NetCDF Java (version 2)

User's Manual

John Caron, Apr 26, 2001

Introduction

This is user documentation for the *ucar.nc2*, *ucar.nc2.dods*, and *ucar.ma2* java packages, also known as "NetCDF Java version 2" and "MultiArray verion 2" packages. These are a separate and logically unrelated Application Programmer's Interface (API) from Unidata's earlier *ucar.netcdf* and *ucar.multiarray* packages [1]. All of these packages are freely available and the source code is released under the Lesser GNU Public License (LGPL) [2].

The ucar.nc2 package is intended to be a straightforward Java interface to NetCDF files. It is built on the ucar.ma2 package, which is an abstraction for multidimensional arrays of primitive types [3].

1. NetCDF

A NetCDF file is a collection of dimensions, attributes, and variables. A *dimension* is a named array index. An *attribute* is a (key, value) pair. A *variable* is a multidimensional array backed by an I/O device. A variable also has a name, and a collection of dimensions and attributes. `NetcdfFile` is used for read-only NetCDF files, while `NetcdfFileWriteable` allows writing, and a `DODSNetcdfFile` allows read-only access to DODS files through the NetCDF API. Note there are no public constructors for `Dimension`, `Attribute` or `Variable`; the only way to create these is through `NetcdfFileWriteable`.

The semantics of NetCDF files are slightly different through this Java API than through the C, C++ or Fortran APIs that you may be used to. A NetCDF file can only have `Dimension`, `Attribute` and `Variable` objects added to it at creation time. Thus it is not possible to put a file into "define mode" to add new variables, dimensions, or attributes, or to change the value of an attribute, after the file has been created. However, you can change data values or append new data to existing variables after creation time.

The following is an overview of the important public interfaces of the ucar.nc2 classes. Consult the javadoc for more complete and recent details.

1.1 ucar.nc2.Dimension

A `Dimension` object contains a name and a length. If the `Dimension` is "unlimited", then the length can increase; otherwise it is immutable. The method `getCoordinateVariable()` returns the associated "coordinate variable" or null if none exists. A coordinate variable is defined as a `Variable` with the same name as the `Dimension`, whose single dimension is the `Dimension`, for example: `float lat(lat);`

```
public class Dimension {
  public String getName();
  public int getLength();
  public boolean isUnlimited();
  public Variable getCoordinateVariable(); // null if none
}
```

1.2 ucar.nc2.Attribute

An attribute is a (name, value) pair, where name is a `String`, and value is a `String`, `Number` or `Number[]`. Valid subclasses of `Number` in this context are `Byte`, `Double`, `Float`, `Integer`, or `Short`.

```
public class Attribute {
  public String getName();
  public Class getValueType(); // String or subclass of Number
  public boolean isString();
  public boolean isArray();
  public int getLength();      // 1 for scalars

  public Object getValue();
  public String getStringValue();
  public Number getNumericValue();
  public Number getNumericValue(int elem);
}
```

1.3 ucar.nc2.Variable

A `Variable` is a multidimensional array stored in a local file or remote network device (a scalar variable is a rank-0 array). It has a name and a collection of dimensions and attributes, as well as logically containing the data itself. The dimensions are returned in `getDimensions()` in order of most slowly varying first (leftmost index for Java and C programmers). The underlying array elements are always primitives, and may have Class types of `double.class, float.class, int.class, short.class, byte.class, or char.class`. (Note the absence of the `long` and `boolean` types: these are not supported in the underlying file format).

Data access is handled by calling `read()`, which returns the data in a memory-resident `Array` object. This `Array` has the same element type as the `Variable`, and the requested shape. A subsection is specified using the origin and shape parameters, which for each dimension must satisfy:

```
origin[ii] + shape[ii] <= Variable.shape[ii].
```

```
public class Variable implements ucar.ma2.IOArray {
  public String getName();

  public int getRank();           // rank of the array
  public int[] getShape();        // shape of the array
  public long getSize();          // total number of elements
  public Class getElementType();  // type of the array elements

  public boolean isUnlimited();   // iff a Dimension isUnlimited
  public boolean isCoordinateVariable();

  public ArrayList getDimensions();      // get all Dimensions
  public Dimension getDimension(int i);  // get the ith Dimension

  public Iterator getAttributeIterator();      // get all Attributes
  public Attribute findAttribute(String attName);
  public Attribute findAttributeIgnoreCase(String attName);

  // read into memory-resident Array
  public Array read() throws IOException; // read all data
  public Array read(int[] origin, int[] shape) throws IOException,
        InvalidRangeException;
}
```

