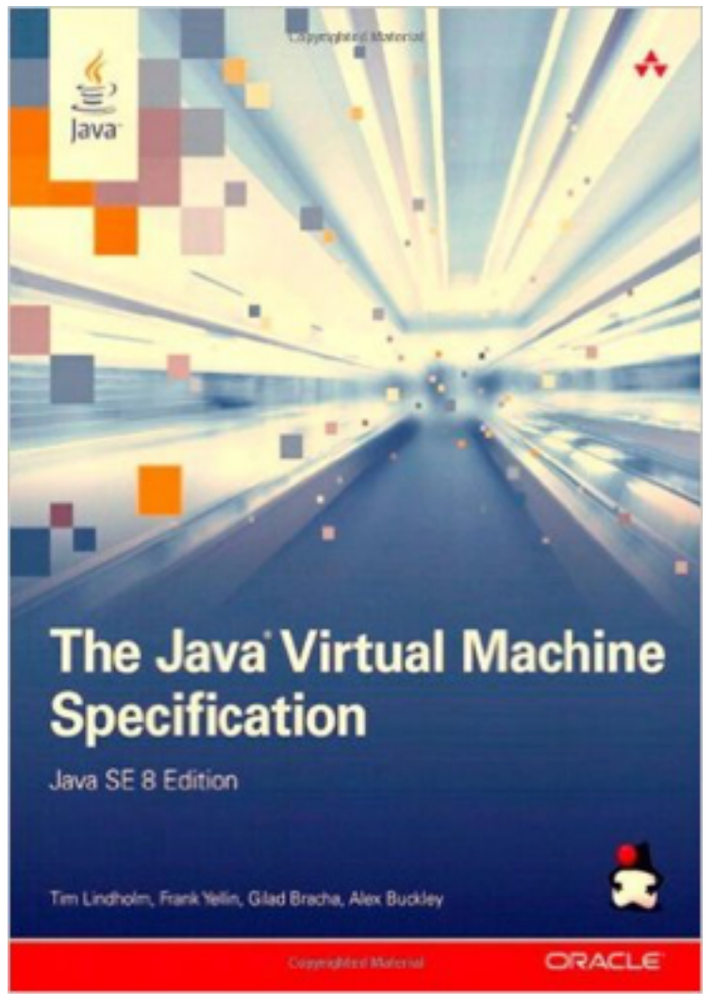
3. Anatomy of a Virtual Machine

Major components inside a virtual machine

1. Maintenance of **state**
    - Explicit state: Memory, registers
    - Implicit state: Translated code, working areas, stateful I/O, …
2. **Execution**
    - Executing the “virtual machine instructions” with side-effects on state
3. **Libraries and support code** to supplied required functionality
    - E.g., the equivalent of OS traps to VM-specific/language-specific functions
4. **External interfaces**
    - OS, foreign function interfaces (FFI) to call user libraries (in other languages)

