Oracle Video Client

Developer's Guide

Release 3.0

February 3, 1998

Part No: A53949-02

Developer's Guide

Part No: 3.0 A53949-02

Primary Author: Rick Herrick

Contributors: Gordon Furbush, Leslie Kew, Nancy Baltz, Richard Bettelheim, John Dowden, Brice Dunwoodie, Dana Izenson, Martin McKendrick, Mason Ng, Matt Prather, Bettina Rafailovic, James Steel, Sean Trabosh, Shiu Wong, John Zussman, Bernie Cohen.

Contents

3 Oracle Video Java Library

4 Oracle Video ActiveX Control

5 Working with Oracle Forms

A Oracle Video Web Plug-in Reference

B Oracle Video Java Library Reference

C Oracle Video ActiveX Control Reference

Preface

The Oracle Video Client software enables you to develop interactive, video-based multimedia applications, such as computer-based training (CBT), interactive kiosks, and movies-on-demand. Oracle Video Client applications can receive and display streamed MPEG-1 (Motion Picture Experts Group) video, MPEG audio, and OSF (Oracle Streaming Format) files from the Oracle Video Server (OVS) system. Other formats can be converted to OSF and streamed also, including WAV and AVI files.

> **Note:** The Oracle Video Client can play both video and audio. For simplicity, this manual usually refers only to video, since this includes both visual and auditory elements, but all information applies equally to audio, unless otherwise noted.

The Oracle Video Server allows many different clients to access media files concurrently. Multiple users can even receive streaming video from different segments of the same media file at the same time. The Oracle Video Client software provides the following tools to help you build media applications:

- The Oracle Video ActiveX Control enables Windows 95, Windows NT 4.0, and Internet multimedia applications to handle streams from the Oracle Video Server. This control is designed to work with any application that hosts ActiveX controls; it has been tested with:
  - Microsoft Visual Basic 4.0 and 5.0
  - Internet Explorer 3.0 and 4.0
  - Oracle Forms 4.5 and 5.0.5
  - Oracle Power Objects 2.1

- The Oracle Video Web Plug-in is a Netscape-compatible plug-in that enables Web-based applications to handle streams from the Oracle Video Server. Using Netscape's LiveConnect interface, you can use JavaScript or Java to dynamically control the Oracle Video Web Plug-in.
- The Oracle Video Java Library is a set of Java 1.1 classes and interfaces that enable Java applications to handle streams from the Oracle Video Server. The library also includes support classes for things like handling player events and querying a video server for a list of available content.

The Oracle Video Client package also includes the Iterated Systems ClearVideo (fractal) video and Voxware MetaSound audio encoder/decoders (also known as *codecs*). These codecs are suitable for low-bit-rate streaming, enabling the Oracle Video Client to receive and display video and audio at low bit rates.

The *Oracle Video Client Developer's Guide* provides the information necessary to develop applications with the Oracle Video Client software, specifically:

- **Conceptual information:** Descriptions of the available client interfaces that you can use to create your own Oracle Video Client applications, as well as the infrastructure that enables the client interfaces to access the Oracle Video Server
- **Step-by-step tutorials**: Instructions for developing applications with Microsoft Visual Basic 4.0 and 5.0, Oracle Power Objects 2.1, Oracle Forms 4.5 and 5.0, Internet applications, and Java applications that stream video locally and from the Oracle Video Server
- **Developer reference:** Descriptions and code examples for the methods and properties associated with the Oracle Video ActiveX Control, the Oracle Video Web Plug-in and the Oracle Video Java Library

You can find installation and configuration information for the Oracle Video Client, including system requirements and software compatibility requirements, in the booklet that came inserted with your Oracle Video Client installation CD-ROM. This information is also available in electronic form in the file **ovcinstl.pdf** in the **\Docs** directory of your installation CD-ROM.

Intended Audience

This manual is for multimedia developers or anyone with the appropriate experience who wants to create stand-alone or Web-based applications that use streaming audio or video from the Oracle Video Server.

How This Manual Is Organized

This manual is divided into the following chapters:

Chapter 1, "Introducing the Oracle Video Client" contains:

- Descriptions of the Oracle Video Client system and its components
- High-level information on the various client interfaces available
- Information on the development and deployment process for creating your own custom video clients

Chapter 2, "Oracle Video Web Plug-in" contains:

- Step-by-step procedures for displaying and controlling media files from the Oracle Video Server in HTML pages, using the Netscape-compatible plug-in client interface
- Sample applications using Javascript and Java to control the plug-in

Chapter 3, "Oracle Video Java Library" contains:

- Information on creating Java applications that can display and control streaming video and audio from the Oracle Video Server
- A sample Java application that loads a media file and controls playback

Chapter 4, "Oracle Video ActiveX Control" provides:

- Procedures for embedding the control in HTML pages for ActiveX-enabled browsers and scripting the control through VBScript and JScript
- Steps for building applications with the Oracle Video ActiveX Control using Oracle Power Objects 2.1 and Microsoft Visual Basic 4.0 and 5.0

Chapter 5, "Working with Oracle Forms" provides:

- Steps for building applications with the Oracle Video ActiveX Control using Oracle Forms 4.5.8

Appendix A, "Oracle Video Web Plug-in Reference" contains reference information for the Oracle Video Web Plug-in interface.

Appendix B, "Oracle Video Java Library Reference" contains reference information for the Oracle Video Java Library interface.

Appendix C, "Oracle Video ActiveX Control Reference" contains reference information for the Oracle Video ActiveX Control interface.

Appendix D, "The Media File" describes how to request a specific content file from the Oracle Video Server using either the path and file name of an MDS file or the asset cookie of the logical content title.

Related Publications

See the *Oracle Video Server Road Map* for related publications.

Conventions Used in This Manual

This guide uses particular notational conventions to clarify syntax, enhance visual access to pages, and otherwise distinguish elements, such as code examples and keystrokes the user should enter from the text provided as background for those examples. These conventions are listed here:

Table Preface–1 Typographical conventions used in this manual

Feature	Example	Explanation
boldface	**mna.h** **timeout**	Identifies file names. Identifies method arguments.
boldface, trailing parenthesis	**mzsInit()**	Identifies method names.
italics	*container1*	Identifies a place holder in command or function-call syntax; replace with a value or string of your own.
ellipses	n, ...	Indicates that the preceding item can be repeated any number of times.

Conventions for Examples

This guide shows command and code examples in a monospace Courier font:

```
file://C:\orawin\vc30\demo\www\client\test.htm
```

For examples that are longer than a single line, a backslash (\) appears at the end of a line to indicate the line continues on the next:

```
<embed name="Video1" width=400 height=400 type="application/oracle-video"\
mediafile="/mds/video_archive/oracle2.mpg">
```

Do not enter the backslash if you type out this example. It is used only as a typographic convention.

The installation application installs Oracle files in different areas on different client platforms. For this reason, file paths in this document are relative to the location indicated by the environment variable ORACLE_HOME, which represents the directory where you installed the Oracle Video Client software. You can find the default ORACLE_HOME directories for various platforms in the table below.

Table Preface–2 Default ORACLE_HOME directories

Platform	Directory
Windows 95	C:\ORAWIN95
Windows NT 4.0	C:\ORANT

Worldwide Customer Support

When you or someone in your company acquired this Oracle product, you probably also purchased some level of customer support. Oracle then sent you a package that includes telephone numbers, email addresses, and web sites you should use to contact customer support.

Oracle provides web-based support for our Oracle*MetaLink* and Oracle*Mercury* services at http://support.us.oracle.com.

If your company did not purchase customer support, you can visit http://www.oracle.com/support to find out about Oracle's Worldwide Customer Support services.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of our printed manuals is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following address or FAX number.

Oracle Video Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Mailstop 6OP5
Redwood Shores, CA 94065
U.S.A.
FAX: 650-506-7615
omsdoc@us.oracle.com

Reader's Comment Form

Name of Document: Oracle Video Client Developer's Guide Version 3.0

Part Number A53949-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Could you have better access to the information that you need? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have suggestions for improvement, please indicate the topic, chapter, and page number below:

_____

_____

_____

_____

_____

_____

_____

Please send your comments to:

Oracle Video Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Mailstop 6OP5
Redwood Shores, CA 94065

You can e-mail comments to: omsdoc@us.oracle.com

You can also fax us at (650) 506-7615.

If you would like a reply, please give your name, address, and telephone number below:

_____

_____

_____

_____

Thank you for helping us improve our documentation.

1

Introducing the Oracle Video Client

The Oracle Video Client (OVC) provides several ways to incorporate streaming video and audio from the Oracle Video Server (OVS) into your own applications. OVC provides easy-to-use client interfaces that you can embed in your applications to make video and audio available to your users. The interfaces include:

- Oracle Video Web Plug-in, a Netscape-compatible plug-in
- Oracle Video ActiveX Control, an ActiveX control
- Oracle Video Java Library, a set of Java classes

These interfaces handle selecting a file to load, video display, and user control of playback by passing client requests to the video server through a software layer. This software layer handles the technical aspects of communicating with the server, controlling the real-time stream, and audio-video playback. By handling the technical aspects of the process and providing many different client interfaces around which you can create your own client applications, the Oracle Video Client provides a flexible and portable client development tool for creating client applications that can run on many different platforms.

This introduction contains the following sections:

- Client Interfaces discusses the different client interfaces, including their individual advantages and attributes, to help you decide which interface to use for your own client applications
- Client Software Components describes the OVC software architecture
- Developing and Deploying a Client Application discusses the development and deployment process for creating your own video clients

Client Interfaces

Note: It is important to understand the multiple meanings of client as used in the OVC/OVS model:

- The first level of client is that of enterprise client: somewhere on your network, there is a video server that can be accessed by users seated at remote workstations. These workstations are clients of the OVS system and are termed “client machines”.
- The client machine contains the Oracle Video Client (OVC) software. This client consists of two parts, the OVI abstraction layer and the client interfaces, such as the Oracle Video Web Plug-in, Oracle Video Java Library, and Oracle Video ActiveX Control.
- OVI is a client of the OVS system and the client interfaces can themselves be considered clients of OVI.
- You can create your own client applications through a combination of client interfaces, scripting commands, and your own applications that use the OVC client interfaces.

The main components of the Oracle Video Client are the client interfaces. These interfaces use the Oracle Video Interface (OVI), described in “Client Software Components” on page 1-6, to access streaming video from the Oracle Video Server. The separation of the client interfaces from the basic streaming functionality supported by OVI means that you can harness the functionality of the Oracle Video Server from a variety of platforms, without worrying about the underlying mechanisms that handle the low-level tasks.

This section describes the interfaces that OVC provides, and discusses factors that can affect your decision as to which of the client interfaces you want to use to create your own custom video clients.

Interface Descriptions

OVC provides the following client interfaces:

- Oracle Video Web Plug-in
- Oracle Video Java Library
- Oracle Video ActiveX Control

Oracle Video Web Plug-in

Use the Oracle Video Web Plug-in to create HTML documents containing streaming audio and video for Netscape-compatible browsers. You can customize the behavior of the plug-in at load time through the attributes to the **<EMBED>** tag used to include the video on your Web page. The plug-in also provides a LiveConnect interface that allows you to create controls on your page that can manage the plug-in after load time, through JavaScript or Java.

You can find usage information on the Oracle Video Web Plug-in in Chapter 2, "Oracle Video Web Plug-in". You can find reference information for the Oracle Video Web Plug-in in Appendix A, "Oracle Video Web Plug-in Reference".

Oracle Video Java Library

The Oracle Video Java Library consists of Java classes and interfaces that enable you to create Java applications that can play streaming audio and video, combining audio and video with any of your other needs.

Due to browser security restrictions, browser-embeddable applets created with the Oracle Video Java Library may not function in all Java-enabled browsers. The library includes an embeddable applet, **PlayerApplet**, which has the same restrictions.

You can find usage information on the Oracle Video Java Library in Chapter 3, "Oracle Video Java Library". You can find reference information in Appendix B, "Oracle Video Java Library Reference".

Oracle Video ActiveX Control

You can embed the Oracle Video ActiveX Control into most applications that support ActiveX control embedding. For example, you can use it in an HTML document for Microsoft Internet Explorer or embed it in an Oracle Power Objects or Visual Basic application. You can manage the control using scripting languages, such as VBScript and JScript. You can also use the Oracle Video ActiveX Control in Oracle Forms.

You can find usage information on the Oracle Video ActiveX Control in Chapter 4, "Oracle Video ActiveX Control". For information on using the control in Oracle Forms, see Chapter 5, "Working with Oracle Forms". You can find reference information for the Oracle Video Web Plug-in in Appendix C, "Oracle Video ActiveX Control Reference".

Choosing a Client Interface

To a large extent, the primary target platform and preferred application environment for your client application determines which client interface you should choose. You also need to consider *how* you want to use the video client. There are two main types of clients you can develop:

- Browser-Hosted Client Applications
- Stand-Alone Client Applications

Browser-Hosted Client Applications

Browser-hosted clients generally consist of an HTML page (static or dynamic) with the plug-in or ActiveX control embedded in the page:

- If your target browser is Netscape Navigator or Communicator, use the Oracle Video Web Plug-in.
- If your target browser is Microsoft Internet Explorer, use the Oracle Video ActiveX Control. Internet Explorer 3.0 and later supports Netscape plug-ins, but not plug-in scripting.

You can also add scripting to an HTML document. In Netscape browsers, use either JavaScript or Java. In Microsoft Internet Explorer, use VBScript or JScript. Scripts can be invoked based on normal events, such as mouse-over, click, and focus events. You can use scripts to create custom interfaces that follow your look-and-feel standards.

Figure 1–1 shows an example of a browser-hosted client.

Figure 1–1 Example of a Browser-Hosted Client

The buttons shown in Figure 1–1 use simple JavaScript methods to start and stop the video and turn looping off and on. Users can click the buttons on the page to turn video playback on and off and can also set the video to loop automatically.

Stand-Alone Client Applications

There are two ways to create stand-alone applications using OVC:

- Using an ActiveX-enabled development tool, such as Visual Basic or Oracle Forms, add the Oracle Video ActiveX Control to the development tool's control palette or equivalent. You can then use the client control in your applications the same as any other custom control.
- Use the Oracle Video Java Library in a Java application. The Java classes provide full control over the client and easy programmatic access to all the features of the client.

Your choice depends on the tools you prefer and the platforms on which you wish to run your client application. If you are developing for Windows platforms on *x*86 hardware, you can use the ActiveX control. If you are concerned about platform independence, you should develop your client application using the Oracle Video Java Library.

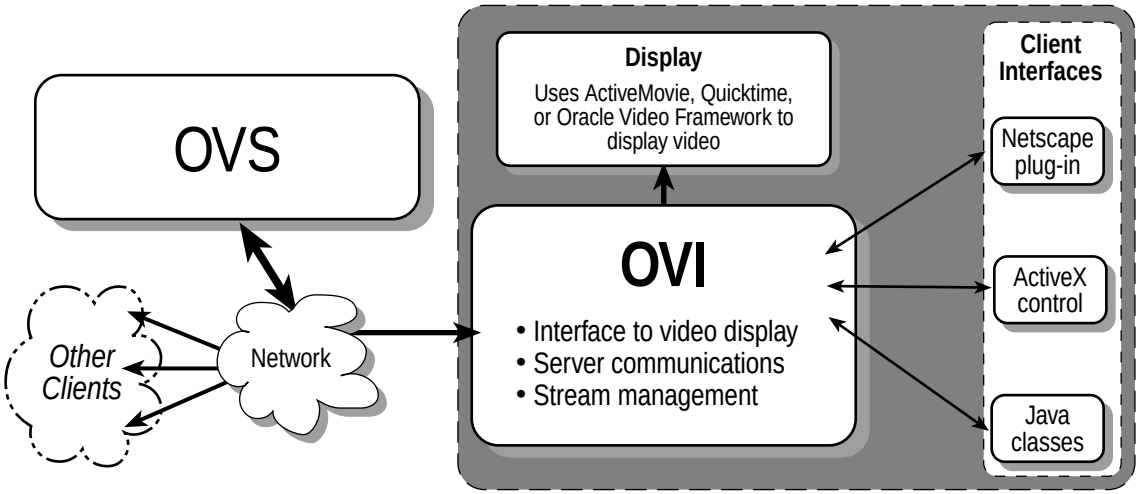
Client Software Components

This section discusses the components that constitute the Oracle Video Server system, from the video server to the viewing client. The system has three parts:

- The Oracle Video Server provides streaming video and audio, as well as a number of supporting services, to the client machine.
- The Oracle Video Interface, running on the client, manages the streams from OVS, handles messages and transactions to and from OVS, and controls the display technology.
- The OVC client interfaces provide user control of streams, set up the display area, and handle transactions with the client environment.

Figure 1–2 show the various components in the system.

Figure 1–2 Software components of the Oracle Video Client



OVS is the backbone of this system. It stores, retrieves, and dispatches media content files throughout the system. You can find out more about OVS in *Introducing Oracle Video Server.*

OVI is the video client's gateway to the OVS system. It provides the following services:

- communicates with the client interface, passing requests from the interface on to the video server
- handles network communications between the client machine and the video server, including managing video and audio streams as they arrive from the video server
- communicates with the playback and display technology to play back the incoming stream using standard display technologies such as Oracle Video Framework and ActiveMovie

Developing and Deploying a Client Application

This section discusses the process of developing your own client application, as well as how to make your new client available to your users. There are a number of things to consider:

1. Decide which client interface you want to use. See "Choosing a Client Interface" on page 1-4.
2. Decide how much functionality you want to make available to your users. Each client interface provides control over the stream playback, but you can specify how much of the provided functionality to make available to the user.

