JHI Design Notes

What it is

The **Java HDF Interface (JHI)** is a Java package (*hdf.hdflib*) that ``wraps'' the HDF-4 library*. The JHI may be used by any Java application that needs to access HDF-4 files. This product cannot be used in most network browsers because it accesses the local disk using native code.

Note: The JHI does not support HDF-5. See the JHI5.

A central part of the JHI is the Java class *hdf.hdflib.HDFLibrary*. The *HDFLibrary* class calls the standard (*i.e.*, `native' code) HDF library, with native methods for most of the HDF functions.

It is extremely important to emphasize that *this package is not a pure Java implementation of the HDF library.* The JHI calls the same HDF library that is used by C or FORTRAN programs.

The Java HDF Interface consists of Java classes and dynamically linked native libraries. The Java classes declare *static native* methods, and the library contains C functions which implement the native methods. The C functions call the standard HDF library, which is linked as part of the same library on most platforms.

Intended purpose

The Java HDF Interface is intended to be the standard interface for accessing HDF-4 files from Java programs. The JHI is a foundation upon which application classes can be built. All classes that use this interface package should interoperate easily on any platform that has HDF-4 installed.

It is unlikely that Java programs will want to directly call the HDF library. More likely, data will be represented by Java classes that meet the needs of the application. These classes can implement methods to store and retrieve data from HDF files using the Java HDF library.

Important Changes

Very Important Change: Version 3.0 (and above) of the JHI packages all HDF library

calls as "hdf.hdflib", note that the "ncsa" has been removed.
Source code which used earlier versions of the JHI should be changed to
reflect this new implementation.

Other changes include:

- Support for Macintosh and PC path names, e.g., in *Hopen()*.
- A separate distribution of the JHI. This contains everything from the full source distribution, omitting all the packages not used by the JHI.
- The **HDFArray** class was extended to support more types of Java arrays, including arrays of any sub-class of **Number** (**Integer**, **Float**, etc.) and arrays of **String**.

How to use it

For example, the HDF library had the function **Hopen** to open an HDF file. The Java interface is the class *hdf.hdflib.HDFLibrary*, which has a method:

```
static native int Hopen(String filename, int access);
```

The native method is implemented in C using the Java Native Method Interface (JNI). This is written something like the following:

```
JNIEXPORT jint JNICALL Java_hdf_hdflib_HDFLibrary_Hopen (JNIEnv *env, jclass class, jstring hdfFile, jint
access) { /* ... */ /* call the HDF library */ retVal = Hopen((char *)file, access, 0); /* ... */ }
```

This C function calls the HDF library and returns the result appropriately.

There is one native method for each HDF entry point (several hundred in all), which are compiled with the HDF library into a dynamically loaded library (*libhdf*). Note that this library must be built for each platform.To call the HDF `*Hopen*' function, a Java program would import the package '*hdf.hdflib.**', and then invoke the '*Hopen*' method on the object '*HDFLibrary*'. The Java program would look something like this:

```
import hdf.hdflib.*; { /* ... */ HDFLibrary.Hopen("myFile.hdf", access); /* ... */ }
```

The *HDFLibrary* class automatically loads the HDF-4 library with the native method implementations and the HDF-4 library.

Technical notes

The following is a list of a few important technical notes of the JHI. For more details, please read JHI Design Notes.

Very Important Change: Please note that Version 2.6 (and above) of the JHI declares all HDF library calls as "static native" methods, and loads the library with

a 'static' constructor. Source code which used earlier versions of the JHI should be changed to reflect this new, more efficient implementation.

- **Data conversion and copying.** The Java HDF Interface must translate data between C data types and Java data types. For instance, when a Java program reads a two dimensional array of floating point numbers (**float [][]**), the HDF native library actually returns an array of bytes. The Java HDF Interface converts this to the correct Java object, and returns that to the calling program. This process uses the Java Core Reflection package to discover the type of the array, and then calls native code routines to convert the bytes from native (C) order to the correct Java object(s). This data conversion is platform specific and clearly adds overhead to the program.
- **The JHI Interface to HDF.** The Java HDF interface follows the HDF C interface as closely as possible. However, Java does not support pass-by-reference parameters, so all parameters that must be returned to the caller require special treatment for Java. In general, such parameters are passed as elements of an array. For example, to return an integer parameter, the Java native method would be declared to use an integer array. For instance, the C function

```
void foo( int inVar /* IN */, int *outVar /* OUT */ )
```

would be rendered in Java as:

```
static public native void foo( int inVar, int []outVar )
```

where the value of 'outVar' would be returned as 'outVar[0]'.

- **Exceptions.** The JHI declares all the methods of HDF to throw Java exceptions of type `*HDFLibraryException*'. Errors returned by the native HDF library will be detected and the appropriate *Exception* raised, just as from regular Java methods. There is no need for the Java program to analyze HDF library error codes or return values. However, in this release many HDF library errors are not detected as exceptions, so exceptions will not be raised except when opening a file. Error handling will be completed in a future release.

JHI Design Notes

Important Note

This documentation has not been updated to reflect changes since 1997.

Most of the information is correct, but please see the release notes for important changes

07/06/2016

Abstract

Applications access Hierarchical Data Format (HDF) files through the HDF library code. In the absence of a Java(TM) implementation of the HDF library (an extremely remote contingency), Java applications that need to directly access data in HDF files require a Java interface to the native code HDF library. This document describes the design of the Java interface for the HDF library.

1. Introduction

The current release of the Hierachical Data Format API (HDF 4) is not a simple or consistent object model. It consists of a dozen or so ``Interfaces'', with different models of data and data access. This reflects the history of development, as new features have been introduced while older features have been maintained for compatibility. The size and complexity of HDF also reflects the diversity of its uses and users. Different communities and applications use particular parts of the HDF API. In this sense, HDF represents the ``union'' of several data models.

While the different data models inherent in the HDF API are related both conceptually and in implementation, there is not really an inheritance relationship between them, nor a formal object model. Also, there are significant inconsistencies across the different data models, *e.g.*, in the operations supported, arguments required and the types of errors returned.

This document describes a release of the ``official'' Java interface to HDF.

Because the HDF API is not object oriented, creating Java classes to access the HDF library is not trivial. It is first necessary to create an object model for HDF, which might be done in many ways. Further, different communities or applications will need appropriate object models, *e.g.*, for imagery or multi-dimensional grids. It is not always easy to discern the border between the abstractions HDF should provide and what should be left to the application program. This document describes a standard set of Java classes to access HDF files, closely modelling the HDF API. This HDF Java Interface forms a solid foundation for constructing and--more importantly--sharing problem specific data models, as part of the HDF distribution or from other parties.

1.1 Previous work

In previous work, we have implemented a Java based browser for HDF files, called the Java HDF Viewer (JHV). Java applets generally cannot access files or use native code libraries (such as HDF), so the JHV is implemented as a Java *application*, that is, as a free standing program, not a network loadable *applet*. The JHV application links to the standard HDF library, through the Java ``native code'' methods to read HDF files on local disk. The lastest release of the JHV uses the HDF Java Interface described here.

