2

A Java Client for `MDSplus`

This book assumes that you, our readers, have had an introductory-level experience with Java programming. You should have written some Java classes and methods and you should have built up a small, multi-class system. The next few sections of this chapter are intended as a revision of the parts of the Java language, and API, that you need in order to write client/server programs: specifically Java's input/output framework including sockets. Following this, we will briefly discuss exception handling and threading and then we will start to build a program which can interact with an `MDSplus` server. This program will build on itself to eventually become our prototype `EScope` application. We will call its various versions "`PreEScope`" to distinguish them from the `EScope` versions which we will develop through the successive application of design patterns.

The Java input/output (IO) framework makes it possible to get data from a server over a network in a completely analogous way to reading it from a file on your local disk. So we will start out by looking at Java IO in general and we will do this using an example.

2.1 An Example: `SimplePlot`

The code in Listing 2.1 reads in two arrays, of X and Y data, and then calls a static method of a `Plotter` class to plot the graph. This example and its data files are available on the CD provided with this book (as are all of the other examples). You can compile and run it providing its class file is located in the same directory as the data files.

For the moment, we are not concerned with the details of the plotting method or about the details of the data itself. We just observe that this program opens up two files called "`data_xVals`" and "`data_yVals`". These files contain *binary* data; you cannot list them at a terminal but you can read them into your Java program using classes from the `java.io` package. The files each contain an `int` (giving the length of the data array) followed by

an array of `double` data. The example foreshadows the way that `EScope` will eventually work!

Listing 2.1. `SimplePlot` reads binary data and constructs an XY plot.

```java
import java.io.*;
/** Read data from 2 binary files and call Plotter.plot*/
public class SimplePlot
{
    public static void main(String[] args)
    {
        double[] xVals=null,yVals=null;
        int xLen=0,yLen=0;
        try
        {   // read Y values from first file
            File f1 = new File("data_yVals");
            DataInputStream in1 =
                new DataInputStream(
                new BufferedInputStream(
                new FileInputStream(f1)));
            yLen = in1.readInt();
            yVals=new double[yLen];
            for (int i = 0; i < yLen; i ++)
            {
                yVals[i] = in1.readDouble();
            }
            // read X values from second file
            File f2 = new File("data_xVals");
            DataInputStream in2 =
                new DataInputStream(
                new BufferedInputStream(
                new FileInputStream(f2)));
            xLen = in2.readInt();
            xVals=new double[xLen];
            for (int i = 0; i < xLen; i ++)
            {
                xVals[i] = in2.readDouble();
            }
            in1.close();
            in2.close();
        }
        catch (IOException e) {System.out.println(e);}
        Plotter.plot(xVals,yVals,"test");
    }
}
```

2.2 Java IO

Java "streams" are objects which deal with the flows of data into, and out of, your program:

- "Input streams" get sequences of bytes from a "source" of data.
- "Output streams" send sequences of bytes to a "sink" of data.

Streams can transfer data to and from files, network connections and other input/output devices as well as internal program variables.

To read data from a *file* we need to

- open the file as a `FileInputStream`
- apply some buffering to our stream using, for example, objects of type `BufferedInputStream`
- call methods of a special "reader" object which can recognize specific data types (for example, the `DataInputStream` methods `readInt()` and `readDouble()` read and recognize `int` and `double` data respectively)

The complete "reader object" is constructed by "chaining together" the opening, buffering and reading objects. The constructor of the reader object accepts a buffering object which itself accepts an opening object in its constructor. For example, in the following code excerpt from `SimplePlot`, `FileInputStream` reads bytes from the file. `BufferedInputStream` buffers the data so that not every read is associated with a disk access and `DataInputStream` reads groups of bytes that are associated with the primitive data types:

```
File f2 = new File("data_yVals");
DataInputStream in2 =
   new DataInputStream(
   new BufferedInputStream(
   new FileInputStream(f2)));
```

Note that binary files can combine data of different types. In the example, we read an `int` which let us know how many elements of a `double` array remained to be read.

In addition to `readInt()` and `readDouble()`, `DataInputStream` has analogous methods to read all the primitive types as well as `readUTF()` which returns a string. It can also read whole lines, read whole files and skip over bytes of data.

The unusual pattern of constructing a `DataInputStream` object by feeding it objects of the other classes is, in fact, a well-known design pattern known as the *decorator*. We will return to it much later in this book in Chapter 11. A schematic representation of the chaining involved in the pattern is shown in Fig. 2.1.

2.2.1 A Remark on Exceptions

All of the file handling methods described above throw *exceptions* when unforeseen error conditions occur. For example, your code may attempt to read
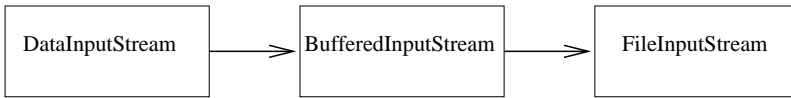
Fig. 2.1. Schematic representation of a `DataInputStream` object showing the chaining to objects of type `BufferedInputStream` and `FileInputStream`.

data from a file which does not exist. The block of code which contains the read method will stop executing and control will transfer to a appropriate `catch` block. Java forces you to wrap methods which throw exceptions inside `try...catch` blocks like the ones shown in the `SimplePlot` example. We will discuss exception handling in Section 2.3.