   For example, suppose you want to use the ActiveX control. This control provides the ability to play, stop, rewind, fast forward, and pause the video. But you're creating a kiosk application, where you don't want your end users to be able to interrupt the video. You can turn the controls off, so that the user can't manipulate the video at all, or add only volume controls so that the user can change the volume, but not interrupt the video.
3. Before any of your end users can use your new client application, they must install the minimum Oracle Video Client installation. This includes OVI, as well as whichever client interfaces your application requires. Make the OVC installation available to users, for example, through a Web page or shared network resource. You then need to give instructions to your users on how to install OVC, such as which client interfaces they require.

   You can also install OVC as part of the installation process for your client application. To find out whether the proper version of OVC is already installed on the user's system, check the Windows registry for the Oracle Video Client version. This key is located in:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Oracle\Oracle Video Client
   ```

There are further considerations that depend on the client interface you choose:

- If you are creating Web page applications using the Oracle Video Web Plug-in or the Oracle Video ActiveX Control, the “application” is really the HTML documents themselves. The documents embed the plug-in or control, but doesn’t require the user to download the client interface, since that should have been installed along with OVC.

  The only exception is if you use a Java applet to control the plug-in through the LiveConnect interface. In that case, the user has to download the applet. This works just like downloading a normal Java applet.

- If you create a stand-alone application using the Oracle Video ActiveX Control or Oracle Video Java Library, you need to make the application available to your users, just as you make the OVC installation available to them.

2

Oracle Video Web Plug-in

This chapter describes the Oracle Video Web Plug-in. This Netscape-compatible plug-in enables you to create Web pages and applications that contain streaming video and audio from the Oracle Video Server. You can use Java or JavaScript to control the plug-in, allowing you to add controls and interfaces and tailor the amount of end-user control.

This chapter contains these sections:

If you install the Typical configuration or choose to install the sample applications in the Custom configuration, your client installation includes a sample that uses the Oracle Video Web Plug-in. Open the file **index.htm** in the directory **vc30\demo\webplugin** in your **ORACLE_HOME**.

You can find reference information about the Oracle Video Web Plug-in in "Oracle Video Web Plug-in Reference" on page A-1, including valid attributes for the **<embed>** HTML tag and methods available on the plug-in itself.

In order to gain the most from this chapter, you should be familiar with HTML, JavaScript, and Java. Specifically, knowledge of the following would be helpful: embedding Netscape-compatible plug-ins in HTML documents, responding to basic events such as mouse clicks, and embedding applets in HTML documents.

Introduction to the Oracle Video Web Plug-in

The Oracle Video Web Plug-in provides a screen for video playback, audio playback, as well as optional controls and status line. Using the plug-in is easy: place the Oracle Video Web Plug-in in your HTML document by adding an **<embed>** statement to an HTML document. Within the **<embed>** statement, you can set attributes to modify the appearance and behavior of the plug-in. For example, you can specify whether the controller appears or whether the video starts playing automatically. You can even make the plug-in completely invisible if you're using the plug-in for audio playback only.

Within Netscape Navigator version 3.0 or later, you can customize the plug-in's functionality by adding JavaScript or Java graphical user interfaces to dynamically control the plug-in. The LiveConnect interface provides easy access to the control from either language. Starting and stopping video, modifying the volume or stream position, and even changing the current media file are all possible.

Each time a user loads a page that uses the plug-in, the **<embed>** statement loads the Oracle Video Web Plug-in automatically (the user first needs to install the Oracle Video Client). The plug-in appears in the browser as a video screen (unless hidden), with some optional components

- Controller with:
  - Play / Pause button
  - Stream position slider
  - Volume slider
- Status line

Controls that you design using JavaScript or Java appear wherever you place them.

Depending on the attributes set in the **<embed>** statement, the plug-in can connect to the video server immediately upon being loaded and play a predetermined video. Or you can make the plug-in load, but not play until the user clicks the Play button. If you are using the built-in controls, the Play button turns into the Pause button when playback begins, and becomes the Play button again when playback stops. When the page is closed, the video stream is automatically deallocated and the plug-in is unloaded by the browser.

Figure 2–1 Plug-in with controls and status line visible

The plug-in also offers other means of control:

- Click the video screen to start the video. To pause, click the video screen again.
- Right-click the plug-in to display a pop-up menu with the following options: Play, Pause, Rewind to Start of Movie, Forward to End of Movie, and more. Choose Play from the pop-up menu to start the video. Choose Pause to pause it.
- The LiveConnect interface lets you call stream control methods on the plug-in through JavaScript or Java.

Requirements

The Oracle Video Client must be installed on all machines on which you want to use the Oracle Video Web Plug-in. In addition to the standard client requirements, the Oracle Video Web Plug-in requires a browser that supports Netscape plug-ins, such as Netscape Navigator 2.0 or greater, or Microsoft Internet Explorer 3.0 or greater.

To control the plug-in with JavaScript or Java, the Oracle Video Web Plug-in requires a browser that supports the Netscape LiveConnect interface, such as Netscape Navigator 3.0 or greater.

Installing the Oracle Video Web Plug-in

To install the Oracle Video Web Plug-in, run the OVC installer and either:

- Select the Typical installation configuration, or
- Select the Custom installation configuration, making sure you select the Oracle Video Web Plug-in check box, when prompted.

When you elect to install the Oracle Video Web Plug-in, the installer automatically installs it for any browsers on your system that support Netscape plug-ins. The plug-in file **npovc.dll** is installed in the default plug-in directory for each browser. When you restart your browser, the plug-in is configured and ready to use. If your browser was running during the installation, you need to close and re-open it.

You can find the complete installation instructions for the Oracle Video Client in the CD insert that came with your installation CD-ROM. This information is also available in electronic form in the file **ovcinstl.pdf** in the **\docs** directory of your installation CD-ROM and, if you selected the Typical installation configuration or selected the Docs option in the Custom configuration, in the **vc30\docs** directory of your **ORACLE_HOME**.

Associating Oracle Video Files with the Oracle Video Web Plug-in

When you download a data file in a network application, such as your browser or e-mail program, you want to be able to handle the data properly without having to save it to your hard drive and manually start another application. For example, if you download a sound file, you don't want to have to save the file, start your sound player, load the file, then finally play the sound. You just want to hear the sound played over your speakers.

The problem is that your browser or e-mail program doesn't necessarily know what application to launch for a particular type of data. This is handled through MIME types (for Multimedia Internet Message Extensions), which provide a way of associating a particular type of data with a particular application. There are three parts to a MIME type:

- **MIME type name.** This comes in the form *content-type*/*subtype*. *content-type* is a broad classification of the data, such as *image*, *application*, or *video*. *subtype* defines the type more specifically. For example, MIDI files are commonly given the MIME type **audio/midi**.

- **Associated file extensions.** In order to identify a file as having a particular MIME type, you either have to know that the file has that MIME type, such as through data packet headers, or find some way of extracting this from the information that you have about the file. Most browsers and HTTP servers use the file extension for this. Thus, each MIME type has one or more file extensions associated with it.
- **Associated application.** This specifies the application that can handle data of a particular MIME type.

Oracle video files are accessed through tag files, which have the extension **.mpi** (for more information on Oracle video files and Oracle video tag files, see the *Oracle Video Server Content Administrator's Guide*). The Oracle Video Web Plug-in requires that Oracle video files be associated with the plug-in through the MIME type **application/oracle-video**. When this MIME type is properly configured, you can access data files with the extension **.mpi**, which are associated with the Oracle Video Web Plug-in, through the MIME type.

There are two places you need to configure the Oracle Video MIME type:

- Client MIME Configuration
- Server MIME Configuration

Client MIME Configuration

Under normal circumstances, your browser automatically configures which MIME types it supports when it starts up. Netscape's plug-in specification requires that browsers check their plug-in directory to see which plug-ins are installed. The browser then queries each plug-in to find which MIME type or types the plug-in supports. Then, when the user tries to access data with a supported MIME type, the browser uses the appropriate plug-in to handle the data.

When you embed an Oracle video file in an HTML document, you need to specify the file's MIME type so the browser knows to load the Oracle Video Web Plug-in. There are two ways to do this:

- The **mediafile** and **src** attributes; **mediafile** indicates to the plug-in which media file to open, while the **src** attribute, recognized by the browser, indicates a dummy file with the **.mpi** extension. The dummy file's name causes the browser to properly set the MIME type for the data and load the plug-in, while the plug-in finds the actual media file to load through the **mediafile** attribute.
- The **mediafile** and **type** attributes; **mediafile** indicates to the plug-in which file to open, while **type** explicitly specifies the MIME type of the embedded object.

Note: The **type** attribute only works in Netscape 3.0 or greater. For other browsers, use the **src** and **mediafile** attributes; see the discussions of these attributes starting on page 2-8.

Although only the extension **.mpi** is associated with the Oracle Video Client through the MIME type **application/oracle-video**, OVC can also play Oracle Stream Format (extension **.osf**) and regular MPEG files (extension **.mpg**). You can embed these files in HTML documents by using the **src** or **type** attribute to specify the MIME type, while the **mediafile** attribute tells the plug-in which media file to actually load. The extension of the media file does not affect the loading of the plug-in. See "Specifying the Media File and MIME Type" on page 2-8 for an example of how to do this.

Server MIME Configuration

If you plan to serve video through HTML pages that embed the Oracle Video Web Plug-in, you need to configure your HTTP server to handle the **application/oracle-video** MIME type. The HTTP server recognizes the MIME type of requested data by the file extension. It then returns the data with the appropriate MIME type specified in the HTTP packet header. The browser maintains its own list of MIME types, which includes the appropriate application or plug-in to load for supported MIME types. The browser loads the appropriate plug-in based on that information.

Configure your HTTP server to associate the extension **.mpi** with the **application/oracle-video** MIME type. Consult the documentation for your HTTP server for information on configuring MIME types.

Embedding the Oracle Video Web Plug-in in an HTML Document

To embed the plug-in into an HTML document, insert an **<embed>** statement with the appropriate attributes in your HTML code. The **<embed>** statement uses attributes to determine which video plays (the **mediafile** attribute), whether the video starts as soon as the plug-in loads (the **autoStart** attribute), whether the video replays every time it ends (the **loop** attribute), and more. See "<Embed> Attributes" on page A-1 for definitions of all of the available **<embed>** attributes.

This section contains the following topics:

Creating the <Embed> Tag

The example below shows a simple HTML document with an **<embed>** tag.

```
<html>
<head>
<title>Oracle Video Web Plug-in Example</Title>
</head>

<body>

<embed type="application/oracle-video" name="video1" width=352
   height=240 mediafile="vstcp://ovs-sun:5000/mds/video/oracle1.mpi">

</body>
</html>
```

where:

type="application/oracle-video"
Specifies the MIME type. In this case, it invokes the plug-in and prepares it to play an Oracle video file.

> **Note:** The **type** attribute only works in Netscape 3.0 or greater. For other browsers, use the **src** and **mediafile** attributes; see the discussions of these attributes starting on page 2-8.

name="video1"
Provides a name for the plug-in for JavaScript and Java calls. In this case, the plug-in is named **video1**. This enables you to call methods on the plug-in using the syntax **document.video1.*method*()**.

width=352
Specifies the plug-in width of 352 pixels. This value includes both the plug-in itself and the border, if one was specified. In this case, no border was specified, so this represents the actual width of the plug-in.

height=240
Specifies the plug-in height of 240 pixels. This value includes both the plug-in itself, the border, if one was specified, and the optional controller and status line. In this case, no border was specified and the controls aren't displayed, so this represents the actual height of the plug-in.

mediafile="vstcp://ovs-sun:5000/mds/video/oracle1.mpi"
Tells the plug-in to go to port 5000 on the server ovs-sun and return the tag file **oracle1.mpi** located in the MDS volume **video**. The resulting stream uses the TCP protocol. For a complete description of the syntax, see the **mediafile** attribute discussion in Appendix D, "The Media File".

Specifying the Media File and MIME Type

To embed an Oracle Video Server file into an HTML document, specify the actual media file to be loaded and played using the **mediafile** attribute of the **<embed>** tag. Browsers themselves don't recognize the **mediafile** attribute: they just pass it on to the plug-in, which uses it to locate the media resource. The browser still needs to identify and recognize the MIME type of the embedded item in order to load the proper plug-in for playback. Use the following methods to identify the desired media file and the MIME type of that media file:

- Type and Mediafile Attributes
- Src and Mediafile Attributes

Type and Mediafile Attributes

The preferred way to specify the MIME type of the embedded media is to use the **type** attribute. The browser checks its list of registered MIME types and loads the appropriate plug-in for that type. The plug-in uses the **mediafile** attribute to load the appropriate media file.

For Oracle Video Server media files, you should always specify the MIME type **application/oracle-video**. When you install the plug-in, the installer associates files with a MIME type of **application/oracle-video** with the Oracle Video Web Plug-in.

Example: This sample **<embed>** tag shows how to embed a media file from the server ovs-sun with the appropriate MIME type.

```
<embed type="application/oracle-video" width=352 height=240
mediafile="vsudp://ovs-sun/mds/video/oracle1.mpi">
```

> **Note:** The **type** attribute only works in Netscape 3.0 or greater. For other browsers, use the **src** and **mediafile** attributes; see the discussions of these attributes starting on page 2-8.

Src and Mediafile Attributes

The **src** attribute should be used for all browsers that don't support the **type** attribute (any browser other than Netscape Navigator 3.0 or greater). Browsers inspect the **src** attribute to determine the MIME type of the embedded file from the file's extension. This works like the **type** attribute: the browser associates **.mpi** files with the MIME type **application/oracle-video**, which is associated with the Oracle Video Web Plug-in.

However, there are a couple of things to be aware of when you're using the **src** attribute:

- The **src** attribute actually specifies a file name, instead of just a plain MIME type like the **type** attribute. This file does not have to be the same as the file specified by the **mediafile** attribute. It must, however, be a real file, specified by a valid URL or path, with the extension **.mpi**.

  The Oracle Video Web Plug-in sample pages handle this by setting the **src** attribute to point to the file **oracle.mpi**, which is installed in the same directory as the sample HTML pages.

- Some browsers can't handle “empty” or “dummy” files (files that are zero-length and contain no data) specified by the **src** attribute. This means that you should specify a file that contains some data of any type.

  Again, the Oracle Video Web Plug-in sample pages handle this with the **oracle.mpi** file, which contains 1K of data.

Example: This sample shows how to embed a media file using the **src** attribute to indicate the MIME type of the media file.

```
<embed src="muse.mpi" mediafile="c:\tmp\muse.mpg" width=352 height=240>
```

Notice that the **src** and **mediafile** attributes specify files with different names and even different extensions. The **.mpi** extension in the **src** attribute instructs the browser to load the Oracle Video Web Plug-in, while the **mediafile** attribute instructs the plug-in to load the file **c:\tmp\muse.mpg**.

Specifying Plug-in Characteristics

The Oracle Video Web Plug-in also recognizes a number of other attributes from the **<embed>** tag. Table 2–1 summarizes these attributes and their effects on the plug-in's look-and-feel and play characteristics. You can find more detailed descriptions of each of these attributes in Appendix A, "Oracle Video Web Plug-in Reference".

Table 2–1 <embed> tag attributes

Attribute	Values—default in italics	Description	Req'd?
autoStart	true | *false*	Specifies whether video starts playing as soon as the plug-in is loaded.	
background	"file"	Specifies a background image	
controls	true | *false*	Specifies whether the plug-in appears with or without controls.	
controlMask	*"controller"* [+] "statusline"	Selects which controls appear in the plug-in; requires the added statement **controls=true**.	
height	nnnn	Specifies the height of the plug-in in pixels.	✔
hidden	true | *false*	Specifies whether the plug-in is displayed. Overrides **height** and **width** settings.	
leftClick	*true* | false	Specifies whether left clicking in the video screen toggles Play and Pause.	
loop	true | *false*	Specifies whether the stream plays continually, returning to the **playFrom** position when it reaches the **playTo** position and continuing playback	

Table 2–1 <embed> tag attributes

Attribute	Values—default in italics	Description	Req'd?
mediafile	mediafile_url	Specifies the protocol, server, and media file to play. ***Note:*** For information on how the **mediafile**, **src**, and **type** attributes work together, see "Specifying the Media File and MIME Type" on page 2-8. For information on media file specifiers, see Appendix D, "The Media File".	✔
name	plugin_name	Specifies a local name for the plug-in. ***Note:*** This parameter is required if you call the plug-in from a JavaScript function or Java applet embedded in the same HTML page as the plug-in.	✔ (See note)
playFrom	{*"beginning"* |"end" |hh:mm:ss:cc |millisecs}	Starts play at the specified stream position. If the **loop** attribute is **true**, then playback loops to the **playFrom** point.	
playTo	{"beginning" |*"end"* |hh:mm:ss:cc |millisecs}	Stops play at the specified stream position. If the **loop** attribute is **true**, then playback loops when it reaches the **playTo** position.	
popupMenu	*true*|false	Specifies whether the pop-up menu appears when the right mouse button is clicked on the plug-in. This menu allows basic operations like play and pause, rewind, forward, and close.	
sliderRate	nnnn	Specifies the increment in milliseconds by which you can adjust the plug-in's seek slider. The default is 1000.	

Table 2–1 <embed> tag attributes

Attribute	Values—default in italics	Description	Req'd?
src	file.extension	Specifies the source file and MIME type of the requested media file. Should be used in conjunction with **mediafile**. ***Note:*** If you don't use **src**, you must use **type**. See "Specifying the Media File and MIME Type" on page 2-8 for more information on **type**, **src**, and **mediafile**.	✔ (See note)
toolTips	"true | *false*"	Specifies whether tool tips show up when the user moves the mouse over the plug-in.	
type	application/oracle-video	Specifies the media file MIME type. The only value you should specify for **type** attribute is **application/oracle-video**, as shown. Should be used in conjunction with **mediafile**. ***Note:*** If you don't use **type**, you must use **src**. See "Specifying the Media File and MIME Type" on page 2-8 for more information on **mediafile**, **src**, and **type**.	✔ (See note)
volume	vol	Specifies a volume level from 0 (inaudible) to 100.	
width	nnnn	Specifies the plug-in's width in pixels.	✔

Playing Audio-Only Streams

You can run the plug-in using audio streams without video, with or without controls:

- If you don't need the controller or status line, you can hide the plug-in by specifying the **hidden** plug-in tag or by specifying a width and height of 0. In this case, you can only control the stream through JavaScript or Java calls.
- To display the controller, add 24 pixels to the **height** attribute. The **width** attribute should be set to at least 144 pixels to allow some slider range. Using the default controls, the stream is controlled by a Play / Pause button and the seek slider. The volume is set by the pop-up volume slider or by setting the **volume** attribute.
- If you need the status line, add an additional 21 pixels to the **height** attribute.

Controlling the Plug-in Using JavaScript and Java

You can dynamically control the Oracle Video Web Plug-in using JavaScript and Java through the Netscape LiveConnect interface. For example, you can have buttons or icons that play the current stream, pause, seek, pop up lists of available movies, and any other type of custom control or function that you want to develop. You can see an example of this in Figure 2–2.

> **Note:** Because you need to use Netscape's LiveConnect interface to control plug-ins through Java or JavaScript, you must use a browser that supports LiveConnect. This means Netscape Navigator version 3.0 or later.

This section contains the following topics:

- "Controlling the Plug-in with JavaScript" on page 2-15
- "Controlling the Plug-in with Java" on page 2-23

You can also examine the sample code files that were installed along with the Oracle Video Client. The directories are in your **ORACLE_HOME**:

- JavaScript: **vc30\demo\webplugin\samples\liveconnect\javascript**
- Java: **vc30\demo\webplugin\samples\liveconnect\java**

Run these sample applets and applications to better understand how plug-ins work. Feel free to modify and re-use these code examples in your Oracle Video Client custom applets and applications. You can find a reference to all of the methods available on the Oracle Video Web Plug-in in "JavaScript Methods" on page A-8.

Figure 2–2 One way to customize a plug-in using JavaScript or Java

Controlling the Plug-in with JavaScript

This section describes a number of techniques you can use to control and customize the Oracle Video Web Plug-in, including:

Naming an Embedded Plug-in

To give JavaScript a handle on the plug-in, specify a value for the **name** attribute in the **<embed>** statement:

```
<embed name="video1" width=352 height=240 type="application/oracle-video"
mediafile="vsudp:///mds/video/oracle1.mpi">
```

The name you specify should be unique within the HTML document. This includes other named items, such as images, tables, forms, and so on. This is the name you'll use whenever you want to address a property or method of the plug-in.

Once you've named your plug-in, you can access it through the **document** object from anywhere within your document by that name. For example, if you want to call the **play()** method for the plug-in named **video1**, use something like this:

```
document.video1.play();
```

Accessing Plug-in Methods and Properties

You can access the Oracle Video Web Plug-in in several ways using JavaScript. The easiest is to load the HTML file that contains the **<embed>** statement, enter a JavaScript statement as a URL in the **Location** box of Netscape Navigator (located right above the main browser window), then press **Enter**.

To start playing the video specified in the example above, you would enter the following statement in the **Location** box:

```
javascript:document.video1.play()
```

To pause the video using the **Location** box, enter:

```
javascript:document.video1.pause()
```

Controlling the Plug-in with Form Buttons

Entering commands in the **Location** box has limited value. Using JavaScript calls to form elements—such as buttons, selection lists, or edit boxes—is more useful.

For example, you can create a form button in your HTML page that calls the **play()** method. The following example creates a button labelled **Play**. When you click this button, the browser searches for an item in the document whose **name** attribute value is **video1**. Once it finds the video, it calls the plug-in's **play()** method, which instructs the plug-in to play the movie specified in the **mediafile** attribute of the **<embed>** statement.

```
<embed name="video1" width=352 height=240 type="application/oracle-video"
mediafile="vsudp:///mds/video/oracle1.mpi">

<form name="form1">
<input type="button" value="play" onClick="document.video1.play()">
</form>
```

Using Graphical Controls


Play icon

You can let users control the plug-in using simple form buttons, but it's more "Web-like" to use a graphical icon to control the video. This example plays the video when the user clicks the **/images/play.gif** image.

```
<embed name="video1"
       width=352
       height=240
       type="application/oracle-video"
       mediafile="vsudp:///mds/video/oracle1.mpi">

<a href="javascript:document.video1.play();">
<img src="/images/play.gif" width=24 height=24 border=0></A>
```

Controlling the Plug-in with Dynamic Parameters

You can extend the concepts introduced in the previous sections and create an input field, which can contain numeric or textual data, in the form. In this example, the user can type a number representing a position in the stream into the input field. When the user clicks the button, the **setPos()** method sets the position in the movie to the value entered in the input field.

```
<embed name="video1" width=352 height=240 type="application/oracle-video"
mediafile="vsudp:///mds/video/oracle1.mpi">

<form name="form1">
<input name="editSeek" type="text" value=0 size=10>
<input type="button" value="Seek"
   onClick="document.video1.setPos(parseInt(form.editSeek.value))">

</form>
```

Be sure to validate the user's input. For the sake of clarity, this example contains no validation code. In a deployment environment, however, you should make sure that the values that users input are valid.

Creating a Pop-up List

When creating custom controls or interfaces using JavaScript and the Oracle Video Web Plug-in, you can use all of the techniques that are normally available to you. The following code allows the user to select a movie from a selection list. When the user selects a new movie from the list, the plug-in loads the movie and starts playing it. This also creates a **Close** button, allowing the user to turn off a currently playing movie.

```
<html><head>
<title>Pop-up Test</title>

<script language="JavaScript">
function playSel()
{
   // get a handle on the pop-up list
   var mlist = document.forms[0].movielist;

   // get the name of the movie to load
   var movie = mlist.options[mlist.selectedIndex].value;

   if (movie != "")// if movie is not empty
   {
       document.video1.load(movie); // load the selected movie
       document.video1.play(); // play the selected movie
   }
}
</script>
</head>

<body>
<embed name="video1" width=352 height=240 type="application/oracle-video"
mediafile="vsudp:///mds/video/oracle1.mpi">

<form name="form1">
<select name="movielist" onChange="playSel()">
   <option value="">Default
   <option value="/mds/video/oracle.mpi">Demo
   <option value="/mds/video/oracle1.mpi">Demo 1
   <option value="/mds/video/oracle2.mpi">Demo 2
   <option value="/mds/video/oracle3.mpi">Demo 3
</select>
<input type=button value="Close" onClick="document.video1.close()">
</form>
</body></html>
```

Using an Image Map

Using client-side image maps can greatly improve the appearance and functionality of a web page. Image maps give the greatest amount of flexibility possible in designing your client interface. They allow you to control the appearance, overall interface metaphor, and user environment. By associating hot spots in the image map to methods and properties of the Oracle Video Web Plug-in, you can produce a sleek, professional interface that still offers the full functionality of the OVC video client.

Spin control

This example shows how to create a spinner that increases or decreases the value in the **editSeek** field in one-second increments.

```
<script language="JavaScript">
function spinUp()
{
   // get the current value of the editSeek field
   var seekVal = parseInt(document.forms[0].editSeek.value);

   // add one second (1000 ms) to the value of the editSeek field
   document.forms[0].editSeek.value = seekVal + 1000;
}

function spinDown()
{
   // get the current value of the editSeek field
   var seekVal = parseInt(document.forms[0].editSeek.value);

   // make sure you don't create a negative number (1000 ms = 1 second)
   if (seekVal < 1000) seekVal = 1000;

   // subtract one second (1000 ms) from the value of the editSeek field
   document.forms[0].editSeek.value = seekVal - 1000;
}
</script>

<!-- Create the image map -->
<img border=0 src=/images/spin.gif usemap="#spinmap">
<map name="spinmap">

<!-- Associate the up arrow in the image with spinUp() -->
<area shape="rect" coords="0,0,11,10" href="javascript:spinUp()">

<!-- Associate the down arrow in the image with spinDown() -->
<area shape="rect" coords="0,10,11,20" href="javascript:spinDown()">
</map>
```

Other Things You Can Do with JavaScript

The examples above are just a few of the things you can do with JavaScript. Some other ideas:

- Create a single image that contains all of the typical VCR controls. Use a client-side image map to correlate mouse click locations with **play()**, **pause()**, and other plug-in methods.
- Create a client-side image map with various hot spots. Map each hot spot to a different seek point (or even a completely different stream) so that when the user clicks on a location, the video changes. For instance, you could create a map of the United States displaying the hometowns of each of the teams in the National Football League. Users click a city to play a movie about that city's home team.
- Create a custom video stream position scrollbar. Get an image you would like to use as a scrollbar. Use a client-side image map to split it up into many sub-regions, each of which seeks to a slightly different position in the stream.
- Create a function that checks stream position and updates a text field or graphic, or changes a document in a frame at various key points in the stream.

Sample Code for JavaScript-Controlled Plug-in

This section shows a very simple sample HTML file that uses JavaScript to let the user control the plug-in. The form buttons below the plug-in let the user play and pause the currently loaded video, or unload the video entirely.

You can find the source code for this example in the file **defvid.htm** in your **ORACLE_HOME** in the directory:

vc30\demo\webplugin\samples\liveconnect\javascript\simple\sample1

Figure 2–3 Sample Plug-in using JavaScript


```
<html><head>
<title>Oracle Video Web Plug-in Simple JavaScript Demo</title>
<script language="JavaScript">
function play() {
   // If player is currently in state realized --> call play
   if (document.OVSplugin.getState() == 4) {
       if (document.OVSplugin.play())
       return true;
   }
   // if player is in any other state --> call resume
   else {
      if (document.OVSplugin.resume())
       return true;
   }
   return false;
}
</script>
</head>
```

```
<body bgcolor="#FFFFFF">
<center>
<font size="1" face="Arial" color="Red">
(Netscape 3.0 or greater only!)
</font><p>

<table border=5 cellpadding=0 cellspacing=0 align=center valign=center>
<tr align="center" valign="middle">
   <td align="center" valign="middle">
   <!-- Embed the Oracle Video Client Web Plug-in  -->
   <embed
       name="ovsplugin"
       border=0
       width=240
       height=200
       type="application/oracle-video"
       autoStart="false"
       controls="false"
       loop=true
       leftClick=true
       toolTips=false
       popupMenu=true
       volume=100
       mediafile="vstcp:///mds/video/ovs_mpg1_2048k.mpi">
   </embed>

   <center>
   <form name=form1>
       <input type=button onclick="play()" value=play>
       <input type=button onclick="OVSplugin.pause()" value=pause>
       <input type=button onclick="document.OVSplugin.unload()" value=close>
   </form>

   </td>
</tr>
</table>
</body>
</html>
```

Controlling the Plug-in with Java

OVC provides two Java classes, **OviPlayer** and **OviObserver**, to allow interaction between the plug-in and your Java applets.

This section covers:

- OviPlayer and OviObserver
- Retrieving the OviPlayer Object

OviPlayer and OviObserver

OVC provides a Java class and an interface that let you control and observe the Oracle Video Web Plug-in from your applet:

- The **OviPlayer** class declares methods to access and control video. This class allows basic operations on a video like **load()**, **unload()**, **play()**, **pause()**, **getLength()**, and **getPos()**. The real implementation of these methods is contained in a section of native code contained in the plug-in.
- The **OviObserver** interface lets you take advantage of "observer" methods. These methods provide a callback mechanism that notifies you when the stream has advanced or reached the end of stream. You can then respond with some action that you define.

Table 2–2 and Table 2–3 give short descriptions of the methods available in the **OviPlayer** and **OviObserver** classes. You can find more detailed references of these classes in "OviPlayer" on page A-9 and "OviObserver" on page A-15.

Table 2–2 OviPlayer methods

Method	Description
advise()	Specifies the **OviObserver** object for this **OviPlayer**.
forward()	Forwards the stream to the end of the movie or the position indicated by the **playTo** property.
getLength()	Gets the total length of the stream (in milliseconds).
getMaxPos()	Returns the maximum stream position.
getMinPos()	Returns the minimum stream position.
getObserver()	Returns the **OviObserver** object associated with this object, if any.
getPos()	Gets the current stream position (in milliseconds).
getState()	Returns the current state of the player.
getVol()	Gets the play volume (0 to 100).

Table 2–2 OviPlayer methods

Method	Description
load()	Loads the stream indicated by the media file specifier contained in the **String** parameter.
pause()	Pauses the stream; stays at current position.
play()	Plays the currently loaded stream. There are two versions: ▪ **boolean play()** ▪ **boolean play(String playFrom, String playTo)**
resume()	Resume playing from the current stream position.
rewind()	Rewind the stream to the beginning of the movie or the position indicated by the **playFrom** property
setAutoStart()	Tell the movie to automatically play when loaded. Takes a **boolean** parameter.
setFullScreen()	Puts the plug-in into full-screen mode.
setLoop()	If true, loop from beginning of stream (or **playFrom** if specified) when the end of stream (or **playTo**, if specified) is reached. Takes a **boolean** parameter.
setPopupMenu()	Activate or deactivate the right mouse button pop-up menu. Takes a **boolean** parameter.
setPos()	Seek to a specified stream position (in milliseconds). After this command, the movie is in the pause state.
setVol()	Set the play volume (range from 0 to 100).
stop()	Stop playing the stream and rewind to the beginning of the stream or the position indicated by the **playFrom** property.
unload()	Unload the current stream.

Table 2–3 OviObserver methods

Method	Description
onPositionChange()	Notify the applet when the movie changes position by increments. The increment is set by the **sliderRate** attribute. If **sliderRate** isn't set, the default increment is 1,000 milliseconds (one second).
onStop()	Notify applet when end of stream reached.

Retrieving the OviPlayer Object

You cannot simply create an **OviPlayer** object in your Java code and invoke the methods on the object, because the Java run-time engine can't resolve the references to the native code in the plug-in and produces unresolved link errors. Instead, retrieve the instance of the **OviPlayer** object used internally by the plug-in. To do this, you must import the **OviPlayer** class, as well as the **JSObject** and the **JSException** classes from the Netscape LiveConnect / Plug-in SDK. You can then follow these steps to get a valid **OviObject** instance that references the plug-in:

1. Retrieve the JavaScript context of the current HTML page using the **JSObject.getWindow()** method. The **getWindow()** method takes a single parameter, the window containing the applet.

   The **JSObject.getWindow()** method is static, so you don't actually need to create a **JSObject** object. The return value is a **JSObject**.

2. Call the **getMember()** method on the **JSObject** object returned in the last step.

   This method takes a **String** parameter, where the **String** specifies the name of the member you want. Use "document" as the parameter. This tells the **getMember()** method to retrieve the document object for the current page. The **getMember()** method returns a **JSObject**.

3. Call the **getMember()** method on the **JSObject** object returned in the last step, passing the plug-in name (from the **name** attribute in the **<embed>** tag) as the parameter to **getMember()**. Cast the return value to **OviPlayer**.

Once you've retrieved the **OviPlayer** object, you can use it just as you'd expect: you can load media files with the **load()** method, play the current media file by calling **play()**, and so on.

