Foreign Function and 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.

Note:

This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Language and VM Features.

For background information about the Foreign Function and Memory API, see JEP 434.

The FFM API is contained in the package java.lang.foreign.

Topics

Off-Heap Memory

Off-heap data is data stored in memory outside the Java runtime, which is also known as the JVM's *heap*. Off-heap data is stored in *off-heap memory*, which is represented by a MemorySegment object. Unlike heap memory, off-heap memory is not subject to garbage collection when no longer needed. You can control how and when off-heap memory is allocated and deallocated.

To invoke a function or method from a different language such as C from a Java application, its arguments must be in off-heap memory.

The following example allocates off-heap memory, stores a Java String, and then prints the contents of the off-heap memory:

```
static void allocateCharArray(String s) {
    try (Arena arena = Arena.openConfined()) {


        // Allocate off-heap memory
        MemorySegment nativeText = arena.allocateUtf8String(s);

        // Access off-heap memory
        for (int i = 0; i < s.length(); i++ ) {
          System.out.print((char)nativeText.get(ValueLayout.JAVA_BYTE, i));
        }

      } // Off-heap memory is deallocated
    }
```

Topics

Segment Scopes and Arenas

All memory segments are associated with a *segment scope*, which controls which threads can access a memory segment and when. A segment scope is represented by a SegmentScope object. The allocateCharArray example uses an *arena scope*.
An *arena* controls the lifecycle of memory segments. When an arena is closed, then all memory segments associated with its scope are invalidated and their backing memory regions are deallocated. When the arena is closed through the try-with-resources statement, then the arena scope is no longer alive and its memory segements are invalidated. In addition, the memory regions backing the segments are deallocated atomically, which means that the deallocation happens completely or doesn't happen at all. Consequently, if you try to access a memory segment associated with an arena scope that's closed, you'll get an IllegalStateException, which the following example demonstrates:

```
    static void allocateCharArrayError(String s) {

      MemorySegment nativeText;
      try (Arena arena = Arena.openConfined()) {

        // Allocate off-heap memory
        nativeText = arena.allocateUtf8String(s);
      }
      for (int i = 0; i < s.length(); i++ ) {
        // Exception in thread "main" java.lang.IllegalStateException: Already closed
        System.out.print((char)nativeText.get(ValueLayout.JAVA_BYTE, i));
      }
    }
```

There are two kinds of arenas, *confined* and *shared*:

- A confined arena, which is created with Arena::openConfined, has an owner thread. This 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.
- A shared arena, which is created with Arena::openShared, has no owner thread. Multiple threads may access the memory segments allocated in a shared arena. In addition, any thread may close a shared arena, and the closure is guaranteed to be safe and atomic.

Allocating Off-Heap Memory

The Arena interface implements the SegmentAllocator interface, which contains methods that both allocate off-heap memory and copy Java data into it.
The allocateCharArray example calls the method SegmentAllocator::allocateUtf8String, which converts a string into a UTF-8 encoded, null-terminated C string and then stores the result into a memory segment.

```
static void allocateCharArray(String s) {
    try (Arena arena = Arena.openConfined()) {

        // Allocate off-heap memory
        MemorySegment nativeText = arena.allocateUtf8String(s);
        // ...
```

See Memory Layouts and Structured Access for information about allocating and accessing more complicated native data types such as C structures.

Accessing Off-Heap Memory

The following code prints the characters stored in the MemorySegment named nativeText:

```
        // Access off-heap memory
        for (int i = 0; i < s.length(); i++ ) {
          System.out.print((char)nativeText.get(ValueLayout.JAVA_BYTE, i));
        }
```

The MemorySegment contains various access methods that enable you to read from or write to them. Each access method takes as an argument a *value layout*, which models the memory layout associated with values of basic data types such as primitives. A value layout encodes the size, the endianness or byte order, the bit alignment of the piece of memory to be accessed, and the Java type to be used for the access operation.
For example, MemoryLayout.get(ValueLayout.OfByte,long) takes as an argument ValueLayout.JAVA_BYTE. This value layout has the following characteristics:

- The same size as a Java byte
- Bit alignment set to 8: This means that the memory layout is stored at a memory address that's a multiple of 8 bits.
- Byte order set to ByteOrder.nativeOrder(): A system can order the bytes of a multibyte value from most significant to least significant (big-endian) or from least significant to most significant (little-endian).

