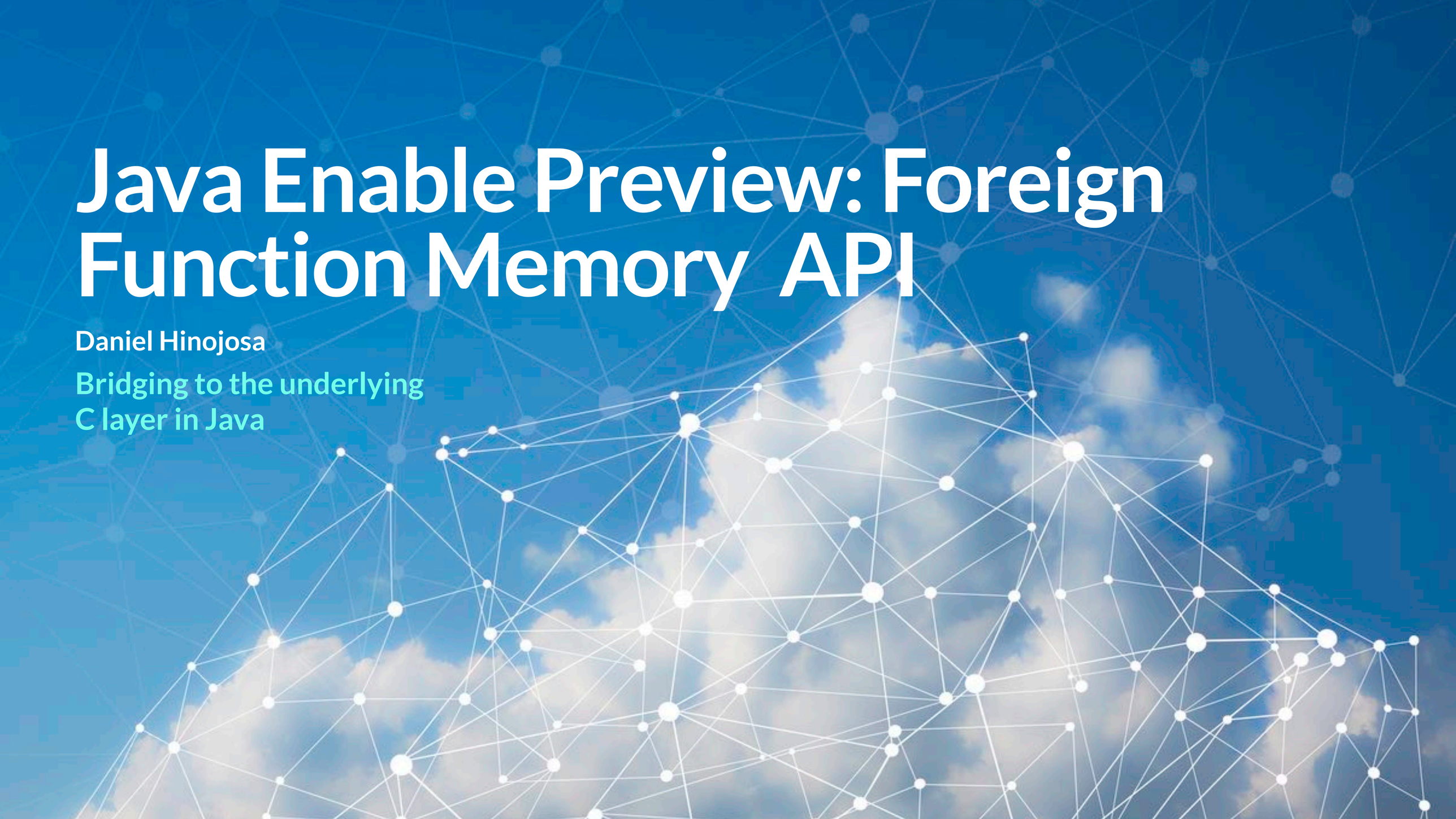
Java Enable Preview: Foreign Function Memory API