1.4 ucar.nc2.VariableStandardized

A `VariableStandardized` is a subclass of `Variable` which implements *packed data* using `scale_factor` and `add_offset` attributes and *invalid data* using `valid_min, valid_max, valid_range`, `missing_data` or `_FillValue` attributes. To use, you simply pass the original `Variable` into the `VariableStandardized` constructor (an example of the *Decorator* design pattern).

If the `Variable` has the `scale_factor` and/or `add_offset` attributes, the data will automatically be converted to type `double` or `float` and computed using the formula:

```
new_value = scale_factor * old_value + add_offset
```

If the `Variable` has any of the invalid or missing data attributes, `hasMissing()` will return true. To test if a specific value is missing, call `isMissing()`.

```
public class VariableStandardized extends Variable {
  public VariableStandardized( Variable orgVar);
  public VariableStandardized( Variable orgVar, boolean useNaNs);

  public boolean hasMissing();
  public boolean isMissing( double val);
  …
}
```

There is a lower level interface if you need to distinguish between _FillValue, missing_value and valid_range attributes; see the javadoc for details. By default, hasMissing() is true if any of these attributes are used. You can modify this behavior in the constructor or by calling setInvalidDataIsMissing(), setFillDataIsMissing(), or setMissingValueIsMissing(). Data values of float or double NaN are considered missing data and will return true if called with isMissing(). (However hasMissing() will not detect if you are setting NaNs yourself). If you do not need to distinguish between _FillValue, missing_value and invalid, then set useNaNs = true in the constructor. When the Variable element type is float or double (or is set to double because its packed), then this sets isMissing() data values to NaNs, which makes further comparisions more efficient.

1.5 ucar.nc2.NetcdfFile

A `NetcdfFile` is an immutable (read-only) collection of dimensions, attributes, and variables:

```
public class NetcdfFile {
  public NetcdfFile(String filename) throws IOException;
  public NetcdfFile(URL url) throws IOException;

  public String getName();  // filename or URL pathname

  public Iterator getVariableIterator();       // get list of Variables
  public Variable findVariable(String varName); // find specific Variable

  public Iterator getDimensionIterator();      // get list of Dimensions
  public Dimension findDimension(String dimName); // get specific Dimension

  public Iterator getGlobalAttributeIterator(); //get list of global Attributes
  public Attribute findGlobalAttribute(String attName); //specific Attribute
  public Attribute findGlobalAttributeIgnoreCase(String attName);

  public String findAttValueIgnoreCase(Variable v, String attName,
       String defValue);
}
```

The `NetcdfFile(String filename)` constructor accesses local files, while the `NetcdfFile(URL url)` constructor uses the HTTP 1.1 protocol to access netCDF files that are stored on web servers. The URL should begin with *http*: for example: *http://www.unidata.ucar.edu/staff/caron/test/mydata.nc*. The efficiency of the remote access depends on how the data is accessed. Reading large contiguous regions of the file should generally be good, while skipping around the file and reading small amounts of data will be poor. In that case you would be better copying the file to a local drive, or putting the file into a DODS server.

The `findAttValueIgnoreCase` is a convenience method for `String`–valued attributes which returns the specified default value (`defValue`) if the attribute name is not found in the file. Specifying a null for the variable means to search global attributes, otherwise search for the attribute in the given variable.

1.6 ucar.nc2.NetcdfFileWriteable

A `NetcdfFileWriteable` allows a NetCDF file to be created or written to. Note this can only be done with local, not remote files.

