Working with Native Libraries in Java

Vladimir Ivanov
HotSpot JVM Compiler
Oracle Corp.

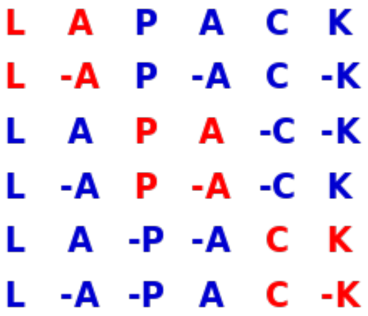
Twitter: @iwan0www
OpenJDK: vlivanov
23.04.2016

MAKE THE
FUTURE
JAVA

ORACLE

Why?

- LAPACK
  - Linear Algebra Package



L	A	P	A	C	K
L	-A	P	-A	C	-K
L	A	P	A	-C	-K
L	-A	P	-A	-C	K
L	A	-P	-A	C	K
L	-A	-P	A	C	-K

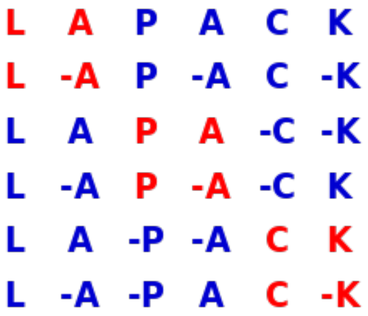
Why?

L A P A C K
L -A P -A C -K
L A P A -C -K
L -A P -A -C K
L A -P -A C K
L -A -P A C -K

- LAPACK
  - Linear Algebra Package
  - written in Fortran 90
  - highly optimized
    - “The original goal of the LAPACK was to … run efficiently on shared-memory vector and parallel processors.”

How?

- LAPACK
  - invoke library code
  - pass data into library
  - access data from Java



L	A	P	A	C	K
L	-A	P	-A	C	-K
L	A	P	A	-C	-K
L	-A	P	-A	-C	K
L	A	-P	-A	C	K
L	-A	-P	A	C	-K

Overview

- Existing
  - Java Native Interface (JNI) & JNR library
  - *java.nio.DirectByteBuffer*
  - sun.misc.Unsafe (get*/set*)
- JDK9
  - j.l.i.VarHandle views over ByteBuffers
- Future
  - Project Panama

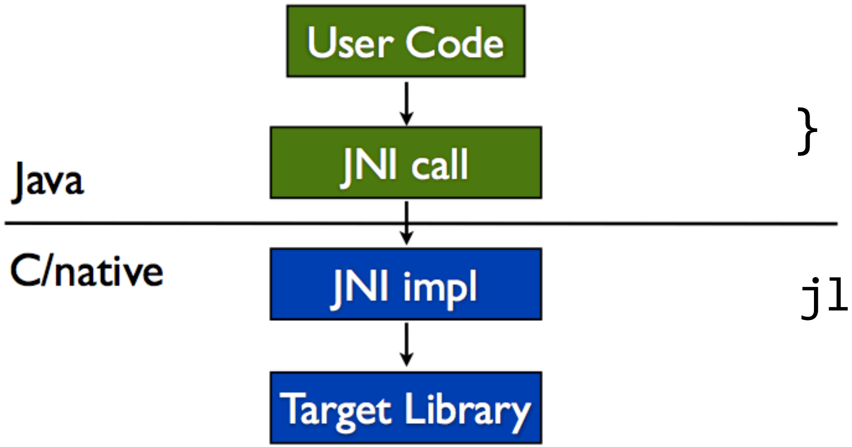
Native Code

JNI

@since 1.1

JNI

Usage scenario



User Code

JNI call

Java

C/native

JNI impl

Target Library

```
class LibC {
  static native long getpid();
}

jlong JNICALL Java_LibC_getpid(
  JNIEnv* env, jclass c) {
    return getpid();
}
```

JNI

Upcall

```
jlong JNICALL Java_...(JNIEnv* env,
                       jclass cls,
                       jobject obj) {


jmethodID mid = env->GetMethodID(cls, “m”, “(I)J”);


jlong result = env->CallLongMethod(obj, mid, 10);
```

JNI

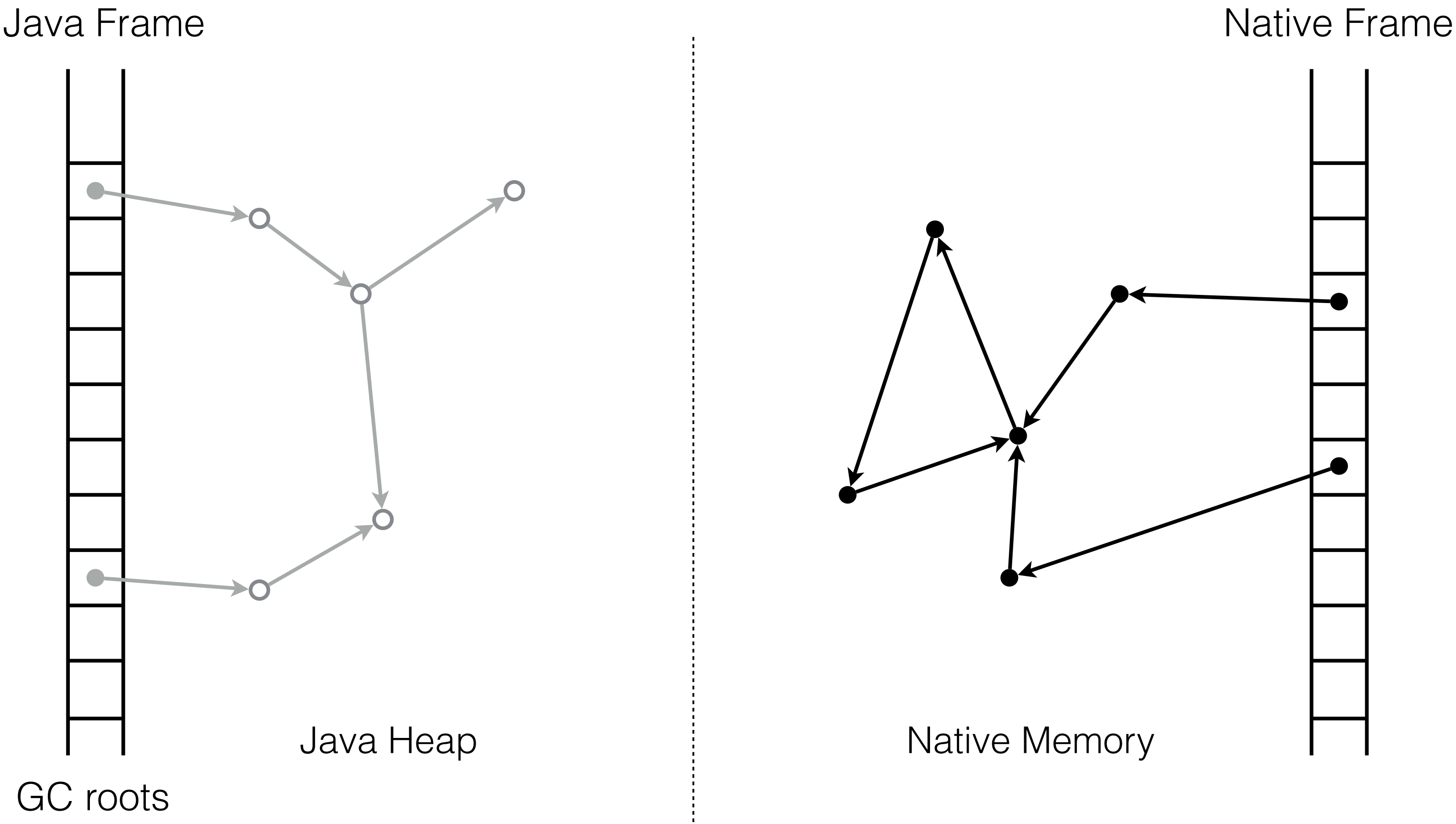
Data access

```
jlong JNICALL Java_...(JNIEnv* env,
                       jclass cls,
                       jobject obj) {

jfieldID fid = env->GetFieldID(cls, “f”, “J”);

 jlong result = env->GetLongField(obj, fid);
 jlong result = env->SetLongField(obj, fid, 10);
```