The JHV has pioneered the implementation of a Java application that uses HDF. The JHV implements classes such as a tree to display the HDF objects in a file, and displays of metadata, annotations, data, and imagery from the HDF file. The JHV also supports subsetting and subsampling of data from the HDF file. In the future, the JHV will be able to transparently access remote HDF files, perhaps using RMI or CORBA.

The initial implementation of the JHV implemented a different Java-to-C wrapper layer. In the initial implementation, only parts of the HDF API were included, specifically, routines needed by the JHV to inquire about metadata and to read data. No write functions were implemented, and many miscellaneous access functions were omitted because they were not needed.

1.2 The Java Development Kit (JDK)

The first implementation of the JHV was based on the JDK 1.0.2. This early release of the Java environment did not support Java-to-C interfaces very well. The interfacing mechanism was incomplete, poorly documented, and (most critically) not standard across platforms. Fortunately, many of the shortcomings of JDK 1.0.2 were addressed by major improvements in JDK 1.1.

The JDK 1.1 provides the *Java Native Interface*, which is standardized for ***all*** platforms. The JNI provides an orthogonal set of C functions to access and manipulate Java classes and objects. For example, it is possible to ``pin'' Java strings and one dimensional arrays in memory for use by C code. Thus, the wrapper layer can move data between the objects of the Java application and the variables of the HDF I/O library.

A second important addition to the JDK is the addition of the *Java Core Reflection* package. This standard set of Java classes provides the ability to completely discover the type and methods of Java objects and classes. Intended to support componentware (such as Java Beans(TM), this package proved crucial to implementing self-describing I/O, such as HDF has provided for many years.

Of particular importance to this effort was the **Array** class, which allows the discovery of the shape, size, and type of arbitrary Java array objects, and also supports the creation and manipulation of Java array objects. This package makes it possible to store and retrieve multi-dimensional Java array objects using HDF.

2. The Object Model

As noted, there are potentially many ways to model access to the data object of HDF. The model described here chooses to model *the HDF library itself*. Thus, the central object in this model is the ``HDF'' object, which has 300 some native methods, corresponding to all the operations provided by the HDF native code library. The purpose of this HDF library object is to provide the foundation for any other object model which wishes to use data in HDF files. It is likely that most Java applications will not directly use the HDF object; rather will use more abstract object models which themselves use HDF to manage and access storage. This is analogous to the role of HDF-EOS: programs create and manipulate HDF-EOS data objects, not the HDF representations underlying them.

2.1 The principle objects

[...need O diagram here]

The objects of the model are:

- **HDF** -- The HDF library, with a native method for each HDF entry point. This is a very simple object, it provides almost one-for-one methods for the operations and constants of the HDF API. As such, this object has a very large number of native methods, and practally nothing else.

- **HDFArray** -- Support for converting between arbitrary shaped java arrays and native code one dimensional arrays of bytes, and vice versa. This supports the **VOIDP** data type, which is crucial to HDF's ability to store and retrieve arbitrary data types. The **HDFArray** class defines two important methods:
  - *byteify()* -- given a multi-dimensional Java Array object, the equivalent C array is constructed as a one dimensional array of bytes.
  - *arrayify()* -- given a multi-dimensional array from C, construct the appropriate Java Array object.
- **HDFCompInfo** -- support for the HDF compression ``comp_info'' union. Different compression schemes require specific auxillary parameters, which are modelled by sub-classes of **HDFCompInfo**. An example subclass is:
  - **HDFJPEGCompInfo** -- The parameters for JPEG compression: ``quality'' and ``force_baseline''.

  Other compression (such as Run Length Encoding (RLE)) can be added by creating another sub-class of HDFCompInfo.
- **HDFCompModel** -- support for the HDF compression ``comp_model'' union.
- **HDFChunkInfo** -- support for the HDF chunking ``chunk_info'' union.
- **HDFException** -- error conditions from HDF operations. There are two main sub-classes, which may in turn be specialized to reflect specific error conditions.
  - **HDFLibraryException** -- represents errors reported by the HDF library code. The Java wrapper code must detect the error from the native library and raise an appropriate exception. Subclasses represent different error conditions, which may be caught and handled by Java applications. The exception may include an HDF error code, and other appropriate information, to be used by an exception handler.
- **HDFJavaException** -- represents errors in the Java interface to HDF, *e.g.*, run time exceptions such as out of memory or inability to load required Java classes. Subclasses represent different error conditions, which may be caught and handled by Java applications.

Our implementation of these classes makes it possible for Java classes to perform essentially any operation that is supported by the HDF library. The Java code to access HDF is very similar to Fortran or C -- a sequence of calls to the routines of the HDF interface. Of course, Java programs will generally want to encapsulate these code sequences in more abstract classes, which represent the data objects of interest.

2.2 Use of the HDF Java classes

The HDF Java Interface classes are intended to be used in two ways:

a. The **HDF** object can be sub-classed to provide the desired interface. For instance, the Java HDF Viewer models the data of an HDF file as a tree, with each node containing an abstract representation of the array, image, or annotation from the HDF file. The description of these objects could be modelled as subclasses of **HDF**. (This can be seen as effectively providing an easy way to extend and customize the HDF API--for better or worse.)

For instance, an ``SDS object'' could be a sub class of **HDF**, which provides methods to access a specific representation of the meta-data and data of an **SDS** object from an HDF file. The methods of the **SDS** object would use the appropriate native code methods of the **HDF** superclass to access the HDF file to construct the required representation.

a. Alternatively, any Java class can use the **HDF** class to obtain or create data in any way desired, in combination with any other code. For instance, a ``HDF-EOS Grid object'' could be implemented as a Java class with appropriate methods. The methods of the **HDF-EOS** class would instantiate HDF objects and invoke appropriate methods to store and retrieve data. In this model, HDF storage can be used in combination with other services in complex objects.

There have been many ideas for both general and application specific object models for data and images. The HDF Java classes described here are the necessary foundation that allows these kinds of models to be implemented quickly and portably using Java, with the ability to create and use data stored in HDF.

3. Implementation

An initial implementation of the HDF Java classes has been completed. The implementation required solution of a number of important problems that are of general interest to any effort to interface Java and C. These include:

1. Data transfer and translation between Java and C, particularly of arbitrary multidimensional arrays of numbers.
2. Performance issues, especially data copying
3. Garbage collection and memory management
4. Exceptions and errors, translating C error conditions into appropriate Java exceptions.

These issues are discussed in this section.

3.1 Data Transfer and Translation Issues

Many HDF functions return several data items in addition to a return value. That is, many of the parameters to the routines of the HDF library are ``OUT'' or ``IN/OUT'' parameters. In C this is expressed by call-by-reference semantics, that is, using pointers. Java passes all arguments by value, so special operations are required to return a value through a parameter. The C code must invoke operations on arrays, or invoke methods of the other types of objects.

