

By the bye: JNI

JNI vs FFM - a (subjective) comparison

About me

Birgit Kratz

- Freelancing IT Consultant, SpringBoot Trainer
- Java-Backend
- More than 25 years experience
- Co-Organizer of Softwerkskammer in Düsseldorf and Köln (Cologne)
- Co-Organizer of SoCraTes-Conf Germany
- Email: mail@birgitkratz.de
- Mastodon: @birgitkratz@jvm.social
- Github: <https://github.com/bkratz>
- Web: <https://www.birgitkratz.de>



Agenda

A little background story

The old days - JNI in a nutshell

The new days - what is the new FFM API and how does it work?

Comparison by example - the sudoku solver project

(subjective) Assessment

First some questions

My background story

Two Problems

- How to allocate and manage off-heap Memory (aka Foreign Memory)
- How to call function in a Native Library (aka Foreign Functions)

Foreign Memory (the old way)

- by using direct ByteBuffers (`ByteBuffer.allocateDirect(int capacity)`)
 - restricted to max 2GB
 - only deallocated when object is garbage collected (not developer controlled)
- by using `sun.misc.Unsafe` API
 - which is fast and allows huge off-heap regions
 - gives developers too much control over (deallocation, dangling pointers, etc.)

Foreign Functions (the old way)

- by using JNI
- even the JVM uses JNI for access platform specific functionalities

JNI (Java Native Interface) in a Nutshell

- allows classes to declare 'native' methods
- interact with code written in other programming languages like C or C++
- implemented in a separate native shared library
- bridge between the bytecode running in our JVM and the native code
- part of Java since Java 1.1

