment, which can then be passed to a native function as a function pointer. Note that, while the CLinker abstraction is mostly focused on providing interoperation support for the C language, the concepts in this abstraction are general enough to be applicable, in the future, to other foreign languages.

Both downcallHandle and upcallStub take a FunctionDescriptor instance,

This abstraction plays a dual role. First, for downcalls (e.g. calls from Java to native code), the downcallHandle method can be used to model native functions as plain

which is an aggregate of memory layouts which is used to describe the signature of a foreign function in full. The CLinker interface defines many layout constants, one for each main C primitive type. These layouts can be combined using a FunctionDescriptor to describe the signature of a C function. For instance, we can model a C function taking a char\* and returning a long with the following descriptor: FunctionDescriptor func = FunctionDescriptor.of(CLinker.C\_LONG, CLinker.C\_POINTER); The layouts in this example map to the layout appropriate to the underlying

Layouts defined in the CLinker class are convenient, since they model the C types we want to work with. They also contain, via layout attributes, hidden pieces of information which the foreign linker uses in order to compute the calling sequence associated with a given function descriptor. For instance, the two C types int and

float might share a similar memory layout (they are both 32-bit values), but are typically passed using different processor registers. The layout attributes attached

platform, so these layouts are platform dependent: C\_LONG will, e.g., be a 32 bit value layout on Windows, but a 64-bit value on Linux. To target a specific platform,

specific sets of platform-dependent layout constants are available (e.g.,

to the C-specific layouts in the CLinker class ensure that arguments and return values are handled in the correct way. Both downcallHandle and upcallStub also accept (either directly or indirectly) a MethodType instance. The method type describes the Java signatures that clients will use when interacting with the generated downcall handles or upcall stubs. The argument and return types in the MethodType instance are validated against the corresponding layouts. For instance, the linker runtime checks that the size of the

Java carrier associated to a given argument/return value is equal to that of the corresponding layout. The mapping of primitive layouts to Java carriers can vary from one platform to another (e.g., C\_LONG maps to long on Linux/x64, but to int

on Windows), but pointer layouts (C POINTER) are always associated with a

are always associated with a MemorySegment carrier.

MemoryAddress carrier and structs (whose layouts are defined by a GroupLayout)

Downcalls Assume we want to call the following function defined in the standard C library: size\_t strlen(const char \*s); To do that, we have to: Lookup the strlen symbol, Describe the signature of the C function using the layouts in the CLinker Select the Java signature to overlay on the native function (this will be the signature that clients of the native method handle will interact with), and

Create a downcall native method handle with the above information, using

MethodHandle strlen = CLinker.getInstance().downcallHandle( LibraryLookup.ofDefault().lookup("strlen"),

The strlen function is part of the standard C library, which is loaded with the VM,

Once we have obtained the downcall native method handle, we can just use it as

try (MemorySegment str = CLinker.toCString("Hello")) { long len = strlen.invokeExact(str.address()); // 5

FunctionDescriptor.of(C\_LONG, C\_POINTER)

so we can just use the default lookup to look it up. The rest is pretty

MethodType.methodType(long.class, MemoryAddress.class),

straightforward. The only tricky detail is how we model size t — typically this type has the size of a pointer, so we can use C LONG on Linux, but we would have to use C LONG LONG on Windows. On the Java side, we model the size t using a long and the pointer is modeled using a MemoryAddress parameter.

CLinker::downcallHandle.

Here's an example of how to do that:

);

any other method handle:

Here we use one of the helper methods in CLinker to convert a Java string into an off-heap memory segment which contains a NULL terminated C string. We then pass that segment to the method handle and store the result in a Java long. Observe that all this has been possible without any intervening native code — all of the interoperation code can be expressed in (low level) Java.

Sometimes it is useful to pass Java code as a function pointer to some native function. We can achieve that by using the foreign-linker support for upcalls. To

void qsort(void \*base, size t nmemb, size t size,

demonstrate this, consider the following function defined in the standard C library:

This is a function that can be used to sort the contents of an array, using a custom comparator function, compar, which is passed as a function pointer. To be able to call the gsort function from Java we have first to create a downcall native method

int (\*compar)(const void \*, const void \*));

# MethodHandle qsort = CLinker.getInstance().downcallHandle(

handle for it:

}

the last parameter (the function pointer).