JNI

Native API: JNIEnv

- Operations on
  - Classes
  - Strings
  - Arrays
  - Monitors



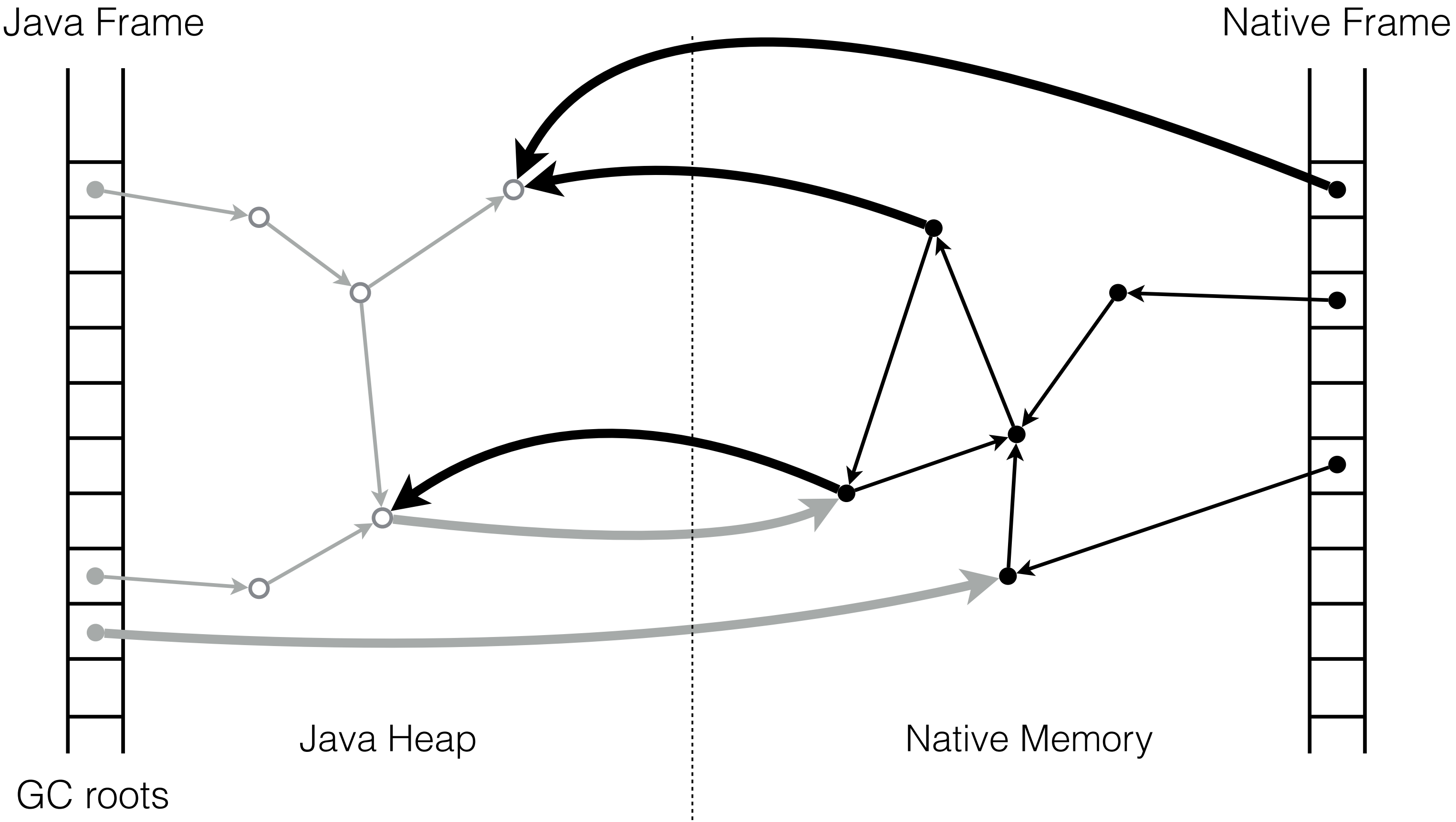
Java Frame

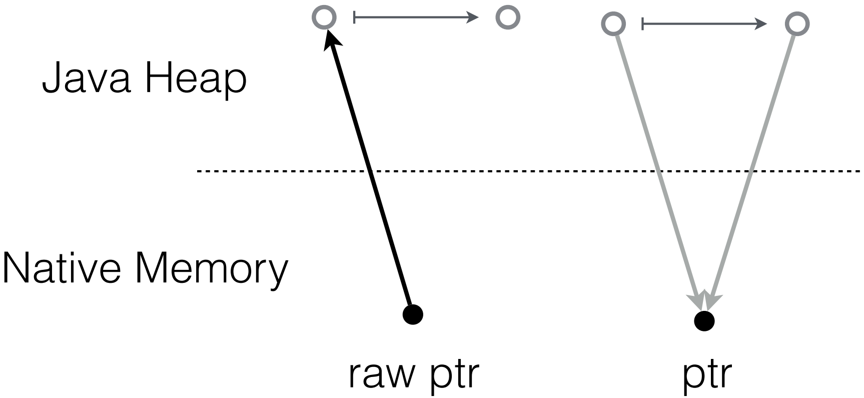
Native Frame

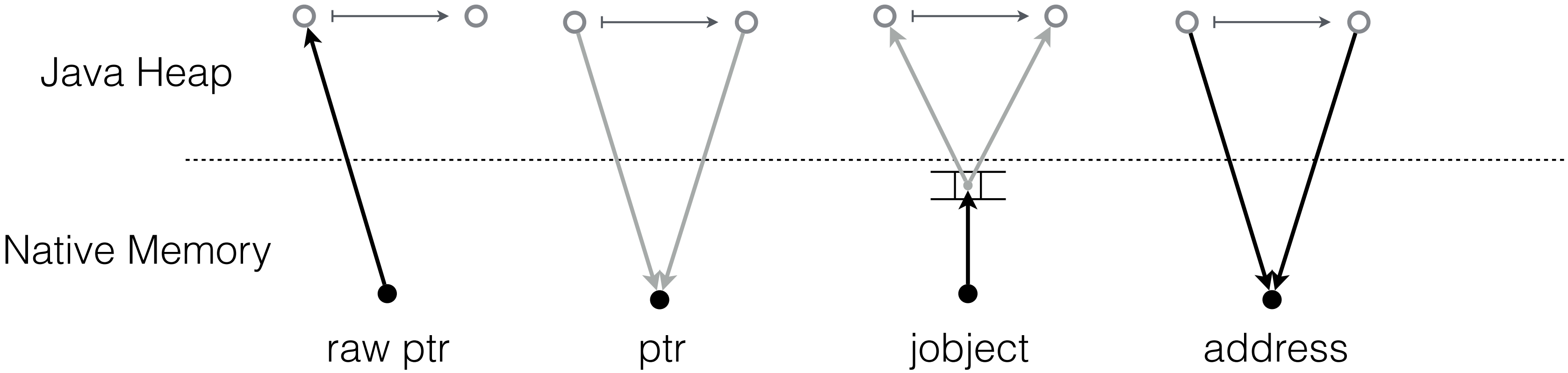
Java Heap

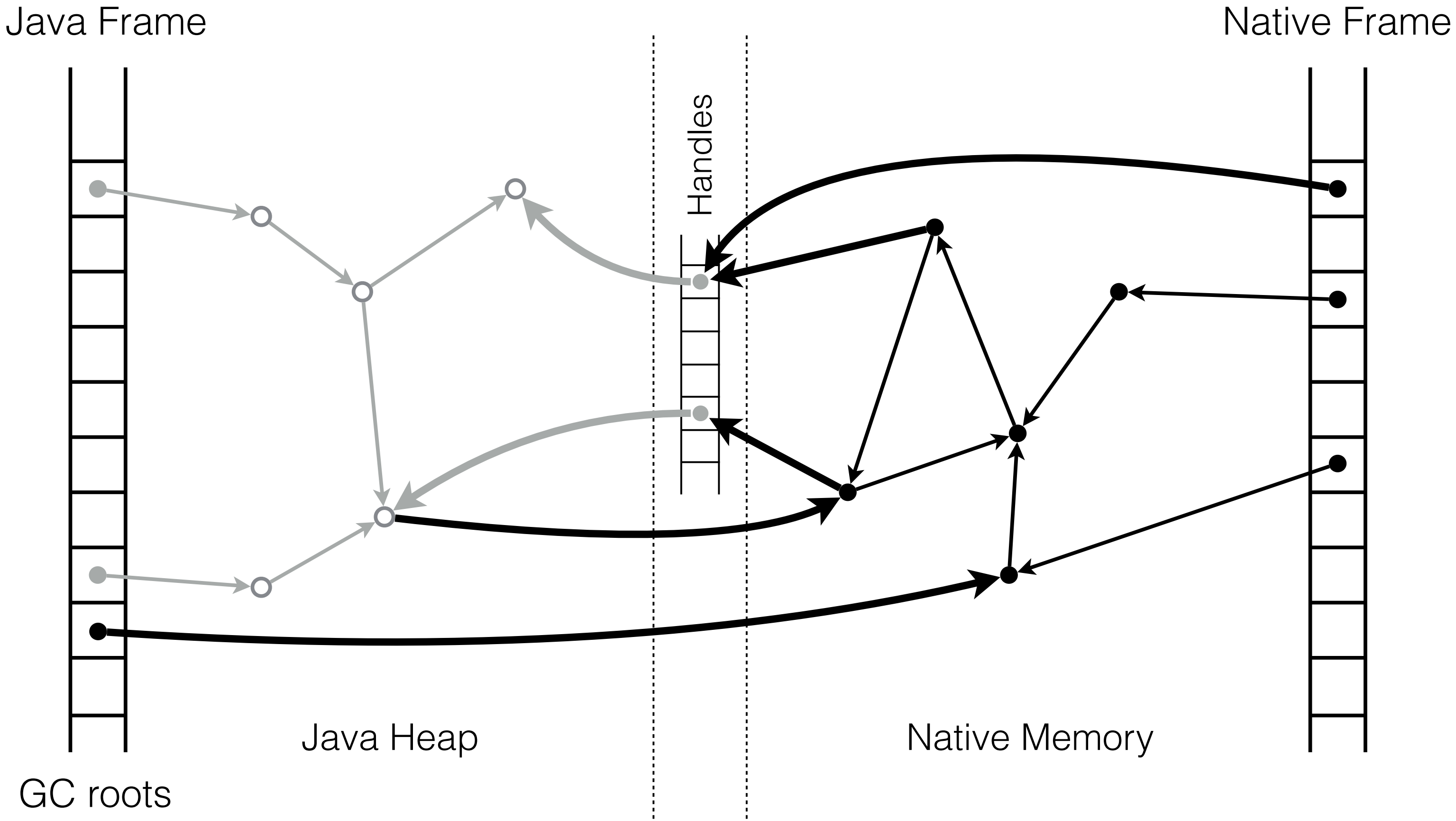
Native Memory

GC roots

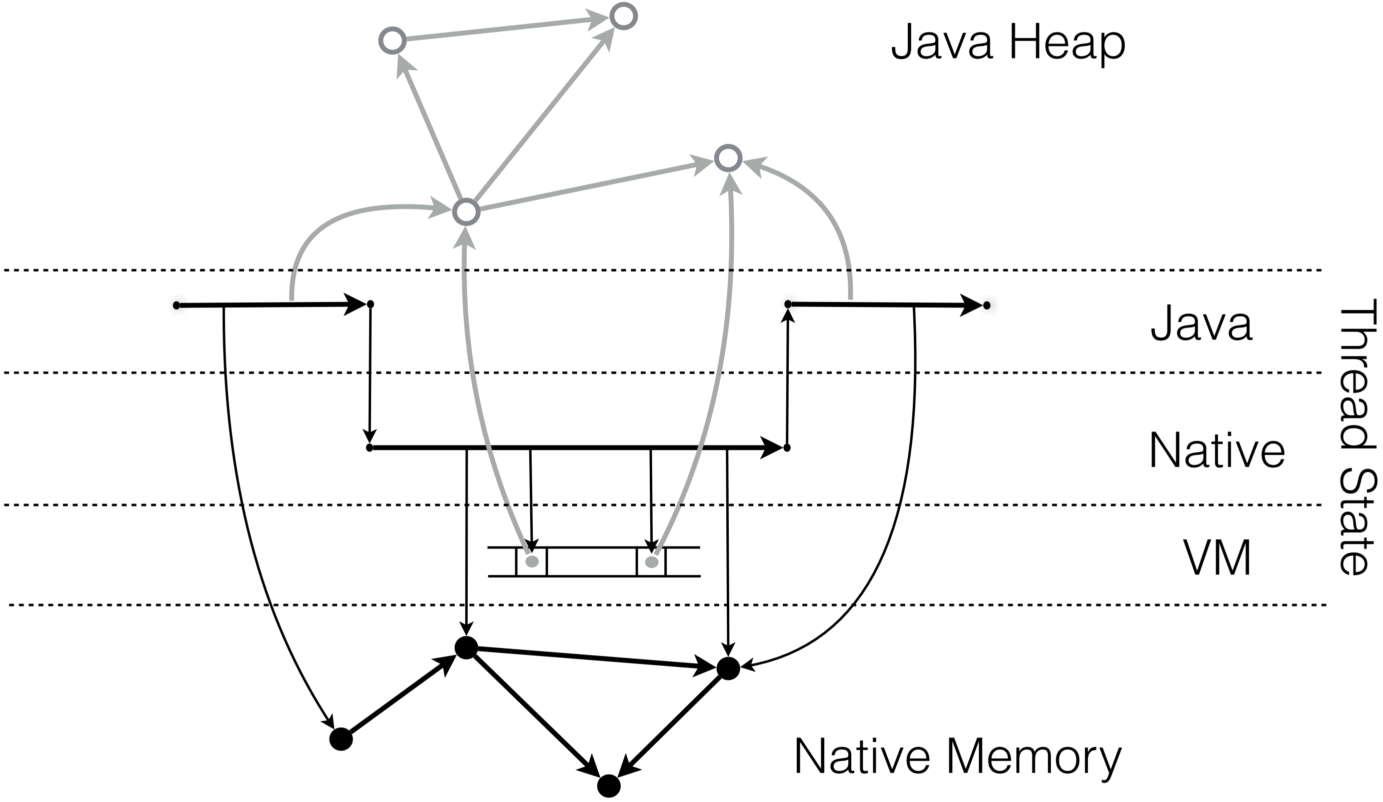

Java Frame
Native Frame
Java Heap
Native Memory
GC roots


Java Heap
Native Memory
raw ptr
ptr


Java Heap
Native Memory
raw ptr
ptr
jobject
address

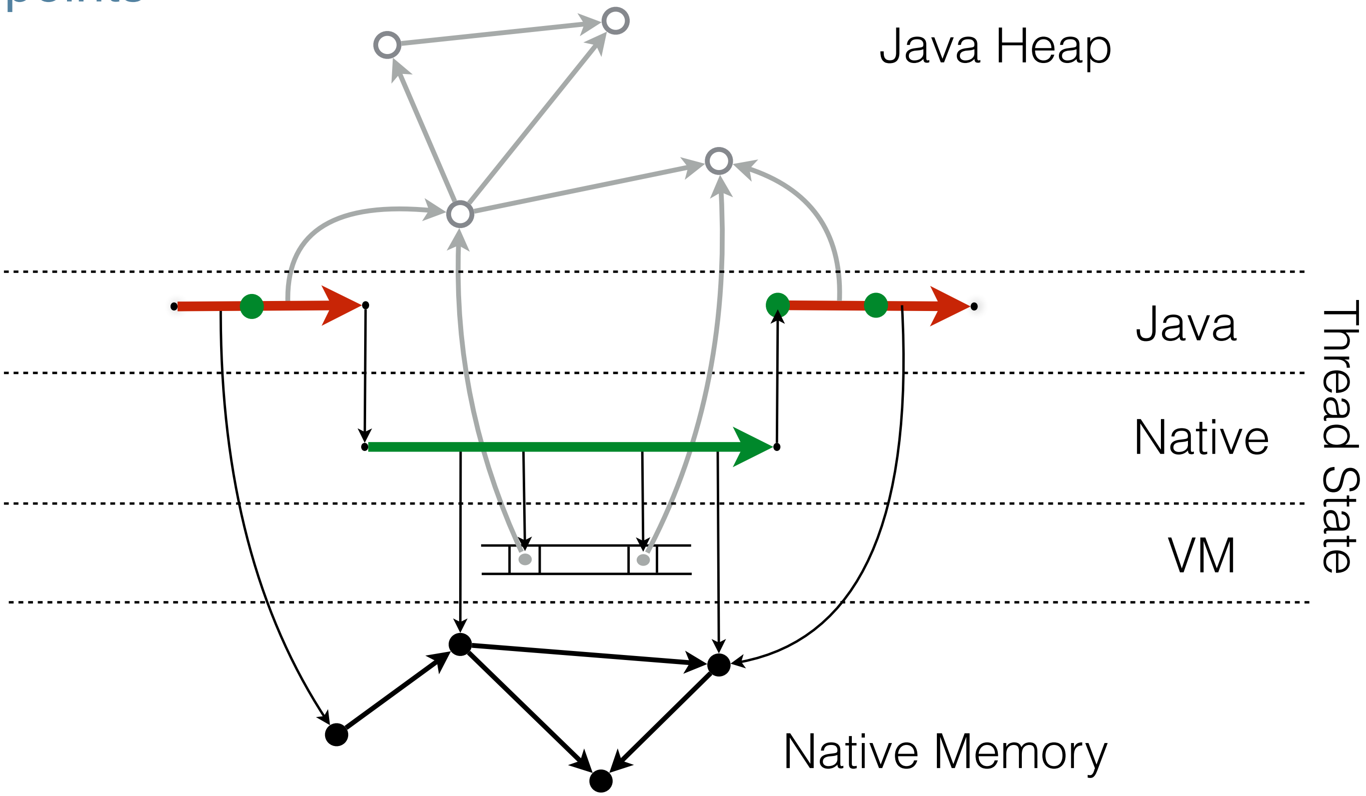

Java Frame
Native Frame
Handles
Java Heap
Native Memory
GC roots

Anatomy of JNI call



Anatomy of JNI call

Safepoints



JNI

- Pros
  - seamless integration
    - looks like a Java method
  - rich native API to interact with Java

- Cons
