ORACLE

Foreign Function & Memory API

A (quick) peek under the hood

Maurizio Cimadamore
Compiler Architect

Beyond “Pure Java”

Native interop frown upon – “Pure Java” used to be the goal

- “Use native methods judiciously” (J. Bloch, Effective Java 3rd edition)

While there are many great Java libraries, there are increasingly many important native-only libraries

- Off-CPU computing (Cuda, OpenCL)
- Machine Learning (Blis, ONNX, Tensorflow)
- Graphics processing (OpenGL, Vulkan, DirectX)
- Others (CRIU, fuse, io_uring, OpenSSL, V8, ucx)

These libraries won’t be, <u>and don’t need to be</u>, rewritten in Java

Java Native Interface

The Java Native Interface (**JNI**) can be used to access to functionalities not available in JDK

JNI allows classes to declare **native** methods

- Native methods do **not** have a body (analogy: *abstract* methods)
- Implementation written in native languages such as C or C++ (or even assembly!)

Problems

- *Native-first* programming model, brittle combination of Java and C
- Expensive to maintain and deploy
- Passing data to/from JNI is cumbersome and inefficient (more on that later)

JNI and data

Native functions often need to exchange (off-heap) data with Java programs

- JNI calling conventions only support primitive types and Java objects

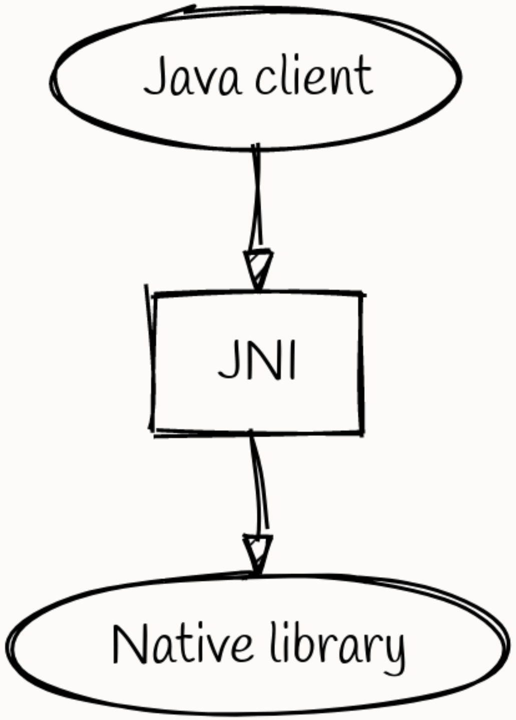
Direct buffers allow developers to allocate and access off-heap memory

- Can be passed to native methods (with some overhead)
- Can be accessed directly from C/C++ code (using JNI functions)