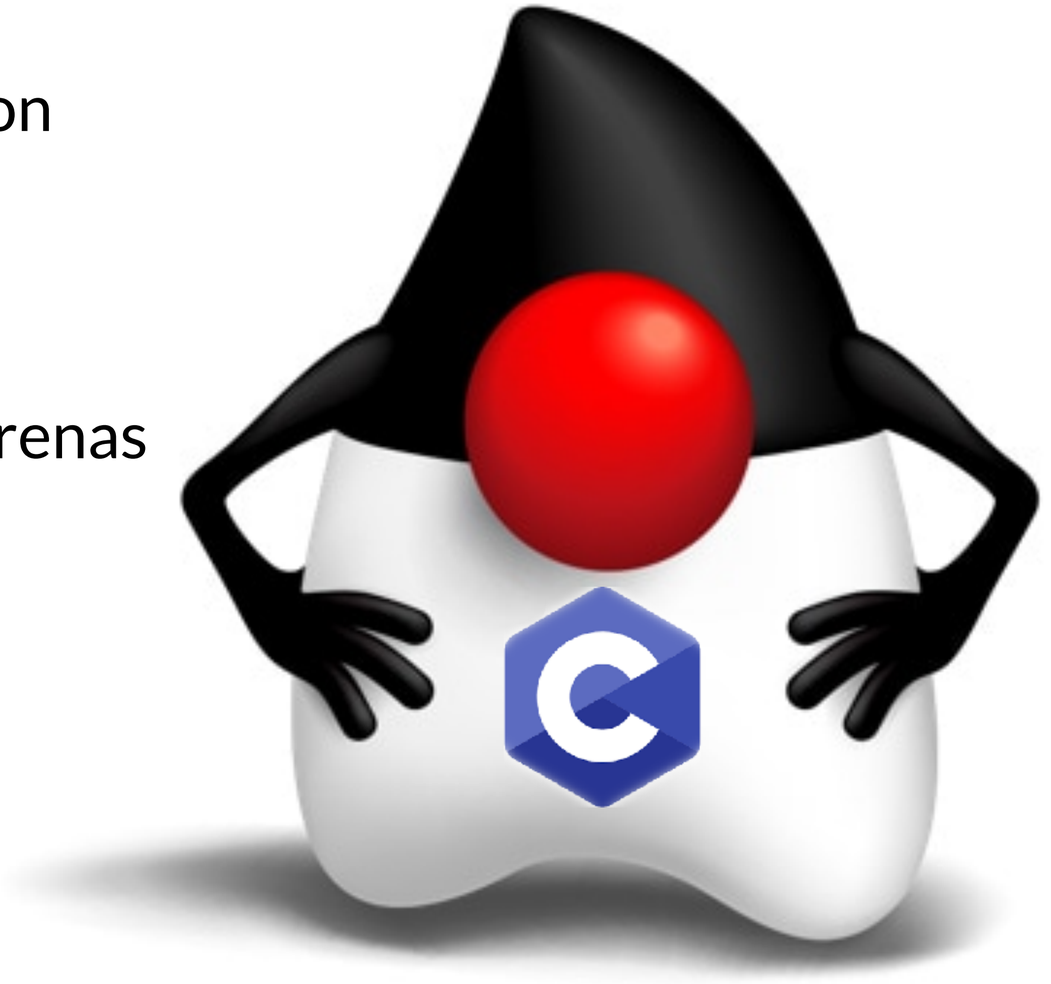
Daniel Hinojosa

Bridging to the underlying
C layer in Java



In this Presentation

- What is Foreign Function Memory API?
- What about JNI?
- Memory Segments & Arenas
- Linkers
- Upcalls and Downcalls
- Pointers
- `jextract`



Slides and Code are located at https://github.com/dhinojosa/java_enable_foreign_function_memory_api

Foreign Function Memory API

Foreign Function Memory API

- The Foreign Function and Memory (FFM) API enables Java programs to interoperate with code and data outside the Java runtime.
- This API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI.
- The API invokes foreign functions, code outside the JVM, and safely accesses foreign memory, memory not managed by the JVM.

Oracle Foreign Function and Memory API Documentation

Foreign Function Memory Features

- Memory Segment and Arenas
- Calling C Libraries
- Pointer Handling
- Memory Layouts and Structured Access
- Checking Native Errors
- Slicing Allocators and Slicing Memory
- Restricted Methods
- jextract

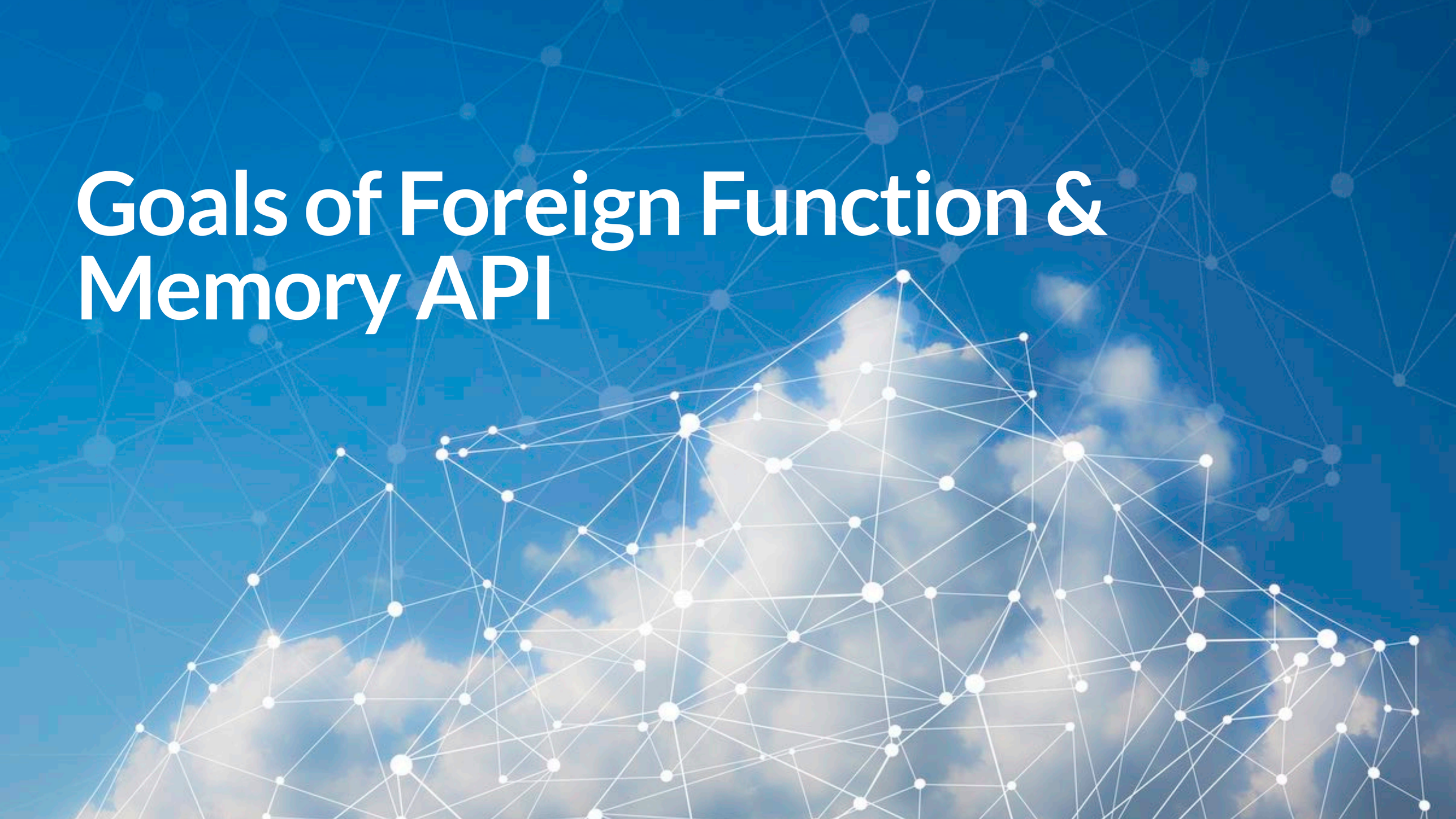
Oracle Foreign Function and Memory API Documentation

