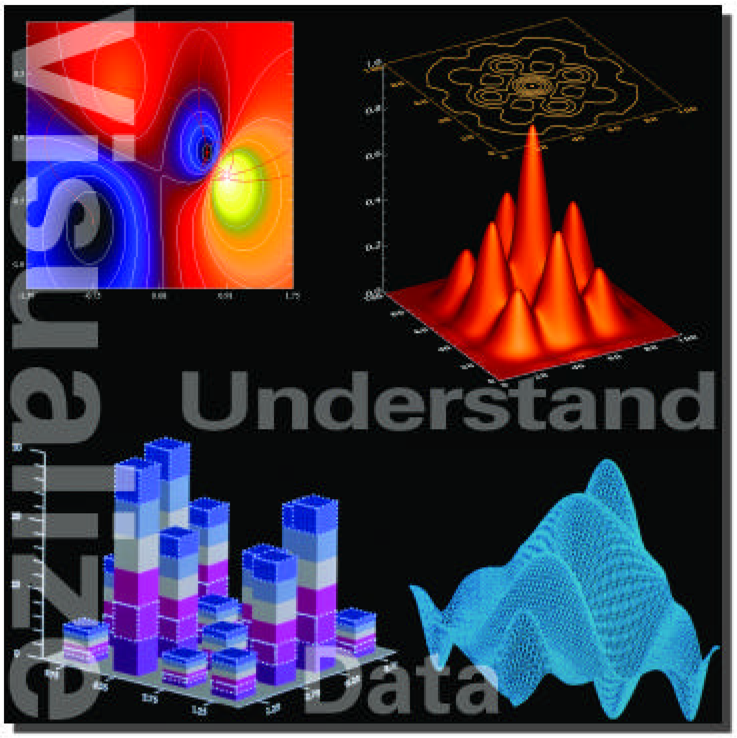
# JWAVE® 3.5



User's Guide

HELPING CUSTOMERS **SOLVE** COMPLEX PROBLEMS

## Visual Numerics, Inc.

| | | |
|---|---|---|
| **Visual Numerics, Inc.**<br>**2500 Wilcrest Drive**<br>**Suite 200**<br>**Houston, Texas  77042-2579**<br>**United States of America**<br>**713-784-3131**<br>**800-222-4675**<br>**(FAX) 713-781-9260**<br>**http://www.vni.com**<br>**e-mail: info@boulder.vni.com** | **Visual Numerics, Inc. (France) S.A.R.L.**<br>**Tour Europe**<br>**33 place des Corolles**<br>**Cedex 07**<br>**92049 PARIS LA DEFENSE**<br>**FRANCE**<br>**+33-1-46-93-94-20**<br>**(FAX) +33-1-46-93-94-39**<br>**e-mail: info@vni-paris.fr** | **Visual Numerics International, Ltd.**<br>**Suite 1**<br>**Centennial Court**<br>**East Hampstead Road**<br>**Bracknell, Berkshire**<br>**RG 12 1 YQ**<br>**UNITED KINGDOM**<br>**+01-344-458-700**<br>**(FAX) +01-344-458-748**<br>**e-mail: info@vniuk.co.uk** |
| **Visual Numerics, Inc.**<br>**7/F, #510, Sect. 5**<br>**Chung Hsiao E. Rd.**<br>**Taipei, Taiwan  110 ROC**<br>**+886-2-727-2255**<br>**(FAX) +886-2-727-6798**<br>**e-mail: info@vni.com.tw** | **Visual Numerics International GmbH**<br>**Zettachring 10**<br>**D-70567 Stuttgart**<br>**GERMANY**<br>**+49-711-13287-0**<br>**(FAX) +49-711-13287-99**<br>**e-mail: info@visual-numerics.de** | **Visual Numerics Japan, Inc.**<br>**Gobancho Hikari Building, 4th Floor**<br>**14 Gobancho**<br>**Chiyoda-Ku, Tokyo, 102**<br>**JAPAN**<br>**+81-3-5211-7760**<br>**(FAX) +81-3-5211-7769**<br>**e-mail: vda-sprt@vnij.co.jp** |
| **VIsual Numerics S.A. de C.V.**<br>**Cerrada de Berna 3, Tercer Piso**<br>**Col. Juarez**<br>**Mexico, D.F. C.P. 06600**<br>**Mexico** | **Visual Numerics, Inc., Korea**<br>**Rm. 801, Hanshin Bldg.**<br>**136-1, Mapo-dong, Mapo-gu**<br>**Seoul 121-050**<br>**Korea** | |

# Table of Contents

CHAPTER

# Preface

This manual is the user's guide for JWAVE$^{TM}$. It gives an overview of the JWAVE system, explains how to create client-side (Java$^{TM}$) applications and server-side (PV-WAVE$^{TM}$) wrapper functions, and includes advanced topics and JWAVE reference information.

## How to Use This Manual

JWAVE, as a system, can be divided into server-side and client-side components. To develop the server-side components (called JWAVE wrapper functions), you must be familiar with the PV-WAVE programming language. Client-side JWAVE components, on the other hand, are written in Java and require Java programming expertise. This manual addresses both server-side and client-side JWAVE developers. A third audience, system administrators and Webmasters, is addressed in a chapter on configuring the JWAVE system.

**TIP** Because the server and client-side components of a JWAVE system are closely related, developers must coordinate their efforts. We recommend that all JWAVE developers (client and server developers) read the introductory chapters, Chapter 1, *JWAVE System Introduction* and Chapter 2, *The Generic JWAVE Applet*, to get an overview of JWAVE. Also, you can refer to Appendix F, *Glossary* for general information about JWAVE.

## Server-Side Developers

If you are writing server-side components (JWAVE wrapper functions), then you can focus on Chapter 5, *JWAVE Server Development*. This chapter explains the mechanisms by which JWAVE wrapper functions retrieve and unpack parameters from a Java client application.

If you want to take advantage of JSPs and servlets to create applications that do not require client-side Java, see Chapter 10, *JSPs, Servlets, and JWAVE*.

---

**TIP** It is helpful for server-side developers to understand how parameters and data are set in client applications before being sent to the server. In particular, be familiar with the Java methods that are used to set parameters in client applications. You can find information on these methods in Chapter 3, *JWAVE Client Development*.

---

## Client-Side Developers

If you are writing client-side applications, you must be a Java programmer. Focus on Chapter 3, *JWAVE Client Development* and Chapter 4, *JWAVE Graphics*. These chapters describe the basic ingredients of JWAVE client applications. For a discussion of more advanced topics, see Chapter 6, *Managing Data* and Chapter 7, *Using JWAVE Beans*.

Again, client-side developers must coordinate their efforts with developers of server-side JWAVE programs. The client-side JWAVE application is, in effect, an interface to a PV-WAVE application on the server. The PV-WAVE programmer who develops the server-side JWAVE wrappers must know what kinds of plots the client intends to produce, the types of parameters that will be passed, and the types of data to expect.

Some convenience classes are provided with JWAVE that let you easily add functionality such as zooming and profile plots to applets. For information on these classes, see Chapter 9, *Advanced Graphics Features*.

## System Managers and Webmasters

Chapter 8, *JWAVE Server Configuration* explains how to set up and configure the JWAVE server software.

Detailed information on installing JWAVE is in the CD booklet. Additional information on installing, configuring, and using JWAVE can be found in the files `Release_Notes.html` and `Tips` in:

**(UNIX)** `VNI_DIR/jwave-3_5`

**(Windows)** `VNI_DIR\jwave-3_5`

where `VNI_DIR` is the main Visual Numerics installation directory.

## *What's in this Manual*

This manual explains how to use JWAVE. With JWAVE, you can create visual and numerical analysis applications written entirely in Java, where the Java client application communicates directly with PV‑WAVE running on a server.

**Chapter 1, *JWAVE System Introduction*** — Describes the client/server architecture of a JWAVE system and gives an example JWAVE application and wrapper.

**Chapter 2, *The Generic JWAVE Applet*** — Uses a simple example to demonstrate how to use JWAVE for publishing PV‑WAVE graphics on a Web page.

**Chapter 3, *JWAVE Client Development*** — Explains how parameters and data are passed from the client and retrieved from the server.

**Chapter 4, *JWAVE Graphics*** — Describes the JWaveView class for returning graphics to the client.

**Chapter 5, *JWAVE Server Development*** — Describes development of JWAVE wrapper functions.

**Chapter 6, *Managing Data*** — Describes the use of JWAVE data proxies.

**Chapter 7, *Using JWAVE Beans*** — Explains how to use and develop JWAVE Beans.

**Chapter 8, *JWAVE Server Configuration*** — Explains the installation directory structures for the client and the server, how to start and stop the JWAVE Manager, and how to configure and test the server.

**Chapter 9, *Advanced Graphics Features*** — Discusses convenience classes that let you easily add functionality to JWAVE applets.

**Chapter 10, *JSPs, Servlets, and JWAVE*** — Explains how to use JSPs and servlets with JWAVE to create dynamically generated Web content.

**Chapter A, *JWAVE Wrapper API*** — Describes the JWAVE component functions that are used in JWAVE wrappers.

**Chapter B, *JWAVE Convenience Wrappers*** — Describes the JWAVE convenience wrapper functions.

**Chapter C, *Keyword and Named Color Parameters*** — Describes the keywords and parameters that can be used with JWAVE convenience wrappers.

**Chapter D, *HTTP Configuration File*** — Describes a configuration file used to configure the JWAVE HTTP Web server.

**Chapter E, *JWAVE Bean Tools Reference*** — Describes the input parameters and customizer features for JWAVE Beans Tools.

**Chapter F, *Glossary*** — Defines JWAVE terms and concepts.

***JWAVE Index***

## Technical Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

| **Office Location** | **Phone Number** |
|---|---|
| Corporate Headquarters<br>Houston, Texas | 713-784-3131 |
| Boulder, Colorado | 303-939-8920 |
| France | +33-1-46-93-94-20 |
| Germany | +49-711-13287-0 |
| Japan | +81-3-5211-7760 |
| Korea | +82-2-3273-2633 |
| Mexico | +52-5-514-9730 |
| Taiwan | +886-2-727-2255 |
| United Kingdom | +44-1-344-458-700 |

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

## FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

| **Office Location** | **FAX Number** |
|---|---|
| Corporate Headquarters | 713-781-9260 |
| Boulder, Colorado | 303-245-5301 |
| France | +33-1-46-93-94-39 |
| Germany | +49-711-13287-99 |
| Japan | +81-3-5211-7769 |
| Korea | +82-2-3273-2634 |
| Mexico | +52-5-514-4873 |
| Taiwan | +886-2-727-6798 |
| United Kingdom | +44-1-344-458-748 |

or by sending E-mail to:

| **Office Location** | **E-mail Address** |
|---|---|
| Boulder, Colorado | `support@boulder.vni.com` |
| France | `support@vni-paris.fr` |
| Germany | `support@visual-numerics.de` |
| Japan | `vda-sprt@vnij.co.jp` |
| Korea | `support@vni.co.kr` |
| Taiwan | `support@vni.com.tw` |
| United Kingdom | `support@vniuk.co.uk` |

## Electronic Services

| Service | Address |
|---|---|
| General e-mail | `info@boulder.vni.com` |
| Support e-mail | `support@boulder.vni.com` |
| World Wide Web | `http://www.vni.com` |
| Anonymous FTP | `ftp.boulder.vni.com` |
| FTP Using URL | `ftp://ftp.boulder.vni.com/VNI/` |
| PV-WAVE Mailing List: | `Majordomo@boulder.vni.com` |
| To subscribe include: | `subscribe pv-wave YourEmailAddress` |
| To post messages | `pv-wave@boulder.vni.com` |

1

CHAPTER

# JWAVE System Introduction

This chapter gives a quick overview of the basic client and server components of JWAVE.

JWAVE lets you create Java client applications that communicate directly with PV-WAVE running on a remote server. On the server side, PV-WAVE code is used to analyze data and generate graphics. On the client side, a Java applet (or application) lets users interact with the PV-WAVE session and display the graphics returned from PV-WAVE.

JWAVE offers four separate client/server connection models:

*Figure 1-1* shows a typical JWAVE system, consisting of client Java applications that communicate with a PV-WAVE server via an HTTP connection (a Web server connection). In this model, you can use any Web server that you wish.

*Figure 1-2* shows a JWAVE client that is connected to the server via a direct socket connection.

*Figure 1-3* shows a JWAVE client that is connected to the JWAVE Web server via an HTTP connection. The JWAVE Web is bundled with JWAVE. It handles client connections and manages JWAVE sessions on the server.

*Figure 1-4* shows a JWAVE client that is connected to the server via the JWAVE Servlet. The JWAVE Servlet plugs into any Web server that accepts servlets.

All of these connection methods achieve the same result: parameters and data can be passed between the client applet/application and PV-WAVE running on the server.

**NOTE** HTTP and socket connection methods can be used simultaneously by multiple JWAVE client applications; the JWAVE server can respond to several client applications at the same time.

Briefly, a client-side Java application with JWAVE components connects to a JWAVE server, usually across the Internet or an intranet. On the server, a process called the JWAVE Manager “listens” for client connections. When a connection is made, the JWAVE Manager starts a PV-WAVE session and executes a “wrapper function.” This wrapper function is a PV-WAVE function that contains JWAVE-specific calls for passing parameters and data back and forth to the Java client. For instance, PV-WAVE may receive a 2D array of image data from the client, process the data, and send a plot back to the client where it is displayed.

**TIP** In general, you must be a Java developer to develop client-side JWAVE applications or applets. If your Java experience is limited, you can still create useful JWAVE applications using the generic JWAVE applet described in Chapter 2, *The Generic JWAVE Applet*.

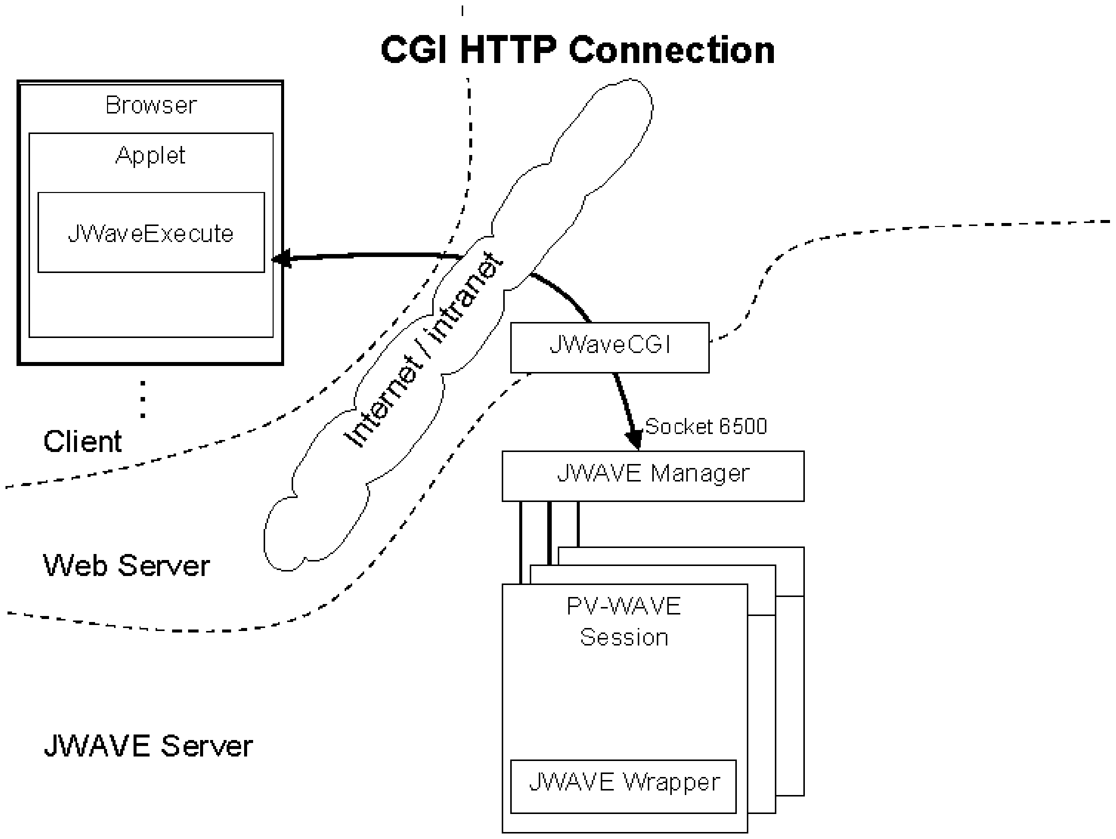


**Figure 1-1** JWAVE client-server configuration. Java applets (or applications) communicate with a JWAVE server through HTTP connections. In this model, the client contacts a CGI program running on a Web server. The CGI then starts the JWAVE Manager. You can use this type of connection simultaneously with a direct socket connection, shown in *Figure 1-2*.

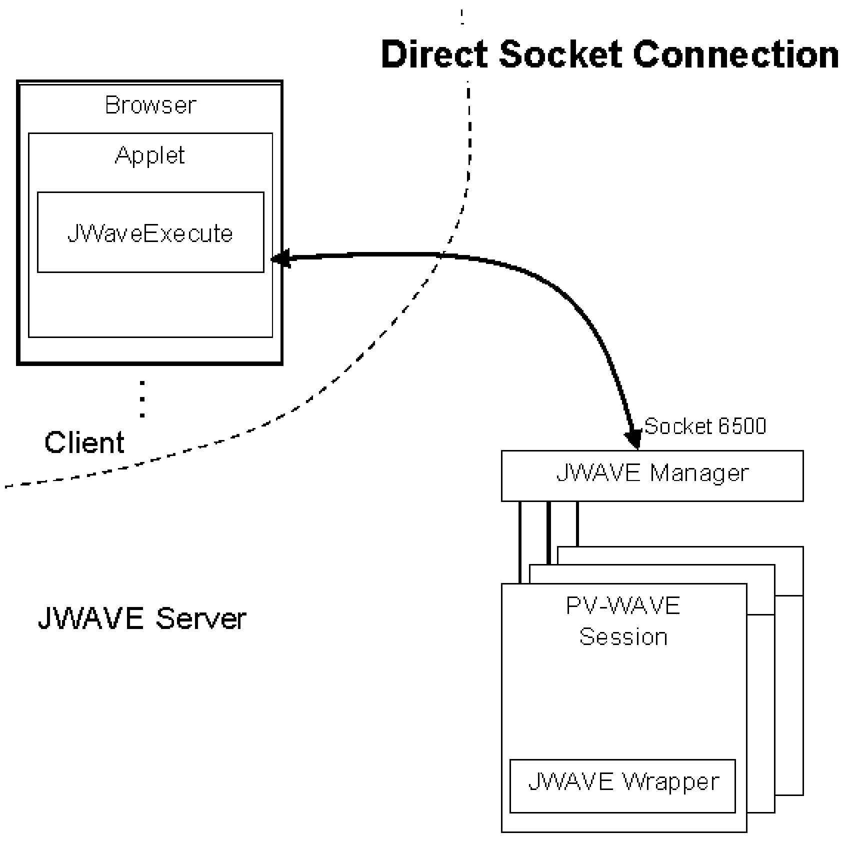


**Figure 1-2** JWAVE client-server configuration. Java applets (or applications) communicate with a JWAVE server via direct socket connections. You can use this type of connection simultaneously with an HTTP connection, shown in *Figure 1-1*.

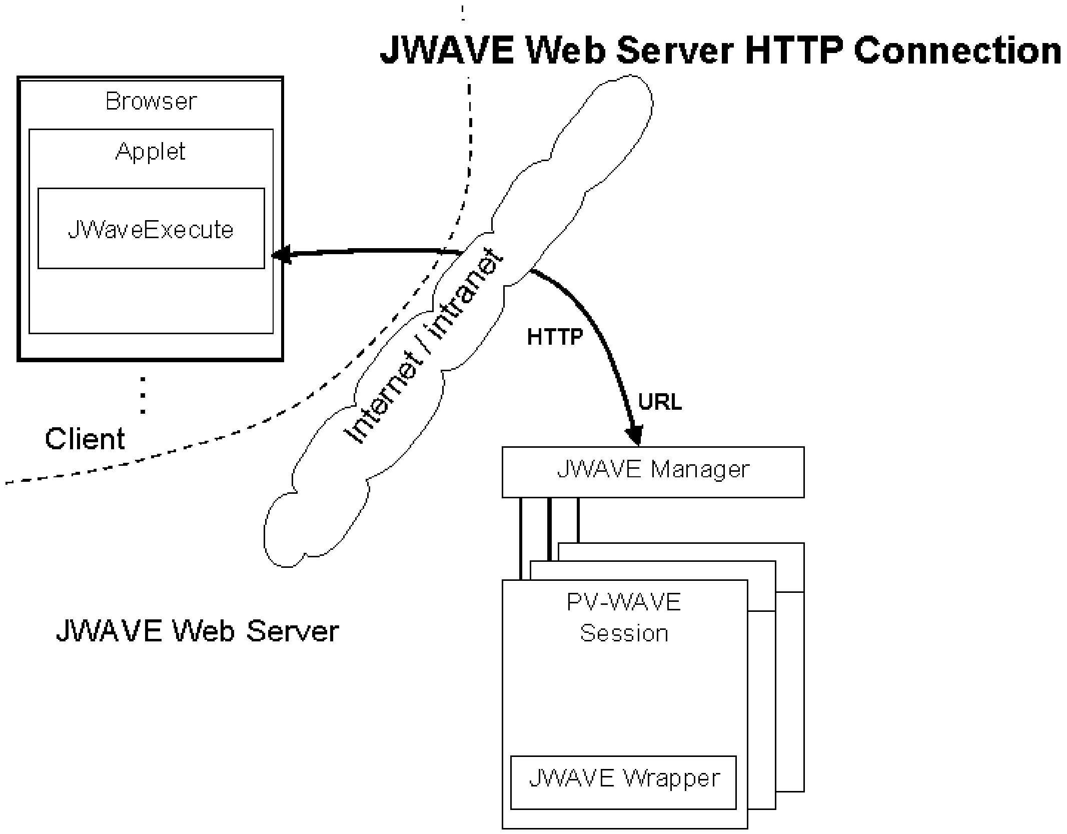


**Figure 1-3** Java applets (or applications) communicate directly with the JWAVE Web server. The JWAVE Web Server includes the JWAVE Manager, therefore, the client can connect to the JWAVE Manager directly with a URL address.

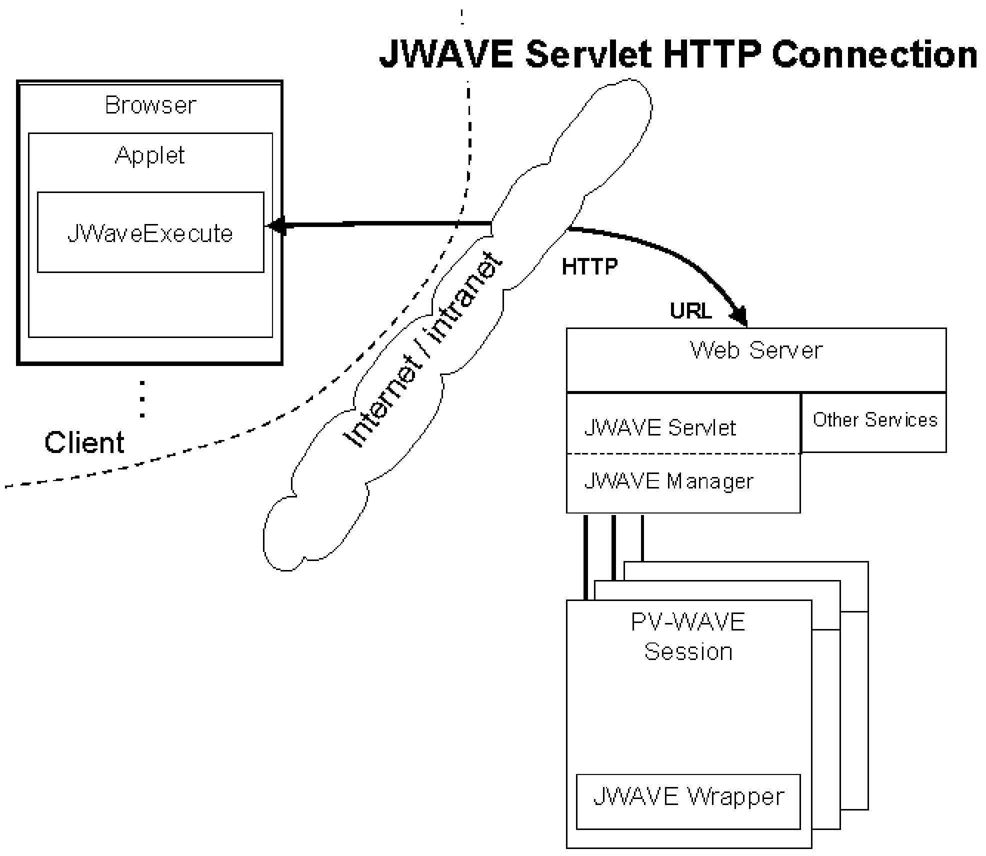


**Figure 1-4** Java applets (or applications) communicate with a Web server using the JWAVE Servlet. The JWAVE Servlet plugs into any Web server that accepts servlets. Once the JWAVE Servlet is configured properly, clients can connect to the JWAVE Manager directly with a URL address.

## The Client Side

The client side of a JWAVE system consists of a Java application or applet.

**NOTE** A Java applet is run within another application, usually a Web browser. A Java application runs on its own, without the need for another controlling program. The distinction is usually irrelevant for JWAVE (with the exception of applet security restrictions), as the applet/application decision depends on how the users access the program, rather than on what the program does.

The client application developer provides a user interface for interacting with the JWAVE server. For example, the client application might provide interactive buttons, text fields, and menus that allow the user to choose the type of plot to create, specify the plot characteristics, import data, filter data, and so on.

JWAVE allows client applications to remain thin, because all of the data analysis can be done on the server. Typical client applications allow users to run remote PV-WAVE applications on the server and then retrieve only the desired results (such as plots and/or analysis results).

*Figure 1-5* shows a basic client configuration, where the client happens to be an applet running in a browser.

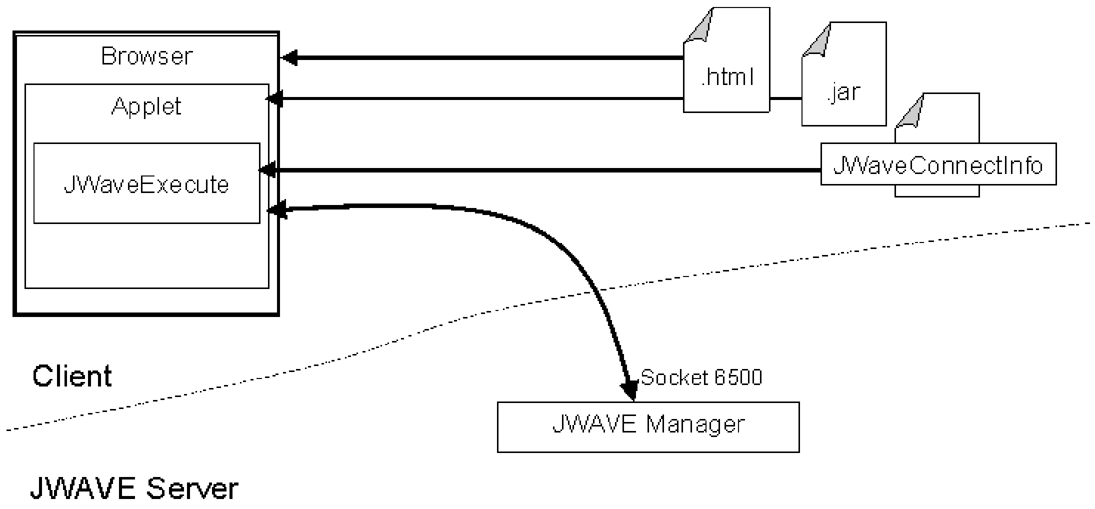


**Figure 1-5** JWAVE client applet. A JWAVE applet requires JWAVE Java classes (in the JAR file) and configuration information. The browser uses an HTML page to load the applet.

As *Figure 1-5* shows, the JWAVE applet (or application), written in Java, requires JWAVE class packages, which are bundled in a Java Archive (JAR) file. This JAR file is located in:

**(UNIX)** `VNI_DIR/classes/JWave.jar`

**(Windows)** `VNI_DIR\classes\JWave.jar`

where `VNI_DIR` is the main Visual Numerics installation directory.

These classes provide the means for Java applications to:

- connect to a JWAVE server
- execute PV-WAVE functions
- pass parameters and data to the server
- retrieve parameters and data from the server

The `JWaveConnectInfo.jar` file is located in the same directory as the `JWave.jar` file. This file provides information necessary (for example, a server socket ID) for the applet to contact the remote JWAVE server.

---

**TIP** If you want to use an IDE for JavaBeans application development, see Chapter *7, Using JWAVE Beans*.

---

## The Server Side

On the server side, the JWAVE Manager listens for client connections and manages individual PV-WAVE sessions.

*Figure 1-6* shows the basic configuration of the JWAVE server components.

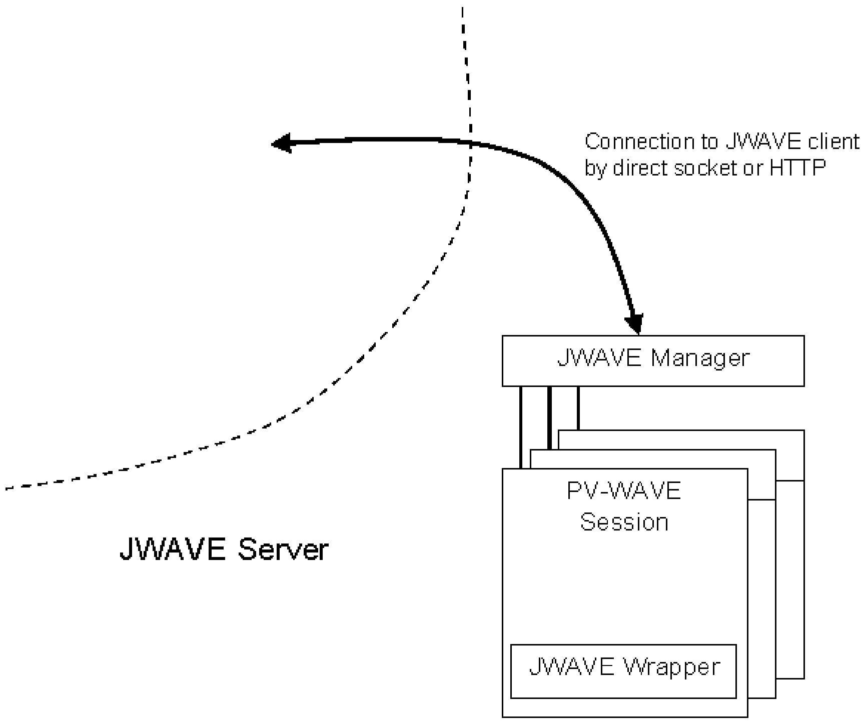


**Figure 1-6** The server side of a JWAVE system. The JWAVE Manager application listens for client connections and takes appropriate actions, such as starting a PV-WAVE session.

### The JWAVE Manager

The JWAVE Manager is a program that runs on the server and listens for client connections. When a connection is made, the JWAVE Manager examines the request from the client and processes it appropriately. If the client request is for an initial contact to a PV-WAVE session, then one is started. If the client request is for a session that is already running, then that session is contacted. The JWAVE Manager sends all parameters and data from the client to the PV-WAVE session. When the request is completed, the reply (such as data or a plot) from PV-WAVE is returned by the Manager to the client. If the client contacts the server with HTTP, then the connection request is routed through a CGI (Common Gateway Interface) program, which then contacts the JWAVE Manager.

The JWAVE Manager can manage multiple PV-WAVE sessions, or a client application can make several sequential requests using the same PV-WAVE session. This allows faster updates (as new sessions are not started for each request). This also allows data to remain with the session (in memory) for use by subsequent requests (rather than making round trips back to the client).

The JWAVE Manager also handles administrative functions, such as starting, shutting down, and configuring the JWAVE server. A script called `manager` (`manager.bat` on Windows) is used to control and configure the JWAVE Manager. This script is described in detail in Chapter 8, *JWAVE Server Configuration*.

As shown in *Figure 1-7*, the JWAVE Manager uses configuration information from the file `jwave.cfg`. (By default, output is sent to the terminal.) For information on using the `manager` command and changing server configuration, see *Setting Up the JWAVE Server* on page 112.

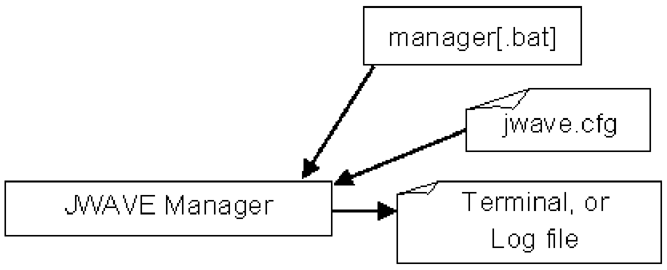


**Figure 1-7** JWAVE Manager handles activity on the JWAVE server

## PV-WAVE Sessions

PV-WAVE sessions are started by the JWAVE Manager, as described previously in this section. PV-WAVE performs the actual data analysis and generates graphics.

The individual PV-WAVE sessions log their output, by default, to the terminal. You can change this default (send information to a log file) using the Configuration Tool. For more information, see *Setting Up the JWAVE Server* on page 112.

## JWAVE Wrappers

A JWAVE wrapper is a PV-WAVE function that includes JWAVE-specific library function calls. These functions enable PV-WAVE to communicate with a remote Java client. For example, the PV-WAVE function GETPARAM retrieves and "unpacks" parameters and data sent from the client. Once unpacked, the parameters and data can be used within the wrapper function or passed to other PV-WAVE routines. Then, when a JWAVE wrapper function returns, the JWAVE Manager automatically sends the parameters, data, and any graphics that were produced by the JWAVE wrapper function back to the client.

**TIP** These functions are called "wrappers" because they "wrap" a PV-WAVE application with JWAVE-specific routines (such as GETPARAM).

For detailed information on writing JWAVE wrapper functions, see Chapter 5, *JWAVE Server Development*.

## PV-WAVE Applications

In most cases, the JWAVE wrapper function is used as a bridge to run a PV-WAVE application. This application can perform most of the functions of PV-WAVE, such as:

- accepting data and parameters from the wrapper
- reading data from files, databases, and so on
- using the extensive mathematics, statistics, and analysis capability of PV-WAVE
- producing plots, images, and other graphical representations of data
- saving files
- returning data

**NOTE** The regular PV-WAVE functions that JWAVE wrappers cannot use mainly include the user interface features, such as WAVE Widgets and VDA Tools.

Next, we will look at a simple JWAVE example where a client application sends data to the server, the server (PV-WAVE) processes the data, and a result is returned to the client.

## A Simple Example

This example shows a simple JWAVE application that processes a transaction between client and server. The Java client sends a number to PV-WAVE, and PV-WAVE returns the square root of that number back to the client.

We've kept this example simple to demonstrate the fundamental building blocks of a JWAVE application: creating a connection, passing parameters and data to from the client to the server, and retrieving results from the server. Later, we will discuss more complex, and practical, scenarios.

This example includes:

- Java client application code
- JWAVE wrapper code
- Sample output

### The Client Java Application

A Java application (or applet) resides on the client side of a JWAVE system. Typically, the Java client provides an interface to a PV-WAVE application running on a remote server. The Java client can send information, including data, to the server for processing. How the processed data is handled is up to the JWAVE programmer. Typically, the client displays graphically the data returned from the server.

Example 1-1 shows the Java client application. This client simply sends a number to the JWAVE server, and then prints out a result that is returned from the server. In this case, PV-WAVE executes a function that returns the square root of the number sent from the client.

**TIP** You can find the code for this function in:

**(UNIX)** `VNI_DIR/classes/jwave_demos/doc_examples/Simple.java`

**(Windows)** `VNI_DIR\classes\jwave_demos\doc_examples\Simple.java`

where `VNI_DIR` is the main Visual Numerics installation directory.

**Example 1-1** Client-side Java code, simple.java

```
// (1) Import JWAVE classes

import com.visualnumerics.jwave.JWaveExecute;

public class Simple {
   public static void main(String[] args) {
      JWaveExecute command = null;
      try {
  // (2) Auto-connect to a new PV-WAVE Session
  // Set a command object to use the Wrapper function named ”SIMPLE”
       command = new JWaveExecute(”SIMPLE”);
     // (3) Set a parameter named ”NUMBER” to the value 2
       // (NUMBER is expected by the SIMPLE wrapper function)
       command.setParam(”NUMBER”, 2);
      // Execute the command:
      //  Pass parameters, Run the Wrapper, get returned parameters
      command.execute();
      // (4) Get the returned data, named ”DATA”
      Double answer = (Double) command.getReturnData(”DATA”);
      // (5) Print the result
      System.out.println(”The answer is: ” + answer);
      } catch (Exception e) {
         // Report any problems
         System.out.println(e.toString());
      }    finally {
          // Shut down the PV-WAVE session (it would eventually
         //  time out and shut itself down if we did not do this)
         try {
         if (command != null) command.getConnection().shutdown();
         } catch (Exception ignore) { }
      }
   }
}
```

Here is a breakdown of the main parts of this program.

**1.** Any required Java and JWAVE class packages must be imported into the Java application. The JWAVE classes reside in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/JWave.jar` |
| **(Windows)** | `VNI_DIR\classes\JWave.jar` |

where `VNI_DIR` is the main Visual Numerics installation directory.

**2.** A connection must be established between the Java client and the JWAVE server. This is accomplished by instantiating the `JWaveExecute` object.

**3.** The name of the JWAVE wrapper function is specified in the `JWaveExecute` constructor. The `JWaveExecute` object contains the methods needed to "pack" the parameters and data to be sent to the server. In this case, a parameter named `NUMBER` with a value of `2` is set. The `execute` method sends the parameter and data to the server and "executes" the JWAVE wrapper function (called `SIMPLE`)

**4.** The `getReturnData` method is used to retrieve the result from the JWAVE server. The result, in this case, is named `DATA`.

**5.** The result received from the server is printed on the client.

## The JWAVE Wrapper Function

A JWAVE wrapper function is a PV-WAVE routine that contains JWAVE-specific function calls. These JWAVE calls typically include the function GETPARAM, which retrieves parameter information and data directly from a Java client application. The following function, `SIMPLE`, demonstrates the basic form of the JWAVE wrapper.

Example 1-2 can be found in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/jwave-3_5/lib/user/simple.pro` |
| **(Windows)** | `VNI_DIR\jwave-3_5\lib\user\simple.pro` |

where `VNI_DIR` is the main Visual Numerics installation directory.

As shown in Example 1-2, JWAVE wrapper functions take a single input parameter, usually called `client_data`. This parameter is automatically passed to the JWAVE wrapper from the Java application (when the `execute` method is called). The JWAVE wrapper receives parameter names and data through this parameter. The function GETPARAM, which is a JWAVE-specific PV-WAVE function, unpacks the parameters and data so that the wrapper can use them.

**Example 1-2** JWAVE wrapper function, simple.pro

```
FUNCTION SIMPLE, client_data
   ; Retrieve the parameter and data from the Java client.
  value = GETPARAM(client_data, 'NUMBER', /Value, Default=1)
   ; Process the data.
  mydata = SQRT(value)
; Return the result to the client (the default parameter
; name is DATA)
```

```
  RETURN, mydata
END
```

The `SIMPLE` function receives a parameter called `NUMBER` from the Java client. The *Value* keyword specifies that the actual value of the parameter be returned to the JWAVE wrapper.

The types of Java variables (and their corresponding PV-WAVE types) that can be passed between the client and server are listed in the following table.

| JAVA Data Types | Corresponding PV-WAVE Data Types |
|---|---|
| Byte | BYTE |
| Short | INTEGER |
| Integer | LONG |
| Float | FLOAT |
| Double | DOUBLE |
| String | STRING |

You can also pass arrays of these basic data types of up to eight dimensions. Note that the Java Short maps to PV-WAVE Integer, and that Java Integer maps to PV-WAVE Long. This is because of differences in the internal data representations of these integers.

**NOTE** You can use either numeric objects or primitives (for example, `java.lang.Short` or `short`).

The PV-WAVE RETURN statement sends the results back to the Java client. "get" methods in the Java client are then used to extract the returned parameters and data.

*Figure 1-8* further illustrates the flow of parameters and data between the JWAVE client and server.

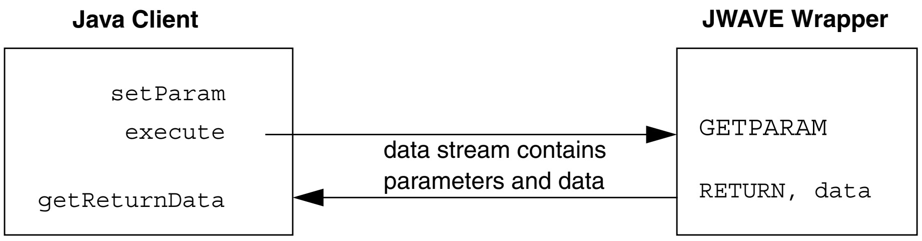


**Figure 1-8** Parameters and data are passed between the Java client and the JWAVE wrapper. Java methods and special PV-WAVE functions are used to package and extract the data and parameters.

## Running the Application

To run the `simple` application:

**Step 1** Start the JWAVE Manager. For instructions, see *Starting the JWAVE Manager* on page 109.

**Step 2** Move to the following directory:

**(UNIX)** `VNI_DIR/classes/jwave_demos/doc_examples`

**(Windows)** `VNI_DIR\classes\jwave_demos/doc_examples`

**Step 3** Run the program by typing the following command:

```
java Simple
```

**NOTE** To run this application, the following items must be included in your `CLASSPATH`:

- `VNI_DIR/classes/JWaveConnectInfo.jar`
- `VNI_DIR/classes/JWave.jar`
- `.` (the current directory)

where `VNI_DIR` is the main Visual Numerics installation directory.

## Sample Output

Here is the output from this simple JWAVE application:

```
The answer is: 1.4142135623730951
```

## *Summary*

The basic parts of a JWAVE system are the client-side Java application/applet and server-side JWAVE wrapper. The JWAVE Manager runs on the server, listens for client connections, and handles communication between the client and a PV-WAVE session.

If you do not want to write original Java programs, you can create useful JWAVE applications using the generic JWAVE applet. This applet, discussed in the next chapter, allows you to run PV-WAVE applications on the server and display results in a Web browser with little or no client-side programming. If you wish, you can use JavaScript and HTML forms to create a user interface for your applet.

2

CHAPTER

# The Generic JWAVE Applet

This chapter explains how you can get started almost immediately using JWAVE to publish PV-WAVE graphics on a Web page. With the generic JWAVE applet provided by Visual Numerics, you can execute a JWAVE wrapper function on the server and display output from the server in a Web browser.

The generic JWAVE applet lets you do this with very little programming. All you need to write is the JWAVE wrapper function (in PV-WAVE) and some simple HTML code. The generic applet takes care of the rest of the work, such as contacting the server, opening a connection to the server, and retrieving data and graphics from the server.

Later, you will see how you can use HTML forms and JavaScript to create a user interface for the generic applet. This interface can be used to modify the appearance of graphics returned from the server, to generate different types of plots, to send client data to the server, and many other possible functions.

## Simple Applet Example

Let's look now at a very simple implementation of the generic applet. This applet asks PV-WAVE to generate and return a 2D plot, which the applet displays. The resulting plot is shown in *Figure 2-1*.

This section discusses the HTML code used to display the applet and the JWAVE wrapper function that is executed by PV-WAVE on the server.

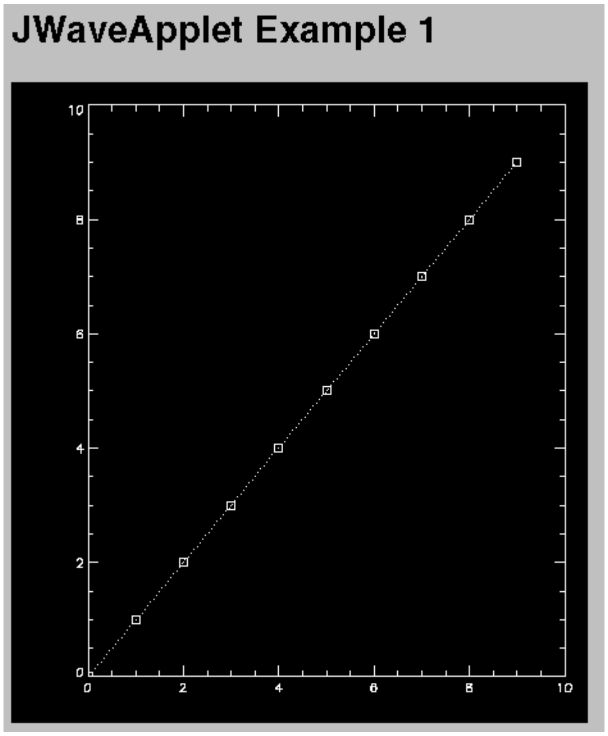


**Figure 2-1** 2D plot displayed using the generic JWAVE applet

## The HTML Code

Example 2-1 shows the simple HTML file in which this sample JWAVE applet is embedded.

**TIP** You can find examples similar to this one in the directory:

**(UNIX)** `VNI_DIR/classes/jwave_demos/JWaveApplet`

**(Windows)** `VNI_DIR\classes\jwave_demos\JWaveApplet`

where `VNI_DIR` is your main Visual Numerics installation directory.

**Example 2-1** Simple HTML code for calling the generic JWAVE applet.

```
<HTML>
<HEAD>
<TITLE>JWaveApplet Example 1</TITLE>
</HEAD>
<BODY>
<APPLET CODE="com.visualnumerics.jwave.JWaveApplet"
        CODEBASE="../../classes"
        ARCHIVE="JWave.jar, JWaveConnectInfo.jar"
        WIDTH=450
        HEIGHT=500>
  <PARAM NAME="FUNCTION"            VALUE="TESTPLOT">
  <PARAM NAME="TRANSIENT_SESSIONS"  VALUE="YES">
</APPLET>
</BODY>
</HTML>
```

The `CODE`, `CODEBASE`, `ARCHIVE`, `WIDTH`, and `HEIGHT` parameters are standard applet parameters used to call any applet. The `CODE` parameter gives the class name of the applet. This class was installed on the server when you installed JWAVE. The `CODEBASE` parameter tells the applet where to find the root of the Java class tree. The `ARCHIVE` parameter specifies Java Archive (JAR) files that contain the JWAVE Java class files that are required by the applet. The JAR file called `JWave.jar` is shipped with JWAVE and contains all of the class files you need to develop JWAVE client applications, including the JWaveApplet class file. The JAR file `JWaveConnectInfo.jar` describes how to connect to the server. The `WIDTH` and `HEIGHT` parameters simply specify the size of the applet display area.

**TIP** The generic JWAVE applet accepts a number of `PARAM` tags, many more than are used in this example. Refer to the Javadoc reference on the JWaveApplet class for detailed information on all of the generic applet's PARAM tags. For information on Javadoc, see *Using the JWAVE Javadoc Reference* on page 40.

The `FUNCTION` parameter takes one argument, `TESTPLOT`, which is the name of the JWAVE wrapper function on the server. This is the PV-WAVE function that is executed on the server. This function will be described later. It creates a plot and sends it back to the client.

By setting the TRANSIENT_SESSIONS parameter to YES, we are asking the JWAVE Manager on the server to shut down the PV-WAVE session as soon as the client-server transaction is completed. This request makes sense because all we want is to get back a single picture from PV-WAVE. No further processing is required. Therefore, it is best to shut down the PV-WAVE session. If TRANSIENT_SESSIONS were set to NO (the default), the PV-WAVE session would remain active on the server until explicitly terminated (when the applet is unloaded, which occurs when you move to a new HTML page in your browser).

## The JWAVE Wrapper Function

A JWAVE wrapper function is a PV-WAVE function that can communicate with a JWAVE client. This particular wrapper function doesn't do much. It simply creates a plot and returns it to the client. In this case, no parameters were passed from the client to the server.