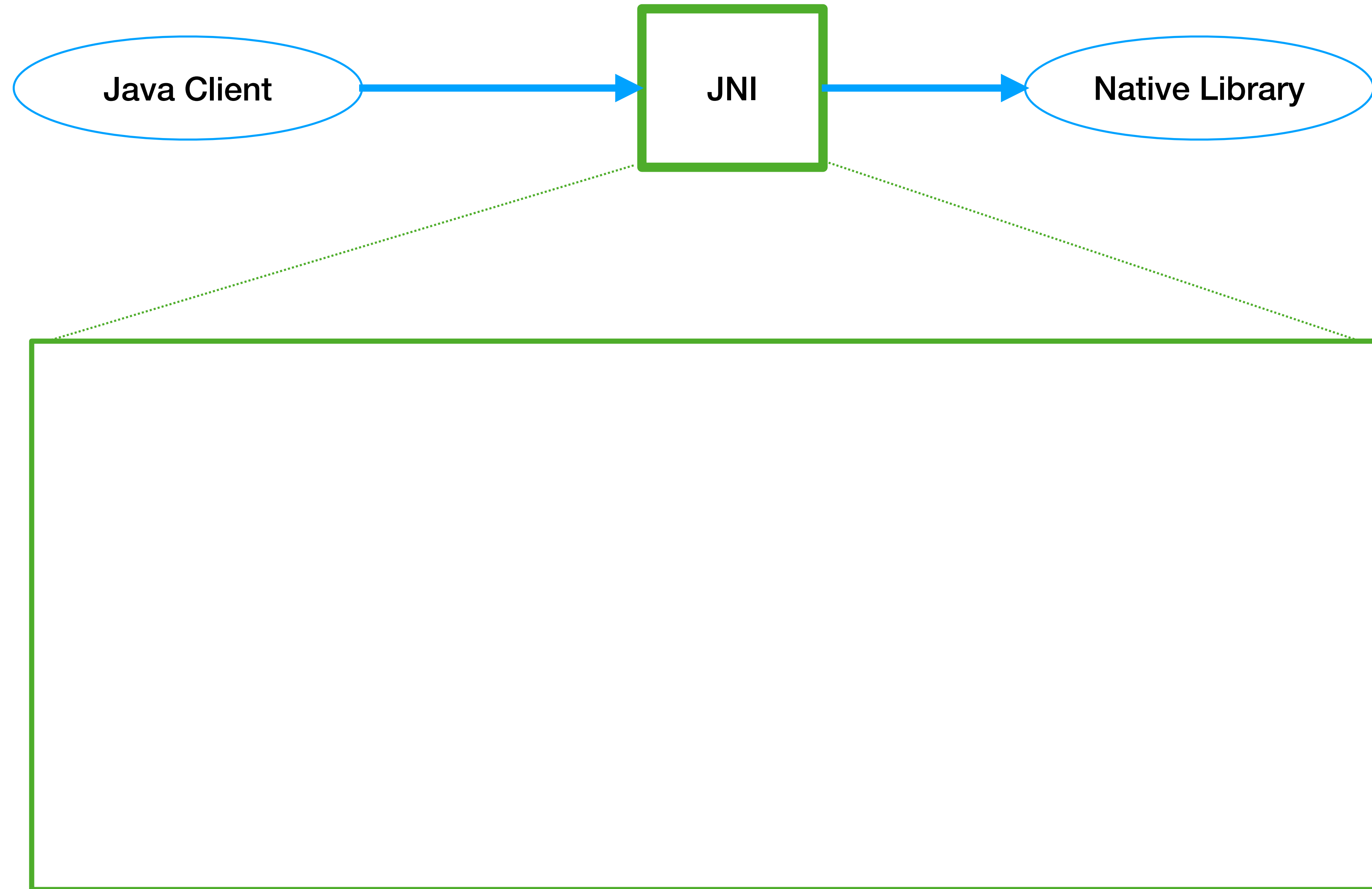
Java Classes containing native Methods

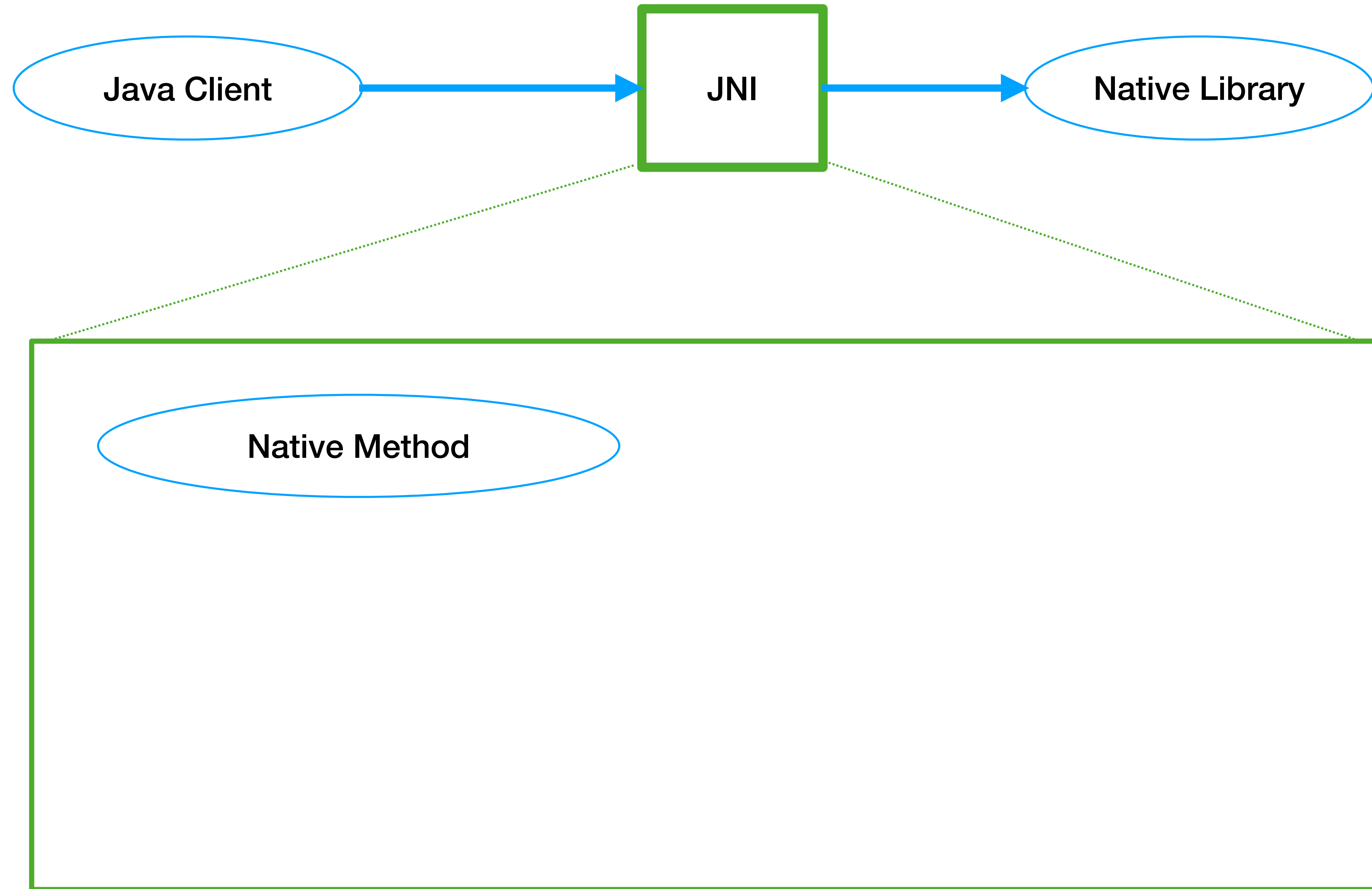
- `java.lang.System`: i.e. `arraycopy()`, `currentTimeMillis()`
- `java.io.FileDescriptor`
- `java.nio.DirectByteBuffer`
- `java.lang.Thread`: i.e. `start()`, `sleep()`
- `java.util.zip.Deflater`, `java.util.zip.Inflater`
- and others...

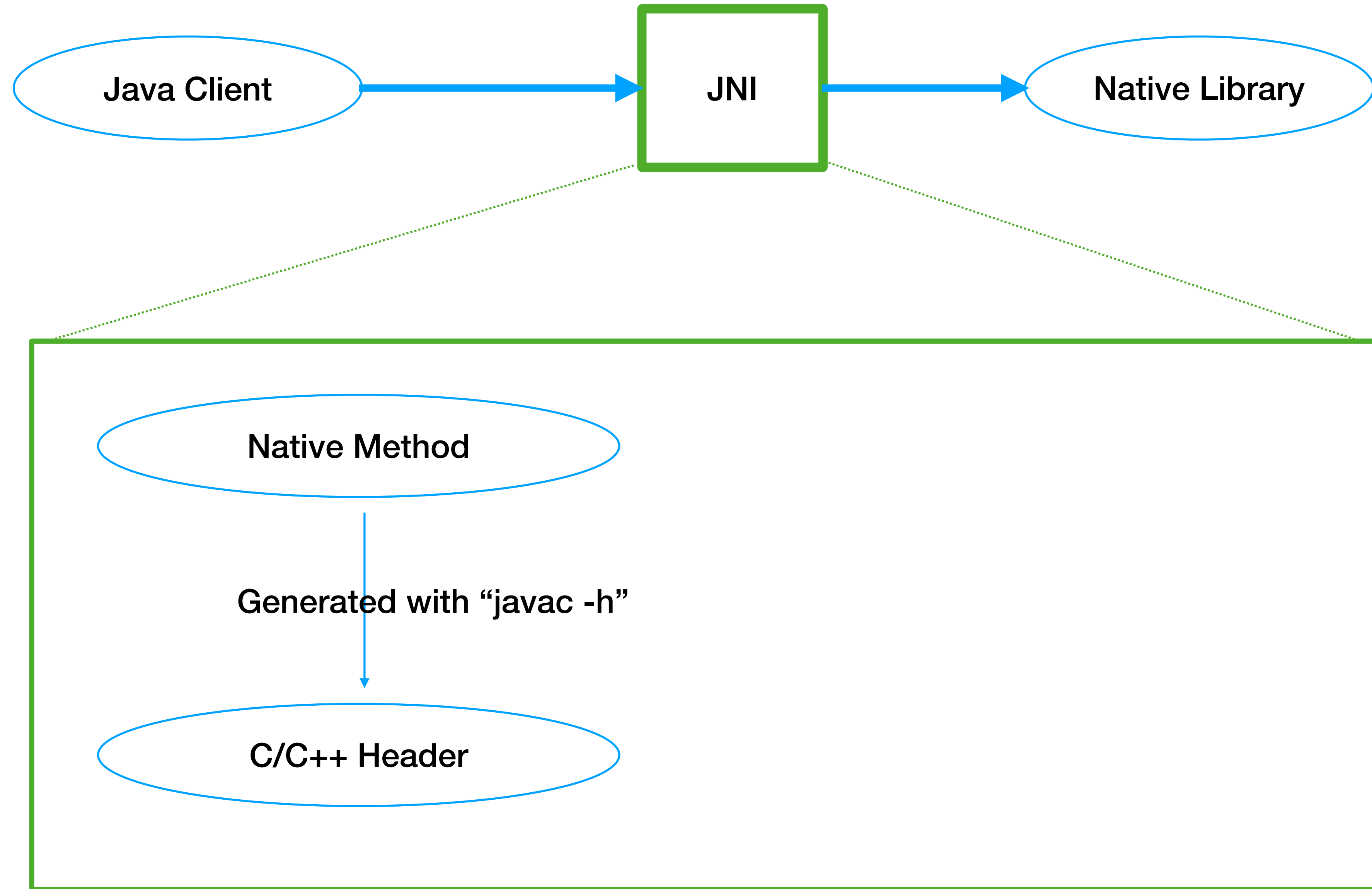
Sometimes you just have to go 'native'