Who is going to use Foreign Function & Memory API?

Who is going to use Foreign Function & Memory API?

- Programmers who want to perform the following:
  - Image Processing
  - System Calls
  - Cryptography
  - Audio Processing
  - Device Interactivity
  - Sockets
- Foreign Function Memory *would typically not be a concern* for business application developers

Goals of Foreign Function & Memory API



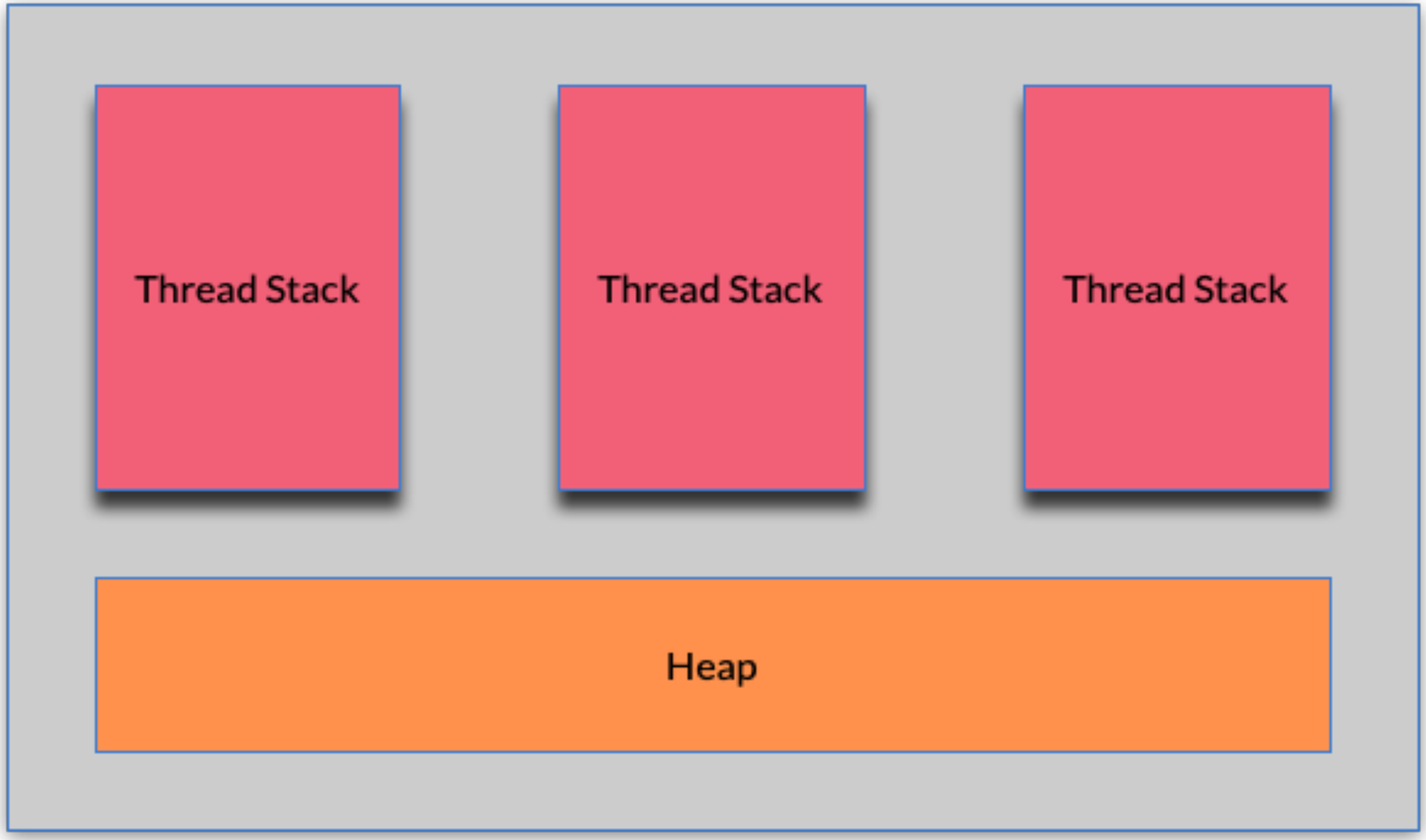
Goals of Foreign Function & Memory API

- **Productivity** — Replace the brittle machinery of native methods and the Java Native Interface (JNI) with a concise, readable, and pure-Java API.
- **Performance** — Provide access to foreign functions and memory with overhead comparable to, if not better than, JNI and `sun.misc.Unsafe`.