Because of the limitations of the underlying implementation, a NetCDF file can only have dimensions, attributes and variables added to it at creation time. Thus, when a file is first opened, it in is "define mode" where these may added. Once `create()` is called, the dataset structure is immutable. After `create()` has been called you can then write the data values. See following example.

```
public class NetcdfFileWriteable extends NetcdfFile {
  // create new file
  public NetcdfFileWriteable();
```

```
  public void setName(String filename);

  // open existing file for writing data only
  public NetcdfFileWriteable(String filename) throws IOException;

  // add new Dimension
  public Dimension addDimension(String dimName, int dimLength);

  // add new global Attributes
  public void addGlobalAttribute(String attName, String value);
    // value must be of type Float, Double, Integer, Short or Byte
  public void addGlobalAttribute(String attName, Number value);
    // value must be 1D array of (double, float, int, short, char, byte)
  public void addGlobalAttribute(String attName, Object arrayValue);

  // add new Variable and variable Attributes
  public void addVariable(String varName, Class varType, Dimension[] varShape);
  public void addVariableAttribute(String varName,String attName,String value);
    // value must be of type Float, Double, Integer, Short or Byte
  public void addVariableAttribute(String varName,String attName,Number value);
    // value must be 1D array of (double, float, int, short, char, byte)
  public void addVariableAttribute(String varName,String attName,Object value);

  // finish structure definition, create file
  public void create() throws IOException;

  // write data to file
  public boolean write(String varName, int[] origin, ArrayAbstract data) throws
        IOException;

  public void close() throws IOException;
}
```

As an example, to create a new file, the program might look like:

```
  NetcdfFileWriteable ncfile = new NetcdfFileWriteable();
  Ncfile.setName("data/mydata.nc");

  // define dimensions
  Dimension latDim = ncfile.addDimension("lat", 64);
  Dimension lonDim = ncfile.addDimension("lon", 128);

  // define Variables
  Dimension[] dims = new Dimension[2];
  dims[0] = latDim;
  dims[1] = lonDim;

  ncfile.addVariable("temperature", double.class, dims);
  ncfile.addVariableAttribute("temperature", "units", "K");

  // add global attribute
  ncfile.addGlobalAttribute("version", new Double(1.2));

  // create the file
  try {
    ncfile.create();
  }  catch (IOException e) {
    System.err.println("ERROR creating file");
    throw e;
  }

  // write some data
  try {
```

```
    ncfile.write("temperature", origin, data);
  catch (IOException e) {
    System.err.println("ERROR writing file");
    throw e;
  }

  // all done
  ncfile.close();
```

1.7 ucar.nc2.dods.DODSNetcdfFile

It is possible to open and manipulate a (read-only) file on a DODS server by instantiating a `DODSNetcdfFile`, which is a subclass of `NetcdfFile` and so inherits the same API.

```
public class ucar.nc2.dods.DODSNetcdfFile extends NetcdfFile {
  public DODSNetcdfFile(String url) throws IOException, DODSException,
       MalformedURLException;
}
```

For example:

```
  NetcdfFile ncfile;
  try {
    ncfile = new DODSNetcdfFile("dods://motherlode.ucar.edu/cgi-bin/dods/nph-
nc/dods/model/2001012200_eta.nc");

  } catch (dods.dap.DODSException e) {
    System.out.println("DODS error = "+e);
  } catch (java.net.MalformedURLException e) {
    System.out.println("bad URL error = "+e);
  } catch (IOException e) {
    System.out.println("IO error = "+e);
  }
```

Note that this uses the protocol "dods" instead of "http", in order to distinguish other possible protocols that also use http. If you use "http" it will be converted to "dods" and returned in this canonical form in the `getName()` method.

2. Multidimensional Arrays

The `ucar.ma2` package is independent of the `ucar.nc2` package, and is intended for general multidimensional array use. Its design is motivated by the needs for NetCDF data to be handled in a general, arbitrary rank, type independent way, and also by the requirements of the JavaGrande numeric working group [3].

It is often critically important for performance that the movement of data between memory and disk is carefully managed. The data in a NetCDF file, for example, is stored out of memory on a local or network file, and the NetCDF library allows efficient extraction of data subsets. At the same time, it is sometimes a useful abstraction to handle data in a general way independent of storage location. A `ucar.ma2.Array` object has its data in the computer's main memory, and so is said to be *memory-resident*. A `ucar.ma2.IOArray` object has its data stored on an I/O device such as a disk file. `ucar.ma2.MultiArray` is the superclass of both, for location-independent manipulation.

