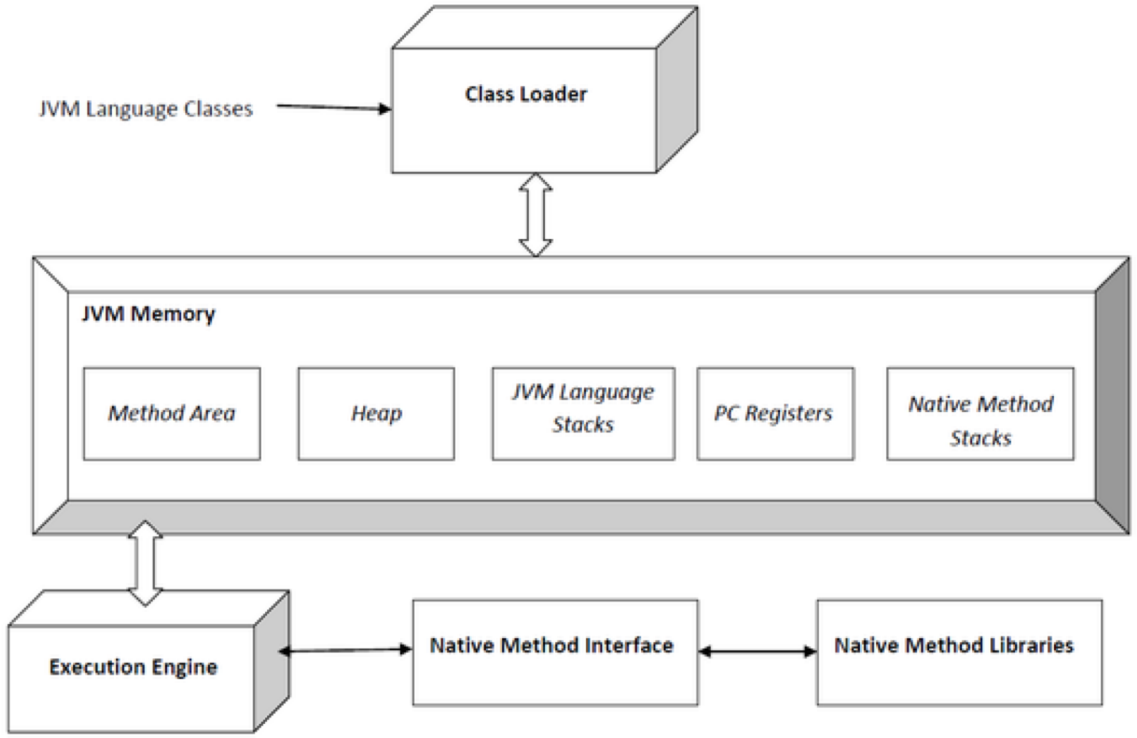
Architecture of the Java Virtual Machine



The Java Native Interface (JNI)

- [Ref: Java Native Interface Specification, https://docs.oracle.com/en/java/javase/16/docs/specs/jni/index.html]
- The JNI is a native programming interface. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

The Java Native Interface (JNI)

- [Ref: Java Native Interface Specification, https://docs.oracle.com/en/java/javase/16/docs/specs/jni/index.html]
- The JNI is a native programming interface. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.
- Reasons for needing `native` methods in Java code
  - To provide platform-dependent features needed by the application that are not supported by the standard Java class library.
  - To make an existing library written in another language accessible to Java code.
  - To implement a small portion of time-critical code in a lower-level language such as assembly.

JNI Usage Models

- Using JNI functions (JNI, §4) to allow `native` methods to:
  - Create, inspect, and update Java objects (including arrays and strings).
  - Call Java methods.
  - Catch and throw exceptions.
  - Load classes and obtain class information.
  - Perform runtime type checking.

JNI Usage Models

- Using JNI functions (JNI, §4) to allow `native` methods to:
  - Create, inspect, and update Java objects (including arrays and strings).
  - Call Java methods.
  - Catch and throw exceptions.
  - Load classes and obtain class information.
  - Perform runtime type checking.