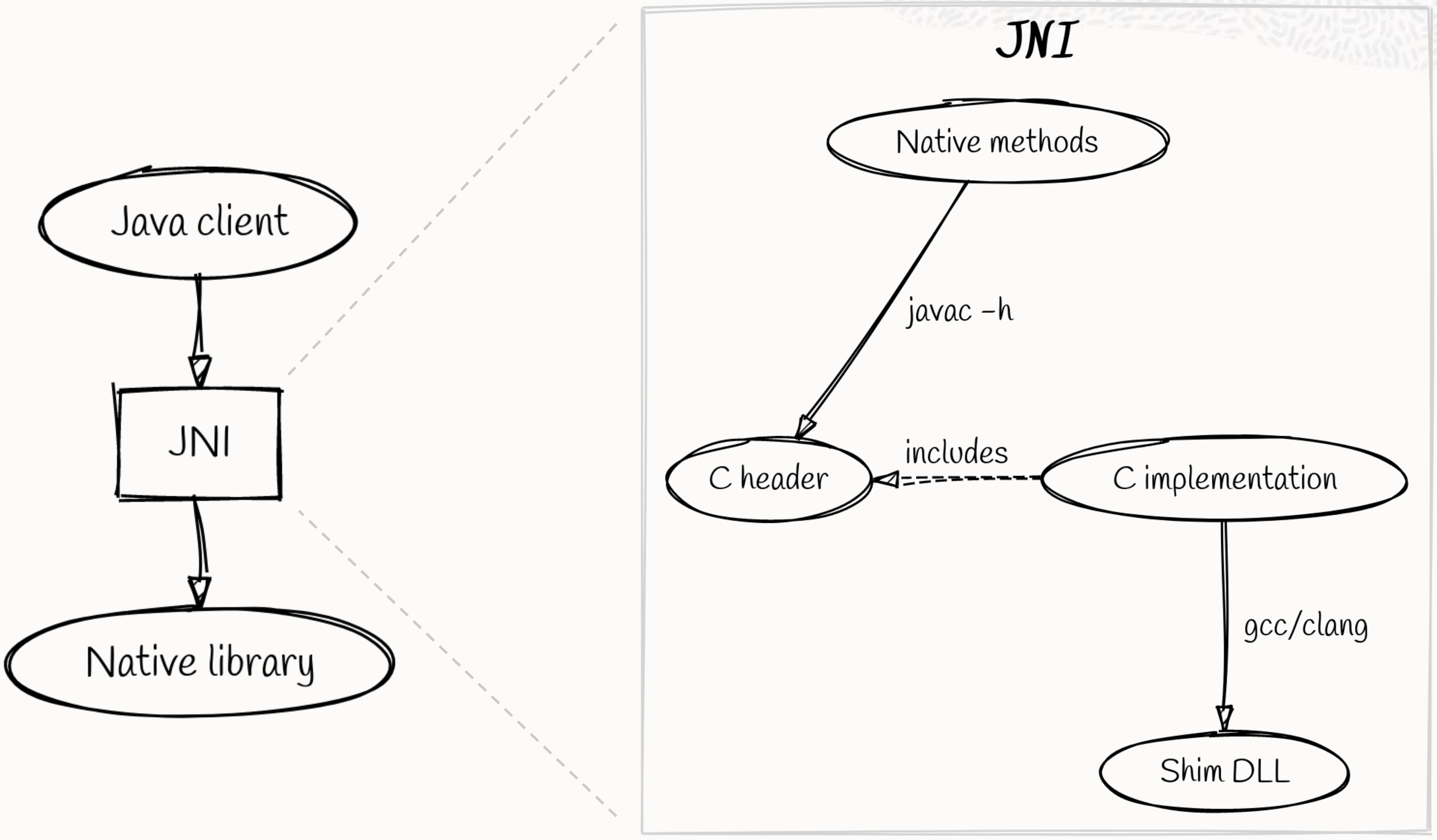
Problems

- No way to free/unmap
- Limited addressing space (2GB)
- Inflexible addressing options (either sequential or offset-based)

JNI workflow



JNI workflow



Enter Panama

We need a new **Java-first** programming model for non-Java resources (both code *and* data)

- Replace JNI with a more direct, pure Java paradigm
- Replace direct buffers with a more safe, efficient and future-proof API
- Simplify building and distributing Java bindings for popular native libraries
- Allow for existing frameworks (JNA, JNR, JavaCPP, …) to be built on top of more solid foundations

Enter Panama

r non-Java resources (both code *and* data)
digm
nt and future-proof API
s for popular native libraries
PP, …) to be built on top of more solid foundations

JEP 454: Foreign Function & Memory API

Owner	Maurizio Cimadamore
Type	Feature
Scope	SE
Status	Completed
Release	22
Component	core-libs / java.lang.foreign
Discussion	panama dash dev at openjdk dot org
Relates to	JEP 442: Foreign Function & Memory API (Third Preview)
Reviewed by	Alex Buckley, Jorn Vernee
Endorsed by	Alan Bateman
Created	2023/06/22 09:36
Updated	2023/12/06 17:32
Issue	8310626

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.

Enter Panama

r non-J

digm

JEP 454: Foreign Function & Memory API

Su

Intr
out
out
ma
pro

JEP 460: Vector API (Seventh Incubator)

Owner	Paul Sandoz
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	22
Component	core-libs
Discussion	panama dash dev at openjdk dot org
Effort	XS
Duration	XS
Relates to	JEP 448: Vector API (Sixth Incubator)
Reviewed by	Vladimir Ivanov
Endorsed by	John Rose
Created	2023/09/08 17:29
Updated	2023/11/08 03:26
Issue	8315945

Summary

Introduce an API to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations.

Babylon

Babylon's primary goal is to extend the reach of Java to foreign programming models such as SQL, differentiable programming, machine learning models, and GPUs. Babylon will achieve this with an enhancement to reflective programming in Java, called code reflection.

This Project is sponsored by the Core Libraries and Compiler Groups.

Summary

Focusing on the GPU example, suppose a Java developer wants to write a GPU kernel in Java and execute it on a GPU. The developer's Java code must, somehow, be analyzed and transformed into an executable GPU kernel. A Java library could do that, but it requires access to the Java code in symbolic form. Such access is, however, currently limited to the use of non-standard APIs or to conventions at different points in the program's life cycle (compile time or run time), and the symbolic forms available (abstract syntax trees or bytecodes) are often ill-suited to analysis and transformation.