```
// The following code gets a handle on the window in which
// the applet (and plug-in) appear
JSObject jsWindow = JSObject.getWindow(this);

// the next line of code gets a handle on the HTML document
// (web page) in which the applet (and plug-in) appear
JSObject jsContext = jsWindow.getMember("document");

OviPlayer playerInstance = (OviPlayer) jsContext.getMember("plugin_name")

playerInstance.load("vstcp:///mds/video/oracle1.mpi");
playerInstance.play();
```

Using OviObserver

The **OviObserver** interface is a Java interface that you implement in one of your own classes. To use **OviObserver** in your applet:

1. Implement the interface in one of your applet's classes.

   Here's a simple example:

   ```
   public class myApplet extends Applet implements OviObserver {
      // The rest of the class implementation goes here...
   }
   ```

2. Implement the **onStop()** and **onPositionChange()** methods of the **OviObserver** interface. Take whatever action you want when these events happen.

   Here's a simple example of an implementing class:

   ```
   class myApplet extends Applet implements OviObserver {
      // notify applet when end of stream reached and do something
      public void onStop() {
          System.out.println ("End of stream reached");
      }

   // notify applet when the stream position advances by one second
   // and do something, like keep track of were you are in the stream
      public void onPositionChange() {
          System.out.println ("The stream just advanced one second");
      }
   }
   ```

3. Retrieve an instance of the **OviPlayer** object, as described in "Retrieving the OviPlayer Object" on page 2-25.

4. Call the **OviPlayer.advise()** method, passing as a parameter the object that implements the **OviObserver** interface:

   ```
   // "this" is the object that implements the OviObserver interface.
   playerInstance.advise(this);
   ```

Once you've registered the observer with the **OviPlayer**, the **onStop()** and **onPositionChange()** methods are called whenever the appropriate events occur.

Simple Plug-in Example using Java

The following code example consists of two parts, the HTML code for the web page and the code for the Java applet. The file containing the Java code, **simpleJava.java**, becomes **simpleJava.class** after compiling it.

Figure 2–4 Sample Plug-in using Java

HTML Code for Simple Plug-in Java Applet This shows the HTML code for a document that contains the Oracle Video Web Plug-in and a Java applet that controls it. You can find the sample source in the file **defvid.htm** in your **ORACLE_HOME** in the directory:

vc30\demo\webplugin\samples\liveconnect\java\simple\sample1

```
<html>
<head>
<title>Simple Java Applet Example of Plug-in Controls</title>
</head>
<body bgcolor="#FFFFFF">
<center>
<table border=1 cellspacing=2 cellpadding=2>
<tr><td align="center" valign="middle" bgcolor="Gray">
<center>
 

<!-- Embed the Oracle Video Client Web Plug-in  -->
<embed
   name="ovsplugin"
   border=0
   width=240
   height=200
   type="application/oracle-video"
   controls=false
   loop=false
   leftClick=true
   toolTips=false
   popupMenu=true
   volume=100
   mediafile="vstcp:///mds/video/ovs_mpg1_2048k.mpi">
</embed>
<p> 
```

```
<!--   ##############################################
                    IMPORTANT!
       ##############################################
   -   The 'MAYSCRIPT="true"' flag in the applet tag is critical. This tells the
       browser's security manager to allow communication between the applet and
       the plug-in.
   -   The value of the applet parameter, pluginName, must agree with the
       'name' attribute of the embed object (the plug-in). In this case the
       plug-in name attribute has the value 'OVSplugin', and the below applet
       parameter, pluginName, also has the value 'OVSplugin'. This agreement of
       attribute / parameter values is imperative for communication between the
       plug-in and the Java applet. -->

<!--  Embed the Java Applet Controls  -->
<applet
   code=simplejava.class
   name=ovc_applet
   mayscript="true"
   width=320
   height=30>
       <param name=pluginname value=ovsplugin>
</applet>
</center>
</td></tr>
</table>
</center>

</body></html>
```

Java Applet Code for Simple Plug-in Example This shows the code for the Java applet that controls the Oracle Video Web Plug-in. You can find the sample source in the file **simpleJava.java** in your **ORACLE_HOME** in the directory:

vc30\demo\webplugin\samples\liveconnect\java\simple\sample1

```
//*****************************************************************************
// simpleJava.java:Applet
//*****************************************************************************
import java.applet.*;
import java.awt.*;
import netscape.javascript.JSObject;    //provides access to objects
import netscape.javascript.JSException; //provides access to objects

import OviObserver; //plug-in listener (not implemented)
import OviPlayer;   //OviPlayer interface

//===========================================================================
// Main Class for applet simpleJava
//
//===========================================================================
public class simpleJava extends Applet
{
   // Members for applet parameters
   // <type>       <MemberVar>    = <Default Value>
   //---------------------------------------------------------------------
   private String m_plgname = "";

   // GUI Objects
   //---------------------------------------------------------------------

   Button btnPlay = null;
   Button btnPause = null;
   Button btnClose = null;
   JSObject document, window;
   OviPlayer ovi = null;
```

```
// The init() method is called by the AWT when an applet is first loaded or
// reloaded.  Override this method to perform whatever initialization your
// applet needs, such as initializing data structures, loading images or
// fonts, creating frame windows, setting the layout manager, or adding UI
// components.
//----------------------------------------------------------------------------
public void init()
{

    // PARAMETER SUPPORT
    // The following code retrieves the value of the applet parameter
    //----------------------------------------------------------------------
    String param;

    // param: Parameter description
    //----------------------------------------------------------------------
    param = getParameter("pluginName");
    if (param != null)
        m_plgname = param;

    resize(320, 30);
    setBackground(Color.gray);

    btnPlay = new Button ("Play");
    btnPause = new Button ("Pause");
    btnClose = new Button ("Close");

    add(btnPlay);
    add(btnPause);
    add(btnClose);

    // GET JAVASCRIPT OBJECT (document)
    // Retrieve the the document object from the Browser environment
    //---------------------------------------------------------------
}

public void start()
{
}

public void stop()
{
}
```

```
public void destroy()
{
    ovi = null;
    document = null;
    window = null;
    System.gc();
}

// MOUSE SUPPORT:
// The mouseUp() method is called if the mouse button is released
// while the mouse cursor is over the applet's portion of the screen.
//---------------------------------------------------------------------------
public boolean mouseUp(Event evt, int x, int y) {

    // GRAB PLUG-IN OBJECT
    // Retrieve the plug-in object handle from the browser context
    //---------------------------------------------------------------------
    ovi = getPlugin();

    boolean cond = false;
    if(evt.target == btnPlay) {
        if (ovi.getState() == ovi.ST_REALIZED)
            ovi.play();
        else
            ovi.resume();
        cond = true;
    }

    if (evt.target == btnPause) {
        ovi.pause();
        cond = true;
    }

    if (evt.target == btnClose) {
        ovi.unload();
        cond = true;
    }

    return cond;
}
```

```
    // GET JS OBJECT HANDLES AND POINTER TO PLUGIN
    // Retrieves JavaScript Object contexts from Browswer.
    // This function depends upon the Netscape supplied Java package,
    // netscape.javascript. This function will NOT work under Internet
    // Explorer.
    //---------------------------------------------------------------------
    public OviPlayer getPlugin() {
        OviPlayer plugin = null;
        ovi = null;
        document = null;
        window = null;
        System.gc();

        try {
            window = JSObject.getWindow(this); //grab window obj
        }
        catch(NullPointerException e) {
            System.out.println("getWindow() failed in getNavWinDoc()...");
        }

        if (window != null)
            try {
                //grab document obj
                document = (JSObject) window.getMember("document");
            }
            catch(NullPointerException e) {
                System.out.println("getMember() failed in getNavWinDoc()...");
            }

        if (document != null) {
            try {
                plugin = (OviPlayer) document.getMember(m_plgname);
            }
            catch(NullPointerException e) {
                System.out.println("unable to grab plugin");
            }
        }
        else {
            System.out.println("Document null in getPlugin()...\n");
        }

        return plugin;

    } //end getPlugin
}
```

3

Oracle Video Java Library

This chapter describes the Oracle Video Java Library, which enables Java applications to play streaming video and audio from the Oracle Video Server. This chapter contains these sections:

- Introduction to the Oracle Video Java Library
- Requirements
- Programming with the Oracle Video Java Library
- Using the PlayerApplet Class
- Quick Start: A Sample Java Application

You can also find the API reference for the Oracle Video Java Library in Appendix B, "Oracle Video Java Library Reference".

Note: Java applets created with the Oracle Video Java Library can be run from the command line, in an applet viewer, or in browsers that permit the execution of unsigned applets. To play video in other browsers, you need to use one of the following:

- The Oracle Video Web Plug-in, as discussed in Chapter 2
- The Oracle Video ActiveX Control, as discussed in Chapter 4

Introduction to the Oracle Video Java Library

The Oracle Video Java Library enables you to create Java applications that play video and audio from an Oracle Video Server. It provides a number of public classes and interfaces that allow you to play back and control video and audio streams, find out information about the current stream, and query the video server for available content titles. This section describes the main groups of public classes and interfaces and how they work together, including:

- Player Classes
- Stream Information Classes
- Content Query Classes

There is also an applet class in the Oracle Video Java Library called **PlayerApplet**. You can use **PlayerApplet** in Java-enabled browsers that allow the use of unsigned applets and as an embedded applet in a Java application. In effect, this means that you can use **PlayerApplet** in Sun's HotJava browser or the JDK **appletviewer** utility, but not in Netscape Navigator or Microsoft Internet Explorer. For more information on **PlayerApplet**, see "Using the PlayerApplet Class" on page 3-31.

This section provides an introduction and description of the Oracle Video Java Library classes and interfaces. For complete reference information on all of the classes and interfaces in the Oracle Video Java Library, see Appendix B, "Oracle Video Java Library Reference".

Player Classes

This section describes the primary classes and interfaces in the Oracle Video Java Library. These are:

- **Player**, which contains all of the functionality needed to directly control media file loading and playback
- **PlayerFactory**, which creates new **Player** objects
- The **PlayerListener** interface, which allows your application to be notified of events affecting the **Player** object
- **PlayerException**, which supports notification of raised exceptions specific to the Oracle Video Java Library

Player

The primary object in the Oracle Video Java Library is the **Player** object. The public methods of **Player** can be loosely grouped into three categories:

- User-interface methods allow you to control the appearance and behavior of the user interface components, including the video screen, playback controls, and status bar
- Media control methods give you VCR-like control over stream playback, including play / pause functionality, stop, rewind, and forward
- Service methods enable you to control the actual **Player** object by monitoring the player's state, adding a listener, or terminating the player

User-interface methods The Java video player provides three separate interface components: the video screen, the playback controls and the status bar. You can retrieve these controls from the **Player** object as a single object, contained in a standard Java **Component** object. This simplifies the layout of each component when the host window is resized. However, you can also get the video screen, the playback controls, or the status bar as separate components for maximum control. Each of these is also returned as a standard Java **Component** object.

You can also implement your own custom controls in place of the default controls or status bar. This is possible since **Player** offers methods that provide VCR-like control over the stream. The only caveat is that you can't override the player's video screen. But you can play audio-only streams without a video screen.

You can find out more about working with the player's interface components in "Retrieving Player Interface Components" on page 3-21.

Media control methods The second group of **Player** methods allows full VCR-like control over the playback of the stream. This includes methods that can pause and resume, play, start and stop, set the position within the stream, and so on. You can use this functionality to provide your own custom user controls or provide hidden control of a stream in the background.

You can find out more about controlling the player in "Loading and Unloading Streams" on page 3-23 and "Controlling Playback" on page 3-25.

Service methods This group of methods provides basic object functionality for the **Player** object. This includes such operations as terminating the **Player** object, retrieving an object's state, and registering an object listener, which provides event-handling methods for the **Player** object.

You can find out more about managing **Player** objects in "Getting and Setting Player Properties" on page 3-10, "Terminating a Player" on page 3-10, and "Handling Player Events" on page 3-17.

PlayerFactory

The **PlayerFactory** class instantiates a **Player** object in a platform-independent manner. Calling the static method **PlayerFactory.getPlayer()** returns a **Player** object.

```
static Player player=null; // create a Player object variable
player = PlayerFactory.getPlayer(); // instantiate the player
```

Note that a **Player** object is not tied to a particular stream. The same **Player** can be reused for many different streams of completely different content types such as MPEG or OSF.

You can find out more about creating a new **Player** object in "Creating a Player" on page 3-9.

PlayerListener

The **PlayerListener** interface specifies a number of methods that a **Player** object may call when specific events occur: errors, end of stream, and changes in the player state. To receive these events, you need to create a class that implements the **PlayerListener** interface. This example shows a simple implementation:

```
public class myApp implements PlayerListener {
   // The actual class implementation goes here...

   public void stateChange(int newState) {
       System.out.println("newState: " + newState);
   }

   public void error(int code, String msg){
       System.out.println("OVC- " + code + ": " + msg);
   }

   public void endOfStream() {
       System.out.println("end of stream reached");
   }
```

Register the class that implements the listener interface by calling the method **Player.addListener()**, passing the object that implements the **PlayerListener** interface. When the appropriate events occur, the **Player** object calls the listener methods on the implementing object.

You can find out more about implementing the **PlayerListener** interface and handling **Player** events in "Handling Player Events" on page 3-17.

PlayerException

Methods in the Oracle Video Java Library throw a **PlayerException** object when an exception is raised. **PlayerException** can indicate a number of conditions:

- The requested operation is not implemented
- The **Player** object is in the incorrect state to execute the requested operation
- An invalid parameter was passed to the method
- An internal error occurred
- A general error occurred
- An untranslated error occurred

PlayerException also contains a numeric code, which provides specific information on the exception that occurred, and a description, which provides a textual explanation of the problem. It also contains a **toString()** method that returns all of the information contained in the exception object.

You can find out more about handling Oracle Video Java Library exceptions in "Handling Player Exceptions" on page 3-30.

Stream Information Classes

This section describes the classes provided by the Oracle Video Java Library to report on the current stream, including:

- Stream position
- Stream information, such as the name, transport protocol, description, frame rate, and so on
- Stream statistics, such as the number of data packets received, number of data packets dropped, average frames per second, and so on

Player methods deal with stream position through the **StmPos** class. **StmPos** gives you flexibility over how you specify the stream position, whether you're querying to find out the current stream position or to set the stream position to a new point. You can set the position as an absolute time from the beginning of the stream in either milliseconds or *hh*:*mm*:*ss*:*cc* format. You can also use the current frame number or set it to the beginning or end of the stream.

Player methods use the **StmInfo** class to pass static stream information back and forth. This class contains information such as the name of the stream, the transport protocol, the media file specifier for the stream, and a text description of the stream. You can get stream information for a player by calling the **Player.getInfo()**, which returns a **StmInfo** object. Then call **StmInfo.toString()**, which returns a formatted string containing all of the above information and more.

You can find the stream's statistics through the **StmStats** class. This class contains network and technical information about the stream, such as the number of data packets received and dropped, consumer (client) and producer (server) playback state, and average frames and bits per second. You can get stream information for a player by calling the **Player.getStats()**, which returns a **StmStats** object. Then call **StmStats.toString()**, which returns a formatted string containing all of the above information and more.

You can find out more about getting stream information in "Getting and Setting Player Properties" on page 3-10.

Content Query Classes

The Oracle Video Java Library provides a set of classes that you can use to query the Oracle Video Server for a list of available content files. The classes are:

- **Content**, which performs the actual content query
- **ContentIter**, which you use to iterate through the returned content entries
- **ContentException**, which is thrown when an exception occurs during the query operation

You can find information on the use of these classes in "Querying Available Content Titles" on page 3-26.

Requirements

There are some basic requirements for creating and running Java applications that use the Oracle Video Java Library, including:

- Installing and Configuring the Oracle Video Java Library
- Run-time Requirements
- Version Requirements

While some Java tutorial information is presented in this chapter, you should be familiar with Java programming concepts such as objects, classes, interfaces, methods, and compiling and running Java applications.

Installing and Configuring the Oracle Video Java Library

To install the Oracle Video Java Library, either:

- Select the Typical option during OVC installation, or
- Select the Custom option and select the Oracle Video Java Library check box.

If you want to develop and compile Java applications or applets that use the Oracle Video Java Library, you need the Java Development Kit (JDK) version 1.1.5.

The Oracle Video Java Library is contained in the Java archive file **ovc.jar**. The installer installs this file in **%ORACLE_HOME%\jbin**.

To run or compile Java classes that use the Oracle Video Java Library, you need to add the **ovc.jar** file to your Java class path. Because this is a JAR archive and not a collection of **.class** files, you explicitly need to indicate the file itself and not just the path to the file. So your CLASSPATH should look something like this:

```
CLASSPATH=C:\JDK1.1.5\Lib;C:\OraWin95\jbin\ovc.jar;.
```

Run-time Requirements

To run Java applications or applets that use the Oracle Video Java Library, you need the following components:

- The Oracle Video Client, with the Oracle Video Java Library option installed
- The Java Runtime Environment (JRE) version 1.1.5; the JRE is available at **http://www.javasoft.com**.

Version Requirements

The Oracle Video Java Library is compliant with Java 1.1. It will not run in any applications that support only Java 1.0, such as Netscape Navigator or Microsoft Internet Explorer, before version 4.0 of either browser. There is an update for Netscape Communicator 4.0 that updates its Java support to Java 1.1.

Programming with the Oracle Video Java Library

This section describes a number of common programming tasks that you can perform when using the Oracle Video Java Library, including:

- Creating a Player
- Terminating a Player
- Getting and Setting Player Properties
- Handling Player Events
- Displaying and Customizing the Player Interface
- Loading and Unloading Streams
- Controlling Playback
- Querying Available Content Titles
- Synchronizing Calls to Player Methods
- Handling Player Exceptions

If you need to find the exact syntax for a specific method referenced in this section, refer to Appendix B, “Oracle Video Java Library Reference”.

Note: Many of the methods in the Oracle Video Java Library can throw **PlayerException** objects. You should always check whether methods you call throw exceptions and handle them appropriately. See “Handling Player Exceptions” on page 3-30 for information on how to handle **PlayerException**.

Importing the Oracle Video Java Library Package

The Oracle Video Java Library is stored in a Java archive (JAR) file named **ovc.jar**. By default, the installer places this file in the **jbin** directory below your **%ORACLE_HOME%** directory. This file needs to be placed in your CLASSPATH. See "Installing and Configuring the Oracle Video Java Library" on page 3-7 for more information on configuring your CLASSPATH.

Once you've set the CLASSPATH, you need to import the Oracle Video Java Library package into your source file. The package name is **oracle.ovc**. You need to import all of the classes and interfaces used in your application.

For example, if your application uses **Player**, **PlayerFactory**, and **PlayerException**, you need to import these:

```
import oracle.ovc.PlayerFactory;
import oracle.ovc.PlayerException;
import oracle.ovc.Player;
```

If you want to import all of the classes and interfaces contained in this package, use the * wildcard in your import statement:

```
import oracle.ovc.*;
```

Creating a Player

A Java application can use the Oracle Video Java Library to create and embed a **Player** in three basic steps:

1. Get a new **Player** object by calling the **PlayerFactory.getPlayer()** method:

   ```
   Player m_player = PlayerFactory.getPlayer();
   ```

   Note that you don't need to create a **PlayerFactory** object before making this call. The **PlayerFactory.getPlayer()** method is static, meaning that you can call it through the **PlayerFactory** class instead of a specific instance of that class. When called, **getPlayer()** checks whether a **PlayerFactory** object has already been initialized. The **PlayerFactory** class stores a central **PlayerFactory** object in a static member. If this member hasn't been initialized, **getPlayer()** initializes it automatically. If it has already been initialized, **getPlayer()** uses the already created **PlayerFactory** object to create the new **Player** object.

2. Call **getPlayerUI()** to get the interface for the player. This method returns a standard Java **Component** object. You can pass this **Component** to the window containing the player just as you would a regular Java component, such as a button or text field.

You can also specify which components of the player interface you want to appear, including the video screen, control panel, and status panel. This example shows how to retrieve the default configuration of the player, which displays all three components:

```
Component m_playerUI = m_player.getPlayerUI();
```

You can find out how to change the configuration of the player's user interface in "Retrieving Player Interface Components" on page 3-21.

3. Add the interface component to the containing window. In this example, it's added to a **Frame** window:

```
Frame m_window = new Frame();
m_window.add(m_playerUI);
```

4. Show the containing window. This example continues from Step 3:

```
m_window.show();
```

Once you've created a player and displayed it in a window, you can load a stream and begin playback, as described in "Loading and Unloading Streams" on page 3-23.

Terminating a Player

You can terminate your player—including unloading the stream, shutting down the player window, and deallocating all resources—by calling the **Player.term()** method:

```
m_player.term();
```

This method takes no parameters and returns **void**.

term() automatically unloads the stream before terminating the player. See "Loading and Unloading Streams" on page 3-23 for more information. Once you've terminated a player, you can no longer use it to load other media files.

Getting and Setting Player Properties

A **Player** object has a number of attributes that you can examine and, in some cases, modify. The basic categories are:

- Stream Position
- Volume Settings
- Stream and Player State

Stream Position

The stream position indicates where the current playback position. You can both query the player to find its current stream position and set the current stream position to wherever you want.

Getting the stream position You can retrieve the current stream position by calling the **Player.getPos()** method. This method takes an **int** parameter that specifies what format you want for the return value. This returns a **StmPos** object containing the current position in its **m_val** member.

StmPos has two public data members that you can examine and modify:

- The **int m_fmt** indicates the format in which **StmPos** returns the current position when queries through the **toString()** method. **m_fmt** can have the values shown in Table 3–1:

Table 3–1 Stream Position Formats

Value	Returns stream position as...
StmPos.POSFMT_TIME	Number of milliseconds from beginning of stream
StmPos.POSFMT_FRAMES	Number of frames from beginning of stream

- The **long m_val** contains the actual position of the stream. This is stored as the number of milliseconds from the beginning, but is converted by **toString()** to the units indicated by **m_fmt**.

You can get the stream position from the returned **StmPos** object by calling the **StmPos.toString()** method. This returns the current position in the current format, as indicated by **m_fmt**. To change the format of the current position returned by **toString()**, change the value of **m_fmt**.

The following example shows how to get the current position from a **Player** object and display it as a time-formatted string.

```
// Get the StmPos object with the current position
StmPos pos = m_player.getPos(StmPos.POSFMT_TIME);
System.out.println(pos.toString());
```

Setting the stream position You can set the current stream to a new position by calling the **Player.setPos()** method, which takes a **StmPos** parameter and returns **void**. There are a number of steps to setting the player to a new position. These steps assume you have already created a **Player** object and loaded a stream. In the example code shown here, the **Player** object is called **m_player**.

1. Create a new **StmPos** object. There are three different ways to do this:

   a. Call the **StmPos(int fmt, long val)** constructor. The **fmt** parameter indicates the format for the new **StmPos** object, while **val** indicates the position you want to set.

   b. The **StmPos(int fmt)** constructor is just like the first constructor, except that you only set the format; the value is assumed to be 0. Set the **StmPos** object's **m_val** to the desired value if you want some other position.

   c. Call the **StmPos.fromString()** method. This method is static, which means you can call it through the **StmPos** class instead of a specific instance of the class. This method works like a constructor, returning a **StmPos** object that you can use as your new instance of the **StmPos** class. **fromString()** takes a single **String** parameter, which it uses this **String** to set the format of the new **StmPos** object. The values recognized for this parameter are shown in Table 3–2. Note that **fromString()** is not case sensitive.

Table 3–2 Meaning of StmPos.fromString() parameters

Parameter	Format
Beginning	Sets **m_fmt** to **StmPos.POSFMT_BEGINNING**; sets the stream position to the beginning
End	Sets **m_fmt** to **StmPos.POSFMT_END**; sets the stream position to the end
Current	Sets **m_fmt** to **StmPos.POSFMT_CURRENT**; sets the stream position to its current value
Default	Sets **m_fmt** to **StmPos.POSFMT_DEFAULT**; sets the stream position to its default position: ▪ For all streams except for unbounded streams (streams with no predetermined end, such as live video), the default position is the beginning of the stream ▪ For unbounded streams, the default position is the end of the stream
string	Sets **m_fmt** to **StmPos.POSFMT_TIME** and **m_val** to the time contained in *string*; *string* should be of the form *hh*:*mm*:*ss*:*cc*

2. Call the **Player.setPos()** method, passing the **StmPos** object you created in Step 1 to the method.

Once this has been successfully completed, the stream position changes to the set value. Playing the stream causes playback to begin at that position.

See "Loading and Unloading Streams" on page 3-23 for information on setting the stream position at load time.

Volume Settings

Getting and setting the volume is fairly straightforward. Volume is represented as an integer from 0 to 100, with 0 meaning no volume and 100 meaning full volume.

Getting the volume You can get the current volume setting of a **Player** object by calling the **getVol()** method. This method takes no parameters and returns an **int** that indicates the current volume. The volume setting goes from 0 (off) to 100 (full volume).

Setting the volume You can change the volume setting of a **Player** object by calling the **setVol()** method. This method takes an **int** that indicates the new volume. The volume setting can be from 0 (off) to 100 (full volume).

You can also set the volume when you load a new stream:

1. Create a new **StmOpts** object.
2. Set the **StmOpts.m_volume** member to the desired volume setting.
3. Pass the **StmOpts** object to the **Player.load()** method.

Stream and Player State

There are three different types of player state you can query:

- The state of the **Player** itself, such as whether it's initialized or not, loading or unloading, playing, and so on
- Static information about the stream, such as its name, description, or length
- Statistics about the stream, such as network performance, average frames or bits per second, or playback state

Player state To find the state of the player, call the **Player.getState()** method. This method takes no parameters and returns an **int**. This return value can have a number of possible values, as shown in Table 3–3.

Table 3–3 Player States

Value	Indicates player is...
Player.ST_UNINIT	Uninitialized
Player.ST_INIT	Initialized but does not have a stream loaded
Player.ST_REALIZED	Ready to play the currently loaded stream
Player.ST_PLAYING	Playing the currently loaded stream
Player.ST_PAUSED	Paused during playback
Player.ST_EOS	At the end of the currently loaded stream
Player.ST_ERROR	In an error state

The sample code below shows how you might use this information.

```
// Create a new player and load a media file
Player m_player = new Player();
m_player.load("/mds/video/oracle.mpi");

while(m_player.getState() == Player.ST_UNINIT) {
   // Just loop while the player's still loading...
   }

// Now that we know it's loaded, we can start playing
m_player.play();
```

Stream Information You can get information about the current stream by calling the **Player.getInfo()** method. This method takes no parameters and returns a **StmInfo** object.

Table 3–4 StmInfo Data Members

Name	Type	Description
m_aspect	int	Aspect ratio * 1000
m_asset	String	Asset cookie in media file
m_bitrate	int	Total bit rate in bits per second
m_bytes	long	File size
m_contStat	int	Content status; this can have the following values: ■ **StmInfo.CSTAT_DISK** indicates the current stream is stored on disk on the video server ■ **StmInfo.CSTAT_FEED** indicates the current stream is a one-step encode file from the video server ■ **StmInfo.CSTAT_LOCALFILE** indicates the current stream is a local file ■ **StmInfo.CSTAT_NETWORK** indicates the current stream is a wide network stream, such as multicast ■ **StmInfo.CSTAT_ROLLING** indicates the current stream is a unbounded stream ■ **StmInfo.CSTAT_TAPE** indicates the current stream is stored on in the Hierarchical Storage Manager (HSM) on the video server ■ **StmInfo.CSTAT_UNKNOWN** indicates unknown status You can get this information as a **String** value by calling the **StmInfo.contStatToString()** method.
m_contType	String	Container type, such as MPEG, OSF, WAV, and so on
m_createTime	Date	Date and time the content file was created (GMT)
m_desc	String	Description
m_fps	int	Frame rate, expressed as frames per second * 1000
m_msecs	int	Total duration of the stream in milliseconds
m_name	String	Printable name of stream
m_proto	String	Transport protocol
m_size	Dimension	Source input size in pixels
m_url	String	Media file specifier for stream

You can also get all of this information in a formatted **String** by calling the **StmInfo.toString()** method.

Stream statistics You can get information about the current stream by calling the **Player.getStats()** method. This method takes no parameters and returns a **StmStats** object. The public data members for **StmStats** are shown in Table 3–5.

Table 3–5 StmStats Data Members

Name	Type	Description
m_bps	int	Average bits per second
m_cnsState	int	Consumer playback state: ■ **StmStats.STM_CONTROL** indicates that the stream is in the middle of a network transaction. ■ **StmStats.STM_ENDED** indicates that the end of stream has been reached. ■ **StmStats.STM_IDLE** indicates that the stream is idle. ■ **StmStats.STM_PAUSED** indicates that the stream is paused. ■ **StmStats.STM_PLAYING** indicates that the stream is playing. ■ **StmStats.STM_STALLED** indicates that the stream has stalled.
m_curFrame	long	Current frame in stream
m_curTime	long	Current time in stream
m_drops	int	Number of data packets dropped
m_fBytes	int	Number of free bytes in cache
m_fps	int	Average frames per second
m_maxTime	long	Last seekable position in stream
m_minTime	long	Earliest seekable position in stream
m_pkts	int	Number of data packets received
m_prdState	int	Producer playback state; possible values are the same as for **m_cnsState**
m_rBytes	int	Number of ready bytes in cache

You can also get all of this information in a formatted **String** by calling the **StmStats.toString()** method.

Handling Player Events

To handle events from a **Player** object, you need to implement the **PlayerListener** interface. This section describes the methods in **PlayerListener** you need to implement and also describes how to implement the listener interface and add it a **Player**'s list of listeners.

> **Note:** You should be familiar with the Java 1.1 event model, as well as the concept of interfaces, before reading this section.

PlayerListener Methods

PlayerListener handles a number of **Player** events, including:

- Change of player state
- End of stream
- Errors

The methods required to handle these events are described below.

Change of player state A change of player state indicates that the player has undergone a change from one state to another. This is often the point at which you may wish to take some sort of action. For example, whenever a user starts playing a stream, you want to change an icon on the interface to indicate the play status. Because playing is one of the state changes handled by the **PlayerListener** interface, you can watch for this event.

The method used to handle changes of state is called **stateChange()**. This method takes a single parameter, an **int**. This parameter indicates the new state of the **Player** object and can have one of the values shown in Table 3–3 on page 14.

End of stream Reaching the end of a stream means that the **Player** has run out of data to play. In short, the movie's over. You may want to take some sort of action whenever a stream finishes. For example, you may want to begin loading the next stream.

The method used to handle the end of a stream is called **endOfStream()**. This method takes no parameters.

Errors You need to do something when an error occurs in your application. Most recoverable errors are handled through the use of exceptions. If an error occurs that isn't handled by an exception, you can assume that it was a fatal error. The **error()** method gives you a chance to do clean up and damage control, including destroying the **Player** object.

The **error()** method takes two parameters:

- An **int** that contains an error code. This code may be used by Oracle technical support in case diagnostic help is required.
- A **String** that contains a text message.

Implementing and Registering a PlayerListener

To implement a **PlayerListener** to handle **Player** events:

1. Create a new class that implements the **PlayerListener** interface.

   You can create any class to implement this interface, although you may find it easiest to implement in whichever class you create the **Player** object. The example below shows the syntax for adding **PlayerListener** to another class:

   ```
   public class myClass implements PlayerListener {
      // Class implementation goes here...
      }
   ```

2. Add the **PlayerListener** methods to the class in which you implemented the **PlayerListener** interface.

   ```
   public class myClass implements PlayerListener {
      // Class implementation goes here...

      // PlayerListener methods
      public void stateChange(int newState) {
          // Take whatever actions you want for this event
          }
      public void error(int code, String msg) {
          // Take whatever actions you want for this event
          }
      public void endOfStream() {
          // Take whatever actions you want for this event
          }
      }
   ```

3. Create an instance of your new class:

   ```
   myClass m_inst = new myClass();
   ```

4. Create a new **Player** object:

   ```
   Player m_player = PlayerFactory.getPlayer();
   ```

5. Call the **addListener()** method on the **Player** object you created in the last step, passing the object you created in Step 3. The **addListener()** method takes a single parameter, an object that implements the **PlayerListener** interface.

   ```
   m_player.addListener(m_inst);
   ```

Once you've registered an implementation of **PlayerListener** with a **Player** through the **addListener()** method, the **Player** object calls the appropriate methods on the listener whenever an event occurs.

The sample code below shows a simple example of how to implement and register a player listener.

```
public class myClass implements PlayerListener {
   Player m_player = null;
   myClass app = null;

   public static void main(String args[]) {
       app = new myClass();

       try {
           m_player = PlayerFactory.getPlayer();
           }
       catch(PlayerException e) {
           System.out.println("Failed to getPlayer()");
           }

       try {
       m_player.addListener(app);
           }
       catch(PlayerException e) {
           System.out.println("Failed to addListener()");
           }

       // Get player interface, add to frame, load stream, and so on...
       }

   // PlayerListener methods
   public void stateChange(int newState) {
       System.out.println("Player state change to: " + newState);
       }
```

```
public void error(int code, String msg) {
    System.err.println("Error code: " + code);
    System.err.println("Error message: " + msg);
    }

public void endOfStream() {
    System.out.println("End of stream reached!");
    }
}
```

Displaying and Customizing the Player Interface

The Oracle Video Java Library player provides three Java **Component**-based interface objects that you can use to provide a video display, status bar, and playback controls. You can also combine any combination of the three interfaces into a single **Component**, simplifying the process of adding the player to your interface.

Because these objects are based on the standard Java **Component**, you can use them in windows, frames, and containers just as you would any standard Java component, such as buttons, list boxes, and icons.

This section covers the following topics:

- Retrieving Player Interface Components
- Customizing Interface Components
- Setting Full-Screen Interface

Retrieving Player Interface Components

There are two ways you can get object references for the interfaces:

- Call individual methods to retrieve each separate component. There are three methods provided by **Player**, one for each component:
  - **getControlComp()**
  - **getStatusComp()**
  - **getVisualComp()**

  Each of these methods returns a **Component** object representing the requested component.
- You can also retrieve one or more of the components combined into a single **Component** object by calling the **Player.getPlayerUI()** method. There are three versions of the this method:
  - The first version takes no parameters. This creates the default player interface, combining a video screen, controls, and status bar.

    This example uses the first version of **getPlayerUI()** to specify the default configuration—all three components displayed:

    ```
    Component m_UI = m_player.getPlayerUI();
    ```
  - The second version takes three **boolean** parameters. This allows you to specify which parts of the default interface—video screen, controls, and status bar, respectively—you want displayed. Specifying **true** for any component means that component should be included in the returned **Component**.

    This example uses the second version of **getPlayerUI()** to create the player with only the video screen showing, with no controls or status bar:

    ```
    Component m_UI = m_player.getPlayerUI(true, false, false);
    ```
  - The third version takes three **boolean** and two **int** parameters. The **boolean** parameters function the same as in the second version, turning various interface components on and off. The **int** parameters let you specify the width and height of the returned **Component** object; that is, the overall size of the player interface.

    This example uses the third version of **getPlayerUI()** to create the player with the video screen and controls showing, with no status bar, and a size of 320 by 240 pixels:

    ```
    Component m_UI = m_player.getPlayerUI(true, false, false, 320, 240);
    ```

Once you've retrieved the **Component** containing the player interface, you can add it to a frame or window just as you would any other **Component**. The code below shows a simple example using the default player interface:

```
public class Spud extends Frame {
   public static void main(String argv[]) {
       Component myUI = null;
       Spud spud = new Spud();
       try {
           Player m_player = PlayerFactory.getPlayer();
           myUI = m_player.getPlayerUI(); // Get the default UI component
           }
       catch(PlayerException e) {
           System.out.println("Exception raised: " + e.toString());
           }

       spud.add(myUI); // Add the UI to the enclosing frame
       spud.show(); // Show the enclosing frame
       }
   }
```

Customizing Interface Components

Because you can request the player's interface components separately, you can mix and match the elements that you use. You can also substitute your own interface components for the default components provided by the player, with the exception of the video screen: you must use the default screen if you want to show video.

To create your own interface component, simply design and lay the component or components out as you would normal Java controls. Through the event handlers for these components, call **Player**'s playback control methods in response to user actions. See "Controlling Playback" on page 3-25 for more information on the playback control methods.

Setting Full-Screen Interface

In addition to displaying the player interface in a container window, you can also change the video window to full-screen mode. To do this, call the method **Player.setFullScreen()**. This method returns **void** and takes a single parameter, a **boolean**. Passing **true** as the parameter puts the player in full-screen mode: the controls, status bar, and other operating system desktop items disappear and the video screen takes up the available real estate.

Note that you don't necessarily have to call this method yourself to provide this functionality to your users. If the pop-up menu is enabled, the user can select full-screen mode through the pop-up menu.

The user can get out of full-screen mode by pressing the Escape key.

Loading and Unloading Streams

A **Player** object is not tied to a particular stream. In fact, you can't create a **Player** with a default stream. In order to use a **Player** to play a stream, you need to explicitly load the stream you want to play by calling the **Player.load()** method.

load() returns **void** and takes two parameters:

- A **String** containing the media file specifier for the stream you want to load. For information on media file specifiers, see Appendix D, "The Media File"
- A **StmOpts** object. **StmOpts** wraps a number of start-up options for the player in a single object. It consists of a number of public data members that you can modify to specify your own start-up options. Table 3–6 shows the available data members in **StmOpts** and the default values given to these members when you call the default **StmOpts** constructor.

Table 3–6 StmOpts Data Members

Name	Type	Default	Description
m_autoStart	boolean	false	When **true**, playback starts at the beginning of the stream or the position specified by m_playFrom once the player loads the media file; otherwise, waits for a play command.
m_img	String	""	Specifies a background image to be displayed when the player is inactive.
m_leftClick	boolean	true	Specifies whether a left mouse click on the video screen can play and pause playback.
m_loop	boolean	false	Specifies whether playback loops when the stream position reaches the end or the position specified by **m_playTo**, if specified.
m_playFrom	StmPos	Beginning	Specifies the position from which to begin playback; see "Stream Position" on page 3-11 for more information on the **StmPos** class.
m_playTo	StmPos	End	Specifies the position at which to end playback; see "Stream Position" on page 3-11 for more information on the **StmPos** class.

Table 3–6 StmOpts Data Members

Name	Type	Default	Description
m_popup	boolean	true	Specifies whether a pop-up menu appears when the right-clicks the video screen.
m_volume	int		Specifies the master volume.

You can call the **load()** method as soon as you've created a valid **Player** object. If you want to load the stream with options other than the default, you first need to create a **StmPos** object, change the desired options, then call **load()** with the modified **StmPos** object.

This example shows how to load a stream using the default options. Notice that the call to **load()** uses **null** in place of a **StmPos** object.