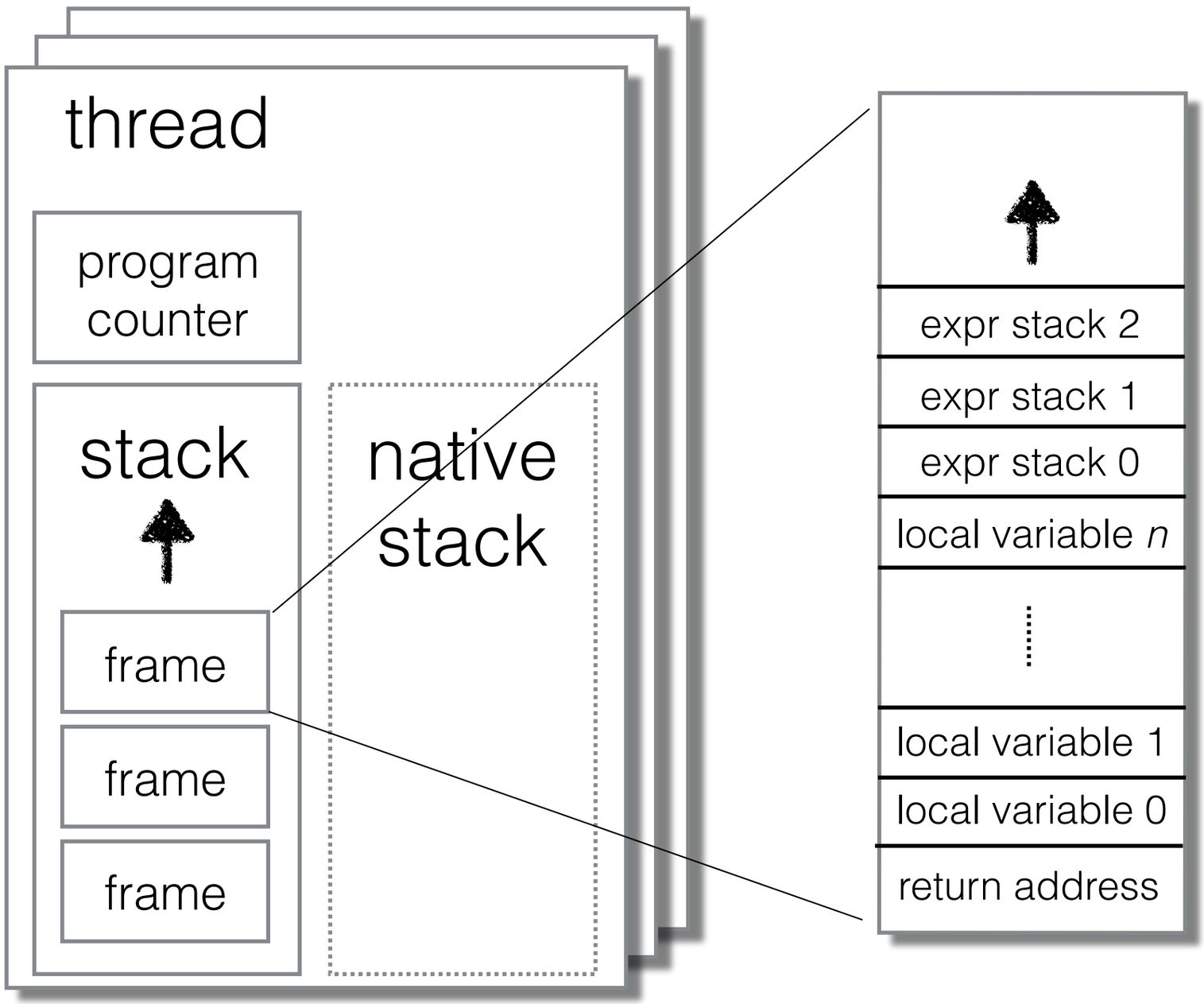
Example: the Specification of the Java™ Virtual Machine

- Overview of abstract state
- Bytecodes
- Libraries/primitives
- JNI — Java Native Interface