Accessing native memory

A **memory segment** provides access to a contiguous region of memory

Two kinds of memory segments

- *Heap* segments → access to memory *inside* the Java heap (e.g. Java array)
- *Native* segments → access to memory *outside* the Java heap (e.g. `malloc/mmap`)

Access to *all* memory segments is governed by the following characteristics

- Size → no out-of-bounds access
- Lifetime → no use-after-free
- Confinement (optional) → no data races

Accessing native memory

```
// struct Point2d {
//   double x;
//   double y;
// } point = { 3.0, 4.0 };

MemorySegment point = Arena.ofAuto().allocate(8 * 2);
point.set(ValueLayout.JAVA_DOUBLE, 0, 3d);
point.set(ValueLayout.JAVA_DOUBLE, 8, 4d);
```


Automatic memory management

Java features **automatic** memory management, using a garbage collector (GC)

Programs create objects (`new`), the GC “frees” them (e.g. recycles them) when no longer needed

- Concept of **reachability**
- One of the corner stones of Java’s success!

Direct buffers rely on GC to perform off-heap memory deallocation, but there’s issues:

- A small *on-heap* Java buffer instance can hold on to a big chunk of off-heap memory
- Materializing reachability graphs is expensive (more so in low-latency collectors)
- GC cannot track usage of off-heap resources from native code

Challenge: provide **deterministic** deallocation in language built on automatic memory management!

Arena-based memory management

An **arena** models the lifecycle of one or more memory segments

- All segments allocated in the arena share the *same* lifetime

Many kinds of arenas, providing different deallocation/access policies

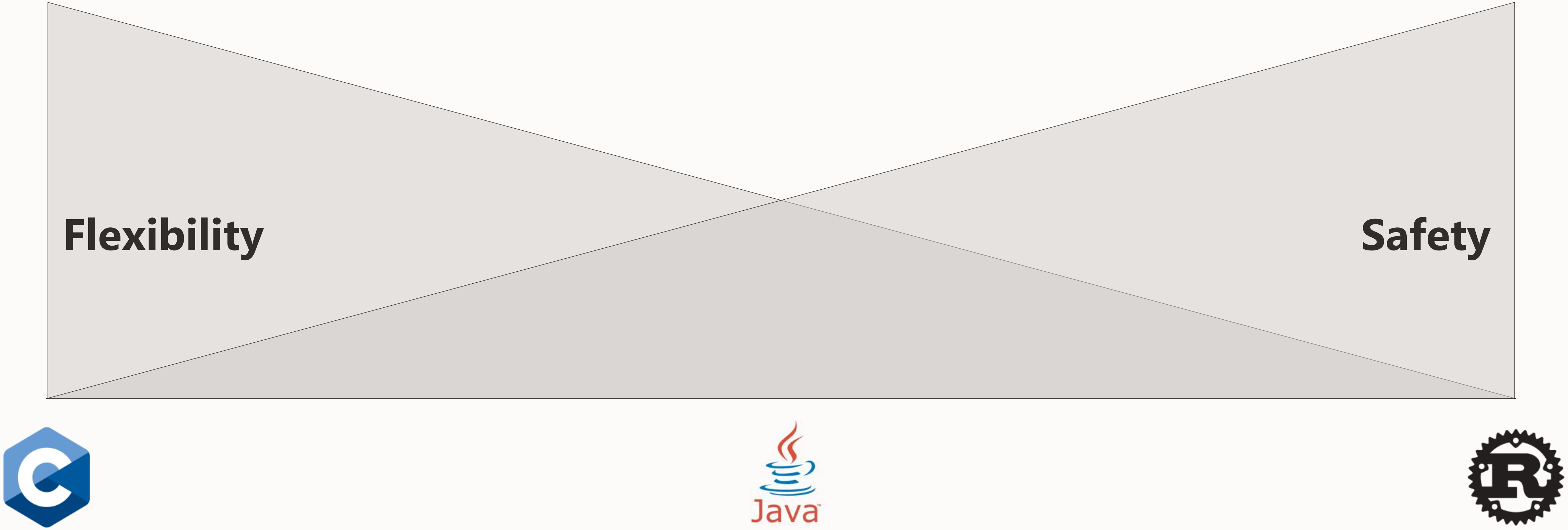
• *Global*	→	unbounded lifetime	multi-thread access
• *Automatic*	→	automatic bounded lifetime	multi-thread access
• *Confined*	→	explicit bounded lifetime	single-thread access
• *Shared*	→	explicit bounded lifetime	multi-thread access

Strong **safety** guarantee: no use-after-free

- When the arena is closed, *all* its segments are invalidated, atomically
- Closing a shared arena triggers a *thread-local handshake* (JEP 312)