```
public class Spud extends Frame {
   public static void main(String argv[]) {
       Spud app = new Spud("Load a file");
       Player player = null;



       try {
           player = PlayerFactory.getPlayer();
           app.add(player.getPlayerUI());
           app.show();
           player.load("/mds/video/intro.mpi", null);
           }
       catch (PlayerException e) {
           System.out.println("Exception: " + e.m_msg);
           }
       }

   public Spud(String title) {
       super(title);
       }
   }
```

You can also unload media files from the player by calling the **unload()** method. This method returns **void** and takes no parameters. If you've already loaded a stream, call **unload()** before trying to load another. You should also call **unload()** before terminating a stream. See "Terminating a Player" on page 3-10 for more information on terminating **Player** objects. The example below shows how you might use the **unload()** method.

```
// Stop playing and get rid of the player
m_player.unload();
m_pleyer.term();
```

Controlling Playback

Player provides a number of methods that provide VCR-like control over media file playback. These methods, which all return **void**, include:

- **play()** starts stream playback; where playback starts depends on which **play()** method you call, **play()** or **play(StmPos from, StmPos to)**:
  - **play()** starts from the beginning of the stream, regardless of the current stream position, as long as you didn't specify a **playFrom** position with the **load()** method. If so, playback starts from that point.
  - **play(StmPos from, StmPos to)** plays from the position indicated by **from** until it reaches the position indicated by **to**.

  You can only call **play()** when the stream is stopped. To start a paused stream, call **resume()**.
- **stop()** stops playback, resetting the position back to the beginning of the stream. If you specified a starting position, **stop()** resets that to the beginning of the stream also.
- **pause()** stops playback, but, unlike **stop()**, doesn't set the current position back to the beginning.
- **resume()** resumes playback at the current position. You can only call **resume()** when the stream is paused. To start a stopped stream, call **play()**.

Querying Available Content Titles

The **Content** class, along with **ContentIter** and **ContentException**, enables you to request a list of available content titles from an Oracle Video Server. This section contains the following topics:

You can find a sample using the content query classes in your OVC installation. Look in the directory **VC30\DEMO\JAVA** in your **ORACLE_HOME**. The file **CtntList.java** contains a demonstration of using the content query classes to get a list of available titles from the Oracle Video Server.

Content Classes

There are three classes used for retrieving list of available content titles from the Oracle Video Server:

Content The **Content** class contains only one method, **query()**, and no data members. Since **query()** is static, you never need to actually create a **Content** object, but instead can just call **query()** through the class.

Content.query() takes three parameters:

- A **String** containing the address (in the form of a media file specifier) of the server you want to query. If you pass **null** for this parameter, **query()** checks the default server. If you don't have a default server specified, or if that server can't be contacted, **query()** raises a **ContentException**.
- A **String** that contains a file specifier. **query()** uses this as a filter to determine which files are retrieved on the query. You can use UNIX-style wildcards to build this. For example, the string "ora*" would retrieve only titles whose names begin with "ora".
- A **ContentIter** object. This object works with **query()** so that you can retrieve the available titles in discreet chunks.

query() returns an array of **StmInfo** objects. Each of these **StmInfo** objects represents a content title available on the specified server that matches the wildcard you passed to **query()**. The number of objects in the returned array depends on the values you passed to the **ContentIter** constructor and the number of titles available. See the description of **ContentIter** for more information on how **query()** uses the **ContentIter** object.

ContentIter The **ContentIter** class works with the **Content.query()** method to retrieve lists of available content from an Oracle Video Server. The **ContentIter** constructor takes two parameters:

- Position to start querying, stored in the **int** data member **m_pos**. For example, you may specify that you want to start at position 30. This means that the first 30 items that meet the file specification passed to the **query()** method are disregarded. Your returned list begins with the 31st.
- Number of titles to retrieve with each query. This is stored in the **int** data member **m_num**. Specify 10 titles and **query()** returns 10 titles each time you call it; at least, **query()** returns that many as long as there are that many available.

Because the number of available titles is potentially quite high, you do the query iteratively; that is, you perform the same steps repeatedly, each time retrieving another bunch of titles until you've exhausted the available ones. You can also specify which position you want to start on, meaning that if you already retrieved some titles, you don't have to start back at the beginning.

The **query()** method updates **m_pos** after each query operation. For example, suppose you start with **m_pos** set to 0 and **m_num** set to 10. After the first call to **query()** (assuming it's successful), **query()** sets **m_pos** to 10. After the second call, **query()** sets **m_pos** to 20, and so on.

Once you've exhausted the available files, **query()** makes some changes to your **ContentIter** object:

- **m_pos** is set to -1
- **m_num** is set to the number of titles successfully retrieved

Continuing from the example above, you call **query()**, passing your **ContentIter** object with a 10-title capacity for each query. If there are 55 titles available on your server, you can call **query()** six times: the first five calls give you ten titles, with **m_pos** becoming progressively larger (0, 10, 20, and so on) and **m_num** staying the same, 10. But when you make the sixth call, **query()** realizes that there aren't enough titles left to retrieve ten titles for you. It sets **m_pos** to -1 and sets **m_num** to 5, since, after retrieving five chunks of ten titles, there are five titles left in the list.

ContentException The **Content.query()** method can raise a **ContentException** if it encounters an exceptional situation. Possible causes are poorly formed server addresses or a down or inactive server.

ContentException is nearly identical to the **PlayerException** object in usage, methods, and data members. See "Handling Player Exceptions" on page 3-30.

Performing a Query

To find a list of available titles:

1. Declare a **StmInfo** array. Don't declare the size of the array. This array is used to store the list of available titles. Your array declaration should look something like this:

   ```
   StmInfo[] contentList;
   ```

2. Create a **ContentIter** object.

   For the **ContentIter** constructor, pass two parameters, the position at which you want to start in the list of available titles and the number of titles you want to retrieve with each query.

   The example below shows an iterator that starts at the first available title and gets 10 titles at a time:

   ```
   ContentIter iter = new ContentIter(0, 10);
   ```

3. Call the **Content.query()** method. Because this method is static, you can call it through the class without creating an instance of the **Content** class.

   query() takes three parameters:

   - A server address (as a **String**)
   - A file specifier (as a **String**)
   - The **ContentIter** object you created in Step 2

   Assign the return value of the **Content.query()** call to the **StmInfo** array you created in Step 1.

   This example returns all available titles from the default server:

   ```
   contentList = Content.query(null, "*", iter);
   ```

4. Check the value of the **m_pos** member of your **ContentIter** object:
   - If **m_pos** is *not* -1, perform whatever operations you want with the list of titles. After that, you can go to Step 3 and get another chunk of titles.
   - If **m_pos** is -1, the last call to **query()** returned the last available titles. Check the value of **m_num** to find out how many titles were returned.

This example shows how to create a **StmInfo** array and a **ContentIter** object, retrieve all of the available titles from the default video server in chunks of 10, and print out their names and descriptions.

```
int i;
ContentIter iter = ContentIter(0, 10);
StmInfo[] contentList;

try {
   while (iter.m_pos!=-1) {
       contentList = Content.query(null,"*",iter);

       for(i=0; i<iter.m_num;i++) {
           System.out.println("Name: " + contentList[i].m_name);
           System.out.println("Description: " + contentList[i].m_desc);
           }
       }
   }
catch(ContentException e) {
   System.out.println("ContentException raised: " + e.toString());
}
```

Synchronizing Calls to Player Methods

All **Player** methods are synchronous in the sense that, when the routine returns, the calling thread can assume the routine has completed. For example, when **Player.load()** returns, you can assume that the **Player** object has completed loading a stream. The methods are not, however, necessarily synchronized in the context of a multi-threaded Java programming environment. In certain cases, it is possible that you may call **Player** methods concurrently or in an inappropriate order. For instance, in the case of a graphical user interface, by default Java spawns a new thread for every mouse event that occurs. It's possible for different threads to call **Player** methods, resulting in improper or contradictory sequences of calls.

Therefore, when writing applications that are based on the Oracle Video Java Library, you must take into account the multi-threaded environment of Java. The potential for concurrent or unordered calls to **Player** methods can be managed by carefully tracking the **Player** object's states. The relationships between **Player** states and **Player** methods are discussed in detail in "Media Control Methods" on page B-10. See the description of the **Player.getState()** method in "Service methods" on page 3-4 and the **PlayerListener** class reference on page B-20 for detailed information on tracking **Player** object states.

Handling Player Exceptions

Most **Player** methods throw the **PlayerException** exception object when they encounter errors or other exceptional situations. **PlayerException** provides information on the exception, so that you can recover from the error or clean up and exit the application if you can't recover from the error.

PlayerException has three public members. The first is an **int** named **m_type**. **m_type** can have one of the following values:

- **PlayerException.EX_BADPARAM** indicates that an invalid parameter was passed to the method.
- **PlayerException.EX_BADSTATE** indicates that the player is in the incorrect state to execute the requested operation.
- **PlayerException.EX_ERROR** indicates a general error state.
- **PlayerException.EX_INTERNAL** indicates an internal error.
- **PlayerException.EX_NOTIMPL** indicates that the requested operation is not implemented.
- **PlayerException.EX_UNTRANS** indicates an untranslated error. Call **PlayerException.toString()** for more information on the exception.

PlayerException also contains an **int** member **m_code** that contains a specific numeric code and a **String** member **m_msg** that provides a more detailed explanation of the problem. You can get all of this information in one string by calling the **toString()** method.

Using the PlayerApplet Class

The **PlayerApplet** is a simple applet that supports the same syntax and features as the Oracle Video Web Plug-in. You can embed this applet in stand-alone Java applications.

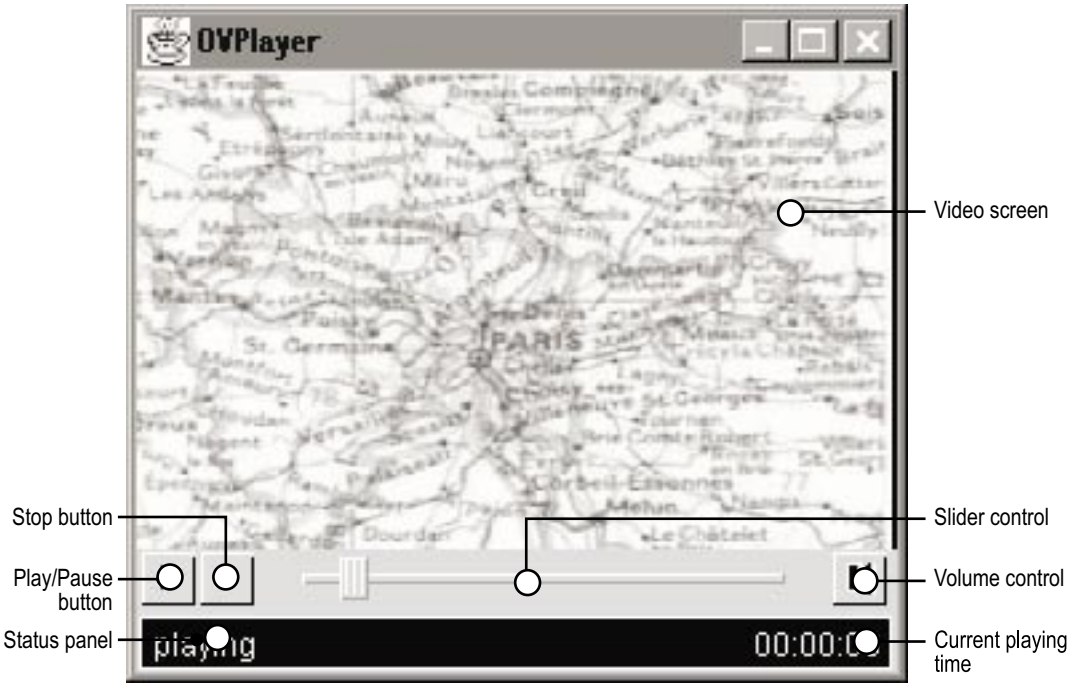
You can also use it to embed video into an HTML document just as you would with the Oracle Video Web Plug-in, except that **PlayerApplet** works only in Java-enabled browsers that allow the use of unsigned applets. In effect, this means that you can use **PlayerApplet** in Sun's HotJava browser or the JDK **appletviewer** utility, but not in Netscape Navigator or Microsoft Internet Explorer. For more information on **PlayerApplet**, see "Using the PlayerApplet Class" on page 3-31.

Quick Start: A Sample Java Application

This section describes how to use the Oracle Video Java Library to create a simple Java application that displays video in a window. A step-by-step tutorial presents each section of code, along with explanations of what's happening. The compiled and source code for this example (and other more complex examples) is installed when the Oracle Video Java Library is installed. You can find the sample code in the file **Simple.java**, located in the directory **vc30\demo\java** in your **ORACLE_HOME**.

Figure 3–1 shows how this simple Java application appears and shows the features of the application's interface.

Figure 3–1 Simple.bat runs Simple.class, the compiled version of Simple.java



Step-by-Step Tutorial of Simple.java

The following is a step-by-step tutorial of the code in the sample application, **Simple.java**. Keep in mind that this is the minimum code required to create a Java client application. Applications using the Oracle Video Java Library can be implemented in many different ways.

1. Import the basic Java packages required: **java.awt** and **java.awt.events**.

   This provides support for windowing and interface event handling.

   ```
   // Sun JDK Imports
   import java.awt.*;
   import java.awt.event.*;
   ```

2. Import the Oracle Video Java Library classes:

   ```
   // Oracle Video Client Imports
   import oracle.ovc.PlayerFactory;
   import oracle.ovc.PlayerException;
   import oracle.ovc.Player;
   ```

3. Create an application class. This is the class that contains the **main()** method, data members, and methods. In the sample application, this class is called **Simple**:

```
public class Simple
{
   // Rest of the class, including member variables and methods, go here...
}
```

4. In your **main()** method, create an application object so that you can call the methods of that object. In the sample application, the application object is called **app**.

```
Simple app = new Simple ();
```

5. Create a window to hold the video player.

   In **Simple.java**, the window is created as a subclass of the Java AWT **Frame** class. The subclass is named **PlayerFrame**. The instantiation of **PlayerFrame** is relatively straightforward:

   - The title parameter is passed on to the superclass
   - A window adapter (**PFAdapter**, a subclass of **WindowAdapter**) that handles only the window closing event is created and added as a window listener
   - The size of the window is set to the size of the height and width parameters passed to the **PlayerFrame** constructor

```
playerframe = new PlayerFrame("OVPlayer", m_Width, m_Height);
```

   Later, after you create the **Player** object (Step 6) and its user interface (Step 7), this frame window hosts the player's interface, which is "added" to the frame window using the **add()** method.

6. Create a **Player** object by calling the **PlayerFactory.getPlayer()** method. This method takes no parameters and returns a **Player** object reference.

   In the sample, this is handled in the **addPlayer()** method of the **Simple** object. You'll note that, in this example, the **PlayerFactory** object is instantiated dynamically: since it's needed only once here, there's no need to keep it around.

Because **Player** and **PlayerFactory** methods can throw exceptions, you must enclose method calls on these objects with a **try-catch** block. There are two specific exceptions you may encounter:

- **UnsatisfiedLinkError** is a standard Java exception thrown when the virtual machine can't satisfy all of the links in a class that it has loaded.
- **PlayerException** is thrown by all members of the **Player** class when an error is encountered. To find out more about exception handling in the Oracle Video Java Library, see "Handling Player Exceptions" on page 3-30.