- Using the Invocation API (JNI, §5) to enable an arbitrary native application to embed the Java VM.
  - This allows programmers to easily make their existing applications Java-enabled without having to link with the VM source code.

JNI Usage Models

- Using JNI functions (JNI, §4) to allow `native` methods to:
  - Create, inspect, and update Java objects (including arrays and strings).
  - Call Java methods.
  - Catch and throw exceptions.
  - Load classes and obtain class information.
  - Perform runtime type checking.

- Using the Invocation API (JNI, §5) to enable an arbitrary native application to embed the Java VM.
  - This allows programmers to easily make their existing applications Java-enabled without having to link with the VM source code.

- JNI and COM
  - Although Java objects are not exposed to the native code as COM objects, the JNI interface itself is binary-compatible with COM.

Example: A `native` Method

```
package p.q.r;
class A {
    native double f(int i, String s);
    static {
        System.loadLibrary("p_q_r_A");
    }
}
```

Example: A `native` Method

```
package p.q.r;
class A {
    native double f(int i, String s);
    static {
        System.loadLibrary("p_q_r_A");
    }
}
```

- [Compiling] As the Java VM is multithreaded, native libraries must be compiled/linked with multithread-aware native compilers.
  - For code complied with `gcc`, use the flags `-D_REENTRANT` or `-D_POSIX_C_SOURCE`.

Example: A `native` Method

```
package p.q.r;
class A {
    native double f(int i, String s);
    static {
        System.loadLibrary("p_q_r_A");
    }
}
```

- [Compiling] As the Java VM is multithreaded, native libraries must be compiled/linked with multithread-aware native compilers.
  - For code complied with `gcc`, use the flags `-D_REENTRANT` or `-D_POSIX_C_SOURCE`.
- [Loading and linking] The argument to `System.loadLibrary` is a library name chosen arbitrarily by the programmer.
  - The system follows a standard, but platform-specific, approach to convert the library name to a native library name.
  - For example, a Linux converts the name `p_q_r_A` to `libp_q_r_A.so`.

Resolving `native` Method Names

```
package p.q.r;
class A {
    native double f(int i, String s);
}
```

- The JNI defines a 1:1 mapping from the name of the Java `native` method to the name of its implementation by concatenating the following components.
  - the prefix (“`Java_“ );`
  - the given binary name, in internal form, of the class which declares the `native` method (the result of escaping the name);
  - an underscore ("_");
  - the escaped method name;
  - only if the `native` method declaration is overloaded by another `native` method: two underscores ("__") followed by the escaped parameter descriptor (JVMS §4.3.3) of the method declaration.

Resolving `native` Method Names

```
package p.q.r;
class A {
    native double f(int i, String s);
}
```

- The JNI defines a 1:1 mapping from the name of the Java `native` method to the name of its implementation by concatenating the following components.
  - the prefix (“`Java_`”);
  - the given binary name, in internal form, of the class which declares the `native` method (the result of escaping the name);
  - an underscore ("_");
  - the escaped method name;
  - only if the `native` method declaration is overloaded by another `native` method: two underscores ("__") followed by the escaped parameter descriptor (JVMS §4.3.3) of the method declaration.
- Short name: `Java_p_q_r_A_f`.
- Long name: `Java_p_q_r_A_f__ILjava_lang_String_2`.