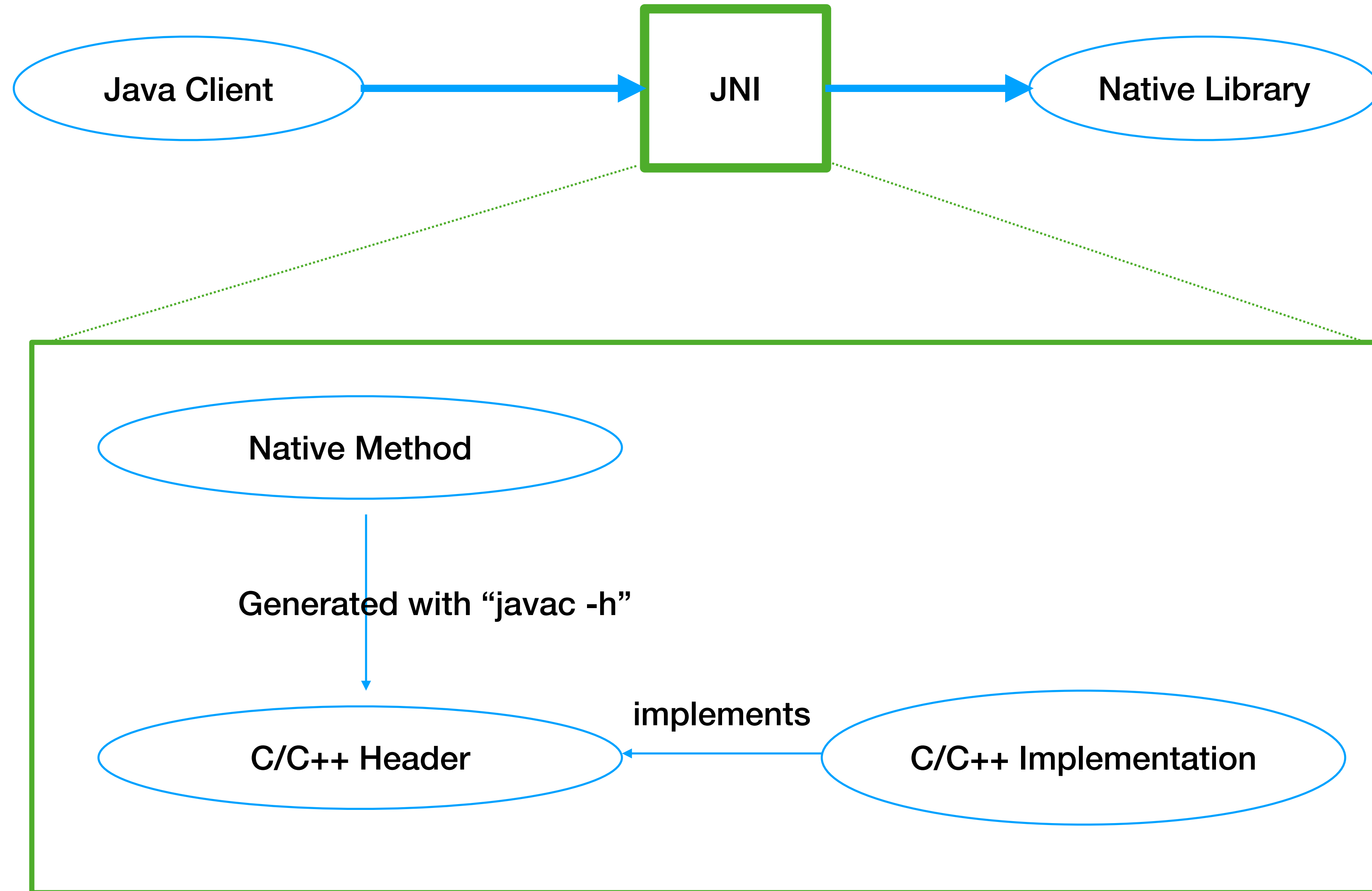
- Off-CPU computing (Cuda, OpenCL)
- Deep learning (Blas, cuBlas, cuDNN, Tensorflow, ...)
- Graphic processing (OpenGL, Vulkan, DirectX)
- others (OpenSSL, SQLite, V8, ...)

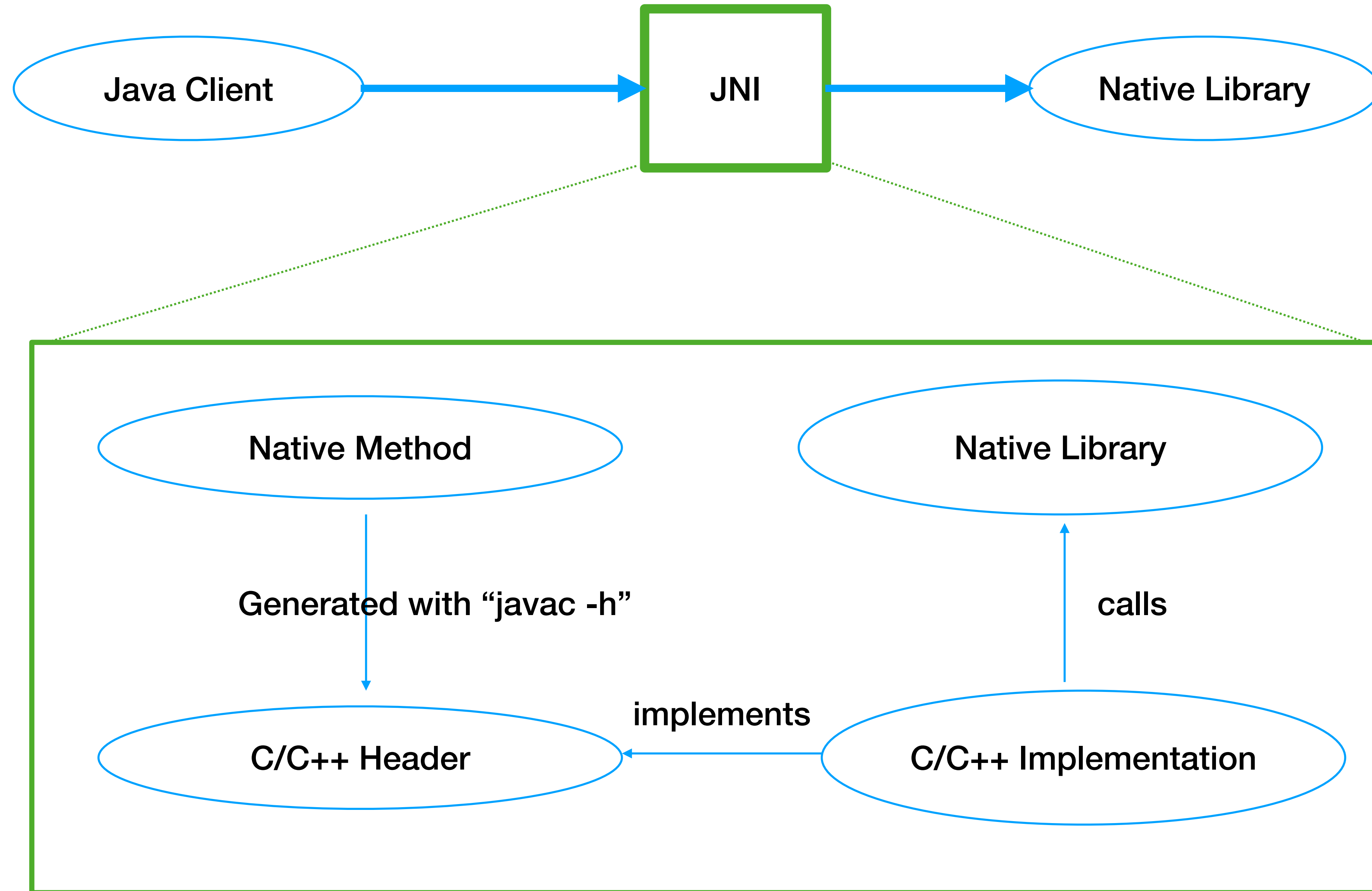
Quelle: <https://www.youtube.com/watch?v=cfxBrYud9KM>