A very important function of HDF is handling (possibly large) multi-dimensional arrays of numbers, with accompanying metadata. The HDF library provides portable, self-describing data storage and access to this type of data. HDF achieves this through the use of untyped arrays (*i.e.*, C type ``void *'', meaning ``any'') and internally converts between portable data representations and the local native data representations. Java is, of course, strongly typed, so the type of all arguments must be specified in the interface or else discovered at run time. Also, the Java VM has its own portable data representations which must be converted to and from native data representations.

Like many C interfaces, the HDF library uses structures (actually unions) to pass related groups of parameters. For instance, HDF supports several types of compression. Each compression algorithm has specific parameters, for example, JPEG compression requires parameters to specify the ``quality'' and ``start_baseline''. In C these parameters are passed by in a *union*, which should contain the parameters appropriate to the selected compression algorithm. In Java, this concept can be modelled by a **CompInfo** class, with sub-classes for the specific cases, such as, **JPEGCompInfo**.

This data handling is, in fact, the heart of the HDF Java interface. All of these issues are handled by a combination of special Java classes and C-code that uses the standard Java Native Interface (JNI). The JNI provides standard functions for C programs to create, access, and manipulate Java objects. These functions are standard across all platforms, assuring that the C code is (in principle) portable to any JDK 1.7 implementation.

3.1.1 Pass-By-Reference Parameters

The HDF library uses ``pass-by-reference'' parameters to retrieve values from the data and metadata of an HDF file. For example, the HDF function **SDfileinfo** returns two items of metadata as integers (Figure 1).

The Java language provides classes to enable basic data types to be passed as Objects -- the class **Integer** wraps an **int**, and so on. These classes can be used to pass data to C native methods. The HDF function **SDfileinfo** can be declared as a Java native method, e.g.,

```
public native boolean SDfileinfo(int sdid, Integer fileinfo, Integer nglobalattr);
```

This method would be invoked in a natural way, as shown in Figure 2. The HDF library is not called directly by the Java code, a C interface is required. An implementation of the interface method is shown in Figure 3. The native method uses elements of the Java Native Interface (JNI) to call methods on the Java **Integer** objects. In this case, the code calls the constructor for the class **Integer**, initializing the object to the value read by the native HDF library. This effectively ``returns'' the value to the Java object that called the native method.

The JNI allows a C routine to invoke any method of any Java class, so Java objects passed to native methods can potentially be accessed and manipulated in many ways besides setting a value.

It is very important to note that the code in Figure 3 can (in principle) be used on any platform that supports JDK 1.7, because the JNI calls are a required part of the JDK 1.7 standard. This means that all the code shown here can be written once and used everywhere. Naturally, both the Java to C code in Figure 3 and the HDF library itself must be compiled and installed on each platform. But only one version of the C code should be required.

3.1.2 Arrays and Strings

HDF is specifically designed to handle multidimensional arrays of numeric data. The HDF interface is ``self describing'', able to handle arrays of up to 15 dimensions, with dimensions of any length, with elements of type **byte**, **short**, **int**, **float** or **double** (and other types).

To give a concrete example, consider the HDF function **SDreaddata**, an HDF operation to read multi-dimensional array of numbers. (Figure 4)

To access this routine from Java, a native method is declared, with appropriate parameters. In this example, the **start**, **stride**, and **edge** can be passed Java arrays. However, the ``OUT'' buffer may be any shape and type of array: up to 15 dimensions, of any size, with elements of type **byt e**, **short**, **int**, **float** or **double**. One way to deal with this is to declare the array to be type ``byte[]'', and force the Java program to interpret the returned bytes, for example, as a two dimensional array of floats (**float[][]**). The second way is to declare the array type ``Object'', and to have the code of the HDF Java Interface discover the type at run time. The HDF Java Interface provides both options, as shown in Figure 5.

The Java API in Figure 5 is intentionally very similar to the C API, and is invoked in a similar manner. Figure 6 shows an example of some Java code that calls this interface to read a two dimensional array of floating point numbers. This code is very similar to the equivalent C code.

As discussed in the previous section, the HDF library is not called directly by the Java object. Figure 7 gives a sketch of the implementation of the native method declared in Figure 5. This code uses several parts of the Java Native Interface. The ``IN'' arrays must be ``pinned'' in memory, because the Java VM might not store an array as a contiguous array of bytes, as expected by C. The JNI has a standard interface for this operation, in this case, *GetIntArrayElements*. The pinned array must be released after use, as shown in this example code. The data array is handled similarly, except that the array is written back to the Java array after it is filled in by the HDF library (note the ``JNI_COMMIT'' argument in the call to *ReleaseByteArrayElements*).

As explained above, the JNI interface is standard, so the C code is portable to any JDK 1.7 platform. However, the implementation of the ``pinning'', release, and write back are platform dependent. On some platforms, this may be implemented as simple pointer operations, on others, a data copy may be necessary. The semantics should be identical, but the computation time and memory use may differ across platforms.

Strings

Strings are handled analogously to the arrays shown in Figure 7. The JNI provides functions to ``pin'' a Java String object, returning an array of bytes for use by C. This operation must also convert from UTF (Java) to ASCII (C). The reverse operations are also provided, to convert a C string to a Java String, and to convert from ASCII to UTF. Most systems will presumably require a data copy to convert between Java and C strings.

Run Time Type Discovery for Multidimensional Arrays

In the example code shown in Figure 6, the class ``HDFArray'' encapsulates methods to discover the shape, size and number type of the array at run time, and to perform the appropriate transformations on the data. In this example, the **HDFArray** class is used to discover the type of the array at run time, to allocate an appropriate buffer (of type **byte[]**), and to perform the appropriate data transformations to convert from an array of bytes returned by HDF to the equivalent two dimensional array of floats object (**float[][]**) required by Java.

The **HDFArray** class has a private class called **ArrayDescriptor**, which illustrates how the *java.lang.reflect* package is used to discover the shape, size, and number type of an arbitrary array object. Given a Java array, a table is built containing a complete description of the array. This table is then used when needed to traverse the array, extracting or inserting data as appropriate, using data type specific methods. Figure 8 illustrates how this table can be constructed.

The **HDFArray** object has two main methods:

- *byteify()* which converts a Java array into a one dimensional array of bytes to be used by C (Figure 9), and
- *arrayify()* which copies from a one dimensional array of bytes into a Java array object (Figure 10).

It is important to point out that in these methods *the entire data array is copied at least once*.

The actual data copies are done on one dimensional arrays, using a call to one of several native methods written in C. There is one such native method for each basic type, **byte**, **short**, **int**, **float**, and **double**. This is necessary because the casting from bytes to longer number types and vice versa is machine dependent. Java code cannot and should not have machine dependent code in it. In any case, the necessary byte ordering and so on is usually handled by the C compiler, so only one version of these routines should be needed, which should do the right thing on each particular platform. An example of this C code is shown in Figure 11.

The **HDFArray** class and supporting methods provides a service that may be used by any Java application that needs to handle arbitrary arrays of numbers. There is nothing specific to HDF in the implementation of this class, it could be used by any Java application that needs to handle arrays.

3.1.3 C Structures and Unions

In some cases, HDF passes related sets of parameters as a C structure or union. In a Java program, this is represented as a class or related set of classes. To actually call the C library routine, the Java object must be converted into the appropriate C union. This conversion can only be done by the Java-to-C wrapper using the JNI (because Java code simply cannot create C unions). The C code copies the fields of the Java object, one by one, into the fields of the C structure. Similar code can be used to populate a Java object from a C structure.

In the case of a C *union*, the C code must convert between one of several subclasses and one of the alternative C structures. The conversion code consists of a *switch* statement, with cases for each possible type of conversion. The C union is filled in by an element by element copy from the fields of the Java object to the fields of the C union.

For a C interface that uses such structures heavily, the Java-to-C wrapper code will contain many such sequences and will be quite voluminous. Also, C code may declare arbitrarily large and complex structures, which may contain arrays, structures, and unions (for an example, see the **HDF _CHUNK_DEF** used by the latest release of HDF). It is perfectly possible to express such data structures with appropriate Java classes, and even to deal with structures which contain C pointers, but the code to translate between Java and C could be long and difficult to write.

In HDF, a prominent example of this type of parameter passing is the *comp_info* structure (Figure 12). HDF supports several types of data compression, and each method may have algorithm specific input parameters. The parameters are encapsulated in the *comp_info* union, which is used to pass the appropriate type of information given the selected compression method.

This union can be expressed as a Java class, such as **HDFCompInfo** (Figure 13), which is specialized by subclasses for each type of compression supported, for example, the input parameters for JPEG compression could be represented in the class **HDFJPEGCompInfo**, a subclass of the generic **HDFCompInfo** class (Figure 14). The generic class is used as an argument to methods, including native methods such as:

public native boolean GRsetcompress(int ri_id, int comp_type, HDFCompInfo c_info);Depending on the type of compression specified, the Java program must set *comp_type* to the appropriate value and create and initialize an object of the appropriate subclass of **CompInfo**. These objects would then be passed to the **GRsetcompress** native method.

The native method must discover the type of compression requested and then create the appropriate *comp_info* union. Figure 15 shows a sketch of a subroutine which implements a simple case. The example takes as input a Java object and an uninitialized C union, and copies the values from the fields of the Java object to the fields of the C union. (Note that the example code uses JNI routines to access static fields of the Java object.) This subroutine shows a very simple case, which could be extended to handle many alternatives, and to copy different types of data, including nested structures and unions.

The subroutine may be called by any native method that needs to use a **CompInfo** object. Figure 16 shows an example of how this might be called from the **GRsetcompress** native method.

Clearly the code to manipulate C structures is awkward and inefficient, especially if reasonable error checking is added. It should be noted that this is scarcely an unprecedented problem: the Fortran interface to HDF has had to implement similar translations, as Fortran cannot directly implement many C data structures. Aside from changing the HDF API, there is no alternative to this kind of translation when attempting to provide multi-language support.

3.2 Performance

All the data conversions discussed above involve data copying. As far as I can determine, these copies are unavoidable. In fact, the majority of the data read or written through the Java HDF API is copied by the Java-to-C code for one reason or another. The actual amount of copying that occurs may vary from platform to platform, as the JNI operations may be implemented through additional (albeit hidden) data copies on some architectures (*e.g.*, to assure correct alignment or byte order between the Java VM and the native machine.)

For large datasets, this data copying is a serious performance problem. There are three major concerns:

1. The processor is occupied doing useless work. Regardless of how ``efficiently'' this is done, data copying is still not a good use of CPU time.
2. Memory to memory copies impact the whole system:
    a. uses up large amounts of memory, as the data is in memory at least twice.
    b. floods the memory bus
    c. flushes memory caches
3. The Java VM dynamic memory management system may not support large memory operations especially well. For instance, it may be necessary to increase the Java VM heap size, and to manually run the garbage collector. These contingencies may impact the overall performance of the Java VM. Also, these effects may be platform and VM specific, producing undesirably platform specific behavior.

It is important to note that there are two key reasons why a data copy is needed:

1. to defeat Java type checking (*i.e.*, to convert from bytes and pointers to objects and vice versa).
2. to assume correct data representation for the Java VM or native code, *i.e.*, native byte order.

The first requirement is a penalty inherent in Java's tight type system, which would be required for translating in and out of any strongly typed language. The second requirement is a penalty inherent in the Java portable Virtual Machine, and would be encountered by any completely portable data representation. Thus, a data copy is a direct and unavoidable consequence of two of the crucial assets of the Java environment. (Ironically, the HDF library itself contains code to perform exactly these functions--sometimes this will be an unfortunate duplication of effort.)

It should be noted that the JNI support code may itself contain data copies on some platforms. These methods are charged with doing whatever is necessary to convert from the internal Java VM representation of data to the correct form for C, and vice versa. In some cases, this can be accomplished with simple and fast pointer operations, but on some architectures it may be necessary to copy the data on each conversions. Thus, there may be even more copies than directly specified by the application, although this will vary across platforms.

3.3 Garbage Collection

The Java virtual machine manages memory, allocating objects when they are created, and garbage collecting unused memory when objects are unused. The data copying described in this paper uses large amounts of memory, as **Array** objects are created and destroyed. The Java language does not allow the program to directly manage memory, so the allocation and reclamation must be left in the hands of the Java Runtime system. When the objects are large and many operations are performed, Java may expend huge amounts of memory, in ways that are not necessarily apparent to the application programmer.

For instance, the **HDFArray** class traverses and creates multidimensional arrays. In Java, an array is an array of objects, a multidimensional array is an array of array objects. The methods of the **HDFArray** class create and destroy many temporary objects as they traverse a single multidimensional array. The result may be that the memory to store the temporary objects amounts to many times that total size of the data.

It is crucial that this memory is reclaimed promptly, so the Java garbage collector should run as the large arrays are processed. There are three options here:

1. manually invoke garbage collection at strategic places in the HDF Java wrapper code, such as at the end of a memory intensive method.
2. create and start a separate Java thread to periodically invoke the garbage collector, and
3. create an alternative garbage collector to install in the Java runtime.

The HDF Java interface will experiment with a combination of the first two possibilities. The third possibility is being investigated, but does not seem feasible.

While the garbage collector is running, less useful computation is accomplished. This overhead appears to be unavoidable. The Java memory management model is designed to be simple, secure, and portable. This works acceptably for reasonable numbers of reasonable sized objects, but becomes a performance problem for data intensive programs handling large amounts of memory. So again, this work has shown how important positive assets of Java have inherent performance consequences.

3.4 Exceptions

The Java language provides a clean, state of the art, mechanism for raising and handling exceptions. C has no such mechanism, but the JNI provides access to the Java model. This support allows the Java-to-C code to detect errors reported by the native code library, and then to create and raise appropriate exceptions. This is actually a fairly elegant solution, allowing the Java code to be written in a natural style, despite the fact that native code is involved.

Implementation requires two steps:

1. a model of the exception conditions must be constructed, defining the exceptions that may occur, and what they mean.
2. implement code to detect errors from the native code, and then raise the appropriate exception.

The HDF Java API defines an exception model with two main subclasses of exceptions,

1. **HDFLibraryException** -- exceptions that indicate error conditions reported by the native HDF library.
2. **HDFJavaException** -- Exceptions that indicate errors in the Java HDF wrapper code itself.

The former is basically a representation of the error codes from HDF. This exception class can be subclassed to capture natural structure in the HDF error conditions, although this has not yet been done. The latter errors may occur in the JNI calls and the code that implements the data translation and copying. In the future, this subclass, too, may be subclassed to reflect important categories of errors. Figure 17 shows a sketch of the **HDFLibraryException** class.

Any native method that will raise an exception will need to be declared, for instance,

```
public native int Hopen(String filename, int access) throws HDFException ;
```

which is exactly the same for a native method as for any other.

The code to implement the native method must include C code to test for errors, and then call JNI methods to raise an exception. Figure 18 shows a sketch of the code to implement the **Hopen** native method. Both of the classes of HDF exception are illustrated in this example.

If the JNI call to **GetStringUTFChars** fails-- if a NULL String is used or due to insufficient memory-- an **HDFJavaException** object is created and thrown. In a few cases, the JNI itself may raise an exception. If so, then this error may be caught and cleared by the C code, in order to raise a different exception.

The second exception may occur if the call to the HDF library fails--if, for instance, the file does not exist and is being opened for reading. The HDF library call will return **FAIL**, and, if applicable, there will be error information which can be retrieved by the **HEvalue** call. (See the HDF manuals for more information about HDF error handling.)

The example in Figure 18 shows one way to deal with these errors. If the **Hopen** call fails, and an error code is set, then an **HDFLibraryException** is thrown, setting the exception message to the standard message returned by the **HEstring** call.

Figure 19 shows an example of Java code that uses the native method shown in this example. The invocation of the **Hopen** is enclosed in a ``try/catch'' block, and the code in the ``catch'' block is executed if one of the exceptions is raised. In the example in Figure 19, if the file ``nosuchfile'' does not exist, then an exception is raised and the message

```
   nosuchfile: HDFLibraryException: Error opening file
```

is printed.

The example shown here gives a simple sketch of what can be done with the Java exception handling and JNI. These mechanisms are very flexible and powerful, so there are many possibilities which have yet to be explored in the HDF Java Interface. Future extensions may include:

1. A more elaborate exception model, with more subclasses of Exceptions reflecting different categories of errors. This is especially attractive if some classes of errors can be automatically handled by some programs, perhaps adjusting the parameters to fit the actual range, breaking up large requests into a series of smaller ones.
2. Perhaps add code to handle some exceptions in Java interface code itself.
3. More elaborate reporting, especially when HDF returns a stack of errors. In general, the error conditions from HDF could be analyzed and filtered, with the HDF Java Interface either masking some failures (*e.g.*, through retries or reasonable defaults) or raising very specific exceptions. This would reduce the need for the application itself to process the exceptions returned.

4. Summary and Conclusions

The implementation described in this paper shows that the JDK 1.1 contains most of the features needed to interface even to a complex, non-object oriented native library such as HDF. However, the HDF Java Interface is far from trivial, and it remains to be seen if it will be useful for many applications.

Performance is of particular concern for HDF and scientific computing in general. The performance issues discussed in this paper appear to be inevitable consequences of the technologies that provide the portability and security of Java. These penalties are not significant for many applications, but become very significant for data intensive computing, as is essential for scientific computing. It is important to realize that Just In Time compiler technology will have little impact on the data copying and memory management issues which are the crux of the problem for the Java HDF Interface.

Usage of the HDF Java Interface

The HDF Java Interface provides the basic access to HDF files, making it possible for Java applications to create, modify, and read data using the HDF library. This has already been used in our Java-based HDF Viewer, and in a collaborative version of the viewer. We are studying using Java servers to serve data from HDF files, using an extensible Web server such as Jigsaw and/or the Java Remote Method Invocation (RMI) client/server distributed object model. Ultimately, Java might be used to implement a Scientific Data Server, such as proposed in.

Another potential use for the HDF Java Interface is to provide a data exchange mechanism for Java applications. The Java Object Serialization package is designed for transporting and exchanging objects. However, it is not well suited for extremely large objects, and is not intended as a means for accessing objects in external storage. In particular, the Java Object Serialization provides no way to selectively read or write a small part of a large object, or to read or write particular objects out of a complex object. HDF is specifically designed for this case, is much better suited for accessing storage, especially for scientific data.

One use of the HDF Java Interface would be for Java applications which export important data by compactly and efficiently storing its representation using HDF. HDF's compression and chunking features may be very useful in this application. This external representation can then be transported as an HDF file to other platforms, taking advantage of HDF's portability. The stored representation can be accessed in different ways, perhaps using chunking and parallel I/O access.

Notably, the data can be read from HDF into objects of different classes than the original data. For instance, a sub-set (*e.g.*, a hyperslab from a multidimensional dataset) of the original dataset could be read into a Java object representing the data of interest. With the Java Object Serialization, the entire data array must be read into memory and then the subset may be extracted.

Consider, for instance, a simulation that produces a large multidimensional dataset for each of a series of time steps. The data for each time step would be represented as a complex Java object containing more elementary objects. This could be stored as an HDF file, with the data stored in arrays and tables.

A visualization program may wish to access a small part of this data, perhaps only part of the data array, or some of the variables. A Java application could define classes to represent the selected view of the data, which might be substantially different from the object model of the original simulation. The visualization application could read from HDF to create the objects needed, reading only the parts of the data needed.

The performance of Java may limit its use for some aspects of scientific computing. For instance, it is unlikely that Java would be useful for the inner loop of a large numeric computation. However, Java will be very useful for portable interfaces, remote data access, and for visualization. It is likely that there will be many hybrid environments that use Java as a framework for interactive computing, liking to specialized native code applications and libraries for efficient computation. Here again, the ability to use HDF will be valuable as a means to exchange data among programs written in different languages, as well as running on different networks.

Acknowledgements

This work was supported by Project Horizon, a cooperative agreement between the University of Illinois and NASA. (OLD URL: http://horizon.ncsa.uiuc.edu/horizon/).

Other funding was provided by the Digital Library Initiative at the University of Illinois. (http://dli.grainger.uiuc.edu) The UIUC DLI is a recipient of a grant in the NSF/DARPA/NASA/Digital Libraries Initiative.

Figures

```
intn SDfileinfo(int32 sd_id, int32 *ndatasets, int32
*nglobal_attr)
```

where the parameters are:

```
sd_id
          IN:
                The SD interface identifier returned from SDstart
 ndatasets
          OUT:
                Number of data sets in the file
 nglobal_attr
          OUT:
                Number of global attributes in the file
```

Figure 1. The HDF function SDfileinfo which had two pass by reference parameters.

```
       import hdf.*;

       HDF hdfinterface = new HDF();

       int sdsid;
       /* initialize the HDF library with appropriate calls
           to Hopen, SDcreate, etc. */

       /* the parameters to be read, initialized to -1 */
       Integer ndatasets = new Integer(-1);
       Integer nglobal_attr = new Integer(-1);

       boolean rval =
          hdfinterface.SDfileinfo(sdsid, ndatasets, nglobal_attr);

       /* rval has the success/failure code */
       System.out.println("ndatasets ="+ndatasets.intValue());
       System.out.println("nclobal_attr ="+nclobal_attr.intValue());
```

Figure 2. Java code to call the native method shown in Figure 1.

```
jint Java_hdf_HDFSDS_SDfileinfo__ILJava_lang_Integer_2LJava_...
( JNIEnv *env,
jobject obj, /* this */
jint sdid,
jobject ndatasets,  /* OUT: Integer */
jobject nglobalattr)  /* OUT: Integer */
{
int32 retVal;
int32  ndataset;
int32  nattr;
jmethodID jmi;
jclass jc;
int args[2];

        /* call HDF library, read into C variables */

        retVal = SDfileinfo((int32)sdid, &ndataset, &nattr);

        /* check for errors, raise exception (omitted) */

        if (retVal == FAIL)
            return JNI_FALSE;
        else {

            /* store values C variables int to Java Objects */
            jc = (*env)->FindClass(env, "java/lang/Integer");
            if (jc == NULL) {
                    return -1;
            }
            jmi = (*env)->GetMethodID (env, jc, "", "(I)V");
            if (jmi == NULL) {
                    return -1;
            }
            args[0] = ndataset;
            args[1] = 0;
            (*env)->CallVoidMethodV( env, ndatasets, jmi, args );
            args[0] = nattr;
            (*env)->CallVoidMethodV( env, nglobalattr, jmi, args );

            return JNI_TRUE;
        }
    }
}
```

Figure 3. The C code to implement the HDF SDfileinfo call: two integers are read from HDF, and the appropriate Java Integer objects are initialized to the returned values.

```
intn SDreaddata(int32 sds_id, int32 start[], int32 stride[],         int32 edge[], VOIDP buffer)
```

where the parameters are:

```
sds_id
      IN:
            The data set identifier returned from SDselect
 start
      IN:
            Array specifying the starting location in each
            dimension
 stride
      IN:
            Array specifying the number of values to skip along
            each dimension
 edge
      IN:
            Array specifying the number of values to read along
            each dimension
 buffer
      OUT:
            Buffer to store the data, sufficient space must be
            allocated by the caller

 Returns: SUCCEED (or 0) if successful and FAIL (or
          -1) otherwise.
```

Figure 4. The HDF function SDreaddata().

```
public native boolean SDreaddata(  int sdsid, int[] start,
                                   int[] stride, int[] count,
                                   byte[] data);

public boolean SDreaddata(  int sdsid, int[] start,
                            int[] stride, int[] count,
                            Object theData )
       {
           byte[] data;
           boolean rval;

           /* discover the shape, size, and type of the array */
           HDFArray theArray = new HDFArray(theData);

           /* allocate a buffer of bytes */
           data = theArray.emptyBytes();

           /* Call the native method above */
           rval= SDreaddata(  sdsid, start, stride, count, data);

           /* Convert the bytes into the appropriate Java objects */
           theData = theArray.arrayify( data );

           return rval;
       }
```

Figure 5. A Java interface for the The HDF function SDreaddata(). The second version discovers the shape of the array at run time.

```
import hdf.*;

HDF hdfinterface = new HDF();

int sdsid;
/* initialize the HDF library with appropriate calls
    to Hopen, SDcreate, etc. */

int start = new int[2];
int count = new int[2];

/* discover the shape of the array using appropriate
   HDF calls, fill in the start and count */

float f[][] = new float[100][50];

boolean rval =
   hdfinterface.SDreaddata(sdsid, start, null, count, f);

/* rval has the success/failure code */
/* f    has the data read from the file */
```

Figure 6. A sample of Java code to call the HDF function SDreaddata() using the interface in Figure 2.

```
#include &lthdf.h>
#include &ltjni.h>

jboolean Java_hdf_HDF_SDreaddata__I_3I_3I_3I_3B
( JNIEnv *env,  /* a la the JNI interface */
jobject obj,    /* a la the JNI interface */
jint sdsid,     /* IN: int */
jarray start,   /* IN: int[] */
jarray stride,  /* IN: int[] */
jarray count,   /* IN: int[] */
jarray data)    /* OUT: byte[], data for a multidimensional
                                array  */
{
  int32 retVal;
  int32 *strt;
  int32 *strd;
  int32 *cnt;
  char *d;

  /* most error checking omitted for brevity */
  strt = (*env)->GetIntArrayElements(env,start,NULL);
  if (stride != NULL) {
          strd = (*env)->GetIntArrayElements(env,stride,NULL);
  } else {
       strd = NULL;
  }
  cnt = (*env)->GetIntArrayElements(env,count,NULL);

  /* Important: check that 'data' is big enough ! */
  d = (*env)->GetByteArrayElements(env,data,NULL);

  /*
   *  Call the HDF library
   */
  retVal = SDreaddata((int32)sdsid, strt, strd, cnt, d);

  /* check for error from library, throw exception (omitted) */

  (*env)->ReleaseIntArrayElements(env,start,strt,JNI_ABORT);
  if (stride != NULL) {
       (*env)->ReleaseIntArrayElements(env,stride,strd,JNI_ABORT);
  }
  (*env)->ReleaseIntArrayElements(env,count,cnt,JNI_ABORT);

  if (retVal == FAIL) {
       /* don't write back if call failed */
       (*env)->ReleaseByteArrayElements(env,data,d,JNI_ABORT);
       return JNI_FALSE;
  } else {
       /* write back the data that was read:
           the bytes from C will be converted into a Java
           array of bytes (byte[]) */
       (*env)->ReleaseByteArrayElements(env,data,d,JNI_COMMIT);
       return JNI_TRUE;
  }
}
```

Figure 7. The Java-C interface code to call the HDF function SDreaddata() using the Java Native Method Interface.

```
package java.lang.reflect.*;
 package hdf;

 class ArrayDescriptor {

   /* some code omitted to save space */
   static int [] dimlen = null;
   static int [] bytetoindex = null;
   static int totalSize = 0;
   static char NT = ' ';  /*  must be B,S,I,L,F,D, else error */
   static int NTsize = 0;
   static int dims = 0;

   public ArrayDescriptor ( Object anArray ) {

       Class tc = anArray.getClass();
       if (tc.isArray() == false) {
           /* Raise exception: not an array */
           return;
       }

       /* parse the type descriptor to discover the
           shape of the array */
       String ss = tc.toString();
       int n = 6;
       dims = 0;
       while (n < ss.length()) {
           NT = ss.charAt(n);
           n++;
           if (NT == '[') {
               dims++;
           }
       }

       /*  must be B,S,I,L,F,D, else error */
       if (NT == 'B') {
           NTsize = 1;
       } else if (NT == 'S') {
           NTsize = 2;
       } else if ((NT == 'I') || (NT == 'F')) {
           NTsize = 4;
       } else if ((NT == 'J') || (NT == 'D')){
           NTsize = 8;
       } else {
           /* Raise exception:  not a numeric type */
           return;
       }

       /* fill in the table */
       dimlen = new int [dims+1];
       bytetoindex = new int [dims+1];

       Object o = anArray;
       objs[0] = o;
       dimlen[0]= 1;
       int i;
       for ( i = 1; i <= dims; i++) {
           dimlen[i] =
                 java.lang.reflect.Array.getLength((Object) o);
           o = java.lang.reflect.Array.get((Object) o,0);
       }

       int j;
       int dd;
       bytetoindex[dims] = NTsize;
       for ( i = dims; i >= 0; i--) {
           dd = NTsize;
           for (j = i; j < dims; j++) {
               dd *= dimlen[j + 1];
           }
           bytetoindex[i] = dd;
       }

       totalSize = bytetoindex[0];
```

```
   }

}
```

Figure 8. Java code to discover the shape and type of an arbitrary array using the *java.reflect.Array* package.

```
package hdf;
/*
 * flatten a Java array into a one-dimensional array of
 * bytes with native byte ordering.
 */

 public byte[] byteify(){

   /* check parameters, and if already is one dimension is
          a special case....omitted here */

   Object oo = _theArray;
   n = 0;  /* the current byte */
   int index = 0;
   int i;
   while ( n < _desc.totalSize ) {
       oo = _desc.objs[0];
       index = n / _desc.bytetoindex[0];
               index %= _desc.dimlen[0];
       for (i = 0 ; i < (_desc.dims); i++) {
           index = n / _desc.bytetoindex[i];
           index %= _desc.dimlen[i];

           if (index == _desc.currentindex[i]) {
               /* then use cached copy */
               oo = _desc.objs[i];
           } else {
               /* check range of index */
               if (index > (_desc.dimlen[i] - 1)) {
                   System.out.println("out of bounds?");
                   return null;
               }
               oo = java.lang.reflect.Array.get((Object)oo,index);
               _desc.currentindex[i] = index;
               _desc.objs[i] = oo;
           }
       }

       /* byte-ify */
       byte arow[];
       if (_desc.NT == 'F') {
           /*
            *  Call a C routine to copy the row
            */
           arow = floatToByte(0,_desc.dimlen[_desc.dims],
               (float [])_desc.objs[_desc.dims - 1]);
       } else if (_desc.NT == 'I') {
           /* other types are similar ... */
       }

       /*
        *  A second data copy here (is this necessary?)
        */
       System.arraycopy(arow,0,_barray,n,
            (_desc.dimlen[_desc.dims] * _desc.NTsize));
       n += _desc.bytetoindex[_desc.dims - 1];

   }

   /* error checks  omitted */
   return _barray;
}
```

Figure 9. Java code to convert an arbitrary array to a contiguous array of bytes a la C. This code uses the *java.reflect.Array* package.

```
  /* give an array of bytes, fill in the Java array */

 public Object arrayify(byte[] bytes){

   /* error checks omitted */
   _barray = bytes; /* hope that the bytes are correct.... */

   /* One dimensional array is special case -- omitted */

   Object oo = _theArray;
   int n = 0;  /* the current byte */
   int index = 0;
   int i;
   while ( n < _desc.totalSize ) {
       oo = _desc.objs[0];
       index = n / _desc.bytetoindex[0];
       index %= _desc.dimlen[0];
       for (i = 0 ; i < (_desc.dims); i++) {
           index = n / _desc.bytetoindex[i];
           index %= _desc.dimlen[i];

           if (index == _desc.currentindex[i]) {
               /* then use cached copy */
               oo = _desc.objs[i];
           } else {
               /* check range of index */
               if (index > (_desc.dimlen[i] - 1)) {
                   System.out.println("out of bounds?");
                   return null;
               }
               oo = java.lang.reflect.Array.get((Object) oo,index);
               _desc.currentindex[i] = index;
               _desc.objs[i] = oo;
           }
       }


       /* byte-ify */
       if (_desc.NT == 'F') {

           /*
            *  Call a C routine to copy the row
            */
           float arow[] = byteToFloat(n,_desc.dimlen[_desc.dims],
                            _barray);

           /*
            *  Insert new row in the array (this might cause
            *  a data copy in some implementations
            */
           java.lang.reflect.Array.set(_desc.objs[_desc.dims - 2] ,
               (_desc.currentindex[_desc.dims - 1]), (Object)arow);

           n += _desc.bytetoindex[_desc.dims - 1];
           _desc.currentindex[_desc.dims - 1]++;

       } else if (_desc.NT == 'I') {
           /* other types are similar... */
       }

   }

   /* error checks omitted */

   return _theArray;
 }
```

Figure 10. Java code to convert a contiguous array of bytes (a la C) into an appropriate Java array object. This code uses the *java.reflec t.Array* package.

```
      /* HDFArray uses C code to copy arrays by row */

public native byte[] intToByte( int start, int len,
                                int[] data);
public native byte[] shortToByte( int start, int len,
                                  short[] data);
public native byte[] floatToByte( int start, int len,
                                  float[] data);
public native byte[] longToByte( int start, int len,
                                 long[] data);
public native byte[] doubleToByte( int start, int len,
                                   double[] data);
```

```
   /*  The native method looks something like this: */

/* returns float [] */
jarray Java_hdf_HDFArray_byteToFloat__II_3B
( JNIEnv *env,
jobject obj, /* this */
jint start,
jint len,
jarray bdata)  /* IN: array of bytes */
{
  intn rval;
  char *bp;
  jbyte *barr;
  jarray rarray;
  int blen;
  jfloat *iarray;
  jfloat *iap;
  int ii;

  /* pin the Java byte array (the source) */

  if (bdata == NULL) {
      printf("Exception: bdata is NULL?\n");
      return NULL;
  }
  barr = (*env)->GetByteArrayElements(env,bdata,NULL);
  if (barr == NULL) {
      /* Exception: GetByteArrayElements failed? */
      return NULL;
  }

  blen = (*env)->GetArrayLength(env,bdata);
  if ((start < 0) ||
        ((start + (len*(sizeof(jfloat)))) > blen)) {
      /*  Exception: start or len is out of bounds? */
      return NULL;
  }

  bp = (char *)barr + start;

  /* allocate a new java float array (the destination) */
  rarray = (*env)->NewFloatArray(env,len);
  if (rarray == NULL) {
      /*  Exception: NewFloatArray failed? */
      return NULL;
  }

  iarray = (*env)->GetFloatArrayElements(env,rarray,NULL);
  if (iarray == NULL) {
      /* Exception: GetFloatArrayElements failed? */
      return NULL;
  }
```

```
/*
 *  Copy using C:  byte ordering, etc., is handled
 *  by the C compiler.
 */
iap = iarray;
for (ii = 0; ii < len; ii++) {
    *iap = *(jfloat *)bp;
    iap++;
    bp += sizeof(jfloat);
}

/*
 *  Write back the results
 */

(*env)->ReleaseFloatArrayElements(env,rarray,
          (jfloat *)iarray, JNI_COMMIT);
(*env)->ReleaseByteArrayElements(env,bdata,barr,JNI_COMMIT);

return rarray;
```

```
}
```