2.2.2 Character-Based Text Streams

Byte streams are efficient ways of storing data but they are not readable by humans! We can use text files to cope with this problem.

Using text files with Java programs creates another problem. Java uses Unicode (2 byte) encoding which is not the usual operating-system default. Unicode is designed to be compatible with all of the human-readable scripts in the world so it contains additional storage requirements to the common, ASCII, encoding.

A Java `InputStreamReader` object turns an input stream that is based on an operating-system default into one that it based on Unicode. Like the `BufferedInputStream`, it is constructed using the Decorator pattern. An argument of type `FileInputStream` is passed to the constructor:

```
InputStreamReader in =
    new InputStreamReader(new FileInputStream("textData"));
```

Once you have created your `InputStreamReader` you can pass it to the constructor of an object of the `BufferedReader` class which can read whole lines of text and return them as strings:

```
BufferedReader in1 = new BufferedReader(
                        new InputStreamReader(
                        new FileInputStream("textData"));
String line;
while ((line=in1.readLine()) != null)
{
    ....
}
```

Now that you have your data as a string, you might need to parse the strings to convert them to other data types. For example,

```
double x = Double.parseDouble(line)
```

would be a valid way of parsing a line which contains just one `double` value. Otherwise, if there are several numbers on any one line you could use a `java.util.StringTokenizer` object to break up the line and then parse each token separately.

2.2.3 Input from the Keyboard

As something of an aside, we remark on the way you *used to have to* read input from your keyboard into a Java program. Although the procedure is somewhat tedious and now unnecessary, it is interesting because of the neat way that it fits in with the other Java IO examples shown here.

Before Java 1.5, you needed to be aware that the `System` class contains a static field, `System.in`, which is an object of type `InputStream`. It also has a static field, `System.out`, which is an object of type `PrintStream`. `InputStream` objects have a method, `read`, which is able to read a single byte at a time but it cannot read a line of text input (which is what you want to do). In order to increase the functionality of `System.in`, it needs to be passed to an `InputStreamReader` object which can read individual characters. This `InputStreamReader` object then needs to be passed to a `BufferedReader` object which has a `readLine` method to read entire lines at once.

This is how it works:

```java
InputStreamReader isReader = new InputStreamReader
                                       (System.in);
BufferedReader bReader = new BufferedReader(isReader);
System.out.println(''What is your name?'');
String name = bReader.readLine();
```

Since Java 1.5, this has been greatly streamlined. There is a class `java.util.Scanner` which can be used to deliver string tokens corresponding to successive lines of input typed at a console and this is how it works:

```java
java.util.Scanner sc = new java.util.Scanner(System.in);
System.out.println(''What is your name?'');
String name = sc.next();
```

2.2.4 Writing Text Output

The `PrintWriter` class is often used for writing text to files:

```java
PrintWriter out = new PrintWriter(
                  new OutputStreamWriter(
                  new FileOutputStream("output.txt")));
```

A `PrintWriter` object has `print` and `println` methods (just like `System.out`). It also has `flush()` and `close()` methods which are good to use to make sure that the output buffers have been emptied. The `PrintWriter` constructor can also be called with a second boolean argument set to `true` in order to "autoflush" buffers.

Note that you can actually leave out the `OutputStreamWriter` because `PrintWriter` adds one by default. (But, in contrast, `BufferedReader` must have `InputStreamReader` in order to read in text data.) The three classes are chained together using the decorator pattern just the same way as for the `InputStream` example of Fig. 2.1.

2.2.5 Other Topics in IO

Java's Input/Output framework has many other subtleties and features which you can explore using Sun's online Java tutorials and documentation and by looking at good books. Some interesting ones are

- parsing strings
- reading from zip and jar compressed files
- reading from URL's

2.3 Exception Handling

The sorts of exceptions that a programmer needs to handle using Java's exception mechanism include

- user input errors,
- trying to read past the end of a file,
- trying to open a file which does not exist,
- trying to send data to a socket which has not been opened.

All of these problems occur when the behavior of the system, or a user, causes an error condition to occur. These errors are quite different from programming errors. As a pedagogical point, you should fix your bugs before your program gets released. You should *not use exceptions* to deal with the existence of possible programming bugs!

Exceptions are objects of classes which extend the `Throwable` class. You can define your own exception classes. Java has an extensive hierarchy of exception classes which you can read about as you need them.

Exceptions are *thrown* by methods and *caught* within the code bodies of other methods: If method B throws an exception and it gets called by another method, A, then method A must either

- throw the same exception as B, or
- the call to method B must take place inside a `try ... catch` block inside method A.

In the `SimplePlot` example, all of the input-output calls are placed inside one big `try` block in the calling method. Because all of the exceptions of the `java.io` library are children of `IOException`, they end up being handled by the one `catch` clause:

```
catch ( IOException e) { System . out . println ( e ) ; }
```

This `catch` clause simply printed out a string identifier of the offending exception object. Using the parent `IOException` object is a very "coarse-grained" way of catching IO exceptions. As an alternative, it is possible to have a number of `catch` clauses following one `try` block. Any number of exception subclasses of `IOException` could be identified one after another and the program would be able to test each one and provide specialized handling for each.