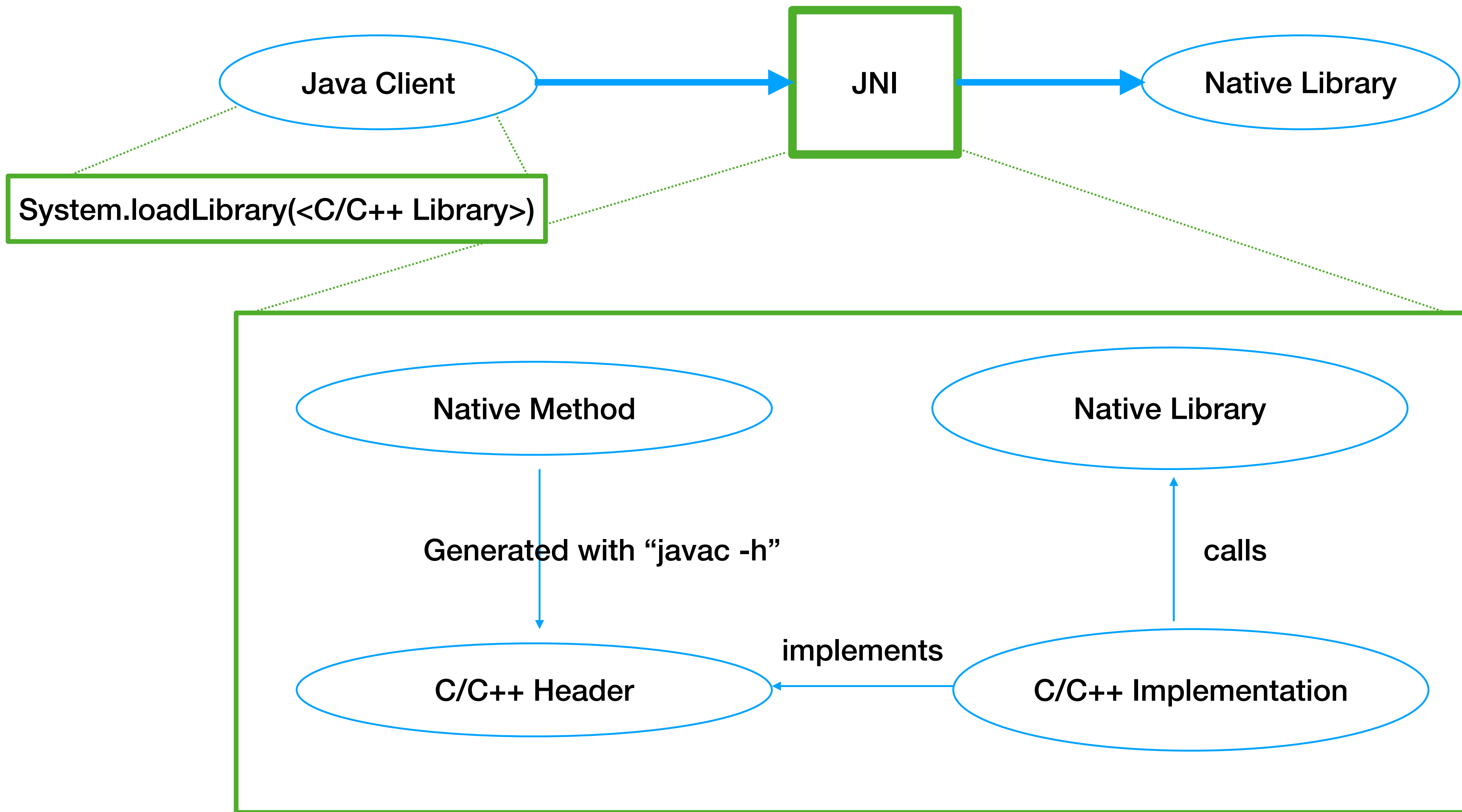


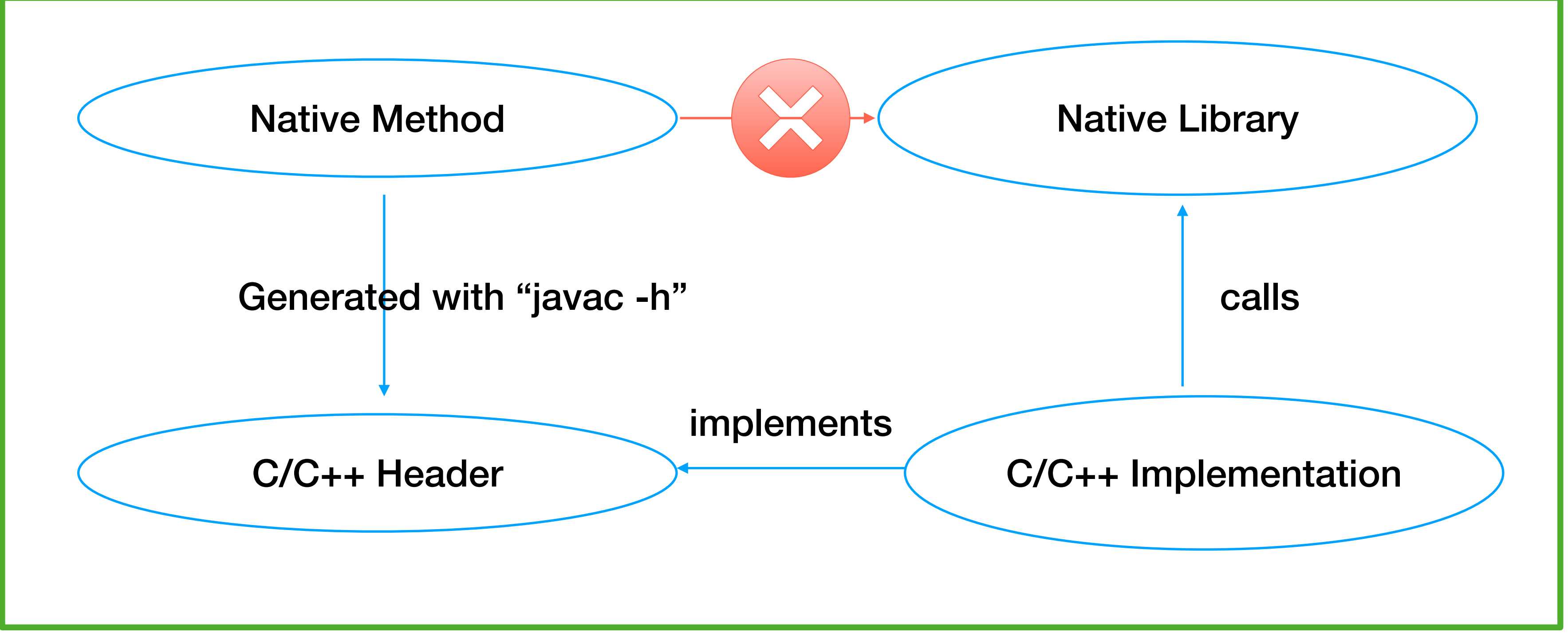
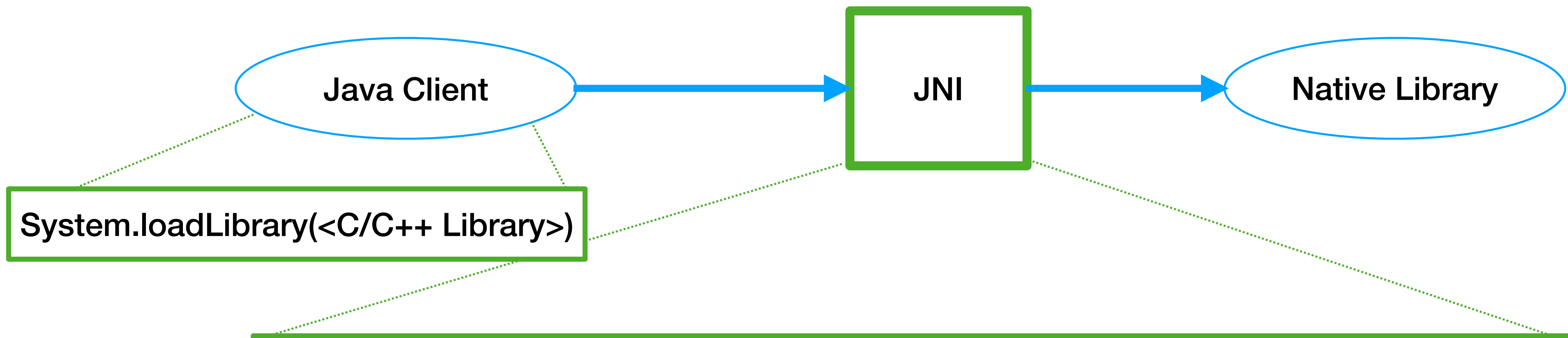












Problems

- Need to have good knowledge of native language such as C/C++ (as a Java programmer)
- Need to know memory management in these languages
- Potential memory leaks -> allocated memory is never freed
- Memory freed too early -> use-after-free

Code example

FFM API - Java Foreign Function & Memory API

- JEP 454
- Part of Project Panama
- Final since Java 22

Managing Memory in Java (the new way)

- **Arena**

- models the lifecycle of Memory Segments
- it's closable
- deterministic deallocation of Memory Segments
- no out-of-bounds access
- no use-after-free access

Arena Characteristics

Kind	Bounded lifetime	Explicitly closeable	Accessible from multiple threads
Global	No	No	Yes
Automatic	Yes	No	Yes
Confined	Yes	Yes	No
Shared	Yes	Yes	Yes

Quelle: <https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/foreign/Arena.html>

Managing Memory in Java (the new way)

- **Memory Segment**

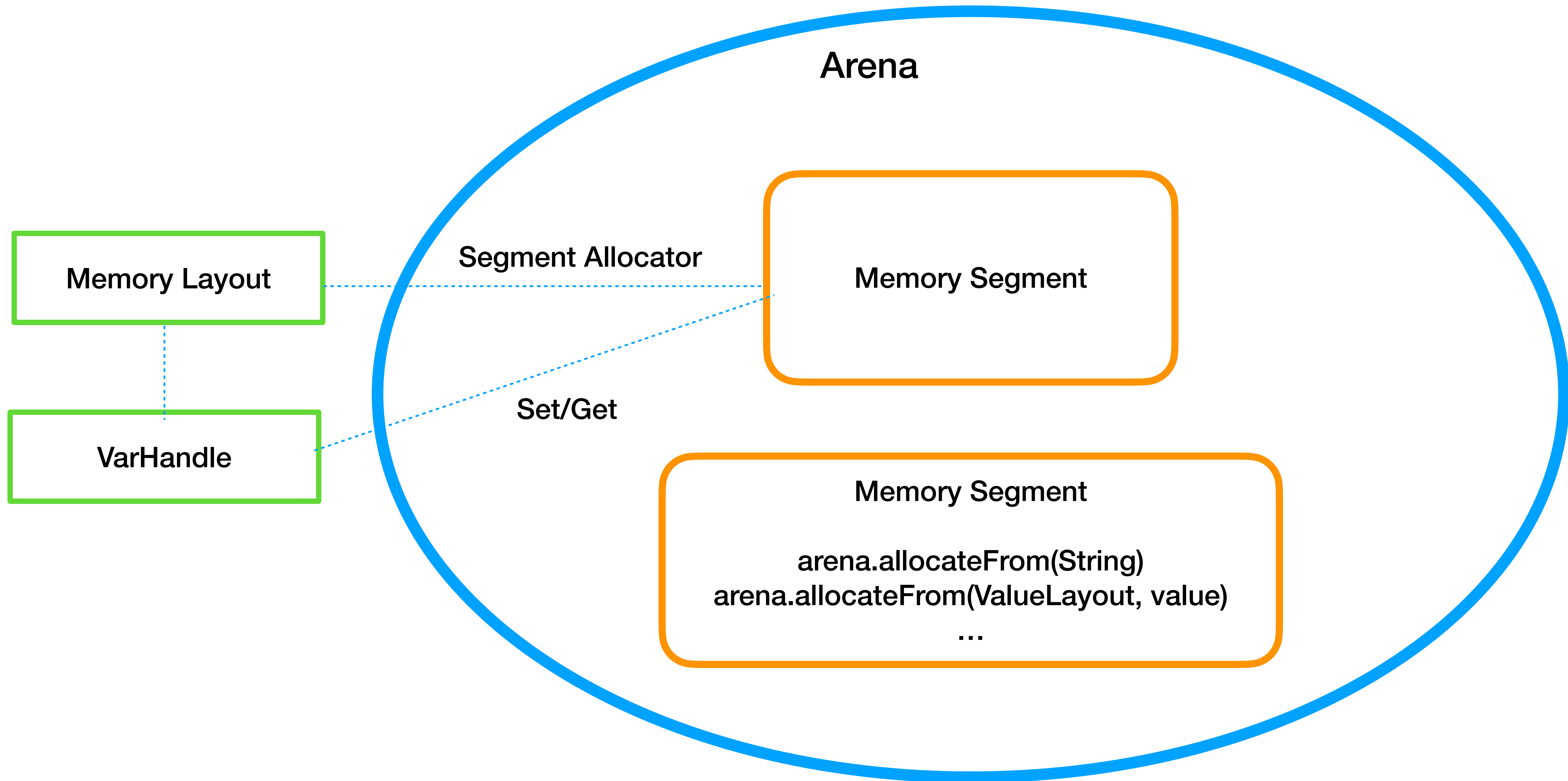
- can represent on-heap or off-heap memory regions
- off-heap Memory Segments belongs to an Arena
- all Memory Segments of an Arena share the same lifetime
- cannot be used after being freed
- when an Arena is closed, all of it's Memory Segments are automatically invalidated

Managing Memory in Java (the new way)

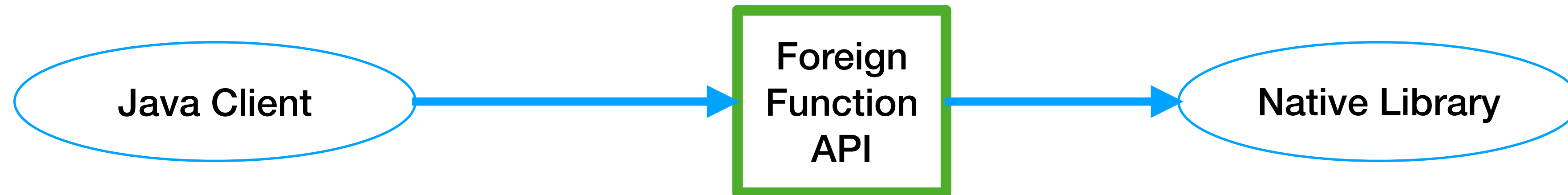
- **Memory Layout**

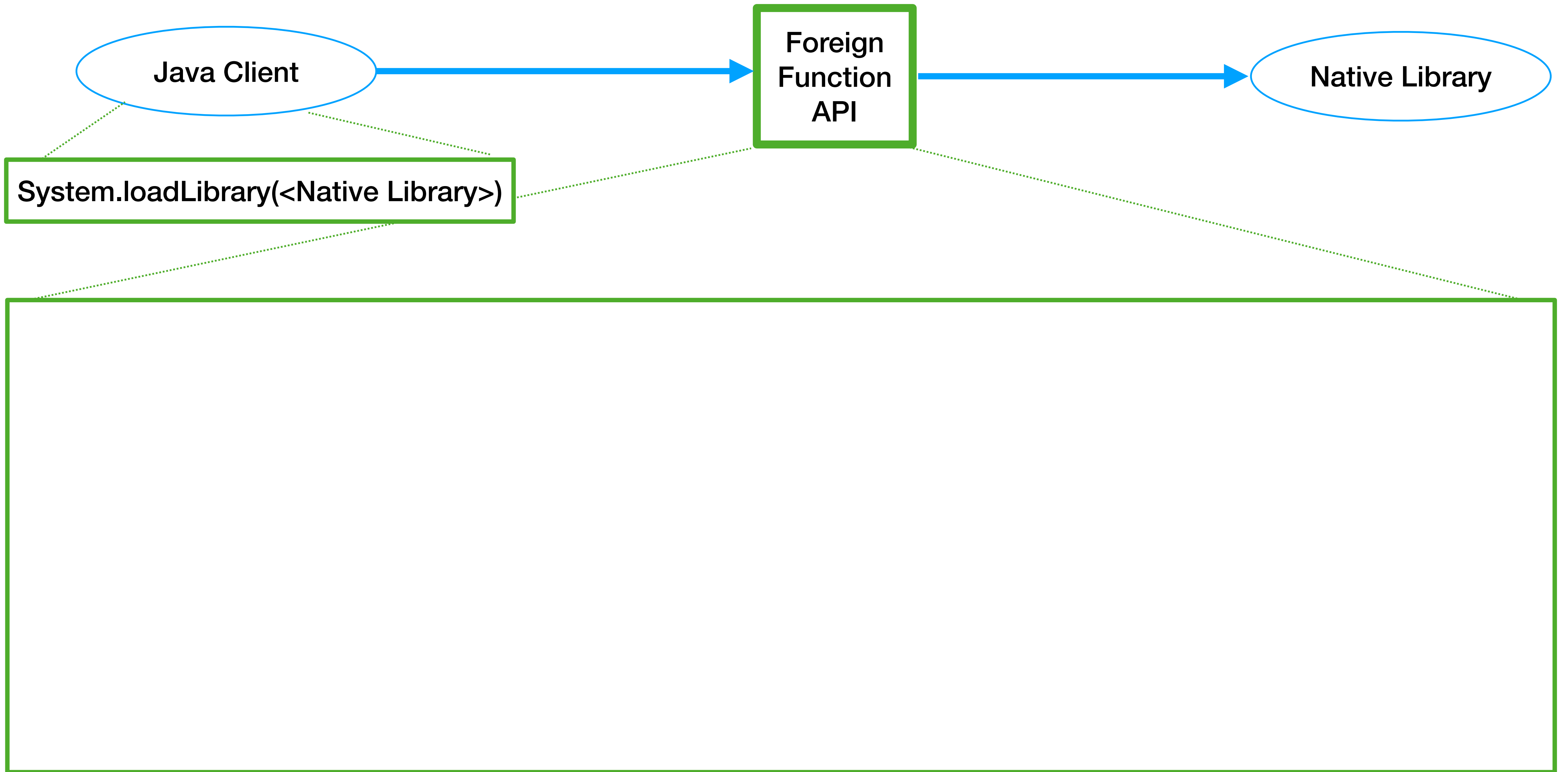
- programmatically describe contents of a memory region
- can be queried for size, alignment and access expressions
- has subtypes like:
 - Group Layout (i.e. for describing structs)
 - Sequence Layout (i.e. for lists)
 - Value Layout (i.e. for pointers, boolean, byte, char, int, long, etc.)

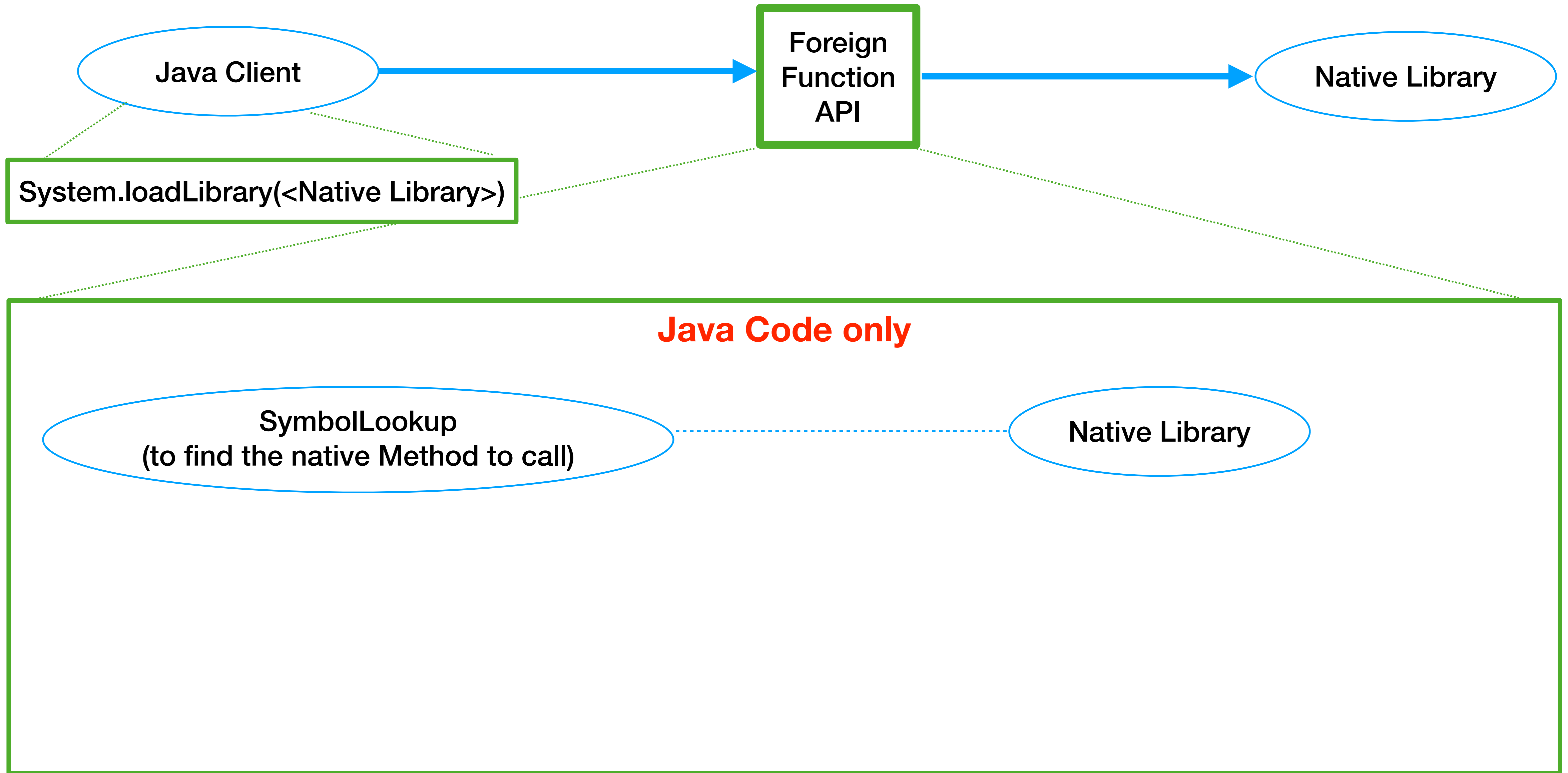
- **VarHandle** - to simplify offset handling

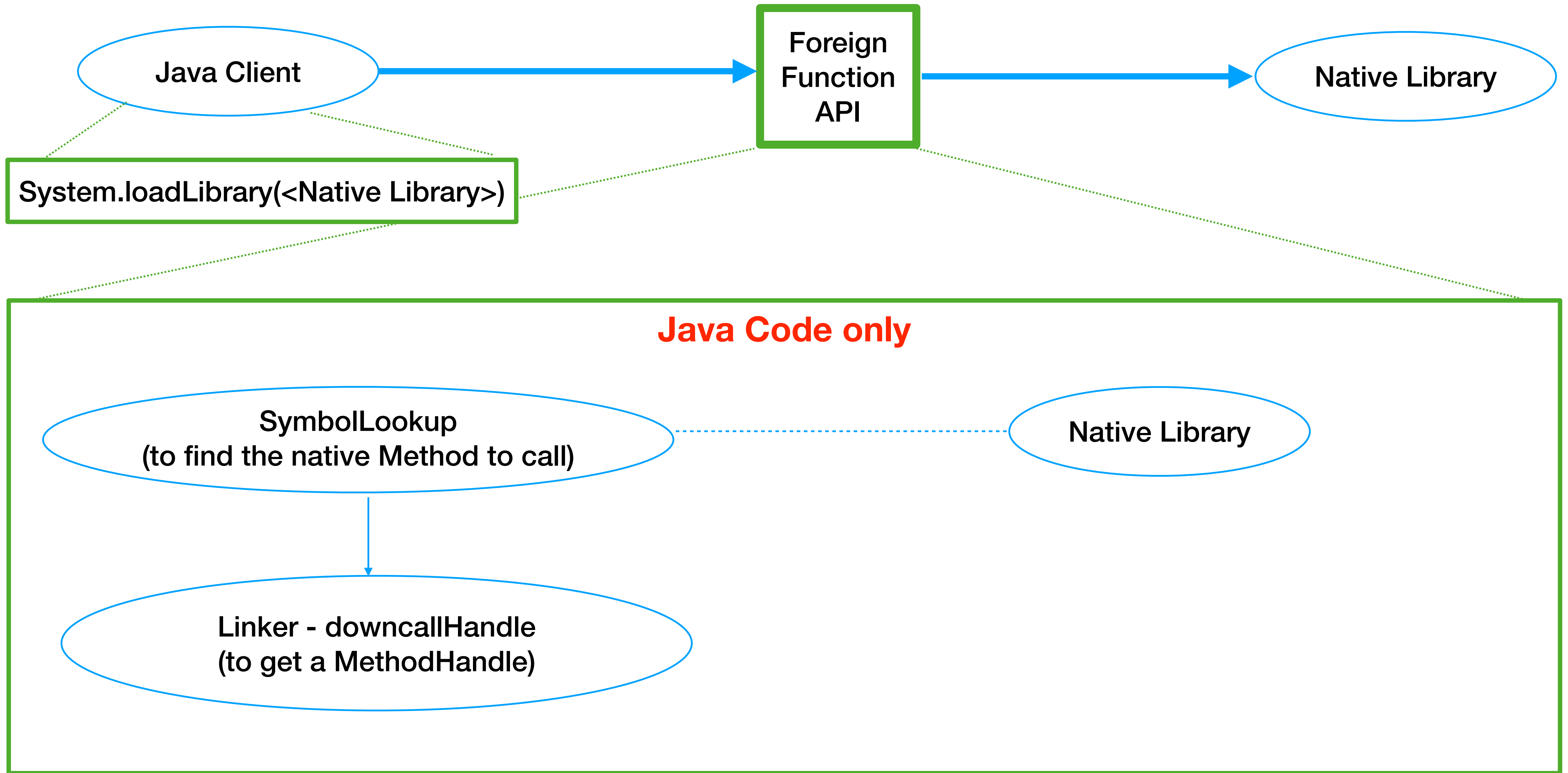


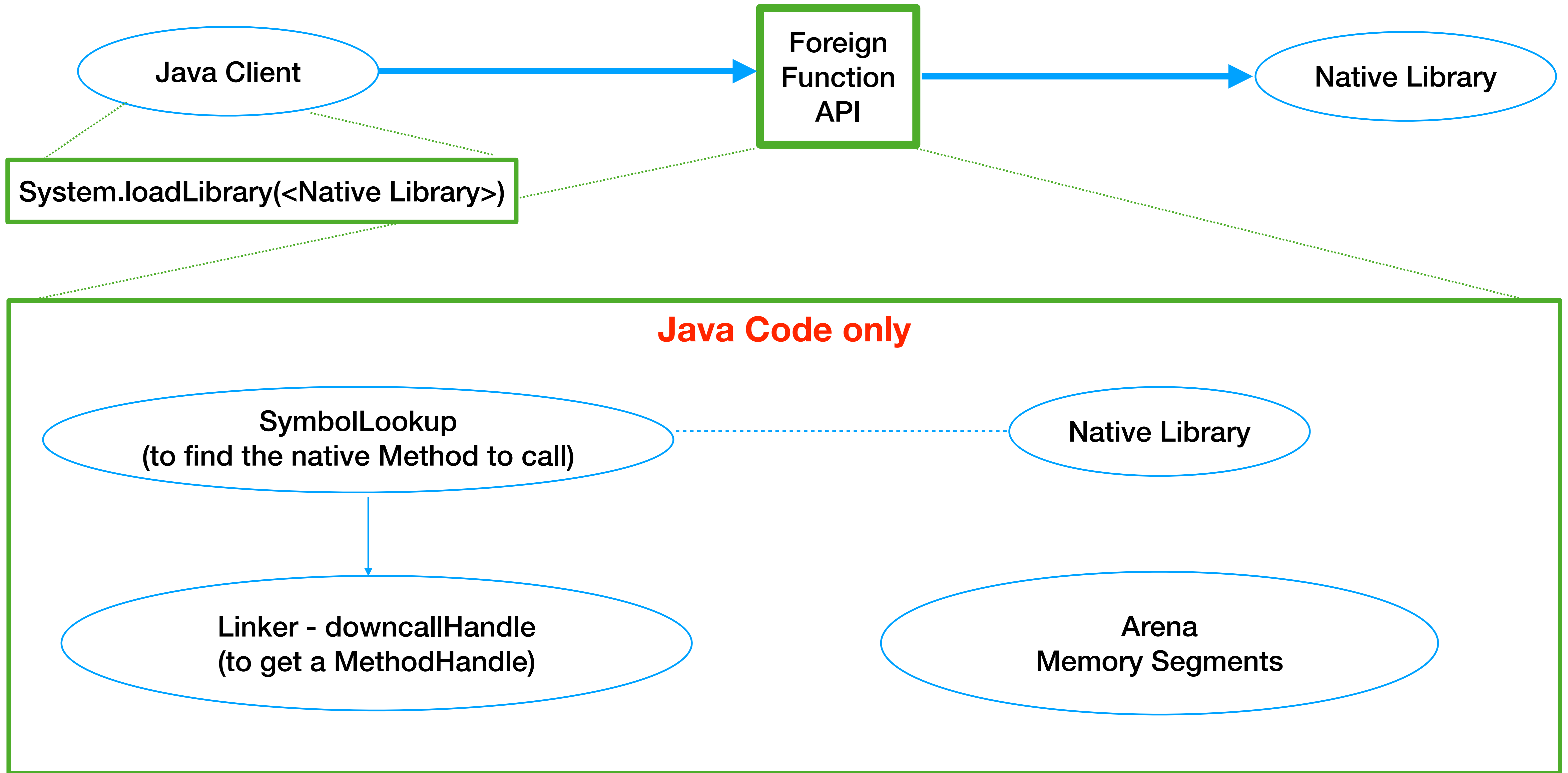
Direct Native Function call with Foreign Function API