```
//  Create a player object.
try {
   player = PlayerFactory.getPlayer();
   }
catch (UnsatisfiedLinkError e) {
   System.out.println(e.getMessage());
   }
catch (PlayerException e) {
       System.out.println("Unable to create a Player object.");
   }
```

7. Create and install the player's user interface. This consists of three steps:

   a. Get a user interface object from the **Player** object by calling the **getPlayerUI()** method. This method returns a **Component** object that contains the various elements of the video player, including the video screen, a control panel, and a status line. You can actually specify which of these elements you want in your interface by specifying different parameters to the **getPlayerUI()** method. See "User Interface Methods" on page B-9 for more information.

   b. Add the user interface to the application's frame window by calling the window's **add()** method, passing as a parameter the object returned by **getPlayerUI()**.

   c. Finally, display the frame window by calling the window's **show()** method.

```
try {
playerUI = player.getPlayerUI();
}
catch (PlayerException e) {
// Handle the exception here...
}
```

```
// Add the visual component of the player object
// to the frame before invoking init().
playerframe.add(playerUI);
playerframe.show();
```

The player object is now ready to load and display video.

8. To load a video file, call the **Player.load()** method. This method takes two parameters:

   - A **String** parameter containing a valid media file specifier. Media file specifiers describe the file to be loaded. You can find the format for media file specifiers in "mediafile Syntax" on page D-1.
   - A **StmOpts** object. **StmOpts** objects allow you to specify options for the video stream, such as whether the video should start automatically, where to start and stop within the stream, and more. You can find out more about the **StmOpts** class on page B-26. Specifying **null** for this parameter indicates that the stream should load with the default options: no automatic start, no looping, full volume, start at the beginning and end at the end.

   ```
   player.load(m_MediaFile, null);
   ```

   In **Simple.java**, the **m_Mediafile** points to one of the sample content files installed with the Oracle Video Server. You could change this so that:

   – the Java application queries the video server for a list of available content (see "Querying Available Content Titles" on page 3-26)

   – the user can select from a preset list of videos. For example, you may want to limit the user to a set of training videos. You could present the titles in a drop-down list and load the appropriate video when one is selected.

   – the user can browse locally stored files. For example, create a dialog box that contains a list of files in a particular directory or provides the ability to browse the local hard drive.

Now the media file is loaded and you can perform whatever operations you want: play the video, change the volume, change the position, and so on.

4

Oracle Video ActiveX Control

The Oracle Video ActiveX Control defines methods, properties, and events that you can incorporate into Visual Basic, Oracle Power Objects, Oracle Forms, and other ActiveX-enabled applications. The ActiveX (formerly OCX and OLE) standard lets you create modular applications that can run on the Internet, embed into ActiveX container applications, or work in your own applications.

This chapter is intended for developers who want to add video capabilities to their ActiveX-compliant applications. It shows you how to load the Oracle Video ActiveX Control in Visual Basic, Oracle Power Objects, and HTML documents, and guides you through creating a simple application.

The procedure for using the Oracle Video ActiveX Control with Oracle Forms is somewhat different, so information about using the Oracle Video ActiveX Control with Oracle Forms is presented in Chapter 5, "Working with Oracle Forms".

This chapter contains these sections:

- Introduction to the Oracle Video ActiveX Control
- Requirements
- Using the Oracle Video ActiveX Control in HTML Documents
- Creating Applications with the Oracle Video ActiveX Control

You can also find reference information about the Oracle Video ActiveX Control in Appendix C, "Oracle Video ActiveX Control Reference", including methods, properties, and events associated with the control.

Introduction to the Oracle Video ActiveX Control

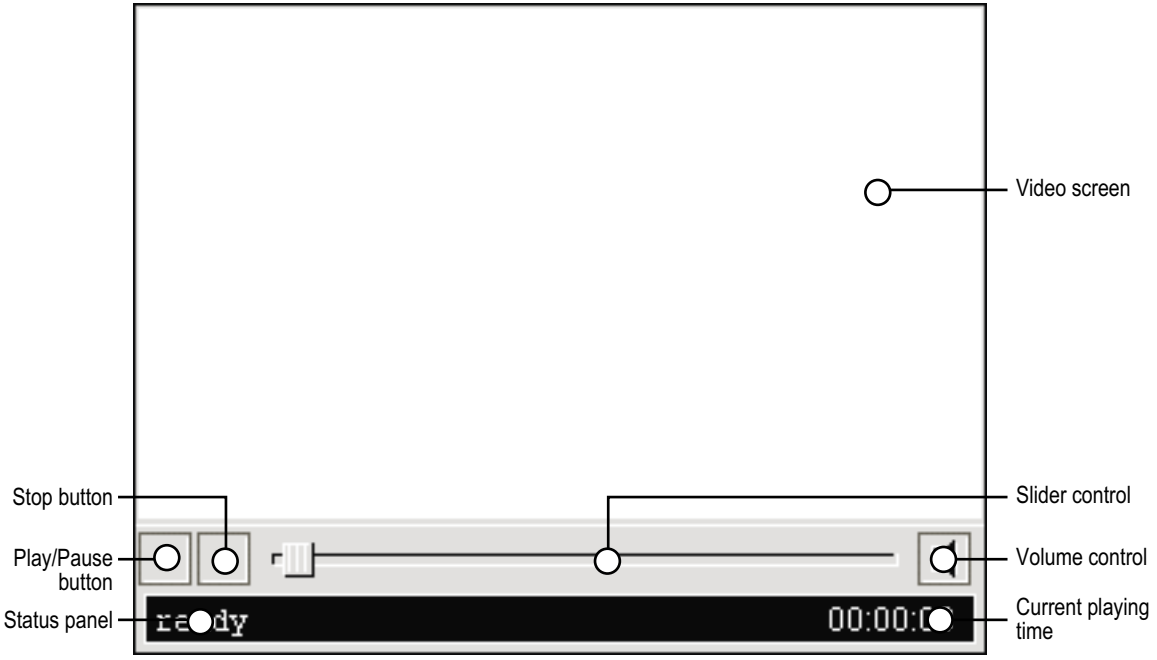
This section provides a basic introduction to the Oracle Video ActiveX Control:

- Becoming Familiar with the Oracle Video ActiveX Control
- Installing the Oracle Video ActiveX Control
- Controlling the Oracle Video ActiveX Control

Becoming Familiar with the Oracle Video ActiveX Control

Figure 4–1 shows the Oracle Video ActiveX Control as it appears on-screen.

Figure 4–1 The Oracle Video ActiveX Control



The controls located at the bottom of the Oracle Video ActiveX Control enable users to play, pause, seek, and stop the video and adjust the volume.

Installing the Oracle Video ActiveX Control

To install the Oracle Video ActiveX Control, make sure you select either the Typical or Compact installation configuration or select the Oracle Video ActiveX Control checkbox if you choose the Custom installation configuration.

The installer installs Oracle files in different areas on different platforms. For this reason, file paths are given relative to **ORACLE_HOME**, which represents the directory where you installed the client.

- On Windows 95 clients, the default **ORACLE_HOME** is **C:\ORAWIN95**
- On Windows NT 4.0 clients, the default **ORACLE_HOME** is **C:\ORANT**

The installer registers the Oracle Video ActiveX Control in the Windows registry during installation.

You can find the complete installation instructions for the Oracle Video Client in the CD insert that came with your installation CD-ROM. This information is also available in electronic form in the file **CDInsert.PDF** in the **\Docs** directory of your installation CD-ROM and, if you selected the Typical installation configuration or selected the Docs option in the Custom configuration, in the directory **VC30\DOCS** in your **ORACLE_HOME**.

Using the Oracle Video ActiveX Control

Because it is an ActiveX control, the Oracle Video ActiveX Control is very versatile. It can be used as:

- An embedded control in an HTML document. Microsoft's Internet Explorer supports ActiveX controls in its Windows-based browsers. To find out how to use the control in an HTML document, see "Using the Oracle Video ActiveX Control in HTML Documents" on page 4-5.
- A custom component in ActiveX-enabled development environments, such as Visual Basic and Oracle Power Objects. You can use this capability to add OVS video to your own stand-alone applications. To find out how to use the control in Visual Basic or Oracle Power Objects, see "Loading the Oracle Video ActiveX Control" on page 4-11. To find out how to use the control with Oracle Forms, see Chapter 5, "Working with Oracle Forms".

The Oracle Video ActiveX Control has been tested and certified to work with various versions of Visual Basic, Oracle Power Objects, Oracle Forms, and Internet Explorer. See the *Oracle Video Client Release Notes* for the most up-to-date compatibility information. The control will function properly in any application that fully supports ActiveX controls, but we cannot guarantee its performance in untested applications or development tools. Oracle strongly recommends that you test the control in any uncertified applications before using it for development.

Controlling the Oracle Video ActiveX Control

You can use a number of techniques to script and control the Oracle Video ActiveX Control. What you use depends on where you are using the control:

- In HTML Documents
- In Application Development Tools

In HTML Documents

You can control the Oracle Video ActiveX Control in an HTML document using JScript and VBScript.

- You can get the highest level of control in Microsoft's Internet Explorer using VBScript. VBScript allows you to call the control's methods, handle any events, and modify and check the properties of the control using a subset of Visual Basic specific to Internet Explorer.
- Internet Explorer also provides JScript, which is similar to Netscape's JavaScript. Both of these are related to (but not identical to) the Java language.

You can find information on controlling the Oracle Video ActiveX Control in an HTML document using VBScript in "Using the Oracle Video ActiveX Control in HTML Documents" on page 4-5.

In Application Development Tools

Embedding the Oracle Video ActiveX Control in your own application is the same as with any ActiveX control. For example, in Visual Basic, you add the control as a component, where it appears as an icon. Either double-click the icon or select the control's icon on the toolbar and click and drag on the form. Once you've added it to your application, you can select it to make its property sheet appear. There, you can modify the control properties, add event handlers, and so on.

You can also reference the control's properties and methods from outside the control. Simply reference the control and method name, passing the appropriate parameters.

For more information on using the Oracle Video ActiveX Control in Visual Basic or Oracle Power Objects, see "Loading the Oracle Video ActiveX Control" on page 4-11 and "Programming with the Oracle Video ActiveX Control" on page 4-12. For information on using the Oracle Video ActiveX Control with Oracle Forms, see Chapter 5, "Working with Oracle Forms".

Requirements

There are some basic requirements for creating client applications using the Oracle Video ActiveX Control:

- You need to be working in a development environment that supports ActiveX controls. This means Windows 95 or Windows NT 4.0.
- If you are creating a stand-alone application, you need an appropriate development tool such as Visual Basic 4.0 or 5.0.
- If you are creating an HTML document, you need an ActiveX-enabled browser, usually Microsoft Internet Explorer 3.0 or later, for testing and development purposes.

Using the Oracle Video ActiveX Control in HTML Documents

This section discusses how to insert the Oracle Video ActiveX Control into an HTML document, including:

- Embedding the Oracle Video ActiveX Control
- Setting Properties for an Embedded Oracle Video ActiveX Control
- Security Requirements in Internet Explorer
- Sample ActiveX Control in HTML

Embedding the Oracle Video ActiveX Control

To embed the Oracle Video ActiveX Control into an HTML document, you must use the **<object>** tag. This tag is recognized by Microsoft Internet Explorer and indicates that you want to use an ActiveX control to display media or somehow interact with the user.

To load the control, the main attributes to the **<object>** tag that you need to specify are:

- **ID**, which you need to specify only if you want to script the control, since you refer to the control in your code by this name
- **ClassID**, which specifies the control you want to load
- **Width** and **Height**

A typical object statement looks something like this:

```
<object id="ovcax1" width=352 height=240
   classid="CLSID:547A04EF-4F00-11D1-9559-0020AFF4F597">
```

If you specify a border, the width and height values include both the control itself and the border. In this case, no border was specified, so this represents the actual width and height of the plug-in. If you do specify a border, you need to add double the border width to the desired size of the control to get the actual width you need to set.

Also, if you include the optional control panel and status line, you need to take these into account when setting the height of the control.

- If you include the control panel, add 24 pixels to the height.
- If you include the status line, add 21 pixels to the height.

You control the appearance of the control panel and status line by setting control properties, as discussed in the next section. Note that the extra height for these optional panels is in addition to any that you need to add for the border.

Setting Properties for an Embedded Oracle Video ActiveX Control

The **<object>** statement actually tells the browser which control you want to embed in the HTML document. The Oracle Video ActiveX Control doesn't actually take any of its start-up information from it, though. Instead it gets it from the **<param>** tags placed within the **<object>** block, that is, between the **<object>** and **</object>** tags. You need to specify information such as:

- Media file to load
- Whether to start playback automatically
- How to display the control's VCR controls

These elements are controlled through the use of properties on the Oracle Video ActiveX Control. You can set any Oracle Video ActiveX Control property (except for those that are read-only) when embedding the Oracle Video ActiveX Control into an HTML document using **<param>** tags within the **<object>** block.

<param> tags have three elements:

- The **<param>** tag itself
- The property name, such as **AutoStart** or **Mediafile**:

  `name="propName"`

- The property value; for string or numeric properties, this should just be the string or value, while for boolean properties, you should specify 1 for true or on and 0 for false or off

  `value="value"`

For example, to set the media file specifier, the **<param>** tag would look something like this:

```
<param name="Mediafile" value="vstcp://server:5000/mds/video/video.mpi">
```

Any properties that you don't set through the use of **<param>** tags use the default value for that property.

Security Requirements in Internet Explorer

Microsoft Internet Explorer offers a great deal of control over whether and how the browser deals with different types of active content, including:

- Allowing the downloading of active content
- Enabling ActiveX controls and plug-ins
- Running ActiveX scripts

In addition, Internet Explorer offers three different security levels:

- High means that any potentially unsafe content is avoided. Unsigned controls are not allowed.
- Medium means that you are notified before potentially unsafe content is downloaded. You must explicitly approve the use of unsigned controls.
- Low means that all content is allowed. You do not need to approve the use of unsigned controls.

Internet Explorer also lets you specify different security settings for different zones, such as Local Intranet Zone or Internet Zone. How you set this depends on where your Web server is on the network.

Because the Oracle Video ActiveX Control is unsigned, you (and the end users of your client application) must set the active content safety level to Medium or Low:

- In Internet Explorer 3.0:
  1. Select the Options command on the View menu.
  2. Switch to the Security tab.
  3. Make sure the **Allow downloading of active content** and **Enable ActiveX controls and plug-ins** check boxes are checked.
  4. Click the **Safety Level** button.

5. In the **Safety Level** dialog box, select either **Medium** or **Low**.

   Medium prompts you before downloading unsigned ActiveX controls. **Low** downloads the control without prompting. Which you select depends on your environment. If your users work exclusively in enterprise, intranet, or other closed environment situations, you may just want to set the security level to **Low** on the assumption that users are safe from unsafe controls within the firewall. If there's a possibility that users may also go onto the Internet or some other uncontrolled area, you may want to set the security level to **Medium** so that users know when a site attempts to pass active content on to them.

6. Click OK, then OK again.

- In Internet Explorer 4.0:

  1. Select the **Internet Options** command on the **View** menu.
  2. Switch to the **Security** tab.
  3. Select the zone containing the Web server or server with HTML documents with the Oracle Video ActiveX Control embedded.

     This will probably be either Local Intranet Zone or the Trusted Sites Zone. For both of these zones, you can also explicitly specify a particular site.

  4. Select either **Medium** or **Low**.

     Medium prompts you before downloading unsigned ActiveX controls. **Low** downloads the control without prompting. If you trust the entire site, you should set this to **Low** to avoid having to approve the download each time you access a page containing the Oracle Video ActiveX Control.

Sample ActiveX Control in HTML

This sample shows a simple example of embedding the Oracle Video ActiveX Control into your HTML documents and controlling it through VBScript routines.