Calling a C Library Function with the Foreign Function and Memory API

The following example calls strlen with the Foreign Function and Memory API:

```
 static long invokeStrlen(String s) throws Throwable {

   try (Arena arena = Arena.openConfined()) {

     // 1. Allocate off-heap memory, and
     // 2. Dereference off-heap memory
     MemorySegment nativeString = arena.allocateUtf8String(s);

     // 3. Link and call C function

     // 3a. Obtain an instance of the native linker
     Linker linker = Linker.nativeLinker();

     // 3b. Locate the address of the C function
     SymbolLookup libc = linker.defaultLookup();
     MemorySegment strlen_addr = libc.find("strlen").get();

     // 3c. Create a description of the native function signature
     FunctionDescriptor strlen_sig =
       FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS);

     // 3d. Create a downcall handle for the C function
     MethodHandle strlen = linker.downcallHandle(strlen_addr, strlen_sig);

     // 3e. Call the C function directly from Java
     return (long)strlen.invokeExact(nativeString);
   }
 }
```

The following is the declaration of the strlen C standard library function:

```
size_t strlen(const char *s);
```

It takes one argument, a string, and returns the length of the string. To call this function from a Java application, you would follow these steps:

1. Allocate *off-heap memory*, which is memory outside the Java runtime, for the strlen function's argument.
2. Store the Java string in the off-heap memory that you allocated.

   The invokeStrlen example performs the previous step and this step with the following statement:

   ```
   MemorySegment nativeString = arena.allocateUtf8String(s);
   ```
3. Build and then call a method handle that points to the strlen function. The topics in this section show you how to do this.

Topics

Obtaining an Instance of the Native Linker

A *linker* provides access to foreign functions from Java code and access to Java code from foreign functions. The *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.

```
Linker linker = Linker.nativeLinker();
```

Locating the Address of the C Function

To call a native method such as strlen, you need a *downcall method handle*, which is a MethodHandle instance that points to a native function. This instance requires the address of the native function. The following statements obtain the address of the strlen function:

```
try (Arena arena = Arena.openConfined()) {
    SymbolLookup libc = SymbolLookup.libraryLookup("libc.so.6", arena.scope());
    MemorySegment strlen_addr = libc.find("strlen").get();
    // ...
}
```

SymbolLookup::libraryLookup(String, SegmentScope) creates a library lookup, which locates all the symbols in a user-specified native library. It loads the native library and

associates it with segment scope. In this example, libc.so.6 is the file name of the C standard library for many Linux systems.
Because strlen is part of the C standard library, you can use instead the native linker's *default lookup*. This is a symbol lookup for symbols in a set of commonly used libraries (including the C standard library). This means that you don't have to specify a system-dependent library file name:

```
// 3a. Obtain an instance of the native linker
Linker linker = Linker.nativeLinker();

// 3b. Locate the address of the C function
SymbolLookup stdLib = linker.defaultLookup();
MemorySegment strlen_addr = stdLib.find("strlen").get();
```

Note:

Call SymbolLookup.loaderLookup() to find symbols in libraries that are loaded with System.loadLibrary(String).

Describing the C Function Signature

A downcall method handle also requires a description of the native function's signature, which is represented by a FunctionDescriptor instance. A *function descriptor* describes the layouts of the native function arguments and its return value, if any.
Each layout in a function descriptor maps to a Java type, which is the type that should be used when invoking the resulting downcall method handle. Most value layouts map to a Java primitive type. For example, ValueLayout.JAVA_INT maps to an int value. However, ValueLayout.ADDRESS maps to a pointer.
Composite types such as struct and union types are modeled with the GroupLayout interface, which is a supertype of StructLayout and UnionLayout. See Memory Layouts and Structured Access for an example of how to initialize and access a C structure.
The following creates a function descriptor for the strlen function:

```
// 3c. Create a description of the C function signature
FunctionDescriptor strlen_sig =
   FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS);
```

The first argument of the FunctionDescriptor::of method is the layout of the native function's return value. Native primitive types are modeled using value layouts whose size matches that of such types. This means that a function descriptor is platform-specific. For example, size_t has a layout of JAVA_LONG on 64-bit or x64 platforms but a layout of JAVA_INT on 32-bit or x86 platforms.
The subsequent arguments of FunctionDescriptor::of are the layouts of the native function's arguments. In this example, there's only one subsequent argument,

a ValueLayout.ADDRESS. This represents the only argument for strlen, a pointer to a string.