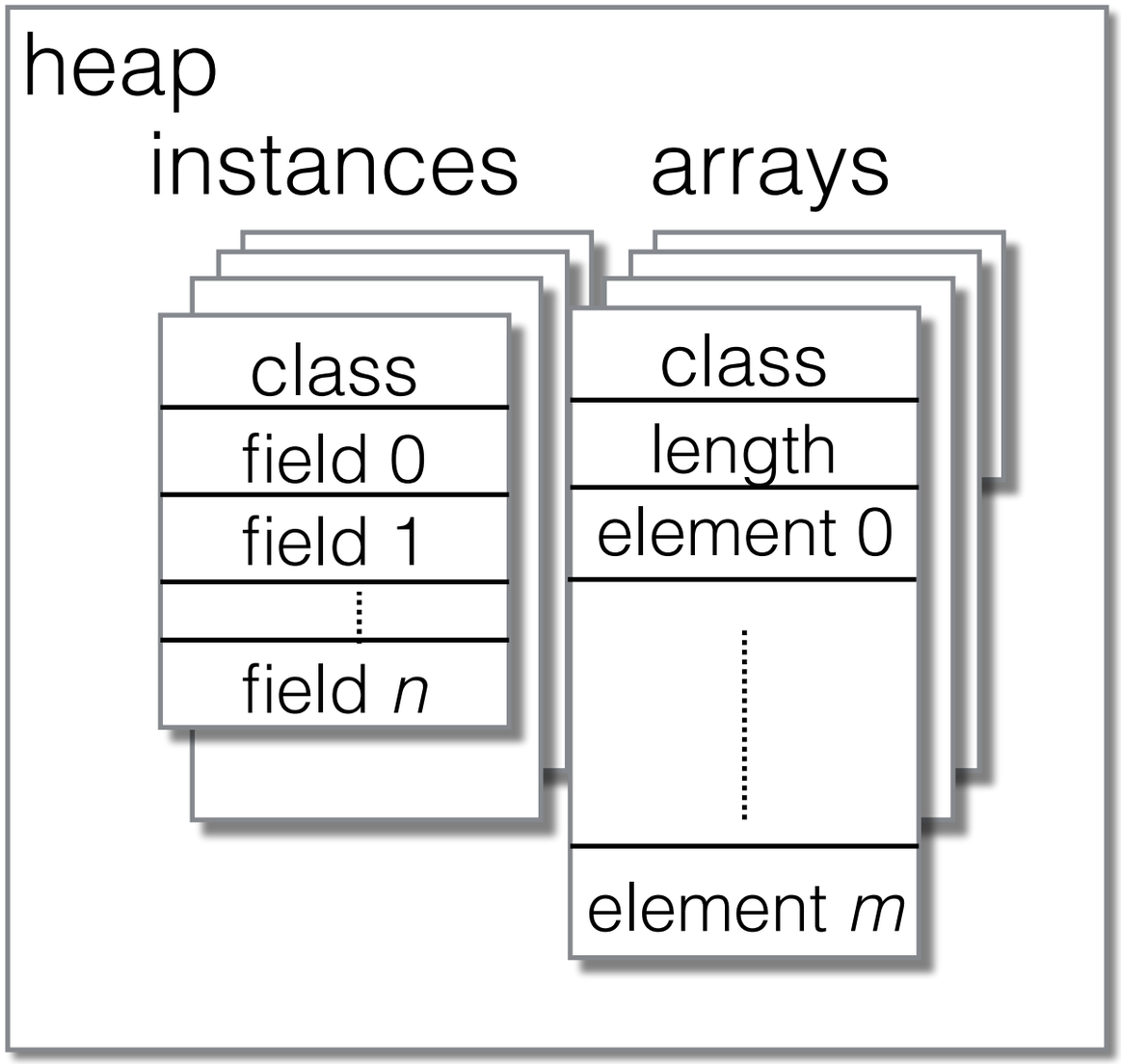
https://docs.oracle.com/javase/specs/

JVM thread state



A program counter or return address is a triple: (current class, current method, bytecode offset); *or* a native code PC

JVM object heap



Object fields

- Each field (or array element) contains either

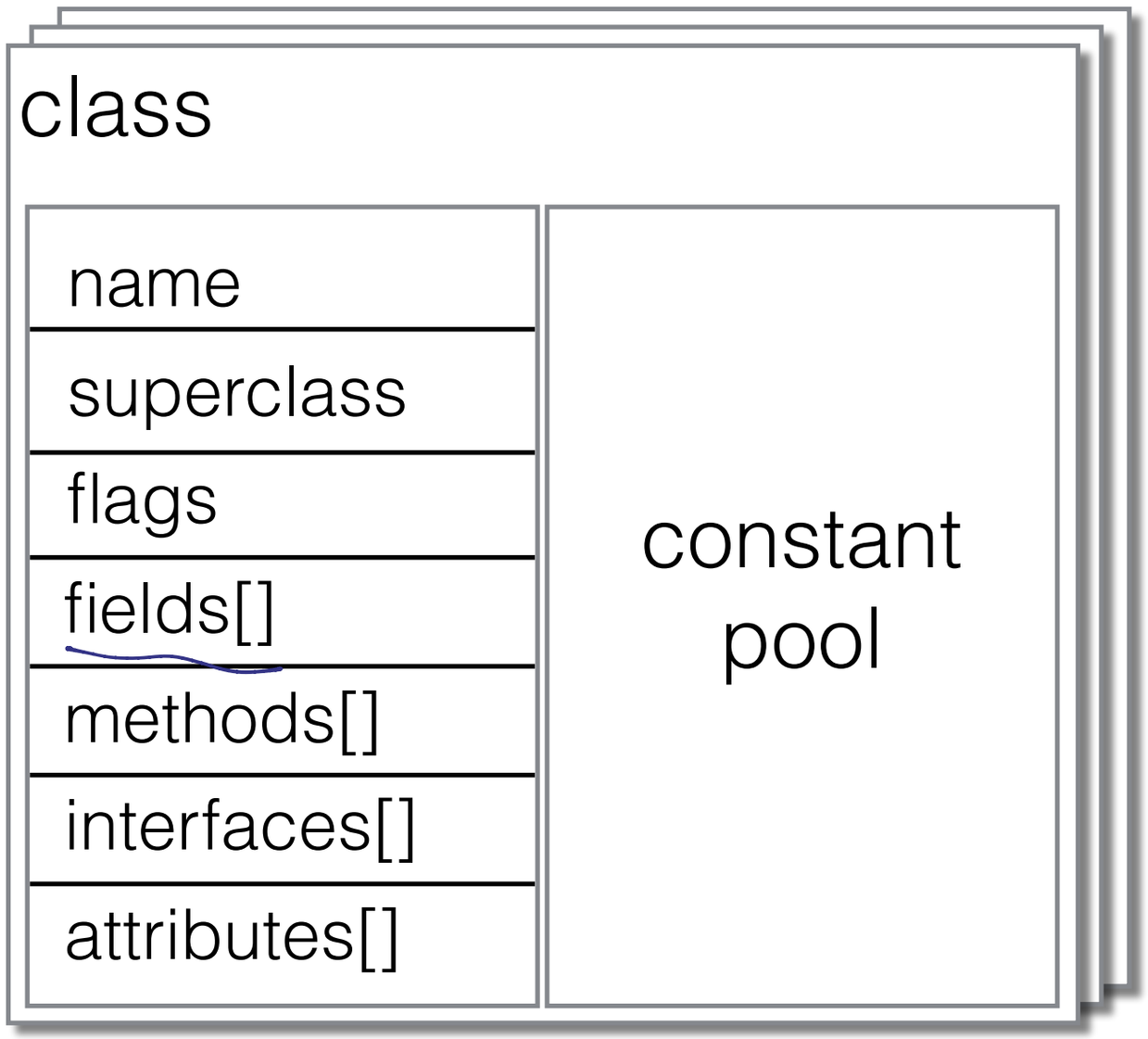
1. A reference to another object (size is implementation-dependent), or