Clients can define **custom** arenas to support efficient allocation strategies

Arena-based memory management



Freeing memory with arenas

```
// struct Point2d {
//   double x;
//   double y;
// } point = { 3.0, 4.0 };

try (Arena offHeap = Arena.ofConfined()) {
    MemorySegment point = offHeap.allocate(8 * 2);
    point.set(ValueLayout.JAVA_DOUBLE, 0, 3d);
    point.set(ValueLayout.JAVA_DOUBLE, 8, 4d);
} // free
```

Memory layouts

Often memory access occurs in a **structured** fashion (`point.y`)

- Manual offset computation is tedious and error-prone

Memory layouts describe contents of a memory region *programmatically*

- Layouts can be queried to obtain sizes, alignments and **var handles**

More declarative code, *less* places for bugs to hide!

```
struct Point2d {
    double x;
    double y;
};
```

→

```
MemoryLayout.structLayout(
    ValueLayout.JAVA_DOUBLE.withName("x"),
    ValueLayout.JAVA_DOUBLE.withName("y")
);
```

Structured access with layouts

```
// struct Point2d {
//   double x;
//   double y;
// } point = { 3.0, 4.0 };

MemoryLayout POINT_2D = MemoryLayout.structLayout(
    ValueLayout.JAVA_DOUBLE.withName(“x”),
    ValueLayout.JAVA_DOUBLE.withName(“y”)
);

VarHandle xHandle = POINT_2D.varHandle(PathElement.groupLayout(“x”));
VarHandle yHandle = POINT_2D.varHandle(PathElement.groupLayout(“y”));

try (Arena offHeap = Arena.ofConfined()) {
    MemorySegment point = offHeap.allocate(POINT_2D);
    xHandle.set(point, 0L, 3d);
    yHandle.set(point, 0L, 4d);
} // free
```


Linking native functions

The **native linker** implements the *calling conventions* of the platform in which the JVM runs

Provides two *core* capabilities:

- Link a library symbol into a **downcall method handle**, callable from Java
- Obtain an **upcall stub**, used to invoke a method handle from native code

The native linker builds on what we have seen so far

- Memory layouts used to describe *signatures* of C functions
- Memory segments used to pass pointers/structs/unions to C functions
- Arenas used to model the lifetime of upcall stubs/loaded libraries

Anatomy of a native call

```
struct Point2d {
    double x;
    double y;
};

extern double distance(struct Point2d p);

void main(void) {
    struct Point2d p = { 3.0, 4.0 };
    distance(p);
}
```


Anatomy of a native call

Linux x64

Passing Once arguments are classified, the registers get assigned (in left-to-right order) for passing as follows:

1. If the class is MEMORY, pass the argument on the stack.
2. If the class is INTEGER, the next available register of the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` is used[13].
3. If the class is SSE, the next available vector register is used, the registers are taken in the order from `%xmm0` to `%xmm7`.
4. If the class is SSEUP, the eightbyte is passed in the next available eightbyte chunk of the last used vector register.
5. If the class is X87, X87UP or COMPLEX_X87, it is passed in memory.

```
struct Point2d {
    double x;
    double y;
};

extern double distance(struct Point2d p);

void main(void) {
    struct Point2d p = { 3.0, 4.0 };
    distance(p);
}
```

```
movsd   xmm0, QWORD PTR .LC0[rip] ; 3
movsd   xmm1, QWORD PTR .LC1[rip] ; 4
call    distance
```

Anatomy of a native call

Windows x64

The following table summarizes how parameters are passed, by type and position from the left:

Parameter type	fifth and higher	fourth	third	second	leftmost
floating-point	stack	XMM3	XMM2	XMM1	XMM0
integer	stack	R9	R8	RDX	RCX
Aggregates (8, 16, 32, or 64 bits) and `__m64`	stack	R9	R8	RDX	RCX
Other aggregates, as pointers	stack	R9	R8	RDX	RCX
`__m128`, as a pointer	stack	R9	R8	RDX	RCX

```
struct Point2d {
    double x;
    double y;
};

extern double distance(struct Point2d p);

void main(void) {
    struct Point2d p = { 3.0, 4.0 };
    distance(p);
}
```

```
movups   xmm0, XMMWORD PTR p$[rsp]
movdqu   XMMWORD PTR $T1[rsp], xmm0
lea      rcx, QWORD PTR $T1[rsp]
call     distance
```