Creating the Downcall Handle for the C Function

The following statement creates a downcall method handle for the strlen function with its address and function descriptor.

```
    // 3d. Create a downcall handle for the C function
    MethodHandle strlen = linker.downcallHandle(strlen_addr, strlen_sig);
```

Calling the C Function Directly from Java

The following statement calls the strlen function with a memory segment that contains the function's argument:

```
    // 3e. Call the C function directly from Java
    return (long)strlen.invokeExact(nativeString);
```

You need to cast a method handle invocation with the expected return type; in this case, it's long.

Upcalls: Passing Java Code as a Function Pointer to a Foreign Function

An *upcall* is a call from native code back to Java code. An *upcall stub* enables you to pass Java code as a function pointer to a foreign function.
Consider the standard C library function qsort, which sorts the elements of an array:

```
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

It takes four arguments:

- base: Pointer to first element of the array to be sorted
- nbemb: Number of elements in the array
- size: Size, in bytes, of each element in the array
- compar: Pointer to function that compares two elements

The following example calls the qsort function to sort an int array. However, this method requires a pointer to a function that compares two array elements. The example defines a comparison method (Qsort::qsortCompare), creates a method handle to represent this comparison method, and then creates a function pointer from this method handle.

```
import java.lang.foreign.*;
import java.lang.invoke.*;
import java.lang.foreign.ValueLayout.*;
```

```
public class InvokeQsort {

  class Qsort {
    static int qsortCompare(MemorySegment addr1, MemorySegment addr2) {
      return addr1.get(ValueLayout.JAVA_INT, 0) - addr2.get(ValueLayout.JAVA_INT, 0);
    }
  }

  // Obtain instance of native linker
  final static Linker linker = Linker.nativeLinker();

  // Create downcall handle for qsort
  final static MethodHandle qsort = linker.downcallHandle(
    linker.defaultLookup().find("qsort").get(),
    FunctionDescriptor.ofVoid(
      ValueLayout.ADDRESS,
      ValueLayout.JAVA_LONG,
      ValueLayout.JAVA_LONG,
      ValueLayout.ADDRESS));

  // A Java description of a C function implemented by a Java method
  final static FunctionDescriptor qsortCompareDesc = FunctionDescriptor.of(
    ValueLayout.JAVA_INT,
    ValueLayout.ADDRESS.asUnbounded(),
    ValueLayout.ADDRESS.asUnbounded());

  // Create method handle for qsortCompare
  final static MethodHandle compareHandle;
  static {
    try {
      compareHandle = MethodHandles.lookup().findStatic(
        InvokeQsort.Qsort.class,
        "qsortCompare",
        qsortCompareDesc.toMethodType());
    } catch (Exception e) {
      throw new AssertionError(
        "Problem creating method handle compareHandle", e);
    }
  }

  static int[] qsortTest(int[] unsortedArray) throws Throwable {
        int[] sorted = null;

        try (Arena arena = Arena.openConfined()) {

            // Allocate off-heap memory and store unsortedArray in it
            MemorySegment array = arena.allocateArray(
                                ValueLayout.JAVA_INT,
                                unsortedArray);

            // Create function pointer for qsortCompare
            MemorySegment compareFunc = linker.upcallStub(
                compareHandle,
                qsortCompareDesc,
                arena.scope());

            // Call qsort
            qsort.invoke(array, (long)unsortedArray.length,
                ValueLayout.JAVA_INT.byteSize(), compareFunc);

            // Access off-heap memory
            sorted = array.toArray(ValueLayout.JAVA_INT);
        }
        return sorted;
    }

    public static void main(String[] args) {
        try {
            int[] sortedArray = InvokeQsort.qsortTest(
                new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 });
            for (int num : sortedArray) {
                System.out.print(num + " ");
            }
            System.out.println();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

The following class defines the Java method that compares two elements, in this case two `int` values:

```
    class Qsort {
        static int qsortCompare(MemorySegment addr1, MemorySegment addr2) {
            return addr1.get(ValueLayout.JAVA_INT, 0) - addr2.get(ValueLayout.JAVA_INT, 0);
    }
  }
```
In this method, the `int` values are represented by `MemorySegment` objects. A *memory segment* provides access to a contiguous region of memory. To obtain a value from a memory segment, call one of its `get` methods. This example calls the `get(ValueLayout.OfInt, long)`, where the second argument is the offset in bytes relative to the memory address's location. The second argument is 0 because the memory segments in this example store only one value.

The following statements create a downcall method handle for the `qsort` function:

```
  // Obtain instance of native linker
  final static Linker linker = Linker.nativeLinker();

  // Create downcall handle for qsort
  final static MethodHandle qsort = linker.downcallHandle(
    linker.defaultLookup().find("qsort").get(),
    FunctionDescriptor.ofVoid(
      ValueLayout.ADDRESS,
      ValueLayout.JAVA_LONG,
      ValueLayout.JAVA_LONG,
      ValueLayout.ADDRESS));
```
The following statement creates a method handle to represent the comparison method `Qsort::qsortCompare`:

```
  // A Java description of a C function implemented by a Java method
  final static FunctionDescriptor qsortCompareDesc = FunctionDescriptor.of(
    ValueLayout.JAVA_INT,
    ValueLayout.ADDRESS.asUnbounded(),
    ValueLayout.ADDRESS.asUnbounded());
  // Create method handle for qsortCompare
  final static MethodHandle compareHandle;
  static {
    try {
      compareHandle = MethodHandles.lookup().findStatic(
        InvokeQsort.Qsort.class,
        "qsortCompare",
        qsortCompareDesc.toMethodType());
    } catch (Exception e) {
      throw new AssertionError(
        "Problem creating method handle compareHandle", e);
    }
  }
```
Note:

ValueLayout.OfAddress::asUnbounded is a *restricted method*, which, if used incorrectly, might crash the JVM or silently result in memory corruption. See Restricted Methods for more information.
The MethodHandles.Lookup::findStatic method creates a method handle for a static method. It takes three arguments:

- The method's class
- The method's name
- The method's type: The first argument of MethodType::methodType is the method's return value's type. The rest are the types of the method's arguments.

The following statements allocate off-heap memory, then store the int array to be sorted in it:

```
try (Arena arena = Arena.openConfined()) {

  // Allocate off-heap memory and store unsortedArray in it
  MemorySegment array = arena.allocateArray(
                   ValueLayout.JAVA_INT,
                   unsortedArray);
```

The following statement creates a function pointer from the method handle compareHandle:

```
  // Create function pointer for qsortCompare
  MemorySegment compareFunc = linker.upcallStub(
    compareHandle,
    qsortCompareDesc,
    arena.scope());
```

The Linker::upcallStub method takes three arguments:

- The method handle from which to create a function pointer
- The function pointer's function descriptor; in this example, the arguments for FunctionDescriptor.of correspond to the return value type and arguments of Qsort::qsortCompare
- The scope to associate with the function pointer

The following statement calls the qsort function:

```
  // Call qsort
  qsort.invoke(array, (long)unsortedArray.length,
    ValueLayout.JAVA_INT.byteSize(), compareFunc);
```

In this example, the arguments of MethodHandle::invoke correspond to those of the standard C library qsort function.

Finally, the following statement copies the sorted array values from off-heap to on-heap memory:

```
    // Access off-heap memory
    sorted = array.toArray(ValueLayout.JAVA_INT);;
```

Memory Layouts and Structured Access

Accessing structured data using only basic operations can lead to hard-to-read code that's difficult to maintain. Instead, you can use *memory layouts* to more efficiently initialize and access more complicated native data types such as C structures.
For example, consider the following C declaration, which defines an array of `Point` structures, where each `Point` structure has two members, `Point.x` and `Point.y`:

```
struct Point {
  int x;
  int y;
} pts[10];
```

You can initialize such a native array as follows:

```
  try (Arena arena = Arena.openConfined()) {

    MemorySegment segment =
      arena.allocate((long)(2 * 4 * 10), 1);

    for (int i = 0; i < 10; i++) {
      segment.setAtIndex(ValueLayout.JAVA_INT, (i * 2),     i); // x
      segment.setAtIndex(ValueLayout.JAVA_INT, (i * 2) + 1, i); // y
    }
    // ...
  }
```

The first argument in the call to the `Arena::allocate` method calculates the number of bytes required for the array. The arguments in the calls to the `MemorySegment::setAtIndex` method calculate which memory address offsets to write into each member of a `Point` structure. To avoid these calculations, you can use a memory layout.
To represent the array of `Point` structures, the following example uses a sequence memory layout:

```
  try (Arena arena = Arena.openConfined()) {

    SequenceLayout ptsLayout
      = MemoryLayout.sequenceLayout(10,
        MemoryLayout.structLayout(
          ValueLayout.JAVA_INT.withName("x"),
          ValueLayout.JAVA_INT.withName("y")));
    VarHandle xHandle
      = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("x"));
    VarHandle yHandle
      = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("y"));

    MemorySegment segment = arena.allocate(ptsLayout);

    for (int i = 0; i < ptsLayout.elementCount(); i++) {
      xHandle.set(segment, (long) i, i);
      yHandle.set(segment, (long) i, i);
    }
    // ...
  }
```

The first statement creates a sequence memory layout, which is represented by a SequenceLayout object. It contains a sequence of ten structure layouts, which are represented by StructLayout objects. The method MemoryLayout::structLayout returns a StructLayout object. Each structure layout contains two JAVA_INT value layouts named x and y:

```
    SequenceLayout ptsLayout
      = MemoryLayout.sequenceLayout(10,
        MemoryLayout.structLayout(
          ValueLayout.JAVA_INT.withName("x"),
          ValueLayout.JAVA_INT.withName("y")));
```

The predefined value ValueLayout.JAVA_INT contains information about how many bytes a Java int value requires.

The next statements create two *memory-access VarHandles* that obtain memory address offsets. A VarHandle is a dynamically strongly typed reference to a variable or to a parametrically-defined family of variables, including static fields, non-static fields, array elements, or components of an off-heap data structure.

```
    VarHandle xHandle
      = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("x"));
    VarHandle yHandle
      = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("y"));
```

The method PathElement.sequenceElement() retrieves a memory layout from a sequence layout. In this example, it retrieves one of the structure layouts from ptsLayout. The method call PathElement.groupElement("x") retrieves a memory layout named x. You can create a memory layout with a name with the withName(String) method.

The for statement calls VarHandle::set to access memory like MemorySegment::setAtIndex. In this example, it sets a value (the third argument) at an index (the second argument) in a memory segment (the first argument). The VarHandles xHandle and yHandle know the size of the Point structure (8 bytes) and the size of its int members (4 bytes). This means you don't have to calculate the number of bytes required for the array's elements or the memory address offsets like in the setAtIndex method.

```
MemorySegment segment = arena.allocate(ptsLayout);

for (int i = 0; i < ptsLayout.elementCount(); i++) {
   xHandle.set(segment, (long) i, i);
   yHandle.set(segment, (long) i, i);
}
```

Restricted Methods

Some methods in the Foreign Function and Memory (FFM) API are unsafe and therefore *restricted*. If used incorrectly, restricted methods can crash the JVM and may silently result in memory corruption.

If you run an application that invokes one of the following restricted methods, the Java runtime will print a warning message. To enable code in a module *M* to use these restricted methods or any unsafe methods without warnings, specify the --enable-native-access=*M* command-line option. Specify multiple modules with a comma-separated list. To enable warning-free use for all code on the class path, specify the --enable-native-access=ALL-UNNAMED option.

Table 13-1 Restricted Methods from the FFM API

Methods	**Description**

Linker::nativeLinker()
SymbolLookup::libraryLookup(String, SegmentScope)
SymbolLookup::libraryLookup(Path, SegmentScope)
	These methods are required to create a downcall method handle, which is intrinsically unsafe. A symbol in a foreign library does not typically contain enough signature information, such as arity and the types of foreign function parameters, to enable the linker at runtime to validate linkage requests. When a client interacts with a downcall method handle obtained through an invalid linkage request, for example, by specifying a function descriptor featuring too many argument layouts, the result of such an interaction is unspecified and can lead to JVM crashes.