- **Broad platform support** — Enable the discovery and invocation of native libraries on every platform where the JVM runs.
- **Uniformity** — Provide ways to operate on structured and unstructured data, of unlimited size, in multiple kinds of memory (e.g., native memory, persistent memory, and managed heap memory).
- **Soundness** — Guarantee *no use-after-free* bugs, even when memory is allocated and deallocated across multiple threads.
- **Integrity** — Allow programs to perform unsafe operations with native code and data, but warn users about such operations by default.

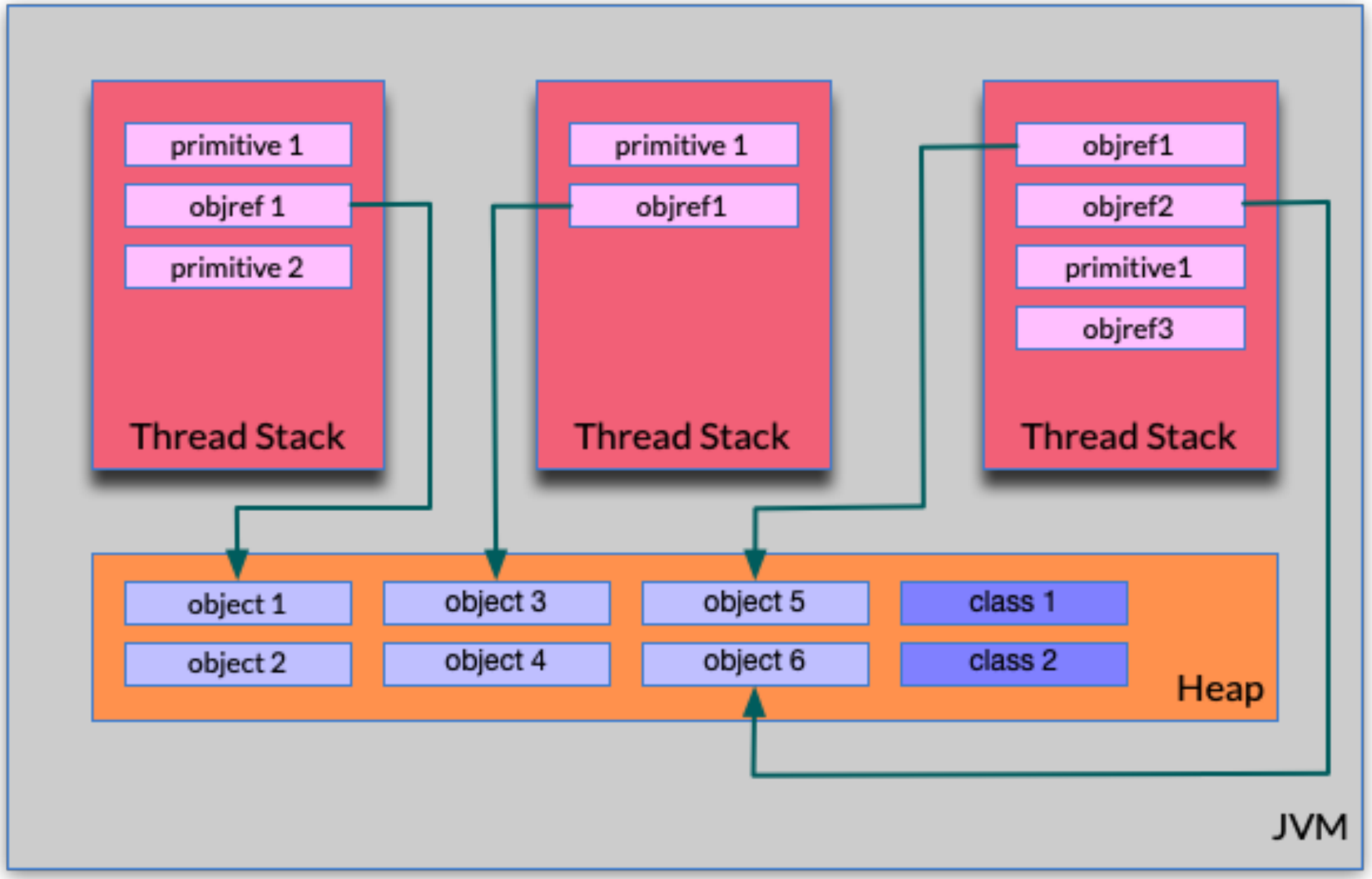
Oracle Foreign Function and Memory API Documentation

