Writing an OPeNDAP Client

Document version 1.1

Dan Holloway

July 20, 2002

Contents

1 Preface 1

2 Writing your own OPeNDAP client 1
- 2.1 Choose a language 1
- 2.2 Client Architecture 1

3 The DAP Architecture 2
- 3.1 The DAP uses HTTP which in turn uses MIME 2
- 3.2 The DAP defines three objects 2
- 3.3 Connecting to the server 2

4 Getting ready to write your client 3

5 Subclassing the data types 5
- 5.1 A quick review of the data types supported by the DAP 5

6 Accessing the DDS object 7

7 Accessing the DAS object 9

8 Accessing the DataDDS object 12

9 Notes 16

1 Preface

This tutorial describe the steps required to enable your client application to interact with the OPeNDAP Data Access Protocol by using the Java or C++ classes provided in the OPeNDAP class libraries.

If your client application currently uses the netCDF API then you only need to relink your application with the OPeNDAP-enabled version of the netCDF client library, allowing you to skip this tutorial entirely.

2 Writing your own OPeNDAP client

2.1 Choose a language

The DODS project provides both a C++ and a Java implementation of the DAP. Each library includes both the classes that implement the various objects which comprise the DAP and support software for handling the server responses, and client-side caching. To choose one of the toolkits, several factors should be weighed. First, with which of the two programming languages are you most comfortable? Also to be considered are: What type of computer will the client run on? For client development, both Java and C++ are supported on win32 and Unix architectures. Java is more likely to be supported on other architectures, such as Mac.

If you plan to OPeNDAP-enable an existing application, to use the Java toolkit the application typically must provide a Java Virtual Machine (VM) with which to run the DAP-specific code. You can still use the Java toolkit code if your application does not provide a Java VM but a mechanism must be implemented to facilitate communication between the Java DAP-specific code and the client application. To use the C++ toolkit you should insure that your client application can be built with, or link with libraries constructed using the GNU C++ compilers (on Unix), or native MS Visual C++ on win32. Check the current list of supported architectures on the web-site, or contact the DODS technical support (support@unidata.ucar.edu), or look at the dods-tech mailing list.

list for information/help.)

2.2 Client Architecture

In essence, an OPeNDAP-enabled client uses overloaded URLs to form the requests to a remote data server. Through the URL, the client connects to the remote server and issues one of several requests. In response to each request, the server will return a well-defined response that the client can use to intern the structure and content of the remote data into local data structures, as well as retrieve any attributes associated with the remote data.

The C++ and Java toolkits share the same characteristics, though the names of the objects and their methods may be slightly different. If you understand how the clients are built, it will be easy to see how your own client application can be OPeNDAP-enabled with minimal effort. The *The DODS Toolkit Programmer's Guide* provides a complete description of the C++ Toolkit and the DODS Java Programming Reference provides a complete description of the Java toolkit.

3 The DAP Architecture

The DAP can be thought of as a layered protocol, with MIME, http, basic objects, and complex presentation-style responses as the layers.

3.1 The DAP uses HTTP which in turn uses MIME

Clients use HTTP when they make requests of DAP servers. HTTP is a fairly straightforward protocol (General information on HTTP, The HTTP/1.1 specification in HTML). It uses MIME documents to encapsulate both the request sent from client to server and the response sent back. This is important for the DAP because the DAP uses headers in both the request and response documents to transfer information. However, for a programmer who intends to write a DAP server, exactly what gets written into those headers and how it get written is not important. Both the C++ and Java class libraries will handle these tasks for you (look at the DODSFilter class to see how). It's important to know about, however, because if you decide not to use the libraries (at least the parts that automatically generate the correct MIME documents) then you'll have to generate the correct headers explicitly.

3.2 The DAP defines three objects

To transfer information from servers to clients, the DAP uses three objects. Whenever a client asks a server for information, it does so by requesting one of these three objects (note: this is not strictly true, but the whole truth will be told in just a bit. For now, assume it's true). These are the Dataset Descriptor Structure (DDS), Dataset Attribute Structure (DAS), and Data object (DataDDS). These are described in considerable detail in other documentation. The Programmer's Guide contains a description of the DDS and DAS objects. These objects contain the name and types of the variables in a dataset, along with any attributes (name-value pairs) bound to the variables. The DataDDS contains data values. We have implemented the SDKs so that the DataDDS is a subclass of the DDS object that adds the capacity to store values with each variable.