Downcall method handles

```
// extern double distance(struct Point2d p);
MemorySegment distanceAddress = SymbolLookup.loaderLookup()
                                           .lookup(“distance”).get();
MethodHandle  distanceHandle  = Linker.nativeLinker().downcallHandle(
                                         distanceAddress,
                                         FunctionDescriptor.of(JAVA_DOUBLE, POINT_2D));

try (Arena offHeap = Arena.ofConfined()) {
    MemorySegment point = offHeap.allocate(POINT_2D);
    xHandle.set(point, 0L, 3d);
    yHandle.set(point, 0L, 4d);
    double dist = distanceHandle.invokeExact(point); // 5d
}
```

Safety

Calling foreign functions is fundamentally **unsafe**

- Returned foreign pointers dereferenced incorrectly
- Provided function descriptors might be bad (wrong arity/types)
- Foreign code attempts to access already freed segments

Access to unsafe functionalities provided by **restricted methods**

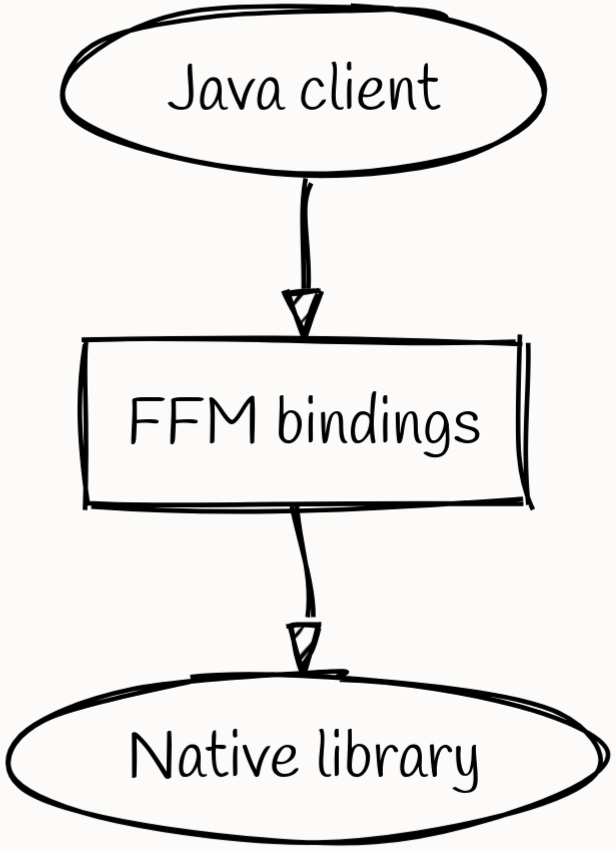
- Part of the SE API, runtime warning generated on first access
- Warnings can be disabled by granting selected modules native access

```
--enable-native-access <module-name>
```

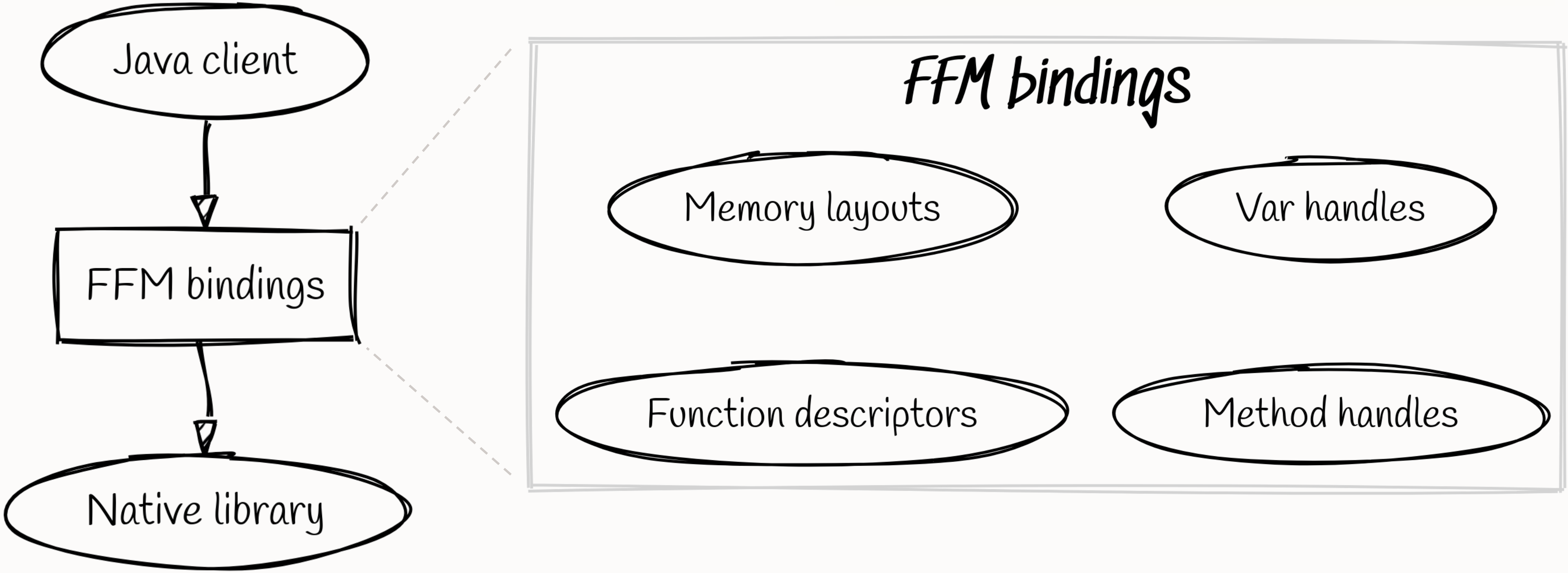
Restricted methods pave the way towards a **safer** Java/native interop