Note that to actually get at the data of a `MultiArray` object, you must explicitly call `read()` to bring the data into memory. An `IOArray` will actually do the read, while an `Array` object will just return itself. The fact that an `IOArray` can throw an `IOException` but an `Array` object cannot may in fact be a critical factor in how these objects are used [4]. Similarly, some applications may need to know whether the `IOException` is from a local or remote file. In this case you should test if the `IOException` is an instance of `java.rmi.RemoteException`.

The following is an overview of the important public interfaces of the `ucar.ma2` classes. Consult the javadoc for more complete and recent details.

2.1 ucar.ma2.MultiArray and ucar.ma2.IOArray

A `MultiArray` has a shape describing its size in each dimension, and an element type which is one of the primitive types: `double.class, float.class, long.class, int.class, short.class, byte.class, boolean.class or char.class`. (Note that this includes all of the Java primitives, even though a `ucar.nc2.Variable` has a restricted subset.)

All of the complicated manipulation of data happens on the memory-resident `Array` object obtained from the `read()`. The one exception is that a MultiArray can be "sliced" by fixing one of the dimensions at a particular value. The new `MultiArray` thus has rank one less than the original. This common operation creates a new "logical" `MultiArray` in which no data is moved until a `read()`.

```
public interface MultiArray {
  // array shape and type
  public long   getSize();              // total # elements
  public int    getRank();              // array rank
  public int[]  getShape();             // array dimension sizes
  public Class  getElementType();       // data type of backing array

  // data access
  public Array read() throws IOException;
  public Array read(int [] origin, int [] shape) throws IOException,
       InvalidRangeException;

  // Create new MultiArray, with logical data reordering
  public MultiArray sliceMA(int which_dim, int index_value);
}
```

An `IOArray` is a subclass of `MultiArray`. Since the `read` method signatures in `MultiArray` must show all subclass exceptions, the methods are identical, and so IOArray is actually empty.

```
public interface IOArray extends MultiArray {
}
```

2.2 ucar.ma2.Array: in-memory multidimensional arrays of primitives

`Array` is the abstraction for multidimensional arrays of primitives with data stored in memory. Arrays can have arbitrary rank, and there are concrete implementations for arrays of rank 0-7 for efficiency. The underlying storage can use any of the Java primitive types (double, float, long, int, short, byte, char, boolean). The data can be accessed in a type independent way, for example `getDouble()` can be called on an `Array` of any numeric type. The implementing class casts the data to the requested type (and throws a runtime `ForbiddenConversionException` if the cast is illegal), or uses a direct assign when the requested type is the same as the data type.

The data type, rank, and shape of an array are immutable, while the data values themselves are mutable. Generally this makes Arrays thread-safe, and no synchronization is done in the Array package. (There is the possibility of non-atomic read/writes on 64 bit primitives (long, double). In this case the user should add their own synchronization if needed. Presumably 64-bit CPUs will make those operations atomic also.)

```
public interface Array extends MultiArray {

    // array shape and type
  public long   getSize();              // total # elements
  public int    getRank();              // array rank
  public int[]  getShape();             // array dimension sizes
  public Class  getElementType();       // data type of backing array

    // accessor helpers
  public Index         getIndex();          // random access
  public IndexIterator getIndexIterator();  // sequential access
  public IndexIterator getRangeIterator(Range[] ranges); // access subset
  public IndexIterator getIndexIteratorFast();  // arbitrary order

    // accessors: for each data type (double, float, long, int, short,
    // byte, char, boolean) there are methods of the form eg:
  public double getDouble(Index ima);
  public void   setDouble(Index ima, double value);
  ...

    // create new Array, no data copy
  public Array flip( int dim);          // invert dimension
  public Array permute( int[] dims);    // permute dimensions
  public Array reduce();         // rank reduction for any dims of length 1
  public Array reduce(int dim);   // rank reduction for named dimension
  public Array section( Range[] ranges);       // create logical subset
  public Array slice(int dim, int val);        // rank-1 subset
  public Array transpose( int dim1, int dim2); // transpose dimensions

    // create new Array, with data copy
  public Array copy();
  public Array sectionCopy( Range[] ranges);   // subset
  public Array reshape( int [] shape);  // total # elements must be the same

    // conversion to Java arrays
  public java.lang.Object copyTo1DJavaArray();
  public java.lang.Object copyToNDJavaArray();
}
```