  - manual binding
  - invocation overhead

JNI

Victim of its own success?

JNI

Sum array elements

```
jint JNICALL Java_...(JNIEnv *env, jclass c, jobject arr) {
  jint len = (*env)->GetArrayLength(env, arr);
  jbyte* a = (*env)->GetPrimitiveArrayCritical(env, arr, 0);
  …
  return sum;
}
```

	empty	sum 1	sum 10^3	sum 10^6
JNI	11.4±0.3 ns	178.0±7.1 ns	798±32 ns	641±51 µs

ORACLE

Critical JNI

Sum array elements

```
jint JNICALL JavaCritical_...(jint length, jbyte* first) {
  ...
  return sum;
}
```

	empty	sum 1	sum 10^3	sum 10^6
JNI	11.4±0.3 ns	178.0±7.1 ns	798±32 ns	641±51 µs
CriticalJNI	11.4±0.3 ns	17.2±0.8 ns	680±22 ns	636±12 µs

Critical JNI

- only static, non-synchronized methods supported
- no JNIEnv*
- arguments: primitives or primitive arrays
  - [I => (length, I*)
  - null => (0, NULL)
- no object arguments

```
int printf(const char *format, ...)
```

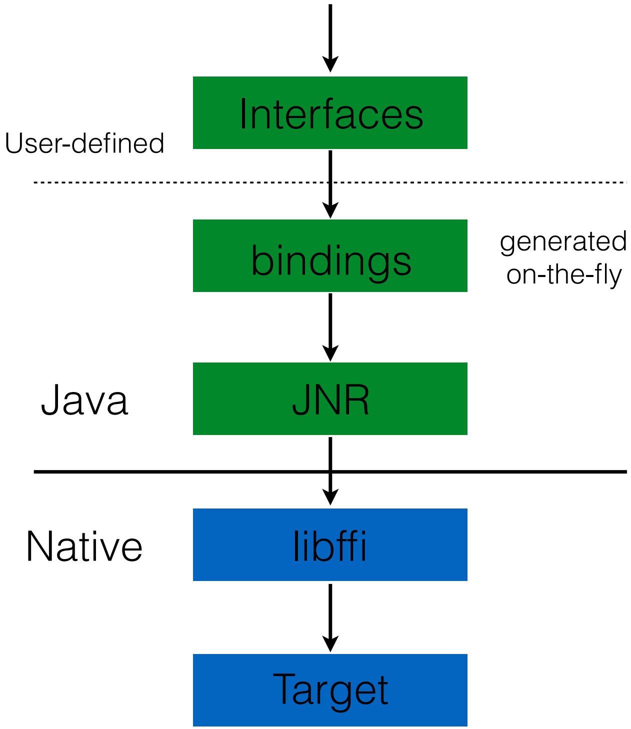
```
void qsort(
    void* base,
    size_t nel,
    size_t width,
    int (*cmp)(const void*, const void*));
```

JNR

Java Native Runtime

JNR

Usage scenario



User-defined

Interfaces

bindings

generated on-the-fly

Java

JNR

Native

libffi

Target

```
public interface LibC {
    @pid_t long getpid();
}

LibC lib = LibraryLoader
      .create(LibC.class)
      .load("c");

libc.getpid()
```

DEMO

- native call
  - getpid
- structs
  - gettimeofday
- upcalls
  - qsort

JNR

- Pros
  - automatic binding of native methods
- Cons
  - manual interface extraction
    - doesn’t scale
  - still uses JNI to perform native calls

Better JNI

Easier, safer, faster!