- JNI to follow, warnings **will** become errors
- Complete the “integrity by default” push started with Java 9

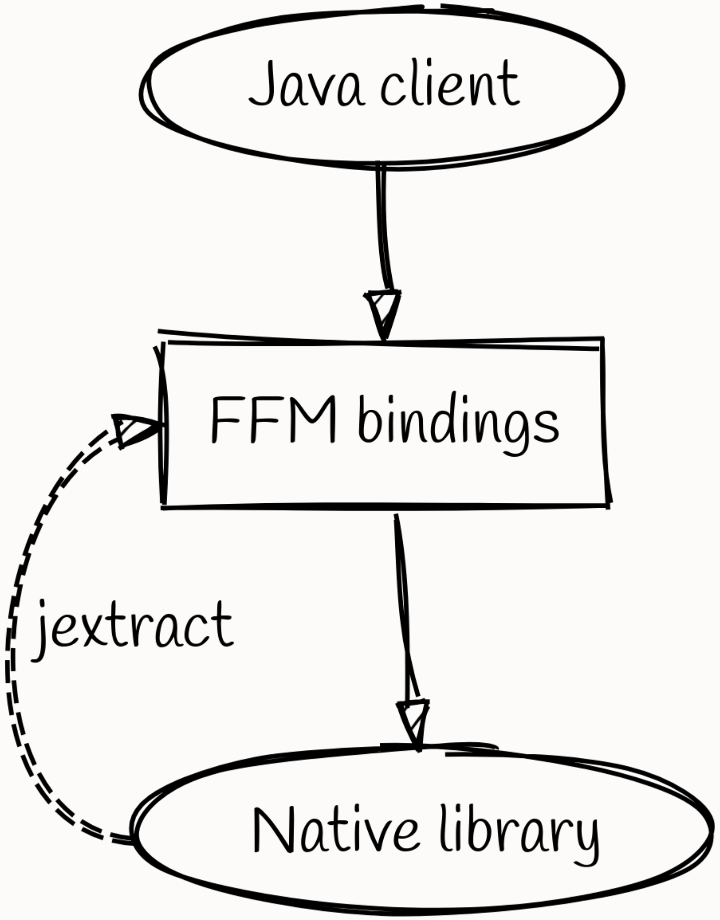
FFM API workflow



FFM API workflow



Enter jextract



Qsort with jextract

```
// stdlib.h
typedef int (*__compar_fn_t) (const void *, const void *);
void qsort (void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar);
```

Qsort with jextract

```
// stdlib.h
typedef int (*__compar_fn_t) (const void *, const void *);
void qsort (void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar);
```

```
$ jextract --target-package org.stdlib /usr/include/stdlib.h
```