Off-Heap Memory

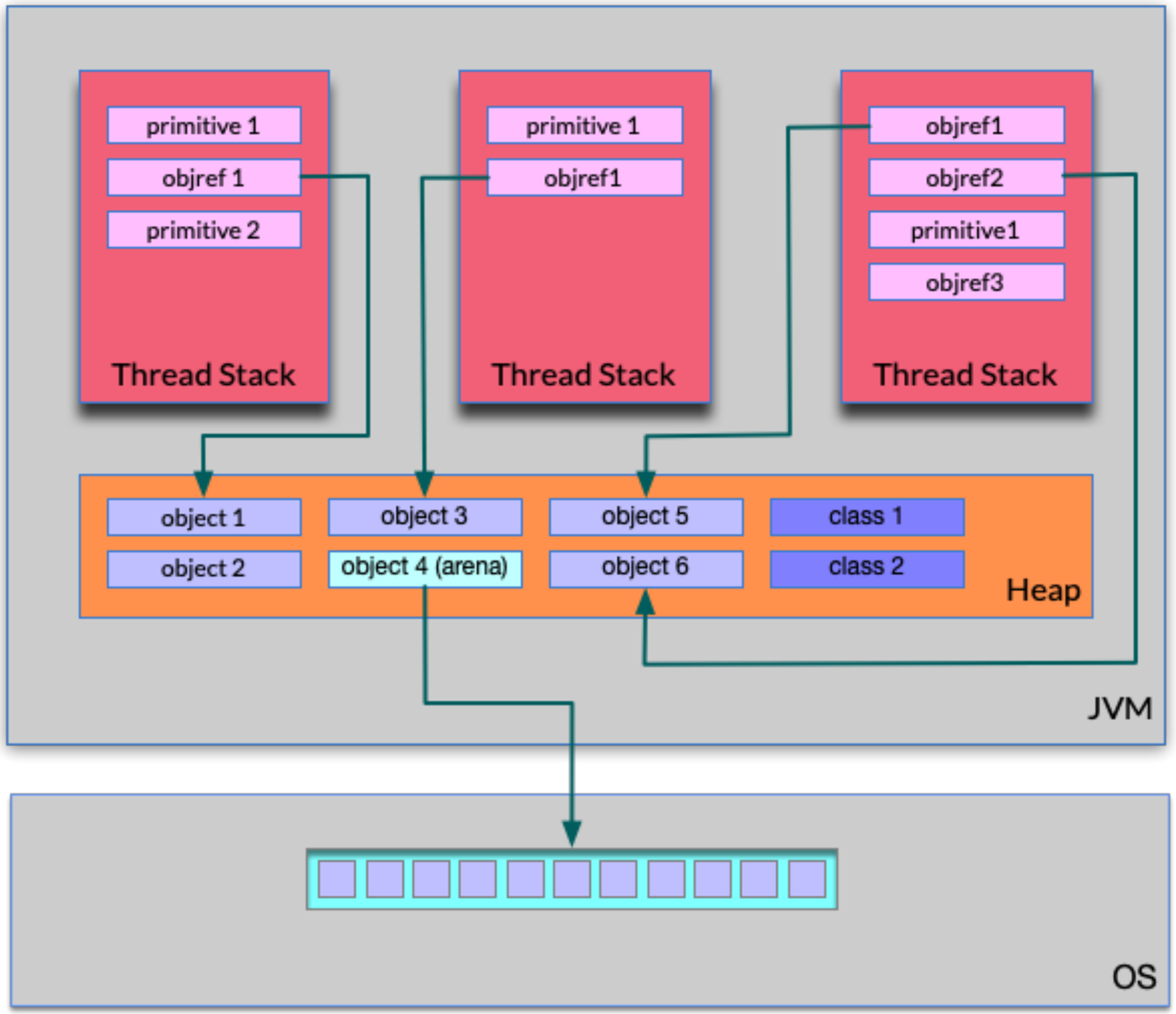
Java Memory Model



Java Memory Model



Java Memory Model with FFM API



The Issue with Garbage Collection

- Objects created with new keyword are stored in the heap where they are subject to garbage collection
- Garbage Collection is amazing and it keeps getting better for most applications where pause times are less than 10ms on large heaps. G1, ZGC, Shenandoah
- But for some performance-critical libraries, the pauses are too costly: *high-frequency trading*, *real-time audio*, *network packet processing*.

How can we Perform Off-Heap now?

- ByteBuffer API
- `sun.misc.Unsafe`

ByteBuffer API

- 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 developers cannot control.

Oracle Foreign Function and Memory API Documentation

`sun.misc.Unsafe`

- 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 intrinsic to 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 developers too much control.
- A library in a long-running application can allocate and interact with multiple regions of off-heap memory over time; data in one region can 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.

Oracle Foreign Function and Memory API Documentation

Foreign Memory in FFM API

- The new Foreign Memory API allows us to:
  - Allocate, manipulate, and share off-heap memory with the same fluidity and safety as on-heap memory
  - Have predictable deallocation with the need to prevent premature deallocation, which can lead to JVM crashes or, worse, to silent memory corruption.

Foreign Functions

What was wrong with JNI?

- Introduced in Java 1.1, JNI allows native calls — but at a cost.
  - Requires multiple toolchains: Java code, generated C headers, and native code.
  - Painful to maintain with evolving native libraries.
  - Limited to C/C++ Application Binary Interface (ABI)— no support for non-C calling conventions.
  - No type system alignment: Java objects ≠ C structs.
  - Passing objects requires manual unpacking in C via JNI API.
  - Workarounds: Load memory using Unsafe and calling JNI with dangerous memory handling

What does FFM API bring?

What does FFM API bring?