COADS Climatology	DAS	DDS
NASA Scatterometer Data	DAS	DDS
Catalog of AVHRR Files	DAS	DDS
AHVRR Image	DAS	DDS

The DAP models all datasets as collections of variables. The DDS and DataDDS objects are containers for those variables. How you represent your dataset using the three objects and the variable's data type hierarchy is covered in section 6.

3.3 Connecting to the server

To manage the connection between the client application and the remote server, the DAP uses two objects. The *Connect* class manages one connection to either a remote data server, or a local access. The *Connections* class is used to manage a set of instances to the class *Connect*. For each data set or file that the client opens, there must be exactly one instance of the *Connect* class. The *The DODS Toolkit Programmer's Guide* provides a description for the C++ toolkit's usage, the DODS Java Programming Reference provides information for the Java toolkit's *DConnect* usage.

4 Getting ready to write your client

An OPeNDAP-enabled client application creates a connection to the remote server using the *Connect* class, and then issues requests to the remote server through the *Connect* class methods. Please refer to the Geturl.java and Geturl.cc sources as examples of command-line based DAP client applications written in Java and C++, respectively.

Most of the software is boilerplate. Following are sections of the Geturl.java client application, later a description of the same example in C++ will be provided. Again, please refer to the complete source listings referenced above.

```
DConnect url = null;
try {
   url = new DConnect(nextURL, accept_deflate);
}
```

This code snippet instantiates a new instance of the *DConnect* class, passing the URL referencing the remote data server, and a boolean flag indicating that the client can accept responses from the server which are compressed.

```
if (get_data) {
   if ((cexpr==false) && (nextURL.indexOf('?') == -1)) {
       System.err.println("Must supply a constraint expression with -D.");
       continue;
   }
   for (int j=0; j<times; j++) {
       try {
          StatusUI ui = null;
          if (gui)
            ui = new StatusWindow(nextURL);
          DataDDS dds = url.getData(expr, ui);
          processData(url, dds, verbose, dump_data, accept_deflate);
       }
       catch (DODSException e) {
         System.err.println(e);
         System.exit(1);
       }
       catch (java.io.FileNotFoundException e) {
         System.err.println(e)
         System.exit(1);
       }
       catch (Exception e) {
         System.err.println(e);
         e.printStackTrace();
         System.exit(1);
       }
   }
}
```

This compound statement block initiaties a data request to the remote server. The *DConnect* method `getData` forms the data request to the remote server by appending the string, *expr*, containing the constraint-expression, onto the URL used in creating the initial *DConnect* to the remote site. The parameter, *ui*, provides an optional `StatusWindow` object to provide the status of the current request to the client.

```
catch (DODSException e) {
  System.err.println(e);
  System.exit(1);
}
catch (java.io.FileNotFoundException e) {
  System.err.println(e);
  System.exit(1);
}
catch (Exception e) {
  System.err.println(e);
  e.printStackTrace();
  System.exit(1);
}
```

Completing the try block is a series of catch blocks that catch exceptions thrown by the DAP library code, and Java input/output and general exceptions.

The C++ toolkit provides similar functionality as the Java toolkit though the parameters to the individual *Connect* methods may vary. The C++ client application geturl.cc uses the similar C++ toolkit classes as the Java toolkit to implement the Geturl client application:

```
string name = argv[i];
Connect url(name, trace, accept_deflate);
url.set_accept_types(accept_types);
```

This code fragment declares an instance of the *Connect* class, passing the URL referencing the remote data server, and a boolean flag indicating that the client can accept responses from the server that are compressed.

```
else if (get_data) {
   if (!(expr || name.find('?') != name.npos)) {
       cerr << "Must supply a constraint expression with -D."
            << endl;
       continue;
   }
   for (int j = 0; j < times; ++j) {
       DDS *dds;
       try {
           dds = url.request_data(expr, gui, async);
           if (!dds) {
               cerr << "Error: " << url.error().error_message() << endl;
               continue;
           }
           process_data(url, dds, verbose, print_rows);
           delete dds; dds = 0;
       }
       catch (Error &e) {
           e.display_message(url.gui());
       }
   }
}
```

This compound statement block initiaties a data request from the remote server. The *Connect* method `request_data` forms the data request to the remote server by appending the string, *expr*, containing the constraint-expression, onto the URL used in creating the initial *Connect* to the remote site. The parameter, *gui*, provides an optional *GUI* object to show the user the status of the current request.

```
catch (Error &e) {
   e.display_message(url.gui());
}
```

Completing the try block is a catch block that picks up exceptions thrown by the DAP library code. The DAP C++ library throws two types of exceptions, *Error* and *InternalErr*. The latter is a subclass of *Error*, so catching just *Error* will get everything.

5 Subclassing the data types

The DAP defines a data type hierarchy as the core of its data model. This collection of data types includes scalar, vector and constructor types. Most of the types are available in all modern programming languages with the exceptions being Url, Sequence and Grid. In the DAP library, the class *BaseType* is the root of the data type tree.

5.1 A quick review of the data types supported by the DAP

The DAP supports the common scalar data types such as Byte, 16- and 32-bit signed and unsigned integers, and 32- and 64-bit floating point numbers. The DAP also supports Strings and Urls as basic scalar types. The DAP includes two vector data types, Arrays (of unlimited size and dimensionality) and Lists. Lists in the DAP must be type homogeneous and we don't support Lists of Lists. The DAP also supports three type-constructors: Structure, Sequence and Grid. A Structure on the DAP mimics a struct in C. A Sequence is a table-like data structure inherited from the JGOFS data system. It can be used to hold information that might be stored in relational databases or tables, either flat or hierarchical. The JGOFS, FreeForm and HDF servers all use the Sequence data type. Lastly, the Grid data type is used to bind an array to a group of 'map vectors,' single dimension arrays that provide non-integral values for the indices of the array. The most typical use of a Grid is to provide latitude and longitude registration for some georeferenced array data (e.g., a projected satellite image). The DAP does not have a pointer data type, but in some cases the Url data type can be used as a pointer to variables between files. More information about the DAP's data type hierarchy is given in the Programmer's Guide.

When you build a DAP client, you must create a collection of data type subclasses. That is each of the leaf classes in the preceding class diagram must be subclassed by your client. This is pretty easy since a good bit of the work is rote.

First we'll illustrate the parts that are mechanical. Here's an example from the C++ Matlab client. The class is the Byte class. In the case of the matlab client, this class doesn't do anything beyond the bare minimum, so it's a good starting point:

```
Byte *
NewByte(const string &n)
{
    return new ClientByte(n);
}

BaseType *
ClientByte::ptr_duplicate()
{
    return new ClientByte(*this);
}

bool
ClientByte::read(const string &)
{
  throw InternalErr(__FILE__, __LINE__, "Called unimplemented read method");
}
```

To create a child of any of the data type leaf classes, you must define three methods and one function. Let's talk about the function first. The function `NewByte` is a what Meyers calls a 'virtual constructor.' it's similar to a low-budget factory class, low-budget because it's not a class. This function is undefined in the DAP library but it used there when creating instances of the *Byte*. Thus the function `NewByte` is an interface that can be used to create instances of Byte without knowing in advance the static type of the object that will be created. If all this sounds a little weird, just remember that your Byte, Int16, ... Grid classes — whatever they may be called — must all contain an implementation of this function and they should all return a pointer to an instance of your child classes. In this case, it's an instance of the *ClientByte* class. If you look in the files for the Matlab server, you'll see that *Grid* returns a pointer to a new *ClientGrid*, and so on.

Second, a constructor must be implemented and should take the name of the variable as its sole argument.

Third, your child classes should also define the `ptr_duplicate()` method. This method returns a pointer to a new instance of an object in the same class. Occasionally objects are indicated with pointers specified as `BaseType *`. If you use new to copy an object referenced this way, you would get an object of type *BaseType* instead of the class you really want, presumably a subclass of that type.

Finally, each of the child classes must provide an implementation of the `read()` method. This method is called by code in the DAP library to read values from the data set. An OPeNDAP client typically will not use this method, since it's getting its data from the server. But the class is defined as `virtual` in the class libraries, so must be sub-classed. In the example above, the implementation simply throws an InternalErr object.

The DAP Java Toolkit uses a similar mechanism to support subclassing of the DAP base classes. Following is an example from the Java Matlab client application, from the source file MatlabFactory.java. In the Java DAP toolkit, for those classes which are not required to be specialized for the underlying client application, an instance of the base class may be returned through the Factory interface.

```
public class MatlabFactory implements BaseTypeFactory {
  //..................................
  /**
   * Construct a new DByte.
   * @return the new DByte
   */
  public DByte newDByte() {
    return new DByte();
  }

  /**
   * Construct a new DByte with name n.
   * @param n the variable name
   * @return the new DByte
   */
  public DByte newDByte(String n) {
    return new DByte(n);
  }
```

For those classes specialized for the underlying client application, an instance to the specialized class is returned through the Factory interface.

```
  //..................................
  /**
   * Construct a new DArray.
   * @return the new DArray
   */
  public DArray newDArray() {
    return new MatlabArray();
  }

  /**
   * Construct a new MatlabArray with name n.
   * @param n the variable name
   * @return the new MatlabArray
   */
  public DArray newDArray(String n) {
    return new MatlabArray(n);
  }
```

6 Accessing the DDS object

The Data Descriptor Structure (DDS) is a data structure used by the DODS software to describe datasets and subsets of those datasets. The DDS may be thought of as the declarations for the data structures that will hold data requested by some DODS client. Part of the job of a DODS server is to build a suitable DDS for a specific dataset and to send it to the client. Depending on the data access API in use, this may involve reading part of the dataset and inferring the DDS. Other APIs may require the server simply to read some ancillary data file with the DDS in it.

For the client, the DDS object includes methods for reading the persistent form of the object sent from a server. This includes parsing the ASCII representation of the object and, possibly, reading data received from a server into a data object.

Note that the class DDS is used to instantiate both DDS and DataDDS objects. A DDS that is empty (contains no actual data) is used by servers to send structural information to the client. The same DDS can becomes a DataDDS when data values are bound to the variables it defines.

For a complete description of the DDS layout and protocol, please refer to The DODS User Guide.

The DDS has an ASCII representation, which is what is transmitted from a DODS server to a client. Here is the DDS representation of an entire dataset containing a time series of worldwide grids of sea surface temperatures:

```
Dataset {
    Float64 lat[lat = 180];
    Float64 lon[lon = 360];
    Float64 time[time = 404];
    Grid {
      ARRAY:
         Int32 sst[time = 404][lat = 180][lon = 360];
      MAPS:
         Float64 time[time = 404];
         Float64 lat[lat = 180];
         Float64 lon[lon = 360];
    } sst;
} weekly;
```

If the data request to this dataset includes a constraint expression, the corresponding DDS might be different. For example, if the request was only for northern hemisphere data at a specific time, the above DDS might be modified to appear like this:

```
Dataset {
    Grid {
      ARRAY:
         Int32 sst[time = 1][lat = 90][lon = 360];
      MAPS:
         Float64 time[time = 1];
         Float64 lat[lat = 90];
         Float64 lon[lon = 360];
    } sst;
} weekly;
```

Since the constraint has narrowed the area of interest, the range of latitude values has been halved, and there is only one time value in the returned array. Note that the simple arrays (lat, lon, and time) described in the dataset are also part of the sst Grid object. They can be requested by themselves or as part of that larger object.

See The DODS User Guide, or the documentation of the *BaseType* class for descriptions of the DODS data types.

Reading data from a DDS object is the heart of writing your own OPeNDAP client. To integrate the information contained in the DDS, you must do two things. First you must decide how the data type hierarchy that is part of the DAP can be represented in your client application. Some client applications cannot represent all possible DAP data types directly, where possible the client developer should strive to support as many data types as possible to facilitate access to the wide variety of data accessible through OPeNDAP servers. Second, once you have determined how to map the DAP data types into the client application you need to write code that can build an instance of DDS representing the remote data. In practice the hardest part of this the first part; once you know how to map variables from the DAP into your client application, writing code to build the DDS instance is easy.

```
    else if (get_dds) {
          for (int j = 0; j < times; ++j) {
              try {
                  if (!url.request_dds(gui))
                      continue;
              }
              catch (Error &e) {
                  e.display_message(url.gui());
                  continue;   // Goto the next URL or exit the loop.
              }
              if (verbose) {
                  cerr << "Server version: " << url.server_version()
                       << endl;
                  cerr << "DDS:" << endl;
              }
              url.dds().print();
          }
      }
```

Above, the *Connect* method `request_dds` is called, passing the optional progress *GUI* reference. Following is an example from the C++ Matlab client illustrates a simple traversal of the DDS object returned from the `connect.request_data` method.

```
static void
process_data(Connect &url, DDS *dds)
{
   if (verbose)
       cerr << "Server version: " << url.server_version() << endl;

   for (Pix q = dds->first_var(); q; dds->next_var(q)) {
       dds->var(q)->print_decl(cout, "", true);
       smart_newline(cout, dds->var(q)->type());
   }
}
```

In the C++ DAP classes, *Pix*, or Pseudo-Indexes, are used to iterate over the DDS object. The `process_data` function begins at the `dds->first_var()` and interates until the *Pix* returned by the `dds->next_var(q)` returns null. For each valid *Pix* provided by `dds->next_var(q)` method, the `dds->var(q)` method is used to access the individual DAP data object referenced by the *Pix*, and invoke the public methods available for that data type. The *The DODS Toolkit Programmer's Guide* provides a description for the each of the data types, and the methods available to operate on them.

In the Java DAP toolkit, an `Enumeration` type is used to iterate over the contents of the DDS class instance.

```
/*
 * Return an Enumeration of the variables in the dataset.
 */
public Enumeration getVariables() {
   if(dds != null)
       return dds.getVariables();
   else
       return null;
}
```

The `Enumeration` type can be used to iterate over the contents of the DDS instance with a simple loop construct:

```
 private static void processData(DConnect url, DataDDS dds, boolean verbose,
                                 boolean dump_data, boolean compress) {

   Enumeration dodsVar = dds.getVariables();

   while (dodsVar.hasMoreElements()) {
      BaseType bt = (BaseType) dodsVar.nextElement();

      bt.printDecl(System.out);
      System.out.println();
   }
}
```

The first line declares an `Enumeration` type variable, `dodsVar` and assigns the `Enumeration` returned by the *DDS* `getVariables` method. While the enumeration has values remaining, the example function calls the `printDecl` and `printVal` methods of the underlying *BaseType* to print out the declaration and values of each variable referenced by the `Enumeration` type.

7 Accessing the DAS object

The Data Attribute Structure (DAS) is a set of name-value pairs used to describe the data in a particular dataset. The name-value pairs are called the "attributes." The values may be of any of the DODS simple data types (Byte, Int16, UInt16, Int32, UInt32, Float32, Float64, String and URL), and may be scalar or vector. (Note that all values are actually stored as string data.)

A value may also consist of a set of other name-value pairs. This makes it possible to nest collections of attributes, giving rise to a hierarchy of attributes. DODS uses this structure to provide information about variables in a dataset.

In the following example of a DAS, several of the attribute collections have names corresponding to the names of variables in a hypothetical dataset. The attributes in that collection are said to belong to that variable. For example, the `lat` variable has an attribute units of `degrees_north.`

```
Attributes {
   GLOBAL {
      String title "Reynolds Optimum Interpolation (OI) SST";
   }
   lat {
      String units "degrees_north";
      String long_name "Latitude";
      Float64 actual_range 89.5, -89.5;
   }
   lon {
      String units "degrees_east";
      String long_name "Longitude";
      Float64 actual_range 0.5, 359.5;
   }
   time {
      String units "days since 1-1-1 00:00:00";
      String long_name "Time";
      Float64 actual_range 726468., 729289.;
      String delta_t "0000-00-07 00:00:00";
   }
   sst {
      String long_name "Weekly Means of Sea Surface Temperature";
      Float64 actual_range -1.8, 35.09;
      String units "degC";
      Float64 add_offset 0.;
      Float64 scale_factor 0.0099999998;
      Int32 missing_value 32767;
  }
}
```

Attributes may have arbitrary names, although in most datasets it is important to choose these names so a reader will know what they describe. In the above example, the GLOBAL attribute provides information about the entire dataset.

Data attribute information is an important part of the the data provided to a DODS client by a server, and the DAS is how this data is packaged for sending (and how it is received).

An example of Attribute handling in a client application is provided in the `www-int` C++ source:

```
void
WWWOutput::write_attributes(AttrTable *attr, const string prefix)
{
    if (attr) {
        for (Pix a = attr->first_attr(); a; attr->next_attr(a)) {
            if (attr->is_container(a))
                write_attributes(attr->get_attr_table(a),
                                 (prefix == "") ? attr->get_name(a)
                                 : prefix + string(".") + attr->get_name(a));
            else {
                if (prefix != "")
                    _os << prefix << "." << attr->get_name(a) << ": ";
                else
                    _os << attr->get_name(a) << ": ";

                int num_attr = attr->get_attr_num(a) - 1 ;
                for (int i = 0; i < num_attr; ++i)
                    _os << attr->get_attr(a, i) << ", ";
                _os << attr->get_attr(a, num_attr) << endl;
            }
        }
    }
}
```

Similar with the DDS class in the C++ DAP toolkit, *Pix*, or Pseudo-Indexes, are used to interate over the attributes contained in the DAS instance. In the example `write_attributes` method, the loop begins at the `das->first_attr()` and interates until the last *Pix* accessed by the `das->next_attr(a)` method returns null. For each valid *Pix* provided by `dds->next_attr(a)` method, various *AttrTable* methods are used to access the individual *AttrTable* objects referenced by the *Pix*. The *The DODS Toolkit Programmer's Guide* provides a description of the Data Attribute Structure, and the methods available to operate on Attributes.

In the Java DAP toolkit, similar to the DDS class an `Enumeration` type is used to iterate over the contents of the DAS and *AttrTable* class instances. The DODS Java Programming Reference provides a description of the Java DAP toolkits implementation of the DAS, *AttrTable*, and *Attribute* classes.

```
public void writeAttributes(AttributeTable aTbl, String indent) {

   if(aTbl != null){

      Enumeration e = aTbl.getNames();

      while(e.hasMoreElements()){
         String aName = (String)e.nextElement();
         Attribute a = aTbl.getAttribute(aName);

         if (a.isContainer()) {
             pWrt.print(indent+aName+":\n");
             writeAttributes(a.getContainer(),indent+"  ");
         }
         else {
             pWrt.print(indent + aName + ": ");
             if(_Debug) { System.out.println("Getting attribute value enumeration for \""+aName +"\"...");}

             Enumeration es = a.getValues();
             if(_Debug) { System.out.println("Attribute Values enumeration: "+es);}
             int i = 0;
             while(es.hasMoreElements()){
                 String val = (String)es.nextElement();
                 if(_Debug) { System.out.println("Value " + i + ": "+val);}

                 pWrt.print(val);

                 if(es.hasMoreElements())
                    pWrt.print(", ");

                 i++;
             }
             pWrt.println("");
         }

      }
   }
}
```

The preceding Java example provides the same functionality as the C++ example, but uses an `Enumeration` type to iterate over the DAS container classes.

8 Accessing the DataDDS object

The *DataDDS* class is an extension of class *DDS* which contains the binary data values returned by the remote server. The methods used to access the *DataDDS* instance are the same as for the *DDS*, with the exception of the *Connect* `get` methods. When requesting a *DataDDS* the DAP allows the client request to include a constraint-expression indicating that an server-side operation is being requested by the client.

A DODS server can accept a ”constraint expression” contained in the URL query string. The DODS constraint expression describes how a DODS server should subsample a DODS dataset before sending the data back to the client. The details of the constraint expresssion syntax are covered in The DODS User Guide[constraint]. What's important here is simply that the constraint expression is a logical expression with two clauses: projection and selection.

```
if (get_data) {
   if ((cexpr==false) && (nextURL.indexOf('?') == -1)) {
       System.err.println("Must supply a constraint expression with -D.");
       continue;
   }
   for (int j=0; j<times; j++) {
       try {
          StatusUI ui = null;
          if (gui)
            ui = new StatusWindow(nextURL);
          DataDDS dds = url.getData(expr, ui);
          processData(url, dds, verbose, dump_data, accept_deflate);
       }
       catch (DODSException e) {
         System.err.println(e);
         System.exit(1);
       }
       catch (java.io.FileNotFoundException e) {
         System.err.println(e)
         System.exit(1);
       }
       catch (Exception e) {
         System.err.println(e);
         e.printStackTrace();
         System.exit(1);
       }
   }
}
```

The *Connect* method getData() is passed an optional constraint-expression requesting that a subsetting, or other server-side operation be performed on the data before returning it to the client application. The client application typically accesses the elements of the `DataDDS` using the subclassed DAP data types which the client has specified.