> “If non-Java programmers find some library useful and easy to access, it should be similarly accessible to Java programmers.”

John Rose, JVM Architect,

Oracle Corporation

Project Panama

“Bridging the gap”

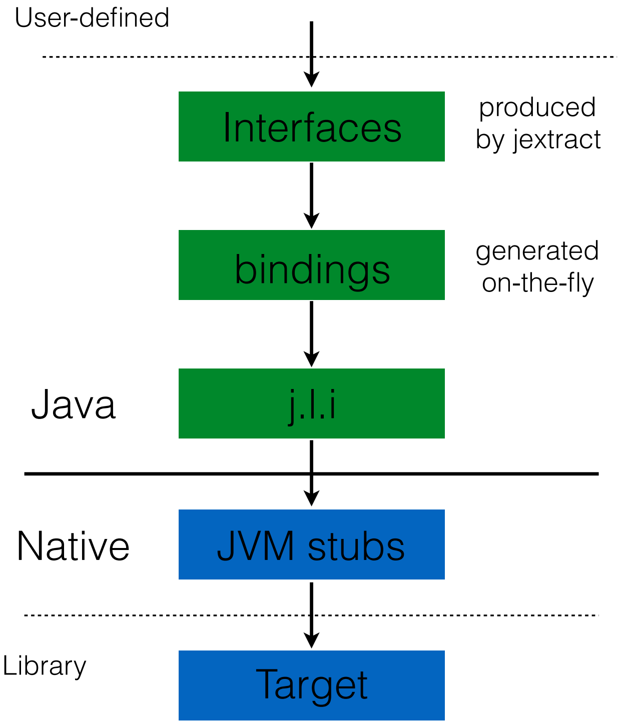
WORK IN
PROGRESS

Better JNI

```
pid_t get_pid();
```

Better JNI

Easier



```
public interface LibC {
    long getpid();
}

LibC libc = Library
      .load(LibC.class, “c”);

libc.getpid();
```

Better JNI

Easier

```
public interface LibC {
    long getpid();
}

LibC libc = Library.load(LibC.class, “c” /* lib_name */ );

libc.getpid();
```

Better JNI

Faster

```
callq 0x1057b2eb0  ; getpid entry
```

Better JNI

Faster

```
MethodType mt = MethodType.methodType(int.class); // pid_t
MethodHandle mh =
    MethodHandles.lookup().findNative("getpid", mt);

int pid = (int)mh.invokeExact();
```

	getpid
JNI	13.7 ± 0.5 ns
Direct call	3.4 ± 0.2 ns

Better JNI

Safer

- no crashes
- no leaks
- no hangs
- no privilege escalation
- no unguarded casts

Better JNI

Trust Levels

SECURITY CHECK

ALL PERSONS AND PROPERTY ARE SUBJECT TO SEARCH

- NO PACKAGES, BACKPACKS, COOLERS OR PARCELS
- NO WEAPONS OR FIREARMS
- NO EXPLOSIVES OR FLAMMABLES
- NO KNIVES OR SHARP OBJECTS
- NO PEPPER SPRAY OR MACE

PLEASE REPORT ANY SUSPICIOUS ACTIVITY TO SECURITY IMMEDIATELY

Untrusted

Better JNI

Trust Levels

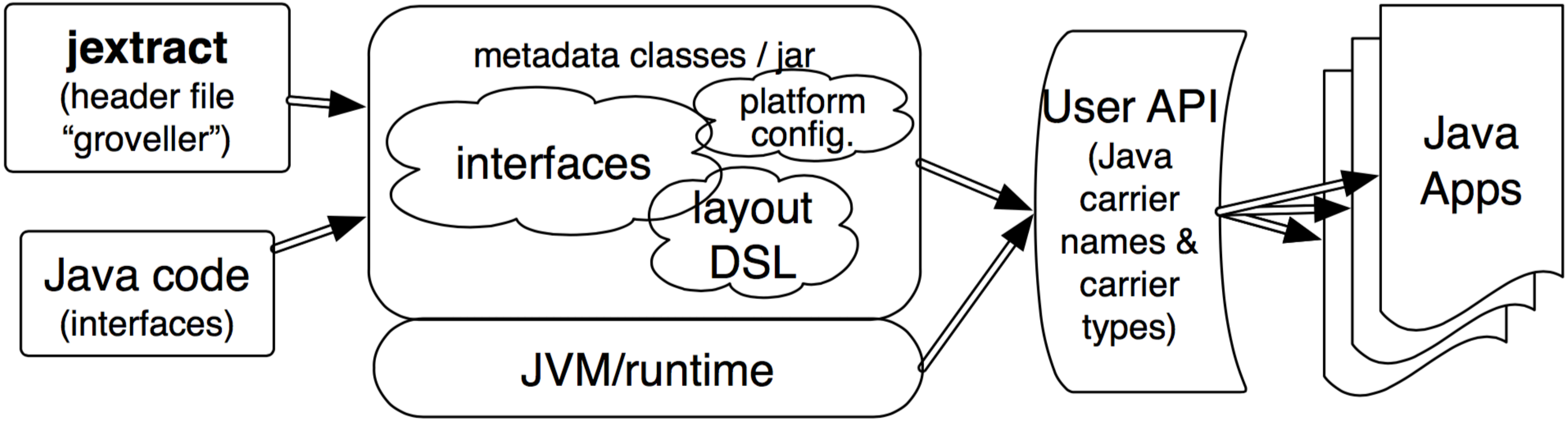
Trusted

Better JNI

Trust Levels

Privileged

Better JNI



jextract
(header file
“groveller”)
Java code
(interfaces)
metadata classes / jar
interfaces
platform
config.
layout
DSL
JVM/runtime
User API
(Java
carrier
names &
carrier
types)
Java
Apps

Better JNI

`gettimeofday`

```
/* time.h */

struct {                                    struct {
    time_t      tv_sec;                         int tz_minuteswest;
    suseconds_t tv_usec;                        int tz_dsttime;
} timeval;                                  } timezone;


int gettimeofday(struct timeval* tv, struct timezone* tz);
```

Carrier Types

- C

  `char`

  `short`

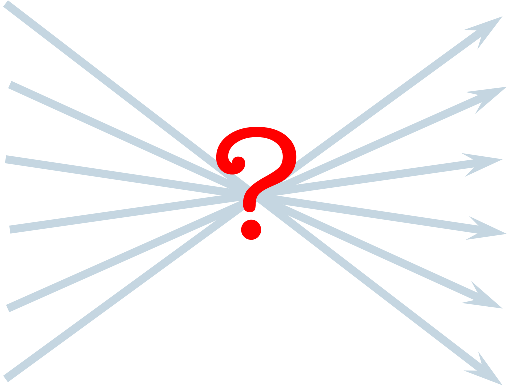
  `float`

  `int`

  `long`

  `long long`

  …



- Java

  `boolean`

  `byte`

  `short`

  `char`

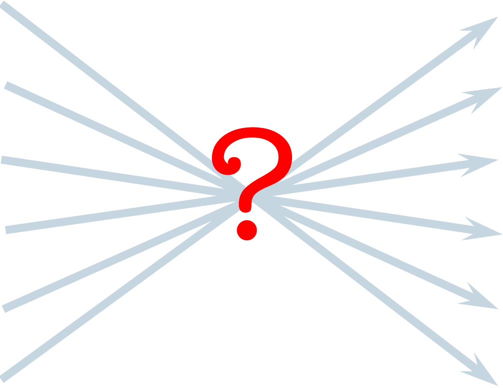
  `int`

  `long`

  …

Carrier Types

- C
  - `char`
  - `short`
  - `float`
  - `int`
  - `long`
  - `long long`
  - …