- Consume any native library deemed useful for a particular task, without the tedious glue and clunkiness of JNI
- Two excellent abstractions
  - **Method handles** - Direct references to method-like entities
  - **Variable handles** - Direct references to variable-like entities
- Simplifies 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.

Capabilities, Interfaces, and Purpose

Capability	FFM Classes/Interfaces	Purpose
Allocate & Deallocate Foreign Memory	`MemorySegment`, `Arena`, `SegmentAllocator`	Manage off-heap memory safely and explicitly
Access Structured Memory	`MemoryLayout`, `VarHandle`	Define and read/write structured native data layouts
Call Foreign Functions	`Linker`, `SymbolLookup`, `FunctionDescriptor`, `MethodHandle`	Locate and invoke native functions across the boundary

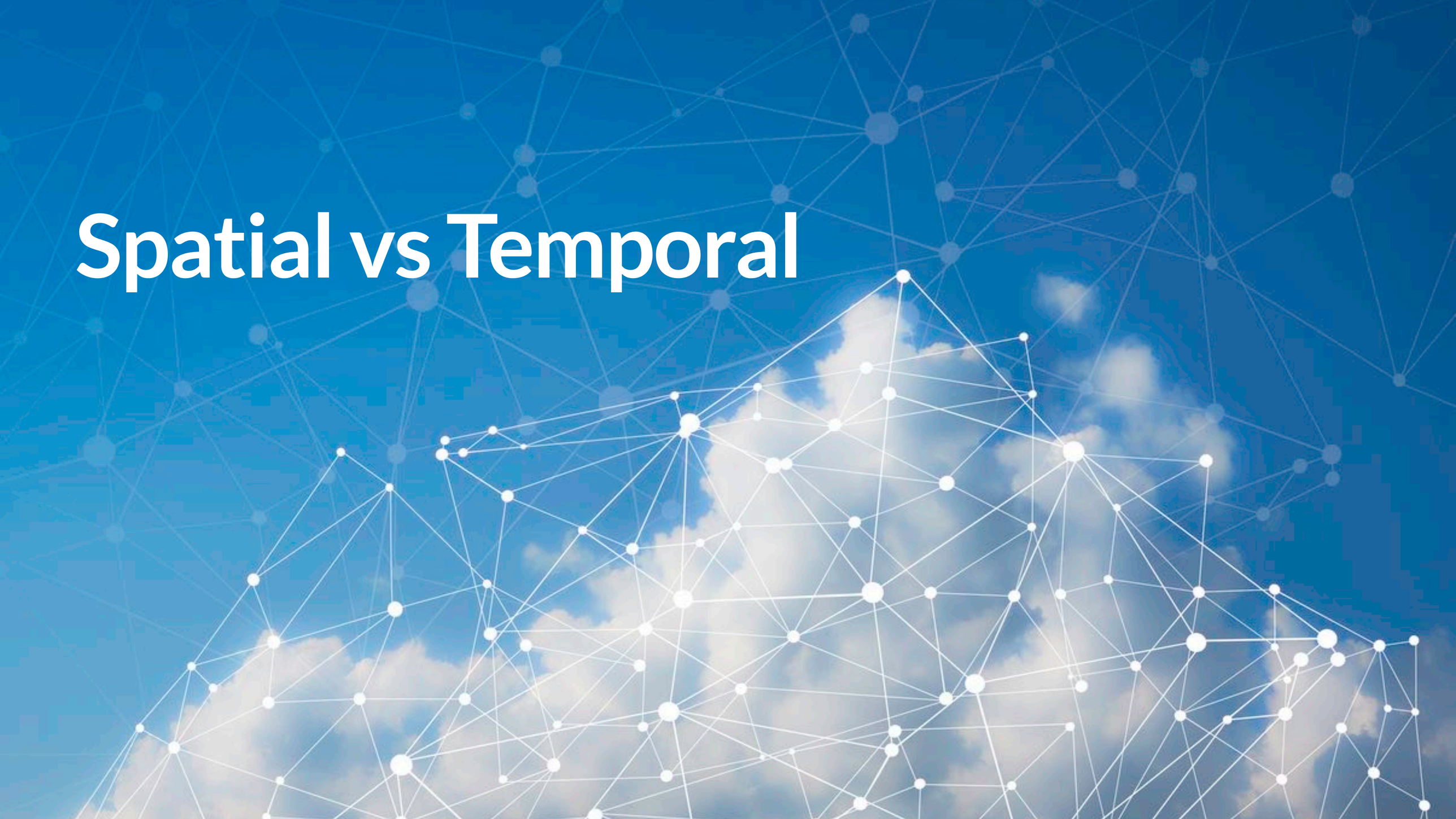
The FFM API resides in the `java.lang.foreign` package of the `java.base` module.

Project Info & Release

Project Info & Release

- Foreign Function Memory API is a part of Project Panama
- Project Panama includes FFM, `jextract`, Vector API (still in incubator)
- Foreign Function Memory API is official as a part of the JDK 22 delivery

Spatial vs Temporal



Spatial vs Temporal

- All memory segments provide spatial and temporal bounds which ensure that memory access operations are safe. Bounds guarantee no use of unallocated memory and no use-after-free.
  - **Spatial bounds** of a segment determine the range of memory addresses associated with the segment
  - **Temporal bounds** of a segment determine its lifetime, that is, the period until the region of memory which backs the segment is deallocated.