The Java exception-handling mechanism allows you to look at only the `try` blocks to identify which parts of a method get executed if all goes well. This is a great advantage over other languages where you need to "pollute" your algorithms with "what if something bad happens here" types of statements in order to deal with user or system errors.

If you want to *throw* an exception from your method, you add a `throws` clause to the header such as

```
public void handleInput () throws IOException
```

You can keep throwing exceptions right up to a program's `main` method. If the `main` method throws an exception, then it gets handled by the Java virtual machine.

There is also an optional `finally` clause which follows the `catch` clauses in a method which handles exceptions. It contains a code block which gets executed after all of the `try`, and possibly `catch`, clauses have been processed (i.e. regardless of whether exceptions were generated during the processing of the `try` block).

2.4 Sockets

A *process* is an active computer program on some computer somewhere in the world. We are all familiar with personal computers which run many different processes (your program, the desktop, a network connection, email...) at the same time. Before the widespread use of parallel supercomputers, scientific programming was concerned with single-process applications which solved problems using a well-defined sequence of steps. On the other hand, much real-time programming for data acquisition and experimental control has needed to deal with *concurrent processes* which are active at the same time. Internet programming has some similarities with real-time systems where the concurrent processes can be located on distant computers.

Concurrent processes on the same or different computers can communicate using "sockets". The Java model for sockets treats them in a very similar way to files. Each socket can be associated with an input stream and an output stream; when your program sends data to a socket's output stream it looks just like you are writing to a file. But, in reality, your data might be being shipped all over the world. (The Java API also contains special classes for

communicating with web pages and you can even configure a Java program
to send people email.)

In order to establish a socket for communication to a remote server, a
Java program needs to know the Internet Protocol (IP) address, and the
port number, on the remote computer. Once your process (the "client") has
connected to the server then both processes are, in some sense, locked together.

2.4.1 A Socket Example: Requesting Data from a Server

In the following listing `SimplePlotClient` sends requests for data over the
internet and then plots the data as before. `SimplePlotClient` uses the IP
"localhost" address of the present computer and the port number 8004 (port
numbers in the 8000's are good ones to use for your networking experiments).
The requests are sent using a very simple "language" having only 3 commands.
Note that the server can accept these commands in any order.

YVALS Asks the server to return an `int` which gives the number of Y values.
This is followed by the array of y values.
XVALS Requests the array of X values in the same way as for the Y values.
CLOSE Closes the connection.

Listing 2.2. `SimplePlotClient` reads binary data from a socket and plots it.

```java
import java.io.*;
import java.net.*;
/* Interacts with the simple plot server */
public class SimplePlotClient
{
    public static void main(String[] args)
    {
        double[] xVals=null, yVals=null;
        int xLen=0, yLen=0;

        try
        {
            Socket s = new Socket("localhost",
                                        8004);
            PrintWriter out =
                new PrintWriter(s.getOutputStream(), true);

            DataInputStream in = new DataInputStream(
                                new BufferedInputStream(
                                        s.getInputStream()));
            out.println("YVALS");
            yLen = in.readInt();
            System.out.println("yLen = " + yLen);
            yVals=new double[yLen];
            for (int i = 0; i < yLen; i++)
```

```
                {
                    yVals[i] = in.readDouble();
                    if (i%100 == 0) System.out.println(yVals[i]);
                }
                out.println("XVALS");
                xLen = in.readInt();
                System.out.println("xLen = " + xLen);
                xVals=new double[xLen];
                for (int i = 0; i < xLen; i ++)
                {
                    xVals[i] = in.readDouble();
                    if (i%100 == 0) System.out.println(xVals[i]);
                }
                out.println("CLOSE");
                s.close();
                Plotter.plot(xVals,yVals,"test");
            }
            catch (IOException e) {System.out.println(e);}
    }
}
```

Here is the corresponding server program to Listing 2.2. Look at the two programs carefully and see how they correspond:

Listing 2.3. `SimplePlotServer` is the corresponding server class to `SimplePlotClient`.

```
import java.io.*;
import java.net.*;
public class SimplePlotServer
{
    public static void main(String[] args)
    {
        SimplePlotServer s = new SimplePlotServer();
    }
    public SimplePlotServer()
    {
        double[] xVals=null,yVals=null;
        int xLen=0,yLen=0;
        try
        {
            ServerSocket s = new ServerSocket(8004);
            Socket connection = s.accept();

            BufferedReader in = new BufferedReader
                (new InputStreamReader(
                            connection.getInputStream()));
            DataOutputStream out =
                            new DataOutputStream(
                            new BufferedOutputStream(
```

```java
                        connection.getOutputStream ()));
boolean done=false;
while (!done)
{
    String line = in.readLine ();
    if (line==null) done=true;
    else
    {
        if (line.trim ().equals ("YVALS"))
        {   // Read data from file
            File f2 = new File ("data_yVals");
            DataInputStream in2 =
                new DataInputStream (
                new BufferedInputStream (
                new FileInputStream (f2 )));
            yLen = in2.readInt ();
            System.out.println ("yLen = " + yLen );
            yVals=new double[yLen ];
            for (int i = 0; i < yLen; i ++ )
            {
                yVals [ i ] = in2.readDouble ();
            }
            // Write data down the socket
            out.writeInt (yLen );
            for (int i = 0; i < yLen; i ++ )
            {
                out.writeDouble (yVals [ i ]);
            }
            out.flush ();
            in2.close ();
        }
        if (line.trim ().equals ("XVALS"))
        {   // Read data from file
            File f3 = new File ("data_xVals");
            DataInputStream in3 =
                new DataInputStream (
                new BufferedInputStream (
                new FileInputStream (f3 )));
            xLen = in3.readInt ();
            System.out.println ("xLen = " + xLen );
            xVals=new double[xLen ];
            for (int i = 0; i < xLen; i ++ )
            {
                xVals [ i ] = in3.readDouble ();
            }
            // Write data down the socket
            out.writeInt (xLen );
            for (int i = 0; i < xLen; i ++ )
            {
```

```
                        out . writeDouble ( xVals [ i ] ) ;
                    }
                    out . flush ( ) ;
                    in3 . close ( ) ;
                }
                if  ( line . trim ( ) . equals ( "CLOSE" ) )
                                            done = true ;
            }
        }
        out . flush ( ) ;
        connection . close ( ) ;
    }
    catch  ( IOException  e )
    {
        System . out . println (
                    " Error  in  Simple  Plot  Server  "  + e ) ;
    }
    System . exit ( 0 ) ;
    }
}
```

2.5 Introduction to Threads

We will briefly consider the subject of threads in Java. This is so that you can
have a better idea of what the `MDSPlus` server is doing when it interacts with
multiple clients at the same time. The subject of threads in Java is important
and it can be quite complicated. This brief excursion into its territory might
be skipped over on a first reading of this book. Multi-threaded programs will
be the subject of Chapter 19.

   Threads are like independent processes except that threads share the same
data context (memory space) of the program which started them. Threads are
easy to create and destroy and communication between threads is much easier
than between processes. They are, therefore, useful candidates for parallel
processing.

   There are two ways to use threads in Java:

- to extend the `Thread` class, or
- to implement the `Runnable` interface.

The second method is available to you if the class you want to make into a
thread already extends another class.

   To start a new thread, you can create an instance of a class which extends
`Thread` or implements `Runnable`. This class *must have* a `run` method. You
then start the run method with a call to `start()`! This very strange pattern
is because a distinction needs to be made between starting an *independent*

thread and making a direct call to `run()` which would start it in the *same thread* as the caller.

2.5.1 Threaded Plot Server

Recall that our SimplePlotServer program of Sect. 2.4.1 had the following lines:

```
ServerSocket s = new ServerSocket(8004);
Socket connection = s.accept();
```

The server waits for a client to connect at the line `s.accept`. Once a client has connected, input and output streams are allocated and the server listens for a set of commands. In order to convert this program into a threaded plot server, we can encapsulate the server logic in the `run()` method of a new class which extends `Thread` and we spawn threads as follows:

```
try
{
    ServerSocket s = new ServerSocket(8004);
    do
    {
        Socket connection = s.accept();
        ThreadedPlotDataReader reader =
                new ThreadedPlotDataReader(connection);
        reader.start();
    }
    while (true);
}
```

This server program now functions in an infinite loop and you will need to crash it if you want it to stop! (Try typing `Crtl-C`.)

The `SimplePlotThreadedServer` class is shown below. You can test it out by running several clients at the same time in several different shell windows (remembering to crash the server when you have finished!) The `mdsip` simulator program, described in Appendix A.1, on page 227, has a structure which is very similar to `SimplePlotThreadedServer`.

Listing 2.4. Threaded version of a plot server.

```
import java.io.*;
import java.net.*;
public class SimplePlotThreadedServer
{
    public static void main(String[] args)
    {
        SimplePlotThreadedServer s =
                        new SimplePlotThreadedServer();
    }
    public SimplePlotThreadedServer()
```

```java
    {
        try
        {
            ServerSocket s = new ServerSocket(8004);
            do
            {
                Socket connection = s.accept();
                ThreadedPlotDataReader reader =
                    new ThreadedPlotDataReader(connection);
                reader.start();
            }
            while (true);
        }
        catch (IOException e)
        {
            System.out.println
                ("Error in SimplePlotThreadedServer " + e);
        }
        System.exit(0);
    }
}
class ThreadedPlotDataReader extends Thread
{
    Socket connection;
    double[] xVals=null, yVals=null;
    int xLen=0, yLen=0;
    public ThreadedPlotDataReader(Socket s)
    {
        this.connection = s;
    }
    public void run()
    {
        try
        {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                connection.getInputStream()));

            DataOutputStream out =
                            new DataOutputStream(
                            new BufferedOutputStream(
                            connection.getOutputStream()));
            boolean done=false;
            while (!done)
            {
                String line = in.readLine();
                if (line==null) done=true;
                else
                {
```

```
                        if ( line . trim (). equals ("YVALS" ))

                        ..... //( same as SimplePlotServer )

                        if ( line . trim (). equals ("CLOSE" ))
                                                done = true ;
                    }
                }
                out . flush ();
                connection . close ();
            }
            catch ( IOException e )
            {
                System . out . println (
                        "IO error in ThreadedPlotDataReader" );
            }
        }
    }
}
```

2.6 A Java API for `MDSplus`

In order to make progress in constructing `EScope`, we will need to have a Application Programming Interface (API) for accessing `MDSplus`. On the accompanying CD, we have provided three helper classes and one interface which will be all that you need in order to download and visualize typical data entries. Their full listings are provided in Appendix C. (These classes are a simplified version of the Java tools provided with the full `MDSplus` package. We admit that their structure is based on the historical development of `MDSplus` rather than being excellent examples of object-oriented design!)

The helper classes comprise the following:

- `MDSDescriptor` is a class which stores and describes a single `MDSplus` dataset. It contains a predefined set of constants which are used to flag the type of data. It also contains the data array itself and a number of methods for storing and accessing it. This class is a simplified version of the information which is stored at every data-containing node in the `MDSplus` tree. The beginning of this class is shown in Listing 2.5.
- `MDSMessage` is the class which negotiates remote access to `MDSplus`. Using the *mdsip* protocol described in Section 2.7.1, string "messages", which consist of a packet of header information and an expression to be evaluated, are sent to the server. The `MDSplus` server evaluates the expression and the result is received by `MDSMessage`, decoded into a `MDSDescriptor` and eventually returned to a client.
- `MDSNetworkSource` wraps up the negotiation with the `MDSplus` server into high-level methods like `connect`, `open`, `evaluate` and so on. All a client

program needs to know about are the method signatures for the some of these methods, so it makes sense to have this class implement an interface, `MDSDataSource`, which, in principle, is provided to the client software.

The way in which these classes work together is shown schematically in Fig. 2.6. Because data is eventually returned to the client program wrapped up in a `MDSDescriptor`, the client needs to know about this class as well as the `MDSDataSource` interface. The meaning of the symbols in this figure will be described later in this book.

The following sections will discuss further details of the data organization and the remote-data-access protocol of `MDSplus`. Their aim is to more fully motivate and describe the helper classes. If you have made a start reading the `MDSplus` web-site then you might find this discussion useful. For many readers, they can be skimmed over on a first pass (perhaps dwelling briefly on the discussion of `MDSNetworkSource` in Section 2.7.3).
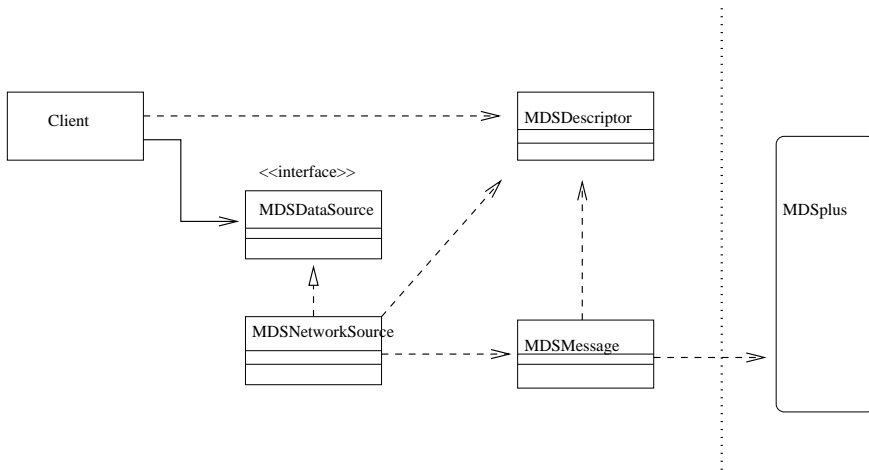
Fig. 2.2. Schematic representation of the helper classes `MDSDescriptor`, `MDSMessage`, `MDSNetworkSource` and the interface `MDSDataSource` showing the way in which they interact with a client class and the `MDSplus` database.

Listing 2.5. Part of the `MDSDescriptor` class used to hold data returned from MDSplus.

```
/** Used to store the response from the MDSPlus server */
public class MDSDescriptor
{   // Predefined constants flag the type of data:
    public static final byte MAX_DIM      = 8;
    // string or character data
    public static final byte DTYPE_CSTRING = 14;
```

```
    public static final byte DTYPE_CHAR     = 6;
    // integer data
    public static final byte DTYPE_BYTE     = 2;
    public static final byte DTYPE_SHORT    = 7;
             //(short is converted to int)
    public static final byte DTYPE_INT      = 8;
    // floating point data
    public static final byte DTYPE_FLOAT    = 10;
             //(float is converted to double)
    public static final byte DTYPE_DOUBLE   = 11;
    // unsigned word
    public static final byte DTYPE_WORDU    = 3;
             //("usigned word" is converted to int)
    // event
    public static final byte DTYPE_EVENT    = 99;

    // The following variables store the data:
    private byte    descriptorType;

    private byte    byteData[];
    private int     intData[];

    private double  doubleData[];

    private String  charData;
    private String  cstringData;
    private String  eventData;
    ....
}
```

2.7 The Data Organization of `MDSplus`

In this section we will briefly discuss the structure of the `MDSplus` database as well as the APIs for accessing and manipulating data. Readers may also wish to consult Appendix A for more information about installing and running `MDSplus`.