In the Java DAP toolkit, an *Enumeration* type is used to iterate over the contents of the *DDS*, and *DataDDS* class instance.

```
private static void processData(DConnect url, DataDDS dds, boolean verbose,
                                boolean dump_data, boolean compress) {

  Enumeration dodsVar = dds.getVariables();

  while (dodsVar.hasMoreElements()) {
     BaseType bt = (BaseType) dodsVar.nextElement();

     bt.printDecl(System.out);
     System.out.println();
  }
}
```

To access the value contained in the element referenced by the *Enumeration* type, the *BaseType* method `getValue()` returns the contents of the data types data buffer.

```
DInt32 val = (DInt32) dodsVar.getValue();
```

In the C++ DAP toolkit, `Pix` operations are used to traverse the *DataDDS* to access the individual DAP data objects contained in the *DataDDS*. The following example simply prints the DAP data objects declaration, to access the binary data returned by the server the DAP provides access methods to retrieve the data object's buffer contents.

```
static void
process_data(Connect &url, DDS *dds)
{
   if (verbose)
       cerr << "Server version: " << url.server_version() << endl;

   for (Pix q = dds->first_var(); q; dds->next_var(q)) {
       dds->var(q)->print_decl(cout, "", true);
       smart_newline(cout, dds->var(q)->type());
   }
}
```

The binary data returned by the server is stored in the `_buf` member of each of the DAP's atomic data types. To retrieve the atomic data type's buffer contents the *BaseType* method `buf2val` is used.

```
n_bytes = dds->var(q)->buf2val((void **) &localVar);
```

The DAP *Sequence* data types can be visualized as a relational table structure consisting of rows and columns. To access the elements of the table the DAP provides accessor methods for each row and column element. The client application uses these methods to reference the individual DAP data objects comprising the *Sequence*.

The following Java example is from the Java Matlab client application and illustrates the use of the Java DAP classes and methods to access elements from a *Sequence* data type.

```
/**
 * This class takes a MatlabSequence object, and provides methods to return
 * the columns of the sequence as arrays of atomic types.  I wrote this before
 * I subclassed DSequence, so I may end up moving these functions into
 * MatlabSequence and doing away with this class.
 *
 * Note: Java doesn't have any unsigned types, so the getU* functions return
 *       a signed variable
 *
 */

class DodsSequenceProcessor extends Object {
    private MatlabSequence dodsSeq;

    public DodsSequenceProcessor(MatlabSequence seq) {
        dodsSeq = seq;
    }

    /**
     * Get a column of DBytes from the sequence and return it as an
     * array of bytes
     * @param name The name of the column
     * @return an array of bytes containing the data.
     */
    public byte[] getByte(String name)
        throws NoSuchVariableException
    {
        int numVars = dodsSeq.getRowCount();
        byte[] values = new byte[numVars];
        BaseType temp[] = null;

        try {
            temp = dodsSeq.getColumn(name);
        }
        catch(NoSuchVariableException e) {
            throw(e);
        }

        if(temp[0] instanceof DByte) {
            for(int i=0; i<numVars; i++) {
                values[i] = ((DByte)temp[i]).getValue();
            }
        }
        return values;
    }

    ...
```

The code example illustrates accessing the values of a *Sequence* element, in this example of type *Byte*. The first thing the client does is determine the number of rows in the *Sequence* which was returned by the server, and creates a vector of `byte` of that size. The client creates an empty array of type *BaseType* which is used to store the return values from the *Sequence* method `getColumn(name)`. Then for each *BaseType* element referenced in the array the client accesses its buffer values with the *BaseType* `getValue()` method.