Arenas

Arenas

- Arenas control the lifecycle of native memory segments.
- Each arena has a scope, which specifies when the region of memory that backs the memory segment will be deallocated and is no longer valid.
- A memory segment can only be accessed if the scope associated with it is still valid or alive.

Four kinds of Arenas

- A **confined** arena
- A **shared** arena
- An **automatic** arena
- A **global** arena

Confined Arena

- Created with `Arena::ofConfined`
- A confined arena provides a bounded and deterministic lifetime
- Its scope is alive from when it’s created to when it’s closed.
- A confined arena has an owner thread, and is typically the thread that created it.
- Only the owner thread can access the memory segments allocated in a confined arena.
- You’ll get an exception if you try to close a confined arena with a thread other than the owner thread.

Shared Arena

- Created with `Arena::ofShared`
- It has no owner thread.
- Multiple threads may access the memory segments allocated in a shared arena.
- Any thread may close a shared arena, and the closure is guaranteed to be safe and atomic.

Automatic Arena

- Created with `Arena::ofAuto`.
- This is an area that's managed, automatically, by the garbage collector. Any thread can access memory segments allocated by an automatic arena.
- If you call `Arena::close` on an automatic arena, you'll get a UnsupportedOperationException.

Global Arena

- Created with `Arena::global`.
- Any thread can access memory segments allocated with this arena.
- In addition, the region of memory of these memory segments is never deallocated
- If you call `Arena::close` on a global arena, you'll get a `UnsupportedOperationException`.

Arena Summary

Type	Lifetime	Thread Access	Temporal Bounds
Global	Unbounded Lifetime	Multithreaded Access	Program Lifetime
Automatic	Automatic Lifetime	Multithreaded Access	Auto on `close` or GC
Confined	Explicit Bounded Lifetime	Single Threaded Access	Manual (you call `close`)
Shared	Explicit Bounded Lifetime	Multithreaded Access	Manual (you call `close`)

All spatial bounds are enforced

Linkers

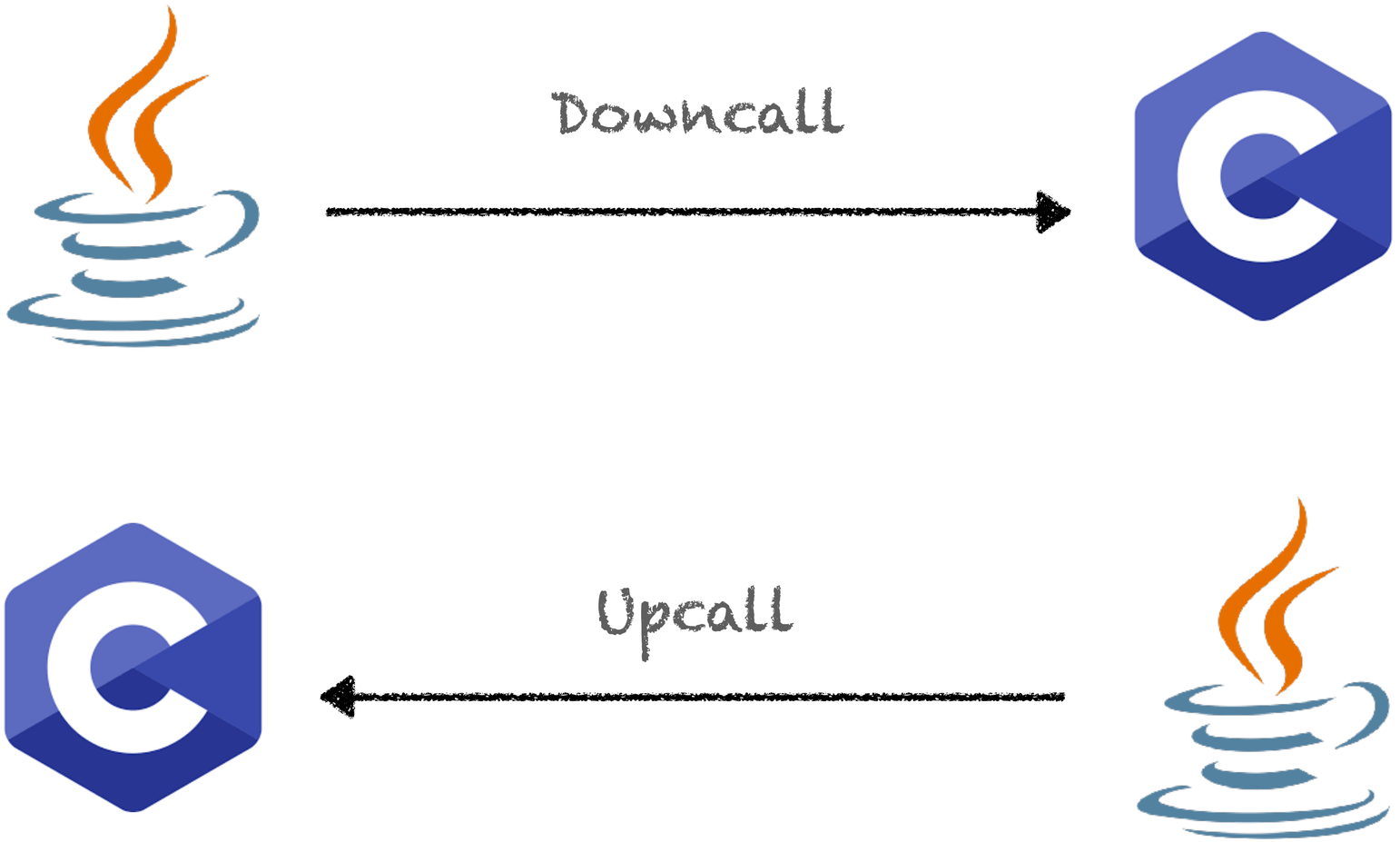
Native Linker

- Native Linker provides access to the libraries that adhere to the calling conventions of the platform in which the Java runtime is running.
- These libraries are referred to as "native" libraries