Native Method Arguments

```
package p.q.r;
class A {
    native double f(int i, String s);
    native double f(int i, Object s);
}
```

```
jdouble Java_p_q_r_A_f__ILjava_lang_String_2 (
    JNIEnv *env, /* interface pointer */
    jobject obj, /* "this" pointer */
    jint i, /* argument #1 */
    jstring s) /* argument #2 */
{
    /* Obtain a C-copy of the Java string */
    const char *str =
        (*env)->GetStringUTFChars(env, s, 0);
    /* process the string */
    ...
    /* Now we are done with str */
    (*env)->ReleaseStringUTFChars(env, s, str);
    return ...
}
```

Native Method Arguments

```
package p.q.r;
class A {
    native double f(int i, String s);
    native double f(int i, Object s);
}
```

- First argument:
  - JNI interface pointer, of type `JNIEnv`.

```
jdouble Java_p_q_r_A_f__ILjava_lang_String_2 (
    JNIEnv *env, /* interface pointer */
    jobject obj, /* "this" pointer */
    jint i, /* argument #1 */
    jstring s) /* argument #2 */
{
    /* Obtain a C-copy of the Java string */
    const char *str =
        (*env)->GetStringUTFChars(env, s, 0);
    /* process the string */
    ...
    /* Now we are done with str */
    (*env)->ReleaseStringUTFChars(env, s, str);
    return ...
}
```

Native Method Arguments

```
package p.q.r;
class A {
    native double f(int i, String s);
    native double f(int i, Object s);
}
```

- First argument:
  - JNI interface pointer, of type `JNIEnv`.
- Second argument:
  - For non-static method, a reference to the object.
  - For static method, a reference to its Java class.

```
jdouble Java_p_q_r_A_f__ILjava_lang_String_2 (
    JNIEnv *env, /* interface pointer */
    jobject obj, /* "this" pointer */
    jint i, /* argument #1 */
    jstring s) /* argument #2 */
{
    /* Obtain a C-copy of the Java string */
    const char *str =
        (*env)->GetStringUTFChars(env, s, 0);
    /* process the string */
    ...
    /* Now we are done with str */
    (*env)->ReleaseStringUTFChars(env, s, str);
    return ...
}
```

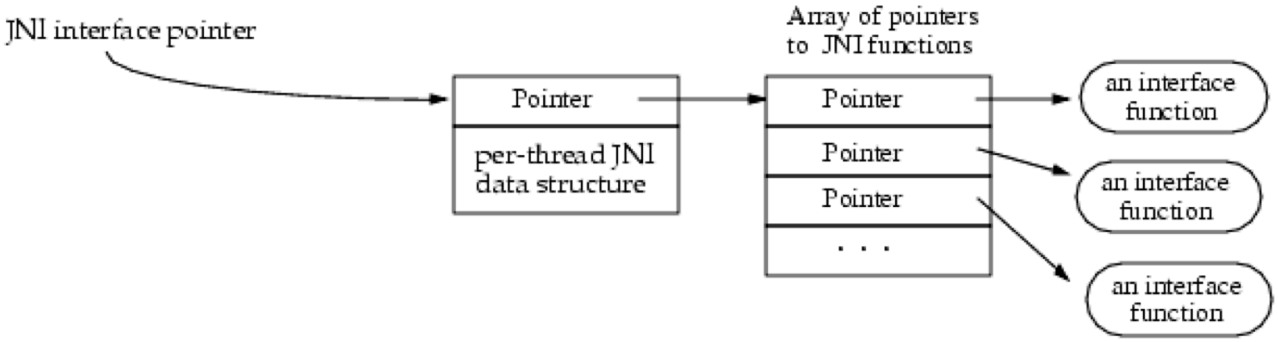
Native Method Arguments

```
package p.q.r;
class A {
    native double f(int i, String s);
    native double f(int i, Object s);
}
```