- Java

`boolean`	`(uint8_t)`
`byte`	`(int8_t)`
`short`	`(int16_t)`
`char`	`(uint16_t)`
`int`	`(int32_t)`
`long`	`(int64_t)`
…	

Better JNI

$ jextract **time.h**

```
interface Time {

interface Timeval {                 interface Timezone {
    long tv_sec$get();                  int  tz_...$get();
    void tv_sec$set(long);              void tz_...$set(int);
    long tv_usec$get();                 int  tz_...$get();
    void tv_usec$set(long);             void tz_...$set(int);
}                                   }


 int gettimeofday(Timeval, Timezone);
```

Foreign Layouts

- Native data requires special address arithmetic
  - Native layouts **should not** be built into the JVM
  - Native types are unsafe, so trusted code must manage the bits

- **Solution**: A metadata-driven Layout API

- As a bonus, layouts other than C and Java are naturally supported
  - Network protocols, specialized in-memory data stores, mapped files, etc.

Better JNI

Data Layout

```
interface Timeval {
…
    @Offset(offset=0L)
    long tv_sec$get();
…
    @Offset(offset=64L)
    long tv_usec$get();
…
```

- work on Layout Definition Language (LDL) is in progress
  - https://github.com/J9Java/panama-docs/blob/master/StateOfTheLDL.html

Better JNI

Runtime

```
Library lib = Library.create(“c”);

Time time = lib.create(Time.class);

Timeval tval = lib.create(Timeval.class);

int res = time.gettimeofday(tval, null);
if (res == 0) {
  long tv_sec  = tval.tv_sec$get();
  long tv_usec = tval.tv_usec$get();
}
```

Better JNI

Resources

```
Timeval tval;
try {
  tval = lib.create(Timeval.class);

  int res = time.gettimeofday(tval, null);
  if (res == 0) {
    long tv_sec  = tval.tv_sec$get();
    long tv_usec = tval.tv_usec$get();
  } else { /* error handling */ }
} finally {
  lib.free(tval);
  tval = null;
}
```

Better JNI

Resources

```
interface Timeval extends AutoCloseable { … }


try (Timeval tval = lib.create(Timeval.class)) {
  int res = time.gettimeofday(tval, null);
  if (res == 0) {
    long tv_sec  = tval.tv_sec$get();
    long tv_usec = tval.tv_usec$get();
  } else { /* error handling */ }
}
```

Better JNI

“Civilizer”

```
interface Timeval {
  void gettimeofday(Timeval, Timezone) throws ErrNo…;
}


try (Timeval tval = lib.create(Timeval.class)) {
  time.gettimeofday(tval, null); // throws exception
  long tv_sec  = tval.tv_sec$get();
  long tv_usec = tval.tv_usec$get();
}
```

Better JNI

Variadic Function

`int printf(const char *format, ...)`

Better JNI

jextract + Civilizer

```
//  int printf(const char *format, ...)

interface Stdio {
…
    // “Civilized”
    void printf(String format, Object… args);

    // “Raw”
    int printf(Pointer<Byte> format, byte[] args);
```

Optimize checks

```
void run(MyClass obj) {
    obj.nativeFunc1(); // checks & state trans.
    obj.nativeFunc2(); // checks & state trans.
    obj.nativeFunc3(); // checks & state trans.
}
```

Optimize checks

```
void run(MyClass obj) {
    obj.f1(); // NPE
    obj.f2(); // NPE
    obj.f3(); // NPE
}
```

Optimize checks

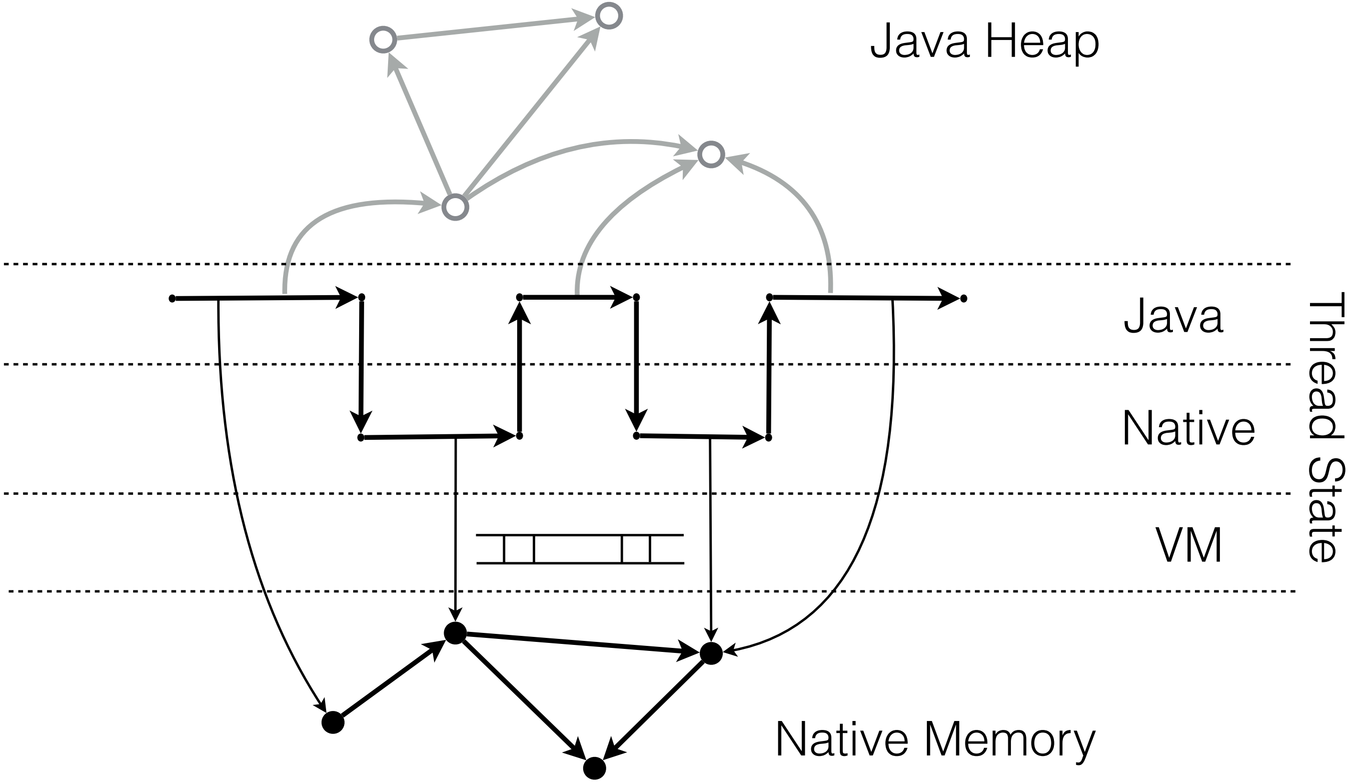
```
void run(MyClass obj) {
    if (obj == null) jump throwNPE_stub;
    call MyClass::f(obj);
    call MyClass::f1(obj);
    call MyClass::f3(obj);
}
```