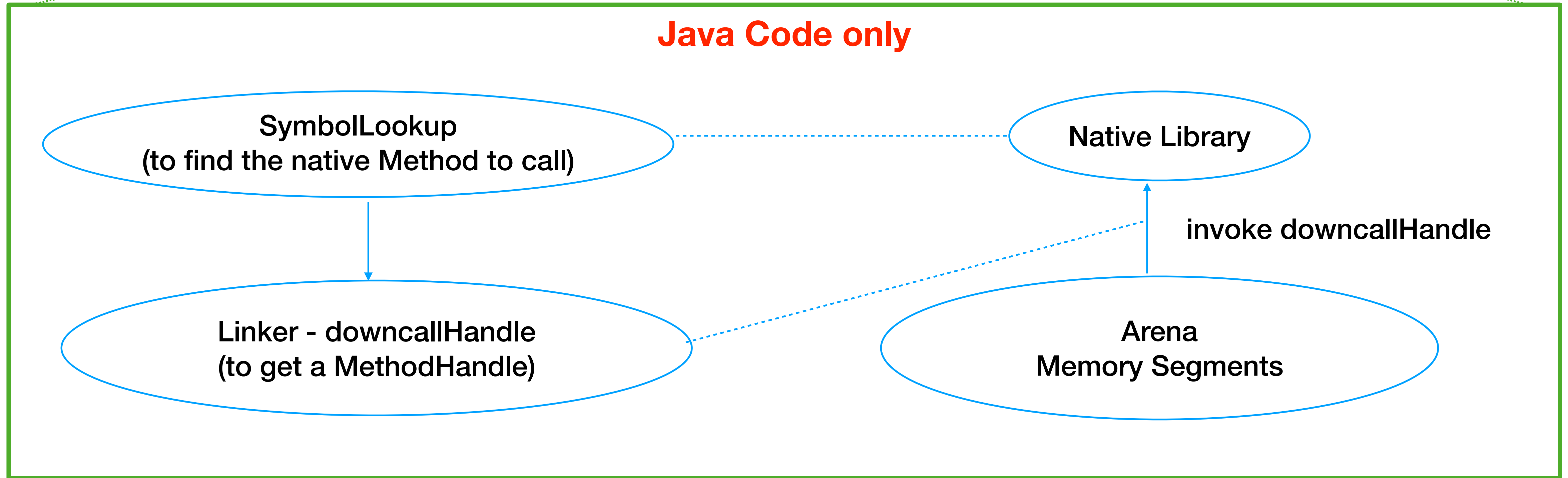
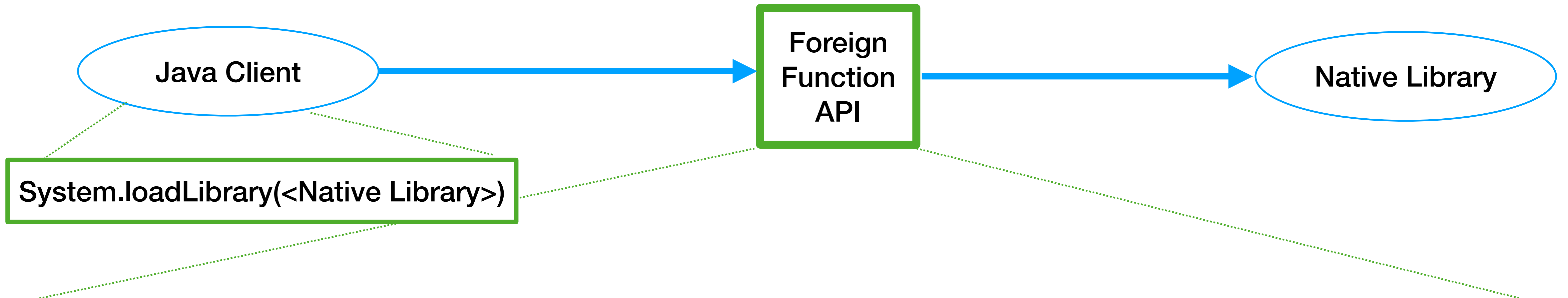












Code example

jextract

- is a tool which mechanically generates Java bindings from a native library headers
- leverages the clang C API in order to parse the headers associated with a given native library, and the generated Java bindings build upon the Foreign Function & Memory API
- was originally developed in the context of Project Panama

Code example

My subjective Assessment

- Both approaches are not platform independent, the native library has to be available for each platform the Java program is running on
- FFM is only available with Java 22 or later, or as a preview-version also with version prior to Java 22
- With FFM I can completely stay in the Java development environment
- No definition of 'native' methods necessary
- Significantly less code compared to JNI, especially when using 'jextract'
- The programming with MemorySegments and Arenas needs some getting used to
- Do not need to worry about memory-leaks or use-after-freed anymore

Some more inspiration

- 1BR - The One Billion Row Challenge, by Gunnar Morling
- reading and processing a REEEAAALLLYYY big file with Java as quick as possible
- <https://www.morling.dev/blog/one-billion-row-challenge/>
- Github Repository: <https://github.com/gunnarmorling/1brc>
- Example solution using Arena and MemorySegment: https://github.com/gunnarmorling/1brc/blob/main/src/main/java/dev/morling/onebrc/CalculateAverage_artsiomkorzun.java

Questions?

Thank you

Slides: https://www.birgitkratz.de/uploads/OCX_2024_ByTheByeJNI.pdf

Sample code repositories:

<https://github.com/bkratz/SudokuSolverNative>

<https://github.com/bkratz/SudokuSolverCPP>

<https://github.com/bkratz/SudokuSolverJNI>

<https://github.com/bkratz/SudokuSolverFFM>

- Email: mail@birgitkratz.de
- Mastodon: [@birgitkratz@jvm.social](https://jvm.social/@birgitkratz)
- Github: <https://github.com/bkratz>
- Web: <https://www.birgitkratz.de>