The `getShape()` method returns an integer array containing the length of the `Array` in each dimension. The `getRank()` method returns the number of dimensions, and `getSize()` returns the total number of elements in the `Array`. The `getElementType()` method returns the data type of the backing store, e.g. `double.class, float.class`, etc.

Data element access is described in the sections following this one.

Logical "views" of the array are created in several ways. The `section()` method creates a subarray of the original array. The `slice()` method is a convenience routine for the common `section()` operation of rank-1 section of the array. The `transpose()` method transposes two dimensions, while `permute()` is a general permutation of the indices. The `flip()` method flips the index of the specified dimension so that it logically runs from n-1 to 0, instead of from 0 to n-1. The `reduce()` method allows user control over rank-reduction. All of these logically reorder or subset the data without copying.

Methods that create new `Arrays` by copying the data are `copy()`, `sectionCopy()` and `reshape()`.

The data can be copied into a Java array using the `copyTo1DjavaArray() and copyToNDjavaArray()` methods. In the first case, a 1D Java array of the appropriate primitive type is created and the data is copied to it in logical order (rightmost indices varying fastest). In the second case, an N-dimensional Java array is created that matches the `Array` shape, and the data is copied into it. The user must cast the returned Object to the appropriate Java array type.

2.3 ucar.ma2.Index

Accesses to specific array elements are made using an `Index` object, for example:

```
 double sum = 0.0;
 Index index = A.getIndex();
 int [] shape = A.getShape();
  for (i=0; i<shape[0]; i++)
   for (j=0; j< shape[1]; j++)
    for (k=0; k< shape[2]; k++)
     sum += A.getDouble(index.set(i,j,k));
```

Note that in this example, A can be of any type convertible to a double. `Index` has various convenience methods for setting the element index:

```
public interface Index {
    // general
  public Index set(int [] index);
  public void setDim(int dim, int value);

    // convenience methods for rank 0-7
  public Index set(int v0);                // set index 0
  public Index set(int v0, int v1);        // set index 0,1
  public Index set(int v0, int v1, int v2); // set index 0,1,2
  ...                                       // ..up to dimension 7

  public Index set0(int v);      // set index 0
  public Index set1(int v);      // set index 1
  public Index set2(int v);      // set index 2
  ...                            // ..up to dimension 7
}
```

Because an `Index` object stores state, threads that share an `Array` object must obtain their own `Index` from the `Array`.

2.4 ucar.ma2.IndexIterator

An `IndexIterator` is used to sequentially traverse all data in an `Array` in logical (row-major) order. For example, logical order for `A(i,j,k)` has k varying fastest, then j, then i. Note that because of the possibility that `A` is a flipped or permuted view, logical order may not be the same as physical order. Example:

```
double sum = 0.0;
IndexIterator iter = A.getIndexIterator();
while (iter.hasNext())
  sum += iter.getDoubleNext();
```

Note that in the above example A can be of arbitrary rank.

```
public interface IndexIterator  {
  public boolean hasNext();

    // for each data type
  public double getDoubleNext();
  public double getDoubleCurrent();
```

```
  public void setDoubleNext(double val);
  public void setDoubleCurrent(double val);
  ...
}
```

There are two special kinds of iterators: Array.getIndexIteratorFast() returns an Iterator that iterates over the array in an arbitrary order. It can be used to make iteration as fast as possible when the order of the returned elements is immaterial, for example in the summing example above. Array.getRangeIterator() returns a "range" iterator that iterates over a subset of an array, in logical order. This is an alternative (and equivalent) to first creating an array section, and then obtaining an Iterator. In the following example, the sum is made only over the first 10 rows, and all columns, of the array:

```
  int sum = 0;
  IndexIterator iter =A.getRangeIterator(new Ranges[2]{new Range(0,9), null});
  while (iter.hasNext())
    sum += iter.getIntNext();
```

2.5 Array Implementation

ArrayAbstract is the superclass for our implementations of Array, which use Java 1-D arrays for the data storage. There is a concrete class that extends ArrayAbstract for each data type, e.g., ArrayDouble. For each data type, there is a concrete subclass for each rank 0-7, for example ArrayDouble.D3 is a concrete class specialized for double arrays of rank 3. These rank-specific classes are static inner classes of their superclass. This design allows handling arrays completely generally (through the Array interface), in a rank-independent way (though the Array<Type> classes), or in a rank and type specific way for ranks 0-7 (through the Array<Type>.D<rank> classes).

The most general way to create an Array is to use the static factory method in ArrayAbstract:

```
  public abstract class ArrayAbstract implements Array, Cloneable {
    static public ArrayAbstract factory( Class type, int [] shape);
    ...
  }
```

The type-specific subclasses can be instantiated directly with an arbitrary rank. These also add type-specific get/set accessors, for example:

```
  public class ArrayDouble extends ArrayAbstract {
      // constructor
    public ArrayDouble(int [] dimensions);

       // type-specific accessors
    public double get(Index i);
    public void set(Index i, double value);
       ...
  }
```

If you create your own Array objects, you should usually use the rank and type specific subclasses, which will provide the most efficient access. These classes also add rank-specific get/set routines, for example:

```
  public static class D3 extends ArrayDouble {
      // constructor
    public D3 (int len0, int len1, int len2);

      // type and rank specific accessors
    public double get(int i, int j, int k);
    public void set(int i, int j, int k, double value);
}
```

You may also create an Array from an N-dimensional Java array:

```
public static ArrayAbstract factory(java.lang.Object javaArray);
```

Note that in this case, the data elements are copied out of the Java array into the private `ArrayAbstract` storage.

`IndexImpl` is a concrete, general rank implementation of `Index`, and is extended by rank specific subclasses for efficiency (we have rank 0-7 implementations). `ArrayAbstract` and its subclasses have an `IndexImpl` of the appropriate rank that is delegated all rank-specific functions. This orthogonal design keeps the number of classes small, and makes adding new ranks or data types quite simple.

The "logical view" operations (flip, section, transpose, slice and permute) are implemented by manipulating the index calculation within the `IndexImpl` object. These operations are affine, as is the operation that transforms the n-dim index into the 1-dim element index, and therefore any composition is an affine transformation. The resulting transformation is immutable, and can be computed during the `IndexImpl` object construction. Therefore there is no extra cost associated with the index calculation for these operations (or any composition of them) during element access. These operations do logical data reordering; physical reordering can be done by making an array copy.

An `IndexIterator` traverses array elements in logical order, which we have defined as row-major (as in C). An iterator can in principle be more efficient than other element accesses because 1) the index values cannot be out of range, and therefore do not need to be bounds checked, and 2) the element calculation usually changes by a fixed stride each time. We take advantage of these facts in our implementation, as well as package-private accessor methods, to reduce the number of method calls per data access from 3 to 2.

Examples

Example 1: Create a netCDF File

```
import ucar.ma2.*;
import ucar.nc2.*;
import java.io.IOException;

/**
 * Simple example to create a new netCDF file corresponding to the following
 * CDL:
 *  netcdf example {
 *  dimensions:
 *     lat = 3 ;
 *     lon = 4 ;
 *     time = UNLIMITED ;
 *  variables:
 *     int rh(time, lat, lon) ;
 *              rh:long_name="relative humidity" ;
 *            rh:units = "percent" ;
 *     double T(time, lat, lon) ;
 *              T:long_name="surface temperature" ;
 *            T:units = "degC" ;
 *     float lat(lat) ;
 *            lat:units = "degrees_north" ;
 *     float lon(lon) ;
 *            lon:units = "degrees_east" ;
 *     int time(time) ;
 *            time:units = "hours" ;
```

```
 *  // global attributes:
 *           :title = "Example Data" ;
 *  data:
 *   rh =
 *     1, 2, 3, 4,
 *     5, 6, 7, 8,
 *     9, 10, 11, 12,
 *     21, 22, 23, 24,
 *     25, 26, 27, 28,
 *     29, 30, 31, 32 ;
 *   T =
 *     1, 2, 3, 4,
 *     2, 4, 6, 8,
 *     3, 6, 9, 12,
 *     2.5, 5, 7.5, 10,
 *     5, 10, 15, 20,
 *     7.5, 15, 22.5, 30 ;
 *   lat = 41, 40, 39 ;
 *   lon = -109, -107, -105, -103 ;
 *   time = 6, 18 ;
 *  }
 */
public class CreateNetcdf {

  static String fileName = "example.nc";

  public void main(String[] arg) {
    NetcdfFileWriteable ncfile = new NetcdfFileWriteable();
    ncfile.setName(fileName);

    // define dimensions
    Dimension latDim = ncfile.addDimension("lat", 3);
    Dimension lonDim = ncfile.addDimension("lon", 4);
    Dimension timeDim = ncfile.addDimension("time", -1);

    // define Variables
    Dimension[] dim3 = new Dimension[3];
    dim3[0] = timeDim;
    dim3[1] = latDim;
    dim3[2] = lonDim;

    // int rh(time, lat, lon) ;
    //    rh:long_name="relative humidity" ;
    //    rh:units = "percent" ;
    ncfile.addVariable("rh", int.class, dim3);
    ncfile.addVariableAttribute("rh", "long_name", "relative humidity");
    ncfile.addVariableAttribute("rh", "units", "percent");

    // double T(time, lat, lon) ;
    //   T:long_name="surface temperature" ;
    //   T:units = "degC" ;
    ncfile.addVariable("T", double.class, dim3);
    ncfile.addVariableAttribute("T", "long_name", "surface temperature");
    ncfile.addVariableAttribute("T", "units", "degC");

    // float lat(lat) ;
    //   lat:units = "degrees_north" ;
    ncfile.addVariable("lat", float.class, new Dimension[] {latDim});
    ncfile.addVariableAttribute("lat", "units", "degrees_north");

    // float lon(lon) ;
    // lon:units = "degrees_east" ;
    ncfile.addVariable("lon", float.class, new Dimension[] {lonDim});
```

```
ncfile.addVariableAttribute("lon", "units", "degrees_east");

// int time(time) ;
//   time:units = "hours" ;
ncfile.addVariable("time", int.class, new Dimension[] {timeDim});
ncfile.addVariableAttribute("time", "units", "hours");

//  :title = "Example Data" ;
ncfile.addGlobalAttribute("title", "Example Data");

// create the file
try {
  ncfile.create();
}  catch (IOException e) {
  System.err.println("ERROR creating file");
  assert(false);
}
System.out.println( "ncfile = "+ ncfile);

// write the RH data one value at a time to an Array
int[][][] rhData = {{{ 1,  2,  3,  4}, { 5,  6,  7,  8}, { 9, 10, 11, 12}},
              {{21, 22, 23, 24}, {25, 26, 27, 28}, {29, 30, 31, 32}}};

ArrayInt rhA = new ArrayInt.D3(2, latDim.getLength(), lonDim.getLength());
int i,j,k;
Index ima = rhA.getIndex();
// write
for (i=0; i<2; i++)
  for (j=0; j<latDim.getLength(); j++)
    for (k=0; k<lonDim.getLength(); k++)
      rhA.setInt(ima.set(i,j,k), rhData[i][j][k]);

// write rhData out to disk
try {
  ncfile.write("rh", rhA);
} catch (IOException e) {
  System.err.println("ERROR writing file");
}

/* Here's an ArrayAbstract approach to set the values of T all at once. */
double[][][] tData = {
    {{  1, 2,   3,  4}, {2,  4,  6,  8}, {  3,  6,  9,   12}},
    {{2.5, 5, 7.5, 10}, {5, 10, 15, 20}, {7.5, 15, 22.5, 30}}
};
try {
  ncfile.write("T", ArrayAbstract.factory(tData));
} catch (IOException e) {
  System.err.println("ERROR writing file");
}

/* Store the rest of variable values */
try {
 ncfile.write("lat", ArrayAbstract.factory(new float[] {41, 40, 39}));
 ncfile.write("lon", ArrayAbstract.factory(new float[]
         {-109, -107, -105, -103}));
 ncfile.write("time", ArrayAbstract.factory(new int[] {6, 18}));
} catch (IOException e) {
 System.err.println("ERROR writing file");
}

// all done
try {
  ncfile.close();
```

```
    } catch (IOException e) {
      System.err.println("ERROR writing file");
    }
  }

}
```