**Example 2-2** A minimal JWAVE wrapper function

```
FUNCTION TESTPLOT, client_data
   PLOT, FINDGEN(10), Linestyle=0, PSym=6
   RETURN, 0
END
```

When RETURN is called, the wrapper automatically returns the plot to the client applet, where it is displayed.

In the next example, we'll add some JavaScript$^{TM}$ functions to the client HTML file. These functions will enable the client to control the appearance of the graphic generated by PV-WAVE on the server.

## Example Summary

In this example, we described a very simple scenario where a client applet executes a function on the JWAVE server, and the server sends a picture back to the client. In that case, no parameters or data were passed from the client to the server. The server simply generated some data, created a plot, and sent it back to the client.

The next section discusses how to pass parameters and data from client to server using JavaScript.

## *Using JavaScript to Control the Applet*

If you do not wish to program Java applications, you can use JavaScript$^{TM}$ to provide a wide range of client-side control for the generic JWAVE applet. JavaScript is a scripting language that allows you to create a user interface for HTML pages.

**TIP** There are many manuals available on JavaScript in bookstores. For online information about JavaScript, refer to the *JavaScript Guide* on the Netscape Web site at:

```
http://developer.netscape.com/docs/manuals
```

Example 2-3 demonstrates how you can add JavaScript controls to an HTML page for passing parameters and data between the generic applet and the server.

For instance, JavaScript can be used to create client-side Web pages with a graphical user interface. The GUI can be used with the generic JWAVE applet, for example, to change plot characteristics such as line color, axis range, plot symbols, and so on. When used with the generic JWAVE applet, JavaScript can be used to create complex client-side applications rapidly and efficiently.

**TIP** Visual Numerics has provided several applet demonstrations, including the one used in this example, with your JWAVE installation. For information on running the demonstration applets, see *Running the Applet Demonstrations* on page 26.

In this section, we discuss:

- The HTML file with JavaScript
- The JWAVE wrapper function
- Demonstration applets

### The HTML File with JavaScript

There are a few differences between this example HTML file and the one shown in Example 2-1. The biggest difference is that this HTML file contains JavaScript commands. JavaScript is an object-based scripting language that can be embedded into HTML files.

In this example, we use JavaScript to call methods that are defined in the JWAVE generic applet.

**TIP** For detailed information on the generic applet methods that can be used with JavaScript, refer to the Javadoc page on JWaveApplet (in the `jwave` package). For information on Javadoc, see *Using the JWAVE Javadoc Reference* on page 40.

**Example 2-3** HTML file with JavaScript calls

```
<HTML>
<HEAD>
<TITLE>JWaveApplet Example 2</TITLE>
</HEAD>

<SCRIPT LANGUAGE=JavaScript>
// Update the plot
  function updatePlot() {
    if (! document.JWavePlot.isStarted()) {
      // Wait for applet to start before trying to updatePlot
      setTimeout("updatePlot()", 250);
      return;
    }
    // Must get a session before we can set anything
    document.JWavePlot.openSession();
    // Set background and data line colors
    document.JWavePlot.setNamedColor('BACKGROUND', 'LightGray');
    document.JWavePlot.setNamedColor('LINE', 'Blue');
    // Set line style (dashed)
    document.JWavePlot.setParam('LINESTYLE', 2);
    // Turn off plot symbols
    document.JWavePlot.setParam('SYMBOL', 0);
    // Update the plot (and close the transient session)
    document.JWavePlot.execute();
  }

<BODY onLoad="updatePlot()">
<H1>JWaveApplet Example 2</H1>
```

```
<APPLET NAME="JWavePlot"
     CODE="com.visualnumerics.jwave.JWaveApplet"
     CODEBASE="../../"
     ARCHIVE="JWave.jar, JWaveConnectInfo.jar"
WIDTH=450
        HEIGHT=500>
  <PARAM NAME="FUNCTION"           VALUE="TESTPLOT">
  <PARAM NAME="EXECUTE_ON_START"   VALUE="NO">
  <PARAM NAME="TRANSIENT_SESSIONS" VALUE="YES">
</APPLET>


</BODY>
</HTML>
```

Let's look at the JavaScript function, `updatePlot`, that is embedded in the HTML file.

First, the "if" clause with the `setTimout` function ensures that the plot does not update until the applet is ready.

The next call in this function is:

```
document.JWavePlot.openSession();
```

This call follows the JavaScript object convention, where `document` is the object name referring to the browser window itself. `JWavePlot` is the name by which JavaScript recognizes the applet (specified with the applet's `NAME` tag), and `openSession` is a method defined in the generic JWAVE applet, `JWaveApplet` (specified with the `CODE` tag).

We need to call `openSession` to open a connection with the JWAVE Manager on the server. Normally, the applet connects and executes immediately upon startup. But we want to delay the execution until the parameters are set. That is, we want to open a session, set the parameters, and then execute the JWAVE wrapper.

The next few JavaScript calls set color values and parameters to be sent to the JWAVE wrapper function on the server.

```
document.JWavePlot.setNamedColor('BACKGROUND', 'LightGray');
document.JWavePlot.setNamedColor('LINE', 'Blue');
document.JWavePlot.setParam('LINESTYLE', 2);
document.JWavePlot.setParam('SYMBOL', 0);
```

The function `setNamedColor` sets a parameter name and a color value. In the JWAVE wrapper function on the server, these parameters are interpreted by corresponding GETPARAM functions, and their values are retrieved. Once retrieved, those parameter values can be plugged directly into PV-WAVE functions. In this case, `TESTPLOT` is going to plug the parameters set in the HTML file into the PV-WAVE PLOT command.

The final call in our JavaScript function is:

```
document.JWavePlot.execute();
```

The `execute` method sends parameters to the server and executes the JWAVE wrapper function. This in turn generates a plot, which is sent back to the client and displayed.

Finally, we use a couple of `PARAM` tags to prevent the applet from executing the JWAVE wrapper immediately when the applet starts. First, we need to set the `EXECUTE_ON_START` parameter to `NO`. This parameter prevents the JWaveApplet from executing when the applet starts. Then, having told the applet what *not* to do, we need to tell the applet what to do when it is loaded. This is the purpose of the `onLoad` parameter that is set in the applet's `BODY` tag. The `onLoad` parameter tells the applet to execute the JavaScript function `updatePlot` when the HTML page loads. And, to reiterate, this JavaScript function does the following:

- Opens a connection to the JWAVE server.
- Sets four different parameters that are used to send values to PV-WAVE.
- Executes the JWAVE wrapper on the server.

---

**TIP** Because JavaScript supports form objects, you can use JavaScript to create interactive GUIs for your client applets without any Java programming. For information about more complex JavaScript demonstrations created by Visual Numerics, see *Running the Applet Demonstrations* on page 26.

---

## The JWAVE Wrapper

To take the actions requested by the client, the JWAVE wrapper must retrieve and "unpack" the parameters and data sent from the client.

The functions used to unpack data sent from the client are: GET_NAMED_COLOR and GETPARAM. PLOT commands are then constructed using the unpacked values.

You can find the following JWAVE wrapper in:

**(UNIX)** `VNI_DIR/jwave-3_5/lib/user/testplot.pro`

**(Windows)** `VNI_DIR\jwave-3_5\lib\user\testplot.pro`

where `VNI_DIR` is the main Visual Numerics installation directory.

**Example 2-4** JWAVE wrapper function, TESTPLOT

```
FUNCTION TESTPLOT, client_data

   ; get colors

   black = '000000'xL

   white = 'FFFFFF'xL

   back_color = GET_NAMED_COLOR("BACKGROUND", Default = black)

   axis_color = GET_NAMED_COLOR("AXES", Default = white)

   line_color = GET_NAMED_COLOR("LINE", Default = white)

   psym_colors = GET_NAMED_COLOR("SYMBOLS", Default = [white],
   /Color_Set)

   ; get data

   data = GETPARAM(client_data, 'DATA', /Value, Default =
   FINDGEN(10))

   ; get plot attributes

   linestyle = GETPARAM(client_data, 'LINESTYLE', /Value, Default =
   1)

   psym = GETPARAM(client_data, 'SYMBOL', /Value, Default = 6)

   ; Plot axes

   PLOT, data, /NoData, Background = back_color, Color = axis_color

   ; plot lines

   IF linestyle GE 0 THEN $

     OPLOT, data, Linestyle = linestyle, Color = line_color

   ; plot symbols

   IF psym NE 0 THEN $

     OPLOT, data, PSym = ABS(psym), Color = psym_colors

RETURN, 0

END
```

### *The GETPARAM Function*

GETPARAM (a JWAVE-specific PV-WAVE function) is designed to retrieve non-color related parameters and values. In Example 2-4, the first GETPARAM call retrieves data values to plot. The next two GETPARAM calls are used to set the linestyle and symbol style for the plot.

The *Value* keyword is used to specify that an actual value be returned by GETPARAM, and the *Default* keyword specifies a default value to use in case no value is received from the client. Therefore, the GETPARAM call:

```
data = GETPARAM(client_data, 'DATA', /Value, Default = FINDGEN(10))
```

returns the actual data values to be plotted, which can be used in a PLOT command, such as:

```
PLOT, data, ...
```

---

**NOTE** Detailed information on the parameters and keywords of the GETPARAM and GET_NAMED_COLOR functions are available in the PV-WAVE online help system and in Appendix A, *JWAVE Wrapper API*.

---

### *The GET_NAMED_COLOR Function*

If a color is specified by the client with the generic applet's `setNamedColor` method, that color can be retrieved by PV-WAVE with the GET_NAMED_COLOR function.

The GET_NAMED_COLOR function converts a named color specified on the client into a color index that PV-WAVE can understand. For instance, the returned value from GET_NAMED_COLOR can be used with the PLOT command's *Background* keyword.

For example, GET_NAMED_COLOR might return a color index for the background color of a 2D plot.

```
back_color = GET_NAMED_COLOR("BACKGROUND", Default = black)
data = GETPARAM(client_data, 'DATA', /Value, Default = FINDGEN(10))
PLOT, data, Background = back_color
```

## Running the Applet Demonstrations

Visual Numerics has provided a set of demonstration HTML files that use JavaScript controls.

**NOTE** To run these applets, you must have a working installation of JWAVE and the JWAVE Manager must be running on the server. To run these demonstrations, you must also have a browser that supports JDK 1.1.

You can find these demonstration files in:

| | |
|---|---|
| **(UNIX)** | `$VNI_DIR/classes/jwave_demos/JWaveApplet` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos/JWaveApplet` |

You are free to copy any of the form and JavaScript functions in these demonstration applets for your own use. The examples include:

- **`applet_demo1.html`** — This HTML page contains a single JWaveApplet that calls JWAVE to make a plot. All plot parameters (and data) are set/generated by the JWAVE wrapper function. This applet is similar to Example 2-1.
- **`applet_demo2.html`** — This HTML page contains a single JWaveApplet that calls JWAVE to make a plot. There is some simple JavaScript in this page that initializes some parameters (colors and linestyle) used by the JWAVE wrapper function. The wrapper function generates its own data. This applet is identical to Example 2-3.
- **`applet_demo3.html`** — This HTML page contains a single JWaveApplet that calls JWAVE to make a plot. There is some JavaScript in this page that sets parameters used by the JWAVE wrapper function. The parameters set by JavaScript are controlled by HTML `FORM` tags. There are some JavaScript helper functions provided in this page to help the `FORM` tags interact with JWAVE. The wrapper function still generates its own data.
- **`applet_demo4.html`** — There are two JWAVE applets running in this HTML page. They share a connection to a single PV-WAVE session, and thus can share data on the server side. One applet is invisible, and uses the JWaveExecute class to create some data on the server. The resulting data stays on the server. It is stored by the Data Manager. The second applet uses the JWaveView class to make a plot of the data created by the first applet.
- **`applet_demo5.html`** — There are two JWAVE applets running in this HTML page. They share a connection to a single PV-WAVE session, and thus can share data on the server side. One applet is invisible, and uses the JWaveExecute class to create some data. The resulting data stays on the server (stored by the Data Manager). The second applet uses the JWaveView class to make a plot of the data created by the first applet, using one of two wrapper functions that each create a particular type of plot.

## Summary

In this chapter we explained how the generic JWAVE applet, JWaveApplet, can be used to create client applications that use PV-WAVE as a numerics and graphics server. The generic applet allows you to use JavaScript functions to control the applet and pass data and parameters between the client and server. Visual Numerics has provided several sample applets that you can run and examine on your own. Feel free to reuse these applets and their helper functions for your own applications.

3

CHAPTER

# JWAVE Client Development

This chapter discusses the following topics:

- The JWaveExecute class
- Passing parameters from the client to the server
- Getting data back from the server
- Casting returned data
- The exception classes
- Managing the server connection
- Data proxies
- Using the JWAVE Javadoc reference

## JWAVE Client Overview

Parameters are used to pass information and data back and forth between client and server.

On the client side, you set parameters to be sent to the server using the `JWaveExecute.setParam` method. These parameters are not actually sent to the server until the `execute` method is called. After an `execute`, the client may retrieve parameters returned by the wrapper function using `JWaveExecute.ReturnData` (to get one parameter at a time) or `getReturnParams` (to get all the parameters at once).

On the server side, parameters passed from the client can be retrieved in a JWAVE wrapper with the GETPARAM function (a JWAVE-specific PV-WAVE function). When a wrapper function returns, its return values are automatically sent to the client.

*Figure 3-1* shows the most basic parameter passing scenario.

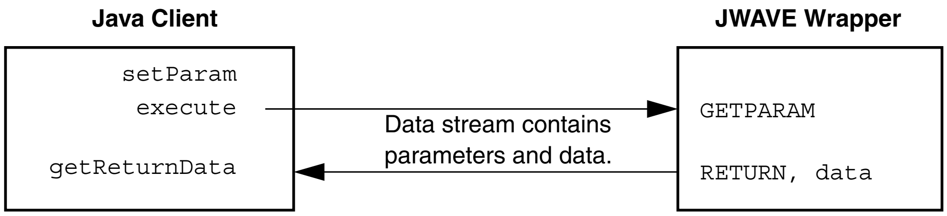


**Figure 3-1** Parameters and data are passed between the Java client and the JWAVE wrapper. Java methods and special PV-WAVE functions are used to package and extract the data and parameters.

This chapter discusses the basic methods for passing parameters between JWAVE client and server applications.

**NOTE** This chapter does not cover the topic of *data proxies*, which allow data stored on the server to be referenced by the client. Proxies permit the efficient management of data and of client resources. For detailed information on proxies, see Chapter 6, *Managing Data*.

## The JWaveExecute Class

The JWaveExecute class provides client access to a JWAVE wrapper function on the server. JWaveExecute methods are used to set the name of the JWAVE wrapper, to set the parameters sent to that wrapper, to set the return parameter mode, and to retrieve data returned from the JWAVE wrapper. The `execute` method sends the parameters that were set (with `setParam` methods) to the wrapper, causes the wrapper to execute, and retrieves any returned data.

You can construct a `JWaveExecute` object simply by giving the JWAVE wrapper name:

```
JWaveExecute(String wrappername)
```

This command automatically connects the `JWaveExecute` object to a new PV-WAVE session. That is, a new `JWaveConnection` object is created automatically.

**TIP** After construction, you can call a different JWAVE wrapper function, if desired, using the method `setFunction`.

If you already have a `JWaveConnection` object that refers to an existing PV-WAVE session, then you can construct a `JWaveExecute` object to use that same connection. For example:

```
JWaveExecute(JWaveConnection connection, String name)
```

Usually, you use this constructor when there are several `JWaveExecute` objects that need to use the same PV-WAVE session (in other words, they need to share data), or if you want to set particular attributes of the connection (such as data compression or a session pinger).

The following method creates a `JWaveConnection` object:

```
static JWaveConnection.getConnection()
```

or

```
JWaveExecute.getConnection()
```

For detailed information on JWaveExecute and JWaveConnection classes, refer to the online Javadoc reference. For information on using Javadoc, see *Using the JWAVE Javadoc Reference* on page 40. See also *Managing the Server Connection* on page 36 for information on the JWaveConnection class.

## Passing Parameters from the Client

The `JWaveExecute.setParam` method is used to pack parameters and data on the client to be sent to the server. The basic information that is packed by `setParam` includes a parameter name and some kind of reference to data.

Overloaded versions of `setParam` are provided for your convenience in specifying this parameter information.

For example, some of the `setParam` configurations are:

```
setParam(String name, double value)
```

Associates a parameter name with a scalar numeric primitive value. This method accepts any primitive numeric scalar value.

**NOTE** Parameter names are not case sensitive; they must begin with a letter, and can contain only letters, digits, and underscores.

`setParam(String name, Object value)`

Associates a parameter name with a scalar or array. The valid data types you can use with this `setParam` method include:

**Scalar Values:**

- numeric Objects (standard subclasses of Number, such as `Integer`), except `Long`
- `String`

**Arrays of up to eight dimensions of:**

- numeric Objects (such as `Integer`)
- numeric primitives (such as `int`), except `long`
- `String`

**A Proxy object referring to one of the above types.** (See Chapter 6, *Managing Data* for more information on proxies.)

`setParam(Parameter param)`

Sets a previously defined `Parameter` object (which encapsulates the parameter name and associated data).

There are several other methods for `setParam`. Some of these are discussed in Chapter 6, *Managing Data*. Others are not commonly used, and are not discussed in this manual. Refer to the Javadoc reference on the JWaveExecute class for detailed information on all of the `setParam` methods. For information on Javadoc, see *Using the JWAVE Javadoc Reference* on page 40.

As you can see, the different permutations of `setParam` allow you to specify precisely how you want parameters to be sent to the client. No matter how this parameter information is specified, `setParam` "packs" the parameter information in a uniform manner so that the server can retrieve and unpack the parameters.

On the server side, the function GETPARAM is used to unpack the parameters so that they can be used by PV-WAVE.

## Getting Data Back from the Server

The most common way to get data back from the JWAVE wrapper is using `JWaveExecute.getReturnData` method. This method accepts one parameter: the name of the returned data. This data name is specified by the JWAVE wrapper, but if a name is not explicitly specified, the default name `DATA` is used. Of course, a JWAVE wrapper can return many data objects at once. You have to know the names of those objects (given in the JWAVE wrapper) to unpack them.

The method `getReturnProxy` is similar to `getReturnData`, except that it returns a `Proxy` object that refers to the data. This is useful when the data is stored on the server. For more information on using proxies, see Chapter 6, *Managing Data*.

Another method, `getReturnParams`, returns all of the returned data, packaged in an array of `Parameter` objects.

**TIP** For debugging purposes, the following command can be useful:

```
Parameter.printInfo( myJWaveExecute.getReturnParams() );
```

This command prints (to `System.out`) the names and data types of all parameters returned by the JWAVE wrapper.

See Chapter 4, *JWAVE Graphics* for information on handling graphics returned from the server.

## Casting Returned Data

Any data returned by `getReturnData` is of type Object. This object may be a scalar or a multi-dimensional array, depending on what was returned by the JWAVE wrapper.

Numerical scalars are returned as one of the `java.lang.Number` subclasses (`java.lang.Integer`, `java.lang.Float`, and so on.). Numerical arrays are returned as an array of one of the primitive numeric types (such as `int[]` or `float[][]`). Strings are returned as `java.lang.String`, `java.lang.String[]`, and so on.

In order to make the data returned by `getReturnData` useful, you need to cast it to something. For example, if you are sure of the data type and array dimensions returned by the wrapper, you can write a statement such as:

```
 int[][] result = (int[][])
   myJWaveExecute.getReturnData("DATA_NAME");
```

This is the easiest technique, and thus it is valuable to make sure that the data types returned by the wrapper are explicitly specified.

If you are unsure of the data type (PV-WAVE is a loosely-typed language), then you must test the returned Object for its type. Usually, you will know something about the data, such as its number of array dimensions, whether it is a string or a number, and so on. So usually you will only need to test for (or assume) things like array size and then just cast or assign the result to whatever variable you will use in the Java program.

Of course, you can use the `instanceof` operator to test for particular types.

The classes `java.lang.Class`, `java.lang.reflect.Array`, and `com.visualnumerics.util.ArrayUtils` are other useful tools for dealing with the object returned by `getReturnData`. Some examples of using these classes are shown in the following example:

**Example 3-1** Array handling

```
Object result = myJWaveExecute.getReturnData("DATA_NAME");


// Test if the returned object is an array

if ( result.getClass().isArray() ) {

   System.out.println("Is Array");


   // Get the size of the array (i.e. [3][4][5])

   int[] dims = ArrayUtils.getArrayDimensions(result);


   System.out.println("Num Dims = "+ dims.length);

   System.out.print("Dims = ");

   for (int i=0; i<dims.length; ++i)

      System.out.println("[" + dims[i] + "]");

   System.out.println();


   // Get data type of array's contents

   // Note that result.getClass() just tells you that it is an array

   // And Class.getComponentType() is only useful for 1D arrays

   // This gives you the Class of the contents of the array,
   // no matter the dimensional size of the array
   Class c = ArrayUtils.getBaseComponentType(result);

   System.out.println("Type = " + c.getName());

   // store into double[]
   // Ensure 1D numeric array
   if (dims.length == 1 && !String.class.isAssignableFrom(c)) {
      double[] dblResult = new double[dims[0]];
      // Store into double array
      for (int i=0; i<dblResult.length; ++i)
         dblResult[i] = Array.getDouble(result, i);

      System.out.println("Stored into double[].");
   } else {
      // Do different things for multi-dim arrays, strings...
      // See ArrayUtils.getAsOneDimArray(), for example
   }

} else { // not array
   System.out.println("Is Scalar");
   System.out.println("Type = " + result.getClass().getName());

   // Store into int scalar
   if (result instanceof Number) {
      double dblResult = ((Number)result).doubleValue();
      System.out.println("Stored into double = "+ dblResult);
   } else {
      // Do different things for strings...
   }
}
```

## The Exception Classes

If there is a problem with JWAVE, a JWaveException class error is thrown. The most commonly encountered subclasses of JWaveException are:

`JWaveConnectionException` — Indicates a problem with the connection to the JWAVE server. The server may be down or unreachable. There may be a problem with the description of the connection method (in the `JWaveConnectInfo.jar` file).

`JWaveServerException` — Indicates a problem with the JWAVE server. You were able to connect to the server, but it produced an error. It may not have been able to start PV-WAVE. You may have run out of JWAVE licences. The server may not have been able to find or run your intended wrapper function.

`JWaveWrapperException` — The JWAVE wrapper function was executed, but exited with an error. The exception text comes from the PV-WAVE MESSAGE procedure (or !Err_String system variable).

## Managing the Server Connection

There are several useful methods in the JWaveConnection class. You can get a connection object explicitly using the `JWaveConnection.getConnection` method, or implicitly by constructing a JWaveExecute object (without using a `JWaveConnection` object in the constructor), and using that `JWaveExecute.getConnecton` method.

### Compressing Data

Once you have a connection, you can control some of the aspects of that connection. First, if you wish to use compression in your communications with the JWAVE server, use the `setCompression` method. Compression can be turned on and off, allowing you to make some `execute` calls on a compressed connection and others without compression.

Generally, compression is beneficial for transferring large data sets (especially graphics, which are usually very compressible) over relatively slow networks. If you have a fast network connection to the server, you may not want to use compression—even for large data sets—because the CPU time of encoding and decoding the data could be inefficient. But if your network connection is slow, you may want to experiment with compression to see if it helps your performance.

## Ending a JWAVE Session

When you are done with a PV-WAVE session, you should call the `shutdown` method of JWaveConnection. This closes the PV-WAVE session, releasing any resources (memory, JWAVE licenses, and so on) that it was using on the server.

For example:

```
myJWaveExecute.getConnection().shutdown();
```

## Using Multiple Clients

If you wish to have several client applications use the same PV-WAVE session (for instance, to share data), then you need to assign that PV-WAVE session a Session ID number (a positive integer). JWaveConnection normally acquires a unique Session ID from the server; however, if you want to use a particular PV-WAVE session, then call `setSessionID` before you contact the server (using methods such as `pingSession` and `execute`). If that PV-WAVE session is running, then you will be connected with it. If it is not running, then it will be started.

## Using Ping Methods

Another set of useful methods of JWaveConnection are `pingManager` and `pingSession`. The `pingManager` method allows you to make sure that the JWAVE Manager is alive, and `pingSession` ensures that your PV-WAVE session has not timed out.

The `pingSession` method can also be used to start a PV-WAVE session. This can help performance; for example, you may call `pingSession` at the beginning of your application so that the PV-WAVE session will be activated by the time the user presses the Plot button of your GUI.

In order to keep the PV-WAVE session alive (prevent the JWAVE Manager from killing the session if it is idle too long), use the `startSessionPinger` method. This method starts a thread that will call `pingSession` every minute (by default) until you call `stopSessionPinger` or `shutdown`. This keeps your PV-WAVE session from becoming idle.

---

**TIP** If you use `startSessionPinger` in all of your JWAVE applications, then your JWAVE Server administrator can reduce the `SESSION_IDLE_TIMEOUT` setting, so that idle processes can be cleaned up more often.

---

## Example: Passing an Array

In this example, the client creates an array of values and passes the array to the server. The server processes the array and sends back an object representing an array to the client where it is printed. The returned object must be explicitly cast to the desired data type (in this example, the object is cast to a floating-point array).

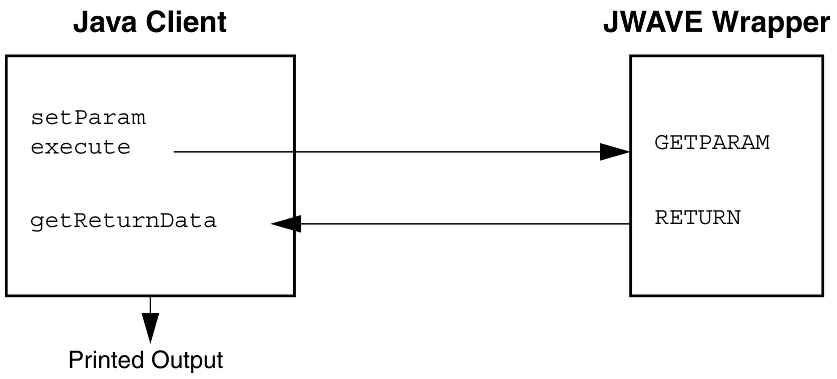


**Figure 3-2** Data is passed from client to server, and the server returns a data object, which is then printed by the client.

**Example 3-2** Client application passes an array to the server, retrieves a result, and prints it

```
import com.visualnumerics.jwave.JWaveExecute;
import com.visualnumerics.jwave.JWaveConnection;
public class PassArray {
   public static void main(String[] args) {
   JWaveExecute command = null;
   // Create a simple array of data values.
   float arr[] = new float[10];
   for (int k=0; k<arr.length; k++) {
      arr[k]=k;
   }
   try {
       // Pass the array parameter to the server to use with
       // the PASSARR JWAVE wrapper function.
       command = new JWaveExecute("PASSARR");
       command.setParam("ARRAY", arr);
       command.execute();
```

```
        // Get the data Object returned from the server and cast
        // to float array.
        float d[]= (float[]) command.getReturnData(”DATA”);

        //Print the returned array.
        for (int j=0; j<d.length; j++) {
           System.out.println(d[j]);
        }

    } catch (Exception e) {
       System.out.println(e.toString());
    } finally {
         if (command != null) {
            JWaveConnection c = command.getConnection();
            if (c != null) c.shutdown();
         }
    }
}
```

**Example 3-3** JWAVE wrapper function receives the array, changes it, and returns it to the client

```
FUNCTION PASSARR, client_data
   ; Unpack parameters and data
   arr =    GETPARAM(client_data, ’ARRAY’, /Value,
   Default=[1.,2.,3.])
   ; change the array
   mydata = arr * 1.5
   ; return the changed array and ensure FLOAT data type
   RETURN, FLOAT(mydata)
END
```

When the Java program in *Example 3-2* is executed, the following output is printed on the client:

```
% java PassArray
0.0
1.5
3.0
4.5
6.0
7.5
9.0
```

```
10.5
12.0
13.5
%
```

## A Note About Data Proxies

The data in the previous example makes a complete round trip from the client to the server, and back to the client. For very large datasets, this round trip can be expensive, both in bandwidth and client memory resources. Sometimes, this round trip is unnecessary, because the data is not used by the client; it is just sent back to the server for further processing. Fortunately, JWAVE allows you to leave all of your data on the server and reference it on the client using a data *proxy*. For information on managing data efficiently with data proxies, see Chapter 6, *Managing Data*.

## Using the JWAVE Javadoc Reference

Reference information on the Java class files used for JWAVE client development is available online in Javadoc format. To view the Javadoc reference for JWAVE, start a Web browser and open the following file:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/docs/api/packages.html` |
| **(Windows)** | `VNI_DIR\classes\docs\api\packages.html` |

where `VNI_DIR` is the main Visual Numerics installation directory.

## Summary

The JWaveExecute class provides client access to a JWAVE wrapper function on the server. Parameters and data that are set with JWaveExecute methods are sent to a JWAVE wrapper function when the `execute` method is called. The methods `getReturnedData`, `getReturnParams`, and `getReturnProxy` are all used to retrieve data returned from the server. Sometimes it is necessary to cast the returned data to the desired data type. The JWaveConnection class allows you to create and manipulate the JWAVE client-server connection. Client-server connection and JWAVE wrapper errors are handled by a set of JWaveException subclasses.

4

CHAPTER

# JWAVE Graphics

This chapter explains how client applications can display graphics returned from PV-WAVE. The following JWAVE classes are used in graphical client applications:

- JWaveCanvas and JWavePanel
- Viewable
- JWaveView

These classes are described in this chapter and in the Javadoc reference for JWAVE. For information on Javadoc, see *Using the JWAVE Javadoc Reference* on page 40.

## Returning Graphics to the Client

In Example 1-1 on page 12 we demonstrated a simple JWAVE application that returned a numerical result to the client. To accomplish this, we used a JWaveExecute object. The JWaveExecute class provides methods for setting and getting parameters (and data), and executing JWAVE wrapper functions on the server; however, a JWaveExecute object can only retrieve numerical and string data returned from the client. If you want your client to retrieve graphical information from the server, then you must use a JWaveView object and a JWaveCanvas or a JWavePanel.

### The JWaveCanvas and JWavePanel Class

The JWaveCanvas is a subclass of java.awt.canvas component class. JWavePanel is the swing version of JWaveCanvas, it is a subclass of javax.swing.JPanel. Both classes provide a canvas for displaying a viewable object. When a plot is returned from the wrapper function, the JWaveCanvas or JWavePanel class paints the chart to the screen.

Since these classes are designed to display graphics or a JWAVE Viewable object, it is required to inform the canvas class of the graphic. This is done using the setViewable method. The setViewable method is used to display the Viewable object retrieved from the JwaveView instance and updates the canvas with the new graphics. There are subclasses of the JWaveCanvas and JWavePanel class that can be used to interact with the graphics. These classes are described in Chapter 9, *Advanced Graphics Features*.

## Viewable Object

The Viewable object contains any graphics, such as plots and images, that are returned from PV-WAVE. This object knows how to paint itself onto a component. This object also retains information about fundamental graphics characteristics, such as size, coordinates, and resizing. Use the `setPreferredResizeMode` to indicate whether or not the graphics are to be resized by the draw method. Several methods exist for transforming to and from PV-WAVE data coordinates and 2D pixel coordinates. A getDataBounds method is useful for retrieving the region depicting the data drawn by the wrapper function. These methods are described in the section Resizing Graphics *on page 48* and the section Coordinate System Transformations *on page 49* of this chapter.

## The JWaveView Class

The JWaveView class is a subclass of the JWaveExecute class. Unlike JWaveExecute, JWaveView gets a Viewable object from the server. The Viewable object is returned by the JWAVE wrapper function.

*Figure 4-1* shows a typical JWAVE scenario where graphical data is created on the server and returned to the client.

**TIP** In this example, data is physically returned from the server to the client. A more efficient method of handling the data is with a data proxy object. For more information on data proxies, see Chapter 6, *Managing Data*.

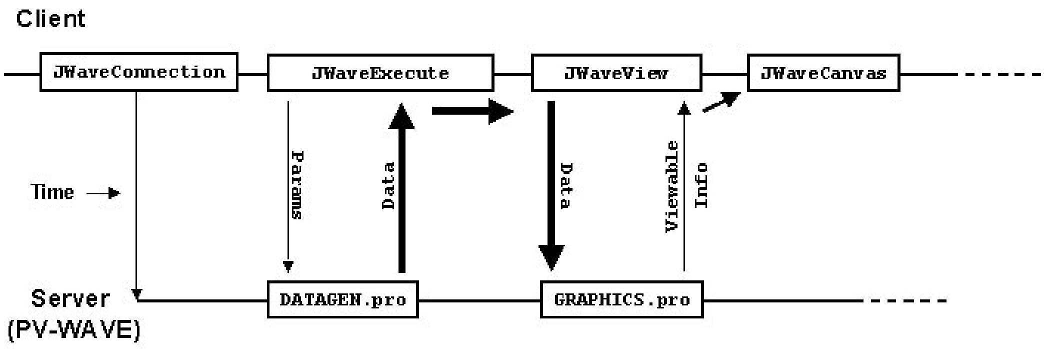


**Figure 4-1** A JWaveView object is used to obtain viewable image data from the PV-WAVE server

## Sample Code

To display a JWAVE chart, create a class that extends JWaveCanvas. This class creates a Viewable object returned from a JWAVE wrapper function that generates graphics (for example, if the wrapper calls the PLOT procedure). The setViewable method is used to register the Viewable object with the parent JWaveCanvas class. The JWaveCanvas class actually paints the chart to the screen.

**Example 4-1** JWaveCanvas used to display a chart

```
class MyChart extends JWaveCanvas {
    MyChart()  throws JWaveException {
        JWaveView view = new JWaveView("myWrapper");
        // Set the size of the Viewable object.
        view.setViewSize(getSize());
        // Execute the JWAVE wrapper function.
        view.execute();
        setViewable(view.getViewable());
    }
}
```

## Example: Displaying a Simple 2D Plot

Example 4-2 lists a Java client program that displays a 2D plot generated by PV‑WAVE. Example 4-3 lists the JWAVE wrapper that creates the 2D plot.

---

**TIP** Example 4-2 (the client Java program) is similar to (but a simplified version of) the demonstration program `ViewTest.java`, which you can find in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/tests` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\tests` |

The actual code for Example 4-2, `SimpleView.java`, can be found in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/doc_examples` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\doc_examples` |

Example 4-3, SIMPLE_VIEW (the JWAVE wrapper), can be found in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/jwave-3_5/lib/user/simple_view.pro` |
| **(Windows)** | `VNI_DIR\jwave-3_5\lib\user\simple_view.pro` |

where `VNI_DIR` is the main Visual Numerics installation directory.

---

The first component class is MainFrame. It is a frame that holds the chart. It registers a WindowListener on itself so that it can cause the application to exit when the user selects the exit icon on the frame.

The other component class is Chart. It is a JWaveCanvas that communicates with the Jwave server to run the JWAVE wrapper `SIMPLE_VIEW`. The class Chart sends the parameters BACKGROUND, COLOR, GRIDSTYLE, TICKLEN, and Y to the server. In response, the viewable is updated and the chart is drawn by the parent class JWaveCanvas.

Figure 4-2 on page 4-47 shows the plot that is displayed on the Client.

**Example 4-2** SimpleView.java displays a 2D plot generated by PV-WAVE

```
import com.visualnumerics.jwave.*;
   import java.awt.*;
   import java.awt.event.*;

   public class SimpleView extends Frame {

       /** Construct the frame that holds a chart */
       public SimpleView() throws JWaveException {
           MainFrame mainFrame = new MainFrame();

           // Add a chart to the frame
           final Chart chart = new Chart();
           mainFrame.add(chart);

           // Make the frame visible and draw the chart
           mainFrame.setVisible(true);
           chart.update();
       }


      public static void main(String[] arg) throws JWaveException {
           SimpleView me = new SimpleView();
       }
   }


   class MainFrame extends Frame {
       MainFrame() {
           setTitle("Simple JWaveView Example");
           setSize(300,300);

           // so Exit button (window menu) will work
           addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent event) {
                   System.exit(0);
               }
           });
       }
   }


   class Chart extends JWaveCanvas {
       public void update()  throws JWaveException {
           // Use SIMPLE_VIEW wrapper function
           JWaveView command = new JWaveView("SIMPLE_VIEW");

           // Set view size same as canvas
```

```
            command.setViewSize(getSize());

            // Set some colors and parameters
            command.setNamedColor("BACKGROUND", Color.white);
            command.setNamedColor("COLOR", Color.black);

           command.setParam("GRIDSTYLE", 1); // dotted lines for grid
            command.setParam("TICKLEN", 0.5); // full grid
            command.setParam("Y", 500); // Number of data points
            // Execute the command
            command.execute();

            // Retrieve the Viewable and give it to the JWaveCanvas
            // JWaveCanvas repaints when it gets a new Viewable
            setViewable( command.getViewable() );

            // Close connection, as we are done with the session
            command.getConnection().shutdown();
        }
    }
```

**Example 4-3** JWAVE wrapper simple_view.pro

```
FUNCTION SIMPLE_VIEW, client_data
       ; Retrieve parameter data
       n = GETPARAM(client_data, 'Y', /Value, Default = 100)
      grid = GETPARAM(client_data, 'GRIDSTYLE', /Value, Default = 0)
      tick = GETPARAM(client_data, 'TICKLEN', /Value, Default = 0.2)
       back = GET_NAMED_COLOR('BACKGROUND', Default = '000000'xL)
       fore = GET_NAMED_COLOR('COLOR', Default = 'ffffff'xL)
       ; Generate some "data" - n points
       vals = RANDOMN(seed, n)
       ; Make the plot
       PLOT, vals, Ticklen = tick, GridStyle = grid, Background =
   back, Color = fore
       ; Return the data (default data name DATA).
       ; Note that this data is returned in addition to the Viewable
   object.
       RETURN, vals
   END
```

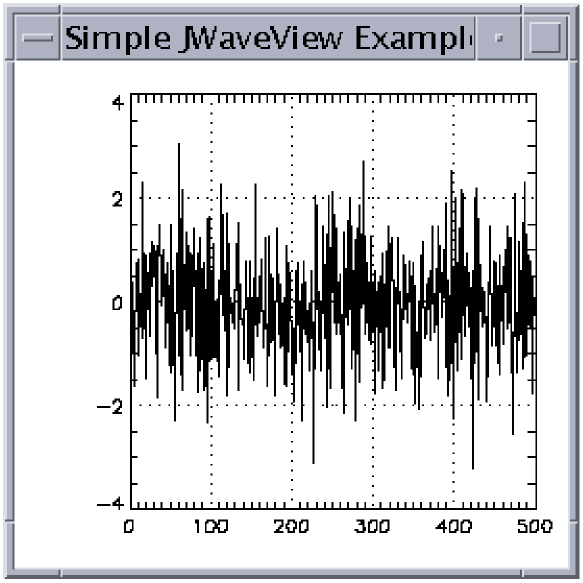


**Figure 4-2** The SimpleView plot is displayed on the client.

## Resizing Graphics

The Viewable class supports resizing of the `Viewable` object in the client without a roundtrip to the server. The default resize behavior is `NOT_RESIZEABLE`. This means that the `Viewable` object will always be drawn at its 'natural' size (as generated by PV-WAVE), regardless of the size of the Component it is drawn into.

You may want to use `Viewable.setPreferredResizeMode` with either `RESIZEABLE` or `PRESERVE_ASPECT` if your Component is resizable. These modes allow the Viewable object to draw a scaled version of the plot into any size Component. The `PRESERVE_ASPECT` will resize the Viewable object, but only as much as is possible while maintaining the original (as generated by PV-WAVE) aspect ratio of the `Viewable` object.

Resizable is a client-side resizing of a bitmap. It is also possible to resize by making a new request to the server. In *Example 4-2* on page 45, the SimpleView constructor can be modified to listen for resize events generated by the frame and request a new chart at the new size.

Following is the modified constructor. Note that this relies on the fact that Chart.update() resets the view size to the current size of the canvas.

```
public SimpleView() throws JWaveException {
        MainFrame mainFrame = new MainFrame();

        // Add a chart to the frame
        final Chart chart = new Chart();
        mainFrame.add(chart);

        // Make the frame visible and draw the chart
        mainFrame.setVisible(true);

        mainFrame.addComponentListener(new ComponentAdapter() {
            public void componentResized(ComponentEvent event) {
                try {
                    chart.update();
                } catch (JWaveException e) {
                    System.out.println(e);
                }
            }
        });
        chart.update();
    }
```

## Coordinate System Transformations

The Viewable class also supports coordinate system transforms. PV-WAVE automatically communicates the data coordinates and 3D transform used to generate the plot, and a `Viewable` object can use these to do coordinate transforms.

The two methods `transformToPoint` and `transformToData` can be used to transform between the plot's data coordinates and a `java.awt.Point` object.

For example, to print the data coordinates where a user clicks on the plot, you might do the following:

```
double[] d = myViewable.transformToData( theClickPoint );
System.out.println( "Click at ("+ d[0] +", "+ d[1] +")" );
```

You can determine the boundary of the data area (usually, this boundary is the plot's axes) using the `getDataBounds` method. For example, to determine if a `Point` object is inside the plot bounds:

```
if ( myViewable.getDataBounds().contains( myPoint ) ) { ... }
```

Note that (due to lossy conversion) the `transformToData` method is only useful for plots with 2D data coordinates (that is, X vs. Y plots). To test for this condition, use the `has2DCoordinates` method.

**TIP** For an example of using the coordinate transforms of the Viewable class, see ViewTest.java which you can find in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/tests` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\tests` |

## Demonstration Programs

You can find other demonstration JWAVE graphics applications in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/jwave_demos` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos` |

## Summary

This chapter introduced the primary graphics classes for JWAVE. You can read more about these classes and their methods in the JWAVE Javadoc reference. For information on Javadoc, see *Using the JWAVE Javadoc Reference* on page 40.

5

CHAPTER

# JWAVE Server Development

This chapter discusses the following topics:

- How to write JWAVE wrappers
- Retrieving parameters from the server
- Using GETPARAM to unpack value, positional, and keyword parameters
- Returning parameters to the client
- Returning multiple results
- Handling errors
- Testing wrapper functions

## JWAVE Server Overview

Client JWAVE Java programs communicate with PV-WAVE through a PV-WAVE function called a JWAVE wrapper. A JWAVE wrapper function is a PV-WAVE function that contains some specific JWAVE API calls. These JWAVE-specific functions are included in the dynamically loaded JWAVE library; therefore, to the JWAVE developer, these functions can be used just like any PV-WAVE function.

**TIP** The JWAVE wrapper API functions are described in Appendix A, *JWAVE Wrapper API*.

Typically, a JWAVE wrapper function is used as an interface between the JWAVE client and one or more PV-WAVE applications. The JWAVE wrapper receives parameters from the client, processes those parameters in some manner, and returns information back to the client. The returned information can be numeric or string values (except complex numbers), including scalars and arrays. The server can also return graphics to the client if PV-WAVE graphics commands were used in the JWAVE wrapper.

## *Writing JWAVE Wrapper Functions*

This section explains how to write JWAVE wrapper functions. JWAVE wrapper functions are PV-WAVE functions that contain JWAVE wrapper API calls, such as GETPARAM and GET_NAMED_COLOR. The wrapper API functions are described in Appendix A, *JWAVE Wrapper API*.

### Example: Simple JWAVE Wrapper

The following JWAVE wrapper:

- retrieves a single parameter from the client,
- "unpacks" the parameter
- processes the parameter
- returns a result

**Example 5-1** Simple JWAVE wrapper function

```
FUNCTION PASSARR, client_data

   ; Unpack parameters received from the client.
   arr =    GETPARAM(client_data, 'ARRAY', /Value)

   ; Change the array.
   mydata = arr * 1.5

   ; Return the changed array.
   RETURN, mydata
END
```

### *The Input Parameter: client_data*

All JWAVE wrappers have a consistent interface. They all are functions that accept one parameter called, by convention, `client_data`.

The `client_data` parameter passes parameter name(s) and data that were "packaged" by the client Java program. (JWAVE client developers use the `setParam` method of the JWaveExecute class to "package" the parameters.) When the Java client calls the `JWaveExecute.execute` method, the packaged parameters are automatically sent in an array to the appropriate JWAVE wrapper function. (The Java client application also specifies the name of the wrapper function it intends to execute.)

### *The GETPARAM Function*

The GETPARAM function unpacks the information sent in the `client_data` parameter. In Example 5-1, GETPARAM happens to retrieve an array. The GETPARAM arguments include:

***client_data*** — The parameter information that was passed to the wrapper function from the client.

***'ARRAY'*** — The name of the parameter to unpack. This name was assigned in the client Java program at the time the parameter was "packaged" (with the `setParam` method).

***/Value*** — The *Value* keyword tells GETPARAM to retrieve just the value of the parameter that was sent. (See the following note.)

**NOTE** In other situations, it is necessary for GETPARAM to return more than the value. For instance, in many cases, the JWAVE wrapper function will be used to execute one or more PV-WAVE functions. Typically, this is accomplished with the PV-WAVE EXECUTE command, and parameters from the client that were unpacked by GETPARAM are used to "build" a string containing the command for EXECUTE. To facilitate these cases, GETPARAM can return a string that is formatted appropriately. For example, if the *Positional* keyword were used instead of *Value*, GETPARAM would return a string of the form: `", ` *param_reference*`"` (where *param_reference* is a symbolic reference to a value). The result could then be used directly in an EXECUTE statement. For example:

```
x1=GETPARAM(client_data, 'ARRAY', /Positional)

status=EXECUTE("PLOT"+ x1)
```

We will discuss the use of the *Positional* and *Value* keywords later in this chapter.

#### *The RETURN Statement*

When RETURN is called in a JWAVE wrapper, the return parameters/data are automatically packaged and sent back to the client Java application. The client can access this data using the default return parameter name `DATA`. See *Returning Multiple Results to the Client* on page 60 for information on sending multiple results back to the client.

### Wrapper Functions Must Be Compiled

The JWAVE server operates in PV-WAVE runtime mode. Therefore, all functions and procedures (including wrapper functions) that run on the JWAVE server must be compiled into `.cpr` files using the PV-WAVE COMPILE procedure.

For example, a typical series of commands that you can use to compile a single PV-WAVE routine is:

```
WAVE> DELPROC, /All

WAVE> DELFUNC, /All

WAVE> .RUN mywrapper.pro

WAVE> COMPILE, /All, File='mywrapper.cpr'
```

---

**TIP** We have provided a set of routines that you can use to test your JWAVE wrapper functions before compiling them and publishing them on your Web server. See *Testing Wrapper Functions* on page 70 for more information.

---

## *Using GETPARAM to Unpack Parameters*

As explained in the previous section, parameters sent from the client are received as input by the JWAVE wrapper function on the server. The GETPARAM function (a PV-WAVE function) is used to retrieve and unpack these parameters. The GETPARAM function provides keywords that let you control how parameters are unpacked. This section discusses GETPARAM and how its keywords are used to unpack parameters.

### What Do You Want To Unpack?

When you write a JWAVE wrapper function, typically your goal is to retrieve parameters from the client and then execute some PV-WAVE functions that use those parameters.

For instance, if your JWAVE client application displays a 2D plot, you might send the following parameter information to the server:

- `X` — An array of floating point values to plot.
- `TITLE` — A title for the plot.
- `LINESTYLE` — Type of line to plot.

The `setParam` methods used on the client to set these parameters might look like this:

```
setParam(”X”, anarray)

setParam(”TITLE”, ”Peak Concentrations”)

setParam(”LINESTYLE”, 1)
```

When the client calls its `execute` method, the JWAVE wrapper on the server receives this information through its input argument, usually called `client_data`. You must decide how to unpack this data.

In this case, the goal is to execute a PV-WAVE PLOT command with the parameters that were received from the client. For example, you might want to run a PV-WAVE PLOT command with these parameters and keywords:

```
PLOT, X, Title=thetitle, Linestyle=thestyle
```

Generally, you can unpack values or command strings. These topics are discussed in following sections.