Methods	**Description**
	JVM crashes can occur with downcalls and upcalls because of a mismatch between the native function type and the FunctionDescriptor passed to the linker. Additionally, downcalls and upcalls can crash if the function that is being called is unloaded. For downcalls, this happens if the library that contains the function is unloaded. For upcalls, this happens if the upcall stub is deallocated because, for instance, the scope it is associated with is closed.
MemorySegment::ofAddress(long, long) MemorySegment::ofAddress(long, long, SegmentScope) MemorySegment::ofAddress(long, long, SegmentScope, Runnable) VaList::ofAddress(long, SegmentScope)	Sometimes it's necessary to turn a raw memory address obtained from native code into a memory segment with full spatial, temporal and confinement bounds so that you can access the memory segment directly. To do this, clients can call one of these methods to obtain a native segment *unsafely* from a memory address by providing the segment size and segment scope. This is a restricted operation because an incorrect segment size could result in a JVM crash when attempting to access the memory segment. For example, a raw memory address might be associated with a region of memory that is 10 bytes long, but the client might overestimate the size of the region and create a memory segment that is 100 bytes long. Later, this might result in attempts to access memory outside the bounds of the region, which might cause a JVM crash or, even worse, result in silent memory corruption.
ValueLayout.OfAddress::asUnbounded()	When the FFM API obtains a pointer such as by interacting with a native function or by reading a memory segment with the ValueLayout.ADDRESS layout, it doesn't know the size of the memory segment being returned. For this reason, such size is set to zero. While this is safe, it also means that clients can't use the returned memory segment for further access operations. To do so, call ValueLayout.OfAddress::asUnbounded() to obtain an *unbounded* address layout, which is mapped to a memory segment with maximal size, such as Long.MAX_VALUE, which clients can use in access operations. The InvokeQsort example described in Upcalls: Passing Java Code as a Function Pointer to a Foreign Function uses

Methods	**Description**
	unbounded address layouts, which enable the compareFunc upcall to quickly access and compare the contents of the two pointers passed to it.

Calling Native Functions with 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/
Obtain the source code for jextract from the following site:
https://github.com/openjdk/jextract
This site also contains steps on how to compile and run jextract, additional documentation, and samples.

Topics

- Run a Python Script in a Java Application
- Call the qsort Function from a Java Application

Run a Python Script in a Java Application

The following steps show you how to generate Java bindings from the Python header file, Python.h, then use the generated code to run a Python script in a Java application. The Python script prints the length of a Java string.

1. Run the following command to generate Java bindings for Python.h:

```
jextract -l <absolute path of Python shared library> \
 --output classes \
 -I <directory containing Python header files> \
 -t org.python <absolute path of Python.h>
```

   For example:

```
jextract -l /lib64/libpython3.6m.so.1.0 \
 --output classes \
 -I /usr/include/python3.6m \
 -t org.python /usr/include/python3.6m/Python.h
```

Note:

- On Linux systems, to obtain the file name of the Python shared library, run the following command. This example assumes that you have Python 3 installed in your system.

  ```
  ldd $(which python3)
  ```

  Running this command prints output similar to the following:

  ```
  linux-vdso.so.1 =>  (0x00007ffdb4bd5000)
  libpython3.6m.so.1.0 => /lib64/libpython3.6m.so.1.0 (0x00007fb0386a7000)
  libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fb03848b000)
  libdl.so.2 => /lib64/libdl.so.2 (0x00007fb038287000)
  libutil.so.1 => /lib64/libutil.so.1 (0x00007fb038084000)
  libm.so.6 => /lib64/libm.so.6 (0x00007fb037d82000)
  libc.so.6 => /lib64/libc.so.6 (0x00007fb0379b4000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fb038bce000)
  ```
- On Linux systems, if you can't find `Python.h` or the directory containing the Python header files, you might have to install the `python-devel` package.
- If you want to examine the classes and methods that the `jextract` tool creates, run the command with the `--source` option. For example, the following command generates the source files of the Java bindings for `Python.h`:

  ```
  jextract --source \
    --output src \
    -I <directory containing Python header files> \
    -t org.python <absolute path of Python.h>
  ```