`MDSplus` assumes that the top level description of data is that of an "experiment". Experiments can be defined in the database to be comprised of a number of files corresponding to data recorded from physical diagnostics and control parameters. This files are arranged in a tree hierarchy which is something which suits fusion experiments which have a large number of diagnostics concerned with notions such as "radio-frequency power", "neutral-beam injection", "coil currents", "spectroscopy" and so on. A physical diagnostic within one of these categories might measure a number of signals so it is easy to imagine that a tree structure for the entire database would be 3 or 4 levels deep.

Each data record for an `MDSplus` experiment is given the name of a "shot" or "pulse". This is the popular jargon for a single plasma discharge: the ionized gas is pumped into the chamber, the magnetic coils are energized and the gas is heated to temperatures approaching those of the sun. Many diagnostics collect megabytes of data over periods of time ranging from some tens of milliseconds to several seconds. Many shots are taken during one experimental campaign.

Every dataset in an `MDSplus` experiment is identified by its path inside the tree structure. For example

`.OPERATIONS:I_FAULT`

specifies the `i_fault` (fault current) dataset which sits under the `operations` sub-tree (of the `h1data` experiment) Note that `MDSplus` names are case-insensitive and they begin with either a "." or a ":". It is also possible to associate a unique name, called a *tag*, with a dataset in order to simplify identification. Tags are a convenience mechanism and are often defined for the most frequently-accessed data, such as significant experimental results.

In `MDSplus` trees the leaves typically store data and internal nodes are typically used to specify the hierarchy without storing data. A dataset may specify a variety of information, ranging from string expressions to scalar values to scalar arrays to arrays composed of a sequence of pairs (with each pair describing a physical measurement and the sampling time). A data descriptor is associated with every data item in an `MDSplus` experiment and contains the definition of the the data type, its dimension and other information.

As mentioned above, `EScope` uses a simplified data interface to `MDSplus` and handles a subset of the `MDSplus` data types. Because, in `EScope`, we will be interested only in those data items which describe the evolution of some quantity over time, we will only need to handle arrays of data values and strings to describe data and error messages.

It often happens that a scientific user is not interested in a simple data item but, rather, in a combination of different items. For example, it might be the case that a user requires the values of one dataset to be scaled or offset by another: a data reference in a shot file `my_data` might need to be processed as

`my_data_gain * (2.3 * my_data) - my_data_offset`

For this reason, data access in `MDSplus` is, in general, carried out by providing generic *expression evaluations*. A specialized language, TDI ("Tree Data Interface"), is provided for this purpose. Expressions may also specify that a specific user-supplied analysis program gets executed at the time the expression is evaluated.

A detailed description of the TDI expression syntax is outside the scope of this book but it is important to realize that every data access in `MDSplus` corresponds to the evaluation of some kind of expression. The methods provided in the helper classes therefore define a string input argument representing the expression to be evaluated (possibly a simple reference to a data item) and return a descriptor object which describes the result of the evaluation.

2.7.1 The *mdsip* Protocol for Remote Data Access

The following discussion of the `mdsip` protocol for remote access to `MDSplus` can be skipped or skimmed on a first reading. It complements the discussion of `mdsip` in Appendix A.

Many modern database systems provide for remote data access, i.e. accessing the database from a machine which is different from that hosting the database itself over a network connection. Remote data access requires the following components:

- A database server, i.e. an application running on the machine hosting the database and accessing data on behalf of the client currently requesting the data.
- A client API, usually implemented as a library. The definitions of the methods defined in the API are usually close to the methods defined for local database access. The implementation of these methods handle network communication with the database server. Typically, a client application first connects to a database server. The established network connection remains active until the client disconnects from the database server or exits.
- A network protocol for data exchange. The client and the server need to communicate properly. Most database applications use the TCP/IP protocol for communication. TCP/IP allows the establishment of a connection, and reliable communication over it. Using a network protocol which guarantees reliable communication has the advantage that the code that handles the client-server communication does not need to handle the many problems which arise in network communication such as loss of data or the corruption of data packets. It is, nevertheless, necessary to build an application-specific protocol over the network layer.

The remote-data-access layer of `MDSplus` defines its own protocol over TCP/IP, called `mdsip`. In a data-access transaction, the client sends a string specifying the expression to be evaluated and the `mdsip` server returns the result of the evaluation. As different kinds of data can be returned by expression evaluations, `mdsip` specifies the transmission of a descriptive header, followed by the data itself. The client can handle the reception of a different number of bytes because this information is available after the fixed-length header has been received. (In TCP/IP it is necessary to know the number of bytes being received in advance since this information is not provided by the network layer.)

The `MDSMessage` class, defines all the information required for sending an expression as a string to the server and retrieving the result of the evaluation. The "packet" exchanged via TCP/IP with `MDSplus` contains the following information:

- `msglen`: the total length of the message in bytes.