Locating the Address of the C Function

- To call a native method, you need a `downcall` method handle, which is a `MethodHandle` instance that points to a native function.
- This instance requires the native function's address
- To obtain this address, you use a *symbol lookup*, which enables you to retrieve the address of a symbol in one or more libraries

Upcalls and Downcalls


Downcall
Upcall

Function Descriptors

Function Descriptors

- A `FunctionDescriptor` describes the native function signature for FFM.
- It maps C types to Java `ValueLayouts`.
- Used by `Linker.downcallHandle(...)` to create a method handle.

Calling C Libraries

Calling a C Library

Calling a C library or anything that was compiled to a C library requires:

1. Obtaining an Instance to the Native Linker
2. Locating the Address of the C Function
3. Describing the C Function Signature
4. Creating a Downcall Handle to the C Function
5. Calling the C Function Directly from Java

Demo: Calling C Methods

- Let’s call a C Method with tests, or katas `testAddInts` and `multiplyDouble`
- We will show the C methods
- Show the libraries
- Show the arena establishment

Memory Segments

Memory Segments

- `MemorySegment` represents a contiguous region of memory, either on-heap (Arrays, FileChannels) or off-heap (which is our focus).
- Used to read/write structured native memory from Java.
- Allocated through an `Arena`
- Various way to get segments but typically done with `allocate`...

Finding Native Libraries

Finding Native Libraries

- Use `Linker.nativeLinker()` to access your platform's calling convention.
- Then, Use `linker.defaultLookup()` to access symbols from system libraries (e.g., `libc`)
- If you have calls you wish to make of your OS, you can do so from Java!

Demo: Calling C Methods

- Let’s call a C Method from the operating system
- We will use strlen to determine the length of a String

Structs

Structs

- Use `MemoryLayout.structLayout(...)` to model C structs in Java.
- Allocate and access struct memory with `MemorySegment` and `VarHandles`.
- Layout defines field order, size, alignment — just like C.
- Memory-safe, endian-aware, and portable

Demo: Structs

- Let's call a C Method `testDistanceSquared`
- This will use a struct

Pointers

Pointers

- In FFM, C pointers are modeled as `MemorySegment` values.
- When passing a `T*` (pointer to something) to a native function:
  - Allocate the memory using an `Arena`
  - Use `MemorySegment` to hold and pass the reference

Demo: Calling Methods with Pointers

- Let's call a C Method `testCopyString`
- This will use two references for String as we send it into a method

Receiving Pointers

- Native functions may return a pointer (e.g., `char*`, `void*`, `T*`)
- FFM represents returned pointers as `MemorySegment`
- Returned segments have a byte size of 0 — they are opaque handles
- You must `reinterpret` them to access the memory they point to

Demo: Calling Methods Returning Pointers

- Let’s call a C Method `testReturnPointerFromC`
- This will return a reference that will need to reinterpret
-

Callbacks

Calling with Callbacks

- FFM supports passing upcalls as C function pointers
- Use `Linker.upcallStub(...)` to convert a Java method → C function pointer
- Pass the stub to the native method as a `MemorySegment`

Restricted Methods?

Restricted Methods

- Restricted methods are powerful APIs in the FFM and JNI space
- They interact with native memory or the OS at a low level
- They are restricted because the can possibly:
  - Break memory safety
  - Crash the JVM
  - Circumvent boundaries

JEP 472: Prepare to Restrict the Use of JNI

Owner	Ron Pressler
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	24
Component	core-libs
Discussion	jdk dash dev at openjdk dot org
Relates to	JEP 454: Foreign Function & Memory API
Reviewed by	Alex Buckley, Dan Heidinga, Jorn Vernee, Mark Reinhold, Maurizio Cimadamore
Endorsed by	Alan Bateman
Created	2023/05/03 09:08
Updated	2025/02/24 23:21
Issue	8307341

Summary

Issue warnings about uses of the Java Native Interface (JNI) and adjust the Foreign Function & Memory (FFM) API to issue warnings in a consistent manner. All such warnings aim to prepare developers for a future release that ensures integrity by

jextract

`jextract`

- The `jextract` tool mechanically generates Java bindings from a native library header file.
- The bindings that this tool generates depend on the Foreign Function and Memory (FFM) API.
- With this tool, you don't have to create downcall and upcall handles for functions you want to invoke; the jextract tool generates code that does this for you.
- Obtain the tool from the following site: https://jdk.java.net/jextract/

Running `jextract`

```
jextract \
  --include-dir /path/to/mylib/include \
  --output src \
  --target-package org.jextract.mylib \
  --library mylib \
  /path/to/mylib/include/mylib.h
```

Demo: Running `jextract`

- Let's run `jextract` to get the signatures of C Libraries

What about other System Languages?

Demo: Creating a Rust Library

- Let’s create a library in Rust that will be called by Java FFM API

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext