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 442.
The FFM API is contained in the package java.lang.foreign.

Topics

On-Heap and Off-Heap Memory

On-heap memory is memory in the Java heap, which is a region of memory managed by the garbage collector. Java objects reside in the heap. The heap can grow or shrink while the application runs. When the heap becomes full, garbage collection is performed: The JVM identifies the objects that are no longer being used (unreachable objects) and recycles their memory, making space for new allocations.
Off-heap memory is memory outside the Java heap. 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. 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 deallocated.
You interact with off-heap memory through a MemorySegment object. You allocate a MemorySegment object with an *arena*, which enables you to specify *when* the off-heap memory associated with the MemorySegment object is deallocated.

Memory Segments and Arenas

You can access off-heap or on-heap memory with the Foreign Function and Memory (FFM) API through the MemorySegment interface. Each memory segment is associated with, or *backed by*, a contiguous region of memory. There are two kinds of memory segments:

- Heap segment: This is a memory segment backed by a region of memory inside the Java heap, an *on-heap* region.
- Native segment: This is a memory segment backed by a region of memory outside the Java heap, an *off-heap* region. The examples in this chapter demonstrate how to allocate and access native segments.

An *arena* controls the lifecycle of native memory segments. To create an arena, use one of the methods in the Arena interface, such as Arena.ofConfined(). You use an arena to allocate a memory segment. 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*.

Most of the examples described in this chapter use a *confined arena*, which is 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. 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.

There are other kinds of arenas:

- A shared arena, which is created with Arena::ofShared, 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. See Slicing Memory Segments for an example of a shared arena.
- An automatic arena, which is 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.
- A global arena, which is 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.

The following example allocates a memory segment with an arena, stores a Java String in the off-heap memory associated with the memory segment, and then prints the contents of the off-heap memory. At the end of the try-with-resources block, the arena is closed, and the off-heap memory associated with the memory segment is deallocated.

```
String s = "My string";
try (Arena arena = Arena.ofConfined()) {

    // 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
```

The following sections describe this example in detail:

- Allocating a Memory Segment with an Arena
- Storing a Java String in the Off-Heap Memory Associated with the Memory Segment
- Printing the Contents of Off-Heap Memory
- Closing an Arena

Allocating a Memory Segment with an Arena

The Arena interface extends the SegmentAllocator interface, which contains methods that both allocate off-heap memory and copy Java data into it. The previous 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.

```
String s = "My string";
try (Arena arena = Arena.ofConfined()) {

    // 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.

Storing a Java String in the Off-Heap Memory Associated with the Memory Segment

The Arena interface extends the SegmentAllocator interface, which contains methods that both allocate off-heap memory and copy Java data into it. The previous 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.

```
String s = "My string";
try (Arena arena = Arena.ofConfined()) {

    // 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.

Printing the Contents of 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 interface contains various access methods that enable you to read from or write to memory segments. 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
- Byte alignment set to 1: 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).

Closing an Arena

When an arena is closed, such as through a try-with-resources statement, then the arena's scope is no longer alive: All memory segments associated with its scope are invalidated, and the memory regions backing them are deallocated.
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:

```
String s = "My String";
MemorySegment nativeText;
try (Arena arena = Arena.ofConfined()) {

  // 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));
}
```

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.ofConfined()) {

        // Allocate off-heap memory and
        // copy the argument, a Java string, into off-heap memory
        MemorySegment nativeString = arena.allocateUtf8String(s);

        // Link and call the C function strlen

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

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

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

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

        // 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.

The following sections describe this example in detail:

Obtaining an Instance of the Native Linker

The following statement obtains an instance of the *native linker*, which 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 native function's address.

The following statements obtain the address of the strlen function:

```
SymbolLookup stdLib = SymbolLookup.libraryLookup("libc.so.6", arena);
MemorySegment strlen_addr = stdLib.find("strlen").get();
```

SymbolLookup.libraryLookup(String, Arena) creates a library lookup, which locates all the symbols in a user-specified native library. It loads the native library and associates it with an arena, which controls the library lookup's lifetime. 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:

```
// Obtain an instance of the C function
Linker linker = Linker.nativeLinker();

// Locate the address of the C function
SymbolLookup stdLib = linker.defaultLookup();
```

```
MemorySegment strlen_addr = stdLib.find("strlen").get();
```

Tip:

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's 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:

```
// 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.