- `status`: the status of the expression evaluation. This field is valid only when the message carries the result of the expression evaluation, i.e. when it is transmitted by the `MDSplus` server back to the client.
- `length`: the length in bytes of the *data field* included in the message. The total length of the message is, therefore, the sum of the data length and of the length of the associated header (which is 48 bytes).
- `nargs`: the number of arguments. The expressions being evaluated are described as strings, but it is possible to define additional arguments, which are sent by the client to the server and then used in the expression evaluation. This feature is useful, for example, when the `mdsip` protocol is used for application servers in which powerful computers are used to run simulation codes to model experimental results. In this case the expression being evaluated can invoke a separate simulation program. It is interesting to observe that the syntax can generalize to cases where the arguments might contain a huge string of data values: The arguments can be identified in the expression by the symbols $1, $2, ..$n, and the message will be followed by other messages containing a binary encoding of the arguments. This is beyond the scope of this book.
- `descr_idx`: the "descriptor index". As noted above, when arguments are defined in the expression to be evaluated, more `mdsip` messages may need to be sent by the client. The server knows the number of arguments it is going to receive after receiving the fist message. The `descr_idx` labels the subsequent arguments. This is beyond the scope of this book.
- `dtype`: the type of the data associated with this message. In `EScope` only string messages will be sent to the server so `dtype` will always be `DTYPE_CSTRING`. On return from the server, `dtype` can take the range of values shown in the `MDSDescriptor` class.
- `client_type`: the "type" of the client computer. This is to register the "endianness" and IEEE floating point encoding of the client computer with the `mdsip` server. Because the byte order of data storage varies between computer manufacturers, it may be the case that bytes transmitted in "big endian" format over a network need to be "swapped" for the client machine. When the `mdsip` server receives the `client_type` flag, it decides whether to swap bytes for transmitting the result of the expression evaluation. At the time of writing, `mdsip` violates the convention that data transmission over TCP/IP should be big endian. Instead it determines the endian type of the client machine and modifies the data accordingly. This may result in an efficiency gain under some circumstances. (The `client_type` flag also determines whether any translation of floating point format needs to be done before transmission of data by the server. We shall hide its complexities inside the `MDSMessage` class and forget about it. Such is the situation with some real-world case studies!)
- `msgid`: an identifier for the message which is copied by the `MDSplus` server onto its reply. This identifier can be used to label messages in, for example, situations where a threaded client might be making several requests in

different threads, possibly with several `mdsip` servers. In this case the `msgid` field can be used to properly associate answer with requests.

- `ndims`: the number of dimensions in an array of binary data. We will not make use of this field in our `EScope` case study as we will only be concerned with one-dimensional arrays.
- `dimensions`: a fixed size array of integers specifying each dimension (up to eight dimensions). This field and the the previous one are used to handle multidimensional data, serialized in row-first order in the associated data buffer. Once again, this will not be relevant to our `EScope`.
- `body`: the data buffer containing serialized data. The reconstruction of the transmitted data from the content of this buffer is unambiguous when combined with the contents of the `mdsip` header (particularly `dtype`, `ndims` and `dimensions`).

2.7.2 Operation of `MDSMessage`

The above variables define the `mdsip` protocol, being the set of rules the client and server must adhere to to communicate unambiguously. The constructor of `MDSMessage`, shown in Appendix C.4, accepts a string argument and constructs a `mdsip` header corresponding to a request to evaluate the expression corresponding to this argument. The method `send()` had an instance of `DataOutputStream` as its argument. Using a Java `DataOutputStream` for sending messages, and a `DataInputStream` for message reception, simplifies the serialization process because `DataOutputStream` provides the correct serialization in its `write...` methods. The `mdsip` protocol defines twos-complement and IEEE 754 formats to serialize integer and float numbers, respectively. These formats are supported by the `DataInputStream` and `DataOutputStream` classes in their `write..` and `read..` methods.

By way of illustration, the `MDSMessage` method `receive(DataInput-Stream s)` reads the incoming `mdsip` message from the `DataInputStream` instance "`s`" and performs the following steps:

1. Fill in the `MDSMessage` fields from the fixed length header. This operation is done in two steps: first a fixed-length byte array corresponding to the bytes of the message header is read from the input stream. Then the desired fields are retrieved, using the support methods `byteArrayToInt`, `byteArrayToShort` and `byteArrayToShort`.
2. Read the rest of the message. When the header has been decoded, the number of incoming bytes is known and it is possible to decode them correctly, based on the knowledge of the data type and of the dimensions.

Once the incoming message has been decoded, the received data, converted into Java data types, is returned by method `receive`. The result is encapsulated into an instance of the `MDSDescriptor` class. This class describes a generic scalar or array data item and resembles the descriptor structure defined in `MDSplus`.