## Unpacking Values

You can use GETPARAM to unpack data in two ways: as values or as command strings. To unpack values, you specify the *Value* keyword with GETPARAM. For example:

```
x_val = GETPARAM( client_data, ’X’, /Value )
```

---

**NOTE** `client_data` is the single parameter passed to a JWAVE wrapper function, as in: `FUNCTION MY_WRAPPER, client_data`

---

In this instance, a reference to the value of the `X` parameter (set by a client’s `setParam` call) is assigned to the PV-WAVE variable `x_val`. Then, `x_val` can be used in any valid context. For example:

```
PLOT, SQRT( x_val )
```

### Using the Default Keyword

When you use the *Value* keyword, you may also use the *Default* keyword to specify a default value to be assigned if the client did not pass that parameter.

For example, if the client sets parameters as follows:

```
setParam(”X”, anarray)

setParam(”TITLE”, ”Peak Concentrations”)

setParam(”LINESTYLE”, 1)
```

The body of a JWAVE wrapper function that retrieves those parameters might look like this:

```
x_val = GETPARAM( client_data, 'X', /Value, Default=FINDGEN(10) )

the_title = GETPARAM( client_data, 'TITLE', /Value, Default='' )

lstyle = GETPARAM( client_data, 'LINESTYLE', /Value, Default=0 )

PLOT, x_val, Title=the_title, Linestyle=lstyle
```

This makes a plot with the specified data, a title, and a linestyle. If the client did not specify the data `X`, then the default dataset `FINDGEN(10)` is used. The default plot title is an empty string, and the default linestyle is 0 (solid).

### The Expect* Keywords

When using the *Value* keyword, you can use a set of *Expect** keywords to help you ensure that the data the client has passed is what you expect. These keywords cause GETPARAM to test the data to ensure that it meets your expectations, and an error is generated if the data is invalid. (If an error occurs, the client's `execute` method throws an exception.)

The *Expect** keyword options are:

- `/ExpectScalar`
- `/ExpectArray` or `ExpectArray` = *array_of_dimensions*
- `ExpectType` = *wave_type_code*
- `/ExpectNumeric`
- `/ExpectString`

These keywords help you to ensure that the client does not “break” the wrapper by passing unexpected or invalid data.

For examples that use the *Expect** keywords, see *Using the Expect Keywords* on page 69.

## Unpacking Command Strings

The other method for unpacking data allows you to build a command string for use with the PV‑WAVE EXECUTE function. In this mode, GETPARAM returns strings that can be concatenated together to form a coherent command string.

There are two distinct kinds of parameters in PV‑WAVE: positional and keyword. Thus there are two methods used to get these strings from GETPARAM. Positional and keyword parameters will be discussed in more detail in the next section, but as an example, the JWAVE wrapper code we saw in the previous section might be re-written as follows:

```
cmd = 'PLOT'
cmd = cmd + GETPARAM( client_data, 'X', /Positional )
cmd = cmd + GETPARAM( client_data, [ 'TITLE', 'LINESTYLE' ] )
status = EXECUTE( cmd )
```

A string representation of `X` is used as a positional parameter, and `LINESTYLE` and `TITLE` are used as keywords. The resulting string `cmd` might look something like this:

`"PLOT,` *x_reference*`, TITLE=`*title_reference*`, LINESTYLE=`*linestyle_reference*`"`

---

**NOTE** GETPARAM does not extract actual data from `client_data`; rather, it extracts a symbolic *reference* to data. The data that is referenced might have been sent by the client or retrieved from memory on the server. Therefore, the command:

```
x = GETPARAM(client_data, 'X', /Positional)
```

returns a string in the form " , *param_reference*", where *param_reference* is a reference to data that was either sent from the client or retrieved from the Data Manager. This concept is discussed further in the next few sections.

---

When you use GETPARAM without the *Value* keyword, you do not have the option of using the *Default* or *Expect** keywords. The returned string for a missing parameter (a parameter not passed by the client) is an empty string (that is, the keyword parameter is left off of the command if it was not supplied by the client.) You can easily check to ensure required parameters are received, as shown in Example 5-5 on page 62.

### *Positional vs. Keyword Parameters*

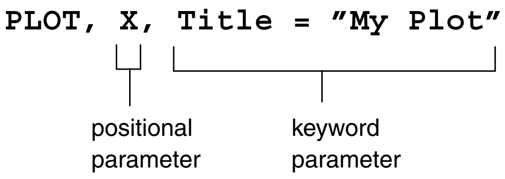
Note that the PLOT command that you need to build (for use in the EXECUTE function) consists of a positional parameter, `X`, and two keyword parameters, `Title` and `Linestyle`. In PV‑WAVE, positional parameters are of the form:

" , *param*"

while keyword parameters are of the form:

" , *KeywordName=Value*"

The following figure shows the PV-WAVE PLOT command with one positional parameter and one keyword:



The way in which you unpack parameters in the JWAVE wrapper depends on whether you are unpacking a positional or a keyword parameter.

### *Unpacking Positional Parameters*

Continuing the PLOT example used previously, the JWAVE wrapper function must use the GETPARAM function to unpack the parameters.

The first parameter to unpack is the data parameter `X`. Note that `X` is a positional parameter in the PV-WAVE PLOT command.

To unpack a positional parameter, use the GETPARAM function with the *Positional* keyword, as follows:

```
param1=GETPARAM(client_data, 'X', /Positional)
```

If the value assigned to `X` is an array, this function might return the following string in the `param1` variable.

" , *array_reference* "

where *array_reference* is a symbolic reference to the data. (The actual data could have been sent from the client or retrieved from the Data Manager.)

As noted before, this returned string is in the expected form of a positional parameter in a PV-WAVE function:

" , *param*"

The returned string can now be used to "build" a PLOT command using the EXECUTE function. For example:

```
ret=EXECUTE('PLOT' + param1)
```

### *Unpacking Keyword Parameters*

If the *Positional* keyword is not used, then GETPARAM assumes the parameter is a keyword parameter and returns a string in the form:

`", KeywordName=`*keyword_reference*`"`

where *keyword_reference* is a reference to the value of the keyword. (This value could have been sent from the client or retrieved from the server.)

For example, to unpack the *Title* keyword, use the following GETPARAM call:

```
title=GETPARAM(client_data, 'TITLE')
```

Here, the function returns the following string in the `title` variable:

`" , TITLE=`*title_reference*` "`

where *title_reference* is a symbolic reference to the title data.

---

**NOTE** Again, remember that GETPARAM does not extract and return an actual value from `client_data`. Rather, the function builds a symbolic *reference* to a value into the string.

---

## Building a PV-WAVE EXECUTE Command

The GETPARAM function returns strings that are formatted appropriately to be used in a PV-WAVE EXECUTE command. EXECUTE compiles and executes one or more PV-WAVE statements contained in a string at runtime.

For example:

The client application (written in Java) packs the data:

```
command= new JWaveView(connection, "JWAVE_PLOT")
int[] data = {1,2,3,4,5};
command.setParam("X", data);
command.setParam("TITLE", "CO2 Data");
command.setParam("LINESTYLE", 1);
```

On the server, JWAVE wrapper commands unpack the parameters and build an EXECUTE command:

```
result=GETPARAM(client_data, 'X', /Positional)
keywords=GETPARAM(client_data, ['TITLE', 'LINESTYLE'])
status=EXECUTE("PLOT" +result + keywords)
```

## Unpacking Color Data

Another JWAVE wrapper function, GET_NAMED_COLOR, is used to retrieve colors that were specified by the client application. See *Unpacking Color Information with GET_NAMED_COLOR* on page 67 for detailed information on this function.

## Returning Multiple Results to the Client

You can easily package multiple results (arrays and scalars) and return them to the client where they can be unpacked and used.

To package multiple results on the server, create and return an associative array of data name/value pairs. For example:

**Example 5-2** JWAVE wrapper function that returns multiple results to the client

```
FUNCTION PASSFFT, client_data
   ; Get the data from the client.
   arr =    GETPARAM(client_data, 'ARRAY', /Value)
   ;Process the data, creating two separate variable results
   freq = FLOAT(ABS(FFT(arr, -1)))
   distrib = HISTOGRAM(freq)
   ;Create an associative array to send results back to the client.
   RETURN, ASARR('FDATA', freq, 'HIST', distrib)
END
```

The resulting arrays, `freq` and `distrib` are packaged and returned to the client in the associative array. The associative array keys become the names of the returned data referenced by the client with the `getReturnData` method. The names of the data returned by this example are `FDATA` and `HIST`.

The following example shows a Java code fragment used to retrieve the parameters sent from the server and to properly cast the values.

**Example 5-3** Client calls to unpack the associative array sent from the server.

```
//Get the FFT data
float[] d = (float[]) command.getReturnData("FDATA");
//Get the histogram data
int[] h = (int[]) command.getReturnData("HIST");
```

## Returning Graphical Data to the Client

Example 5-4 shows a simple JWAVE wrapper that calls the PV-WAVE PLOT command to generate a 2D plot of the array received from the client. Graphics are automatically created on the server and sent back to the client for display.

---

**NOTE** The client developer uses the JWaveView class to request that the server return graphical data in addition to numerical data. Whenever JWaveView is used to execute a client request to the server, the server automatically creates a `Viewable` object. This object is then packaged and streamed back to the client where it can be displayed. If the client calls the wrapper with the JWaveExecute class, the graphics are discarded, and only the data are returned. For more information on JWaveView and graphics, see Chapter 4, *JWAVE Graphics*.

---

**Example 5-4** Simple JWAVE wrapper that returns a 2D plot

```
FUNCTION APLOT, client_data
; Unpack the parameters and data from the client.
   arr =    GETPARAM(client_data, 'ARRAY', /Value)
   PLOT, arr
RETURN, 0
END
```

---

**NOTE** You must make sure that all coordinate system information is correct before you return a plot to the client. Also, note that the SURFACE and AXIS procedures create a temporary axis transformation that is not automatically saved by the PV-WAVE session. To ensure that the correct transformation and coordinate system information is sent back to the client, use the *Save* keyword with these procedures. This causes the correct transformation information to be sent automatically to the client. For more information on coordinate transformations, see *Coordinate System Transformations* on page 49.

---

## Example: A Typical JWAVE Wrapper

This example demonstrates and reinforces concepts that were discussed previously in this chapter.

This JWAVE wrapper function retrieves positional and keyword parameters from the client, unpacks the parameters, and builds a command string for use in a PV‑WAVE EXECUTE function. This example also demonstrates the GET_NAMED_COLOR function, which allows colors to be passed by name from the client to the JWAVE wrapper.

**TIP** Refer to the *PV-WAVE Reference* for information on the EXECUTE, PLOT, and OPLOT commands. These PV‑WAVE commands are used in the following example.

The code for Example 5-5 is a simplified version of `jwave_plot.pro`, which can be found in:

| | |
|---|---|
| **(UNIX)** | `jwave-3_5/lib` |
| **(Windows)** | `jwave-3_5\lib` |

**Example 5-5** JWAVE wrapper that unpacks positional and keyword parameters and builds a command string

```
FUNCTION JWAVE_PLOT_SIMPLE, client_data

   ; Determine plot type (Can also be done with [XY]Type)
   CASE GETPARAM(client_data, 'SCALING', /Value, Default=0) OF
      1: cmd = 'PLOT_OI'
      2: cmd = 'PLOT_IO'
      3: cmd = 'PLOT_OO'
      ELSE: cmd = 'PLOT'
   ENDCASE

   ; Get colors -- default is black lines on white background
   back = GET_NAMED_COLOR('BACKGROUND', Default='000000'xL)
   fore = GET_NAMED_COLOR('COLOR', Default = 'ffffff'xL)
   acol = GET_NAMED_COLOR('AXIS', Default = fore)
   color_kwds = ', Background=back, Color=fore'

   ; Get allowed keywords for the PLOT command

    plot_kwds = GETPARAM(client_data, $
        [ 'Box', 'Charsize', 'Charthick', 'Clip', 'Gridstyle', $
        'Linestyle', 'Noclip', 'Nsum', 'Polar', 'Position', 'Psym',
   $
        'Solid_Psym', 'Subtitle', 'Symsize', 'Thick', 'Tickformat',
   $
        'Ticklen', 'Title', 'XCharsize', 'XGridstyle', 'XMargin', $
        'XMinor', 'XRange', 'XStyle', 'XTickformat', 'XTicklen', $
        'XTickname', 'XTicks', 'XTickv', 'XTitle', 'XType', $
        'YCharsize', 'YGridstyle', 'YMargin', 'YMinor', $
        'YNozero', 'YRange', 'YStyle', 'YTickformat', $
        'YTicklen', 'YTickname', 'YTicks', 'YTickv', $
        'YTitle', 'YType' ] )

   ; Get positional data
   y = GETPARAM(client_data, 'Y', /Positional) ; REQUIRED

   IF y EQ '' THEN $
     MESSAGE, 'Parameter Y is required.'

   x = GETPARAM(client_data, 'X', /Positional) ; Optional

   ; Execute the plotting function to draw the axes
   status = EXECUTE( cmd + x + y + color_kwds + plot_kwds )

   RETURN, 0
END
```

## Unpacking the Parameters

First, positional and keyword parameters sent from the client must be unpacked with GETPARAM.

**NOTE** Parameter names are not case sensitive. They must begin with a letter, and can contain only alphanumeric characters and the underscore character (_).

**NOTE** A client application that provides controls for generating and modifying the appearance of plots or other kinds of graphics probably needs to communicate positional and keyword parameter information to the server. The client user might use option menus to change the colors used in a plot, text fields to add plot titles, push buttons to add axes, and so on. The client developer must retrieve the parameter names and values from the user interface, package those parameters (with `setParam` and
`setNamedColor` method calls), and send them to the server. As shown here, the JWAVE wrapper function then unpacks the parameters, generates the plot, and sends back a graphic for display on the client.

### *Unpacking Values*

The first use of GETPARAM in Example 5-5 is basically the same as we've seen in previous examples (such as Example 5-1). Here, GETPARAM is used in a CASE statement to retrieve the appropriate PLOT command (PLOT, PLOT_IO, PLOT_OO, or PLOT_OI). The correct command is saved based on the value of the `SCALING` parameter that was passed from the client. As in previous examples, the *Value* keyword specifies that a simple value be returned. Also, the *Default* keyword is used to return a value, 0, if no `SCALING` parameter was sent from the client. In this case, no scaling causes the CASE statement to save the regular PLOT command.

```
CASE GETPARAM(client_data, 'SCALING', /Value, Default=0) OF
      1: cmd = 'PLOT_OI'
      2: cmd = 'PLOT_IO'
      3: cmd = 'PLOT_OO'
      ELSE: cmd = 'PLOT'
   ENDCASE
```

### *Unpacking Keywords*

The next GETPARAM call retrieves any of the supported plot keywords that may have been sent from the client. In this case, we include all of the keywords associated with the PV-WAVE PLOT command. Positional parameters that are specifically related to the *x* and *y* data will be retrieved in a separate GETPARAM statement.

```
plot_kwds = GETPARAM(client_data, $
      [ 'Box', 'Charsize', 'Charthick', 'Clip', 'Gridstyle', $
      'Linestyle', 'Noclip', 'Nsum', 'Polar', 'Position', 'Psym', $
      'Solid_Psym', 'Subtitle', 'Symsize', 'Thick', 'Tickformat', $
      'Ticklen', 'Title', 'XCharsize', 'XGridstyle', 'XMargin', $
      'XMinor', 'XRange', 'XStyle', 'XTickformat', 'XTicklen', $
      'XTickname', 'XTicks', 'XTickv', 'XTitle', 'XType', $
      'YCharsize', 'YGridstyle', 'YMargin', 'YMinor', $
      'YNozero', 'YRange', 'YStyle', 'YTickformat', $
      'YTicklen', 'YTickname', 'YTicks', 'YTickv', $
      'YTitle', 'YType' ] )
```

This statement extracts from `client_data` any keywords that were specified in the string array and their associated values. With this usage, the GETPARAM function returns a string array in the following form:

`", keyname_1=`*value_reference*`, keyname_2=`*value_reference*`, ... "`

where *value_reference* is a symbolic reference to a value that was either sent by the client or retrieved from the Data Manager.

For example, the client sends keyword/value pairs that the PLOT command expects, and GETPARAM unpacks them into a string array, containing keyword names and references to keyword values.

`" , Title=`*title_ref*`, Ticklen=`*tick_ref*`, Charsize=`*charsize_ref* `"`

As noted previously, this "string array" can be used as part of a command in an EXECUTE call to produce a plot.

**NOTE** The list of keywords given in this GETPARAM function example represents *all* of the keywords that can be extracted. Any keywords in the list that are not sent by the client are simply ignored by GETPARAM. If no keywords are sent, this GETPARAM function would return a null (empty) string.

**TIP** We recommend that you use a string array (as was done in this example) to specify which keywords you wish to retrieve with GETPARAM. By specifying a string of names in GETPARAM, rather than using the */All* keyword, you prevent your program from failing if the client sends unexpected information.

### *Unpacking Positional Parameters*

The final two GETPARAM calls use the *Positional* keyword to retrieve positional parameters.

```
y = GETPARAM(client_data, 'Y', /Positional) ; REQUIRED


   IF y EQ '' THEN $

     MESSAGE, 'Parameter Y is required.'


   x = GETPARAM(client_data, 'X', /Positional)
```

The difference between keyword and positional parameters is their form. Keywords are of the form *Keyname=value*. For example, `XTitle="Units Sold"`. Positional parameters, on the other hand, of the form: *paramname*. For example, the PLOT command takes a required positional parameter (`Y`), an optional positional parameter (`X`), and numerous optional keywords. For example:

```
PLOT, y, Title="CO2 Content", Charsize=3
```

where *y* is a required positional parameter, and the other parameters are keywords.

When the *Positional* keyword is specified, GETPARAM returns the specified parameter in a string in the format: `", ` *param_reference* `"`, where *param_reference* is a symbolic reference to the data. The leading comma is needed because when this string is used in an EXECUTE command, it is concatenated with other strings to "build" a command.

If the required *y* parameter is not supplied by the client, this error is "trapped" (because GETPARAM returns an empty string) and the MESSAGE procedure is called. The effect of the MESSAGE procedure is discussed in *Error Handling* on page 68.

## Unpacking Color Information with GET_NAMED_COLOR

The GET_NAMED_COLOR function retrieves a color index from a color name set by the client. The client specifies a color name and a Java `Color` object. This object is sent to the JWAVE wrapper and GET_NAMED_COLOR translates that color object into a color that can be used by PV-WAVE.

In this example, GET_NAMED_COLOR retrieves three colors that were specified by the client: `BACKGROUND`. `COLOR`, and `AXIS`.

**TIP** For more information on JWAVE graphics and color parameters, see Chapter 4, *JWAVE Graphics*. See also *Managing the Color Table* on page A-18.

**Example 5-6** JWAVE wrapper calls retrieve colors sent from the client.

```
back = GET_NAMED_COLOR('BACKGROUND', Default='000000'xL)
fore = GET_NAMED_COLOR('COLOR', Default = 'ffffff'xL)
axis = GET_NAMED_COLOR('AXIS', Default = fore)
```

**TIP** The following lines show the corresponding calls that were made in the Java client program to set the colors retrieved in Example 5-6:

```
myJWaveView.setNamedColor("BACKGROUND", java.awt.Color.lightGray)
myJWaveView.setNamedColor("COLOR", java.awt.Color.red)
myJWaveView.setNamedColor("AXIS", java.awt.Color.black)
```

These calls set colors to be sent to the server. The calls associate a color name (e.g., `BACKGROUND`) with a Java color object (e.g., `Color.lightGray`). These colors are packaged with the rest of the parameters and sent to the client.

## The RETURN Statement

The RETURN statement in a JWAVE wrapper can always be used to return data to the client. In this example, the primary function of the wrapper is to return a graphic—this particular wrapper does not generate any data that needs to be returned to the client. Thus, the RETURN statement simply returns 0. There is no reason, however, why this RETURN statement could not be used to return some other data.

## *You Can Only Retrieve Parameters Once*

GETPARAM can only retrieve a parameter (positional, keyword, or value) once.

For example, the following GETPARAM calls retrieve parameters from the client:

```
foo = GETPARAM(client_data, 'foo')
```

```
bar = GETPARAM(client_data, /All)
```

However, after the `foo` parameter is retrieved, it cannot be retrieved again in the subsequent call with the */All* keyword. Therefore, in this case, the `bar` command string will not contain the `foo` parameter.

**TIP** You can use the *Ignore_Used* keyword with GETPARAM to request that all requested parameters will be returned whether they have been used or not.

## *Error Handling*

This section discusses some error handling techniques and tips for use in JWAVE wrapper functions.

### Using the MESSAGE Procedure

Use the PV-WAVE MESSAGE procedure to generate error messages from JWAVE wrapper functions. The error text that MESSAGE produces is automatically sent to the client as a Java Exception (specifically, a `JWaveWrapperException` object). The MESSAGE procedure also causes the wrapper function to stop processing (by default).

For detailed information on MESSAGE, refer to the *PV-WAVE Reference*.

### Trapping Errors

If you use PV-WAVE error handling routines (such as ON_ERROR_GOTO or ON_ERROR) to trap and handle errors in a JWAVE wrapper function, be sure to set !Error=0 immediately after the error trap to ensure that the error is not reported back to the client.

ON_ERROR and ON_ERROR_GOTO are described in the *PV-WAVE Reference*.

## Using the Expect Keywords

GETPARAM takes a set of optional keywords that can be used for trapping input errors. These keywords are only used in conjunction with the *Value* keyword. (This is because they are used to test values that are passed from the client.)

These keywords all begin with the word *Expect*. They are:

- *ExpectType = type_code*
- */ExpectNumeric*
- */ExpectString*
- *ExpectArray* = [ *dim1*, *dim2*, ... ] or */ExpectArray*
- */ExpectScalar*

The first three keywords (*ExpectType*, *ExpectNumeric*, and *ExpectString*) produce an error if the returned parameter does not match the expected data type. The valid data type codes that you can use with JWAVE are listed in the following table.

| Type Code | Data Type |
|---|---|
| 1 | Byte |
| 2 | Integer |
| 3 | Longword integer |
| 4 | Floating point |
| 5 | Double precision floating |
| 7 | String |

The keyword *ExpectArray* produces an error if the input parameter is not an array of the specified dimensions. For more information on the *Expect** keywords, see *GETPARAM Function* on page 11.

The keyword *ExpectScalar* produces an error if the input parameter is not a scalar or a 1-element array. If the parameter is a 1-element array, then the function returns that element as a scalar.

As the following lines show, you can use reasonable combinations of the *Expect** keywords in a single GETPARAM function.

```
; y must be a numerical array
y = GETPARAM(client_data, 'Y', /Value, /ExpectArray, /ExpectNumeric
   )
; xcnt and ycnt must be numerical scalars
xcnt = GETPARAM(client_data, 'XCENTER', /Value, Default = 0.5, $
      /ExpectScalar, /ExpectNumeric )
ycnt = GETPARAM(client_data, 'YCENTER', /Value, Default = 0.5, $
     /ExpectScalar, /ExpectNumeric )
; title must be a scalar string.
title = GETPARAM(client_data, 'TITLE', /Value, Default = 'The
    Title', $
     /ExpectScalar, /ExpectString)
```

## Testing Wrapper Functions

This section explains how to test JWAVE wrapper functions without writing client Java programs. The WRAPPER_TEST_* procedures are PV-WAVE procedures that you can use to imitate the behavior of the Java client application, such as setting parameters, setting colors, and setting and executing the JWAVE wrapper.

### Testing a Numerical Program

For example, let's look at how you can test the JWAVE wrapper `simple.pro` that we discussed in Chapter 1, *JWAVE System Introduction*. This wrapper accepts a number and returns the square root of the number to the Java client. To test this wrapper without writing the Java client application yet, you can run the following test procedures in PV-WAVE (assuming that the directory that contains `simple.pro` is in the !Path system variable of PV-WAVE):

```
WAVE> WRAPPER_TEST_INIT, 'SIMPLE'
WAVE> WRAPPER_TEST_SETPARAM, 'NUMBER', 2
WAVE> WRAPPER_TEST_EXECUTE
WAVE> WRAPPER_TEST_RETURN_INFO
DATA            FLOAT     =       1.41421
WAVE>
```

Here we set the wrapper, set a parameter, executed the wrapper, and printed the return data.

The procedures used to test this wrapper correspond to the Java methods used to set parameters, and so on, in the client application. The following table shows the correspondence between these PV-WAVE test routines and the Java methods they imitate:

| **Test Procedure** | **JWAVE Java Method** |
|---|---|
| `WRAPPER_TEST_INIT` | `JWaveExecute.setFunction` and `JWaveView.setSize` (optional) |
| `WRAPPER_TEST_SETCOLOR` | `JWaveView.setNamedColor` |
| `WRAPPER_TEST_SETPARAM` | `JWaveExecute.setParam` |
| `WRAPPER_TEST_EXECUTE` | `JWaveExecute.execute` |
| `WRAPPER_TEST_GETRETURN` | `JWaveExecute.getReturnData` |
| `WRAPPER_TEST_RETURN_INFO` | `Parameter.printInfo` |

## Testing a Graphics Program

You can also use the WRAPPER_TEST_* routines to test wrapper functions that generate graphics. To do this, specify the window size to WRAPPER_TEST_INIT, as shown in one of the examples below. If the wrapper returns a graphic, then the test procedure WRAPPER_TEST_EXECUTE displays the graphic in a PV-WAVE window.

Here is an example that tests the wrapper `jwave_plot.pro`. You can find this wrapper function in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/jwave-3_5/lib` |
| **(Windows)** | `VNI_DIR\jwave-3_5\lib` |

where `VNI_DIR` is the main Visual Numerics installation directory.

```
WAVE> WRAPPER_TEST_INIT, 'JWAVE_PLOT', 300, 300

WAVE> WRAPPER_TEST_SETCOLOR, 'BACKGROUND', '000000'xL ; black

WAVE> WRAPPER_TEST_SETCOLOR, 'LINE', 'ff0000'xL ; blue

WAVE> WRAPPER_TEST_SETPARAM, 'Y', HANNING(20,20)

WAVE> WRAPPER_TEST_SETPARAM, 'PSYM', 1

WAVE> WRAPPER_TEST_EXECUTE
```

Here, note that the WRAPPER_TEST_SETCOLOR procedure is used to set line and background colors. When WRAPPER_TEST_EXECUTE is run, the plot is displayed in a PV-WAVE graphics window.

For detailed information on each of the WRAPPER_TEST_* routines, see Appendix A, *JWAVE Wrapper API*.

## Summary

This chapter explained how to develop and test JWAVE wrapper functions. A JWAVE wrapper function is a PV-WAVE function that communicates, through the JWAVE Manager, with a JWAVE client (Java) application. JWAVE wrappers use functions for “unpacking” parameters sent from the client. The RETURN statement in a JWAVE wrapper automatically packs and returns parameters and data to the Java client.

You can use standard PV-WAVE error trapping routines to trap errors in a JWAVE wrapper function. If you trap and handle errors within the wrapper function, be sure to reset the !Error system variable to 0 after the error is trapped to prevent an error message from being returned to the client.

Remember that JWAVE wrapper functions, and any PV-WAVE functions that run on the JWAVE server, must be compiled with the PV-WAVE COMPILE command.

6

CHAPTER

# Managing Data

If you are working with large datasets, you probably want to keep them on the server. Typically, the server has the memory and storage resources that large datasets require. On the other hand, client machines typically have limited memory and storage capacity. Furthermore, the transfer of large datasets between the client and server (for example, across the Internet) can consume network resources.

JWAVE addresses this problem with *data proxies*. A proxy is an object-oriented design term that refers to a surrogate or placeholder that controls access to another object. A data proxy, then, is a surrogate object that refers to data.

If the data is, for instance, a large multidimensional array stored in a PV-WAVE session, the client can use a proxy object to refer to that data. Using a proxy object, the client can request operations to be performed on the data, such as:

- running an analysis or filtering program on the data
- copying the data
- destroying the data
- renaming the data
- retrieving the data (returning it to the client)

All of these operations are controlled from the client while the data physically resides on the server, which greatly reduces the burden on client and network resources. Unless data is explicitly retrieved, it never has to be downloaded to the client.

## *What is a JWAVE Data Proxy*

A Proxy class is an interface that describes a data proxy. The subclass of the Proxy class used by JWAVE clients to refer to data that is stored on the server is JWaveDataProxy.

Usually, a JWAVE wrapper function processes data in some way and stores the results. The JWAVE client may wish to use another JWAVE wrapper to perform further analysis on the stored results. To do this, the client needs to be able to reference the data stored on the server. The most efficient way to do this is to ask the server to return a data proxy, which the client can then use in the `setParam` method of subsequent function calls from the first wrapper.

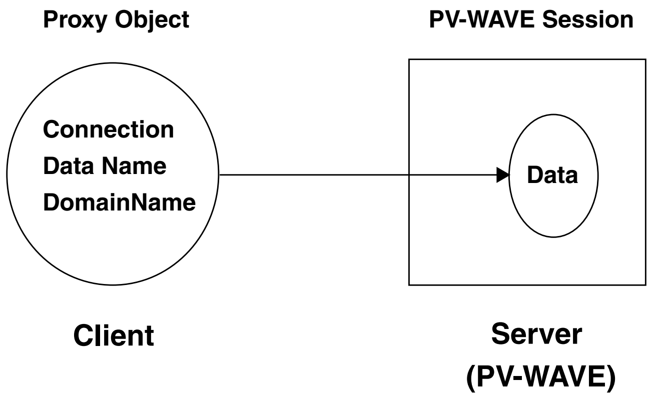


**Figure 6-1** A Proxy object refers to data stored in a PV-WAVE session. The proxy knows where to find the server (JWaveConnection), the name of the data domain name for the data (ServerDataID).

While the end result of this processing might be some kind of plot or image that is sent back to the client, the important point to remember is that the actual data never has to leave the server.

How the data arrives on the server in the first place is up to the JWAVE application developer. The data could have originated on the client and been uploaded to the server. Or, the data could have been loaded on the server from a file or database, or calculated by a JWAVE wrapper function.

The data referred to by a `JWaveDataProxy` object persists on the server until it is explicitly deleted or the PV-WAVE session is shut down.

## Instantiating a JWaveDataProxy Object

A `JWaveDataProxy` object is instantiated with:

- a valid connection to the server (a `JWaveConnection` object)
- a name by which to identify the data on the server, and
- a domain name (used to group datasets)

For instance, one way to construct a `JWaveDataProxy` object is:

```
      new JWaveDataProxy(myconnection, 'MYDATA', 'MYDOMAIN')
```

where `myconnection` is the previously instantiated `JWaveConnection` object (tells the proxy how to find the server), `MYDATA` is the name of the data stored on the server, and `MYDOMAIN` is the domain.

When you construct a `JWaveDataProxy` object in this way, you are only providing the name of the data. This `JWaveDataProxy` object refers to a place to store data, but there is no guarantee that there is actually data there. If you do not know that there is already data on the server, you can use the `Proxy.store` method to send data to the PV-WAVE session.

## Other Ways to Instantiate a JWaveDataProxy Object

There are two other ways to create a `JWaveDataProxy` object, and they both are used in conjunction with a `JWaveExecute` object.

The first way is to supply a `ServerDataID` object (which encapsulates the name and domain of a `JWaveDataProxy` object) when you use the `setParam` method.

```
setParam(String name, Object value, ServerDataID dataName)
```

This associates values with the parameter name, as usual, but additionally the server stores the data as named by the `ServerDataID` object when it is sent with the next `execute` call.

This technique is useful when you have some data on the client, and you want to use that same dataset with several JWAVE wrapper functions. So you use the above `setParam` method to save the data on the first call, and then you use the following `setParam` on subsequent calls to associate a named parameter with that stored data:

```
setParam(String name, ServerDataID dataName)
```

This technique stores data on the server, even though you are not manipulating `JWaveDataProxy` objects directly. If you need direct access to the data (like to delete it when you are done) you can use the same `ServerDataID` object that identifies the data name in a constructor to a new `JWaveDataProxy` object.

Another way to create a data proxy is when data is returned by a JWAVE wrapper function. Normally, the JWAVE wrapper returns values back to the client, but you can change this. Before you call the `execute` method, you can use the `setReturnParamMode` method to change how returned data will be handled.

There are a couple of overloaded methods for `setReturnParamMode`, but the primary one is:

```
setReturnParamMode(String name, boolean returnVals, ServerDataID
   dataName)
```

If you use this method, and specify a `ServerDataID` object for the data name, then that parameter will be stored on the server when the JWAVE wrapper function returns. You can also use the boolean flag with one of the values `JWaveExecute.RETURN_NO_VALUES` or `JWaveExecute.RETURN_VALUES`. If you set this flag to `RETURN_NO_VALUES`, then no values are returned, but a `JWaveDataProxy` object to the stored data is returned.

This technique is useful for "chaining" the output of one JWAVE wrapper to the input of another, as you will see in a later example.

There are other ways to use `setReturnParamMode` object. If you set the flag to `RETURN_VALUES` and you set a `ServerDataID` object, then the data is stored and also returned to the client. You can even discard data that you do not want by setting `RETURN_NO_VALUES` without a `ServerDataID` object.

**TIP** For detailed information on all of the constructors, variables, and methods of the JWaveDataProxy class, refer to the Javadoc reference. For information on Javadoc, see *Using the JWAVE Javadoc Reference* on page 40.

## The Efficiency of Using Data Proxies

This section illustrates the efficiency of using data proxies in JWAVE applications.

### Inefficient System: The Data Makes Two Round Trips

In one typical JWAVE scenario, the client asks the server (a JWAVE wrapper function) to load or generate some data and then asks another JWAVE wrapper to process the data and return a `Viewable` object back to the client.

*Figure 6-2* illustrates an inefficient version of this scenario, where data makes two complete round trips between the client and the server. This scenario can be summarized as follows:

- A `JWaveExecute` object executes a JWAVE wrapper function called `DATAGEN` on the PV-WAVE server.
- The generated data is then returned by the JWAVE wrapper to the client.
- The client passes the data to another `JWaveExecute` object, which sends it back to the server as input to another JWAVE wrapper called `FILTER`. (The data has made one round trip.)
- The filtered data is returned by the JWAVE wrapper to the client.
- The client passes the data to a `JWaveView` object, which sends it back to the server as input to another PV-WAVE routine called `GRAPHICS`. (The data has now made two round trips.)
- Finally, a plot is passed back to the client and displayed.

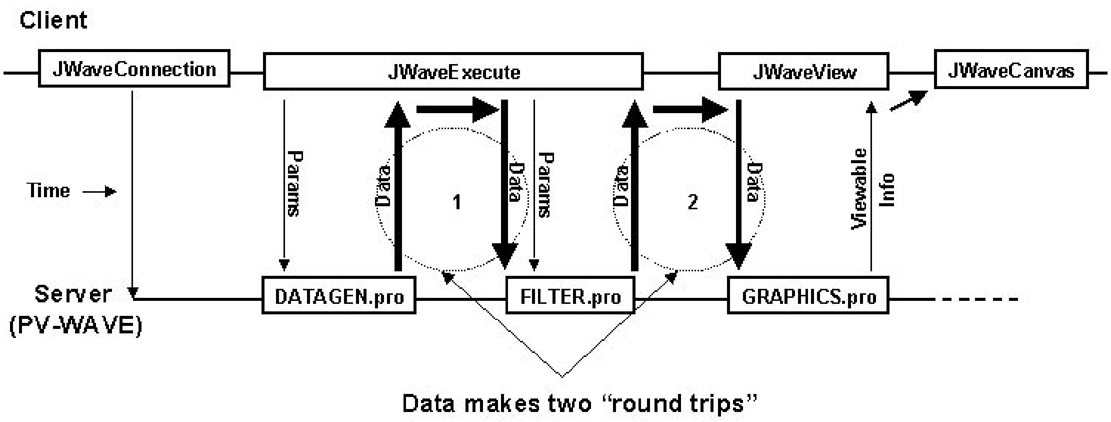


**Figure 6-2** Inefficient use of JWAVE, as data is physically passed between server and client

## Efficient System: No Round Trips

*Figure 6-3* depicts the same type of processing as the previous figure; however, this time the client uses data proxies to refer to the data on the server. The actual dataset is never physically returned to the client. Only after processing is completed is an image object returned to the client and displayed.

In this case, note that the object returned to the client each time is a data proxy, rather than actual data. The client uses the proxy object in subsequent `JWaveExecute` methods (where, in the previous (inefficient) model, the actual data was passed).

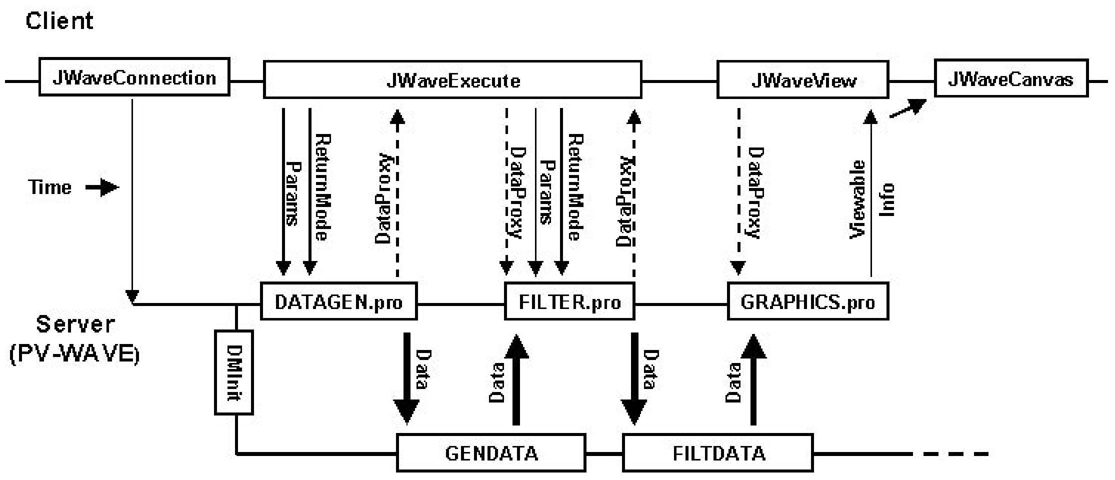


**Figure 6-3** Efficient use of JWAVE, as data is not physically passed between server and client

## Setting the Return Parameter Mode

By default, a JWAVE wrapper function returns physical data to the client. The client has to specify that it wants the JWAVE wrapper to return a proxy object. To do this, the client uses the `JWaveExecute.setReturnParamMode` method.

In the next section, we present an example illustrating the use of data proxies.

## Example: Using Data Proxies

As previously noted, sending large datasets back and forth between client and server can be expensive in terms of network and client resources. In many cases, it is often more efficient to leave data on the server for at least part of the processing cycle.

This example discusses a JWAVE application that:

- creates an array of data on the client
- sends the data to the server where it is "processed" by a JWAVE wrapper function
- tells the server to return a proxy object rather than the actual data
- requests the server to execute a *different* JWAVE wrapper function to process the data further
- returns the actual processed data to the client and prints it

*Figure 6-4* illustrates the flow of this JWAVE application.

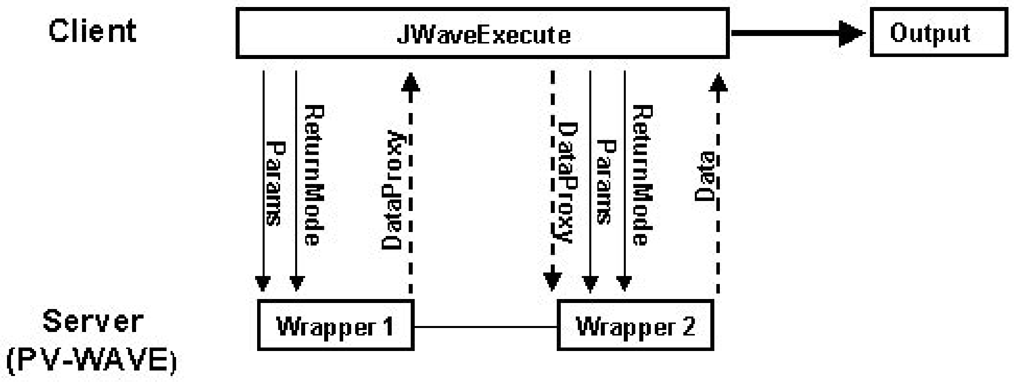


**Figure 6-4** JWAVE scenario using data proxies

**NOTE** This client application calls two different JWAVE wrapper functions on the server. From the first wrapper, the client asks for a proxy to be returned. From the second wrapper, the client asks for the actual data values, which it then prints.

## *The Java Client Program*

**Example 6-1** proxyarry.java: Sends data to the server; retrieves a data proxy; uses the proxy in a subsequent JWaveExecute object

```
import com.visualnumerics.jwave.*;

public class ProxyArray {

   public static void main(String[] args) {

      try {

  // Create a simple array as data

  float[] arr = new float[10];

  for (int i=0; i<arr.length; ++i)

     arr[i] = i;

  // Connect to JWave server

  JWaveExecute command = new JWaveExecute(”PROX1”);

  // Set the data as ARRAY1 parameter

  command.setParam(”ARRAY1”, arr);

  // Ask that the result data (named DATA) be stored on the server,

  // and only return a Proxy

  ServerDataID saveData =  new ServerDataID(”SAVE_NAME”);

  command.setReturnParamMode(”DATA”, JWaveExecute.RETURN_NO_VALUES,

        saveData);

  // Execute PROX1

  command.execute();

  // Switch to new wrapper function

  command.setFunction(”PROX2”);

  // Clear previous parameters and return modes

  command.clearParams();

  command.clearReturnParamModes();

  // Set input to PROX2 to use data stored by PROX1

  // Could also use getReturnProxy(”DATA”) here rather than saveData

  command.setParam(”ARRAY2”, saveData);

  // Execute PROX2 and get result

  command.execute();
  float[] answer = (float[]) command.getReturnData(”DATA”);

  // Print result

  for (int i=0; i<answer.length; ++i)

     System.out.println(answer[i]);

      } catch (Exception e) {

  // Report problems

  System.out.println(e.toString());
      }
   }
}
```

### The First JWAVE Wrapper

**Example 6-2** prox1.pro: Receives an array from the client and multiplies the elements by 1.5. The client asks the server to return a data proxy rather than the actual data.

```
FUNCTION PROX1, client_data

   ; Unpack data and parameters sent from the client.
   arr =    GETPARAM(client_data, ’ARRAY1’, /Value, Default=11)

   ; Change the array.
   mydata = arr * 1.5

   ; Return the changed array.

   RETURN, mydata
END
```

### The Second JWAVE Wrapper

**Example 6-3** prox2.pro: The client asks this server program to process the array it stored previously. The client refers to the data on the server with a proxy. Finally, the client asks the server to send back the actual data so that it can be printed. (This wrapper multiplies the previously stored array by 100.5.)

```
FUNCTION PROX2, client_data

   ; Unpack data and parameters sent from the client.

   arr =    GETPARAM(client_data, ’ARRAY2’, /Value, Default=11)

   ; Change the array.

   mydata = arr * 100.5

   ; Return the changed array.

   RETURN, mydata

END
```

### *Java Client Output*

When the Java program in *Example 6-1* is executed, the following output is printed on the client. (The numbers result from multiplying the first array by 1.5 and multiplying the resulting array by 100.5.)

```
% java proxyarr
0.0
150.75
301.5
452.25
603.0
753.75
904.5
1055.25
1206.0
1356.75
```

### Data Proxies Are Controlled by the Client

Note that in this example there is no special code in the JWAVE wrapper functions to enable the storing of data as proxies. The use of proxy data is totally under client control, and you do not need to do anything special in the wrapper functions to control this. This client-side control of proxies allows you to build JWAVE wrappers as modules, without planning all of the possible uses for them. As you build the client Java application, these JWAVE wrapper modules can then be "hooked together" in the best possible ways.

### How Long is Proxy Data Stored on the Server

Data that is referred to by a proxy object persists for the duration of the PV-WAVE session in which it is stored. When the session ends, the data is lost.

**TIP** You can use the Data Manager routines DMSave and DMRestore to store data between sessions. These JWAVE wrapper procedures are described in Appendix A, *JWAVE Wrapper API*.

## *Summary*

A proxy is an object that represents data stored on the server (in PV-WAVE). Use data proxies to make JWAVE applications more efficient. By using proxies to refer to server data, you can reduce the burden on client and network resources.

7

CHAPTER

# Using JWAVE Beans

JWAVE Beans are JavaBeans$^{TM}$ that provide a set of graphical views for displaying data. As JavaBeans, JWAVE Beans are portable and platform independent. You can often use JWAVE Beans in visual application builder tools.

Visual Numerics has provided a set of JWAVE Beans that correspond to the main graphical views of PV-WAVE. The JWAVE Beans shipped with JWAVE are:

- JWAVE Bar3d Tool — a Bean for displaying bar charts
- JWAVE Contour Tool — a Bean for displaying contour images
- JWAVE Generic Tool — a Bean for displaying the results of your own PV-WAVE function and procedure files
- JWAVE Histogram Tool — a Bean for displaying histograms
- JWAVE Pie Tool — a Bean for displaying pie charts
- JWAVE Plot Tool — a Bean for displaying plot images
- JWAVE Surface Tool — a Bean for displaying surface plot images

## Using JWAVE Beans with the BeanBox

This section is a step-by-step exercise that explains how to use JWAVE Beans with JavaBeans Development Kit from Sun Microsystems, Inc. (You must use BDK version 1.0 July 98, or a later version). The latest version of the BDK can be downloaded from the JavaSoft web site:

```
http://java.sun.com/beans/index.html
```

This exercise uses one invisible Bean, JWAVE Bean tester (which generates some data), and two visible Beans, JWAVE Surface Tool and OrangeButton, to produce a surface plot.

## Step 1: Modify Your CLASSPATH

You must add the following files to your `CLASSPATH` variable:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/JWaveConnectInfo.jar` |
| **(UNIX)** | `VNI_DIR/classes/JWave.jar` |
| **(UNIX)** | `SWINGDIR/swing.jar` |
| **(Windows)** | `VNI_DIR\classes\JWaveConnectInfo.jar` |
| **(Windows)** | `VNI_DIR\classes\JWave.jar` |
| **(Windows)** | `SWINGDIR\swing.jar` |

where `VNI_DIR` is the main Visual Numerics Installation directory, and `SWINGDIR` is the directory where Swing version 1.1 or later is installed (see the *Glossary*).

## Step 2: Copy JWaveBeans.jar

Copy the file:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/JWaveBeans.jar` |
| **(Windows)** | `VNI_DIR\classes\JWaveBeans.jar` |

where `VNI_DIR` is the main Visual Numerics installation directory.

to

| | |
|---|---|
| **(UNIX)** | `BDKDIR/jars` |
| **(Windows)** | `BDKDIR\jars` |

where `BDKDIR` is the directory in which the BeanBox is installed.

When the BeanBox starts, this JAR file is automatically opened and inspected.

## Optional: Modify the Startup Script

Normally, the following script is used to start the BeanBox:

| | |
|---|---|
| **(UNIX)** | `BDKDIR/beanbox/run.sh` |
| **(Windows)** | `BDKDIR\beanbox\run.bat` |

where `BDKDIR` is the directory where the BeanBox is installed.

**TIP** You can make the `CLASSPATH` modifications described in Step 1 directly in this script, so that `CLASSPATH` includes `$CLASSPATH` (UNIX) or `%classpath%` (Windows).

## Step 3: Start the BeanBox

Before you start the BeanBox, make sure the JWAVE Manager is running and properly configured (see *Testing to See If the JWAVE Manager is Running* on page 110).

**NOTE** You must CD to the `BDKDIR/beanbox` directory to run `run.sh` or `run.bat`. where `BDKDIR` is the directory in which the BeanBox is installed.

To start the BeanBox, from the `BDKDIR` type:

| | |
|---|---|
| **(UNIX)** | `run.sh` |
| **(Windows)** | `run.bat` |

When you run the BeanBox, the graphical JWAVE Beans and the "invisible" JWAVE Bean tester Bean are listed in the ToolBox along with other Beans that came with the BDK or that you added (*Figure 7-1*).