2. A primitive value, one of:

   i) 8/16/32/64-bit integer

   ii) 32/64-bit IEEE 754 floating-point

JVM classes



Each field is an index into the *constant pool* of that class (part of the class file).

The constant pool contains:

- Constants (numbers, strings)
- Structures describing fields, methods, interfaces and attributes

Field spec in class files

- From the JVM spec (§4.5):

```
field_info {
    u2 access_flags; // public,private,protected,
static,final,volatile,transient,synthetic,enum
    u2 name_index;
    u2 descriptor_index; // see next slide
    u2 attributes_count;
    attribute_info attributes[attributes_count];
                                  // e.g., ConstantValue
}
```

Index fields are indices into the constant pool.

Field and method descriptors

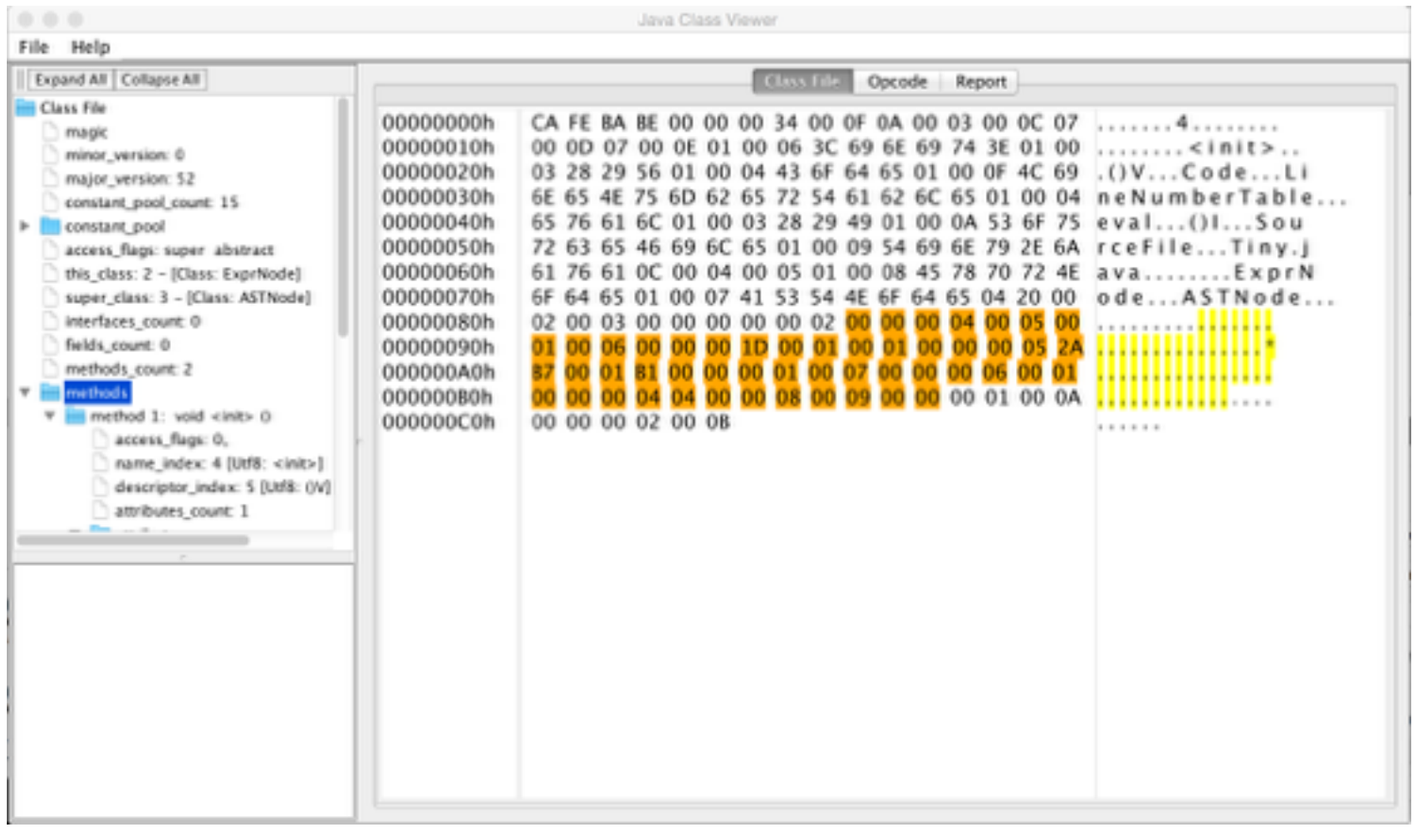
- Field descriptors describe field types
  - Base types: B - byte; C - char; D - double; F - float; I - int; J - long; S - short; Z - boolean
  - Object types: L<*classname*>;
  - Array types [*ComponentType*
- Method descriptors describe a method signature:
  - (*ParameterDescriptor**) *ReturnDescriptor*
    where each is a field type, or V (void) for return.
  - Example:
    Object mymethod(int i, double d, Thread t) has descriptor
    (IDLjava/lang/Thread;)Ljava/lang/Object;