2.7.3 Operation of `MDSNetworkSource`

As we have just seen, `MDSMessage` is responsible for the proper management of the `mdsip` network protocol once a proper network connection has been established. Managing the connection to `MDSplus` is done by the `MDSNetworkSource` class whose public interface has the following methods:

- `connect(String serverAddrCPort)`: represents the first action carried out to establish a connection. The string argument specifies the IP address and the port number of the `mdsip` server (separated by a colon in the string). The method establishes a socket and input and output streams as we did in our earlier `SimplePlotClient` example.
- `open(String experiment, int shot)`: opens the specified experiment and shot database. This is done by sending a special expression `JAVAOPEN(experiment,shot)` to `MDSplus`.
- `evaluate(String expression)` calls the `MDSMessage send` and `receive` methods to evaluate the string expression in its argument. The result is returned as a `MDSDescriptor` object.
- `close()` closes a currently opened shot database.
- `disconnect()` terminates a currently established network connection to `MDSplus`.

    At any time a connection to a `mdsip` server is in one of the following states:

- *initial*: no connection has been established.
- *connected*: a connection has been established, but no shot database has been opened.
- *open*: a shot database is currently open.
- *closed*: a previously opened shot database has been closed and no other database is now open.
- *disconnected*: a network connection to a `mdsip` server has been closed and no other connection is currently active.

    The behavior of the methods of `MDSNetworkSource` depends on the current state of the connection. For example, the method `evaluate` can be successfully executed if and only if the connection is open. The implementation of this method checks the status of flags `isConnected` and `isOpen` to determine whether or not the expression can be sent. Exceptions are thrown if it cannot be sent. We will return to the structure of this class as our case study evolves.

2.8 `PreEScope0`: A Program to Connect to `MDSPlus`

We would like to propose that this section be accomplished as an exercise by the reader. It is not "compulsory" and you could turn to Appendix D to find

the solution if you wished. But it is only a small step beyond what you have learnt from the earlier examples in this chapter and you can use the supplied helper classes to connect to the `MDSplus` database or to the simulator program described in Appendix A.1.

The specifications of this program are as follows:

1. It will run in the command line and take four arguments: the server address and port (in the form "address:portNo"), the name of the experiment, the shot number and the name of a dataset to be downloaded.
2. After checking that the arguments have been supplied correctly, the program will open a connection with the `MDSplus` database and download the data corresponding to the particular dataset. One example dataset is the ".`operations:i_fault`" leaf node.
3. The X and Y data arrays of this dataset are then written to two binary files `data_yVals` and `data_xVals`.
4. The `Plotter.plot` method will be called to plot the data.

Does this sound familiar? In fact, this program will set up the files that you need to run `SimplePlot`. And you can run `SimplePlot` to test it out.

You can also try sending other commands to the `MDSplus` server and printing out its response. For example

- Sending "`10+2`" should return the number "`12`".
- Sending "`units_of(.operations:i_fault)`" should return a string "Amps".
- Other datasets such as `.operations:diamag`, ... can be downloaded, plotted up and written to file.

2.9 Programming Exercises

The following exercises will help readers become familiar with the parts of the Java API described in this chapter.

1. As well as the binary input classes, Java has a completely analogous set of classes to *output* data to binary files. You can look these up in Sun's Java API documentation and then try modifying the `SimplePlot` program to write out the data arrays into different files. You can then read the data back in from these new files to make sure that they have been written correctly. You could make this slightly more complicated by writing the data into files in different directories on your computer.
2. You could now try to modify `SimplePlot` to read text data. Just type some X and Y data into two text files using an editor. Calling your modified `SimplePlot` will now construct an unlabelled line graph of the data you typed! Modify the data to draw some interesting waveform shapes.
3. You could now write a program to solicit a number of (X,Y) data pairs from a user and then plot them up.

4. If you have already converted `SimplePlot` to read text data, you will be forced to handle `NumberFormatException`s when you convert this data to Java primitive types. You might wish to practice catching these exceptions and testing your program with good and bad data.

5. Download the `SimplePlotServer` examples from the CD to your computer. You run them by starting the server from a different terminal window to the client or by running it in background from the same terminal window. Play with the examples and verify that the commands can be sent in a different order from the client to the server. You might then like to set up your own client/server system using a simple language to talk between them:

   ```
   CLIENT: Hello
   SERVER: Hello. I am a server. How can I help?
   CLIENT: Time
   SERVER: 1126870937252
   ```

6. The `SimplePlotServer` program exits if a client sends a `CLOSE` command or a null line. You might think that it would handle multiple clients if it were modified to have an infinite server loop. This turns out not to be the case and, as described in the next section, you need to turn to Java threads to build a true, multiple-client server.

   Try out making the loop infinite by putting a line "`done =false;`" at the end of the `SimplePlotServer` while-loop. If you start one client then you will see the graph. If you start another client then you will not see another graph and the client will hang. Remember to stop the server as well, by crashing it, when you have finished with this exercise. (On Linux, you can find the process number of the server by typing `ps` and looking for `java`. You can "kill" this process by typing `kill -9` followed by the process number.)

2.10 Further Reading

Sun's web documentation on the Java API [12] and Java Tutorial [13] are book-marked by every Java programmer.

There are many Java programming texts which are suitable for background reading about Java IO. We particularly acknowledge, and recommend, those by Hunt ([14]), Horstmann ([15]) and Horstmann and Cornell ([16]).