```
// 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:

```
// 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 the 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 the 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 named `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 elem1, MemorySegment elem2) {
      return Integer.compare(elem1.get(ValueLayout.JAVA_INT, 0), elem2.get(ValueLayout.JAVA_INT, 0));
    }
  }

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

  static int[] qsortTest(int[] unsortedArray) throws Throwable {

    int[] sorted = null;

    // Create downcall handle for qsort
    MethodHandle qsort = linker.downcallHandle(
      linker.defaultLookup().find("qsort").get(),
```

```
    FunctionDescriptor.ofVoid(ValueLayout.ADDRESS,
                              ValueLayout.JAVA_LONG,
                              ValueLayout.JAVA_LONG,
                              ValueLayout.ADDRESS));

    // Create method handle for qsortCompare
    MethodHandle comparHandle = MethodHandles.lookup()
        .findStatic(Qsort.class,
                "qsortCompare",
                MethodType.methodType(int.class,
                                      MemorySegment.class,
                                      MemorySegment.class));

    // Create a Java description of a C function implemented by a Java method
    FunctionDescriptor qsortCompareDesc = FunctionDescriptor.of(
        ValueLayout.JAVA_INT,
        ValueLayout.ADDRESS.withTargetLayout(ValueLayout.JAVA_INT),
        ValueLayout.ADDRESS.withTargetLayout(ValueLayout.JAVA_INT));


    // Create function pointer for qsortCompare
    MemorySegment compareFunc = linker.upcallStub(comparHandle,
                                    qsortCompareDesc,
                                    Arena.ofAuto());

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

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

        // 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 sections describe this example in detail:

- Defining the Java Method That Compares Two Elements
- Creating a Downcall Method Handle for the qsort Function
- Creating a Method Handle to Represent the Comparison Method qsortCompare
- Creating a Function Pointer from the Method Handle compareHandle
- Allocating Off-Heap Memory to Store the int Array
- Calling the qsort Function
- Copying the Sorted Array Values from Off-Heap to On-Heap Memory

Defining the Java Method That Compares Two Elements

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

```
class Qsort {
  static int qsortCompare(MemorySegment elem1, MemorySegment elem2) {
    return Integer.compare(elem1.get(ValueLayout.JAVA_INT, 0), elem2.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.

Creating a Downcall Method Handle for the qsort Function

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

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

static int[] qsortTest(int[] unsortedArray) throws Throwable {

    int[] sorted = null;

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

Creating a Method Handle to Represent the Comparison Method qsortCompare

The following statement creates a method handle to represent the comparison method Qsort::qsortCompare:

```
    // Create method handle for qsortCompare
    MethodHandle comparHandle = MethodHandles.lookup()
        .findStatic(Qsort.class,
                "qsortCompare",
                MethodType.methodType(int.class,
                                      MemorySegment.class,
                                      MemorySegment.class));
```

The MethodHandles.Lookup.findStatic(Class, String, MethodType) 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.

Creating a Function Pointer from the Method Handle compareHandle

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

```
// Create a Java description of a C function implemented by a Java method
FunctionDescriptor qsortCompareDesc = FunctionDescriptor.of(
  ValueLayout.JAVA_INT,
  ValueLayout.ADDRESS.withTargetLayout(ValueLayout.JAVA_INT),
  ValueLayout.ADDRESS.withTargetLayout(ValueLayout.JAVA_INT));

// Create function pointer for qsortCompare
MemorySegment compareFunc = linker.upcallStub(comparHandle,
                            qsortCompareDesc,
                            Arena.ofAuto());
```

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 arena to associate with the function pointer. The static method Arena.ofAuto() creates a new arena that is managed, automatically, by the garbage collector.

Allocating Off-Heap Memory to Store the int Array

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

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

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

Calling the qsort Function

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.

Copying the Sorted Array Values from Off-Heap to On-Heap Memory

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);
```

Foreign Functions That Return Pointers

Sometimes foreign functions allocate a region of memory, then return a pointer to that region. For example, the C standard library function void *malloc(size_t) allocates the requested amount of memory, in bytes, and returns a pointer to it. However, when you invoke a native function that returns a pointer, like malloc, the Java runtime has no insight into the size or the lifetime of the memory segment the pointer points to. Consequently, the FFM API uses a *zero-length memory segment* to represent this kind of pointer.

The following example invokes the C standard library function malloc. It prints a diagnostic message immediately after, which demonstrates that the pointer returned by malloc is a zero-length memory segment.

```
static MemorySegment allocateMemory(long byteSize, Arena arena) throws Throwable {

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

    // Locate the address of malloc()
    var malloc_addr = linker.defaultLookup().find("malloc").orElseThrow();

    // Create a downcall handle for malloc()
    MethodHandle malloc = linker.downcallHandle(
      malloc_addr,
      FunctionDescriptor.of(ValueLayout.ADDRESS, ValueLayout.JAVA_LONG)
    );

    // Invoke malloc(), which returns a pointer
    MemorySegment segment = (MemorySegment) malloc.invokeExact(byteSize);

    // The size of the memory segment created by malloc() is zero bytes!
    System.out.println(
      "Size, in bytes, of memory segment created by calling malloc.invokeExact(" +
      byteSize + "): " + segment.byteSize());

    // Localte the address of free()
    var free_addr = linker.defaultLookup().find("free").orElseThrow();

    // Create a downcall handle for free()
    MethodHandle free = linker.downcallHandle(
      free_addr,
      FunctionDescriptor.ofVoid(ValueLayout.ADDRESS)
    );
```

```
    // This reintepret method:
    // 1. Resizes the memory segment so that it's equal to byteSize
    // 2. Associates it with an existing arena
    // 3. Invokes free() to deallocate the memory allocated by malloc()
    //    when its arena is closed

    Consumer<MemorySegment> cleanup = s -> {
      try {
        free.invokeExact(s);
      } catch (Throwable e) {
        throw new RuntimeException(e);
      }
    };

    return segment.reinterpret(byteSize, arena, cleanup);
  }
```

The example prints a message similar to the following:

```
Size, in bytes, of memory segment created by calling malloc.invokeExact(100): 0
```

The FFM API uses zero-length memory segments to represent the following:

- Pointers returned from a foreign function
- Pointers passed by a foreign function to an upcall
- Pointers read from a memory segment

If you try to access a zero-length memory segment, the Java runtime will throw an `IndexOutOfBoundsException` because the Java runtime can't safely access or validate any access operation of a region of memory whose size is unknown. In addition, zero-length memory segments are associated with a fresh scope that's always alive. Consequently, even though you can't directly access zero-length memory segments, you can pass them to other pointer-accepting foreign functions.
However, the `MemorySegment::reinterpret` method enables you to work with zero length memory segments so that you can safely access them and attach them to an existing arena so that the lifetime of the region of memory backing the segment can be managed automatically. This method takes three arguments:

- The number of bytes to resize the memory segment: The example resizes it to the value of the parameter `byteSize`.
- The arena with which to associate the memory segment: The example associates it to the arena specified by the parameter `arena`.

- The action to perform when the arena is closed: The example deallocates the memory allocated by malloc by invoking the C standard library function void free(void *ptr), which deallocates the memory referenced by a pointer returned by malloc. Note that this is an example of passing a pointer pointing to a zero-length memory segment to a foreign function.

Note:

MemorySegment::reinterpret 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 following example calls allocateMemory(long, Arena) to allocate a Java string with malloc:

```
String s = "My string!";
try (Arena arena = Arena.ofConfined()) {

   // Allocate off-heap memory with malloc()
   var nativeText = allocateMemory(
      ValueLayout.JAVA_CHAR.byteSize() * (s.length() + 1), arena);

   // Access off-heap memory
   for (int i = 0; i < s.length(); i++ ) {
      nativeText.setAtIndex(ValueLayout.JAVA_CHAR, i, s.charAt(i));
   }

   // Add the string terminator at the end
   nativeText.setAtIndex(
       ValueLayout.JAVA_CHAR, s.length(), Character.MIN_VALUE);

   // Print the string
   for (int i = 0; i < s.length(); i++ ) {
      System.out.print((char)nativeText.getAtIndex(ValueLayout.JAVA_CHAR, i));
   }
   System.out.println();
} catch (Throwable t) {
   t.printStackTrace();
}
```

See Zero-length memory segments in the java.lang.foreign.MemorySegment API specification and Functions returning pointers in the java.lang.foreign.Linker API specification for more information.

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.ofConfined()) {

      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.ofConfined()) {

      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);
}
```

Checking for Native Errors Using errno

Some C standard library functions indicate errors by setting the value of the C standard library macro errno. You can access this value with a FFM API linker option.
The Linker::downcallHandle method contains a varargs parameter that enables you to specify additional linker options. These parameters are of type Linker.Option.
One linker option is Linker.Option.captureCallState(String...), which you use to save portions of the execution state immediately after calling a foreign function associated with a downcall method handle. You can use it to capture certain thread-local variables. When used with the "errno" string, it captures the errno value as defined by the C standard library. Specify this linker option (with the "errno" string) when creating a downcall handle for a native function that sets errno.
An example of a C standard library function that sets errno is log, which computes the natural (base *e*) logarithm of its argument. If this value is less than zero, then errno is set to the value 33, which represents a domain error. As most users won't recognize 33 as a domain error, you can invoke the C standard library function strerror, which returns a textual description of the errno value.
The following example invokes the log function then uses captureCallState("errno") to obtain error messages set by the log function:

```
static double invokeLog(double v) throws Throwable {

    double result = Double.NaN;

    // Setup handles
    Linker.Option ccs = Linker.Option.captureCallState("errno");
    StructLayout capturedStateLayout = Linker.Option.captureStateLayout();
    VarHandle errnoHandle = capturedStateLayout.varHandle(PathElement.groupElement("errno"));

    // log C Standard Library function
    Linker linker = Linker.nativeLinker();
    SymbolLookup stdLib = linker.defaultLookup();
    MethodHandle log = linker.downcallHandle(
      stdLib.find("log").orElseThrow(),
      FunctionDescriptor.of(ValueLayout.JAVA_DOUBLE, ValueLayout.JAVA_DOUBLE),
      ccs);

    // strerror C Standard Library function
    MethodHandle strerror = linker.downcallHandle(
      stdLib.find("strerror").orElseThrow(),
      FunctionDescriptor.of(ValueLayout.ADDRESS, ValueLayout.JAVA_INT));

    // Actual invocation
    try (Arena arena = Arena.ofConfined()) {
```

```
        MemorySegment capturedState = arena.allocate(capturedStateLayout);

        result = (double) log.invokeExact(capturedState, v);

        if (Double.isNaN(result)) {
          // Indicates that an error occurred per the documentation of
          // the 'log' command.

          // Get more information by consulting the value of errno:
          int errno = (int) errnoHandle.get(capturedState);
          System.out.println("errno: " + errno); // 33

          // Convert errno code to a string message:
          String errrorString = ((MemorySegment) strerror.invokeExact(errno))
            .reinterpret(Long.MAX_VALUE).getUtf8String(0);
          System.out.println("errno string: " + errrorString); // Domain error
        }
      }
      return result;
    }
```

In this example, the method captureStateLayout() returns a structure layout of the errno function. See Memory Layouts and Structured Access for more information. Suppose that you call invokeLog(double) as follows:

```
System.out.println("log(2.718): " + invokeLog(2.718));
System.out.println("log(-1): " + invokeLog(-1));
```

The example then prints output similar to the following:

```
log(2.718): 0.999896315728952
errno: 33
errno string: Domain error
log(-1): NaN
```

Tip:

Use the following code to obtain the names of the supported captured value layouts for the Linker.Option.captureCallState(String...) option for your operating system:

```
List<String> capturedNames = Linker.Option.captureStateLayout()
    .memberLayouts()
    .stream()
    .map(MemoryLayout::name)
    .flatMap(Optional::stream)
    .toList();
```

Slicing Allocators and Slicing Memory Segments

A *slicing allocator* returns a segment allocator that responds to allocation requests by returning consecutive contiguous regions of memory, or *slices*, obtained from an existing memory segment. You can also obtain a slice of a memory segment of any location within a memory segment with the method MethodSegment::asSlice.

Topics

Slicing Allocators

The following example allocates a memory segment named segment that can hold 60 Java int values. It then uses a slicing allocator by calling SegmentAllocator.slicingAllocator(MemorySgement) to obtain ten consecutive slices from segment. The example allocates an array of five integers in each slice. After, it prints the contents of each slice.

```
try (Arena arena = Arena.ofConfined()) {
   SequenceLayout SEQUENCE_LAYOUT =
      MemoryLayout.sequenceLayout(60L, ValueLayout.JAVA_INT);
   MemorySegment segment = arena.allocate(SEQUENCE_LAYOUT);
   SegmentAllocator allocator = SegmentAllocator.slicingAllocator(segment);

   MemorySegment s[] = new MemorySegment[10];

   for (int i = 0 ; i < 10 ; i++) {
      s[i] = allocator.allocateArray(
         ValueLayout.JAVA_INT, 1, 2, 3, 4, 5);
   }

   for (int i = 0 ; i < 10 ; i++) {
      int[] intArray = s[i].toArray(ValueLayout.JAVA_INT);
      System.out.println(Arrays.toString(intArray));
   }

} catch (Exception e) {
   e.printStackTrace();
}
```

You can use segment allocators as building blocks to create arenas that support custom allocation strategies. For example, if a large number of native segments will share the same bounded lifetime, then a custom arena could use a slicing allocator to allocate the segments efficiently. This lets clients enjoy both scalable allocation (thanks to slicing) and deterministic deallocation (thanks to the arena).

The following example defines a *slicing arena* that behaves like a confined arena but internally uses a slicing allocator to respond to allocation requests. When the slicing arena is closed, the underlying confined arena is closed, invalidating all segments allocated in the slicing arena.
To keep this example short, it implements only a subset of the methods of `Arena` and `SegmentAllocator` (which is a superinterface of `Arena`).

```
public class SlicingArena implements Arena {
  final Arena arena = Arena.ofConfined();
  final SegmentAllocator slicingAllocator;

  SlicingArena(MemoryLayout m) {
    slicingAllocator = SegmentAllocator.slicingAllocator(arena.allocate(m));
  }

  public MemorySegment allocateArray(
    ValueLayout.OfInt elementLayout, int... elements) {
    return slicingAllocator.allocateArray(elementLayout, elements);
  }

  public MemorySegment.Scope scope() {
    return arena.scope();
  }

  public void close() {
    arena.close();
  }
}
```

With this slicing arena, you can rewrite the first example in this section more succinctly:

```
    SequenceLayout SEQUENCE_LAYOUT =
      MemoryLayout.sequenceLayout(60L, ValueLayout.JAVA_INT);
    try (Arena slicingArena = new SlicingArena(SEQUENCE_LAYOUT)) {
      MemorySegment s[] = new MemorySegment[10];

      for (int i = 0 ; i < 10 ; i++) {
        s[i] = slicingArena.allocateArray(
```

```
            ValueLayout.JAVA_INT, 1, 2, 3, 4, 5);
        }

        for (int i = 0 ; i < 10 ; i++) {
            int[] intArray = s[i].toArray(ValueLayout.JAVA_INT);
            System.out.println(Arrays.toString(intArray));
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
```

Slicing Memory Segments

When a slicing allocator returns a slice, the slice's starting address is right after the end of the last slice that the slicing allocator returned. You can call MemorySegment.asSlice(long,long) to obtain a slice of a memory segment of any location within the memory segment and of any size, provided that slice's size stays within the spatial bounds of the original memory segment. The following example obtains a slice of a memory segment, then prints its contents:

```
String s = "abcdefghijklmnopqrstuvwxyz";
char c[] = s.toCharArray();

MemorySegment textSegment = MemorySegment.ofArray(c);
long b = ValueLayout.JAVA_CHAR.byteSize();
long firstLetter = 5;
long size = 6;

MemorySegment fghijk = textSegment.asSlice(firstLetter*b, size*b);

for (int i = 0; i < size; i++) {
    System.out.print((char)fghijk.get(ValueLayout.JAVA_CHAR, i*b));
}
System.out.println();
```

This example prints the following output:

```
fghijk
```

The method MemorySegment.elements(MemoryLayout) returns a stream of slices whose size matches that of the specified layout. Multiple threads could work in parallel to access these slices. To do this, however, the memory segment has to be accessible from multiple threads. You can do this by associating the memory segment with a shared arena, which you can create with Arena::ofShared.

The following example sums all int values in a memory segment in parallel.

```
  void addRandomNumbers(int numElements) throws Throwable {

    int[] numbers = new Random().ints(numElements, 0, 1000).toArray();

    try (Arena arena = Arena.ofShared()) {
      SequenceLayout SEQUENCE_LAYOUT = MemoryLayout.sequenceLayout((long)numElements,
ValueLayout.JAVA_INT);
      MemorySegment segment = arena.allocate(SEQUENCE_LAYOUT);
      MemorySegment.copy(numbers, 0, segment, ValueLayout.JAVA_INT, 0L, numElements);

      int sum = segment.elements(ValueLayout.JAVA_INT).parallel()
          .mapToInt(s -> s.get(ValueLayout.JAVA_INT, 0))
          .sum();

      System.out.println(sum);
    }
  }
```

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 12-1 Restricted Methods from the FFM API

Methods	Reasoning Behind Restricting the Methods
java.lang.ModuleLayer.Controller.enableNativeAccess(Module)	The method enables native access for the specified module if the caller's module has native acces. This method is restricted because it propagates privileges to call restricted methods.
AddressLayout.withTargetLayout(MemoryLayout)	Once you have an address layout with a given target layout, you can use it in a dereference operation (for example, MemorySegment.get(AddressLayout, long) to *resize* the segment being read, which is unsafe.
Linker.downcallhandle(FunctionDescriptor, Linker.Option...) Linker.downcallhandle(MemorySegment, FunctionDescriptor, Linker.Option...)	Creating a downcall method handle is intrinsically unsafe. A linker has no way to verify that the provided function descriptor is compatible with the function being called. 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

Methods	Reasoning Behind Restricting the Methods
	such an interaction is unspecified and can lead to JVM crashes.
Linker.upcallStub(MethodHandle, FunctionDescriptor, Arena, Linker.Option...)	As with creating downcall handles, the linker can't check whether the function pointer you are creating (like the qsort comparator in the example in Upcalls: Passing Java Code as a Function Pointer to a Foreign Function) is the correct one for the for the downcall you are passing it to (like the qsort method handle in the same example).
MemorySegment.reinterpret(long) MemorySegment.reinterpret(Arena, Consumer) MemorySegment.reinterpret(long, Arena, Consumer)	These methods allows you to change the size and lifetime of an existing segment by creating a new alias to the same region of memory. See Foreign Functions That Return Pointers for more information. The spatial or temporal bounds associated with the memory segment alias returned by these methods might be incorrect. For example, consider a region of memory that's 10 bytes long that's backing a zero-length memory segment. An application might overestimate the size of the region and use MemorySegment::reinterpret to obtain a segment that's 100 bytes long. Later, this might result in attempts to dereference memory outside the bounds of the region, which might cause a JVM crash or, even worse, result in silent memory corruption.
SymbolLookup.libraryLookup(String, Arena) SymbolLookup.libraryLookup(Path, Arena)	Loading a library can always cause execution of native code. For example, on Linux, they can be executed through dlopen hooks.

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
   ```

   Tip:

   - 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.ofConfined()) {
      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 21 -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.Arena;
import java.lang.foreign.MemorySegment;
import java.lang.foreign.ValueLayout;

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.ofConfined()) {
```

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

        // Create upcall for comparison function
        //
        // MemorySegment allocate(__compar_fn_t, Arena) is from
        // __compar_fn-t.java, generated by jextract

        MemorySegment comparFunc = allocate(
          (addr1, addr2) ->
            Integer.compare(
              addr1.get(ValueLayout.JAVA_INT, 0),
              addr2.get(ValueLayout.JAVA_INT, 0)),
            a);

        // 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 allocate(__compar_fn_t, SegmentScope) is from
        // __compar_fn-t.java, generated by jextract

        MemorySegment comparFunc = allocate(
          (addr1, addr2) ->
            Integer.compare(
              addr1.get(ValueLayout.JAVA_INT, 0),
              addr2.get(ValueLayout.JAVA_INT, 0)),
```

```
        a);
```

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 21 -cp classes QsortMain.java
   ```

4. Run `QsortMain` with the following command:

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