The following C++ example is from the C++ Matlab client application and illustrates the use of the C++ DAP classes and methods to access elements from a *Sequence* data type.

```
void
ClientSequence::print_one_row(ostream &os, int row, string space,
                              bool print_row_num)
{

    const int elements = element_count();
    for (int j = 0; j < elements; ++j) {
        BaseType *bt_ptr = var_value(row, j);

        if (bt_ptr) {           // data
            bt_ptr->print_val(os, space, true);
        }
    }

}

void
ClientSequence::print_val_by_rows(ostream &os, string space,
                                  bool print_decl_p,
                                  bool print_row_numners)
{
    const int rows = number_of_rows();
    for (int i = 0; i < rows; ++i) {
        print_one_row(os, i, space, false);
    }
}
```

Like the Java example, the C++ client uses rows and columns to access the individual elements of the *Sequence*. The C++ Matlab client uses two methods to accomplish the extraction, the first, `print_val_by_row()` determines the number of rows in the *Sequence* and calls the `print_one_row()` for each of the rows in the *Sequence*. The C++ DAP implementation of the *Sequence* data type provides the `var_value(row,col)` method to access the individual elements of the *Sequence*. The `var_value()` method returns a *BaseType* pointer to the row, column element of the *Sequence*. To access the binary data value stored in that element, the *BaseType* method `buf2val()` can be used. The preceding example simply prints the contents of the element, most client applications would assign the contents to a local variable in the workspace.

```
    /**
     * Return the data held in the MatlabArray as an array of
     * atomic types.
     * @return The data.
     */
    public Object getData() {
        PrimitiveVector pv = getPrimitiveVector();
        if( (pv instanceof BaseTypePrimitiveVector) == false)
            return pv.getInternalStorage();

        else {
            BaseTypePrimitiveVector basePV = (BaseTypePrimitiveVector)pv;
            BaseType varTemplate = (BaseType)basePV.getValue(0);
            if(varTemplate instanceof MatlabString) {
                char[][] arrayData = new char[basePV.getLength()][];
                for(int i=0;i<pv.getLength();i++) {
                    arrayData[i] = ((MatlabString)basePV.getValue(i)).getValue().toCharArray();
                }
                return arrayData;
            }
            else if(varTemplate instanceof MatlabURL) {
                char[][] arrayData = new char[basePV.getLength()][];
                for(int i=0;i<pv.getLength();i++) {
                    arrayData[i] = ((MatlabURL)basePV.getValue(i)).getValue().toCharArray();
                }
                return arrayData;
            }
            else return null;
        }
    }
void
ClientArray::print_val(ostream &os, string, bool print_decl_p)
{
    if (print_decl_p) {
        os << type_name() << endl << var()->type_name() << " "
           << get_matlab_name() << " " << dimensions(true)
           << endl;

        // Write the actual dimension sizes on a spearate line.
        for (Pix p = first_dim(); p; next_dim(p))
            os << dimension_size(p, true) << " ";

        os << endl;
    }

    for (int i = 0; i < length(); ++i)
        var(i)->print_val(os, "", false);
}
```

9 Notes

Here's a collection of information that might be important to specific clients but is hard to fit into a general tutorial.

- Because there are no metadata requirements to serve data via the OPeNDAP protocol, client applications may not find all the information they require to make use of the data. The DAP currently supports ancillary DAS files at the remote server site. In development is an Ancillary Information Service (AIS) which will permit these external metadata resources to be located with the remote data itself, at other remote server sites, or on the client. Any metadata augmented by the AIS will be clearly indicated in the attributes.

- It is the client application's responsibility to provide the initial base URLs to the remote server site. In development are data discovery services, including an ImportWizard which can query existing directory services such as the GCMD, to provide the base URLs to providers with installed OPeNDAP data servers.
- The DODS Project has developed two tools to help with serving datasets that contain many files. The first is to set up a 'file server' a kind of catalog of URLs that is itself a DODS data set. The second is called the Aggregation Server (AS). The AS can automatically aggregate discrete datasets, accessed as either as files (in some cases) or URLs to produce a single data set. See the DODS Home page and/or contact tech support (support@unidata.ucar.edu) for help with this.
- You can get help from the DODS Home page, the dods-techand the DODS user support desk (support@unidata.ucar.edu).