## Step 4: Try Out the JWAVE Bean Tester and Surface Tool

**NOTE** Remember, the JWAVE Manager must be running to use JWAVE Beans.

**Step 1** From the ToolBox, select JWAVE Bean tester, then click in an open area of the BeanBox window to place it as an object in the BeanBox.

**Step 2** Select a button (such as OrangeButton) from the ToolBox and place it near the bottom of the BeanBox window.

**Step 3** Select the button object in the BeanBox, and then select **Edit=>Events=>button push=>actionPerformed**.

**Step 4** Click over the JWAVE Bean tester object in the BeanBox. This links the button object to the JWAVE Bean tester object.

**Step 5** In the EventTargetDialog dialog box, select the `start2d` target method (this method will form the connection between the JWAVE Bean tester and the push button), then click OK. Now, every time you click the button, the tester Bean will generate new 2D data.

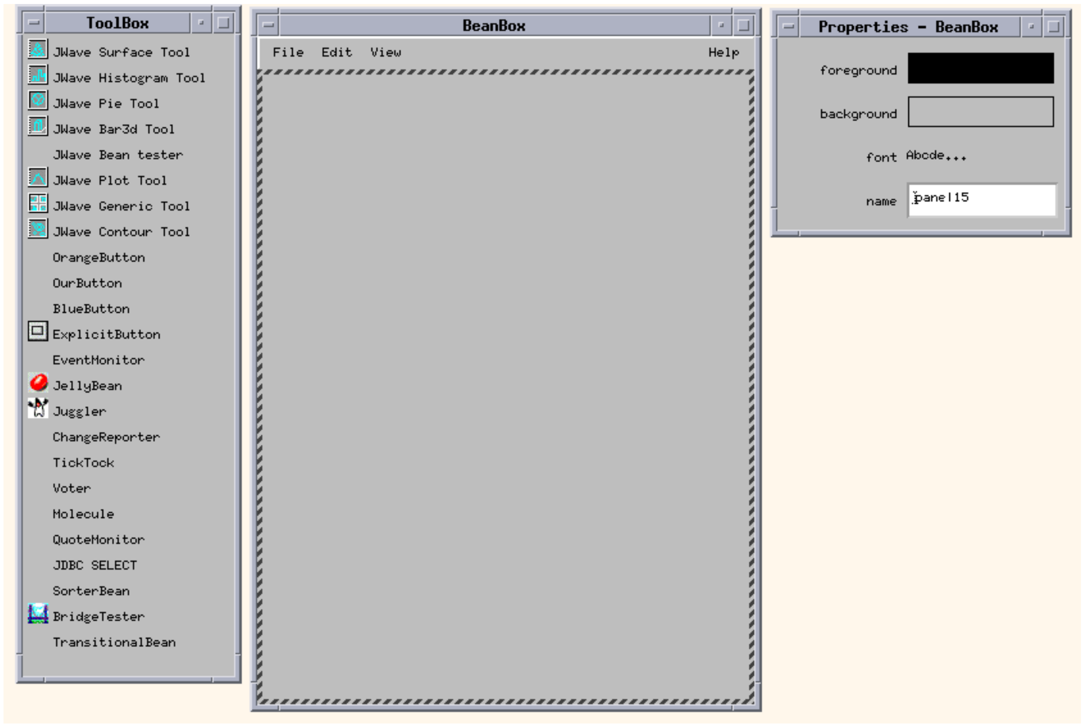


**Figure 7-1** JWAVE Beans listed in the BeanBox ToolBox

**Step 6** In the ToolBox, select JWAVE Surface Tool and place it above the button object in the BeanBox.

**Step 7** Resize the Surface Tool Bean until it is big enough to hold a surface plot.

**Step 8** Select the JWAVE Bean tester object in the BeanBox, then from the Edit menu, select **Events=>propertyChange=>propertyChange**.

**Step 9** Click over the **Surface Tool** in the BeanBox. This links the Bean tester to the Surface Tool.

**Step 10** In the EventTargetDialog dialog box, select the `propertyChange` target method as the event, then click OK.

**Step 11** Click the button object in the BeanBox.

The Bean tester generates 2D data and sends that data to the Surface Tool. Next, the Surface Tool instructs JWAVE to generate a plot. Then the surface plot displays in the BeanBox (see *Figure 7-2*).

## Step 4: Customize the Surface Bean

Each visible JWAVE Bean has a Customizer interface for changing the values of parameters that affect the appearance of the plot.

**TIP** The customizable parameters for JWAVE Beans are described in Appendix E, *JWAVE Bean Tools Reference*.

**Step 1** Select the surface object in the BeanBox, then from the Edit menu, select **Events=>Customize...** The Customizer window for the JWAVE Surface Tool Bean appears.

**Step 2** Change the settings for some of the surface plot's parameters, then click Done.

**Step 3** Click the button in the BeanBox. The surface plot is re-generated and then displayed with the new parameter values applied.

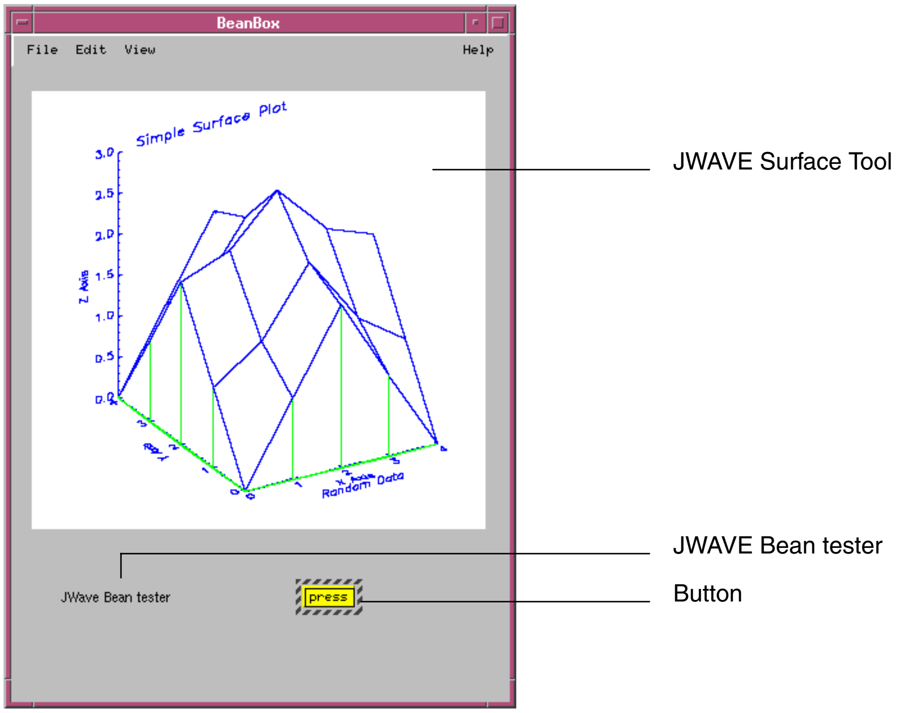


**Figure 7-2** The surface plot displayed by the JWAVE Surface Tool. The JWAVE Surface Tool has been customized to add a skirt and change the default colors.

## Building a JWAVE Bean

This section describes how to construct a JavaBean and use it with the JWAVE Beans. Throughout this description, the source for the JWAVE Bean tester Bean (used in the previous section) is used in the discussion of the Bean development techniques. You can find this source code in:

**(UNIX)** `VNI_DIR/classes/com/visualnumerics/jwave/beans/JWaveBeanTest.java`

**(Windows)** `VNI_DIR\classes\com\visualnumerics\jwave\beans\JWaveBeanTest.java`

where `VNI_DIR` is the main Visual Numerics installation directory.

**NOTE** The intended audience for this section is Java developers and those familiar with object-oriented programming. This section is by no means a comprehensive source for constructing a Bean. There are many books and Web sites on the subject that take a much more thorough look at the capabilities of JavaBeans.

### Deciding What the Bean Will Do

Before a JavaBean can be developed, you must decide what the Bean needs to do. JavaBeans are just objects. It is a good idea to create an overall design document of your project to determine which specific objects are needed and exactly what each object will do.

Once the Bean's requirements are defined, it should be fairly obvious what the Bean should look like. Most Beans are visible components (buttons, fields, or canvases–the JWAVE Beans Tools are canvases). However, it is not required that Beans have any visible attributes at all. The JWAVE Bean tester is one of these "invisible" Beans that has no GUI elements at all. It is just a class that has input and output, and that generates data and events.

If your Bean does need to have visible components, just code them as you would any other class. Beans are required to have a `no argument` constructor, but you may use this constructor like any other, building the elements your class needs in it.

### Adding Properties to the Bean

*Properties* are named attributes that define the state of an object. JavaBeans are not required to have any properties, but since they define the state of the Bean, they most likely will. A property can be something as simple as the string appearing on a button, to more complex things such as a multi-dimensional dataset.

In terms of Beans, properties are exposed to visual programming tools and are usually editable through some sort of property editing interface (see *Telling a Bean Environment How to Use Your Bean* on page 95, and *Building a Customizer for the Bean* on page 104).

The naming convention followed by Beans developers for naming methods that access Bean properties is:

```
    public void set*PropertyName*(*PropertyType* value);
public *PropertyType* get*PropertyName*();
```

As shown in Example 7-1, the `set` method ***writes*** the data to the Bean, while the `get` method ***reads*** the data from the Bean. By having both methods, the property is defined as read/write enabled. By omitting one of the methods, the property becomes either read-only or write-only.

In the case of the JWAVE Bean tester Bean, the only properties included are the data objects that contain canned datasets.

**Example 7-1** A get/set pair, taken from JWAVE Bean tester

```
DoubleTable dataTable_;


/**
* Set the dataTable Proxy
* @see JWaveBeanTest#getDataTable
* @param Proxy The dataTable Proxy this bean will use
*/
public void setDataTable(Proxy p)
{
    proxy_=p;
    Object pObject=proxy_.retrieve();

    try
    {
        dataTable_=(DoubleTable)pObject;
    }
    catch(ClassCastException e)
    {
        System.out.println(”Data was not a DoubleTable”);
    System.out.println(e);
    }
}

/**
* Get the dataTable Proxy
* @see JWaveBeanTest#setDataTable
* @return Proxy The dataTable Proxy
*/
public Proxy getDataTable()
{
    if(dataTable_!=null)
    {
        Proxy p=new
   com.visualnumerics.data.LocalProxyImpl(dataTable_);
        return(p);
    }
    else
        return(null);
}
```

**TIP** Properties do not have to be data members of the class. There can be `get/set` methods that calculate values or return hardcoded constants.

## Handling Data

### *Using the Proxy Class to Exchange Data*

All of the JWAVE Beans take and send data as `com.visualnumerics.data.Proxy` objects. The Proxy class is an interface that contains a reference to a data object. The intent of the Proxy class is to allow data to be passed that is not necessarily inside your virtual machine (such as a `CORBA` object).

The class `com.visualnumerics.jwave.JWaveDataProxy` implements the Proxy class, making it valid to pass to the Beans. The `JWaveDataProxy` class is a client side proxy for data that is resident on the JWAVE server.

When data is to be sent to a JWAVE Bean (either through an event or a bound property), it must be a `Proxy` object.

The three methods shown in Example 7-2 are the input (`set2D_Data`) and output (`get2D_Data` and `start2d`) for a two-dimensional array of data (`data2d_`). The use of these methods is discussed further in a later section (page 103), but for now, notice how the data either comes in or goes out of the methods as a `Proxy` object. The class `com.visualnumerics.data.LocalProxyImpl` is an implementation of `Proxy` for data that is already resident inside your virtual machine.

**Example 7-2** Data being sent as a Proxy object in JWAVE Bean tester

```
double[][] data2d_;


/**
* Set the 2D data Proxy
* @see JWaveBeanTest#get2D_Data
* @param Proxy The 2D data Proxy this bean will use
*/
public void set2D_Data(Proxy p)
{
    proxy_=p;
    Object pObject=proxy_.retrieve();

    try
    {
        data2d_=(double[][])pObject;
       Proxy p=new com.visualnumerics.data.LocalProxyImpl(data2d_);
        changes_.firePropertyChange(”data_2d_change”,null,p);
    }
    catch(ClassCastException e)
    {
        System.out.println(”Data was not a 2D double”);
        System.out.println(e);
    }
}
```

```
/**
* Get the 2D data Proxy
* @see JWaveBeanTest#set2D_Data
* @return Proxy The 2D data Proxy
*/
public Proxy get2D_Data()
{
    if(data2d_!=null)
    {
       Proxy p=new com.visualnumerics.data.LocalProxyImpl(data2d_);
        return(p);
    }
    else
        return(null);
}

/**
* Fire an event that the 2D data has changed
*/
public void start2d()
{
    Proxy p=new com.visualnumerics.data.LocalProxyImpl(data2d_);
    changes_.firePropertyChange("data_2d_change",null,p);
}
```

### *Using Events to Exchange Data*

All JavaBeans essentially communicate with *events*. When a Bean has something happen to it (the user pressed a button, new data arrived, a condition was met, and so on), the Bean may want to communicate this event to other Beans. The communication can be in the form of actions, mouse clicks, data, or any other event that Java is capable of producing. Any class that is registered as a listener can receive this event and handle it in its own way.

In the case of the JWAVE Beans, one way of passing data between them is via a `PropertyChangeEvent`. This class is part of the `java.beans` package and is a standard way for Beans to communicate.

Most Beans implement `java.beans.PropertyChangeListener` and are therefore capable of receiving and handling `PropertyChangeEvents`.

All JWAVE Beans contain an instance of `java.beans.PropertyChangeSupport`. This class maintains a list of classes that have registered as listeners and allows events to be sent to those listeners.

**Example 7-3** Instances of PropertyChangeEvents from JWAVE Bean tester

```
PropertyChangeSupport changes_=new PropertyChangeSupport(this);
/**
* Adds a property change listener
* @param PropertyChangeListener listener
*/
public synchronized void addPropertyChangeListener(PropertyChangeListener l)
{
    changes_.addPropertyChangeListener(l);
}

/**
* Removes a property change listener
* @param PropertyChangeListener listener
*/
public synchronized void removePropertyChangeListener(PropertyChangeListener
             l)
{
    changes_.removePropertyChangeListener(l);
}

/**
* Fire an event that the 2D data has changed
*/
public void start2d()
{
    Proxy p=new com.visualnumerics.data.LocalProxyImpl(data2d_);
    changes_.firePropertyChange(”data_2d_change”,null,p);
}
```

The PropertyChangeSupport class `changes_` maintains a list of classes that wish to listen to this Bean's events. Any class that calls `addPropertyChangeListener` will be added to that list. Similarly, any class that calls `removePropertyChangeListener` will be removed from the list.

When the `start2d` method is called, any class that has registered as a listener will receive the data found in the `data2d_` array (as a `Proxy` object of course).

The `firePropertyChange` method of the `PropertyChangeSupport` object takes as arguments the name of the property that has changed, the old value of that property, and the new value of the property. These are the *exact* fields of the `java.beans.PropertyChangeEvent` class that `PropertyChangeListeners` expect to see.

In a Bean development environment (such as the BeanBox), when the `start2d` method is invoked (by a button click, for instance), any Bean connected to the JWAVE Bean tester `propertyChange` event will receive the `data2d_` proxy.

## Including Bound Properties to Exchange New Data

*Bound properties* are properties that support change events. When a Bean's property changes (new data, user change, and so on), other Beans may want to know about it. When a bound property changes, an event is fired to all listeners.

All of the data properties in the JWAVE Beans are bound properties. When the data in a Bean changes, it will fire an event to all registered listeners. This event will contain the new data.

In *Example 7-4*, notice how, when the Bean has received new 2D data, it fires a `PropertyChangeEvent`. All Beans listening will receive a `Proxy` with the new data array inside.

**Example 7-4** An input method to the JWAVE Bean tester Bean

```
/**
* Set the 2D data Proxy
* @see JWaveBeanTest#get2D_Data
* @param Proxy The 2D data Proxy this bean will use
*/
```

```
public void set2D_Data(Proxy p)
{
    proxy_=p;
    Object pObject=proxy_.retrieve();

    try
    {
        data2d_=(double[][])pObject;
       Proxy p=new com.visualnumerics.data.LocalProxyImpl(data2d_);
        changes_.firePropertyChange("data_2d_change",null,p);
    }
    catch(ClassCastException e)
    {
        System.out.println("Data was not a 2D double");
        System.out.println(e);
    }
}
```

Bound properties have another advantage. In the BeanBox, if you select a Bean that has bound properties, a Bind Property ... choice is available under the Edit menu for displaying a list of the Bean's bound properties. Selecting one of those bound properties allows you to connect it to another Bean's bound property. The source bean will have its `get` method called and the destination Bean will have its `set` method called with the data retrieved from the `get` call. In this way, two Beans can share property values without any events.

## Telling a Bean Environment How to Use Your Bean

When your Bean is placed in a Bean environment (such as the BeanBox), the first thing the environment does is called *reflection*. Essentially, the environment is using naming conventions to figure out what your Bean is capable of doing. However, there are times when there is more you want known about a Bean than can be supplied through naming conventions. There are also times when conventions fall by the wayside. When this happens, it is time to use a BeanInfo class to provide explicit information about your Bean.

## *Using a BeanInfo Class*

A BeanInfo class must implement the `java.beans.BeanInfo` interface. However, since implementing this class requires implementations for each method in the BeanInfo interface, Java has provided a `java.beans.SimpleBeanInfo` class. Subclassing or extending the `java.beans.SimpleBeanInfo` class allows you to use only the methods you need for your Bean (the remainder have dummy implementations already defined). These methods allow you to tell the Bean environment what your Bean can do.

BeanInfo class names follow this convention:

> *BeanName*`BeanInfo.java`

When the Bean environment looks for a BeanInfo about your Bean, it searches only for the above naming convention. If the BeanInfo class is not in the same package as your Bean, your `CLASSPATH` is searched. The down side of all of this is that the Bean and the BeanInfo are tightly coupled and it's up to the developer to keep this relationship. The developer must be careful to not allow one class to evolve without the other.

Let's take a look at for the BeanInfo for the JWAVE Bean tester Bean. The Java code, purpose, and details of these methods are presented on the following pages:

- BeanDescriptors — Example 7-5 on page 96
- Bean Events — Example 7-6 on page 97
- Bean Methods — Example 7-7 on page 98
- Bean Properties — Example 7-8 on page 101

**Example 7-5** BeanDescriptors

```
private BeanDescriptor  beanDescriptor_;


private final static Class beanClass_ =
   com.visualnumerics.jwave.beans.JWaveBeanTest.class;


/**
* Default Constructor. This just sets-up the BeanDescriptor
*/
public JWaveBeanTestBeanInfo()
{
    beanDescriptor_ = new BeanDescriptor(beanClass_);
    beanDescriptor_.setDisplayName("JWAVE Bean tester");
    beanDescriptor_.setShortDescription("A test bean for use with
   JWAVE Tools");
}

/**
* This method returns the BeanDescriptor.
* @return BeanDescriptor
*/
public BeanDescriptor getBeanDescriptor()
{
    return beanDescriptor_;
}
```

The `Constructor` and the `getBeanDescriptor` methods both deal with the `java.beans.BeanDescriptor` class. This class is needed by the Bean environments to describe features of the Bean. The only fields in this class are the Bean's `Customizer` (see *Building a Customizer for the Bean* on page 104) and the Bean itself.

JWAVE Bean tester's `BeanDescriptor` only defines:

- the Bean itself
- the display name to use when showing the Bean
- a short description of the Bean's purpose.

**Example 7-6** Bean Events

```
/**
* This method describes which events of the Bean will be exposed.
* The only event exposed is propertyChange.
* @return EventSetDescriptor[] an array of EventSetDescriptors
* detailing which events of this bean are exposed.
*/
public EventSetDescriptor[] getEventSetDescriptors()
{
    try
    {
        EventSetDescriptor changed = new EventSetDescriptor
                  (beanClass, "propertyChange",
                  java.beans.PropertyChangeListener.class,
                  "propertyChange");
        changed.setDisplayName("bound property change");
        EventSetDescriptor[] rv = {changed};
        return rv;
    }
    catch (IntrospectionException e)
    {
        throw new Error(e.toString());
    }
}
```

The `getEventSetDescriptors` method returns an array of `java.beans.EventSetDescriptor` objects. Each object in the array describes an event supported by the Bean. The `getEventSetDescriptors` method therefore allows you to limit which events your Bean can fire. This is especially useful when your Bean is a Java component. Most components have many events they can deal with (actions, mouse events, window events, and so on) and you may not want your Bean to have to deal with all of them. The `getEventSetDescriptors` method allows you to set which events your Bean can handle.

In the source code in *Example 7-6*:

- only one event, `propertyChange`, will be seen by the Bean environments
- an `EventSetDescriptor` class is instantiated
- a display name to be used when showing events is set
- an array of just the one `EventSetDescriptor` is passed back to the caller

**Example 7-7** Bean Methods

```
/**
* This method describes which methods of the Bean will be exposed.
* @return MethodDescriptor[] an array of MethodDescriptors
* detailing which methods of this bean are exposed.
*/

public MethodDescriptor[] getMethodDescriptors()
{
    // First find the "method" objects.
    Method proxy2d,proxy2dRand,proxy1dRand;
    Method proxySin,proxyCos;
    Method proxyDataTable,proxyPieTable;
    Method start2d,startSin,startCos,startDataTable,startPieTable;
    Method start2dRand,start1dRand;
    Class proxyEventArgs[] = {com.visualnumerics.data.Proxy.class};
    try
    {
        proxy2d = beanClass_.getMethod(”set2D_Data”,
                                            proxyEventArgs);
        proxy2dRand = beanClass_.getMethod(”set2D_RandomData”,
                                            proxyEventArgs);
        proxy1dRand = beanClass_.getMethod(”set1D_RandomData”,
                                            proxyEventArgs);
        proxySin = beanClass_.getMethod(”setSineData”,
                                        proxyEventArgs);
        proxyCos = beanClass_.getMethod(”setCosineData”,
                                        proxyEventArgs);
        proxyDataTable = beanClass_.getMethod(”setDataTable”,
                                              proxyEventArgs);
        proxyPieTable = beanClass_.getMethod(”setPieTable”,
                                             proxyEventArgs);
        startPieTable= beanClass_.getMethod(”startPieTable”,
                                            null);
        startDataTable= beanClass_.getMethod(”startDataTable”,
                                             null);
        startSin = beanClass_.getMethod(”startSine”,
                                        null);
        startCos = beanClass_.getMethod(”startCosine”,
                                        null);
        start2d = beanClass_.getMethod(”start2d”,
                                       null);
        start2dRand = beanClass_.getMethod(”start2dRandom”,
                                           null);
        start1dRand = beanClass_.getMethod(”start1dRandom”,
                                           null);
    }
    catch (Exception ex)
    {
        throw new Error( "Missing method: " + ex );
    }

    // Now create the MethodDescriptor array
    // with visible event response methods:
    MethodDescriptor result[] = {new MethodDescriptor(proxy2d),
                             new MethodDescriptor(proxy2dRand),
                             new MethodDescriptor(proxy1dRand),
                             new MethodDescriptor(proxySin),
                             new MethodDescriptor(proxyCos),
                          new MethodDescriptor(proxyDataTable),
                          new MethodDescriptor(startDataTable),
                           new MethodDescriptor(proxyPieTable),
                           new MethodDescriptor(startPieTable),
                             new MethodDescriptor(startSin),
                             new MethodDescriptor(startCos),
                             new MethodDescriptor(start2d),
                             new MethodDescriptor(start2dRand),
                            new MethodDescriptor(start1dRand)};
    return result;
}
```

The `getMethodDescriptors` method returns an array of `java.beans.MethodDescriptor` objects. This object describes a method supported by a Bean. This array of descriptors details all of the methods your Bean will have exposed to the Bean environment. The use of the `getMethodDescriptors` methods allows you to limit which methods can be exposed.

In the source code in *Example 7-7*, 14 methods are returned to the caller:

- First, a `java.lang.reflect.Method` class is created for each method to be exposed. The `Method` class is created by supplying the name and the arguments of the Bean's method. When a method has no arguments, a `null` is supplied to the `Method` constructor.
- Second, a `MethodDescriptor` class is created using each of the `Method` classes created earlier. These `MethodDescriptor` classes are returned to the caller.

**Example 7-8** Bean properties

```
/**
* This method details which bean properties are exposed and how
* they are to be edited.
* @return PropertyDescriptor[]
*/

public PropertyDescriptor[] getPropertyDescriptors()
{
    try
    {
      PropertyDescriptor data2PD = new PropertyDescriptor(”data2d”,
                                                          beanClass_,
                                                        ”get2D_Data”,
                                                       ”set2D_Data”);
       data2PD.setDisplayName(”2D Data”);
       data2PD.setBound(true);

      PropertyDescriptor sinPD = new PropertyDescriptor(”sine wave”,
                                                          beanClass_,
                                                       ”getSineData”,
                                                      ”setSineData”);
       sinPD.setDisplayName(”Sine Data”);
       sinPD.setBound(true);

       PropertyDescriptor cosPD = new PropertyDescriptor(”cosine
   wave”,
                                                          beanClass_,
                                                     ”getCosineData”,
                                                    ”setCosineData”);
       cosPD.setDisplayName(”Cosine Data”);
       cosPD.setBound(true);

       PropertyDescriptor tablePD = new
   PropertyDescriptor(”dataTable”,
                                                          beanClass_,
                                                               ”getDataTable”,
                                                              ”setDataTable”);
          tablePD.setDisplayName(”2D Table Data”);
          tablePD.setBound(true);

         PropertyDescriptor piePD = new PropertyDescriptor(”pieTable”,
                                                                   beanClass_,
                                                               ”getPieTable”,
                                                               ”setPieTable”);
          piePD.setDisplayName(”Pie Data”);
          piePD.setBound(true);

          PropertyDescriptor data2RPD=new
      PropertyDescriptor(”randData2d”,
                                                                   beanClass_,
                                                          ”get2D_RandomData”,
                                                         ”set2D_RandomData”);
          data2RPD.setDisplayName(”2D Random Data”);
          data2RPD.setBound(true);

          PropertyDescriptor data1RPD=new
      PropertyDescriptor(”randData1d”,
                                                                   beanClass_,
                                                          ”get1D_RandomData”,
                                                         ”set1D_RandomData”);
          data1RPD.setDisplayName(”1D Random Data”);
          data1RPD.setBound(true);
         PropertyDescriptor rv[] = {data2PD,sinPD,tablePD,piePD,cosPD,
                                   data2RPD,data1RPD};
          return rv;
      }
      catch (IntrospectionException e)
      {
          throw new Error(e.toString());
      }
   }
```

The `getPropertyDescriptors` method returns an array of `java.beans.PropertyDescriptor` objects. These objects describe properties the Bean will expose. The use of `getPropertyDescriptors` not only allows you to limit which properties are available to the Bean environment, but specific property attributes (such as names, which methods `get/set` this property, bound status, and so on) and specific property editor classes can also be set.

### *Using Property Editor Classes*

Property editors are classes that a Bean environment uses to allow the property to be changed. The BeanBox arranges these property editors on its Property Sheet window. For every property listed, the BeanBox attempts to assign an editor class. For instance, if the property is a `Color` object, the BeanBox assigns a class that allows the user to edit colors. If the property is a `String`, the BeanBox assigns a `TextField` so the user can change the string.

There are two instances when this automatic assignment of editors fails:

- The first case is when the property accepts the assignment of only specific values (such as a list of linestyles). The developer can write a specific class that only allows the user to select one of the specific values applicable to the property. This class can then be assigned to the `PropertyDescriptor` object using the `setPropertyEditorClass` method.
- The second case where automatic editor assignment fails is when the property is of a type for which a known editor does not exist. This case occurs in the source code in *Example 7-8*. Because the properties are arrays or specific classes, the BeanBox reports that it cannot find a property editor for these properties. This condition is only a problem if you want those properties to be editable. If you do, then you must write a class to edit the properties and assign it using the `setPropertyEditorClass` method. If the property does not need and editor (such as is the case in this source), then simply don't assign one.

When using the property editors on the Property Sheet window, you are limited in the presentation of the editors. You cannot arrange them or use complex GUI components. A better alternative is to use a Customizer class (see *Building a Customizer for the Bean* on page 104).

The source code shown in *Example 7-8* creates a `PropertyDescriptor` object for each property to be exposed. This object is created by supplying:

- the property name
- the Bean

- the method name to '`get`' this property
- the method name to '`set`' this property

You can see this is a way to get around the `get/set` convention; you may supply any method name you wish. Each property is supplied with a display name. This is the name that appears on the Property Sheet. Finally, each property is set as a bound property. The default for this value is false, so this method call is not needed if the property is not to be bound. All of these `PropertyDescriptor` objects are then returned to the caller.

## Building a Customizer for the Bean

*Customizers* are classes that edit Bean properties. There is one Customizer class per Bean; however, a Bean is not required to have a Customizer. Although there are no naming conventions for Customizer classes, they must implement the `java.bean.Customizer` class and they must be a subclass of `java.awt.Component`. Customizers are most often GUI interfaces that allow the user to change Bean properties. It is a way for a developer to design a professional looking editor class for the Bean.

### *Methods*

The `java.beans.Customizer` interface has just three methods:

- `addPropertyChangeListener`
- `removePropertyChangeListener`
- `setObject(Object` *bean*`)`

The `add/removePropertyChangeListener` methods are described in *Using Events to Exchange Data* on page 92. The Customizer should fire a `PropertyChangeEvent` when it changes a property of the Bean.

The `setObject(Object` *bean*`)` method of `java.beans.Customizer` is automatically called by the Bean environment when a Customizer is started. This method should query the *bean* object for its current status and use that information to fill in the GUI Customizer components. The method of querying the Bean is the `set/get` methods which are used in almost all Bean transactions.

Finally, a Customizer must have a `no argument` constructor. As with all other Bean classes, there is no method instantiating the Customizer so there are no available arguments to pass.

Since there is no naming convention for Customizers, the Bean environment has no way of telling which class is the Customizer unless it is told. This is accomplished in the BeanInfo class.

The BeanDescriptor class has another constructor that will take the Customizer class in addition to the Bean class.

**Example 7-9** Example from a BeanInfo class

```
/**
* java.lang.Class object. This is just the name of the bean
* this BeanInfo is representing.
*/
private final static Class beanClass =
   com.visualnumerics.jwave.beans.JWaveBar3dTool.class;

/**
* java.lang.Class object. This is just the name of the Customizer
   class
* this bean uses.
*/
private final static Class beanCustomizerClass =
   com.visualnumerics.jwave.beans.JWaveBar3dToolCustomizer.class;

public JWaveBar3dToolBeanInfo()
{
   beanDescriptor_ = new
   BeanDescriptor(beanClass,beanCustomizerClass);
   String displayName = resources.getString("display_name");
   beanDescriptor_.setDisplayName(displayName);
   String descName = resources.getString("short_description");
   beanDescriptor_.setShortDescription(descName);
}
```

The JWAVE Bean tester Bean has no Customizer. This is because it has no properties that are editable by the user. However, all of the remaining JWAVE Beans do have Customizers. These Customizers are based on the Java Foundation Classes (JFC) graphical components and allow the user to edit every property of the Bean except for the data.

**TIP** To open a Customizer in the BeanBox, select a Bean and choose **Edit=>Customize**. If the Bean does not have a Customizer, then this option is not available.

## Adding Serializability to the Bean

One of the advantages to using JavaBeans is the ability to generate applications from the Beans you put together. When **File=>MakeApplet** is chosen on the BeanBox, the BeanBox attempts to serialize the Beans and make them into an applet. Serialization is the process of making the state of each Bean persistent. When you make an applet, the BeanBox captures the properties and hookups of each Bean and uses them in the resulting applet. The values found in the Beans are written to a `java.io.FileOutputStream` object to end-up in the applet code.

Unfortunately, serializing is not something that happens automatically. Each Bean must implement the `java.io.Serializable` interface. In most cases, this makes the Bean serializable. There are, of course, cases when this is not enough.

When serialization takes place, Java does not deal with transient variables of your Beans. They're transient; they are not part of the state of your Bean. However, any data member that is part of your class will be serialized. In general, most standard Java API classes are serializable and won't pose any problem. There are a few however, that are not (such as `java.awt.Image`). For these classes, or those not part of the Java API, you need to implement two methods to do the serialization. These methods are the `writeObject` and `readObject` methods of the Serializable class. The implementation of the methods is beyond the scope of this document, but it entails the breaking down of classes that aren't serializable into their serializable parts. For instance, `java.awt.Image` is not serializable. To serialize `java.awt.Image` you must get the data values out of the class and write those values to the stream. For a more thorough discussion on serializing, consult a book or Web site on JavaBeans development.

**TIP** If you generate an applet from your JWAVE Beans, you must modify the HTML file generated by the BDK. The `ARCHIVE` tag in the HTML file must include `JWave.jar`, `JWaveBeans.jar`, `JWaveConnectInfo.jar`, and `swing.jar`.

**NOTE** If you want to run your generated applet with `appletviewer`, you must put the JAR files listed in the previous Tip in your `CLASSPATH`. This is because `appletviewer` does not pick up the settings of the `ARCHIVE`.

8

CHAPTER

# JWAVE Server Configuration

This chapter explains how to set up and configure the main portions of your JWAVE system. This chapter discusses:

- Setting up the JWAVE server
- Testing the JWAVE server installation
- Setting up for JWAVE client development

## Installation Overview

To install JWAVE, follow the instructions in the CD-ROM booklet. Additional information on the software installation is in the `README` file on the installation CD. When you install JWAVE from the CD, you are presented with the following three installation options:

- ***PV-WAVE*** — Installs PV-WAVE, which includes JWAVE components used for server-side development of JWAVE wrappers.
- ***JWAVE Client Development Kit*** — Installs components used for client Java application development (JWAVE class library, reference documents, and JWAVE Beans). See *Figure 8-1*.
- ***JWAVE Server*** — Installs JWAVE server-side components required for deployment of JWAVE applications (JWAVE Manager, JWAVE Web Server, JWAVE Servlet, communication with JWAVE clients, and PV-WAVE). See *Figure 8-2*.

## Client-Side after JWAVE Installation

The JWAVE Client Development installation produces the following directory structure:

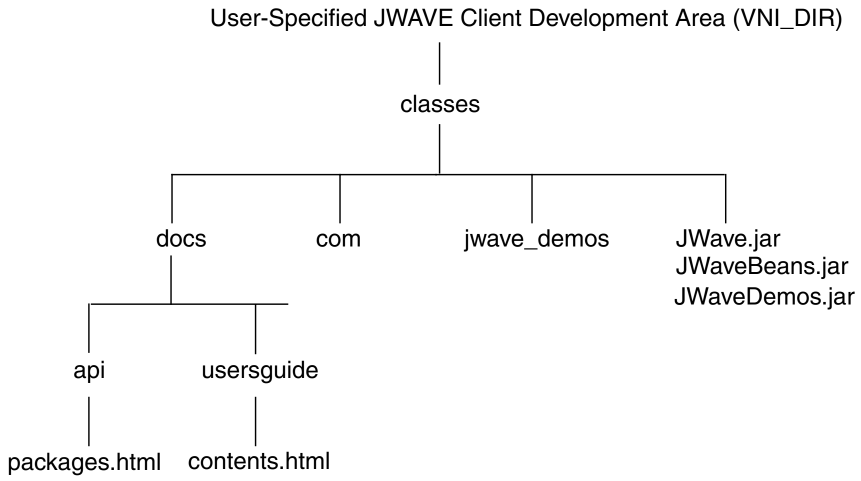


**Figure 8-1** JWAVE client directory structure

## Server-Side after JWAVE Installation

The JWAVE Server installation produces the following directory structure:

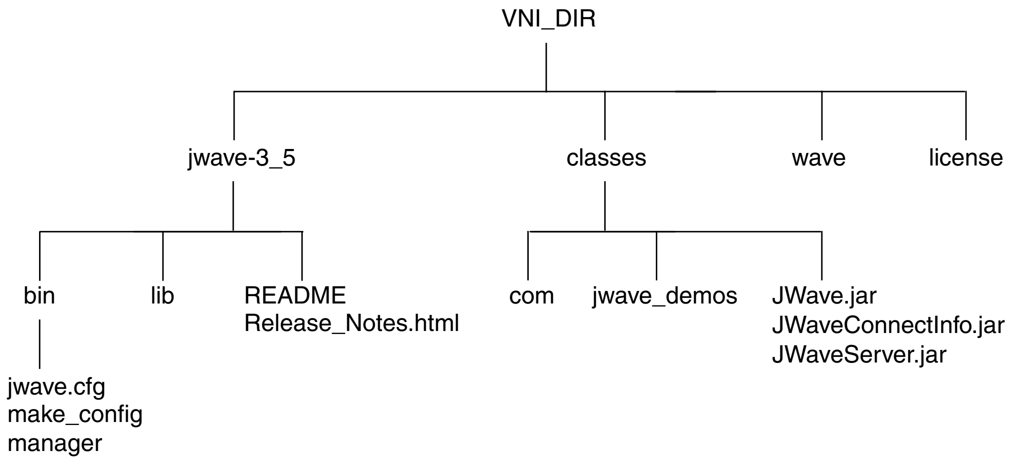


**Figure 8-2** JWAVE server directory structure, where VNI_DIR is the main Visual Numerics installation directory. Other directories may exist, depending on installation options.

### Additional Software Requirements

The following software must also be installed before you can use JWAVE:

***JWAVE Client Development*** — A Java Development Kit (JDK$^{TM}$) (Version 1.2 or later).

***JWAVE Client Side*** (optional for using Beans) — Beans Development Kit (BDK) (Version 1.0 July 98 or later), Swing (Version 1.1 or later).

***JWAVE Server*** — A Java Runtime Environment (JRE) (Version 1.2 or later).

## *Running and Testing the JWAVE Server*

The JWAVE server includes the JWAVE Manager software, JWAVE class files, and PV-WAVE. This section describes:

- Starting the JWAVE Manager
- Testing to see if the JWAVE Manager is Running
- Configuring the JWAVE Manager for HTTP Connections

The third section, Configuring the JWAVE Manager for HTTP Connections, is a short, step-by-step procedure that explains how to start the JWAVE Manager as a Web server and contact the server from a client Web browser.

For more detailed information on the JWAVE Manager, see *Setting Up the JWAVE Server* on page 112.

### *Starting the JWAVE Manager*

To start the JWAVE Manager on the JWAVE server:

**Windows USERS** If the JWAVE Service has been installed and configured (see *Installing the JWAVE Service* on page 128), open the Services window from the Control Panel, select JWAVE Service, then click Start to start the JWAVE Service).

**(UNIX)**

```
cd VNI_DIR/jwave-3_5/bin
manager start
```

**(Windows)**

```
cd VNI_DIR\jwave-3_5\bin
manager start
```

where `VNI_DIR` is the main Visual Numerics installation directory.

When it starts up, the JWAVE Manager writes information to the location currently defined for the MANAGER_LOG parameter in the JWAVE Configuration Tool (see page 123). The information in the log will vary depending on how the server is configured. Generally, the messages report all of the PV-WAVE sessions that are started and other status information.

```
Parsing persistent sessions:

Session Pool is starting new session (1/1)....

[Information on individual PV-WAVE sessions started on the
 server appears here...]

Waiting for connection...
```

For information on stopping the JWAVE Manager, see *Shutting Down the JWAVE Manager* on page 113.

## *Testing to See If the JWAVE Manager is Running*

A simple “ping” test verifies that the JWAVE Manager is running and communicating on the server. Simply enter the following command on the same machine as the JWAVE Manager is running:

**(UNIX)**

```
cd VNI_DIR/jwave-3_5/bin
manager ping
```

**(Windows)**

```
cd VNI_DIR\jwave-3_5\bin
manager ping
```

where `VNI_DIR` is the main Visual Numerics installation directory.

If the test succeeds, the JWAVE Manager returns the following message:

```
PING was successful with SocketConnection(localhost:6500)
```

If the test fails (other message), start the JWAVE Manager as described in *Starting the JWAVE Manager* above.

## *Configuring the JWAVE Manager for HTTP Connections*

This section explains how to set up and test the JWAVE Manager running as a Web Server. This test allows you to ensure that JWAVE is installed properly, and provides a quick introduction to the JWAVE Manager configuration tool.

**Step 1** Start and “ping” the JWAVE Manager, as explained in the previous two sections.

**Step 2** On the same host, but in a separate shell or command window, enter the following command to start the JWAVE Manager configuration tool.

```
manager config
```

**NOTE** When the configuration tool starts, a dialog appears that informs you that you cannot save server connection parameters when the JWAVE Manager is running. Dismiss this dialog box. For this exercise, you will only change client connection parameters.

The first panel you see is the **Manager Properties** panel. For now, you can leave the manager properties settings as they are. The test we are going to run requires a change to the **Client Connection Info** panel.

**Step 3** Click the Client Connection Info button to view the **Client Connection Info** panel.

**Step 4** In the JWave HTTP URL text field, enter the correct URL to the JWAVE server. By default, this URL is:

```
http://myhost:6580/JWave
```

where `myhost` is the name of the host where the JWAVE Manager is running.

**Step 5** Deselect the Use Socket Connection checkbox. We are going to connect to the JWAVE Web server through an HTTP connection.

**Step 6** Select the Use HTTP Connection checkbox.

**Step 7** Click the Save Configuration button, and then select **File=>Exit** to exit the dialog box.

**Step 8** Start a Web browser.

**Step 9** Point the browser to the following URL:

```
http://myhost:6580/JWave
```

The browser displays a test page that says: “The JWAVE HTTP Server is ready to accept JWAVE connections”. This page indicates that you have successfully started the JWAVE Manager as a Web server, and that you were able to reach the Web server from a client using a URL address.

**Step 10** Try pointing your browser to the JWAVE demonstration area and running some of the demonstration JWAVE applets. The URL for the demonstration area is:

```
http://myhost:6580/jwave_demos
```

Click on any of the example JWAVE applets to run them.

## Setting Up the JWAVE Server

The JWAVE server includes the JWAVE Manager software, JWAVE class files, and PV-WAVE. This section describes:

- **Using the JWAVE Manager** — Includes startup command options and stopping the JWAVE Manager.
- **Configuring the JWAVE Manager** — Describes how to remake configuration files in which the JWAVE Manager and connection information for clients and servers is stored.
- **Using the JWAVE Configuration Tool** — Explains the features of the JWAVE Manager Configuration Tool, which allows you to configure server and client connection parameters.
- **JWAVE Server Options** — Describes the methods by which the JWAVE Server can communicate with clients. These methods include direct sockets, CGI, Web server, and servlet connections.
- Installing and configuring JWAVE as a Service on a Windows NT$^{TM}$ server

### Using the JWAVE Manager

The JWAVE Manager controls communication between JWAVE clients and the JWAVE server. JWAVE Manager is a process that runs on the server and listens for client connections. When a connection is made, the JWAVE Manager takes an appropriate action, such as starting a PV-WAVE session.

There are several methods by which client applications can communicate with the JWAVE Manager. These methods are described in *JWAVE Server Options* on page 114.

This section explains how to:

- Use the `manager` command.
- Shut down the JWAVE Manager
- Test the default server configuration

**NOTE** Client applications cannot contact the JWAVE server unless the JWAVE Manager is running.

### JWAVE Manager Startup Command Options

To display the options available for the JWAVE Manager startup command:

**(UNIX)**
```
cd VNI_DIR/jwave-3_5/bin
manager
```

**(Windows)**
```
cd VNI_DIR\jwave-3_5\bin
manager
```

where `VNI_DIR` is the main Visual Numerics installation directory.

The `manager` command takes one argument and, in most cases, an optional parameter. Here is the list of arguments and parameters used with the `manager` command.

***Usage***

`manager` *argument* [*parameter*]

***Arguments and Optional Parameters***

***manager start*** [*path_to_jwave.cfg*] — Starts the JWAVE Manager daemon. Enter the path to the `jwave.cfg` file if it is not located in the same directory as the `manager` command.

***manager ping*** — Tests to see if the JWAVE Manager is running.

***manager config*** [*path_to_jwave.cfg*] — Opens the JWAVE Manager Configuration tool. Enter the path to the `jwave.cfg` file if it is not located in the same directory as the `manager` command.

***manager session_info*** — Reports session information (number of active sessions, unused pool sessions, and persistent sessions).

***manager shutdown*** [*password*] — Shuts down the JWAVE Manager.

### Shutting Down the JWAVE Manager

**Windows USERS** If you started a JWAVE Service through the Service's dialog box, use the Service's dialog box to shut it down instead of using the `manager shutdown` command.

To shut down the JWAVE Manager, enter the following command:

**(UNIX)**
```
cd VNI_DIR/jwave-3_5/bin
manager shutdown [password]
```

**(Windows)**
```
cd VNI_DIR\jwave-3_5\bin
manager shutdown [password]
```

where `VNI_DIR` is the main Visual Numerics installation directory and a password is required if a password was defined during server configuration (see page 123).

The window in which you typed the shutdown command displays the following message:

```
SHUTDOWN was successful.
```

The window in which the JWAVE Manager was started displays messages as well. These messages may vary depending on how the JWAVE Manager is configured. Here is a sample of such messages:

```
JWaveTcpipServer shutting down
JWaveHttpServer shutting down
JWaveManager shutting down
Shutting down all WaveSessions
[additional information on PV-WAVE sessions...]
Wave closed, releasing other resources...
```

---

**TIP** Since `manager shutdown` can be executed from any machine that has client access (even across the Web), it is recommended that a password be set for the JWAVE server. Assignment of a password for `shutdown` will prevent accidental shutdown of the JWAVE Manager by unauthorized users. (The instructions for assigning a password for `shutdown` are given in *Using the JWAVE Configuration Tool* on page 118).

---

## JWAVE Server Options

The JWAVE Manager can be contacted directly through either sockets or HTTP.

With socket connections, the client connects to the JWAVE Manager either: a) through a direct socket connection, or b) through a CGI program that then hands a socket connection to the server.

With HTTP connections, the client connects to the JWAVE Manager either: a) through the JWAVE Server, or b:) through the JWAVE Servlet.

### *The Direct Socket Connection Option*

To configure the JWAVE Manager to communicate with clients through direct sockets, do the following:

- In the Client Connection Info dialog box, select the Use Socket Connection option. Also, in this dialog box, set the Port and Host Name fields.
- In the JWAVE Manager Configuration Properties dialog box, set the variable `MANAGER_START_TCPIP` to `TRUE`.

---

**NOTE** To configure the JWAVE Manager to accept both socket/CGI and HTTP connections, set both `MANAGER_START_TCPIP` and `MANAGER_START_HTTP` to `TRUE`.

---

### The CGI Connection Method

To configure the JWAVE Manager to communicate with clients through a CGI program, do the following:

- In the Client Connection Info dialog box, select the Use CGI Connection option. Also, in this dialog box, set the JWaveCGI URL field.
- In the JWAVE Manager Configuration Properties dialog box, set the variable `MANAGER_START_TCPIP` to `TRUE`.
- Install `VNI_DIR/jwave-3_5/bin/bin.<arch>/JWaveCGI[.exe]` in your Web server's CGI area, where `<arch>` is the architecture of your system (such as `solaris` or `i386nt`), and `[.exe]` is a filename extension found on Windows systems only.

### Using the JWAVE Web Server

To configure the JWAVE Manager to run as a Web Server, do the following:

- In the Client Connection Info dialog box, select the Use HTTP Connection option.
- In the JWAVE Manager Configuration Properties dialog box, set the variable `MANAGER_START_HTTP` to `TRUE`.
- Edit the Web Server configuration file, which contains general settings, directory mappings, and MIME type mappings that you may wish to change. The configuration file is located in:

  ```
  VNI_DIR/jwave-3_5/bin/jwave_http.cfg
  ```

  This file is described in Appendix D, *HTTP Configuration File*.

### Using the JWAVE Servlet

The JWAVE Servlet can be used to replace the JWaveManager program. To configure the JWAVE Manager to run through the JWAVE Servlet, do the following:

- Install `com.visualnumerics.jserver.JWaveServlet` onto your Web server (such as JavaWeb Server).
- Set the `VNI_DIR` parameter for the servlet to point to the Visual Numerics installation directory (`VNI_DIR`).
- In the Client Connection Info dialog box, select Use HTTP Connection.

---

**TIP** Once you have installed the JWAVE Servlet, you can test the URL by pointing your Web browser there. For example, type

```
http://myserver/servlet/JWave
```

in the browser's URL field. The browser will display a message from the servlet saying "JWAVE Servlet is ready". Use this same URL in the client configuration HTTP connection field.

---

**NOTE** Do not use the `manager shutdown` command with the JWAVE Servlet. Instead, use the Web server's servlet administration tools to shut down the server.

---

## Configuring the JWAVE Manager

This section explains how to configure the JWAVE Manager. Although the default configuration is usually adequate, you can modify the defaults if necessary.

### *Configuration Files*

When you installed the JWAVE server, the following three files were created automatically. The first file, `manager`, is the JWAVE Manager startup script, which is discussed in the previous section. The other files are configuration files that contain information required for JWAVE to run properly.

- `VNI_DIR/jwave-3_5/bin/manager[.bat]`

  A script (batch file) that controls the JWAVE Manager.

- `VNI_DIR/jwave-3_5/bin/jwave.cfg`

  A file that contains configuration information for the JWAVE server.

- `VNI_DIR/classes/JwaveConnectInfo.jar`

  A file that describes to the JWAVE clients how to connect to the JWAVE server.

- `VNI_DIR/jwave-3_5/bin/jwave_http.cfg`

  JWAVE Web server configuration. Lets you specify URL-to-directory maps, MIME types, and so on.

### *Remaking the Configuration Files*

If you change the location of either the runtime PV‑WAVE installation (`VNI_DIR`) or the Java Runtime Environment (JRE) (`JAVA_BIN_DIR`), you must do the following:

**Step 1** Shut down JWAVE Manager.

**Step 2** Run the `make_config` script, as follows:

**(UNIX)**

```
cd VNI_DIR/jwave-3_5/bin
manager shutdown
make_config <VNI_DIR> <JAVA_BIN_DIR>
```

**(Windows)**

```
cd VNI_DIR\jwave-3_5\bin
manager shutdown
make_config <VNI_DIR> <JAVA_BIN_DIR>
```

where `VNI_DIR` is the main Visual Numerics installation directory and `JAVA_BIN_DIR` is the JRE directory.

The `make_config` script:

- makes backup copies of the `manager[.bat]` and `jwave.cfg` files
- creates a new `manager[.bat]` script that contains only the `VNI_DIR` and `JAVA_BIN_DIR` property definitions
- resets all other JWAVE configuration properties in the `jwave.cfg` to their built-in default values

If you had customized the properties in the old `jwave.cfg` file, you may copy those definitions from the `jwave_cfg.bak` file into the new `jwave.cfg` file, or use the JWAVE Configuration Tool to modify the new `jwave.cfg` file (described in *Using the JWAVE Configuration Tool* on page 118).

**Step 3** Restart the JWAVE manager:

**(UNIX)**

```
cd VNI_DIR/jwave-3_5/bin
manager start
```

**(Windows)**

```
cd VNI_DIR\jwave-3_5\bin
manager start
```

where `VNI_DIR` is the main Visual Numerics installation directory.

**Windows USERS** We recommend that you start and stop the JWAVE Manager using the JWAVE Service (page 109).

## Using the JWAVE Configuration Tool

The JWAVE Configuration Tool has two windows:

- Manager Properties (described on page 118)
- Client Connection Info (described on page 125)

General help for the JWAVE Configuration Tool is available from the Help menu.

### *Starting the JWAVE Configuration Tool*

To open the JWAVE Configuration Tool on the JWAVE server, shut down the JWAVE Manager as explained in *Shutting Down the JWAVE Manager* on page 113, then start the tool:

**(UNIX)**

```
cd VNI_DIR/jwave-3_5/bin
manager shutdown
manager config
```

**(Windows)**

```
cd VNI_DIR\jwave-3_5\bin
manager shutdown
manager config
```

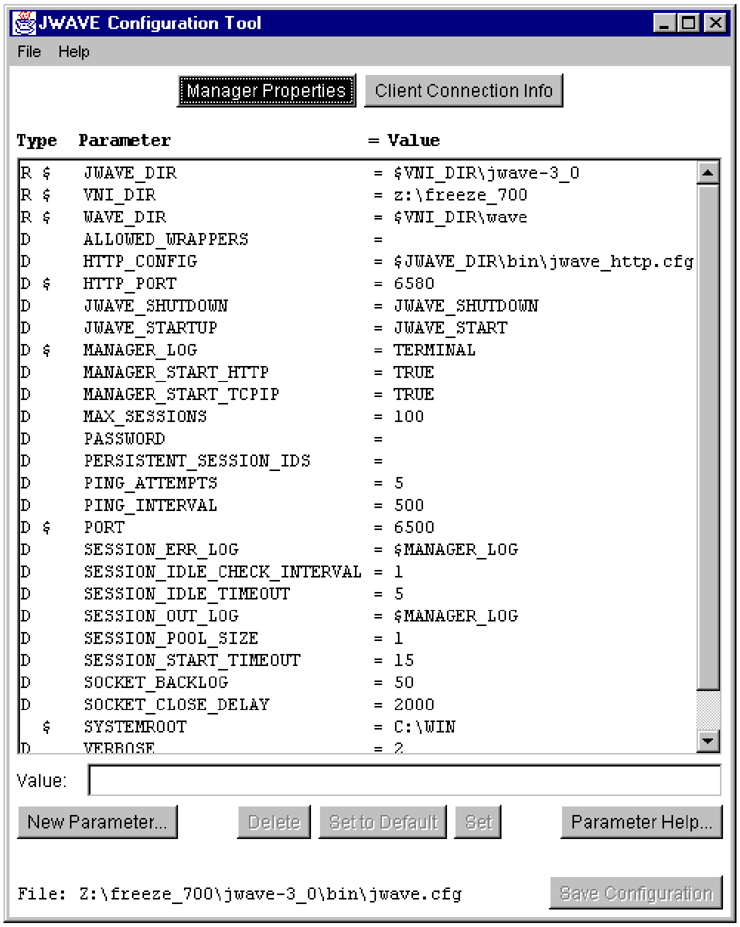
where `VNI_DIR` is the main Visual Numerics installation directory.

### *Manager Properties*

Your JWAVE server's current configuration is displayed in the Manager Properties list.

Use the Manager Properties window in the JWAVE Configuration Tool (*Figure 8-3*) to:

- change the defaults of JWAVE server configuration properties
- set up definitions for the log file for JWAVE Manager and PV‑WAVE sessions
- define custom properties for setting environment variables on the JWAVE server



```
Type  Parameter                       = Value
R $   JWAVE_DIR                       = $VNI_DIR\jwave-3_0
R $   VNI_DIR                         = z:\freeze_700
R $   WAVE_DIR                        = $VNI_DIR\wave
D     ALLOWED_WRAPPERS                =
D     HTTP_CONFIG                     = $JWAVE_DIR\bin\jwave_http.cfg
D $   HTTP_PORT                       = 6580
D     JWAVE_SHUTDOWN                  = JWAVE_SHUTDOWN
D     JWAVE_STARTUP                   = JWAVE_START
D $   MANAGER_LOG                     = TERMINAL
D     MANAGER_START_HTTP              = TRUE
D     MANAGER_START_TCPIP             = TRUE
D     MAX_SESSIONS                    = 100
D     PASSWORD                        =
D     PERSISTENT_SESSION_IDS          =
D     PING_ATTEMPTS                   = 5
D     PING_INTERVAL                   = 500
D $   PORT                            = 6500
D     SESSION_ERR_LOG                 = $MANAGER_LOG
D     SESSION_IDLE_CHECK_INTERVAL     = 1
D     SESSION_IDLE_TIMEOUT            = 5
D     SESSION_OUT_LOG                 = $MANAGER_LOG
D     SESSION_POOL_SIZE               = 1
D     SESSION_START_TIMEOUT           = 15
D     SOCKET_BACKLOG                  = 50
D     SOCKET_CLOSE_DELAY              = 2000
  $   SYSTEMROOT                      = C:\WIN
D     VERBOSE                         = 2

File: Z:\freeze_700\jwave-3_0\bin\jwave.cfg
```

**Figure 8-3** Manager Properties in the JWAVE Configuration Tool

---

**TIP** To get help for a specific parameter, select the parameter, then click Parameter Help. You can also refer to *JWAVE Manager Configuration Properties* on page 122 for an description of each property.

---

**Type Column** — This column may contain any of the following codes that describe the parameter's type:

- R — Read Only. Parameters with this type are displayed for information purposes only and their values cannot be changed in the JWAVE Configuration Tool.

- D — Indicates that the parameter's current value is the default value.
- U — Indicates a parameter defined by a user for passing environment variables to PV-WAVE sessions. This type of parameter is not used by JWAVE Manager.
- $ — Indicates that the parameter can be referenced by name in other parameters. More information on this type of parameter is available in Parameter Help...

**Parameter Column** — This column lists the names of the parameters currently defined.

**= Value Column** — This column lists the values of the parameters currently defined. These parameters are described in *JWAVE Manager Configuration Properties* on page 122.

**Value Field** — This field allows you to enter a value for any parameter that does not have the R type in the Type column.

**New Parameter...** — Displays the Define New Parameter dialog box for adding a new parameter (see *Adding Properties* on page 122).

**Delete** — Removes the selected user-defined parameter from the list of parameters.

**Set to Default** — Sets the value of the selected parameter to its default value.

**Set** — Sets the value of the selected parameter to the value entered in the Value field. (Pressing the Enter key in the Value field is equivalent to clicking Set.)

**Parameter Help...** — Displays the Help on Parameters dialog, which contains information about the selected parameter.

**Save Configuration** — Saves the configuration to the `jwave.cfg` file (path shown to the left of the button).

If the JWAVE Manager is running when you try to save the configuration by clicking Save Configuration, a message displays reminding you to shut down JWAVE Manager. Shut down the Manager and then click Save Configuration again. (See *Shutting Down the JWAVE Manager* on page 113.)

### *Modifying a Property*

To change a configuration property:

**Step 1** Shut down the JWAVE Manager.

**Step 2** Start the JWAVE Configuration Tool (described on page 118).

**Step 3** If necessary, click Manager Properties to display the list of configuration parameters.

**Step 4** Click the parameter you wish to change.

**Step 5** Type the new value for the property in the Value field.

**Step 6** Click Set or press Enter. The new value for the parameter appears in the = Value list.

**Step 7** Choose Save Configuration to save your changes. The changes are saved in the `jwave.cfg` file.

### *Resetting a Single Property to Its Default*

To reset the default value of a previously modified property:

**Step 1** Shut down the JWAVE Manager.

**Step 2** Start the JWAVE Configuration Tool (described on page 118).

**Step 3** If necessary, click Manager Properties to display the list of configuration parameters.

**Step 4** Click the parameter you wish to set to its default value.

**Step 5** Click Set to Default. A letter D appears next to the parameter in the Type list. The default value for the parameter appears in the = Value list

**Step 6** Choose Save Configuration to save your changes. The changes are saved in the `jwave.cfg` file.

### *Adding Properties*

To add a new property, such as an environment variable, to the `jwave.cfg` file:

**Step 1** Shut down the JWAVE Manager.

**Step 2** Start the JWAVE Configuration Tool (described on page 118).

**Step 3** If necessary, click Manager Properties to display the list of configuration parameters.

**Step 4** Click New Parameter. The Define New Parameter dialog box appears.

**Step 5** Type the name for the new parameter in the Name field. For example: `mydata`

**Step 6** Type the new parameter's definition in the Value field. For example: `/user/data/mydata`

**Step 7** Click Set. The new parameter appears in the Parameter list of the Manager Properties window. U appears next to the parameter in the Type list.

**Step 8** Click Save Configuration to save your changes. The changes are saved in the `jwave.cfg` file.

### *JWAVE Manager Configuration Properties*

The predefined JWAVE Manager properties that you can set or modify in the JWAVE Configuration Tool are described in this section.

---

**NOTE** Some properties (usually directories) can be referenced by name in other properties. For example, the default for the property `WRAPPER_PATH` is `$JWAVE_DIR/lib/user` and the property `JWAVE_DIR` is an "expandable" property. Such properties are marked with a **`$`** in the descriptions that follow.

---

**`ALLOWED_WRAPPERS`** — For security, you can specify a list of regular expressions to limit the wrapper functions that the PV-WAVE session will execute. Leave this field blank (empty string) to allow any valid JWAVE function. Separate multiple expressions with a comma (,). (Default: [`null`])

**`HTTP_CONFIG`** — Specifies the HTTP configuration file. (Default: `VNI_DIR/jwave/bin/jwave_http.cfg`)

**`HTTP_PORT $`** — Specifies the port number used by the JWAVE Web server. (Default: 6580).

**NOTE** The normal default for Web servers is port 80; however, you must be an administrator or super user to start a server on port numbers less than 1024.

**JWAVE_DIR $** — Read-only. Specifies the directory where JWAVE is installed. (Default: `VNI_DIR/jwave-3_5`)

**JWAVE_SHUTDOWN** — Specifies the shutdown procedure for the PV‑WAVE session. You can use this procedure to do site-specific shutdown after every PV‑WAVE session. (Default: `JWAVE_SHUTDOWN`)

**JWAVE_STARTUP** — Specifies the startup procedure for the PV‑WAVE session. You can use this procedure to do site-specific initialization for every PV‑WAVE session. (Default: `JWAVE_START`)

**MANAGER_LOG $** — Specifies the JWAVE Manager log file location, filename, and file continuation instructions. Leave this field blank (empty string) to discard logging. Use `TERMINAL` to have the logs go to the JWAVE Manager's terminal (`stdout`). Prefix the file with a “+” to append to the log when the manager is restarted (otherwise a new file will be created). (Default: `TERMINAL`)

**MANAGER_START_HTTP** — If set to `TRUE`, the command:

```
manager start
```

configures the JWAVE Manager to run the JWAVE Web Server. Clients can connect to the server using an HTTP connection.

**MANAGER_START_TCPIP** — If set to `TRUE`, the command:

```
manager start
```

configures the JWAVE Manager to accept direct socket connections.

**MAX_SESSIONS** — Specifies the maximum number of simultaneous PV‑WAVE sessions the JWAVE Manager will allow. Note that the session limit is also controlled by licensing, but this parameter may be useful for server performance tuning. (Default: 100)

**PASSWORD** — Specifies the server password for remote access to configuration information and JWAVE Manager shutdown. (Default: [`null`])

**PERSISTENT_SESSION_IDS** — Specifies a list of PV‑WAVE session ID numbers (positive numbers). The specified sessions do not time out as do regular sessions.

**PING_ATTEMPTS** — After a PV‑WAVE session is started, the JWAVE Manager attempts to contact (ping) it. This property specifies the number of ping attempts that will be made before the session is considered dead (and an error exception is returned to the client). (Default: 5)

**`PING_INTERVAL`** — Specifies the delay (in milliseconds) between `PING_ATTEMPTS`. (Default: 500)

**`PORT $`** — Specifies the socket port number where the JWAVE Manager listens for client connections. Usually just a number, but may also be specified as *host-name:port_number* if you have multiple network addresses and want the JWAVE Manager to listen on only one. (Default: 6500)

---

**NOTE** The port number must be ≥ 1024 for JWAVE Manager to be contacted via CGI (the Web).

---

**`SESSION_ERR_LOG`** — Specifies the log file location, filename, and file continuation instructions for ERROR output from individual PV-WAVE sessions. A “`#`” character in the parameter will be replaced by a session ID number. If there is no “`#`”, all sessions log to the same file. Leave this field blank (empty string) to discard logging. Use `TERMINAL` to have the logs go to the JWAVE Manager's terminal (`stdout`). (Default: `$MANAGER_LOG`)

**`SESSION_IDLE_CHECK_INTERVAL`** — Specifies how often (in minutes) to check for idle (unused) PV-WAVE sessions. (Default: 1)

**`SESSION_IDLE_TIMEOUT`** — Specifies how long (in minutes) idle (unused) PV-WAVE sessions will remain alive. After this time, idle sessions are closed. If a session is timed out, it is no longer available to its client. (Default: 5)

**`SESSION_OUT_LOG`** — Specifies the log file location, filename, and file continuation instructions for output from individual PV-WAVE sessions. A “`#`” character in the parameter will be replaced by a session ID number. If there is no “`#`”, all sessions log to the same file. Leave this field blank (empty string) to discard logging. Use `TERMINAL` to have the logs go to the JWAVE Manager's terminal (`stdout`). (Default: `$MANAGER_LOG`)

**`SESSION_POOL_SIZE`** — Sets the number of PV-WAVE sessions to pre-start. If there is a PV-WAVE session in the pool, a client’s first contact becomes faster because it will not have to wait for the PV-WAVE process to start up. These are only used for auto-assigned sessions.

**`SESSION_START_TIMEOUT`** — Specifies the maximum time (in seconds) to wait for a PV-WAVE session to start. Sessions that take longer will be killed and an error exception will be returned to the client. You may wish to increase this value on slow or heavily loaded servers. (Default: 15)

**`SOCKET_BACKLOG`** — Specifies the maximum queue length (number of concurrent connections) for incoming connection requests (used by the ServerSocket class). If a connection request arrives when the queue is full, the connection is refused. (Default: 50)

**`SYSTEMROOT $`** — Windows NT only. Required for servers running on Windows NT. Set this property to your Windows NT directory. (Default: `C:\Windows` or the value of `SYSTEMROOT` at install time)

**`VERBOSE`** — Specifies logging level. Valid values are 0 (silent) to 3 (verbose). (Default: 2)

**`VNI_DIR $`** — Read-only. Required. The value is set by the `manager[.bat]` script.

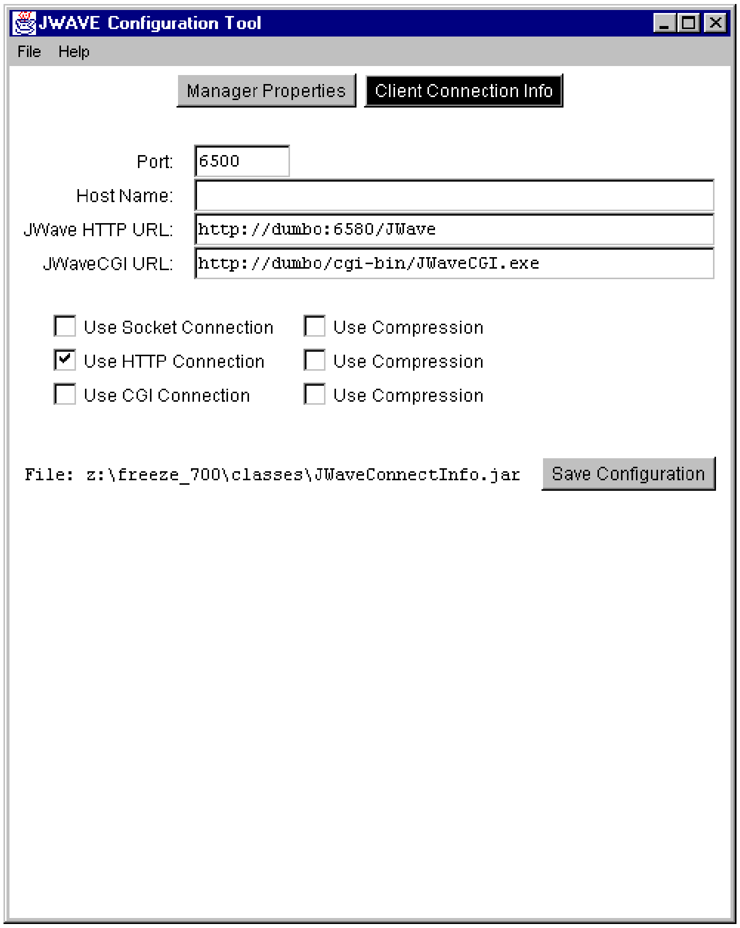
**`WAVE_DIR $`** — Read-only. Specifies the directory where PV‑WAVE is installed. (Default: `$VNI_DIR/wave`)

**`WRAPPER_PATH`** — Specifies directories that contain your custom JWAVE wrapper functions (`.cpr` files). You can separate multiple directories on Windows NT with a `;` (semi-colon) character, on UNIX with a : (colon) character. The standard PV‑WAVE and JWAVE `lib` directories are automatically included. (Default: `$JWAVE_DIR/lib/user`)

### *Client Connection Info*

You can use the Client Connection Info window in the JWAVE Configuration Tool (*Figure 8-4*) to:

- define the port number where the JWAVE Manager is running
- define the host name of the machine where JWAVE Manager is running
- define the URL (as seen through your Web server) of the JWaveCGI executable
- define a URL to the JWAVE Servlet or the JWAVE Web Server
- select the means by which clients contact JWAVE Manager (direct Socket, JWaveCGI URL, or HTTP connection URL)



Port: 6500
Host Name:
JWave HTTP URL: http://dumbo:6580/JWave
JWaveCGI URL: http://dumbo/cgi-bin/JWaveCGI.exe

☐ Use Socket Connection ☐ Use Compression
☑ Use HTTP Connection ☐ Use Compression
☐ Use CGI Connection ☐ Use Compression

File: z:\freeze_700\classes\JWaveConnectInfo.jar

**Figure 8-4** Client Connection Info in the JWAVE Configuration Tool

The following rules apply to the fields in the Client Connection Info window of the JWAVE Configuration Tool:

✔ You must fill in either Port or Host Name (it is preferred that both fields be filled in).

✔ You must select Use Socket Connection, Use HTTP Connection, Use CGI Connection (or any combination of the three).

✔ If Use CGI Connection is selected, JWaveCGI URL must be filled in.

✔ If Use HTTP Connection is selected, Jwave HTTP URL must be filled in.

**Port** — Specifies the port number where the JWAVE Manager is running. (See also the `PORT` parameter in the Manager Properties list of the JWAVE Configuration Tool and its description in *JWAVE Manager Configuration Properties* on page 122).

**Host Name** — Specifies the name of the machine where the JWAVE Manager is running. If this field is blank or set to `localhost`, clients will only be able to connect (via sockets) when running on the same machine where the JWAVE Manager is running.

**JWave HTTP URL** — Specifies the URL to the Web server or JWAVE Servlet that is associated with the JWAVE Manager. It can be an `http` or `https` protocol URL.

**JWaveCGI URL** — Specifies the URL to the CGI program that is associated with the JWAVE Manager. It can be an `http` or `https` protocol URL. If you want clients using CGI to contact a different JWAVE Manager than the one defined in the Port and Host Name fields, append a question mark and that information to the end of the URL. For example:

```
http://webhost/cgi-bin/JWaveCGI?manager_host:6501
```

(Default: `http://<SERVER-NAME>/cgi-bin/JWaveCGI[.exe]`)

---

**NOTE** The https protocol uses the Secure Socket Layer (SSL) to connect to the server. If you specify `https`, both the client and server must support `https`. Most browsers and Web servers do support `https`; however, the Java Development Kit (JDK) does not. Therefore, if you save the client configuration with an `https` protocol, you will always receive a message that says “unable to test https”.

---

**Use Socket Connection** — When checked, specifies that clients will be able to contact the JWAVE Manager using direct Socket connections.

**Use HTTP Connection** — When checked, specifies that clients will be able to contact the JWAVE Manager as a Web server or servlet.

**Use CGI Connection** — When checked, specifies that clients will be able to contact the JWAVE Manager using the JWaveCGI URL.

**Use Compression** — When checked, specifies that the data stream be compressed. Use compression if you intend to send very large datasets. Compression saves network time in transmission of the data, but costs some CPU time for compression and decompression on the client and server.

**Save Configuration** — Clicking this button saves the configuration to the `JWaveConnectInfo.jar` file (path shown to the left of the button).

**TIP** The client checks the selections Use Socket Connection, Use HTTP Connection, and Use CGI Connection in the order they are listed: 1) Socket, 2) HTTP, 3) CGI. Normally, we recommend that you only choose one method. The client-side getConnection method is optimized to be faster for the initial connection if only one method is checked.

If you wish to establish a secondary or "fallback" connection, check more than one connection type. For example, if you want clients to normally connect through HTTP protocol, but you want to use CGI if the HTTP connection is inoperative, then check both the HTTP and CGI connection methods.

If you wish to establish a connection order other than the one that is available in the dialog box (Socket, HTTP, CGI), then you need to edit the file:

```
VNI_DIR/classes/JWaveConnectInfo.jar
```

For information on editing this file, refer to the following file:

```
      VNI_DIR/classes/JWaveConnectInfo.txt
```

## Installing the JWAVE Manager as a Service on Windows NT

The JWAVE Manager can be installed and run as a Service on a Windows NT server. Installing JWAVE Manager as a Windows NT Service allows you to:

- run the JWAVE Manager as a background process
- keep the JWAVE Manager running when there are no interactively logged-in users
- shut down the JWAVE Manager by stopping the JWAVE Service

### *Installing the JWAVE Service*

The JWAVE Service executable is installed with the JWAVE Server installation. To install the JWAVE Service on a Windows NT server:

**Step 1** Log in with Administrator or Domain Administrator privilege levels.

**Step 2** Type:

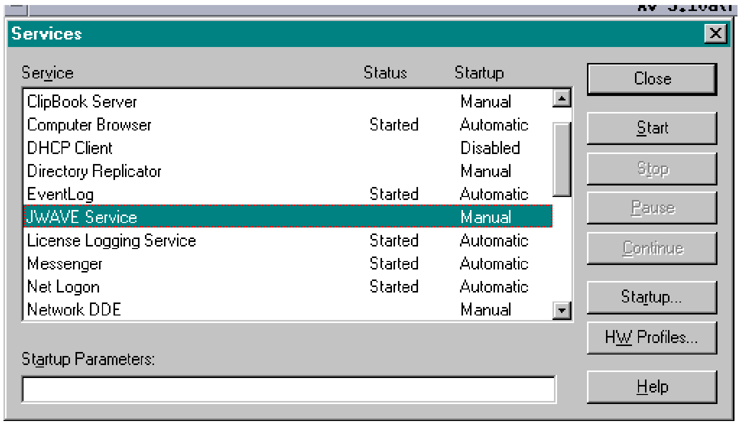
```
VNI_DIR\jwave-3_5\bin\bin.i386nt\jwaveservice -install
```

where `VNI_DIR` is the main Visual Numerics installation directory.

## *Configuring the JWAVE Service*

To configure the JWAVE Service:

**Step 1** In the Control Panel, double-click the Services icon. The Services window appears (see *Figure 8-5*).



Services

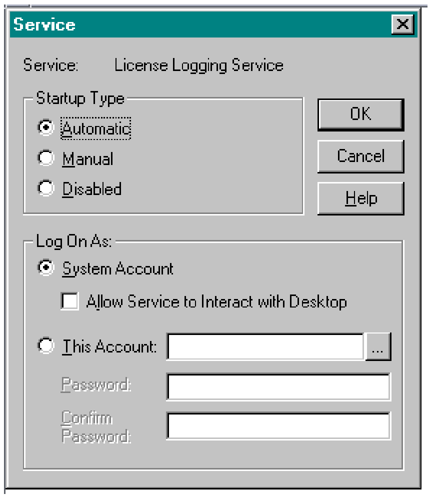
| Service | Status | Startup |
|---|---|---|
| ClipBook Server | | Manual |
| Computer Browser | Started | Automatic |
| DHCP Client | | Disabled |
| Directory Replicator | | Manual |
| EventLog | Started | Automatic |
| JWAVE Service | | Manual |
| License Logging Service | Started | Automatic |
| Messenger | Started | Automatic |
| Net Logon | Started | Automatic |
| Network DDE | | Manual |

Startup Parameters:

**Figure 8-5** Windows NT Services window listing the JWAVE Service

**Step 2** Select JWAVE Service.

**Step 3** Click Startup... The Service window appears (see *Figure 8-6*).



Service

Service: License Logging Service

Startup Type

- (●) Automatic
- ( ) Manual
- ( ) Disabled

Log On As:

- (●) System Account
  - [ ] Allow Service to Interact with Desktop
- ( ) This Account:

Password:

Confirm Password:

**Figure 8-6** Windows NT Service window for configuring the JWAVE Service

**Step 4** In the Service window, set the Startup Type to Automatic. This causes the JWAVE Manager to run as a background process and keep running even when there are no interactively logged-in users.

**Step 5** By default, the JWAVE Service runs under the Local System account which has no access to network resources. If you need the account that runs the JWAVE Service to have access to network resources (for example, if your JDK is installed on a network drive), change the default user to a user with more privileges.

**Step 6** Click OK in the Service window to close it and accept your changes.

**Step 7** Click Start in the Services window to start up the JWAVE Manager in the background.

### *Stopping the JWAVE Service*

If you started JWAVE Manager as a Service, it is recommended you use the Services window to stop the JWAVE Service. When the JWAVE Service is stopped, it will issue a shutdown command using the password from the `jwave.cfg` file, if a password has been defined.

To shut down JWAVE Manager running as a Service:

**Step 1** In the Control Panel, double-click the Services icon. The Services window appears (see *Figure 8-5*).

**Step 2** Select JWAVE Service.

**Step 3** Click Stop.

**Step 4** Click Close.

### *Monitoring the JWAVE Service*

The JWAVE Service can be monitored remotely using the Server Manager (`SrvMgr.exe`) supplied with both the Windows NT Server and the Windows NT Workstation Resource Kit.

### *Starting and Stopping the JWAVE Service from the Command Line*

The JWAVE Service can be started and stopped from the command line using the `net` command after installing and configuring the Service as described above.

To start the JWAVE Service from the command line:

```
net start jwaveservice
```

To stop the JWAVE Service from the command line:

```
net stop jwaveservice
```

### *Removing the JWAVE Service*

The JWAVE Service can be removed from a Windows NT server when you need to install a JWAVE upgrade or move the Service to another server.

To remove the JWAVE Service from your Windows NT server:

**Step 1** Log in with the same privileges used to install the Service on the JWAVE server, as explained in *Installing the JWAVE Service* on page 128.

**Step 2** Stop the Service.

**Step 3** Type:

```
VNI_DIR\jwave-3_5\bin\bin.i386nt\jwaveservice -remove
```

where `VNI_DIR` is the main Visual Numerics installation directory.

## *Testing the JWAVE Server Installation*

The following directory contains Java programs you can run to test the JWAVE server installation:

**(UNIX)** `VNI_DIR/classes/jwave_demos/tests`

**(Windows)** `VNI_DIR\classes\jwave_demos\tests`

where `VNI_DIR` is the main Visual Numerics installation directory.

To run the following tests:

- the JWAVE Manager must be running (see page 109)
- the following must be in the `CLASSPATH`:

**(UNIX)**

```
VNI_DIR/classes/JWaveConnectInfo.jar
VNI_DIR/JWave.jar
VNI_DIR/classes/jwave_demos/tests
```

**(Windows)**

```
VNI_DIR\classes\JWaveConnectInfo.jar
VNI_DIR\JWave.jar
VNI_DIR\classes\jwave_demos\tests
```

where `VNI_DIR` is the main Visual Numerics installation directory.

## Scalar Data Test

`ScalarDataTest` sends scalar data to PV-WAVE and back.

## Array Data Test

`ScalarArrayTest` sends array data to PV-WAVE and back.

## Return Mode Test

`ReturnModeTest` sends scalar data to PV-WAVE and asks that some of it be stored in the Data Manager, then calls PV-WAVE again to use the stored data.

## View Test

`ViewTest` tests the JWaveView class. JWaveView displays a window with a view area and Plot, Surface, and Close Session buttons.

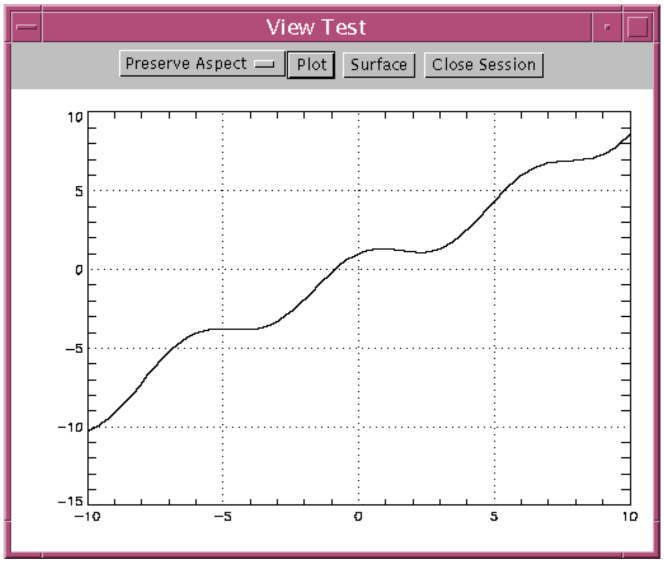


**Figure 8-7** The View Test window showing the 2D plot.

Try the following in the View Test window (*Figure 8-7*):

- ✔ Click Plot — a PV-WAVE session starts and a 2D plot displays.
- ✔ Click Surface — The same PV-WAVE session makes a surface and displays it.
- ✔ Click Plot again — The 2D plot displays again.
- ✔ Click on the plot to see data coordinates printed on the plot.
- ✔ Experiment with the Resize modes.
- ✔ Click Close Session — This should close the PV-WAVE session on the JWAVE server.

## Running JWAVE Demonstrations

If you want to run the JWAVE demonstrations, refer to the `README` files in the subdirectories of:

**(UNIX)** `VNI_DIR/classes/jwave_demos`

**(Windows)** `VNI_DIR\classes\jwave_demos`

where `VNI_DIR` is the main Visual Numerics installation directory.

## Setting Up for JWAVE Client Development

The only thing you need to set for JWAVE client development is the `CLASSPATH` environment variable. Set the `CLASSPATH` on a JWAVE client to include (at least):

**(UNIX)** `VNI_DIR/classes/JWave.jar`

**(Windows)** `VNI_DIR\classes\JWave.jar`

where `VNI_DIR` is the main Visual Numerics installation directory.

The JWAVE Client documentation is referenced with:

**(UNIX)** `VNI_DIR/classes/docs/api/packages.html`

**(Windows)** `VNI_DIR\classes\docs\api\packages.html`

Example client side development code is located in subdirectories of:

**(UNIX)** `VNI_DIR/classes/jwave_demos`

**(Windows)** `VNI_DIR\classes\jwave_demos`

If you want to use JWAVE Beans, you need to:

- Add Swing 1.1 or later to the `CLASSPATH` (`swing.jar` or `swingall.jar`).
- Import the `JWaveBeans.jar` into BDK 1.0_mar98 or an IDE like JavaStudio.
- To run client applets, you need a browser or appletviewer with access to the JWAVE Manager (usually through a Web server). Make sure that the `CLASSPATH` does not include any JWAVE JAR files or classes. (JWAVE applets can only communicate with the JWAVE Manager if the applets are served by that server—having these classes in `CLASSPATH` causes a security error.)

---

**NOTE** You must have access to the JWAVE server for JWAVE client applications to work properly.

---

- To run client applications (in other words, applications outside of a browser), you also must have `JWaveConnectInfo.jar` in your `CLASSPATH`.
- See notes in the `README` file in the subdirectories of `VNI_DIR/classes/jwave_demos` for other hints on setting up your environment.

9

CHAPTER

# Advanced Graphics Features

JWavePanel2D, JWavePanel3D, JWaveCanvas2D, and JWaveCanvas3D extend JWavePanel's and JWaveCanvas' capability of providing a canvas to display graphics so that you can interact with the graphics on the canvas. Here are the interactions allowed:

- Zoom in or zoom out of a chart
- Row or column profiling across an image
- Select a point on a chart
- Interactively rotate a 3D chart

This chapter explains how to use each of the features and how to run example programs that demonstrate the features.

## Advanced Features

JWaveCanvas2D and JWaveCanvas3D, derived from JWaveCanvas, provide additional functionality. Additionally, there are also the classes JWavePanel2D and JWavePanel3D, derived from JWavePanel, which provide equivalent functionality.

## JWaveCanvas2D and JWavePanel2D

The classes JWaveCanvas2D and JWavePanel2D provide the ability to interact with a two-dimensional chart in several ways:

- Pressing and dragging the mouse to zoom in on a chart.
- Selecting a point on a chart.
- Selecting a profile of an image.

## Pick Point

The pick feature allows an object to be notified when the user selects a point on a chart with a mouse click. The `JWaveCanvas2D` and `JWavePanel2D` parent classes send a notification event to all registered listeners that a `PickPointEvent` has occurred. When the listener receives the notification event, the data coordinates of the selected point may be retrieved from the `getSelectedX` and `getSelectedY` methods. The returned values may be displayed or used as indexes to the data arrays to select actual values.

To use this feature with a chart object, pick point must first be enabled. This can be done either by calling the object's

```
setPickPointEnabled(true)
```

method or by sending the object an action with the command string "pick-enabled". Listeners can be added to the object by calling the object's

```
addPickPointListener(PickPointListener listener)
```

method. The listener must implement the nested interface `JWaveCanvas2D.PickPointListener` or `JWavePanel2D.PickPointListener`. The interface requires the implementation of the method

```
public void pointPicked(PickPointEvent event)
```

### *Example*

Assume that we have a chart object that extends JWaveCanvas2D and we want to update a label object with the user-selected coordinates. This could be implemented by adding the following code fragment to the class that sets up the GUI.

```
    final Label label = new Label();
    chart.setPickPointEnabled(true);
    chart.addPickPointListener(new
                                JWaveCanvas2D.PickPointListener() {
        public void pointPicked(JWaveCanvas2D.PickPointEvent event) {
                    int x = event.getSelectedX();
                    int y = event.getSelectedY();
                    label.setText("("+x+","+y"+)");
        }
    }
}
```

## Zoom

The zoom feature allows the user to zoom into a section of a chart. The `JWaveCanvas2D` and `JWavePanel2D` parent classes draw a box indicating the area where the user has pressed and dragged the left mouse button across the chart. When the left mouse has been released all registered listeners will be notified that a `zoomEvent` has occurred. When the listener receives the notification event, the new X and Y ranges may be retrieved from the `getXrange` and `getYrange` methods. The returned values may be used as XRANGE and YRANGE settings for PV-WAVE commands such as PLOT, AXIS, CONTOUR, and SURFACE. For a list of additional PV-WAVE commands, see *PV*-WAVE *Reference Guide, Chapter 3, Graphics and Plotting Keywords*.

To use this feature with a chart object, zoom must first be enabled. This can be done either by calling the object's

```
setZoomEnabled(true)
```

method or by sending the object an action with the command string "zoom-enabled". Listeners can be added to the object by calling the object's

```
addZoomListener(ZoomListener listener)
```

method. The listener must implement the nested interface `JWaveCanvas2D.ZoomListener` or `JWavePanel2D.ZoomListener`. The interface requires the implementation of the method

```
public void zoomUpdate(ZoomEvent event)
```

### *Example*

Assume that we want to add the zoom feature to a chart. This chart is generated by the JWAVE wrapper WImage. The WImage wrapper looks for parameters `xrange` and `yrange`, each of which is an array containing two doubles. These parameters specify the area in which the data is to be charted.

The following class fragment shows the implementation of the zoom feature. The code to initially get the data and generate the chart is omitted for simplicity.

```
public class ZoomChart extends JWavePanel2D implements
                          JWavePanel2D.ZoomListener {
       private JWaveConnection connection;
       private JWaveView  imageView;
       private Viewable      imageViewable;
       private double          xrange[];
       private double          yrange[];

       public ZoomChart(JWaveConnection connection) {
           this.connection = connection;
```

```
            imageView = new JWaveView(connection, "WImage");
            setPreferredSize(new Dimension(500,350));
            addZoomListener(this);
        }

        public void zoomUpdate(JWavePanel2D.ZoomEvent event) {
            xrange = event.getXrange();
            yrange = event.getYrange();
            imageView.setParam("xrange", xrange);
            imageView.setParam("yrange", yrange);
            // Set the size of the plot to be produced to match our
    Panel
            imageView.setViewSize(new Dimension(500,350));
            try {
            imageView.execute();
            imageViewable = imageView.getViewable();
            imageViewable.setPreferredResizeMode
                           (Viewable.NOT_RESIZEABLE);
                setViewable(imageViewable);
            } catch (JWaveException e) {
                System.out.println("Problem getting new
                           Viewable: " + e);
            }
        }
    }
```

## Profile

The profile feature allows the user to select a vertical or horizontal profile of an image by selecting a point on the chart. The `JWaveCanvas2D` and `JWavePanel2D` parent classes draw a line representing the row or column of the image selected. The `JWaveCanvas2D` and `JWavePanel2D` parent classes will send a notification event to all registered listeners that a `ProfileEvent` has occurred. When the listener receives the notification event, the data coordinates of the selected point may be retrieved from the `getSelectedX` and `getSelectedY` methods. The returned values may be used to extract the values of the image along the line selected. For an example, see `ProfileChart.java` provided in the `jwave_demos/canvas2d` directory.

To use this feature with a chart object that displays the image, profile must first be enabled. This can be done either by calling the object's

```
setProfileEnabled(true)
```

method or by sending the object an action with the command string "profile-enabled". Listeners can be added to the object by calling the object's

```
addProfileListener(ProfileListener listener)
```

method. The listener must implement the nested interface `JWaveCanvas2D.ProfiletListener` or `JWavePanel2D.ProfileListener`. The interface requires the implementation of the method

```
public void profilePicked(ProfileEvent event)
```

### Example

Assume that we have a chart object that extends JWaveCanvas2D and we want to plot in a separate chart object the profile of an image and allow the user to select the profile to be displayed. This could be implemented by adding the following code fragment to the class which sets up the GUI for the chart object that displays the image:

```
    chart.setProfileEnabled(true);
    chart.setProfileType(chart.COLUMN);
```

The setProfileType can be used to draw the vertical line representing the column or the horizontal line representing the row.

Next add the ProfileListener to the chart object that will display the line plot or profile plot:

```
chart.addProfileListener(new JWaveCanvas2D.ProfileListener() {
    public void profilePicked(JWaveCanvas2D.ProfileEvent event) {
    int x = event.getSelectedX();
    int y = event.getSelectedY();
    Graphics pg = this.getGraphics();
    drawProfile(pg);

    }

}
```

The drawProfile code has been left out for simplicity. The ProfileChart class provided in the `jwave_demos/canvas2d` directory can easily be used as it is or modified to better meet your needs.

## JWaveCanvas3D and JWavePanel3D

The classes JWaveCanvas3D and JWavePanel3D provide the ability to interactively rotate a 3D chart. Pressing and dragging the mouse pointer changes the view of the chart.

### Rotate

The rotation feature allows an object to be notified when the user has pressed and dragged the mouse on a chart. A wire frame object is drawn in place of the image while the user drags the mouse. When the mouse is released a notification event is sent to the listener, the listener can retrieve the new x and z degrees of rotation from the `getDegX` and `getDegZ` methods. The returned values can be used as AX and AZ keywords settings for PV-WAVE commands such as SURFACE, SHADE_SURFACE, BAR3D or SHOW3.

To use this feature with a chart object, it must first be enabled. This can be done either by calling the object's

```
setRotateEnabled(true)
```

method or by sending the object an action with the command string "rotate-on". Listeners can be added to the object by calling the object's

```
addRotateListener(RotateListener listener)
```

method. The listener must implement the nested interface `JWaveCanvas3D.RotateListener` or `JWavePanel3D.RotateListener`. The interface requires the implementation of the method

```
public void rotateUpdate(RotateEvent event)
```

#### *Example*

Assume that we have a chart object that extends JWaveCanvas3D and we want to allow users to change the view of the 3D chart. This could be implemented by adding the following code fragment to the class that sets up the GUI:

```
ShadeSurfChart chart = new ShadeSurfChart();
   chart.setRotateEnabled(true);
```

The class ShadeSurfChart extends JWaveCanvas3D and can implement rotation by adding the following code fragment to ShadeSurfChart.

```
public class ShadeSurfChart extends JWaveCanvas3D
                         implements JWaveCanvas3D.RotateListener {
```

```
public void rotateUpdate(JWaveCanvas3D.RotateEvent event) {
        int ax = event.getDegX();
        int az = event.getDegZ();
        jwaveWrapper.setParam("ax",ax);
        jwaveWrapper.setParam("az",az)'
    try {
        jwaveWrapper.execute();
    viewable = jwaveWrapper.getViewable();
    setViewable(viewable);
    } catch  (JWaveException e) {
        System.out.println(e);
                    }
        }
}
```

The remaining ShadeSurfChart code has been left out for simplicity. For more details, see the ShadeSurfChart class in `jwave_demos/canvas3d`.

## Getting Started

As with all JWAVE applets, you need to create a client-side component (the applet) and a server-side component (the JWAVE wrapper), as explained in Chapter 4, *JWAVE Graphics* and Chapter 5, *JWAVE Server Development*. To use the advanced features effectively, your JWAVE applet must register as a listener for the desired feature. A basic understanding of the Java 1.1 event model will be useful in implementing the advanced features.

### Client-Side Development

To interact with a JWAVE Viewable object you must first plan and create a JWAVE applet that does the following:

- Extend one of the 2D or 3D JWaveCanvases or JWavePanels.
- Register as a listener for the desired user interaction feature.
- Create a connection to the server.
- Execute a PV-WAVE wrapper function.
- Display graphical results, such as an image or surface.

## Server-Side Development

As with all JWAVE applets, you also need to write a JWAVE wrapper function, or set of functions, to generate the graphical output that is displayed in the client applet.

In some cases, the wrapper function may need to return data if the applet is to execute some of the features on the client side. For example, with the PickPoint feature, JWavePanel2D will return the x and y location where the point was selected on the Panel. If the applet is to display the value of the data where the mouse click occurred, then x and y can be used as indices into the 2D array of data to obtain the data value and display the data in the graphical user interface. For an example see `VNI_DIR/classes/jwave_demos/canvas2d/demo.java.`

## Example Applets

To get a feel for how these advanced features work, you can run a set of example applets.

Each example demonstrates one or more of the advanced features and contains comments that explain how they are added.

For more information on using these example applets, see the next section, *Running the Demonstration Applets* on page 143.

## *Running the Demonstration Applets*

Several applets that demonstrate the use of the advanced graphics features are located in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/canvas2d` |
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/canvas3d` |
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/panel2d` |
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/panel3d` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\canvas2d` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\canvas3d` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\panel2d` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\panel3d` |

where `VNI_DIR` is the main Visual Numerics installation directory.

These demonstrations include:

- ShadeSurfDemo, an applet that demonstrates the use of Rotation features.
- Demo, an applet that integrates PickPoint, Zoom, Profile features.

### Running the Demo Applets

To run these applets, you need to have a properly installed version of JWAVE — 3.5 or higher.

#### *Using appletviewer*

If you are using appletviewer, do the following:

**Step 1** Start the JWAVE manager.

**Step 2** Change to the directory:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/jwave_demos/canvas2d` |
| **(Windows)** | `VNI_DIR\classes\jwave_demos\canvas2d` |

where `VNI_DIR` is the main Visual Numerics installation directory.

**Step 3** Enter:

```
appletviewer appletname.html
```

where *appletname* is the name of the applet you wish to run. For example:

```
$ appletviewer demo.html
```

### *Using a Browser*

If you are using a browser to connect to JWAVE, do the following:

**Step 1** Start the JWAVE Manager.

**Step 2** Point the browser to the applet you wish to run, for example:

```
http://opus:6580/jwave_demos/canvas2d/demo.html
```

where *machine* is the name of the machine where the JWaveManager is running; *port* is the port number where the JWaveManager is configured to listen for HTTP connections (by default, 6580); and *appletname* is the name of the demonstration applet you wish to run. For example:

```
http://opus:6580/jwave_demos/canvas2d/demo.html
```

## PV-WAVE Wrappers Used by the Demos

The wrappers used by the example applets are located in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/jwave-3_5/lib/user` |
| **(Windows)** | `VNI_DIR\jwave-3_5\lib\user` |

where `VNI_DIR` is the main Visual Numerics installation directory.

The wrappers handle reading a data file, storing data in the Data Manager, and creating plots using the data and parameters retrieved from the applet.