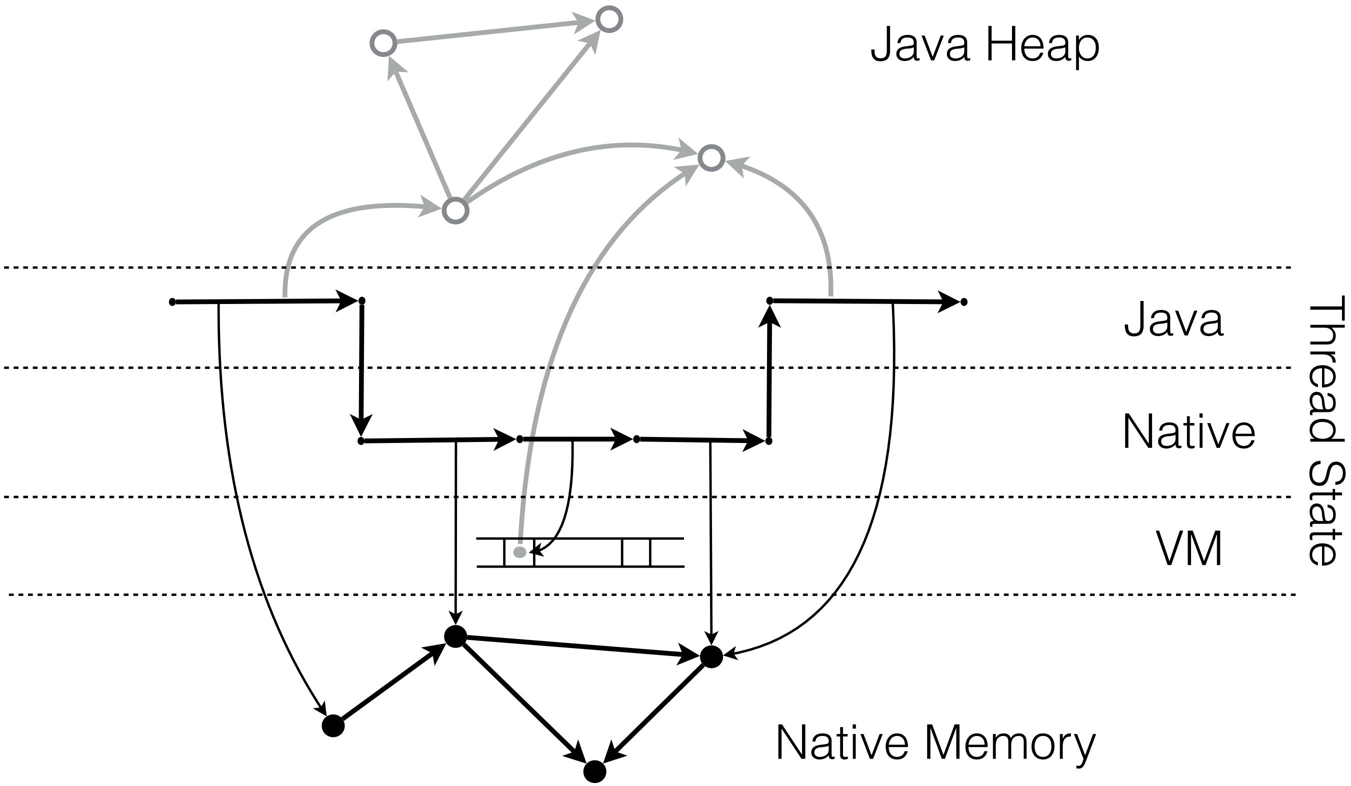
Optimize checks

```
void run(MyClass obj) {
    obj.nativeFunc1(); // checks & state trans.
    obj.nativeFunc2(); // checks & state trans.
    obj.nativeFunc3(); // checks & state trans.
}
```

Optimize checks

```
void run(MyClass obj) {
    if (!performChecks())  jump failed_stub;
    call transJavaToNative();
    MyClass::nativeFunc1(env, obj);
    MyClass::nativeFunc2(env, obj);
    MyClass::nativeFunc3(env, obj);
    call transNativeToJava();
}
```


Java Heap
Java
Native
VM
Thread State
Native Memory


Java Heap
Java
Native
VM
Thread State
Native Memory

Better JNI

Easier, Safer, Faster!

- Native access between the JVM and native APIs
  - Native code via FFIs
  - Native data via safely-wrapped access functions
  - Tooling for header file API extraction and API metadata storage
- Wrapper interposition mechanisms, based on JVM interfaces
  - add (or delete) wrappers for specialized safety invariants
- Basic bindings for selected native APIs

Native Data

NIO

@since 1.4

NIO

“New I/O”

- Provides access to the low-level I/O operations
  - Buffers for bulk memory operations
    - on-heap and off-heap
  - Character set encoders and decoders
  - Channels, a new primitive I/O abstraction
  - File interface
    - supports locks and memory mapping of files
  - Multiplexed, non-blocking I/O

java.nio.Buffer

- java.nio.ByteBuffer / CharBuffer / …
  - MappedByteBuffer extends ByteBuffer
    - *memory-mapped region of a file*
  - DirectByteBuffer extends MappedByteBuffer
    - malloc’ed native memory
  - HeapByteBuffer
    - backed by byte[]

java.nio.DirectByteBuffer

Usage

```
ByteBuffer dbb = ByteBuffer.allocateDirect(size);

dbb.rewind(); // reset position
while (dbb.hasRemaining()) {
  byte b = dbb.get();
}
```

java.nio.Buffer

- < 2GiB
  - ByteBuffer.allocateDirect(int size)
- Stateful
  - Buffer.position
  - not thread-safe
- Resource deallocation
  - GC-based (Cleaner) memory management
- Zeroing
  - on initialization
- Bounds checking

sun.misc.Unsafe

Anti-JNI

sun.misc.Unsafe

Use case	Example methods
Concurrency primitives	compareAndSwap*
Serialization	allocateInstance
Efficient memory management, layout, and access	allocateMemory/freeMemory get*/put*
Interoperate across the JVM boundary	get*/put*
…	…

sun.misc.Unsafe

- Unsafe.get*/put*
  - getInt(Object base, long offset)
  - putInt(Object base, long offset, int value);

- double-register addressing mode
  - getInt(o, offset) == o + offset
  - getInt(null, address) == address

- long allocateMemory(long size)

sun.misc.Unsafe

- Absolute addresses
  - byte getByte(long address)
  - int getInt(long address)
  - …

`UNSAFE.putInt(new Object(), 0L, 0)`

UNSAFE.putInt(null, 0L, 0)

Object UNSAFE.getObject(**long** address)

long UNSAFE.getAddress(**long** address)

UNSAFE.getObject(addr)

Unsafe =?= Fast

Unsafe != Fast

Unsafe != Fast

- public native Object **allocateInstance**(Class<?> cls) throws …;
- Array index vs raw offset

```
long[] base = new long[…];
int idx = …; long offset = (((long) idx) << SCALE + OFFSET)
long value = Unsafe.getLong(base, offset);
```

- JDK-8078629: “VM should constant fold Unsafe.get*() loads from final fields”

- How many of you have used the Unsafe API?

. . .