Qsort with jextract

```
// stdlib.h
typedef int (*__compar_fn_t) (const void *, const void *);
void qsort (void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar);
```

```
$ jextract --target-package org.stdlib /usr/include/stdlib.h
```

```
import static org.stdlib.stdlib_h.*;
…

try (Arena offHeap = Arena.ofConfined()) {
    MemorySegment array = offHeap.allocateFrom(C_INT, 0, 9, 3, 4, 6, 5, 1, 8, 2, 7);

    var compareFunc = __compar_fn_t.allocate((a1, a2) ->
            Integer.compare(a1.get(C_INT, 0), a2.get(C_INT, 0)), offHeap);
    qsort(array, 10L, 4L, comparFunc);

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

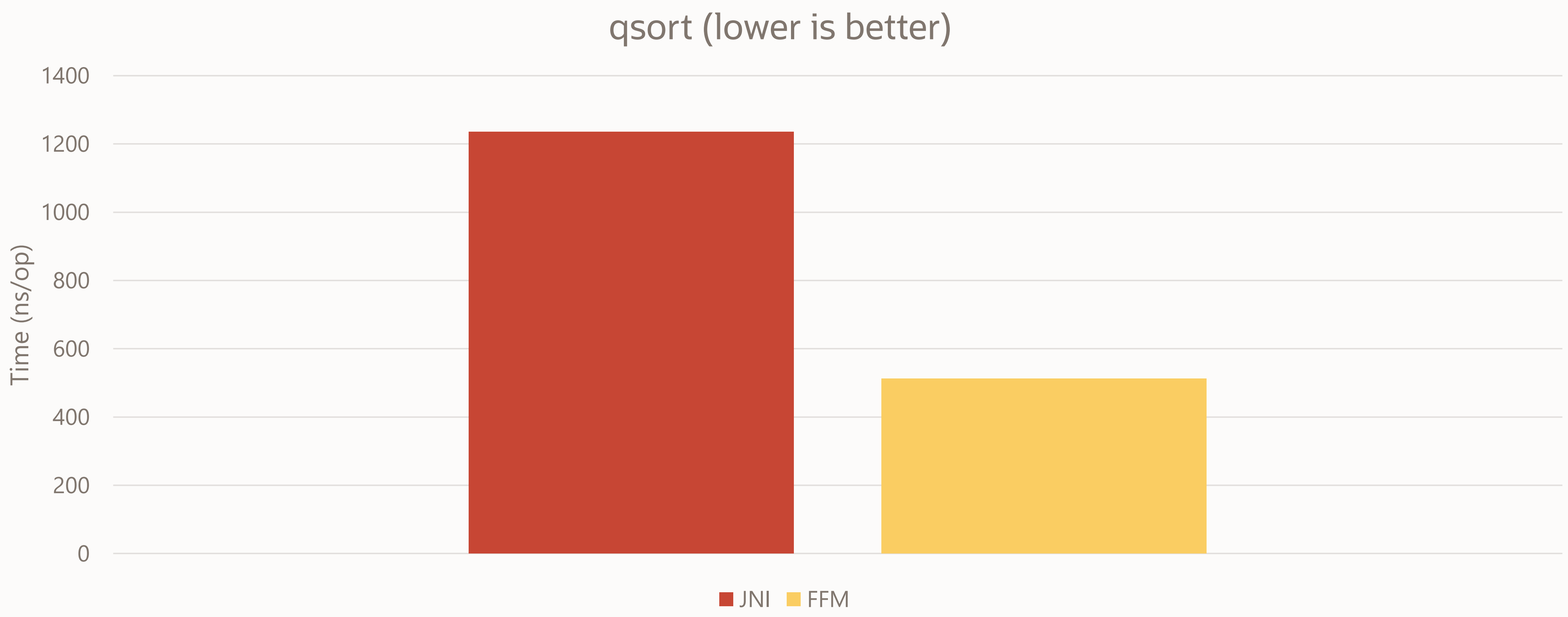

Qsort with JNI

```java
//qsort.java
class qsort {
    static {
        System.loadLibrary(“libqsort”);
    }

    static native void jni_qsort(int[] array);

    static int jni_upcall_compar(int j0, int j1) {
        return Integer.compare(j0, j1);
    }
}
```

```c
//qsort.h
#include <jni.h>
/* Header for class qsort */

#ifndef _Included_qsort
#define _Included_qsort
/*
 * Class:     qsort
 * Method:    jni_qsort
 * Signature: ([I)V
 */
JNIEXPORT void JNICALL Java_qsort_jni_1qsort
                       (JNIEnv *, jclass, jintArray);

#endif
```

```c
// libqsort.c
#include "qsort.h"

JavaVM* VM = NULL;

int java_cmp(const void *a, const void *b) {
   int v1 = *((int*)a);
   int v2 = *((int*)b);

   JNIEnv* env;
   (*VM)->GetEnv(VM, (void**) &env, JNI_VERSION_10);

   jclass qsortClass = (*env)->FindClass(env, "qsort");
   jmethodID methodId = (*env)->GetStaticMethodID(env, qsortClass, "jni_upcall_compar", "(II)I");

   return (*env)->CallStaticIntMethod(env, qsortClass, methodId, v1, v2);
}

JNIEXPORT void JNICALL Java_qsort_jni_1qsort(JNIEnv *env, jclass cls, jintArray arr) {
    if (VM == NULL) {
        (*env)->GetJavaVM(env, &VM);
    }