The following PV-WAVE wrappers are used by the demonstration plugin applets.

| | |
|---|---|
| Demo | `wimage.pro`<br>`wprofile.pro`<br>`wreaddata.pro` |
| ShadeSurfDemo | `wsurface.pro`<br>`wreaddata.pro` |

10

CHAPTER

# JSPs, Servlets, and JWAVE

JavaServer Pages (JSP) technology provides a way to create web pages that display dynamically-generated content. This technology allows you to create Web-based applications that do not require a Java VM to be running on the client. JSPs and servlets, running in a Web server, perform all of the application processing and deliver results in HTML format back to the client.

This chapter explains how you can take advantage of JSPs and servlets to create a server-side JWAVE application. The basic architecture of such a system is illustrated in *Figure 10-1*. In this figure, the JWaveJSPServlet is a servlet that manages JWAVE connections and parameter passing to and from PV-WAVE. This servlet is described in detail later in this chapter.

## *Benefits of this Architecture*

Using a server-based JWAVE architecture has the following benefits:

- Little or no Java development is required by the developer.
- The client does not require a Java VM. Only HTML is delivered to the client.
- Multiple plots can be generated and displayed in a single Web page through one call to PV-WAVE.
- Because all JWAVE connections are handled on the server, it is possible to create connections to multiple JWAVE Managers running on multiple servers.

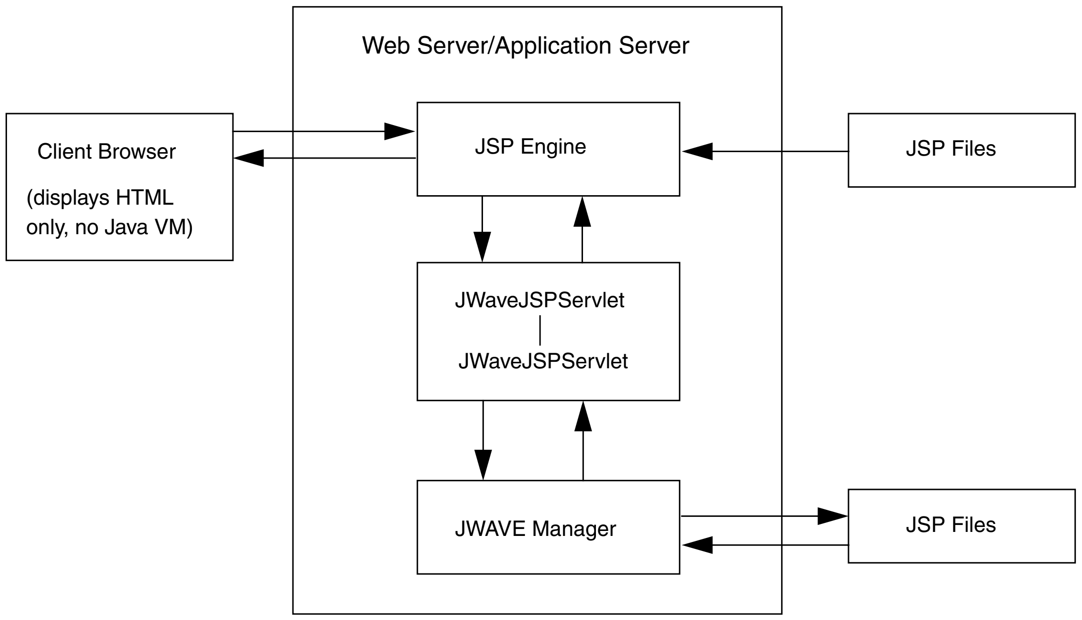


**Figure 10-1** A server-side JWAVE architecture using JSPs and servlets.

## *What is the JWaveJSPServlet?*

The JWaveJSPServlet provides a mechanism for creating a JWAVE solution where all processing occurs on the server. In this architecture, the client only displays HTML. Applets, on the other hand, are Java programs that require a Java VM running on the client. Java Server Pages (JSP) have become a standard technology for displaying content that is dynamically generated in this way.

### Location of the JWaveJSPServlet

The source code for the JWaveJSPServlet is located in:

**(UNIX)** `VNI_DIR/classes/com/visualnumerics/servlet`

**(Windows)** `VNI_DIR\classes\com\visualnumerics\servlet`

where `VNI_DIR` is the main Visual Numerics installation directory.

## Purpose of the JWaveJSPServlet

The JWaveJSPServlet serves two primary purposes:

- a functional program that you can use “as is” to create JSP/JWAVE interactions.
- an example that demonstrates how you can create a custom servlet that lets JWAVE work with JSP pages (dynamically generated content). The JWaveJSPServlet demonstrates how a set of JWAVE utility classes are used to manage images.

## Overview of the JWaveJSPServlet

The JWaveJSPServlet accepts requests from a JSP page. Typically, the JSP page is used to display an HTML form interface through which a user can specify the parameters for a task, such as creating a plot or processing an image. The JSP page calls on a servlet to process these requests. The servlet is responsible for calling PV-WAVE, which generates results and returns them to the client.

The output generated by PV-WAVE is either graphical or numerical. When the results are returned to the servlet, the servlet responds by displaying a new JSP page that contains the results of the analysis. Numerical results can be forwarded directly back to the client as tags in the JSP Response object. Graphical results can either be streamed directly back to the client, or can be stored on the server for later retrieval. (A class called the JWaveImageManager, described in the next section, manages the storage and retrieval of images.) For numerical results, the JSP page simply retrieves variables from the Response object and displays them. For graphical results, the JSP page obtains a URL that allows the client to retrieve a graphic that was stored on the server by the JWaveImageManager.

## The JWaveImageManager

The JWaveImageManager is a convenience class that manages images on the server. Because images can take time to generate, users might normally experience a delay between the time when they submit a request from their browser and the time the image is received. If many users happen to be contacting the JWAVE server simultaneously, the delay could be substantial. By storing images on the server and returning a URL to the client, the user will not experience any appreciable delay in the server’s response. Numerical results are displayed almost immediately, while the images appear as soon as they are available. For security, each image stored by the JWaveImageManager is given a unique ID, which is returned to the client in the URL.

For more information, see *Understanding the JWaveJSPServlet* on page 150.

## *Setting Up the JWaveJSPServlet*

A good way to get started using JWAVE with JSPs and servlets is to set up and run a demonstration using the JWaveJSPServlet.

This section, along with the detailed installation instructions included with the JWAVE 3.5 distribution, explains how to set up the JWaveJSPServlet and JSP files on a Web server, and how to run a demonstration.

The JWaveJSPServlet, or any servlet that you write that uses the JWAVE JSP classes, must be installed in a Web server that contains a Java Servlet engine. Such servers are common.

### System Requirements

- JWAVE 3.5 and PV-WAVE 7.01 or later must be installed and properly configured on the JWAVE server.
- Netscape 4.76, or Internet Explorer 5.0 or later.
- A Web server that contains a Java Servlet engine.
- Java Advanced Imaging.
- Java 2 Runtime Environment 1.2.

**NOTE** Each Web server's configuration is somewhat different; therefore, you need to rely on your Web server's documentation for explicit instructions.

One method of deploying JSPs and Servlets on a Web server is to create a Web Application, or webapp. The webapp is a standard configuration that specifies where servlets and JSPs are located in the Web server. *Figure 10-2* shows a sample webapp directory structure, where the JWAVE webapp is called `jwavejsp`.

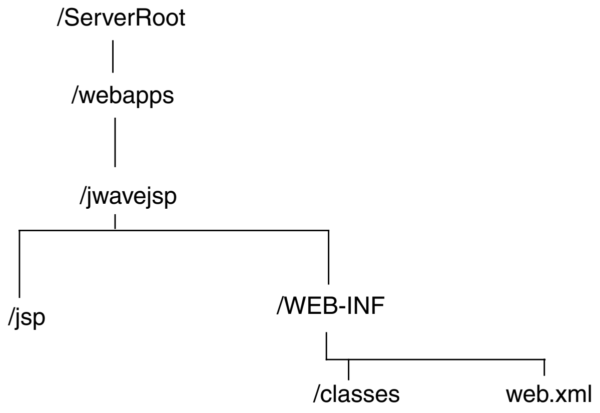


**Figure 10-2** A sample webapp directory structure.

- The `/jsp` directory contains JSP files.
- The `/classes` directory contains the required Java classes; the JWaveJSPServlet class path directory structure and its related classes are copied to this directory.

---

**NOTE** The server must be configured to recognize a webapp. Typically, the Web server has a configuration file that is used to specify the names of webapps. The documentation for your Web server will include instructions on configuring a webapp.

---

**TIP** Refer to the file `VNI_DIR/jwave-3_5/jspservletUtils/README_install.html`, which contains detailed instructions for installing the JWaveJSPServlet on three common Web servers.

---

## Setting Up the JWAVE Server

To use the JWaveJSPServlet, the JWAVE Manager must be running on the server. Typically, the JWAVE Manager runs on the same machine as the Web server.

## Running the JWaveJSPServlet

To run the JWaveJSPServlet demonstrations, do the following:

**Step 1** Start the Web server.

**Step 2** Start the JWAVE Manager.

**Step 3** From any machine that has access to the server, point a Web browser to the file: `index.jsp`. For example:

```
http://machine:port/jwavejsp/jsp/index.jsp
```

where machine is the hostname of the server machine, and port is the number of the port that the Web server is configured to listen on. For example, assuming that a Web server named “opus” is configured to recognize a webapp called `jwavejsp`, and the `/jsp` directory of the webapp contains a JSP page called `loan.jsp`:

```
http://opus:7001/jwavejsp/jsp/loan.jsp
```

If everything is configured properly, you will see a JSP page with form inputs. When you submit a form, the JSP page is dynamically updated with graphics and text.

## Understanding the JWaveJSPServlet

The JWaveJSPServlet receives requests from JSP pages and passes them on to PV-WAVE through standard JWAVE methods. Results from PV-WAVE are passed back through the servlet to a reply JSP page, which displays the results in HTML format on the client.

This section discusses the JWaveJSPServlet and its related pieces: JSP pages and JWAVE wrappers.

**TIP** The source code for the JWaveJSPServlet is provided with your JWAVE installation in: `VNI_DIR/classes/com/visualnumerics/servlet`.

## The JSP Files

JSP files are basically HTML documents with special tags that invoke the JWave-JSPServlet in response to submitting a form or clicking on a plot. Result JSP files include tags that are filled in by the JWaveJSPServlet with content from the server-side PV-WAVE process.

## JWAVE Wrappers

JWAVE wrappers are passed the following kinds of data from the client through the JSP engine:

- Fields from HTML forms (including hidden fields used to specify the JWAVE wrapper, a PV-WAVE session ID, and others).
- Data coordinates of where a user clicked in an image map.

The JWAVE wrapper code contains all of the application logic. Data is accessed, manipulated, and numerical and graphical results are produced.

The wrappers use utility routines to help package and return graphics and tables. These utilities are discussed in *Writing the JWAVE Wrappers* on page 154.

## Inside the JWaveJSPServlet: GET Requests, POST Requests, and the JWaveImageManager

The JWaveJSPServlet handles GET and POST requests. Typically, a GET request is used to retrieve an image that has been stored by the JWaveImageManager, or to return it directly to the client. For example, an IMG SRC command in HTML makes a GET request to get an image. A POST request is made when the client needs to send parameters and data to the server for processing. For example, a POST request is typically made when the user fills in a form and clicks the SUBMIT button.

### *The POST Request*

When the JWaveJSPServlet receives a POST request, the doPost method is called. This method retrieves the posted parameters from the request object and calls the callWave method. The callWave method handles most of the JWAVE work, including making a connection to PV-WAVE, passing the parameters to PV-WAVE, executing the JWAVE wrapper, retrieving the results from PV-WAVE, and dispatching the results back to the client (by forwarding the response to another JSP page).

Graphical results are identified in the doPost method and registered (stored) in the JWaveImageManager. The JWaveImageManager contains a small number of public methods. The methods that are used in the doPost method are:

```
public String registerImages(JWaveExecute jexecute, String imageName)
```

```
public void setFormat(String encodeformat)
```

```
public void setTimeout(int seconds)
```

```
public String getUname()
```

The registerImages method is used to store a named image in the JWaveImageManager. As explained previously, a URL that points to the stored image is forwarded from the servlet to a JSP page that the client displays. The URL is embedded in an IMG SRC command in the HTML page that is sent to the client, enabling the client to retrieve and display the stored image.

Use the setFormat method to set the image format. For example, PNG is a commonly used format for delivering images over the Internet.

The setTimeout method allows you to configure how much time a graphic can be stored on the server before it is deleted and becomes a candidate for garbage collection.

### *The GET Request*

The JWaveJSPServlet also has a doGet method. This method is called whenever an image is requested by the client. Typically, the client requests an image when it displays an HTML page containing an IMG SRC command.

The doGet method calls the JWaveImageManager method processRequest:

```
public boolean processRequest(HttpServletRequest req,
   HttpServletResponse res, String uname)
```

This method retrieves the image that is specified in the Request object (a URL plus a unique filename) and streams it back to the client.

## Writing Your Own JWaveJSPServlet

The JWaveJSPServlet provided with JWAVE may not be suitable in every situation. You might want to write your own servlet that manages JWAVE operations. If you write your own servlet, you can use the JWaveJSPServlet as a template. The JWaveJSPServlet relies on several convenience classes that primarily help manage images. These classes include:

- ConnectionContainer.class
- JWaveImageControlThread.class
- JWaveImageContainer.class
- JWaveImageManager.class
- connectionControlThread.class

The public interfaces for these classes are documented in the JWAVE Javadoc. JWAVE Javadocs are discussed in *Using the JWAVE Javadoc Reference* on page 40.

## How Image Maps are Handled

An image map is a graphic that is displayed in an HTML page that allows limited user interaction. When you click in an imagemap, the x/y coordinates of the click are appended to a GET request that is sent to the server. The server can then use the coordinates to perform some additional processing, such as zooming in on a region surrounding the selected point.

When an HTML browser downloads an image with ISMAP appended to its URL, that image is interpreted to be an image map. When a user clicks in an image map, the resulting GET request contains the x/y coordinates of the click in its URL. On the server, the servlet that handles the GET request must include code that handles the image map coordinates. In the JWaveJSPServlet, for example, the doGet method tests for the presence of appended x/y coordinates, strips off the *x* and *y* values, and stores them along with other parameters and passes them to a JWAVE wrapper function that is executed by PV-WAVE. It is up to the developer to determine what the wrapper function actually does with the image map coordinates that are passed to it.

If you want an image that is returned from PV-WAVE to the JWaveJSPServlet (or a custom servlet that you write) to be an image map, set the *Ismap* keyword in the PACKIMAGE wrapper function. The PACKIMAGE function is described in the next section.

## Writing the JWAVE Wrappers

As with any JWAVE application, a JWaveJSPServlet application relies on JWAVE wrapper functions to perform the data and graphical analysis functions. This section describes a typical wrapper for a JWaveJSP application and discusses three related wrapper utilities.

**NOTE** For an introduction to JWAVE wrappers, see Chapter 5, *Writing JWAVE Wrapper Functions*.

With JWaveJSPServlet applications, there are three PV-WAVE functions that you will use in your wrappers. If you examine the `loan.pro` wrapper, you will see these three routines in use:

- PACKIMAGE
- PACKTABLE
- SETIMAGESIZE

These routines are located in:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/jwave-3_5/lib` |
| **(Windows)** | `VNI_DIR\jwave-3_5\lib` |

where `VNI_DIR` is the main Visual Numerics installation directory.

### PACKIMAGE Function

This function takes as its parameters the name of an associative array and the name of an image or plot. This routine simply packages the 2D byte array representing the image or plot and three 1D byte arrays representing the RGB color values for the graphic. These variables are stored in an associative array which is returned to the calling procedure. The calling procedure then returns these items to the servlet.

This function has one keyword, *Ismap*, that lets you specify if the returned image is an image map.

For more information, see *PACKIMAGE Procedure* on page 22.

## PACKTABLE Function

This function takes string data and converts it into an HTML table format. The HTML table data is converted to a 1D byte array. The array is then stored in an associative array which is returned to the calling procedure. The calling procedure then returns this data to the servlet.

The HTML table data is converted to byte because of a limitation in the size of strings that JWAVE can handle. In the servlet, this byte array is converted back into a string, and it is then forwarded to a JSP page for the client to display.

For more information, see *PACKTABLE Procedure* on page 22.

## SETIMAGESIZE Procedure

This procedure sets the size of the image appropriately for the current PV‑WAVE device driver.

For more information, see *SETIMAGESIZE Procedure* on page 24.

## Example

This example JWaveJSPServlet demonstration uses the JSP `VNI_DIR/classes/jwave_demos/loan.jsp` and the wrapper `VNI_DIR/jwave-3_5/lib/user/loan.pro`.

More JSP demonstration files are located in:

**(UNIX)** `VNI_DIR/jwave_demos/jsp`

**(Windows)** `VNI_DIR\jwave_demos\jsp`

where `VNI_DIR` is the main Visual Numerics installation directory.

Wrappers for this and other JWaveJSPServlet demonstrations are located in:

**(UNIX)** `VNI_DIR/jwave-3_5/lib/user`

**(Windows)** `VNI_DIR\jwave-3_5\lib\user`

where `VNI_DIR` is the main Visual Numerics installation directory.

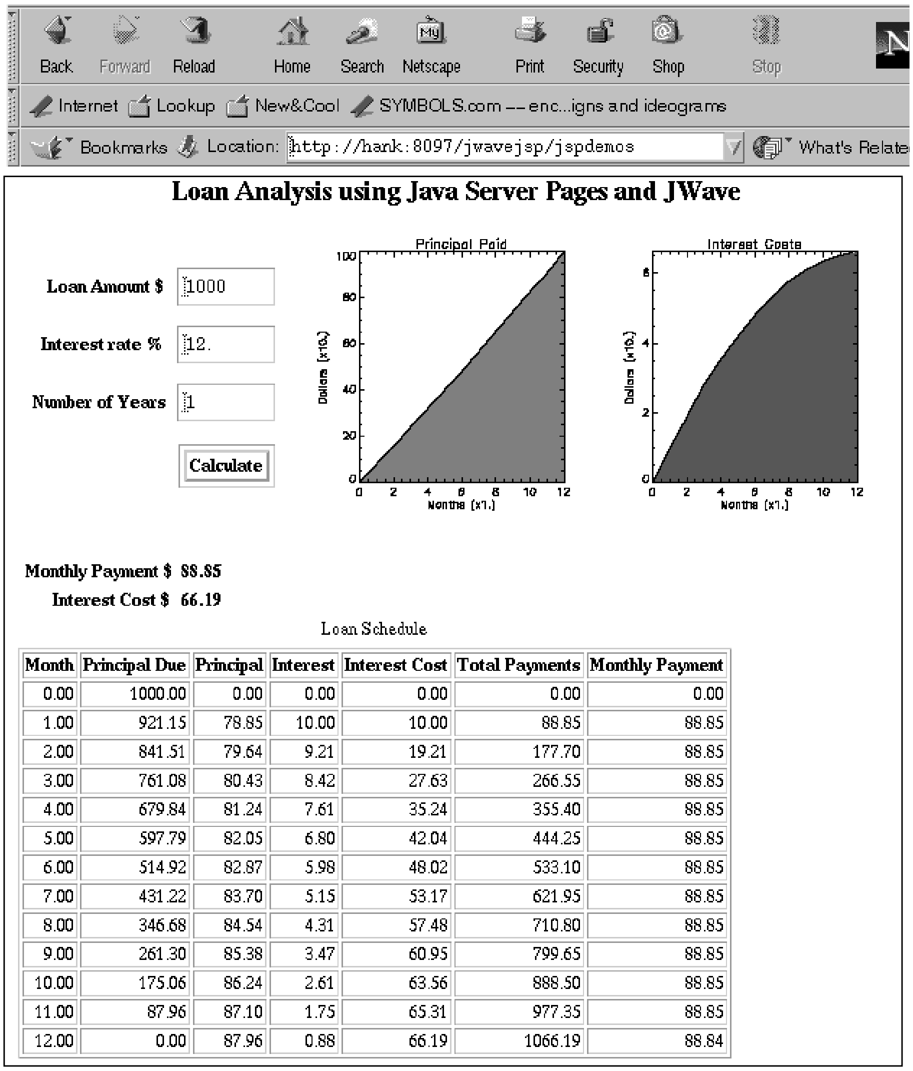
Parameters are sent by `loan.jsp` to the wrapper `loan.pro` using standard HTML inputs and calls to request.getParameter and request.getAttribute, enclosed in scriptlet tags (`"<%"` and `"%>"`). The first time `loan.jsp` is invoked, each call to request.getParameter *wrapper* is null; this causes default values to be generated for each input field, but the wrapper `loan.pro` is not yet called. Then, when the **Calculate** button is pressed, the parameters entered for **Loan Amount $**,

**Interest Rate %**, and **Number of Years** are passed to the wrapper, which is defined to be `loan.pro` in the SUBMIT parameter list.

```
<BODY BGCOLOR="white">
<H2><CENTER>Loan Analysis using Java Server Pages and JWave</CEN-
   TER></H2>
<FORM METHOD=POST ACTION=/jwavejsp/jspdemos>
   <TABLE>
   <TR><TH>
   <TABLE><FONT SIZE=3>
   <TR ALIGN=RIGHT><TH>Loan Amount $</TH>
   <TH><INPUT NAME=amount TYPE=TEXT VALUE=
   <% if ( request.getParameter("wrapper") != null ) { %>
      <%= request.getAttribute("AMOUNT") %>
   <% } else { %>
      100000
   <% } %>
   SIZE=8></TH></TR>
   <TR ALIGN=RIGHT><TH>Interest rate %</TH>
   <TH><INPUT NAME=interest TYPE=TEXT VALUE=
   <% if ( request.getParameter("wrapper") != null ) { %>
      <%= request.getAttribute("INTEREST") %>
   <% } else { %>
      8.0
   <% } %>
   SIZE=8></TH></TR>
   <TR ALIGN=RIGHT><TH>Number of Years</TH>
   <TH><INPUT NAME=years TYPE=TEXT VALUE=
   <% if ( request.getParameter("wrapper") != null ) { %>
      <%= request.getAttribute("YEARS") %>
   <% } else { %>
      10
   <% } %>
   SIZE=8></TH></TR>
   <INPUT NAME="jsp" TYPE=HIDDEN VALUE="/jsp/loan.jsp">
   <INPUT NAME="wrapper" TYPE=HIDDEN VALUE=loan>
   <TR ALIGN=RIGHT><TH></TH>
   <TH><INPUT TYPE=SUBMIT VALUE="Calculate"></TH></TR>
   </FONT></TABLE>
   </TH><TH>
   <% if ( request.getParameter("wrapper") != null ) { %>
      <%= request.getAttribute("PRINCPLOT") %>
      <%= request.getAttribute("INTPLOT") %>
   <% } %>
   </TH></TR>
   </TABLE>
</FORM>
<% if ( request.getParameter("wrapper") != null ) { %>
   <TABLE><FONT SIZE=3>
   <TR ALIGN=RIGHT><TH>Monthly Payment $</TH>
   <TH><%= request.getAttribute("PAYMENT") %></TH></TR>
   <TR ALIGN=RIGHT><TH>Interest Cost $</TH>
   <TH><%= request.getAttribute("COST") %></TH></TR>
   </FONT></TABLE>
   <%= request.getAttribute("SCHEDULE") %>
<% } %>
</BODY>
</HTML>
```

This wrapper uses the getParam function to retrieve input that was sent from the servlet. The parameters are then processed and several results are generated, including a table and two plots. The PackImage and PackTable routines are used to store graphical and tabular results in an associative array, which is then passed back to the servlet when the function returns.

```
FUNCTION Loan, client_data
   ; Get params from HTML FORM
   amount   = GetParam(client_data, 'AMOUNT', /Value)
   interest = GetParam(client_data, ‘INTEREST’, /Value)
   years    = GetParam(client_data, ‘YEARS’, /Value)
   ; Calculate results
   result = CalcLoan(amount, interest, years)
   ; Convert 2D float array to an HTML table
   schedule = result(‘schedule’)
   collab = [‘Month’,’Principal Due’,’Principal’,’Interest’, $
             ‘Interest Cost’,’Total Payments’,’Monthly Payment’]
   PackTable, TRANSPOSE(schedule), ‘SCHEDULE’, ret, $
               /Right, ColLabels=collab, $
               Border=1, Caption=’Loan Schedule’
   TEK_COLOR
   ; Create principal paid plot
   SetImageSize, 200, 200
   PlotLoan, ‘principal’
   PackImage, ret, ‘princplot’
   ; Create interest cost plot
   SetImageSize, 200, 200
   PlotLoan, ‘interest’
   PackImage, ret, ‘intplot’
   ; Return original values to be used to fill in form
   ret(‘AMOUNT’)   = amount
   ret(‘INTEREST’) = interest
   ret(‘YEARS’)    = years
   ; Return calculated values and loan schedule table
   ret(‘PAYMENT’)  = result(‘payment’)
   ret(‘COST’)     = result(‘interest_cost’)
   RETURN, ret
END
```



**Loan Analysis using Java Server Pages and JWave**

| | |
|---|---|
| Loan Amount $ | 1000 |
| Interest rate % | 12. |
| Number of Years | 1 |

Calculate

**Monthly Payment $ 88.85**

**Interest Cost $ 66.19**

Loan Schedule

| Month | Principal Due | Principal | Interest | Interest Cost | Total Payments | Monthly Payment |
|---|---|---|---|---|---|---|
| 0.00 | 1000.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 1.00 | 921.15 | 78.85 | 10.00 | 10.00 | 88.85 | 88.85 |
| 2.00 | 841.51 | 79.64 | 9.21 | 19.21 | 177.70 | 88.85 |
| 3.00 | 761.08 | 80.43 | 8.42 | 27.63 | 266.55 | 88.85 |
| 4.00 | 679.84 | 81.24 | 7.61 | 35.24 | 355.40 | 88.85 |
| 5.00 | 597.79 | 82.05 | 6.80 | 42.04 | 444.25 | 88.85 |
| 6.00 | 514.92 | 82.87 | 5.98 | 48.02 | 533.10 | 88.85 |
| 7.00 | 431.22 | 83.70 | 5.15 | 53.17 | 621.95 | 88.85 |
| 8.00 | 346.68 | 84.54 | 4.31 | 57.48 | 710.80 | 88.85 |
| 9.00 | 261.30 | 85.38 | 3.47 | 60.95 | 799.65 | 88.85 |
| 10.00 | 175.06 | 86.24 | 2.61 | 63.56 | 888.50 | 88.85 |
| 11.00 | 87.96 | 87.10 | 1.75 | 65.31 | 977.35 | 88.85 |
| 12.00 | 0.00 | 87.96 | 0.88 | 66.19 | 1066.19 | 88.84 |

**Figure 10-3** Loan Analysis using Java Server Pages and JWAVE.

A

APPENDIX

# JWAVE Wrapper API

## DMCopyData Procedure

Copies a named dataset in the current PV-WAVE session.

### Usage

DMCopyData, *data_name*, *new_var_name*

### Input Parameters

***data_name*** — A scalar string containing a data name (in the default domain, `$GLOBAL`) or a 2-element string array containing a domain name and data name of the data you wish to copy.

***new_var_name*** — A string containing a new variable name.

### Discussion

This function cannot be used to copy data to another domain. Both datasets (the original and new ones) will exist after this function is called.

Storage for the domain and data are automatically created if they do not already exist. If data with the *new_var_name* already exists, it is overwritten. The data name and domain name are used to uniquely identify a dataset stored on the JWAVE server. Normally, these names are set in the JWAVE client application.

To be valid, the domain name and data name must begin with a letter. The name is not case-sensitive, and may contain letters, underscores, and numbers.

### Examples

This command makes a copy of data named `STUFF` from the default domain (`$GLOBAL`) and stores it in data named `STUFF_COPY` (in the same domain).

```
DMCopyData,  ['$GLOBAL', 'STUFF' ], 'STUFF_COPY'
```

### See Also

DMDataExists, DMEnumerateData, DMGetData, DMInit, DMRemoveData, DMRenameData, DMStoreData

## *DMDataExists Function*

Determines if the named data exists in the current PV-WAVE session.

### Usage

*result* = DMDataExists(*data_name*)

### Input Parameters

***data_name*** — A scalar string containing a data name (in the default domain, `$GLOBAL`) or a 2-element string array containing a domain name and data name.

### Returned Value

***result*** — Returns `true` if the named data exists; returns `false` otherwise. Also returns `false` if the *data_name* is invalid.

### Discussion

To be valid, the *data_name* input parameter name must begin with a letter. The name is not case-sensitive, and may contain letters, underscores, and numbers.

### Examples

See the example for DMRestore.

### See Also

DMCopyData, DMEnumerateData, DMGetData, DMInit, DMRemoveData, DMRenameData, DMRestore, DMStoreData

## *DMEnumerateData Function*

Returns the data names stored in the specified domain in the current PV-WAVE session.

### Usage

*data_names* = DMEnumerateData(*domain_name*)

### Input Parameters

***domain_name*** — A string containing the name of the domain that you wish to enumerate. If not specified (i.e., `DMEnumerateData()`), then the default domain (`$GLOBAL`) is enumerated.

### Returned Value

***data_names*** — A string array containing the data names for the data stored in the specified domain.

### Discussion

This function returns an empty string if there are no variables in the specified domain or if the specified domain does not exist.

To be valid, the *data_name* input parameter name must begin with a letter. The name is not case-sensitive, and may contain letters, underscores, and numbers.

### Examples

See the example for DMSave.

### See Also

DMCopyData, DMDataExists, DMGetData, DMInit, DMRenameData, DMRemoveData, DMSave, DMStoreData,

## *DMGetData Function*

Returns a stored dataset in the current PV-WAVE session.

### Usage

*values* = DMGetData(*data_name*)

### Input Parameters

***data_name*** — A scalar string containing a data name (in the default domain, `$GLOBAL`) or a 2-element string array containing a domain name and data name.

### Returned Value

***values*** — The data values stored under the specified *data_name*.

### Discussion

This function produces an error if there is no data stored in the specified location. Use DMDataExists to test for this condition.

To be valid, the *data_name* input parameter name must begin with a letter. The name is not case-sensitive, and may contain letters, underscores, and numbers.

### Examples

The following command gets data named `STUFF` from the default domain (`$GLOBAL`) and stores it in a variable named `my_data`.

```
my_data = DMGetData( ['$GLOBAL', 'STUFF' ] )
```

### See Also

DMCopyData, DMDataExists, DMEnumerateData, DMInit, DMRenameData, DMRemoveData, DMStoreData,

## *DMInit Procedure*

Initializes the JWAVE Data Manager.

### Usage

DMInit

### Input Parameters

None.

### Discussion

This procedure must be run before any DM routines can be used.

**NOTE** Normally, you do not need to run this procedure, because it is run automatically by the PV-WAVE server and by the WRAPPER_TEST_INIT routine.

### See Also

DMCopyData, DMDataExists, DMEnumerateData, DMGetData, DMRenameData, DMRemoveData, DMStoreData, WRAPPER_TEST_INIT

## *DMRemoveData Procedure*

Deletes a dataset from the current PV-WAVE session.

### Usage

DMRemoveData, *data_name*

### Input Parameters

***data_name*** — A scalar string containing a data name (in the default domain, `$GLOBAL`) or a 2-element string array containing a domain name and data name of the data you wish to remove.

### Discussion

No error is produced if you attempt to delete data that does not exist; however, an error is produced if you specify an incorrectly formed *data_name*.

To be valid, the *data_name* input parameter name must begin with a letter. The name is not case-sensitive, and may contain letters, underscores, and numbers.

### Examples

See the example for DMSave.

### See Also

DMCopyData, DMDataExists, DMEnumerateData, DMGetData, DMInit, DMRenameData, DMSave, DMStoreData

## *DMRenameData Procedure*

Renames a dataset stored in the current PV-WAVE session.

### Usage

DMRenameData, *data_name*, *new_var_name*

### Input Parameters

***data_name*** — A scalar string containing a data name (in the default domain, `$GLOBAL`) or a 2-element string array containing a domain name and data name of the data you wish to rename.

***new_var_name*** — A string containing a new variable name.

### Discussion

This function cannot be used to move data to a new domain. Only the name of the data can be changed.

The original *data_name* data no longer exists after this function is called.

Storage for the domain and data are automatically created if they do not already exist. If data with the *new_var_name* already exists, it is overwritten. The data name and domain name are used to uniquely identify a dataset stored on the JWAVE server. Normally, these names are set in the JWAVE client application.

To be valid, the *data_name* input parameter name must begin with a letter. The name is not case-sensitive, and may contain letters, underscores, and numbers.

### Examples

This command renames (moves) the data named `STUFF` from the default domain (`$GLOBAL`) and gives it a new name of `NEW_STUFF` (in the same domain).

```
DMRenameData,  ['$GLOBAL', 'STUFF' ], 'NEW_STUFF'
```

### See Also

DMCopyData, DMDataExists, DMEnumerateData, DMGetData, DMInit, DMRemoveData, DMStoreData,

## *DMRestore Procedure*

Restores the Data Manager from a file.

### Usage

DMRestore, *filename*

### Input Parameters

***filename*** — A string containing the name of the data file to restore. This file must have been created with DMSave.

### Keywords

***Overwrite*** — Erases everything in the Data Manager before restoring the contents of the file. By default, data from the file is added to the current Data Manager.

***Verbose*** — Prints information to the screen about the data that is being restored.

### Examples

This example can be used in an initialization routine—a JWAVE wrapper called initially by the client, or by the procedure indicated with the `JWAVE_STARTUP` configuration parameter. This configuration parameter is specified with the Configuration Tool, described in *Setting Up the JWAVE Server* on page 112. (By default, the initialization routine is called `JWAVE_START`.)

This example restores DM data from a file (saved previously by DMSave) and adds data named `STUFF` to the default domain (`$GLOBAL`). This data is then available for use by other wrapper functions (using DM routines), or the data can be accessed by the JWAVE client (using a `JWaveDataProxy` or `ServerDataID` object).

```
; Restore data from previous session
   DMRestore, my_dm_data_file

   ; Add new 'STUFF' data if it does not already exist
   IF (NOT DMDataExists( ['$GLOBAL', 'STUFF'] ) THEN BEGIN
      status = DC_READ_FREE(my_ascii_data, stuff)
      DMStoreData, ['$GLOBAL', 'STUFF'], stuff
   ENDIF
```

### See Also

DMCopyData, DMRemoveData, DMSave, DMStoreData

## *DMSave Procedure*

Saves all data that is managed by the JWAVE Data Manager to a file.

### Usage

DMSave, *filename*

### Input Parameters

***filename*** — A string containing the name of the file in which to save the data.

### Keywords

***Verbose*** — Prints information to the screen about the data that is being saved.

### Examples

This example code can be used in a shutdown routine (a JWAVE wrapper called initially by the client, or by `JWAVE_SHUTDOWN`). This example first removes (deletes) any data that has been stored in a domain named `TEMP`. Then, everything else is saved to a DM data file. That file may be restored (with DMRestore) for later use by another session.

```
; Enumerate all data stored in the TEMP domain
temp_data = DMEnumerateData('TEMP')
IF temp_data(0) NE '' THEN BEGIN
   ; Remove all data from TEMP domain
   FOR i = 0, N_ELEMENTS(temp_data)-1 DO BEGIN
      DMRemoveData, ['TEMP', temp_data(i) ]
   ENDFOR
ENDIF
; Save everything else
DMSave, my_dm_data_file
```

### See Also

DMCopyData, DMRemoveData, DMRestore, DMStoreData

## *DMStoreData Procedure*

Stores a dataset in the current PV-WAVE session.

### Usage

DMStoreData, *data_name*, *value*

### Input Parameters

***data_name*** — A scalar string containing a data name (in the default domain, `$GLOBAL`) or a 2-element string array containing a domain name and data name.

***value*** — The data that you wish to store.

## Discussion

Storage for the domain and data are automatically created if they do not already exist. If *data_name* already exists, it is overwritten. The *data_name* is used to uniquely identify a dataset stored on the JWAVE server. Normally, the *data_name* is set in the JWAVE client application.

To be valid, the *data_name* input parameter name must begin with a letter. The name is not case-sensitive, and may contain letters, underscores, and numbers.

The data specified with *value* can be of any PV-WAVE data type. Note that client applications can only store on the server scalars and arrays (up to eight dimensions) of the following data types:

| JAVA Data Types | Corresponding PV-WAVE Data Types |
|---|---|
| Byte | BYTE |
| Short | INTEGER |
| Integer | LONG |
| Float | FLOAT |
| Double | DOUBLE |
| String | STRING |

## Examples

See the example for DMRestore.

## See Also

DMCopyData, DMDataExists, DMEnumerateData, DMGetData, DMInit, DMRenameData, DMRemoveData, DMRestore

## *GETPARAM Function*

Retrieves parameters and data sent from a JWAVE client application.

### Usage

*result* = GETPARAM(*client_data*, *param_name*)

*result* = GETPARAM(*client_data*, *param_name*, /Value)

*result* = GETPARAM(*client_data*, *param_name*, /Positional)

### Input Parameters

***client_data*** — A variable containing parameters and data that were passed to the JWAVE wrapper function from a JWAVE client application. (This parameter receives the information that was set with calls to the `setParam` method on the client.)

***param_name*** — A string or string array specifying the name(s) of the parameter(s) to extract from the *client_data* variable.

**NOTE** The *param_name* parameter cannot be an array when either the *Value* or *Positional* keyword is specified.

### Keywords

***All*** — If nonzero, returns all of the keywords in the *client_data* variable. You do not need to specify the *param_name* parameter if you use the *All* keyword. Any keywords that were previously retrieved are ignored. Note that *All* retrieves parameters as keywords only. It does not retrieve positional parameters or values. The *All* keyword cannot be used when either the *Value* or *Positional* keyword is specified.

***ClientID*** — If nonzero, returns a unique number identifying the client making this request. You can use this keyword without specifying a *param_name* parameter.

***Default*** — Specifies a default value to be used if the given *param_name* was not provided by the client. This keyword can only be used when the *Value* keyword is specified. The default value can be any valid PV-WAVE data type.

***ExpectType*** — Provides type checking of returned values. This keyword is only allowed when the */Value* keyword is specified. For example:

```
ExpectType = type_number
```

where *type_number* is the PV-WAVE data type number (for instance, the number returned from the PV-WAVE function: `SIZE(val, /Type)`.

This test fails if the returned parameter does not match the expected type. On failure, the MESSAGE procedure is called.

***ExpectNumeric***,
***ExpectString*** — If nonzero, provides type checking of returned values.

These tests fail if the returned parameter does not match the expected type. On failure, the MESSAGE procedure is called.

***ExpectArray*** — Ensures that the function returns an array of the specified dimensions. If the *param_name* represents a scalar, a one-element array is returned. For example:

```
ExpectArray = [400, 600]
```

tests for a 400-by-600 element array. If any dimension is a zero (0), that dimension is taken as a wildcard (that dimension may be of any size).

This test fails if the returned parameter is not of the specified dimensions. On failure, the MESSAGE procedure is called.

***ExpectScalar*** — Ensures that the function returns a scalar. If the parameter is a one-element array, it is converted to a scalar, and a scalar is returned. This test fails if the returned parameter is an array of more than one element. On failure, the MESSAGE procedure is called.

***Keyword_Names*** — A string or string array of keyword names. Keywords are returned in the form:

```
" , param_name=param_ref "
```

where *param_name* is the keyword name and *param_ref* is a symbolic reference to the value associated with the keyword. For example:

```
result = GETPARAM(client_data, 'PARAM_NAME', $
   Keyword_Name = 'PARAM_KEY')
```

produces a string of the form:

```
" ,PARAM_KEY=param_name_ref "
```

By default, the parameter name and keyword name are the same. If *Keyword_Names* is an array, it must be the same number of elements as *param_name*.

**NOTE** The *Keyword_Names* keyword cannot be used when either the *Value* or *Positional* keyword is specified.

See the *Discussion* section for more information on *Keyword_Names*.

***IgnoreUsed*** — If nonzero, the parameters you request are returned regardless of if they have been previously retrieved. In addition, the parameters that you request are not added to the list of "used" parameters.

***Positional*** — If nonzero, indicates that the requested parameter is a positional parameter. The returned string is of the form:

" , *param_ref* "

where *param_ref* is a symbolic reference to data. See the *Discussion* section for more information on this keyword.

***SessionID*** — If nonzero, returns a unique number identifying this PV-WAVE session. This keyword is useful if you need to build a unique temporary filename. You can use this keyword without specifying a *param_name* parameter.

***Value*** — If nonzero, indicates that the actual data be returned rather than a string. See the *Discussion* section for more information on this keyword.

***WrapperName*** — If nonzero, returns a string naming the JWAVE wrapper function called by the client. You can use this keyword without specifying a *param_name* parameter.

## Discussion

You can use GETPARAM to return:

- single values
- positional parameters
- keyword parameters

See Chapter 3, *JWAVE Client Development* for additional information on using GETPARAM. The rest of this section describes these types of results briefly.

### *Returning Single Values*

To return a single value with GETPARAM, use the *Value* keyword. For example:

```
result = GETPARAM(client_data, 'X', /Value, Default=FINDGEN(100))
```

In this case, the actual value associated with the parameter `X` (which was passed to the JWAVE wrapper from the client application) is stored in `result`. The value can then be used in any PV-WAVE routine within the JWAVE wrapper. For example:

```
PLOT, result
```

**NOTE** If the *param_name* parameter is not set by the client, then GETPARAM returns either zero (0) or the value specified with the *Default* keyword.

### *Returning Positional Parameters*

To return a positional parameter string, use the *Positional* keyword. For example:

```
p1 = GETPARAM(client_data, 'X', /Positional)
```

returns a string of the form:

" , *param_ref* "

where *param_ref* is a symbolic reference to the value of the parameter `X`. Usually this value is a data reference or function call.

The comma (,) is included in the string so you can concatenate strings of this form together to build a command. Such a command string, then, can be used as input to an EXECUTE function. For example:

```
status = EXECUTE('PLOT' + p1)
```

**NOTE** If the *param_name* parameter to GETPARAM was not set by the client, GETPARAM returns an empty string.

For more information on positional parameters, see *Unpacking Command Strings on page 57*.

### *Returning Keyword Parameters*

To return a string of keyword parameters, call GETPARAM without either the *Value* or *Positional* keywords. For example:

```
title = GETPARAM(client_data, 'TITLE')
```

returns a string of the form:

" , *param_name*=*param_ref* "

where *param_name* is the name of the parameter (for example, `TITLE`) and *param_ref* is a symbolic reference to the value of the parameter.

The comma (,) is included in the string so you can concatenate strings of this form together to build a command. Such command strings, then, can be used as input to an EXECUTE function. For example:

```
status = EXECUTE('PLOT' + p1 + title)
```

If the second parameter to GETPARAM is an array, the function returns a string in the following form:

`" ,` *param_name_1=param_ref_1, param_name_2=param_ref_2, ...* `"`

---

**NOTE** If the *param_name* parameter to GETPARAM was not set by the client, GETPARAM returns an empty string.

---

For more information on positional parameters, see *Unpacking Command Strings* on page 57.

### *Returning All Keyword Parameters*

Use the *All* keyword to return all of the keyword parameters that were sent by the client in one string array. For example, the command:

```
result = GETPARAM(client_data, /All)
```

returns a string of the form:

`",` *param_name_1=param_ref1*`,` *param_name_2=param_ref2*`, ... "`

where *param_name_** are all parameters sent by the client, and *param_ref** are symbolic references to the values of those parameters.

---

**NOTE** Call GETPARAM with the *All* keyword after you have retrieved all of the positional and value parameters to ensure that you retrieve only the remaining keywords.

---

---

**TIP** We suggest that you use a *param_name* array rather than *All* so that the client cannot accidently send invalid parameters to the JWAVE wrapper function.

---

### *Parameters Are Retrieved Once*

In all the cases above, you can only retrieve a parameter once. After you have retrieved a parameter, it is marked as used, and further calls to GETPARAM will not retrieve it again. This feature allows you to call GETPARAM for some parameters, and then call GETPARAM with */All* to obtain a string containing all the rest of the keyword parameters. You can use the *IgnoreUsed* keyword to circumvent this restriction.

### *Notes and Restrictions*

- The keywords *All*, *Positional*, and *Value* are mutually exclusive, that is you can only use one of them for each call to GETPARAM.
- *All* and *Positional* are exclusive because the returned order of the parameters is not known.
- Similarly, if you specify either *Positional* or *Value*, then *param_names* cannot be an array.
- If you specify *Positional*, *Value*, or *All*, then you may not use *Keyword_Names*.
- *Keyword_Names* must be a string (array) with the same number of elements as *param_names*.
- *ExpectNumeric*, *ExpectString*, and *ExpectType* are mutually exclusive.
- *ExpectArray* and *ExpectScalar* are mutually exclusive.
- The *Expect** tests do not check your *Default* value, but only values supplied by the client.
- The *client_data* parameter must be a plain variable reference, and not a subscripted array or expression.
- *ClientID*, *SessionID*, and *WrapperName* are mutually exclusive. These keywords also cause all other keywords to be ignored.

## Examples

Here is an example showing how the *SessionID* keyword can be used to build a unique, temporary filename:

```
temp_file = STRTRIM(getParam(client_data), /SessionID), 2)

temp_file = FILEPATH(temp_file, /Tmp)
```

See the previous *Discussion* section for other examples. See also Chapter 5, *JWAVE Server Development* for additional information and examples.

## See Also

GET_NAMED_COLOR

In the *PV-WAVE Reference*: EXECUTE

## GET_NAMED_COLOR Function

Gets a color from a color name supplied by a JWAVE client application.

### Usage

*color* = GET_NAMED_COLOR(*color_name*)

### Input Parameters

***colorName*** — A string containing the name of the color you wish to retrieve from the client. (This name must have been supplied in the client Java application using the `JWaveView.setNamedColor`—or `setNamedColorSet`—method.)

### Input Keywords

***Color_Set*** — If set, GET_NAMED_COLOR returns an array of colors corresponding to a named color set. (In other words, use this keyword to retrieve colors that were packed by the client with the `JWaveView.setNamedColorSet` method.) You may have a color and a color set with the same name.

***DefaultRGB*** — Specifies a long integer (RGB value) representing the default color if the named color does not exist. (Default: '000000'xL (black))

### Output Keywords

***Range_Of_Colors*** — Retrieves the a two-element array containing the range of colors that are available for use by images. The first element represents the first color in the range, and the second element represents the last color. This range is equivalent to the number of colors in the color table minus the number of named colors that have been retrieved. See the *Discussion* for information on how this keyword is used.

### Returned Value

***Color*** — A color value that can be used by PV-WAVE.

### Discussion

GET_COLOR_NAME unpacks a color object sent from a Java client and returns a corresponding PV-WAVE color index (or, if you are using a 24-bit device, a 24-

bit color is returned). The returned color can be used in any PV‑WAVE context that uses color, such as the *Color* keyword associated with many of the graphics routines.

For example, the following calls might appear in a Java client application. They associate names with color objects. These name/color object pairs are sent to the JWAVE wrapper function when the `execute` method is called in the Java application.

```
myJWaveView.setNamedColor(”BACKGROUND”, java.awt.Color.lightGray)
```

```
myJWaveView.setNamedColor(”COLOR”, java.awt.Color.red)
```

The following GET_NAMED_COLOR calls retrieve these name/color pairs in the JWAVE wrapper function.

```
back = GET_NAMED_COLOR(”BACKGROUND”, Default=’000000’xL)
```

```
fore = GET_NAMED_COLOR(”COLOR”, Default = ’ffffff’xL)
```

Now, `back` and `fore` can be used in any PV‑WAVE expression that takes a color value. For example:

```
PLOT, x, Color=fore, Background=back
```

### *Managing the Color Table*

To load a color table in a JWAVE wrapper, you must call the JWAVE_LOADCT procedure. The resulting color table is subsetted into two parts that include (a) the named colors returned by GET_NAMED_COLOR and (b) the rest of the colors in the specified color table.

*Figure A-1* illustrates how a color table is created in a JWAVE wrapper. When LOAD_JWAVECT is called, a color table is created with the named colors loaded into a subset of the color table.