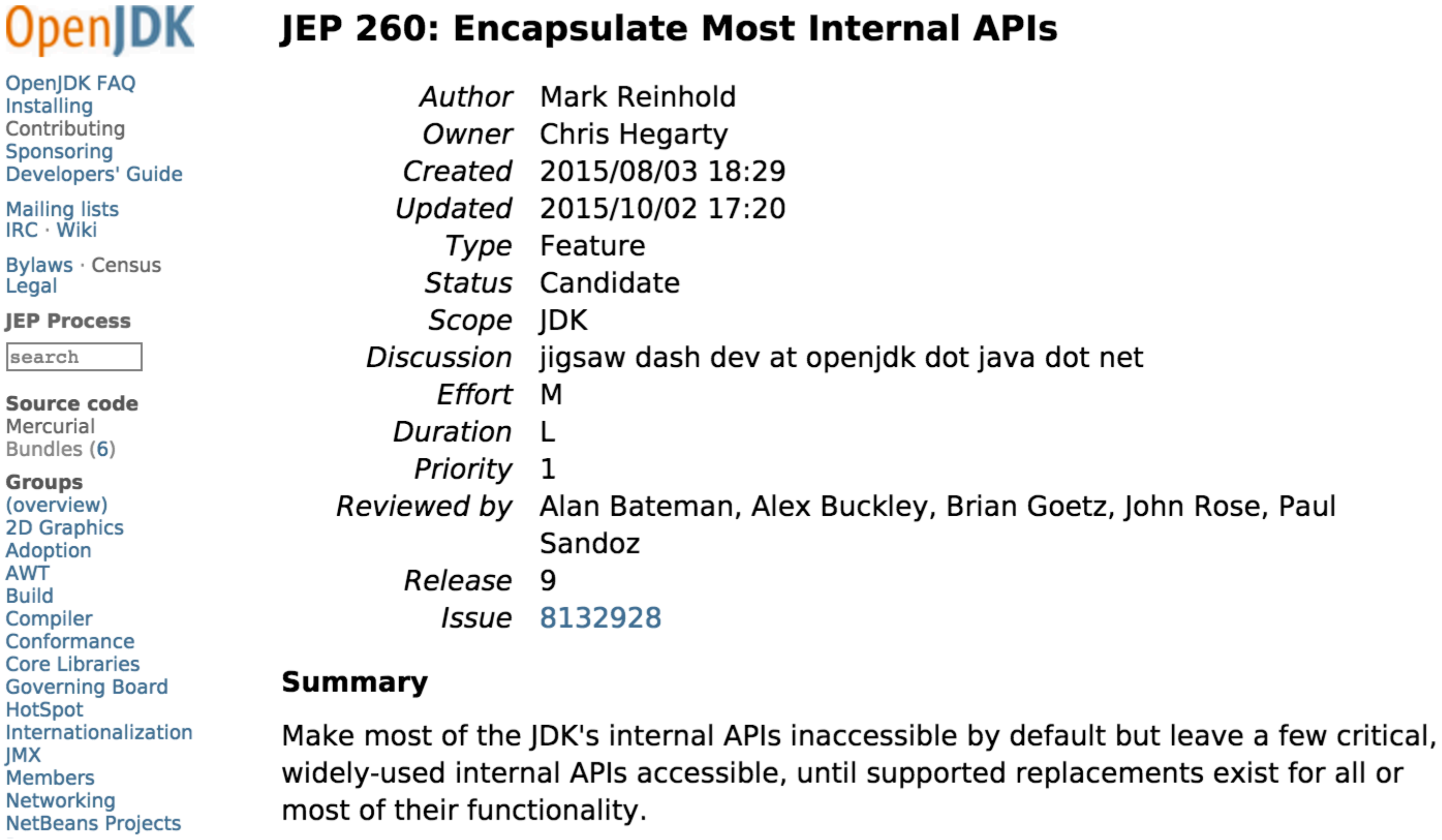
John Rose, JVM Architect, Oracle

JVM Language Summit 2014

> - How many of you have used the Unsafe API?
>
> …
>
> - A lot of you. Gosh. I'm sorry.

John Rose, JVM Architect, Oracle

JVM Language Summit 2014



OpenJDK

OpenJDK FAQ
Installing
Contributing
Sponsoring
Developers' Guide

Mailing lists
IRC · Wiki

Bylaws · Census
Legal

JEP Process

search

Source code
Mercurial
Bundles (6)

Groups
(overview)
2D Graphics
Adoption
AWT
Build
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
Internationalization
JMX
Members
Networking
NetBeans Projects

JEP 260: Encapsulate Most Internal APIs

Author	Mark Reinhold
Owner	Chris Hegarty
Created	2015/08/03 18:29
Updated	2015/10/02 17:20
Type	Feature
Status	Candidate
Scope	JDK
Discussion	jigsaw dash dev at openjdk dot java dot net
Effort	M
Duration	L
Priority	1
Reviewed by	Alan Bateman, Alex Buckley, Brian Goetz, John Rose, Paul Sandoz
Release	9
Issue	8132928

Summary

Make most of the JDK's internal APIs inaccessible by default but leave a few critical, widely-used internal APIs accessible, until supported replacements exist for all or most of their functionality.

sun.misc.Unsafe

Use case	Example methods
Concurrency primitives	compareAndSwap*
Serialization	allocateInstance (**ReflectionFactory.newConstructorForSerialization**)
Efficient memory management, layout, and access	allocateMemory/freeMemory get*/put*
Interoperate across the JVM boundary	get*/put*

sun.misc.Unsafe

Use case	Replacement
Concurrency primitives	JEP 193 Variable Handles
Serialization	Reboot JEP 187 Serialization Improvements
Efficient memory management, layout, and access	Project Panama, Project Valhalla, Arrays 2.0, Better GC
Interoperate across the JVM boundary	Project Panama, JEP 191 Foreign Function Interface

java.lang.invoke.
VarHandle

@since 9

[JEP 193: Variable Handles]

VarHandle

ByteBuffer View

```
MethodHandles.Lookup:
   VarHandle byteBufferViewVarHandle(Class<?> viewArrayClass,
                                     boolean bigEndian) {…}
```

*“Produces a **VarHandle** giving access to elements of a **ByteBuffer viewed** as if it were an **array of elements** of a different primitive component type to that of byte, such as int[] or long[].”*

VarHandle

ByteBuffer View

```
VarHandle VH =
    MethodHandles.byteBufferViewVarHandle(
        int[].class,
        ByteOrder.nativeOrder() == ByteOrder.BIG_ENDIAN);
```

```
ByteBuffer dbb = ByteBuffer.allocateDirect(size);
```

```
int v = (int)VH.get(dbb, idx);
```

java.nio.ByteBuffer vs VarHandle View

	DirectByteBuffer	VarHandle
Size	< 2 GiB	< 2 GiB
State	Yes	No
Resource management	GC-based	No (delegates to DBB)
Zeroing	Yes	No (delegates to DBB)
Bound checks	Yes (optimized)	Yes (optimized)

Optimized Bounds Checks

int[]

```
// null check + (index u< array.length)
return array[index];
```

Optimized Bounds Checks

int[]: Unsafe access

```
// bounds and null check
if (index < 0 || index >= array.length)
  throw new …();

long offset = BASE + (((long) index) << 2);
return UNSAFE.getInt(array, offset);
```

Optimized Bounds Checks

int[]: Unsafe access

```
// bounds and null check
index = Objects.checkIndex(index, array.length);



long offset = BASE + (((long) index) << 2);
return UNSAFE.getInt(array, offset);


@HotSpotIntrinsicCandidate
public static int checkIndex(int index, int length, …);
```

Summary

- Existing
  - Java Native Interface (JNI) & JNR library
  - *java.nio.DirectByteBuffer*
  - sun.misc.Unsafe (get*/set*)
- JDK9
  - j.l.i.VarHandle views over ByteBuffers
- Future
  - Project Panama

Project Panama

Foreign Function Interface
Data Layout Control
Vector API
Arrays 2.0

http://openjdk.java.net

Project Panama

panama-dev@openjdk.java.net

http://hg.openjdk.java.net/panama/panama

OpenJDK

http://openjdk.java.net

Safe Harbor Statement

MAKE THE
FUTURE
JAVA

JavaOne™

ORACLE®