    jint* carr = (*env)->GetIntArrayElements(env, arr, 0);
    jsize length = (*env)->GetArrayLength(env, arr);
    qsort(carr, length, sizeof(jint), java_cmp);
    (*env)->ReleaseIntArrayElements(env, arr, carr, 0);
}
```

Performance



Wrapping up

The FFM API provides **safe** and **efficient** access to native memory

- Deterministic deallocation, layout API to enable structured access

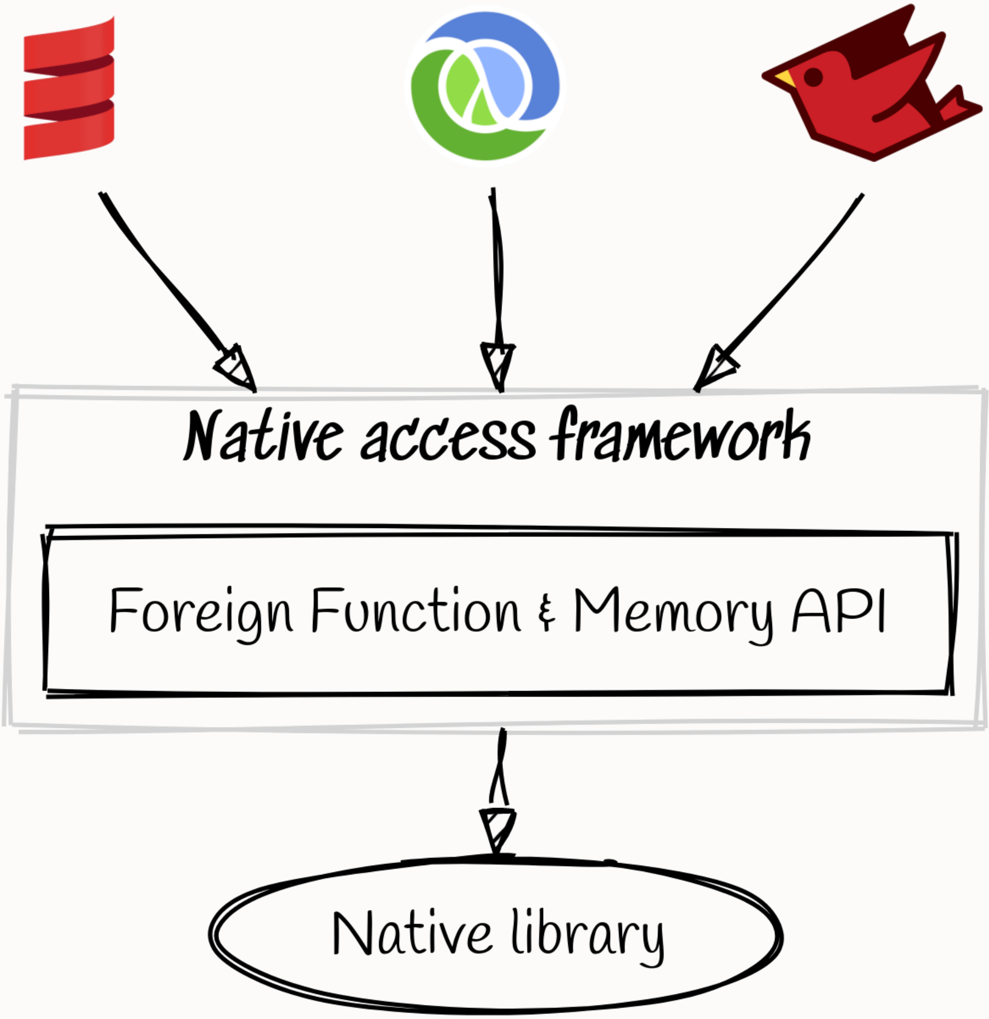
The FFM API provides **general**, **direct** and **efficient** access to native functions

- 100% Java, no need to write (and maintain!) native code

The FFM API provides the **foundations** of the Panama interop story

- Tooling (e.g. jextract) to generate layouts, var/method handles

A substrate for native access in the JVM



Adoption

Netty

Lucene

Apache Tomcat

TORNADOVM

Apache DataSketches

Infinileap

Useful links

Try the FFM API in JDK 22!

- https://jdk.java.net/22/
- https://openjdk.org/jeps/454
- Subscribe to panama-dev@openjdk.org and send feedback!

Generate FFM bindings with the jextract tool

- https://jdk.java.net/jextract/

Build the latest version of the FFM API & jextract

- https://github.com/openjdk/panama-foreign
- https://github.com/openjdk/jextract

ORACLE

Foreign Function & Memory API

A (quick) peek under the hood

Maurizio Cimadamore
Compiler Architect