- First, a named color is passed to the wrapper and retrieved with GET_NAMED_COLOR.
- Next, the retrieved color is given a reserved spot in the color table.
- Finally, the color table specified by LOAD_JWAVECT (the image colors) is loaded into the remainder of the color table.

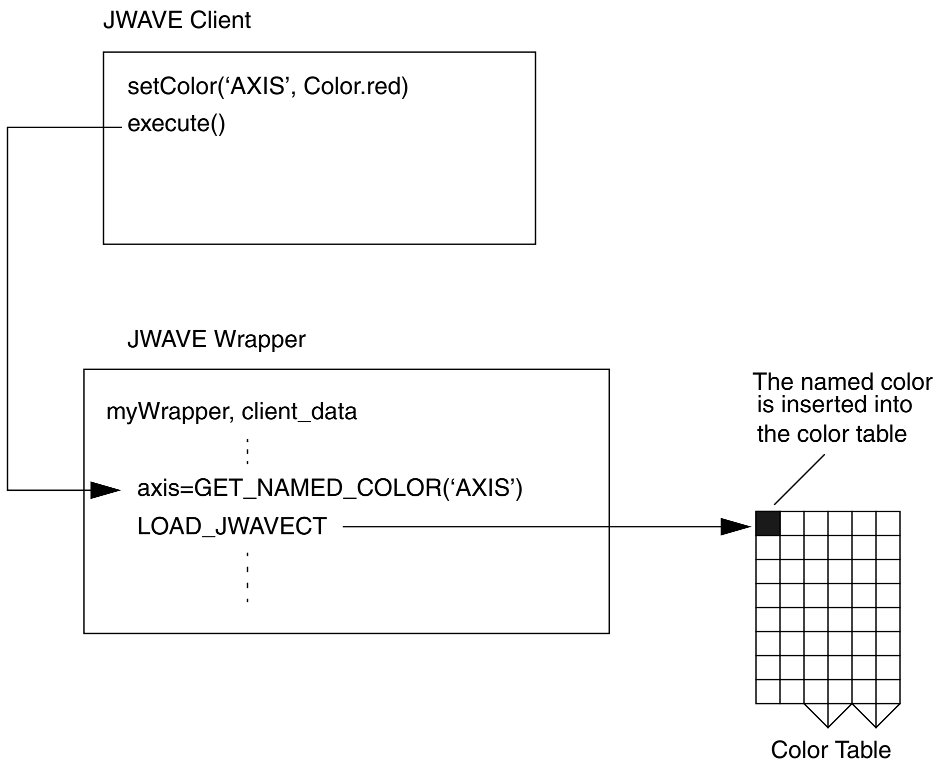

JWAVE Client
setColor('AXIS', Color.red)
execute()
JWAVE Wrapper
myWrapper, client_data
axis=GET_NAMED_COLOR('AXIS')
LOAD_JWAVECT
The named color is inserted into the color table
Color Table

**Figure A-1** A color is retrieved by GET_NAMED_COLOR in the JWAVE wrapper. When LOAD_JWAVECT is called, the named color is loaded into a subset of the specified color table. All remaining colors in the color table are available for use by images.

Use the *Range_Of_Colors* keyword to obtain the range of colors that are allocated for images in the color table (that is, all of the colors except the named colors retrieved by GET_NAMED_COLOR). For instance, in *Figure A-2*, the first five colors are allocated to the named colors. The remaining colors fall in the range {5..255}. These are the colors that are available for use by images, and {5..255} is the range that is returned by the *Range_Of_Colors* keyword. This range is expressed as a two-element array, such as: `range=[5, 255]`.

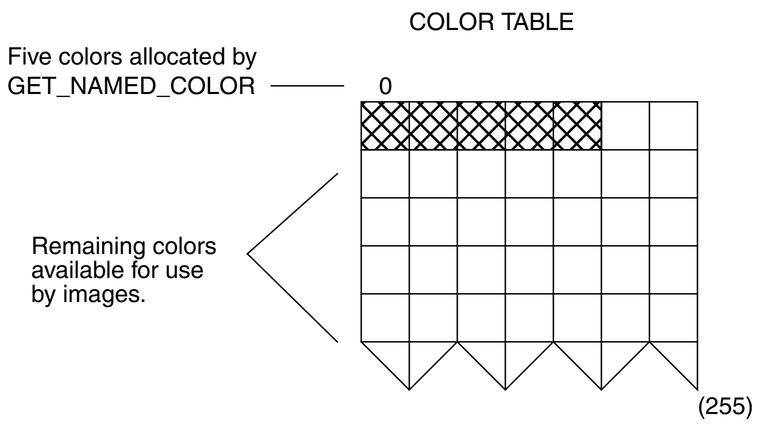

COLOR TABLE
Five colors allocated by GET_NAMED_COLOR
0
Remaining colors available for use by images.
(255)

**Figure A-2** Named colors occupy a subset of the color table.

Example 1-1 demonstrates how the *Range_Of_Colors* keyword is used to identify the image portion of the color table, and then this range is used to “smooth” this portion of the color table using the BYTSCL function.

**Example 1-1**

```
; Get colors
   JWAVE_LOADCT, 1
   back = get_Named_Color("BACKGROUND", Default = '000000'xL)
   fore = get_Named_Color("COLOR", Default = 'ffffff'xL)
   bot  = get_Named_Color("BOTTOM", Default = fore, $
           Range_Of_Colors=crange)


; Re-map image values into the range of image colors.
   s = BYTSCL(s, Top = crange(1)-crange(0)) + crange(0)
```

### *Notes and Restrictions*

- Only 256 colors are available for use in JWAVE wrapper functions.
- You may use the output of a previous call to GET_NAMED_COLOR as a default color.
- Valid color names must start with a letter and contain only letters (A-Z), digits (0-9), and underscores (_). They are not case sensitive.

- If you request a color set (set by the client with the method `setNamedColorSet`), then GET_NAMED_COLOR returns an array of color indices. This is useful for things such as the CONTOUR procedure's *C_Color* keyword.
- To ensure that the value of *Range_Of_Colors* keyword is valid, use the value of *Range_Of_Colors* from the *last* call to either GET_NAMED_COLOR or JWAVE_LOADCT.

---

**TIP** To create a default color, supply a long integer containing red, green, and blue components of the desired color. For example, the color chartreuse is represented by red=127, green=255, and blue=0 (in hex, 7F, FF, and 00). To create this color, use '00ff7f'xL as a constant. In an equation, you can form this constant using PV-WAVE expressions such as:

```
red + 256L*(green + 256L*blue)
```

or

```
LONG(red) OR ISHFT(LONG(green), 8) OR ISHFT(LONG(blue), 16)
```

---

## Examples

See the previous *Discussion* section for examples. See also Chapter 5, *JWAVE Server Development* for additional information and examples.

## See Also

GETPARAM

For more information on color tables and using color in PV-WAVE, refer to the *PV-WAVE User's Guide*.

## PACKIMAGE Procedure

Packs a 2D byte array (image) and three 1D byte arrays (RGB values) into an associative array.

### Usage

PACKIMAGE, *assarr*, *name*

### Discussion

This function takes as parameters the name of an associative array and the name of an image or plot. This routine simply packages the 2D byte array representing the image or plot and three 1D byte arrays representing the RGB color values for the graphic. These variables are stored in an associative array which is returned to the calling procedure. The calling procedure then returns these items to the servlet.

### Example

```
PackImage, ret, 'princplot'
```

### See Also

PACKTABLE, SETIMAGESIZE

## PACKTABLE Procedure

Converts string data to an HTML table.

### Usage

PACKTABLE, *table_text*

### Input Parameters

***table_text*** — An (*m*, *n*) string array of text to put in a table with *m* columns and *n* rows. A 1D array builds an *m*-column, 1-row table.

## Keywords

**NOTE** Whenever a specified attribute is not supported by a particular browser, the attribute is simply ignored by that browser.

***Border*** — The size of the border around cells in the table.

***Bottom*** — Places the cell content at the bottom of each cell.

***Caption*** — The table caption.

***CBottom*** — Table caption displayed beneath the table.

***CellPadding*** — Specifies the space between the borders and the content of the cell.

***CellSpacing*** — Specifies the space between each individual cell.

***Center*** — Centers the cell content.

***ColLabels*** — The column labels.

***EqualWidth*** — Defines all cells as having the same width as the largest one used.

***Left*** — Left-justifies the cell contents in the cell.

***Middle*** — Places the content in the middle of each cell.

***NoWrap*** — When set, the cell contents don't wrap onto multiple lines within the cell.

***Right*** — Right-justifies the cell contents in the cell.

***RowLabels*** — The row labels.

***Safe*** — Handles HTML special characters (see `HTML_SAFE`).

***TCenter*** — Centers the table on the page (left-right centering).

***TLeft*** — Left-justifies the table on the page. (Default: set)

***Top*** — Places the cell content at the top of each cell.

***TRight*** — Right-justifies the table on the page.

## Discussion

This function takes string data and converts it into an HTML table format. The HTML table data is converted to a 1D byte array. The array is then stored in an associative array which is returned to the calling procedure. The calling procedure then returns this data to the servlet.

The HTML table data is converted to byte because of a limitation in the size of strings that JWAVE can handle. In the servlet, this byte array is converted back into a string, and it is then forwarded to a JSP page for the client to display.

### Example

```
PackTable, TRANSPOSE(schedule), 'SCHEDULE', ret, $
               /Right, ColLabels=collab, $
               Border=1, Caption='Loan Schedule'
```

### See Also

PACKIMAGE, SETIMAGESIZE

## *SETIMAGESIZE Procedure*

Sets the size of the image appropriately for the current PV-WAVE device driver.

### Usage

SETIMAGESIZE, *xsize*, *ysize*

### Input Parameters

***xsize*** — The x-dimension size of the image, in pixels.

***ysize*** — The *y*-dimension size of the image, in pixels.

### Example

```
SetImageSize, 200, 200
```

### See Also

PACKIMAGE, PACKTABLE

## *UPDATE_LOG Procedure*

Writes logging and debugging information to JWAVE log file(s).

### Usage

UpdateLog [, *text* ]

### Input Parameters

***text*** — (optional) The text to output to the log. By default, the name of the calling procedure and a time stamp are output if no text is supplied.

### Keywords

***NoManagerLog*** — If nonzero, the log text is output to only the session log, and not the manager log.

***TimeStamp*** — If nonzero, a time stamp is prefixed to the text. The default is no time stamp if you supply text. The format of a timestamped log is:

*mm/dd/yyyy hh:mm:ss.sss : text*

### Discussion

This procedure can output to two different log files: the Session log and the Manager log.

- The Session log is a separate log file created for each PV-WAVE session (process).
- The Manager log is used by the JWAVE Manager for all PV-WAVE sessions.

UPDATE_LOG always writes to the Session log. By default, it also logs to the Manager log, but you can turn this off with the *NoManagerLog* keyword.

The JWAVE Manager controls whether or not any logging occurs. If you not receiving a log file, you must configure the JWAVE Manager to produce a log file. See *Using the JWAVE Configuration Tool* on page 118 for information on changing the log output.

The JWAVE server automatically executes the command:

```
UPDATE_LOG, /TimeStamp
```

every time a JWAVE wrapper function is called.

## WRAPPER_TEST_EXECUTE Procedure

Executes the JWAVE wrapper function named in WRAPPER_TEST_INIT.

### Usage

WRAPPER_TEST_EXECUTE

### Parameters

None.

### Discussion

The wrapper function is run with the parameters and colors that were set with WRAPPER_TEST_SETPARAM and WRAPPER_TEST_SETCOLOR.

WRAPPER_TEST_RETURN_INFO and WRAPPER_TEST_GETRETURN can be used to retrieve the results returned from the JWAVE wrapper.

If the JWAVE wrapper produces a plot, that plot is displayed in a PV‑WAVE window.

### Example

See WRAPPER_TEST_INIT.

### See Also

WRAPPER_TEST_GETRETURN, WRAPPER_TEST_INIT, WRAPPER_TEST_RETURN_INFO, WRAPPER_TEST_SETCOLOR, WRAPPER_TEST_SETPARAM

## *WRAPPER_TEST_GETRETURN Function*

Retrieves a value returned from a JWAVE wrapper function.

### Usage

*value* = WRAPPER_TEST_GETRETURN(*param_name*)

### Input Parameters

***param_name*** — A string containing the name of the return parameter that you wish to retrieve.

### Returned Value

***value*** — The value of the parameter.

### Discussion

You can retrieve the returned value from the JWAVE wrapper only after GET_TEST_EXECUTE has been run.

WRAPPER_TEST_GETRETURN issues a warning message and returns 0 if the requested parameter does not exist.

This function imitates the behavior of the `JWaveExecute.getReturnData` method in the Java client application.

### Example

See WRAPPER_TEST_INIT.

### See Also

WRAPPER_TEST_EXECUTE, WRAPPER_TEST_INIT, WRAPPER_TEST_RETURN_INFO

## WRAPPER_TEST_INIT Procedure

Initializes a JWAVE wrapper function test.

### Usage

WRAPPER_TEST_INIT, *wrapper_name* [, *width*, *height* ]

### Input Parameters

***wrapper_name*** — A string containing the name of the wrapper function that you wish to test.

***width***, ***height*** — (optional) Specifies the width and height of the graphics window, in pixels.

### Discussion

Before using the other WRAPPER_TEST_* routines, you must execute WRAPPER_TEST_INIT.

If *width* and *height* are not set, then WRAPPER_TEST_EXECUTE does not display a plot, even if the JWAVE wrapper returns a graphic. If *width* and *height* are set, a PV‑WAVE window appears even if the JWAVE wrapper does not return any graphics.

### Examples

The following lines demonstrate the use of the WRAPPER_TEST_* commands to test a JWAVE wrapper called `testplot.pro`. You can find this wrapper in:

**(UNIX)** `VNI_DIR/jwave-3_5/lib/user`

**(Windows)** `VNI_DIR\jwave-3_5\lib\user`

where `VNI_DIR` is the main Visual Numerics installation directory.

```
WAVE> WRAPPER_TEST_INIT, 'TESTPLOT', 400, 300

WAVE> WRAPPER_TEST_SETCOLOR, 'BACKGROUND', '919191'xL

WAVE> WRAPPER_TEST_SETCOLOR, 'LINE', 'ff0000'xL

WAVE> WRAPPER_TEST_SETCOLOR, 'SYMBOLS', $

  ['ff00ff'xL, '00ffff'xL, 'ffff00'xL], /Color_Set

WAVE> WRAPPER_TEST_SETPARAM, 'DATA', HANNING(20)
WAVE> WRAPPER_TEST_SETPARAM, 'SYMBOL', 1
WAVE> WRAPPER_TEST_EXECUTE
```

(At this point, the returned graphic appears in a PV-WAVE graphics window.)

```
WAVE> WRAPPER_TEST_RETURN_INFO
      DATA            INT       =         0
```

The following lines demonstrate, additionally, the use of the function WRAPPER_TEST_GETRETURN. Here, the `simple.pro` wrapper is used. This wrapper returns the square root of the input parameter. You can find this procedure in the same directory as `testplot.pro`, described previously. This wrapper only returns a numerical result and not graphics.

```
WAVE> WRAPPER_TEST_INIT, 'SIMPLE'
WAVE> WRAPPER_TEST_SETPARAM, 'NUMBER', 2
WAVE> WRAPPER_TEST_EXECUTE
WAVE> PRINT, WRAPPER_TEST_GETRETURN('DATA')
        1.41421
WAVE> WRAPPER_TEST_RETURN_INFO
      DATA            FLOAT     =       1.41421
```

## See Also

WRAPPER_TEST_EXECUTE, WRAPPER_TEST_RETURN_INFO, WRAPPER_TEST_SETCOLOR, WRAPPER_TEST_SETPARAM

## *WRAPPER_TEST_RETURN_INFO Procedure*

Prints information on all returned values after WRAPPER_TEST_EXECUTE is called.

### Usage

WRAPPER_TEST_RETURN_INFO

### Parameters

None.

### Discussion

This procedure prints the same information as the PV-WAVE INFO command.

This function imitates the behavior of the `Parameter.printInfo` method in a Java client application.

### Example

See WRAPPER_TEST_INIT.

### See Also

WRAPPER_TEST_EXECUTE, WRAPPER_TEST_GETRETURN, WRAPPER_TEST_INIT

## *WRAPPER_TEST_SETCOLOR Procedure*

Sets a named color to be used by a JWAVE wrapper function.

### Usage

WRAPPER_TEST_SETCOLOR, *color_name*, *rgb*

### Input Parameters

***color_name*** — A string specifying the name of the color to set.

***rgb*** — A long integer (RGB value) specifying the color value to set. For example, black is represented by the long value: '000000'xL.

### Keywords

***Color_Set*** — If set, the procedure sets a named array of RGB values. In this case, the *rgb* parameter must specify an array of long integers.

### Discussion

The JWAVE wrapper receives this parameter with the GET_NAMED_COLOR function. Run this procedure before running WRAPPER_TEST_EXECUTE.

This function imitates the behavior of the `JWaveView.setNamedColor` method in a Java client application.

The *Color_Set* keyword allows this procedure to imitate the behavior of the `JWaveView.setNamedColor` method in a Java client application.

### Example

See WRAPPER_TEST_INIT.

### See Also

GET_NAMED_COLOR, WRAPPER_TEST_EXECUTE,
WRAPPER_TEST_INIT

## *WRAPPER_TEST_SETPARAM Procedure*

Sets a parameter to be used as input to a JWAVE wrapper function.

### Usage

WRAPPER_TEST_SETPARAM, *param_name*, *val*

### Input Parameters

***param_name*** — A string containing the name of the parameter to set.

***val*** — The value to be associated with the parameter.

### Discussion

The JWAVE wrapper receives this parameter with the GETPARAM function. Run this procedure before running WRAPPER_TEST_EXECUTE.

This function imitates the behavior of the `JWaveExecute.setParam` method in a Java client application.

### Example

See WRAPPER_TEST_INIT.

### See Also

GETPARAM, WRAPPER_TEST_EXECUTE, WRAPPER_TEST_INIT

# JWAVE Convenience Wrappers

This appendix describes a set of JWAVE wrappers that are provided by Visual Numerics. These wrappers are provided primarily to give JWAVE client developers a way to access the basic PV-WAVE graphics routines without having to write their own JWAVE wrappers.

The JWAVE wrappers described in this appendix include:

- JWAVE_BAR3D Function — Produces a 3D bar chart.
- JWAVE_CONTOUR Function — Produces a contour plot.
- JWAVE_HISTOGRAM Function — Produces a histogram plot.
- JWAVE_LOADCT Procedure — Loads a specified color table.
- JWAVE_PIE Function — Produces a pie chart.
- JWAVE_PLOT Function — Produces 2D plots.
- JWAVE_SURFACE Function — Produces surface plots.

You can use these wrappers just like any JWAVE wrappers. This appendix tells you specifically which parameters the wrappers can accept.

For example, to use the JWAVE_PLOT wrapper, set the JWAVE wrapper function to `JWAVE_PLOT` in the client application:

```
JWaveConnection(”JWAVE_PLOT”)
```

Then, use the `setParam` method to set the parameters you wish to pass to the wrapper. For example:

```
setParam(”X”, an_array);

setParam(”TITLE”, ”The Plot Title”);
```

## JWAVE_BAR3D Function

This JWAVE wrapper function provides a convenient interface to the PV‑WAVE BAR3D procedure. This function always returns a `Viewable` object (a 3D bar chart) to the JWAVE client. Its parameters let you control many aspects of a plot's appearance.

### Parameters

This section lists the parameters that the JWAVE_BAR3D wrapper can retrieve, unpack, and use to produce a 3D bar chart. These parameters correspond to the parameters and keywords of the PV‑WAVE BAR3D procedure. You must set these parameters in the client application with the `JWaveView.setParam` method.

***Z*** — (required) A 2D numeric array containing elevation values.

#### *Keyword Parameters*

This section lists the keyword parameters that can be retrieved and unpacked in this wrapper function. For detailed information on these keywords, refer to Appendix C, *Keyword and Named Color Parameters*.

| | | |
|---|---|---|
| Ax | Ticklen | [XYZ]Style |
| Az | Title | [XYZ]Ticklen |
| Charsize | [XYZ]Charsize | [XYZ]Tickname |
| Charthick | [XYZ]Gridstyle | [XYZ]Ticks |
| Gridstyle | [XYZ]Margin | [XYZ]Title |
| Position | [XYZ]Minor | ZAxis |
| Subtitle | [XYZ]Range | |

## Named Color Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColor` method.

| | |
|---|---|
| Background | Color |

## Named ColorSet Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColorSet` method.

| | |
|---|---|
| ColumnColors | RowColors |

## Returns

This wrapper returns a `Viewable` object (a 3D bar chart) to the JWAVE Java client application.

## Example

In the client Java application, the `setParam` method is used to set the parameters to be passed to this JWAVE wrapper on the server. JWAVE_BAR3D unpacks the parameters and builds a PV-WAVE BAR3D command to produce a 3D bar chart.

These lines of Java code establish a connection to the server with the JWAVE wrapper JWAVE_BAR3D. Then, a data parameter and two configuration parameters are set.

```
JWaveView myJWaveView = new JWaveView(connection, ”JWAVE_BAR3D”)
myJWaveView.JWaveView.setParam(”Z”, threed);
myJWaveView.JWaveView.setParam(”CHARSIZE”, 2);
myJWaveView.JWaveView.setParam(”TITLE”, ”Snow Depth”);
myJWaveView.setNamedColor(”BACKGROUND”, java.awt.Color.blue);
```

## See Also

For more information on using JWAVE wrapper functions, see Chapter 5, *JWAVE Server Development*.

## *JWAVE_CONTOUR Function*

This JWAVE wrapper function provides a convenient interface to the PV‑WAVE CONTOUR procedure. This function always returns a `Viewable` object (a contour plot) to the JWAVE client.

### Parameters

This section lists the parameters that the JWAVE_CONTOUR wrapper can retrieve, unpack, and use to produce a contour plot. These parameters correspond to the parameters and keywords of the PV‑WAVE CONTOUR procedure. You must set these parameters in the client application with the `JWaveView.setParam` method.

***Z*** — (required) A 2D array containing the values that make up the contour surface.

***X*** — A 1D or 2D array specifying the *x*-coordinates for the contour surface.

***Y*** — A 1D or 2D array specifying the *y*-coordinates for the contour surface.

#### *Keyword Parameters*

This section lists the keyword parameters that can be retrieved and unpacked in this wrapper function.

***Nx***, ***Ny*** — If set to 1, *Z* (and *X* and *Y*) are interpolated from their given size into these dimensions. If set to 0 (the default), the data is not interpolated.

***Filled*** — If set to 1, the space between contours is filled with color. If set to 0 (the default), contours are not filled.

For detailed information on the keywords listed in the following table, refer to Appendix C, *Keyword and Named Color Parameters*.

| | | |
|---|---|---|
| Charsize | Max_Value | [XY]Margin |
| Charthick | NLevels | [XY]Minor |
| Clip | Noclip | [XY]Range |
| C_Annotation | Position | [XY]Style |
| C_Charsize | Spline | [XY]Tickformat |
| C_Labels | Subtitle | [XY]Ticklen |
| C_Linestyle | Thick | [XY]Tickname |
| C_Thick | Tickformat | [XY]Ticks |

|  |  |  |
|---|---|---|
| Follow | Ticklen | [XY]Tickv |
| Font | Title | [XY]Title |
| Gridstyle | [XY]Charsize | [XY]Type |
| Levels | [XY]Gridstyle |  |

## Named Color Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColor` method.

|  |  |
|---|---|
| Background | Color |

## Named ColorSet Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColorSet` method.

|  |  |
|---|---|
| C_Colors | Fill_Colors |

## Returns

This wrapper returns a `Viewable` object (of a contour plot) to the JWAVE Java client application.

## Example

In the client Java application, the `setParam` method is used to set the parameters to be passed to this JWAVE wrapper on the server. JWAVE_CONTOUR unpacks the parameters and builds a PV-WAVE CONTOUR command to produce a contour plot.

These lines of Java code set the name of the wrapper function and some parameters. These lines of code would appear in the JWAVE client application.

```
JWaveView myJWaveView = new JWaveView(connection, ”JWAVE_CONTOUR”)
myJWaveView.JWaveView.setParam(”X”, elev_data);
myJWaveView.JWaveView.setParam(”CHARSIZE”, 2);
myJWaveView.JWaveView.setParam(”TITLE”, ”Boulder”);
myJWaveView.setNamedColor(”BACKGROUND”, java.awt.Color.blue);
```

### See Also

For more information on using JWAVE wrapper functions, see Chapter 5, *JWAVE Server Development*.

For more information on the PV‑WAVE CONTOUR procedure, see the *PV‑WAVE Reference*.

## *JWAVE_HISTOGRAM Function*

This JWAVE wrapper function provides a convenient interface to the PV‑WAVE HISTOGRAM procedure. This function always returns a `Viewable` object (a histogram plot) to the JWAVE Java client.

### Parameters

This section lists the parameters that the JWAVE_HISTOGRAM wrapper can retrieve, unpack, and use to produce a histogram plot. These parameters correspond to the parameters and keywords of the PV‑WAVE HISTOGRAM procedure. You must set these parameters in the client application with the `JWaveView.setParam` method.

***Y*** — (required) The array for which the density function will be computed. The size of each dimension of *Y* may be any integer value.

#### *Keyword Parameters*

This section lists the keyword parameters that can be retrieved and unpacked in this wrapper function.

***Axiscolor*** — (integer) Specifies the index of the axis color.

***Binsize*** — Specifies the width of the bins displayed in the histogram. (Default: 1)

***Fillcolor*** — (integer) Specifies the index of the color used to fill the histogram. (Default: *Color*)

***Filled*** — If present and nonzero, the histogram is filled with color. (Default: 0)

***Stepped*** — If present and nonzero, the histogram is plotted as “steps” rather than as “bars”. (Default: 0)

***Xmax*** — The maximum value for which histogram data is plotted. Any data that falls above this value will be clipped.

***Xmin*** — The minimum value for which histogram data is plotted. This corresponds to the leftmost point on the *x*-axis where the plot begins. By default, this minimum is set to zero. If there are negative values in your histogram data, you may need to adjust this value to shift the data to the left. Otherwise, the plot starts at the origin.

For detailed information on the keywords listed in the following table, refer to Appendix C, *Keyword and Named Color Parameters*.

| | | |
|---|---|---|
| Clip | Thick | [XY]Ticklen |
| Nodata | Title | [XY]Title |
| Noerase | [XY]Range | [XY]Type |
| Position | [XY]Style | |

## Named Color Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColor` method.

| | |
|---|---|
| Background | Color |

## Returns

This wrapper returns a `Viewable` object (of a histogram plot) to the JWAVE Java client application.

## Example

In the client Java application, the `setParam` method is used to set the parameters to be passed to this JWAVE wrapper on the server. JWAVE_HISTOGRAM unpacks the parameters and builds a PV‑WAVE HISTOGRAM command to produce a histogram plot.

These lines of Java code set the name of the wrapper function and some parameters. These lines of code would appear in the JWAVE client application.

```
JWaveView myJWaveView = new JWaveView(connection,
   "JWAVE_HISTOGRAM")

myJWaveView.JWaveView.setParam("X", histdata);

myJWaveView.JWaveView.setParam("CHARSIZE", 2);

myJWaveView.JWaveView.setParam("TITLE", "CO2 Content");

myJWaveView.setNamedColor("BACKGROUND", java.awt.Color.blue);
```

### See Also

For more information on using JWAVE wrapper functions, see Chapter 5, *JWAVE Server Development*.

For more information on the PV‑WAVE HISTOGRAM procedure, see the *PV‑WAVE Reference*.

## *JWAVE_LOADCT Procedure*

Loads a predefined color table.

### Usage

JWAVE_LOADCT, *table_num*

### Input Parameters

***table_number*** — A number between 0 and 16; each number is associated with a predefined color table. You must set this parameter in the client application with the `JWaveView.setParam` method.

### Input Keywords

***Silent*** — If nonzero, suppresses the message indicating that the color table is being loaded.

### Output Keywords

***Range_Of_Colors*** — Retrieves the a 2-element array containing the range of colors that are available for use by images. The first element represents the first color in the range, and the second element represents the last color. This range is equivalent to the number of colors in the color table minus the number of named colors that have been retrieved with GET_NAMED_COLORS. See the *Discussion* for information on how this keyword is used.

### Discussion

The color tables associated with JWAVE wrappers are subsetted into two parts. First, all of the colors retrieved by GET_NAMED_COLORS are stored in the color table. Then, when JWAVE_LOADCT is called, the remaining color table positions are filled with colors from the specified color table.

**TIP** Call GET_NAMED_COLOR before calling JWAVE_LOADCT. This ensures that the colors retrieved by GET_NAMED_COLOR will be stored in the color table. The JWAVE_LOADCT procedure stores colors in the remaining color table positions.

Predefined color tables are stored in the file `colors.tbl`. There are 17 predefined color tables, with indices ranging from 0 to 16, as shown in the following table.

| Number | Name |
|---|---|
| 0 | Black and White Linear |
| 1 | Blue/White |
| 2 | Green/Red/Blue/White |
| 3 | Red Temperature |
| 4 | Blue/Green/Red/Yellow |
| 5 | Standard Gamma-II |
| 6 | Prism |
| 7 | Red/Purple |
| 8 | Green/White Linear |
| 9 | Green/White Exponential |
| 10 | Green/Pink |
| 11 | Blue/Red |
| 12 | 16 Level |
| 13 | 16 Level II |
| 14 | Steps |
| 15 | PV-WAVE Special |
| 16 | Black and White Reversed |

**NOTE** Values returned by the *Range_Of_Colors* keyword in previous calls to GET_NAMED_COLOR may no longer be valid. Use the value of the *Range_Of_Colors* keyword from the *last* call to either GET_NAMED_COLOR or JWAVE_LOADCT.

## Examples

This example demonstrates the use of the *Range_Of_Colors* keyword. The color table range returned by *Range_Of_Colors* is used in a BYTSCL call to “smooth” the image portion of the color table.

```
; Retrieve a color. This color is stored in the color table.
lc = GET_NAMED_COLOR('LINE', DefaultRGB='FFFFFF'xL)
; Load a color table. These colors are loaded into the remaining
; positions of the color table.
JWAVE_LOADCT, 15, Range_Of_Colors=range
; Byte scale the range of colors that were stored in the ”image”
   portion
; of the color table.
TV, BYTSCL(my_image, Top=range(1)-range(0)) + range(0)
; Make a plot using the color retrieved by GET_NAMED_COLOR.
PLOTS, overlay_data, Color=lc
```

## See Also

GET_NAMED_COLOR

In the *PV-WAVE Reference*:

LOADCT

## *JWAVE_PIE Function*

This JWAVE wrapper function provides a convenient interface to the PV-WAVE PIE_CHART procedure.This function always returns a `Viewable` object (a pie chart) to the JWAVE Java client.

### Parameters

This section lists the parameters that the JWAVE_PIE wrapper can retrieve, unpack, and use to produce a pie chart. These parameters correspond to the parameters and keywords of the PV-WAVE PIE_CHART2 procedure. You must set these parameters in the client application with the `JWaveView.setParam` method.

***Y*** — (required) A 1D array of values to plot (30 maximum).

#### *Keyword Parameters*

This section lists the keyword parameters that can be retrieved and unpacked in this wrapper function.

***Charsize*** — Relative character size. (Default: 1.0)

***Explode*** — Explode (move out from center). Normalized displacement of each slice from the center (between 0 and 1). Must be an array with the same length as X, if supplied. (Default: 0 for all slices)

***Label*** — Text labels for each slice. If specified, must be an array of strings the same length as X.

***Radius*** — Normalized radius of the pie chart. Between 0 and 0.5. (Default: 0.3)

***Tborder*** — Draw borders around the labels. A boolean 0 or 1. (Default: 0)

***Tperct*** — Add notation of percentage of each slice to the label. A boolean 0 or 1. (Default: 0)

***Tvalue*** — Add notation of value of each slice to the label. A boolean 0 or 1. (Default: 0)

***Tposition*** — Label positions: 0 = Internal to the slice, 1 = External, 2 = External Aligned.

***Xcenter***, ***Ycenter*** — Normalized position of the center of the chart. Between 0 AND 1. (Default: 0.5, 0.5)

***Shade*** — Draw a shadow under the chart. Normalized displacement of the shadow from the center of the chart (between 0 and 1). (Default: no shadow)

## Named Color Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColor` method.

***Background*** — The background color. (Default: black)

***Color*** — Color for lines (pie outline) and title text. (Default: white)

***Tbord_Color*** — Color for the label border (if `Tborder` is set). (Default: the value of the *Color* keyword)

## Named ColorSet Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColorSet` method.

***Slices*** — Color for each slice. If specified, must have the same number of colors as there are data points in X. (Default: shades of gray)

***Tcolor*** — Color for the label of each slice. If specified, must have the same number of colors as there are data points in X. (Default: slices)

## Returns

This wrapper returns a `Viewable` object (of a pie chart) to the JWAVE Java client application.

## Example

In the client Java application, the `setParam` method is used to set the parameters to be passed to this JWAVE wrapper on the server. JWAVE_PIE unpacks the parameters and builds a PV-WAVE PIE_CHART command to produce a pie chart.

These lines of Java code set the name of the wrapper function and some parameters. These lines of code would appear in the JWAVE client application.

```
JWaveView myJWaveView = new JWaveView(connection, ”JWAVE_PIE”)

myJWaveView.JWaveView.setParam(”Y”, piedata);

myJWaveView.JWaveView.setParam(”CHARSIZE”, 2);

myJWaveView.JWaveView.setParam(”TITLE”, ”Relative Weights”);

myJWaveView.setNamedColor(”BACKGROUND”, java.awt.Color.blue);
```

### See Also

For more information on using JWAVE wrapper functions, see Chapter 5, *JWAVE Server Development*.

For more information on the PV-WAVE PIE_CHART procedure, see the *PV-WAVE Reference*.

## *JWAVE_PLOT Function*

This JWAVE wrapper function provides a convenient interface to the PV-WAVE PLOT procedure. This function always returns a `Viewable` object (a 2D plot) to the JWAVE Java client.

### Parameters

This section lists the parameters that the JWAVE_PLOT wrapper can retrieve, unpack, and use to produce a 2D plot. These parameters correspond to the parameters and keywords of the PV-WAVE PLOT procedure. You must set these parameters in the client application with the `JWaveView.setParam` method.

***Y*** — (required) A 1D array. If only this parameter is supplied, *Y* is plotted on the vertical axis as a function of the number of points. In other words, unit spacing is assumed along the horizontal axis.

***X*** — A 1D array. If both positional parameters are supplied, the first variable is the independent variable, and it is plotted along the horizontal axis. The second variable is the dependent variable, and it is plotted along the vertical axis (as a function of the independent variable).

***Yn*** — You can produce overlays of up to nine additional plot lines by specifying parameters [ , *Y1*, *Y2*, ... *Y9* ].

***Xn*** — You can produce overlays of additional plot lines by specifying parameters [ , *X1*, *X2*, ... *X9* ] for the dependent variables. These are only used if their corresponding *Yn* value is set. If a *Yn* is set and an *Xn* is not set, then the value for *X* is used (or unit spacing if *X* is not set).

***Scaling*** — Specifies the type of axis scaling for the plot.

| | |
|---|---|
| 0 | Produces a simple XY plot with linear axes. Corresponds to the PLOT procedure. (Default) |
| 1 | Produces an XY plot with logarithmic scaling on the *y*-axis and linear scaling on the *x*-axis. Corresponds to the PLOT_IO procedure. |
| 2 | Produces an XY plot with linear scaling on the *y*-axis and logarithmic scaling on the *x*-axis. Corresponds to the PLOT_OI procedure. |
| 3 | Produces an XY plot with logarithmic scaling on both the *x*-axis and the *y*-axis. Corresponds to the PLOT_OO procedure. |

## *Keyword Parameters*

This section lists the keyword parameters that can be retrieved and unpacked in this wrapper function. For detailed information on these keywords, refer to *Appendix C, Keyword and Named Color Parameters*.

| | | |
|---|---|---|
| Box | * Solid_Psym | [XY]Range |
| Charsize | Subtitle | [XY]Style |
| Charthick | * Symsize | [XY]Tickformat |
| * Clip | * Thick | [XY]Ticklen |
| Gridstyle | Tickformat | [XY]Tickname |
| * Linestyle | Ticklen | [XY]Ticks |
| * Noclip | Title | [XY]Tickv |
| * Nsum | [XY]Charsize | [XY]Title |
| * Polar | [XY]Gridstyle | [XY]Type |
| Position | [XY]Margin | YNozero |
| * Psym | [XY]Minor | |

* These parameters can be used to specify the properties of specific overlay plot lines. For example, if the overlay parameter `Y1` is specified, then `Psym1` sets the plot symbols for that particular data. In other words, by appending a number {1..9} after these particular keywords, you associate the keyword with data having the same suffix. Note that if `Y1` is specified, but `Psym1` is not specified, then `Y1` is plotted using the value of `Psym` (no suffix) or its default.

## Named Color Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColor` method.

| | | |
|---|---|---|
| Axis | Background | Color |

## Returns

This wrapper returns a `Viewable` object (of a 2D plot) to the JWAVE Java client application.

## Example

In the client Java application, the `setParam` method is used to set the parameters to be passed to this JWAVE wrapper on the server. JWAVE_PLOT unpacks the parameters and builds a PV-WAVE PLOT command to produce a 2D plot.

These lines of Java code set the name of the wrapper function and some parameters. These lines of code would appear in the JWAVE client application.

```
JWaveView myJWaveView = new JWaveView(connection, ”JWAVE_PLOT”)

myJWaveView.setParam(”Y”, myArray);

myJWaveView.setParam(”CHARSIZE”, 2);

myJWaveView.setParam(”TITLE”, ”CO2 Content”);

myJWaveView.setNamedColor(”BACKGROUND”, java.awt.Color.blue);
```

`Y` is a positional parameter; `CHARSIZE` and `TITLE` are keyword parameters. The JWAVE_PLOT wrapper function knows how to retrieve these parameters, construct a 2D plot, and return a `Viewable` object to the client.

## See Also

For more information on using JWAVE wrapper functions, see Chapter 5, *JWAVE Server Development*.

For more information on the PV-WAVE PLOT procedure, see the *PV-WAVE Reference*.

## *JWAVE_SURFACE Function*

This JWAVE wrapper function provides a convenient interface to the PV-WAVE SURFACE procedure. This function always returns a `Viewable` object (a surface plot) to the JWAVE Java client.

### Parameters

This section lists the parameters that the JWAVE_SURFACE wrapper can retrieve, unpack, and use to produce a surface plot. These parameters correspond to the parameters and keywords of the PV-WAVE SURFACE procedure. You must set these parameters in the client application with the `JWaveView.setParam` method.

***Z*** — (required) A 2D array containing the values that make up the surface. If *X* and *Y* are supplied, the surface is plotted as a function of the X,Y locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of *Z*.

***X*** — A 1D or 2D array specifying the *x*-coordinates for the surface.

- If *X* is a 1D array, each element of *X* specifies the *x*-coordinate for a column of *Z*. For example, *x*(0) specifies the *x*-coordinate for z(0, *).
- If *X* is a 2D array, each element of *X* specifies the *x*-coordinate of the corresponding point in *z* ($x_{ij}$ specifies the *x*-coordinate for $z_{ij}$).

***Y*** — A 1D or 2D array specifying the *y*-coordinates for the surface.

- If *Y* is a 1D array, each element of *y* specifies the *y*-coordinate for a row of *Z*. For example, *y*(0) specifies the *y*-coordinate for *z* (*, 0).
- If *Y* is a 2D array, each element of *Y* specifies the *y*-coordinate of the corresponding point in *z* ($y_{ij}$ specifies the *y*-coordinate for $z_{ij}$).

#### *Keyword Parameters*

This section lists the keyword parameters that can be retrieved and unpacked in this wrapper function.

***Ctable*** — A PV-WAVE color table to use for shading, an integer between 0 and 16. See the LOAD_JWAVECT command. (Default: 0 — gray-scale)

***Mesh*** — Boolean, indicating whether to draw a mesh surface (will overlay mesh on shading, if shaded). (Default: 1)

***Nx***, ***Ny*** — If set, *Z* (and *X*, *Y*, and *Shades*) are interpolated from their given size into these dimensions.

***Shaded*** — Boolean, indicating whether to shade the surface with light source. See also *Ctable* and *Shaded*. (Default: 0)

***Shades*** — An array expression, of the same dimensions as *z*, containing the color index at each point. The shading of each pixel is interpolated from the surrounding *Shades* values. For most displays, this parameter should be scaled into the range of bytes. If this keyword is omitted, light source shading is used. Will be scaled to fit in the range of colors used by `Ctable`. (Default: no shading)

For detailed information on the keywords listed in the following table, refer to Appendix C, *Keyword and Named Color Parameters*.

| | | |
|---|---|---|
| Ax | Subtitle | [XYZ]Range |
| Az | Thick | [XYZ]Style |
| Charsize | Tickformat | [XYZ]Tickformat |
| Charthick | Ticklen | [XYZ]Ticklen |
| Clip | Title | [XYZ]Tickname |
| Gridstyle | [XYZ]Charsize | [XYZ]Ticks |
| Noclip | [XYZ]Gridstyle | [XYZ]Tickv |
| Position | [XYZ]Margin | [XYZ]Title |
| Skirt | [XYZ]Minor | ZAxis |

## Named Color Parameters

These parameters must be set by the JWAVE client Java application with the `JWaveView.setNamedColor` method.

| | | |
|---|---|---|
| Background | Bottom | Color |

## Returns

This wrapper returns a `Viewable` object (of a surface plot) to the JWAVE Java client application.

## Example

In the client Java application, the `setParam` method is used to set the parameters to be passed to this JWAVE wrapper on the server. JWAVE_SURFACE unpacks the parameters and builds a PV-WAVE SURFACE command to produce a surface plot.

These lines of Java code set the name of the wrapper function and some parameters. These lines of code would appear in the JWAVE client application.

```
JWaveView myJWaveView = new JWaveView(connection, ”JWAVE_SURFACE”)

myJWaveView.JWaveView.setParam(”X”, elev_data);

myJWaveView.JWaveView.setParam(”CHARSIZE”, 2);

myJWaveView.JWaveView.setParam(”TITLE”, ”Snow Depth”);

myJWaveView.setNamedColor(”BACKGROUND”, java.awt.Color.blue);
```

## See Also

For more information on using JWAVE wrapper functions, see Chapter 5, *JWAVE Server Development*.

For more information on the PV-WAVE SURFACE procedure, see the *PV-WAVE Reference*.

C

APPENDIX

# Keyword and Named Color Parameters

Three categories of parameters described in this appendix:

- *Keyword Parameters* on page C-2
- *Named Color Parameters* on page C-19
- *Named ColorSet Parameters* on page C-20

## Using These Parameters

Visual Numerics has provided, for your convenience, a set of JWAVE wrapper functions for use in graphics applications. These wrapper functions can be called from a JWAVE Java client application to generate most of the types of plots that are available in PV-WAVE. These JWAVE wrappers include:

- JWAVE_BAR3D Function — Produces a 3D bar chart.
- JWAVE_CONTOUR Function — Produces a contour plot.
- JWAVE_HISTOGRAM Function — Produces a histogram plot.
- JWAVE_PIE Function — Produces a pie chart.
- JWAVE_PLOT Function — Produces 2D plots.
- JWAVE_SURFACE Function — Produces surface plots.

They are described in Appendix B, *JWAVE Convenience Wrappers*.

This appendix describes the parameters that you can set in the Java client application for use by these JWAVE wrapper functions. In the Java client application, you set these parameters using methods such as:

```
JWaveExecute.setParam
```

```
JWaveView.setNamedColor
```

```
JwaveView.setNamedColorSet
```

For example, the following method sets a color parameter called *Background* in the Java client application:

```
myJWaveView.setNamedColor("BACKGROUND", Java.awt.Color.yellow)
```

Then, in the JWAVE wrapper function, this color is retrieved with:

```
back = GET_NAMED_COLOR("BACKGROUND", Default='000000'xL)
```

The *Background* parameter is commonly used to set the background color for plots. It is described on page C-19.

## Keyword Parameters

**NOTE** *Named Color Parameters* are listed on page C-19. *Named ColorSet Parameters* are listed on page C-20.

### Ax

**Used With:** JWAVE_BAR3D, JWAVE_SURFACE

Specifies the angle of rotation about the *x*-axis, in degrees, towards the viewer. The default is +30 degrees.

The surface represented by the 2D array is first rotated, *Az* (see the *Az* keyword) degrees about the *z*-axis, then by *Ax* degrees about the *x*-axis, tilting the surface towards the viewer (*Ax* > 0), or away from the viewer.

**NOTE** This keyword is effective only if the PV-WAVE !P.T3d system variable is *not* set. If !P.T3d is set, the 3D to 2D transformation used by is contained in the 4-by-4 array !P.T. Refer to the *PV-WAVE Reference* for information on system variables.

## Az

**Used With:** JWAVE_BAR3D, JWAVE_SURFACE

Specifies the counterclockwise angle in degrees of rotation about the *z*-axis (when looking down the *z*-axis toward the origin). The order of rotation is *Az* first, then *Ax*.

**NOTE** This keyword is effective only if the PV‑WAVE system variable !P.T3d is *not* set. Refer to the *PV‑WAVE Reference* for information on system variables.

## Box

**Used With:** JWAVE_PLOT

Places a box around the labels in a Date/Time axis. If you set the keyword to a value of 1, boxes are drawn around all the labels of the Date/Time axis.

By default, no boxes are drawn. For information on Date/Time axes, refer to the *PV‑WAVE User's Guide*.

## C_Annotation

**Used With:** JWAVE_CONTOUR

Sets the label that will be drawn on each contour.

Usually, contours are labeled with their value. This parameter, an array of strings, allows any text to be specified. The first label is used for the first contour drawn, and so forth. If *Levels* is specified, the elements of *C_Annotation* correspond directly to the levels specified, otherwise, they correspond to the default levels chosen by the PV‑WAVE CONTOUR procedure. If there are more contour levels than elements in *C_Annotation*, the remaining levels are labeled with their values.

Use of *C_Annotation* implies use of the *Follow* keyword.

## C_Charsize

**Used With:** JWAVE_CONTOUR

Sets the size of the characters used to annotate contour labels.

Normally, contour labels are drawn at three-fourths the size used for the axis labels (specified by the *Charsize* keyword or the !P.Charsize system variable in PV‑WAVE). This keyword allows the contour label size to be specified independently. Use of this keyword implies use of the *Follow* keyword.

## Charsize

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PIE, JWAVE_PLOT, JWAVE_SURFACE

Sets the overall character size for the annotation. A *Charsize* of 1.0 is normal. The size of the annotation on the axes may be set, relative to *Charsize*, with *XCharsize*, *YCharsize*, and *ZCharsize*. The main title is written with a character size of 1.25 times this parameter.

## Charthick

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Sets the thickness of characters drawn with the software fonts. Normal thickness is 1.0, double thickness is 2.0, and so on. (If this keyword is omitted, the value of the PV-WAVE system variable !P.Charthick is used.)

## C_Labels

**Used With:** JWAVE_CONTOUR

Specifies which contour levels should be labeled. By default, every other contour level is labeled.

*C_Labels* allows you to override this default and explicitly specify the levels to label. This parameter is an array, converted to integer type if necessary. If the *Levels* keyword is specified, the elements of *C_Labels* correspond directly to the levels specified, otherwise, they correspond to the default levels chosen by the PV-WAVE CONTOUR procedure. Setting an element of the array to zero causes that contour level to not be labeled. A nonzero value forces labeling.

Use of this keyword implies use of the *Follow* keyword.

## C_Linestyle

**Used With:** JWAVE_CONTOUR

Specifies the linestyle used to draw each contour.