Figure 11. Example C code to copy from C to Java. This uses the Java Native Interface (JNI) to manipulate the Java array.

```
typedef union tag_comp_info
 {
     struct
      {
          intn    quality;
          intn    force_baseline;
      }
     jpeg;
     struct
      {
          int32   nt;
          intn    sign_ext;
          intn    fill_one;
          intn    start_bit;
          intn    bit_len;
      }
     nbit;
     struct
      {
          intn    skp_size;
      }
     skphuff;
     struct
      {
          intn    level;
      }
     deflate;
 }
comp_info;
```

Figure 12. The HDF ``comp_info'' union, used to pass parameters required by different compression algorithms. (From hcomp.h)

```
package hdf;


public class HDFCompInfo {
       public static int ctype;
       public HDFCompInfo() {
               ctype = HDFConstants.COMP_CODE_NONE;
       } ;

}
```

Figure 13. A Java class for generic compression information.

```
package hdf;


public class HDFJPEGCompInfo extends HDFCompInfo {

          /* Struct to contain information about how to compress */
          /* or decompress a JPEG encoded 24-bit image */

           static public int    quality;

           static public int    force_baseline;

       public HDFJPEGCompInfo() {
               ctype = HDFConstants.COMP_JPEG;
       }

}
```

Figure 14. A Java class encapsulating the input parameters used byt JPEG compression.

```
#include "hdf.h"
 #include "hfile.h"
 #include "hcomp.h"

 #include "jni.h"

 jboolean getCompInfo( JNIEnv *env, jobject obj, comp_info *cinf)
 {
 jfieldID jf;
 jclass jc;
 jint ctype;

   /* Read the compression type from the Java object */
   jc = (*env)->FindClass(env, "ncsa/hdf/HDFCompInfo");
   if (jc == NULL) {
       /* exception */
       return JNI_FALSE;
   }
   jf = (*env)->GetStaticFieldID(env, jc, "ctype", "I");
   if (jf == NULL) {
       /* exception */
       return JNI_FALSE;
   }
   ctype = (*env)->GetStaticIntField(env, jc, jf);

   /* based on the type, the sub class of the object
      is known, and so copy the fields.
      Only JPEG is shown. */
   switch(ctype) {
       case COMP_NONE:
       case COMP_RLE:
       case COMP_IMCOMP:
       default:
           /* omitted */
           break;

       case COMP_JPEG:

               /* Use the HDFJPEGCompInfo class, copy two fields */
           jc = (*env)->FindClass(env, "ncsa/hdf/HDFJPEGCompInfo");
           if (jc == NULL) {
               return JNI_FALSE;
           }

           jf = (*env)->GetStaticFieldID(env, jc, "quality", "I");
           if (jf == NULL) {
               return JNI_FALSE;
           }
           cinf->jpeg.quality =
                  (*env)->GetStaticIntField(env, jc, jf);

           jf =
            (*env)->GetStaticFieldID(env, jc, "force_baseline", "I");
           if (jf == NULL) {
               return JNI_FALSE;
           }
           cinf->jpeg.force_baseline =
                      (*env)->GetStaticIntField(env, jc, jf);

           break;
   }

   return JNI_TRUE;
}
```

Figure 15. A sketch of C-code to translate from a Java CompInfo object to a C comp_info union.

```
jboolean Java_hdf_HDFGR_GRsetcompress
 ( JNIEnv *env,
 jobject obj, /* this */
 jint ri_id,
 jint comp_type,
 jobject c_info) /* IN:  CompInfo */
 {
   intn rval;
   comp_info cinf;
   jboolean bval;

   /* Fill in the appropriate parts of the comp_info
      structure. */
   bval = getCompInfo(env,obj,&cinf);

   /* check for success... */

   /* call HDF */

   rval = GRsetcompress((int32) ri_id, (int32) comp_type,
               (comp_info *)&cinf);

   if (rval == FAIL) {
       /* check for errors */
       return JNI_FALSE;
   } else {
       return JNI_TRUE;
   }
 }
```

Figure 16. Example of a native method that uses a CompInfo object, and passes a comp_info union to HDF.

```
public class HDFLibraryException extends HDFException {

       int HDFerror;
       String msg;

       public HDFLibraryException() {
               HDFerror = 0;
       }

       public HDFLibraryException(String s) {
               msg = "HDFLibraryException: "+s;
       }

       public HDFLibraryException(int err) {
               HDFerror = err;
       }

       public String getMessage() {
               return msg;
       }
}
```

Figure 17. A simple implementation of the Java Class HDFLibraryException

```
jint Java_hdf_HDF_Hopen(
JNIEnv *env,
jobject obj, /* this */
jstring hdfFile,
jint access)
{
   char* file;
   int  retVal;
   int errval;
   jclass jc;

   file =(char *) (*env)->GetStringUTFChars(env,hdfFile,0);

   if (file == NULL) {
       /* JNI call failed */
       jc =
         (*env)->FindClass(env, "ncsa/hdf/HDFJavaException");
       if (jc == NULL) {
               return -1; /* exception is raised */
       }
       (*env)->ThrowNew(env,jc,"Hopen: GetStringUTFChars failed");
   }

   /* open HDF file specified by hdf_HDF_file */
   retVal = Hopen((char *)file, access, 0);
   (*env)->ReleaseStringUTFChars(env,hdfFile,file);

   if (retVal == FAIL) {
       /* check for HDF error */
       /* for now:  use top of exception stack:  idealy this
          should do the whole stack, and analyze the HDF error
          codes */
       errval = HEvalue(1);
       if (errval != DFE_NONE) {
           jc =
             (*env)->FindClass(env,"ncsa/hdf/HDFLibraryException");
           if (jc == NULL) {
                   return -1; /* fatal error is raised by JNI? */
           }
           (*env)->ThrowNew(env,jc,HEstring(errval));
       }
       return -1;
   }
   else {
       return retVal;
   }

}
```

Figure 18. Example of a native code method that raises exceptions.

```
       import hdf.*;

       HDF h = new HDF();

       String theFile = "nosuchfile";
       int hid = 0;
       try {
               hid = h.Hopen(theFile,HDFConstants.DFACC_RDONLY);
       } catch (Exception e) {
               /* do something to recover if possible.... */
               System.out.println(theFile+": "+e.getMessage());
       }
       System.out.println("Hopen returned: "+hid);
```

Figure 19. Example of Java code that calls a native method with exception handling.