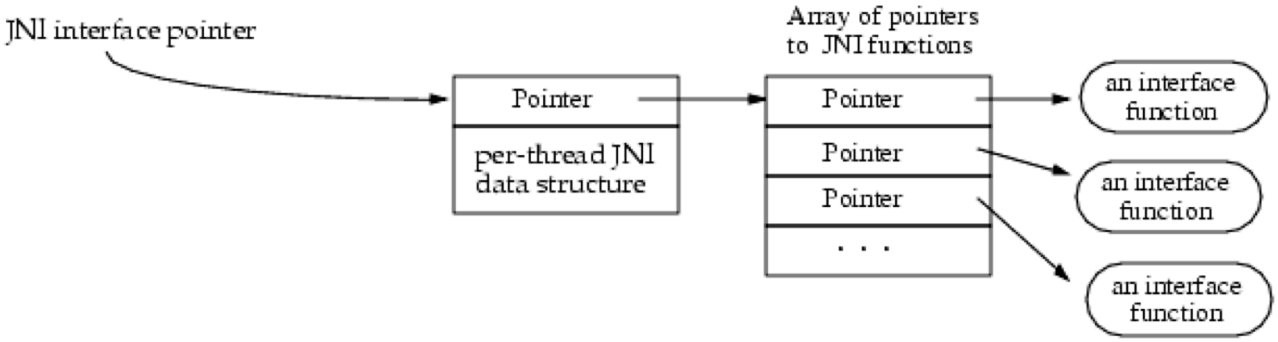
- First argument:
  - JNI interface pointer, of type `JNIEnv`.
- Second argument:
  - For non-static method, a reference to the object.
  - For static method, a reference to its Java class.
- Method arguments follow, with Java types mapped to native equivalents.
- Results passed back via return value.

```
jdouble Java_p_q_r_A_f__ILjava_lang_String_2 (
    JNIEnv *env, /* interface pointer */
    jobject obj, /* "this" pointer */
    jint i, /* argument #1 */
    jstring s) /* argument #2 */
{
    /* Obtain a C-copy of the Java string */
    const char *str =
        (*env)->GetStringUTFChars(env, s, 0);
    /* process the string */
    ...
    /* Now we are done with str */
    (*env)->ReleaseStringUTFChars(env, s, str);
    return ...
}
```

JNI Interface Pointer

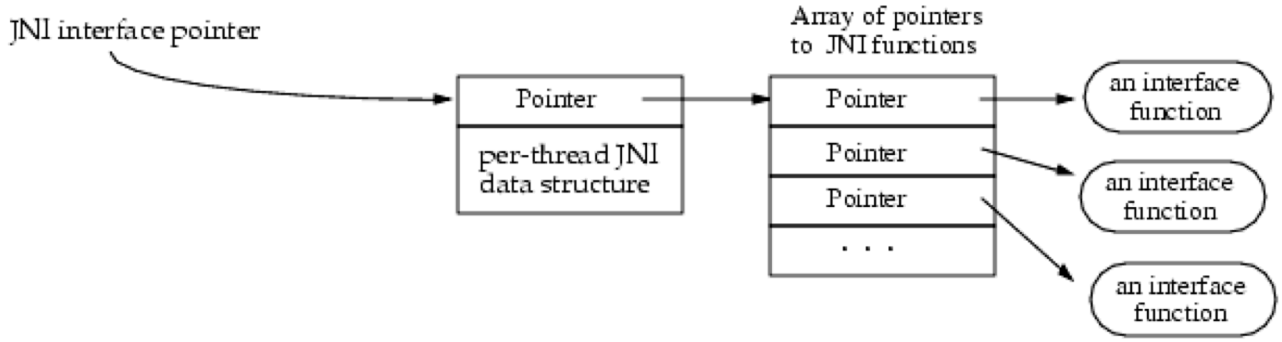


JNI Interface Pointer



- Organized like a C++ virtual function table.
  - Over 200 interface functions (JNI §4) for class and module operations, exceptions, global and local references, object operations, reflection, etc.
  - Separates the JNI name space from the native code, allowing for multiple versions of JNI function tables (e.g., for debugging and for production).
  - The JNI interface pointer has *thread scope*.

JNI Interface Pointer



- Organized like a C++ virtual function table.
  - Over 200 interface functions (JNI §4) for class and module operations, exceptions, global and local references, object operations, reflection, etc.
  - Separates the JNI name space from the native code, allowing for multiple versions of JNI function tables (e.g., for debugging and for production).
  - The JNI interface pointer has *thread scope*.
- Primitive types are copied, but arbitrary Java objects are passed by reference.
  - This has implications for the JVM's garbage collector.