As with *C_Colors*, *C_Linestyle* is an array of linestyle indices. If there are more contour levels than linestyles, the linestyles are cyclically repeated. The following table lists the available linestyles and their keyword indices:

| Index | X Windows Style | Windows Style |
|---|---|---|
| 0 | Solid | Solid |
| 1 | Dotted | Short dashes |
| 2 | Dashed | Long dashes |
| 3 | Dash dot | Long-short dashes |
| 4 | Dash-dot-dot-dot | Long-short-short dashes |
| 5 | Long dashes | Long dashes |

**NOTE** The current contouring algorithm draws all the contours in each cell, rather than following contours. Hence, some of the more complicated linestyles will not be suitable for some applications.

## Clip

**Used With:** JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

Specifies the coordinates of a rectangle used to clip the graphics output. Graphics that fall inside the rectangle are displayed; graphics that fall outside the clipping rectangle are not displayed.

The rectangle is specified as an array of the form [$X_0$, $Y_0$, $X_1$, $Y_1$], giving data coordinates of the lower-left and upper-right corners, respectively.

## C_Thick

**Used With:** JWAVE_CONTOUR

Specifies the line thickness of lines used to draw each contour level. As with *C_Colors*, *C_Thick* is an array of line thickness values, although the values are floating-point. If there are more contours than thickness elements, elements are repeated. If omitted, the overall line thickness specified by the *Thick* keyword parameter or the PV-WAVE system variable !P.Thick is used for all contours.

## Follow

**Used With:** JWAVE_CONTOUR

If set to a nonzero value, forces the PV-WAVE CONTOUR procedure to use the line-following method instead of the cell-drawing method.

CONTOUR can draw contours using one of two different methods:

- The cell-drawing method, used by default, examines each array cell and draws all contours emanating from that cell before proceeding to the next cell. This method is efficient in terms of computer resources but does not allow contour labeling.
- The line-following method searches for each contour line and then follows the line until it reaches a boundary or closes. This method gives better looking results with dashed linestyles, and allows contour labeling, but requires more computer time. It is used if any of the following keywords is specified: *C_Annotation*, *C_Charsize*, *C_Labels*, *Follow*, or *Path_Filename*.

Although these two methods both draw correct contour maps, differences in their algorithms can cause small differences in the resulting plot.

## Font

**Used With:** JWAVE_CONTOUR

An integer that specifies the graphics text font index.

- Font index –1 selects the software fonts, which are drawn using vectors.
- Font number 0 selects the hardware font of the output device.

**NOTE** Hardware font drivers that support 3D transformations include X Windows, WIN32 (on Windows NT platforms only), PostScript, and WMF (on Windows NT platforms only).

## Gridstyle

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Lets you change the linestyle of tick intervals.

The default is a solid line. Other linestyle choices and their index values are listed in the following table:

| Index | X Windows Style | Windows Style |
|---|---|---|
| 0 | Solid (default) | Solid (default) |
| 1 | Dotted | Short dashes |
| 2 | Dashed | Long dashes |
| 3 | Dash dot | Long-short dashes |
| 4 | Dash-dot-dot-dot | Long-short-short dashes |
| 5 | Long dashes | Long dashes |

One possible use for this keyword is to create an evenly spaced grid consisting of dashed lines across your plot region. To do this, first set the *Ticklen* keyword to 0.5. This ensures that the dashed tick style will appear correctly on your plot. Then set the *Gridstyle* keyword to the style you want to use.

## Levels

**Used With:** JWAVE_CONTOUR

Specifies an array containing the contour levels (maximum of 150) drawn by the PV-WAVE CONTOUR procedure.

A contour is drawn for each level specified in *Levels*. If *Levels* is omitted, the data range is divided into approximately six equally-spaced levels.

## Linestyle

**Used With:** JWAVE_PLOT

Specifies the linestyle used to draw the lines or connect data points.

**UNIX USERS** The line join style is “miter,” that is, the outer edges of two lines extend to meet at an angle.

**Windows USERS** The line join style is “round.”

The linestyle index is an integer, as shown in the following table:

| Index | X Windows Style | Windows Style |
|---|---|---|
| 0 | Solid | Solid |
| 1 | Dotted | Short dashes |
| 2 | Dashed | Long dashes |
| 3 | Dash dot | Long-short dashes |
| 4 | Dash-dot-dot-dot | Long-short-short dashes |
| 5 | Long dashes | Long dashes |

Sets the maximum number of levels on a Date/Time axis. For example, assume that the Date/Time data contains years, months, days, hours, minutes, and seconds. If this keyword is set to three, then the Date/Time axis will show three levels: seconds, minutes, and hours.

## Max_Value

**Used With:** JWAVE_CONTOUR

Data points with values equal to or above this value are ignored when contouring. Cells containing one or more corners with values above *Max_Value* will have no contours drawn through them.

## NLevels

**Used With:** JWAVE_CONTOUR

The number of equally-spaced contour levels that are produced by CONTOUR. The maximum is 150. (Default: 6)

If the *Levels* parameter, which explicitly specifies the value of the contour levels, is present this keyword has no effect. If neither parameter is present approximately six levels are drawn.

If the minimum and maximum Z values are $Z_{min}$ and $Z_{max}$, then the value of the *i*th level is:

$$Z_{min} + (i + 1)(Z_{max} - Z_{min})/(NLevels + 1)$$

where *i* ranges from 0 to *NLevels* – 1.

## Noclip

**Used With:** JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Enforces the default clipping behavior, which is to clip graphics at the boundary of the Plot Data Region (area bounded by the coordinate axes).

## Nodata

**Used With:** JWAVE_HISTOGRAM

If this keyword is set to a nonzero value, only the axes, titles, and annotation are drawn. No data points are plotted.

## Noerase

**Used With:** JWAVE_HISTOGRAM

If set to a nonzero value, specifies that the screen or page is not to be erased. By default the screen is erased, or a new page is begun, before a plot is produced.

## Nsum

**Used With:** JWAVE_PLOT

Indicates the number of data points to average when plotting.

If *Nsum* is larger than 1, every group of *Nsum* points is averaged to produce one plotted point. If there are *m* data points, then *m* / *Nsum* points are displayed. On logarithmic axes a geometric average is performed.

It is convenient to use *Nsum* when there is an extremely large number of data points to plot because it plots fewer points, the graph is less cluttered, and it is quicker.

## Polar

**Used With:** JWAVE_PLOT

Polar plots are produced when this keyword is set to a nonzero value.

The *X* and *Y* parameters, both of which must be present, are first converted from polar to cartesian coordinates. The first parameter is the radius, and the second is θ, expressed in radians.

## Position

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

Allows direct specification of the plot window.

*Position* is a four-element array giving, in order, the coordinates $[(x_0, y_0), (x_1, y_1)]$ of the lower-left and upper-right corners of the data window. Coordinates are expressed in normalized units ranging from 0.0 to 1.0.

When setting the position of the window, be sure to allow space for the annotation, which resides outside the window. PV-WAVE outputs the message

```
%, Warning: Plot truncated.
```

if the plot region is larger than the screen or page size. The plot region is the rectangle enclosing the plot window and the annotation.

When plotting in three dimensions, the *Position* keyword is a six-element array with the first four elements describing, as above, the XY position, and with the last two elements giving the minimum and maximum *z*-coordinates. The Z specification is always in normalized coordinate units.

## Psym

**Used With:** JWAVE_PLOT

Specifies the symbol used to mark each data point.

| !P.Psym Value | Symbol Drawn |
|---|---|
| 0 | No symbol, connect points with solid lines |
| 1 | Plus sign |
| 2 | Asterisk |
| 3 | Period |
| 4 | Diamond |
| 5 | Triangle |
| 6 | Square |
| 7 | X |
| 8 | User-defined, see the USERSYM procedure |
| 9 | Undefined |

| **!P.Psym Value** | **Symbol Drawn** |
|---|---|
| 10 | Data points are plotted in the histogram mode. Horizontal and vertical lines connect the plotted points, as opposed to the normal method of connecting points with straight lines. |
| *–value* | Negative values connect symbols with solid lines |

Normally, *Psym* is 0, data points are connected by lines, and no symbols are drawn to mark the points. Specify this keyword to mark data points with symbols. The keyword *Symsize* is used to set the size of the symbols.

Negative values of *Psym* cause the symbol designated by |*Psym*| to be plotted at each point with solid lines connecting the symbols. For example, a *Psym* value of –5 plots triangles at each data point and connects the points with lines.

The *Psym* keyword can specify an array of plot symbols. If an array is used, each plot symbol value in the array is applied, in order, to create the plot symbols that make up the graph. The symbols are repeated, as needed, to complete the entire graph of the data set.

**NOTE** Forty-one new graphic symbols have been added for PV-WAVE plot routines. These new symbols include:

```
Psym=9
```

```
Psym=11...Psym=41
```

(`Psym=10` is reserved)

See also *Solid_Psym*.

## Skirt

**Used With:** JWAVE_SURFACE

A skirt around the array at a given *z* value is drawn if this keyword parameter is nonzero. The *z* value is expressed in data units.

If the skirt is drawn, each point on the four edges of the surface is connected to a point on the skirt which has the given *z* value, and the same *x* and *y* values as the edge point. In addition, each point on the skirt is connected to its neighbor.

## Solid_Psym

**Used With:** JWAVE_PLOT

If this parameter is set to a nonzero value, symbols are drawn with solid lines no matter which linestyle is used to connect the symbols. By default, symbols are drawn with the currently specified linestyle.

## Spline

**Used With:** JWAVE_CONTOUR

If this parameter is set to a nonzero value, specifies that contour paths are to be interpolated using cubic splines.

Use of this keyword implies the use of the *Follow* keyword. The appearance of contour plots of arrays with low resolution may be improved by using spline interpolation. In rare cases, contour lines that are close together may cross because of interpolation.

Splines are especially useful with small data sets (less than 15 array dimensions). With larger data sets the smoothing is not as noticeable and the expense of splines increases rapidly with the number of data points.

You may specify the length of each interpolated line segment in normalized coordinates by including a value with this keyword. The default value is 0.005 which is obtained when the parameter *Spline* is present. Smaller values for this parameter yield smoother lines, up to the resolution of the output device, at the expense of more computations.

## Subtitle

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Produces a subtitle underneath the *x*-axis containing the text in this string parameter.

## Symsize

**Used With:** JWAVE_PLOT

Specifies the size of the symbols drawn when *Psym* is set. The default size of 1.0 produces symbols approximately the same size as a character.

The *Symsize* keyword can specify an array of symbol sizes. If an array is used, each plot symbol size in the array is applied, in order, to size the plot symbols that make up the graph. The symbol sizes are repeated, as needed, to complete the entire graph of the data set.

## Thick

**Used With:** JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

Controls the thickness of the lines connecting points. A thickness of 1.0 is normal, 2.0 is double-wide, etc.

## Tickformat

**Used With:** JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Lets you use FORTRAN-style format specifiers to change the format of tick labels on the *x*-, *y*-, and *z*-axes.

The resulting plot's tick labels are formatted with a total width of five characters carried to two decimal places. As expected, the width field expands automatically to accommodate larger values.

Note that only the `I` (integer), `F` (floating-point), and `E` (scientific notation) format specifiers can be used with *Tickformat*. Also, you cannot place a quoted string inside a tick format. For example, `("<", F5.2, ">")` is an invalid *Tickformat* specification.

See also *[XYZ]Tickformat*.

## Ticklen

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Controls the length of the axis tick marks, expressed as a fraction of the window size. The default value is 0.02. Ticklen of 0.5 produces a grid, while a negative *Ticklen* makes tick marks that extend outside the plot region, rather than inwards.

## Title

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

Sets a string used for the main title centered above the plot window.

The text size of this main title is larger than the other text by a factor of 1.25.

## XCharsize, YCharsize, ZCharsize

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

The size of the characters used to annotate the *x*-, *y*-, and *z*-axes and their titles.

This field is a scale factor applied to the global scale factor set by the PV-WAVE system variable !P.Charsize or the keyword *Charsize*.

See also *Charsize*.

## XGridstyle, YGridstyle, ZGridstyle

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Lets you change the linestyle of tick intervals on the *x*-, *y*-, and *z*-axes.

The default is a solid line.

| Index | X Windows Style | Windows Style |
|---|---|---|
| 0 | Solid (default) | Solid (default) |
| 1 | Dotted | Short dashes |
| 2 | Dashed | Long dashes |
| 3 | Dash dot | Long-short dashes |
| 4 | Dash-dot-dot-dot | Long-short-short dashes |
| 5 | Long dashes | Long dashes |

See also *Gridstyle*.

## XMargin, YMargin, ZMargin

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

A two-element array specifying the margin around the sides of the plot window, in units of character size. Default margins are 10 (left margin) and 3 (right margin) for the *x*-axis, 4 (bottom margin) and 2 (top margin) for the *y*-axis. For the *z*-axis the default margins are both 0.

## XMinor, YMinor, ZMinor

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

The number of minor tick intervals on the *x*-, *y*-, and *z*-axes. If set to 0, the default, PV-WAVE automatically determines the number of minor ticks in each major tick mark interval. Setting this parameter to –1 suppresses the minor ticks, and setting it to a positive, nonzero number *n* produces *n* minor tick intervals, and *n* – 1 minor tick marks.

## XRange, YRange, ZRange

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

The desired data range of the *x*-, *y*-, and *z*-axes, a two-element array. The first element is the axis minimum, and the second is the maximum. PV-WAVE will frequently round this range.

## XStyle, YStyle, ZStyle

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

Allows specification of axis options such as rounding of tick values and selection of a box axis. Each option is encoded in a bit. See the following table for details:

| Bit | Value | Function |
|---|---|---|
| 0 | 1 | Exact. By default the end points of the axis are rounded in order to obtain even tick increments. Setting this bit inhibits rounding, making the axis fit the data range exactly. |
| 1 | 2 | Extend. If this bit is set, the axes are extended by 5% in each direction, leaving a border around the data. |
| 2 | 4 | None. If this bit is set, the axis and its text is not drawn. |
| 3 | 8 | No box. Normally, JWAVE_PLOT and JWAVE_CONTOUR draw a box style axis with the data window surrounded by axes. Setting this bit inhibits drawing the top or right axis. |
| 4 | 16 | Inhibits setting the *y*-axis minimum value to zero, when the data are all positive and nonzero. The keyword *YNozero* sets this bit temporarily. |

**NOTE** The *ZStyle* keyword has no effect in Date/Time plots.

## XTickformat, YTickformat, ZTickformat

**Used With:** JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

Lets you use FORTRAN-style format specifiers to change the format of tick labels for the *x*-, *y*-, and *z*-axes.

This keyword works basically the same way as the *Tickformat* keyword.

See also *Tickformat*.

## XTicklen, YTicklen, ZTicklen

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

Functions the same as the keyword *Ticklen*. *[XYZ]Ticklen*, however, can be applied to the *x*-, *y*-, and *z*-axes. *[XYZ]Ticklen* supersedes the value of the *Ticklen* setting.

## XTickname, YTickname, ZTickname

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

A string array, of up to 30 elements, containing the annotation of each major tick mark.

If omitted, or if a given string element that contains the null string, PV-WAVE labels the tick mark with its value. To suppress the tick label, supply a string array of one-character-long blank strings. You can do this with the command:

```
REPLICATE(' ', N)
```

(Null strings cause PV-WAVE to number the tick mark with its value.) Note that if there are *n* tick mark intervals, there are *n* + 1 tick marks and labels.

## XTicks, YTicks, ZTicks

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

The number of major tick intervals to draw for the *x*-, *y*-, and *z*-axes. If omitted PV-WAVE will select from three to six tick intervals. Setting this field to *n*, where *n* > 0, produces exactly *n* tick intervals, and *n* + 1 tick marks.

## XTickv, YTickv, ZTickv

**Used With:** JWAVE_CONTOUR, JWAVE_PLOT, JWAVE_SURFACE

The data values for each tick mark, an array of up to 30 elements.

This keyword allows you to directly specify tick data values, producing graphs with non-linear tick marks. PV-WAVE scales the axis from the first tick value to the last, unless you directly specify a range. If you specify *n* tick intervals, you must specify *n* + 1 tick values.

## XTitle, YTitle, ZTitle

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT, JWAVE_SURFACE

Specifies a string to be used as a title below the *x*-, *y*-, and *z*-axes.

See also *Title*

## XType, YType, ZType

**Used With:** JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PLOT

Specifies a linear axis if zero; specifies a logarithmic axis if one; and if set to 2, enables compressed Julian numbers to be used directly with the graphics procedures.

**NOTE** *YType* has no effect in Date/Time plots.

## YNozero

**Used With:** JWAVE_PLOT

Inhibits setting the minimum *y*-axis value to zero when the *y* data are all positive and nonzero, and no explicit minimum *y* value is specified (using *Yrange*).

By default, the *y*-axis spans the range of 0 to the maximum value of *y*, in the case of positive *y* data.

## ZAxis

**Used With:** JWAVE_BAR3D , JWAVE_SURFACE

Specifies the placement of the *z*-axis.

By default, the *z*-axis is drawn at the upper-left corner of the axis box. To suppress the *z*-axis, use `ZAxis = -1` in the call. The position of the *z*-axis is determined from *ZAxis* as follows:

1 = lower-right, 2 = lower-left, 3 = upper-left, and 4 = upper-right.

## *Named Color Parameters*

**NOTE** *Keyword Parameters* are listed on page C-2. *Named ColorSet Parameters* are listed on page C-20.

### Axis

**Used With:** JWAVE_PLOT

Specifies an AWT color object used to draw the axis. (Default: `Color.white`)

### Background

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PIE, JWAVE_PLOT, JWAVE_SURFACE

Specifies an AWT color object to which the screen is set when the ERASE procedure is called. (Default: Color.black)

**NOTE** Not all devices support erasing the background to a color index.

### Bottom

**Used With:** JWAVE_SURFACE

Specifies an AWT color object used to draw the lower part of the surface. If not specified, the bottom is drawn with the same color as the top (the top is specified with the *Color* keyword).

**NOTE** If the *x*-axis rotation is between 90 and 270 degrees, the top of the surface will be colored with the color specified by the *Bottom* keyword.

### Color

**Used With:** JWAVE_BAR3D, JWAVE_CONTOUR, JWAVE_HISTOGRAM, JWAVE_PIE, JWAVE_PLOT, JWAVE_SURFACE

Specifies an AWT color object to set the color of text, lines, solid polygon fill, data, axes, and annotation. (Default: `Color.white`)

## *Named ColorSet Parameters*

**NOTE** *Keyword Parameters* are listed on page C-2. *Named Color Parameters* are listed on page C-19.

### C_Colors

**Used With:** JWAVE_CONTOUR

An array of AWT color objects used to set the color used to draw each contour.

If there are more contour levels than elements in *C_Colors*, the elements of the color array are cyclically repeated.

### ColumnColors

**Used With:** JWAVE_BAR3D

An array of AWT color objects specifying the color for each column of bars. Must have the same number of colors as the second dimension of *Z*. Only one of *Row-Colors* or *ColumnColors* may be used.

### Fill_Colors

**Used With:** JWAVE_CONTOUR

***Color_Index*** — If present, specifies an array of AWT Color objects to be used in the contour plot. Element *i* of this array contains the color of contour level number *i* – 1. Element 0 contains the background color. There must be one more color than there are number of contour levels. This keyword is only valid of the *Filled* key-word is set.

### RowColors

**Used With:** JWAVE_BAR3D

An array of AWT color objects specifying the color for each row of bars. Must have the same number of colors as the first dimension of *Z*. Only one of *RowColors* or *ColumnColors* may be used.

D

APPENDIX

# HTTP Configuration File

This appendix describes options that you can set in a file used to configure the JWAVE HTTP Web server. The HTTP Web server is active if the property `MANAGER_START_HTTP` is set to `TRUE` in the JWAVE Configuration Tool, as described in *Using the JWAVE Configuration Tool* on page 118.

By default, the location of this configuration file is:

```
VNI_DIR/jwave-3_5/bin/jwave_http.cfg
```

where `VNI_DIR` is the main Visual Numerics installation directory.

To change the default location of the `jwave_http.cfg` file, edit the property `HTTP_CONFIG` in the JWAVE Configuration Tool.

**NOTE** On Windows platforms, you must use two back slashes (\\) as a directory separator, rather than the normal single back slash (\). For example:

```
C:\\Program Files\\MyDocs
```

You must stop and restart the JWAVE Manager to make any changes you make to this file take effect. For example:

```
manager shutdown
manager start
```

Or, if you are using the JWAVE Windows NT Service:

```
net stop jwaveservice
net start jwaveservice
```

## General Parameters

***homeDir*** — The default directory to serve for the URL of “/”.

Default: `VNI_DIR/classes`

Example: `homeDir = c:\\program files\\myWebHome`

***indexFile*** — The default file to serve for a URL that maps to a directory.

Default: `index.html`

Example: `indexFile = index.html`

***JWaveURL*** — The URL that serves a JWAVE connection. Set the HTTP connection method to use this URL (for example: `http://myhost:8080/JWave`). Should start with a slash (/).

Default: `/JWave`

Example: `JWaveURL = /JWave`

***MimeDefault*** — The default mime type to use for unknown file types.

Default: `application/octet-stream`

Example: `MimeDefault = application/octet-stream`

## Directory Mapping

To map a URL to a directory (so that when you point your browser to a URL, the server knows the directory in which to find files).

Syntax:

```
dir.<url> = <full-path-to-directory>
```

Replace `<url>` with the top-level URL you wish to map (no slashes are allowed)

Replace `<full-path-to-directory>` with the full path to the directory that should map to that URL

For example, if you want the URL `http://myhost/someplace/file.html` to map to the server’s file `C:\Program Files\MyDocs\file.html`, then use:

```
dir.someplace = C:\\Program Files\\MyDocs
```

Default:

```
dir.classes = VNI_DIR/classes
```

where `VNI_DIR` is the JWAVE installation directory.

## *Mime Types Mapping*

To map a file extension (`file.ext`) to a Mime type.

Syntax:

```
mime.<ext> = <mime-type>
```

- Replace `<ext>` with the file extension (whatever follows the last period in the file).
- Replace `<mime-type>` with the Mime type to use when serving files with the given extension.

For example, to serve a file `foo.html` as Mime type `text/html`, use:

```
mime.html = text/html
```

The defaults are:

```
mime.html = text/html
mime.htm = text/html
mime.jpeg = image/jpeg
mime.jpg = image/jpeg
mime.gif = image/gif
mime.png = image/png
mime.txt = text/plain
mime.class = application/octet-stream
mime.jar = application/octet-stream
mime.properties = text/plain
```

# JWAVE Bean Tools Reference

This appendix describes a set of JWAVE Beans that are provided by Visual Numerics. The data input and customizable parameters for each tool are described.

See *Using JWAVE Beans with the BeanBox* on page 83 for more information on using these Bean Tools.

## List of JWAVE Bean Tools

## JWAVE Bar3d Tool

The JWAVE Bar3D Tool is a Bean that displays a 3D bar plot.

See Chapter 7, *Using JWAVE Beans* for information on using this Bean in the BeanBox.

### Input

The JWAVE Bar3d Tool uses a `DoubleTable` object as data input to the PV-WAVE BAR3D procedure. (Default: 30 columns of data)

### Customizer Reference

The JWAVE Bean Tools provided by Visual Numerics each have a Customizer for modifying the appearance of the plot produced by the Bean. This section describes the features of the Bar3D Tool Customizer.

#### *Titles Tab*

**Title** — Defines the text for the main title that appears centered above the plot. The text size of this main title is larger than the other text by a factor of 1.25.

**Sub Title** — Defines the text for the subtitle that appears underneath the *x*-axis.

**X Axis Title** — Defines the text for the title that appears below the *x*-axis.

**Y Axis Title** — Defines the text for the title that appears below the *y*-axis.

**Z Axis Title** — Defines the text for the title that appears below the *z*-axis.

**Character Size** — Sets the overall character size for the annotation. A value of 1.0 is normal. The main title is written with a character size of 1.25 times this parameter.

#### *Rotations Tab*

**X Rotation** — Specifies the clockwise rotation about the *x*-axis. (Default: 30˚)

**Z Rotation** — Specifies the clockwise rotation about the *z*-axis. (Default: 30˚)

#### *Colors Tab*

**Background Color** — Brings up a color tool for selecting the background color of the plot.

**Foreground Color** — Brings up a color tool for selecting the foreground color of the plot.

#### *Columns Tab*

**Column Names** — Sets the column you want to work with while selecting the Column Color parameter.

**Column Color** — Brings up a color tool for selecting the color of the selected column.

## *JWAVE Contour Tool*

The JWAVE Contour Tool is a Bean that displays a contour plot.

See Chapter 7, *Using JWAVE Beans* for information on using this Bean in the BeanBox.

### Input

This Tool uses up to three data arrays (*z*, *x*, and *y*) as input to the PV-WAVE CONTOUR procedure.

*z* — A 2D array containing the values that make up the contour surface.

*x* — (optional) A 1D or 2D array specifying the *x*-coordinates for the contour surface.

*y*— (optional) A 1D or 2D array specifying the *y*-coordinates for the contour surface.

### Customizer Reference

The JWAVE Beans provided by Visual Numerics each have a Customizer for modifying the appearance of the plot produced by the Bean. This section describes the features of the Contour Tool Customizer.

#### *Contour Parameters Tab*

**Grid On** — When selected, a grid is displayed behind the contour.

**Fill Contours** — When selected, a color other than the background color is used to fill the areas between contour lines. When Fill Contours is selected, the intervals between the contour lines are filled with the contour line colors.

**Number of Levels** — Sets the number of equally-spaced contour levels that are produced. The maximum is 150. (Default: 6)

**X Range** — Sets the data range of the *x*-axis, a two-element vector. The first element is the axis minimum and the second is the maximum.

**Y Range** — Sets the data range of the *y*-axis, a two-element vector. The first element is the axis minimum and the second is the maximum.

**Cell** — The cell-drawing method, the default, examines each array cell and draws all contours emanating from that cell before proceeding to the next cell. This method is efficient in terms of computer resources but does not allow contour labeling.

**Follow Method** — The line-following method searches for each contour line and then follows the line until it reaches a boundary or closes. This method gives better looking results with dashed line styles and allows contour labeling, but requires more computer time.

**Spline** — Allows you to spline the contour lines. This function can be used when either the Follow method of contour drawing or Fill Contours is selected.

Splines are especially useful with small data sets (less than 15 array dimensions). With larger data sets the smoothing is not as noticeable and the expense of splines increases rapidly with the number of data points.

**Spline Size** — Specifies the length of each interpolated line segment in normalized coordinates. Smaller values for this parameter yield smoother lines, up to the resolution of the output device, at the expense of more computations. (Default: 0.005)

**Interpolation (CONGRID)** — When selected, applies interpolation, which shrinks or expands the number of elements in the contour by interpolating values at intervals where there might not have been values before.

**Number of X Points** — Sets the number of columns desired in the output image.

**Number of Y Points** — Sets the number of rows desired in the output image.

### *Plot Basics Tab*

**Title** — Defines the text for the main title that appears centered above the plot. The text size of this main title is larger than the other text by a factor of 1.25.

**Sub Title** — Defines the text for the subtitle that appears underneath the *x*-axis.

**X Axis Title** — Defines the text for the title that appears below the *x*-axis.

**Y Axis Title** — Defines the text for the title that appears below the *y*-axis.

**Character Size** — Sets the overall character size for the annotation. A value of 1.0 is normal. The main title is written with a character size of 1.25 times this parameter.

**Background Color** — Brings up a color tool for selecting the background color of the plot.

**Foreground Color** — Brings up a color tool for selecting the foreground color of the plot.

### *Levels Tab*

**Default Annotation** — Labels the selected contour level with its elevation.

**No Annotation** — Turns off annotation of the selected contour level.

**Special Annotation** — Applies the specified annotation for the selected contour level. When the Special Annotation button is selected, you can enter annotation text in the text field located below the button.

**Line Color** — Brings up a color tool for selecting the line color for the selected level.

**Fill Color** — Brings up a color tool for selecting the fill color for the selected level.

**Line Thickness** — Sets the value for the thickness of the lines in the selected level. A value of 1.0 is normal thickness, 2.0 is double-wide, and so on.

**Line Style** — Sets the line style of the lines in the selected level. Choices are: Solid, Dotted, Dashed, Dash dot, Dash-dot-dot-dot, and Long dashes.

## *JWAVE Generic Tool*

The JWAVE Generic Tool is a Bean that contains a property that determines which PV-WAVE procedure to invoke to generate and display a plot. This Bean can take up to ten data `Proxy`'s (all 2D arrays) as input to the PV-WAVE procedure.

See Chapter 7, *Using JWAVE Beans* for information on using this Bean in the BeanBox.

### Customizer Reference

The JWAVE Beans provided by Visual Numerics each have a Customizer for modifying the appearance of the plot produced by the Bean. This section describes the features of the Generic Tool Customizer.

#### *Plot Properties Tab*

**Plot Type** —Specifies the name of a JWAVE procedure file stored in:

**(UNIX)** `VNI_DIR/jwave-3_5/lib`

**(Windows)** `VNI_DIR\jwave-3_5\lib`

where `VNI_DIR` is the main Visual Numerics installation directory.

**Title** — Defines the text for the main title that appears centered above the plot.

#### *Colors Tab*

**Background Color** — Brings up a color tool for selecting the background color of the plot.

**Foreground Color** — Brings up a color tool for selecting the foreground color of the plot.

## JWAVE Histogram Tool

The JWAVE Histogram Tool is a Bean that displays a histogram plot.

### Input

This Tool uses a 1D array containing histogram data as input to the PV-WAVE HISTOGRAM procedure.

See Chapter 7, *Using JWAVE Beans* for information on using this Bean in the BeanBox.

### Customizer Reference

The JWAVE Beans provided by Visual Numerics each have a Customizer for modifying the appearance of the plot produced by the Bean. This section describes the features of the Histogram Tool Customizer.

#### *Titles Tab*

**Title** — Defines the text for the main title that appears centered above the plot.

**X Axis Title** — Defines the text for the title that appears below the *x*-axis.

**Y Axis Title** — Defines the text for the title that appears below the *y*-axis.

#### *Histogram Properties Tab*

**Bin Size** — The range of values to consider as having a single value. If no value is specified, the Bin Size range defaults to a value of 1.

**Stepped** — When selected, the histogram is displayed without the vertical lines between the bars, which produces a histogram plot that resembles ascending and descending stair steps.

#### *Colors Tab*

**Background Color** — Brings up a color tool for selecting the background color of the histogram plot.

**Foreground Color** — Brings up a color tool for selecting the foreground color of the histogram plot.

**Fill Histogram** — When selected, the color set with the Fill Color parameter is used to fill the bins of the histogram.

**Fill Color** — Brings up a color tool used to select a color for filling the bins in the histogram.

#### *Axis Tab*

**Axis Color** — Brings up a color tool for selecting a color for the axes.

**X Max** — Defines the maximum for the data range of the *x*-axis.

**X Min** — Defines the minimum for the data range of the *x*-axis

## *JWAVE Pie Tool*

The JWAVE Pie Tool is a Bean that displays a pie chart.

### Input

This Tool uses a `DoubleTable` object as input to the PV-WAVE PIE_CHART procedure.

See Chapter 7, *Using JWAVE Beans* for information on using this Bean in the BeanBox.

### Customizer Reference

The JWAVE Beans provided by Visual Numerics each have a Customizer for modifying the appearance of the plot produced by the Bean. This section describes the features of the Pie Tool Customizer.

#### *Labels Tab*

**Title** — Defines the text for the main title that appears centered above the pie chart.

**Show Slice Percentages** — When selected, each slice is labeled with the percentage it represents out of the total pie.

**Show Slice Values** — When selected, each slice is labeled with the value it represents.

**Show Label Border** — When selected, a filled rectangle displays behind each label.

**Slice Label Position** — Defines the position of slice labels.

Internal — Places the label in the slice.

External — Places the label outside the slice.

External Aligned — Places the label outside the slice and lined-up on each side of the pie.

### Dimensions Tab

**X Center** — The *x*-axis coordinate for the center of the pie.

**Y Center** — The *y*-axis coordinate for the center of the pie.

**Shade Displacement** — Specifies the percentage of displacement for the center of the drop shadow displayed behind the pie chart. The direction of the displacement is 315˚.

### Colors Tab

**Background Color** — Brings up a color tool for selecting the background color of the pie chart.

**Foreground Color** — Brings up a color tool for selecting the foreground color of the pie chart.

**Label Border Color** — If Show Label Border (in the Labels tab) is selected, brings up a color tool for selecting the color of the filled rectangle that displays behind each label.

### Slices Tab

**Slices** — Sets the slice you want to work with while using the parameters in the Slices tab.

**Slice Color** — Brings up a color tool for selecting the color for the selected slice.

**Slice Label Color** — Brings up a color tool for selecting the color of the text in the label for the selected slice.

**Percent Explode for Slice** — Defines the percentage to explode the selected slice from the center of the pie.

## JWAVE Plot Tool

The JWAVE Plot Tool is a Bean that displays an XY (2D) graph.

See Chapter 7, *Using JWAVE Beans* for information on using this Bean in the BeanBox.

### Input

This Tool uses one or two data arrays (X and Y) as input to the PV‑WAVE PLOT procedure.

### Customizer Reference

The JWAVE Beans provided by Visual Numerics each have a Customizer for modifying the appearance of the plot produced by the Bean. This section describes the features of the Plot Tool Customizer.

#### *Titles Tab*

**Title** — Defines the text for the main title that appears centered above the plot. The text size of this main title is larger than the other text by a factor of 1.25.

**Sub Title** — Defines the text for the subtitle that appears underneath the *x*-axis.

**X Axis Title** — Defines the text for the title that appears below the *x*-axis.

**Y Axis Title** — Defines the text for the title that appears below the *y*-axis.

**Character Size** — Sets the overall character size for the annotation. A value of 1.0 is normal. The main title is written with a character size of 1.25 times this parameter.

#### *Colors Tab*

**Background Color** — Brings up a color tool for selecting the background color of the plot.

**Axis Color** — Brings up a color tool for selecting the axis color of the plot.

**Y Data Color** — Brings up a color tool for selecting the color of the *y* data.

**X Data Color** — Brings up a color tool for selecting the color of the *x* data.

### *Plot Tab*

**Grid On** — When selected, a grid is displayed behind the plot.

**Polar Plot** — When selected, a polar plot is produced.

**Axis Scaling** — Sets the type of scaling to be used for the X and Y axes.

Linear – Linear — Produces a plot with linear scaling on both axes.

Log – Linear — Produces a plot with logarithmic scaling on the *x*-axis and linear scaling on the *y*-axis.

Linear – Log — Produces a plot with linear scaling on the *x*-axis and logarithmic scaling on the *y*-axis.

Log – Log — Produces a plot with logarithmic scaling on both axes.

**X Range** — Defines the data range of the *x*-axis, a two-element vector. The first element is the axis minimum and the second is the maximum.

**Y Range** — Defines the data range of the *y*-axis, a 2-element vector. The first element is the axis minimum and the second is the maximum.

### *Data Tab*

**Plot X vs. Y** — If one data array is present, graphs the array vs. the index. If two data arrays are present, graphs one array vs. the other.

**Plot X and Y data arrays** — Only available if two data arrays are present. Graphs both arrays vs. the index.

### *Y Data*

**Line Style** — Sets the line style of the *y* data line. Choices are: Solid, Dotted, Dashed, Dash dot, Dash-dot-dot-dot, and Long dashes.

**Line Thickness** — Sets the value for the thickness of the *y* data line. A value of 1.0 is normal thickness, 2.0 is double-wide, and so on.

**Symbol** — Sets the type of symbol to used to plot each *y* data point.

None — Specifies that no symbols display at each *y* data point. The None setting results in a line drawn between data points.

Plus Sign — Specifies a **+** at each *y* data point.

Asterisk — Specifies an ***** at each *y* data point.

Period — Specifies a **.** at each *y* data point.

Diamond — Specifies a diamond shape at each *y* data point.

Triangle — Specifies a triangle shape at each *y* data point.

Square — Specifies a square shape at each *y* data point.

X — Specifies an **X** at each *y* data point.

**Symbol Size** — Sets the value for the size of *y* data plot symbols. A value of 1.0 is normal size, 2.0 is double-size, and so on.

**Connect Symbols with Lines** — When selected, symbols are connected with lines. This parameter can be used when the value of Symbol is other than None.

### *X Data*

**Line Style** — Sets the line style of the *x* data line. Choices are: Solid, Dotted, Dashed, Dash dot, Dash-dot-dot-dot, and Long dashes.

**Line Thickness** — Sets the value for the thickness of the *x* data line. A value of 1.0 is normal thickness, 2.0 is double-wide, and so on.

**Symbol** — Sets the type of symbol to used to plot each *x* data point.

None — Specifies that no symbols display at each *x* data point. The None setting results in a line drawn between data points.

Plus Sign — Specifies a **+** at each *x* data point.

Asterisk — Specifies an ***** at each *x* data point.

Period — Specifies a **.** at each *x* data point.

Diamond — Specifies a diamond shape at each *x* data point.

Triangle — Specifies a triangle shape at each *x* data point.

Square — Specifies a square shape at each *x* data point.

X — Specifies an **X** at each *x* data point.

**Symbol Size** — Sets the value for the size of *x* data plot symbols. A value of 1.0 is normal size, 2.0 is double-size, and so on.

**Connect Symbols with Lines** —When selected, symbols are connected with lines. This parameter can be used when the value of Symbol is other than None.

## *JWAVE Surface Tool*

The JWAVE Surface Tool is a Bean that displays the surface of a 2D array. The surface can be displayed with hidden lines removed or shaded. This Tool uses the PV-WAVE SURFACE procedure to render the plot.

See Chapter 7, *Using JWAVE Beans* for information on using this Bean in the BeanBox.

### Input

This Tool takes up to four data arrays (*z*, *x*, *y*, and shade values).

*z* — A 2D array containing the values that make up the surface. If *x* and *y* are supplied, the surface is plotted as a function of the X,Y locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of *z*.

*x* — (optional) A 1D or 2D array specifying the *x*-coordinates for the surface.

- If *x* is a 1D array, each element of *x* specifies the *x*-coordinate for a column of *z*. For example, *x*(0) specifies the *x*-coordinate for *z*(0, *).
- If *x* is a 2D array, each element of *x* specifies the *x*-coordinate of the corresponding point in *z*.

*y* — (optional) A 1D or 2D array specifying the *y*-coordinates for the surface.

- If *y* is a 1D array, each element of *y* specifies the *y*-coordinate for a row of *z*. For example, *y*(0) specifies the *y*-coordinate for *z*(*, 0).
- If *y* is a 2D array, each element of *y* specifies the *y*-coordinate of the corresponding point in *z*.

### Customizer Reference

The JWAVE Beans provided by Visual Numerics each have a Customizer for modifying the appearance of the plot produced by the Bean. This section describes the features of the Surface Tool Customizer.

#### *Titles Tab*

**Title** — Defines the text for the main title that appears centered above the plot. The text size of this main title is larger than the other text by a factor of 1.25.

**Sub Title** — Defines the text for the subtitle that appears underneath the *x*-axis.

**X Title** — Defines the text for the title that appears below the *x*-axis.

**Y Title** — Defines the text for the title that appears below the *y*-axis.

**Z Title** — Defines the text for the title that appears below the *z*-axis.

**Character Size** — Sets the overall character size for the annotation. A value of 1.0 is normal. The main title is written with a character size of 1.25 times this parameter.

### *Colors Tab*

**Background Color** — Brings up a color tool for selecting the background color of the plot.

**Foreground Color** — Brings up a color tool for selecting the foreground color of the plot.

**Bottom Color** — Brings up a color tool for selecting the color of the bottom of the plot.

**Color Table** — Sets the color table used for the plot.

### *Plot Tab*

**Grid** — When selected, a grid is displayed behind the plot.

**Shade** — When selected, the plot will be shaded.

**Overplot with Surface** — When selected, plot lines appear on top of the surface if Shade is also selected.

**Z Axis Location** — Sets the position of the *z*-axis origin to lower-right, lower-left, upper-right, or upper-left.

**Z Rotation** — Specifies the rotation about the *z*-axis. (Default: 30˚)

**X Rotation** — Specifies the rotation about the *x*-axis. (Default: 30˚)

**Skirt** — When selected, a skirt is added to the surface. A skirt helps establish a frame of reference between the surface and the *x*-, *y*-, and *z*-axes. Skirt can be used when Shade is not set.

**Skirt Value** — Defines the value along the *z*-axis at which the bottom of the skirt begins. (Default = minimum *z* value for the variable)

**X Range** — Sets the data range of the *x*-axis, a two-element vector. The first element is the axis minimum and the second is the maximum.

**Y Range** — Sets the data range of the *y*-axis, a two-element vector. The first element is the axis minimum and the second is the maximum.

**Z Range** — Sets the data range of the *z*-axis, a two-element vector. The first element is the axis minimum and the second is the maximum.

### *Data Tab*

**Interpolation** — When selected, applies interpolation, which shrinks or expands the number of elements in the surface plot by interpolating values at intervals where there might not have been values before.

**Number of X Points** — Sets the number of columns desired in the output image.

**Number of Y Points** — Sets the number of rows desired in the output image.

# Glossary

### *API*

An acronym for Application Programming Interface. The JWAVE wrapper API is a set of PV-WAVE functions used specifically for creating JWAVE applications.

### *applet (Java)*

A Java application that runs inside a Web browser or an application such as an applet viewer. Unlike an application, an applet has no `main` method and must be referenced in an HTML.

### *application (Java)*

A command-line executable written in Java. Unlike an applet, a Java application does not have to run inside a Web browser or applet viewer, and an application has a `main` method.

### *BDK*

The JavaBeans$^{TM}$ Development Kit from Sun Microsystems, Inc.

### *BeanBox*

A tool for testing the functionality of a JavaBean. Available in the Beans Development Kit (BDK version 1.0 March 98, or later) from Sun Microsystems.

### *Beans*

See JavaBeans.

### *CGI*

An acronym Common Gateway Interface. Used to run programs through a Web server.

### *class*

Basic unit of compilation and execution in Java. All Java programs are classes.

### *client (JWAVE)*

In a JWAVE system, a local processor connected to the JWAVE server. The JWAVE client is used to develop Java applications that communicate directly with PV‑WAVE running on the JWAVE server. JWAVE classes and JAR files, a Java compiler, and (optional) a BDK reside on the JWAVE Development client.

### *configuration tool*

A graphical user interface used to configure the JWAVE server.

### *customizer*

A graphical user interface used to configure a JWAVE Beans Tool.

### *Data Manager*

JWAVE server software that keeps track of data in a PV‑WAVE session. Although usually accessed indirectly by client applications using data proxies, JWAVE wrapper developers can use Data Manager (DM) functions directly in JWAVE wrappers to store and access data.

### *domain*

A named dataspace in which JWAVE client applications can store data on the server.

### *event*

Something important that happens at a specific point in time during runtime of a Java application (such as a mouse click, a condition being met, or new data arriving).

### *HTML*

An acronym for HyperText Markup Language. HTML is the source text for most Web pages.

### *JAR files*

A Java Archive file used for packaging related class files, serialized JavaBeans, and other resources.

### *Java*

An object-oriented programming language developed by Sun Microsystems. Java programs are architecture neutral, which means that they can run on any machine that implements the Java Virtual Machine. The JWAVE client is certified as 100% Pure Java$^{TM}$.

### *JavaBeans*

JavaBeans allow you to write self-contained, reusable software units that can be visually assembled in a visual application builder tool.

The component architecture for Java. components in graphical user environments. JavaBeans are a core capability of the JDK 1.1 from Sun Microsystems.

### *Javadoc*

A tool used to produce reference documentation for Java class files. To use JWAVE "Javadoc", open the following file in a Web browser:

| | |
|---|---|
| **(UNIX)** | `VNI_DIR/classes/docs/api/packages.html` |
| **(Windows)** | `VNI_DIR\classes\docs\api\packages.html` |

where VNI_DIR is the main Visual Numerics installation directory.

### *JavaScript*

An object-based scripting language for embedding programming scripts in HTML files. JavaScript can be used with the generic JWAVE applet to create a graphical user interface for a Web page.

### *JDK*

An acronym for Java Development Kit. This is the core software development package for Java from Sun Microsystems.

### *JWAVE*

JWAVE is a visualization and computational environment that allows you to quickly and easily develop cross-platform applications to analyze and visually interpret data. JWAVE is a 100% Pure Java client that lets you deliver applications and solutions across the Internet or your intranet.

### *JWAVE Beans*

JavaBeans that are written specifically to use JWAVE components. Visual Numerics provides a set of JWAVE Beans for JWAVE client developers. Note that JWAVE Beans are supported with BDK version 1.0 (March 98), or a later version.

### *JWAVE Manager*

A process on the JWAVE server that "listens" for client connections. When a connection is made, the JWAVE Manager starts a PV-WAVE session and executes a "JWAVE wrapper function."

### *JWAVE wrapper*

A PV-WAVE function that contains JWAVE-specific functions for passing parameters between JWAVE client and server applications.

### *keyword*

An optional PV-WAVE function or procedure parameter that is of the form: *keyword*=*value*. For instance, the PV-WAVE PLOT command has numerous keywords, such as *Title*, *Color*, and *XRange*.

### *log*

Information that is output from a JWAVE program. You can configure the JWAVE Manager to output log information to the terminal or to a file.

### *Manager*

See JWAVE Manager and Data Manager.

### *method*

The term used for a procedure or function that is part of a class in an object-oriented programming language, such as Java.

### *pack*

Parameters and data must be "packed" before being sent between JWAVE client and server programs. Packing simply refers to the formatting of data in a consistent manner so that it can be interpreted (or "unpacked") correctly after being received.

### *persistence (of data)*

This term simply refers to the time during which data is available to a PV-WAVE session. In general, data stored in the JWAVE Data Manager is persistent as long as the PV-WAVE session is active. When the session is closed, the data (which was in memory) is lost, unless it was explicitly saved.

### *ping*

A ping refers to a program that determines whether or not a specified process is running. The JWAVE Manager has several ping options that let you test if the JWAVE Manager is running.

### *proxy*

Generally speaking, a proxy is an object-oriented programming term that refers to an object that represents or refers to another object, such as data. In JWAVE, proxies are used by client applications to refer to data that is stored on the server.

### *PV-WAVE*

PV-WAVE, from Visual Numerics, Inc., provides the fundamental components for developing visual data analysis applications. These components include a highly developed, array oriented 4GL language as well as robust graphical and numerics routines. PV-WAVE is the graphical and numerical "engine" for JWAVE.

### *PV-WAVE application*

A program written in the PV-WAVE language and using PV-WAVE graphics and numerical routines. JWAVE allows client applications, written in Java, to communicate with PV-WAVE applications running on a server machine.

### *PV-WAVE session*

A single PV-WAVE process running on a server. JWAVE client applications can contact a single PV-WAVE session multiple times, or multiple PV-WAVE sessions at once. Any data that is stored by the JWAVE Data Manager is persistent for the life of the PV-WAVE session with which it is associated.

### *serialize*

When “serializable” JavaBeans are linked together to form an application in an visual development environment such as the BeanBox, that application can be saved as a stand-alone Java application.

### *server (JWAVE)*

In a JWAVE system, the remote processor where PV‑WAVE, the JWAVE Manager, the configuration files, and class files, JWAVE wrappers, and the JRE (Java Runtime Environment) reside.

### *Service, Windows NT*

The JWAVE Manager can be installed and run as a Service on a Windows NT server. Installing JWAVE Manager as a Windows NT Service allows you to:

- run the JWAVE Manager as a background process
- keep the JWAVE Manager running when there are no interactively logged-in users
- shut down the JWAVE Manager by stopping the JWAVE Service

### *socket*

A specific port, identifiable by a number, for connecting clients and servers on a network. JWAVE can be configured to accept client communications through a socket. See also Web server.

### *Swing components*

The Swing component set is a graphical user interface (GUI) toolkit used for creating menus, text fields, dialog boxes, and so on. Swing consists of 100% Java components, and are completely platform independent. You can download the Swing component toolkit from the Sun Microsystems Web site.

### *unpack*

Parameters and data that are sent between client applications and JWAVE wrappers are sent in as a stream of data objects that must be interpreted, or “unpacked” after they are received.

### *Web server*

Software that listens for and handles requests transmitted from client programs across the Internet or an intranet. JWAVE can be configured to accept client communications through a Web server. See also socket.

### *wrapper*

See JWAVE wrapper.

# JWAVE Index