2. In the same directory as `classes`, which should contain the Python Java bindings, create the following file, `PythonMain.java`:

```
import java.lang.foreign.Arena;
import java.lang.foreign.MemorySegment;
import static java.lang.foreign.MemorySegment.NULL;
import static org.python.Python_h.*;

public class PythonMain {

  public static void main(String[] args) {
    String myString = "Hello world!";
    String script = """
          string = "%s"
          print(string, ': ', len(string), sep='')
          """.formatted(myString).stripIndent();
    Py_Initialize();
```

```
        try (Arena arena = Arena.openConfined()) {
          MemorySegment nativeString = arena.allocateUtf8String(script);
          PyRun_SimpleStringFlags(
            nativeString,
            NULL);
          Py_Finalize();
        }
        Py_Exit(0);
      }
    }
```

3. Compile PythonMain.java with the following command:
   ```
   javac --enable-preview -source 20 -cp classes PythonMain.java
   ```
4. Run PythonMain with the following command:
   ```
   java --enable-preview -cp classes:. --enable-native-access=ALL-UNNAMED PythonMain
   ```

Call the qsort Function from a Java Application

As mentioned previously, qsort is a C library function that requires a pointer to a function that compares two elements. The following steps create Java bindings for the C standard library with jextract, create an upcall handle for the comparison function required by qsort, and then call the qsort function.

1. Run the following command to create Java bindings for stdlib.h, which is the header file for the C standard library:
   ```
   jextract --output classes -t org.unix <absolute path to stdlib.h>
   ```
   For example:
   ```
   jextract --output classes -t org.unix /usr/include/stdlib.h
   ```
   The generated Java bindings for stdlib.h include a Java class named stdlib_h, which includes a Java method named qsort(MemorySegment, long, long, MemorySegment), and a Java interface named __compar_fn_t, which includes a method named allocate that creates a function pointer for the comparison function required by the qsort function. To examine the source code of the Java bindings that jextract generates, run the tool with the --source option:
   ```
   jextract --source --output src -t org.unix <absolute path to stdlib.h>
   ```
2. In the same directory where you generated the Java bindings for stdlib.h, create the following Java source file, QsortMain.java:
   ```
   import static org.unix.__compar_fn_t.*;
   import static org.unix.stdlib_h.*;
   import java.lang.foreign.*;
   import java.lang.invoke.*;

   public class QsortMain {
   ```

```
  public static void main(String[] args) {

    int[] unsortedArray = new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 };

    try (Arena a = Arena.openConfined()) {

      // Allocate off-heap memory and store unsortedArray in it
      MemorySegment array = a.allocateArray(
        ValueLayout.JAVA_INT,
        unsortedArray);

      // Create upcall for comparison function
      MemorySegment comparFunc = allocate(
        (addr1, addr2) ->
          Integer.compare(
            addr1.get(ValueLayout.JAVA_INT, 0),
            addr2.get(ValueLayout.JAVA_INT, 0)),
          a.scope());

      // Call qsort
      qsort(array, (long) unsortedArray.length,
        ValueLayout.JAVA_INT.byteSize(), comparFunc);

      // Deference off-heap memory
      int[] sortedArray = array.toArray(ValueLayout.JAVA_INT);

      for (int num : sortedArray) {
        System.out.print(num + " ");
      }
      System.out.println();
    }
  }
}
```

The following statement creates an upcall, `comparFunc`, from a lambda expression:

```
      // Create upcall for comparison function
      MemorySegment comparFunc = allocate(
        (addr1, addr2) ->
          Integer.compare(
            addr1.get(ValueLayout.JAVA_INT, 0),
            addr2.get(ValueLayout.JAVA_INT, 0)),
          arena.scope());
```

Consequently, you don't have to create a method handle for the comparison function as described in Upcalls: Passing Java Code as a Function Pointer to a Foreign Function.

3. Compile `QsortMain.java` with the following command:

```
javac --enable-preview -source 20 -cp classes QsortMain.java
```

4. Run `QsortMain` with the following command:

```
java --enable-preview -cp classes:. --enable-native-access=ALL-UNNAMED QsortMain
```