Upcalls

MethodType.methodType(void.class, MemoryAddress.class, long.class, long.class, MemoryAddress.class), FunctionDescriptor.ofVoid(C\_POINTER, C\_LONG, C\_LONG, C\_POINTER) );

As before, we use C\_LONG and long.class to map the C size\_t type, and we use MemoryAddess.class both for the first pointer parameter (the array pointer) and

This time, in order to invoke the qsort downcall handle, we need a function pointer to pass as the last parameter. This is where the upcall support of the foreign-linker

LibraryLookup.ofDefault().lookup("qsort"),

abstraction comes in handy, since it allows us to create a function pointer from an existing method handle. First, we write a static method that can compare two int elements, passed as pointers: class Qsort { static int qsortCompare(MemoryAddress addr1, MemoryAddress addr2) { return MemoryAccess.getIntAtOffset(MemorySegment.ofNativeRestricted(), addr1.toRawLongValue()) -

Then we create a method handle pointing to the above comparator function: MethodHandle comparHandle = MethodHandles.lookup() .findStatic(Qsort.class, "qsortCompare", MethodType.methodType(int.class, MemoryAddress.class, MemoryAddress.class)); Now that we have a method handle for our Java comparator we can create a

function pointer. Just as for downcalls, we describe the signature of the foreign-

We finally have a memory segment, comparFunc, whose base address points to a stub that can be used to invoke our Java comparator function, and so we now have

try (MemorySegment array = MemorySegment.allocateNative(4 \* 10)) {

all we need to invoke the gsort downcall handle:

MemoryAccess.getIntAtOffset(MemorySegment.ofNativeRestricted(),

addr2.toRawLongValue());

function pointer using the layouts in the CLinker class: MemorySegment comparFunc = CLinker.getInstance().upcallStub(comparHandle, FunctionDescriptor.of(C\_INT, C POINTER, C POINTER)); );

array.copyFrom(MemorySegment.ofArray(new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 }));

} 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 foreign linker. As a side effect, 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

qsort.invokeExact(array.address(), 10L, 4L, comparFunc.address());

int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

This advanced example shows the full power of the foreign-linker abstraction, with full bidirectional interoperation of both code and data across the Java/native boundary. **Alternatives** Keep using JNI, or other third-party native interoperation frameworks.

### Risks and Assumptions The JIT implementations will require some work to ensure that uses of the native method handles retrieved form the API are at least as efficient and optimizable as uses of existing JNI native methods.

sorted elements.

 Allowing foreign function calls always implies relaxing some of the safety requirements typically associated with the Java Platform. (This is already

- the case when invoking JNI native methods, although developers might not be aware of that). For instance, there is no way for the Foreign Linker API to validate, e.g., that the number of arguments in a function descriptor matches that of the symbol being linked. To help troubleshoot some of the
- most common failure causes, additional debugging capabilities may be provided, similar to the existing -Xcheck: jni option. Since the Foreign Linker API is intrinsically unsafe, obtaining a foreign linker instance is a privileged, restricted operation which requires the -Dforeign.restricted=permit flag.
- **Dependencies** The API described in this JEP represents a significant milestone towards the

383.

 The work described in this JEP 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

native interoperation support that is a goal of Project Panama, and builds heavily upon the Foreign-Memory Access API described in JEP 370 and JEP

interoperate with that library. This will further reduce the overhead of using native libraries from Java.

> © 2024 Oracle Corporation and/or its affiliates Terms of Use · License: GPLv2 · Privacy · Trademarks

Introduce an API that offers statically-typed, pure-Java access to native code. This API, together with the Foreign-Memory API (JEP 393), will considerably simplify the The Foreign-Memory Access API, which provides the foundations for this JEP, was first proposed by JEP 370 and targeted to Java 14 in late 2019 as an incubating API,