Example 2: Print all the data in a netCDF file

```
import ucar.ma2.*;
import ucar.nc2.*;
import java.io.IOException;
import java.util.Iterator;

/**
 * Simple example to print contents of an existing netCDF file of
 * unknown structure, much like ncdump.  A difference is the nesting of
 * multidimensional array data is represented by nested brackets, so the
 * output is not legal CDL that can be used as input for ncgen.
 */

public class DumpNetcdf {

  public static void main(String[] args) {
        if (args.length == 1)
            new DumpNetcdf(args[0]);
        else {
            new DumpNetcdf("example.nc");
        }
  }

  public DumpNetcdf(String fileName) {

    try {
      NetcdfFile nc = new NetcdfFile(fileName); // open it readonly
      System.out.println(nc); // output schema in CDL form (like ncdump)
      System.out.println("data:");
      Iterator vi = nc.getVariableIterator();
      while(vi.hasNext()) {
        Variable v = (Variable) vi.next();
        Array varMa = v.read();
        System.out.print(v.getName() + " =");
        System.out.println(ArrayToString(varMa));
      }

    } catch (java.io.IOException e) {
        e.printStackTrace();
    }

  }


  public String ArrayToString(Array ma) {
    StringBuffer buf = new StringBuffer(ArrayToStringHelper(ma, new
IndentLevel()));
    return buf.toString();
  }

  /**
   * Maintains indentation level for printing nested structures.
   */
  static class IndentLevel {
```

```
    private int level = 0;
    private int indentation;
    private StringBuffer indent;
    private StringBuffer blanks;

    public IndentLevel() {
        this(4);
    }

    public IndentLevel(int indentation) {
        if (indentation > 0)
            this.indentation = indentation;
        indent = new StringBuffer();
        blanks = new StringBuffer();
        for (int i=0; i < indentation; i++)
            blanks.append(" ");
    }

    public void incr() {
        level += indentation;
        indent.append(blanks);
    }

    public void decr() {
        level -= indentation;
        indent.setLength(level);
    }

    public String getIndent() {
        return indent.toString();
    }
  }

  private String ArrayToStringHelper(Array ma, IndentLevel ilev) {

    final int rank = ma.getRank();
    Index ima = ma.getIndex();
    if (rank == 0) {
      return ma.getObject(ima).toString();
    }

    StringBuffer buf = new StringBuffer();
    buf.append("\n" + ilev.getIndent() + "{");
    ilev.incr();
    final int [] dims = ma.getShape();
    final int last = dims[0];
    for(int ii = 0; ii < last; ii++) {
      Array slice = ma.slice(0, ii);
      buf.append(ArrayToStringHelper(slice, ilev));
      if(ii != last - 1)
        buf.append(", ");
    }
    ilev.decr();
    if (rank > 1) {
      buf.append("\n" + ilev.getIndent());
    }
    buf.append("}");

    return buf.toString();
  }
}
```

References

[1] Davis, G. P., and D. F. Fulker, 1999: Extending the NetCDF Model to Java . *AMS IIPS Conference*, January 1999, http://www.unidata.ucar.edu/packages/netcdf/java/ams99.html

[2] http://www.gnu.org/copyleft/lesser.html

[3] John Caron, Unidata's Design for Multidimensional Arrays in Java, ACM Java Grande 2000 Conference, June 2000 http://www.unidata.ucar.edu/staff/caron/ma2/JavaGrande2000.htm

[4] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, A Note on Distributed Computing *Sun Microsystems Technical Note TR-94-29*, 1994 http://www.sun.com/tech/techrep/1994/abstract-29.html