Methods in class files

```
method_info {
   u2 access_flags; // public,private,...
   u2 name_index;
   u2 descriptor_index;
   u2 attributes_count;
   attribute_info attributes[attributes_count];
                                    // e.g., Code
}
```

Exploring class files

- On your own, explore class file structure interactively using the Java Class Viewer by Amos Shi (http://www.codeproject.com/Articles/35915/Java-Class-Viewer).



Bytecodes

- Code of a method is held in its *Code* attribute:

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack; // max number of words on the operand stack
    u2 max_locals; // number of local vars
    u4 code_length;
    u1 code[code_length]; // bytecodes live here
    u2 exception_table_length;
    {   u2 start_pc;  // if catch_type is thrown between start_pc and end_pc,
        u2 end_pc;   // execute handler at handler_pc
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count]; // line numbers, stack maps,…
}
```

The bytecode ISA

- Arithmetic
- Memory (locals, fields, statics, arrays): access and object creation, synchronization
- Stack manipulation
- Type checking and conversion
- Control (comparisons and branches, method call and return, exceptions)

(We will look at this in some detail later.)

Primitives and intrinsics

Some of the methods required in the Java language can either not be expressed in Java, or would be too inefficient

Examples:
Object getClass()
Object clone()
Object hashCode() /
    System identityHashCode(java.lang.Object)

The JVM must provide an implementation of these.

Native libraries

Abstract the OS below

- Typically for platform-dependent code (i.e., the platform *below* the JVM — POSIX/Win32/…, etc.)

Example:

```
package java.io;
public class RandomAccessFile implements DataOutput, DataInput {
...
 public native void open(String name, boolean writeable)
             throws IOException;
 public native int readBytes(byte[] b, int off, int len)
             throws IOException;
 public native void writeBytes(byte[] b, int off, int len)
             throws IOException;
 public native long getFilePointer() throws IOException;
 public native void seek(long pos) throws IOException;
 public native long length() throws IOException;
 public native void close() throws IOException;
}
```

- Also typically written in the JVM implementation language

JNI — the Java Native Interface

- This allows programmers to call out (or call back) from external libraries (written in some other, compiled language, such as C)
- Too much detail to be covered here (it has a book of its own!)…summary:

1. Declare `native` methods in your class
2. Generate a C header file using javah
3. Include `<jni.h>` and the generated header in your C code
4. Write wrappers in C for the functions declared in (1)
5. Compiled your C into a library
6. Load it in Java using `System.loadLibrary(String)`
7. Call away!

- Also includes ways to access C data, and for C to call into to the JVM (to access Java objects, create instances, etc.)

C always breaks almost everything. JNI uses another process to isolate the possible
d or from the C. in order to figure out which side's error.