```
<html>
<head>
<title>New Page</title>
</head>

<body>
   <object ID="ovcax1" Width=352 Height=240
      ClassID="CLSID:547A04EF-4F00-11D1-9559-0020AFF4F597">
      <param name="Mediafile" value="vstcp://server:5000/mds/video/oracle1.mpi">
      <param name="ShowControls" value="1">
      <param name="AutoStart" value="0">
      <param name="BorderStyle" value="1">
      <param name="Loop" value="0">
      <param name="EnablePopup" value="1">
      <param name="EnableLeftClick" value="1">
      <param name="PlayFrom" value="beginning">
      <param name="PlayTo" value="end">
      <param name="ShowPositionAndStatus" value="1">
      <param name="TimerFrequency" value="500">
   </object>

   <form name="Button1">
      <input type=Button value="Play" onClick="ovcax1.Play()">
   </form>
</body>
</html>
```

Creating Applications with the Oracle Video ActiveX Control

This section discusses how to create stand-alone applications in Oracle Power Objects and Visual Basic, using the Oracle Video ActiveX Control as a component to provide streaming media capabilities. There are three main topics in this section:

- Loading the Oracle Video ActiveX Control
- Programming with the Oracle Video ActiveX Control
- A Simple Application

Loading the Oracle Video ActiveX Control

Movie Tool icon

Before you can use the Oracle Video ActiveX Control, it must be available to your application. To see if the control is available, start Oracle Power Objects or Microsoft Visual Basic and verify that the Movie Tool icon (shown here) is visible.

- In Oracle Power Objects, look in the Object Palette
- In Microsoft Visual Basic, look in the Toolbox (if you can't find the Toolbox, choose the Toolbox command on the View menu)

If you can't see the movie icon, follow these steps to add the icon.

Oracle Power Objects 2.1

When you start Oracle Power Objects for the first time after installing the Oracle Video Client, the Movie Tool icon is not visible in the Object Palette.

To load the 32-bit Oracle Video ActiveX Control (**ovcax.dll**):

1. Choose **Custom Controls** from the **Edit** menu.
2. Select "Oracle Video ActiveX Control" from the list of installed controls.

   If it's not listed, click the **Register** button, locate and select the **ovcax.dll** file, and click **OK**. By default, this file is located in the **bin** directory in your **ORACLE_HOME**.

Microsoft Visual Basic

When you begin a Visual Basic project, the movie icon is not visible in the Toolbox. You must load the custom control for each new project, as follows:

1. Choose **Components** from the **Project** menu.
2. Find the Oracle Video ActiveX Control in the list box on the **Controls** tab.
3. Check the box next to the Oracle Video ActiveX Control entry.
4. Click the **OK** button.

Programming with the Oracle Video ActiveX Control

Movie Tool icon

Once you have loaded the control, you can use the Oracle Video ActiveX Control as you would any other item in the Oracle Power Objects Object Palette or Visual Basic Toolbox:

- Select the Movie Tool icon and add the Oracle Video ActiveX Control to your application.

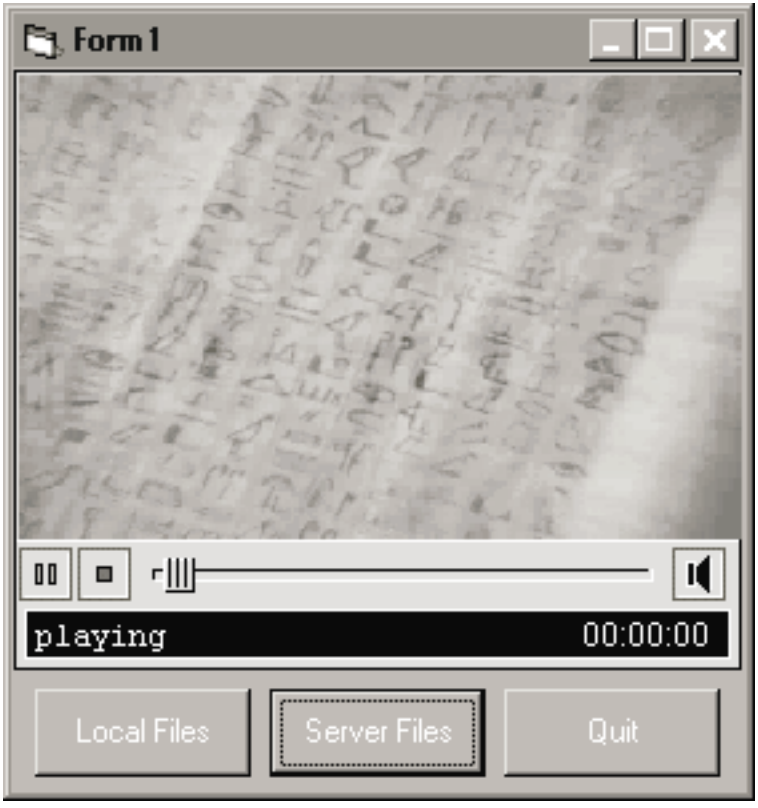
> **Note:** You can have only one active Oracle Video ActiveX Control in any one application.

- Set the properties of the control dynamically at design time using the Oracle Video ActiveX Control's property sheet.
- Add code to handle any of the events defined for the control just as you would with any other ActiveX control.
- Create buttons to execute any of the control's methods. For example, create your own control set, such as play and pause, instead of the built-in controls.

A Simple Application

This tutorial is designed to guide you through creating a simple Oracle Video Client application in Visual Basic and Oracle Power Objects. The finished application lets users view, select, and play media files from a local hard disk or from a video server on the network. Use the finished application, shown in Figure 4–2, as a starting point for developing your own applications.

Figure 4–2 The Finished Application



1. Add the Oracle Video ActiveX Control object to your application.

   For Visual Basic:

   a. Double click the Movie Tool icon in the Toolbox to add it to your project. The default name of an Oracle Video ActiveX Control object is **ovcax1**.

b. Alternatively you can click the Movie Tool icon, then drag in your form. This allows you to easily customize the size of the control.

For Oracle Power Objects:

a. Click on the Oracle Video Control icon in the Object Palette.

b. Drag it to the application window. The default name of an Oracle Video Control object is **ovcax1**.

Note: When you first open the Oracle Video ActiveX Control, you might not be able to see the controls if the initial size of the object is too small. If necessary, click on the control and drag the corners to resize it in the application window.

---

2. If you can't see the VCR controls on the Oracle Video ActiveX Control, set the **ShowControls** property to true (1).

3. Add a button labeled **Local Videos**. This example uses the **ImportFileAs()** method to bring up a list of videos available from the hard disk. For more information on this method, see "Methods" on page C-1.

   For Visual Basic:

   a. Double-click on the **CommandButton** icon in the Toolbox. A command button appears on the form. If the command button becomes deselected, click this button to reselect it.

   b. Change the Caption property to **Local Videos**.

   c. Double-click on the **Local Videos** button. This opens the script window and places the cursor in the **Command1_Click()** method.

   d. Type the following in the script window:

      ```
      ovcax1.ImportFileAs
      ```

      The **ImportFileAs()** method opens a dialog box that lets the user browse for a media file stored locally.

   e. Close the script window.

For Oracle Power Objects:

a. Double-click on the **PushButton** icon in the Object Palette, and drag the button to the form. A push button appears on the form. If the push button becomes deselected, click this button to reselect it.

b. Change the Label property to **Local Videos**.

c. Edit the **Click()** method on the Property Sheet to read as follows:

```
ovcax1.ImportFileAs
```

The **ImportFileAs()** method opens a dialog box that lets the user browse for a media file stored locally.

4. Add a button labeled **Server Videos**. When clicked, this button calls the **ImportStreamAs()** method, which opens a dialog box that lets the user select from a list of videos available on the video server. For more information on this, see "Methods" on page C-1.

For Visual Basic:

a. Add a button to the form, as described in Step 3.

b. Change the **Caption** property to **Server Videos**.

c. Double-click on the **Server Videos** button. This opens the script window and places the cursor in the **Command2_Click()** method.

d. Type the following in the script window:

```
ovcax1.ImportStreamAs("")
```

e. Close the script window.

For Oracle Power Objects:

a. Add a button to the form, as described in Step 3.

b. Change the **Label** property to **Server Videos**.

c. Edit the **Click()** method on the Property Sheet to read as follows:

```
ovcax1.ImportStreamAs("")
```

5. Finally, add a **Quit** button to the form.

For Visual Basic:

a. Add a command button to the form.

b. Change the **Caption** property to **Quit**.

c. Double-click on the **Quit** button. This opens the script window and places the cursor in the **Command3_Click()** method.

d. Type the following in the script window:

```
End
```

e. Close the script window.

For Oracle Power Objects:

a. Add a push button to the form.

b. Change the **Label** property to **Quit**.

c. Edit the **Click()** method:

```
Form1.CloseWindow
```

d. Close the script window.

6. Run the application.

   - In Visual Basic, from the Run menu, choose **Start.**
   - In Oracle Power Objects, from the Run menu, choose **Run Application** or **Run Form**.

You can now select and play videos from a local hard disk or a networked video server, and can use the play, pause, stop, controls when viewing the video.

You can also use the slider control to seek to any location in the video. If the slider control is used while viewing a video content file, the video frame is not updated until play is resumed.

This is all you need to do to use video in your application. You can also use the features of the Oracle Video ActiveX Control to create much richer applications. For example, you can use the **Play()**, **Pause()**, **Resume()**, **SetPos()**, and **Stop()** methods to create your own start, pause, seek, and stop buttons.

5

Working with Oracle Forms

This chapter is intended for developers who want to use the Oracle Video ActiveX Control to add video capabilities to their Oracle Forms applications. Oracle Forms is a component of Oracle Developer / 2000. This chapter guides you through creating a simple Oracle Video Client application, then provides reference material to help you build more elaborate applications.

Although there are no functional differences in the Oracle Video ActiveX Control between Oracle Forms and Oracle Power Objects (or Microsoft Visual Basic), there are some substantial differences in how to load and use the Oracle Video ActiveX Control from Oracle Forms.

For information on the methods, properties, and events provided by the Oracle Video ActiveX Control, see Chapter 4, "Oracle Video ActiveX Control". You can find reference information for the Oracle Video ActiveX Control in Appendix C, "Oracle Video ActiveX Control Reference".

This chapter contains these sections:

A Simple Application

This example uses the Oracle Video ActiveX Control to create a simple application which enables users to view, select, and play videos from a local hard disk or the Oracle Video Server. To see this same application created in Oracle Power Objects and Visual Basic, see Chapter 4, "Oracle Video ActiveX Control".

Note: When you initially create the Oracle Video ActiveX Control object, you will not be able to modify the properties of the object until you deselect it (by clicking anywhere else on the screen) and then reselect it.

---

1. Open the Forms Designer and go to the Layout Editor in the **Tools** menu.
2. Create the ActiveX object:
   a. Click on the **OLE2** object icon in the Tool Palette.
   b. Position the cursor on the grid, then click and drag to create a box for the object.
   c. Right-click on the box and choose **Insert Object**. The **Insert Object** dialog box appears.
   d. Select the **Create Control** radio button.
   e. Scroll down the **Object Type** list and choose Oracle Video ActiveX Control. Click **OK**.
   f. Expand the size of the box as necessary to see all the buttons.
   g. Right-click on the ActiveX object and select **Properties** from the pop-up menu.
   h. Under the **Functional** section, change the **OLE In-place Activation** setting to **True**.
   i. Before closing the properties box, change the name to "MY_VIDEO". Click on a setting other than Name to implement the name you entered.
   j. Close the **Properties** dialog box.

3. Create a button to choose videos from the local hard disk.
   This button uses the **ImportFileAs()** method to display videos available on the local hard disk or CD-ROM.
   a. Click on the **Push Button** icon in the Tool Palette.
   b. Click on the grid to place the button.
   c. Double-click on the button to display its properties.
   d. Under the **Functional** section, change the **Label** property to **Local Videos**.
   e. Close the **Properties** dialog box.
4. Create a trigger for the button:
   a. With the button selected in the Layout Editor, choose **PL/SQL Editor** from the **Tools** menu.
   b. Choose **New**.
   c. Select **WHEN-BUTTON-PRESSED** as the type of trigger and click **OK**.
   d. Enter the following text:

```
declare
video_obj ole2.obj_type;
begin
video_obj := forms_ole.get_interface_pointer ('MY_VIDEO');
ole2.invoke(video_obj, 'ImportFileAs');
end;
```

   e. Choose **Compile**.
   f. Choose **Close**.
5. Create the button to choose a video from the server:
   a. Click on the **Push Button** icon in the Tool Palette.
   b. Click on the grid to place the button.
   c. Double-click on the button to display its properties.
   d. Under the **Functional** section, change the **Label** property to **Server Videos**.
   e. Close the **Properties** dialog box.

6. Create a trigger for the button:

   a. With the button selected in the Layout Editor, choose **PL/SQL Editor** from the **Tools** menu.

   b. Choose **New**.

   c. Select **WHEN-BUTTON-PRESSED** as the type of trigger and click **OK**.

   d. Enter the following text:

```
declare
video_obj ole2.obj_type;
myArgList ole2.list_type;




begin
video_obj := forms_ole.get_interface_pointer ('MY_VIDEO');
myArgList := ole2.create_arglist;

ole2.add_arg(myArgList, 'svraddr:port');
ole2.invoke(video_obj, 'ImportStreamAs', myArgList);
end;
```

   Substitute a valid server name and port for *svraddr* and *port* in the **ole2.add_arg** call.

   e. Choose **Compile**.

   f. Choose **Close**.

7. Create another trigger. This trigger prevents Forms from trying to log into a database when you run the form:
   a. Choose **PL/SQL Editor** from the **Tools** menu.
   b. Change **Object** to **Form Level.**
   c. Choose **New**.
   d. Choose **ON-LOGON** as the type of trigger and click **OK**.
   e. Enter the following:

      ```
      null;
      ```

   f. Choose **Compile**.
   g. From the **File** menu, Choose **Save**.
   h. Choose **Close**.
8. View your final application by choosing **Run** from the **File** menu. A dialog box appears asking if you want to log on before performing compilation. Choose **No**.
9. Click either the **Local Videos** or **Server Videos** button to select a video. After you select a video, the **Play** button will turn green. Click on this button to play the video.

Accessing Methods and Properties

The commands in the PL/SQL scripts that work with the Oracle Video ActiveX Control take different forms depending on whether they are:

- Executing a method
- Setting the value of a property
- Getting the value of a property

Executing a Method

For buttons that execute a method, the PL/SQL script for the WHEN-BUTTON-PRESSED trigger should follow this form:

```
declare
   video_obj ole2.obj_type;

begin
   video_obj := forms_ole.get_interface_pointer('OLE-object-name');
   ole2.invoke(video_obj, 'method');
end;
```

Where:

- ***video_obj*** is a variable name.
- ***OLE-object-name*** is the name of the video object.
- ***method*** is the name of the method to invoke (such as **play**, **stop**, and so on).

Executing a Method with Parameters

For buttons that execute a method that takes parameters, such as **GetStats()**, the PL/SQL script for the WHEN-BUTTON-PRESSED trigger should follow this form:

```
declare
   video_obj ole2.obj_type;
   myArgList ole2.list_type;

begin
   video_obj := forms_ole.get_interface_pointer('OLE-object-name');
   myArgList := ole2.create_arglist;

   ole2.add_arg(myArgList, "arg");

   ole2.invoke(video_obj, 'method', myArgList);
end;
```

Where:

- ***video_obj*** is a variable name.
- ***myArgList*** is a list of items.
- ***arg*** is a parameter for the method call.
- ***OLE-object-name*** is the name of the video object.
- ***method*** is the name of the method to invoke (such as **play**, **stop**, and so on).

Repeat the call to **ole2.add_arg()** for each parameter of the ***method***.

Setting the Value of a Property

For buttons that set a property, the PL/SQL script for the WHEN-BUTTON-PRESSED trigger should follow this form:

```
declare
   video_obj ole2.obj_type;

begin
   video_obj := forms_ole.get_interface_pointer('OLE-object-name');
   ole2.set_property(video_obj , 'property','value');
end;
```

Where:

- ***video_ob***j is a variable name.
- ***OLE-object-name*** is the name of the video object.
- ***property*** is the name of the property to be set (such as **ShowControls**).
- ***value*** is the value to be assigned to the property (such as **True** or **False**).

Getting the Value of a Property

For buttons that get the numeric value of a property, the PL/SQL script for the WHEN-BUTTON-PRESSED trigger should follow this form:

```
declare
   video_obj ole2.obj_type;
   variable number;

begin
   video_obj := forms_ole.get_interface_pointer('OLE-object-name');
   variable := ole2.get_num_property(video_obj , 'num-property');
end;
```

Where:

- ***video_ob***j is a variable name.
- ***variable*** is the variable that holds the returned property value.
- ***OLE-object-name*** is the name of the video object.
- ***num-property*** is the name of the property to be queried (such as **IsLoaded**).

For buttons that get the string of a property, here's the PL/SQL script for the WHEN-BUTTON-PRESSED event:

```
declare
   video_obj ole2.obj_type;
   variable char;

begin
   video_obj := forms_ole.get_interface_pointer('OLE-object-name');
   variable := ole2.get_char_property(video_obj, 'char-property');
end;
```

Where:

- ***video_obj*** is a variable name.
- ***OLE-object-name*** is the name of the video object.
- ***char-property*** is the name of a property that contains a character value, such as **Mediafile**.
- ***num-property*** is the name of a property that returns a numeric value, such as **PlayFrom**.

Modifying Properties

To set the Oracle Video ActiveX Control properties, click the right mouse button in the Oracle Video ActiveX Control object, then drag down to the Oracle Video ActiveX Control Object | Properties.

Do not try to set properties by double-clicking on the Oracle Video ActiveX Control object. You can't set the properties from the dialog box that appears.

Note: If you set any visual properties, such as **ShowControls**, to 0, the controls disappear in the designer, but the change will not be reflected in the run-time version. To see the changes in the run-time version, you must close the Layout Editor to save the changes and run the form.

Troubleshooting

When using Oracle Forms, you might receive this run-time error:

`FRM-41344: OLE object not defined for` ***`object`*** `in current record.`

which can occur for either of these reasons:

- The OLE container has lost the definition of the Oracle Video ActiveX Control.

  To fix this problem, go into the Forms Designer. Re-insert the Oracle Video ActiveX Control by clicking the right mouse button inside the OLE container and choosing **Insert Object.**

- The Oracle Video ActiveX Control has not been initialized.

  To fix this problem, modify the form so that it can navigate to the block that contains the Oracle Video ActiveX Control. You can either make this block the first block on the form or add a GO_BLOCK command in the WHEN-NEW-FORM-INSTANCE script to navigate to that block. If necessary, you can add a GO_BLOCK command followed by SYNCHRONIZE before any commands that access the Oracle Video ActiveX Control. (You can tell when the Oracle Video ActiveX Control has been initialized because the video control buttons will be visible.)

A

Oracle Video Web Plug-in Reference

This appendix contains the reference information for the Oracle Video Web Plug-in. This plug-in allows you to embed the Oracle Video Client in applications that support the Netscape plug-in standard. For details on how to embed the plug-in in an HTML page and other tasks, see Chapter 2, "Oracle Video Web Plug-in".

This appendix contains the following information:

- <Embed> Attributes
- JavaScript Methods
- Java Classes

<Embed> Attributes

This section lists all of the available **<embed>** statement attributes recognized by the Oracle Video Web Plug-in. The following attributes are available:

autoStart	loop	sliderRate
background	mediafile	src
controls	name	toolTips
controlMask	playFrom	type
height	playTo	volume
hidden	popupMenu	width
leftClick		

autoStart

Syntax: autoStart="true | false"

Specifies whether the plug-in begins playing as soon as the player has completed loading the stream.

If true, the stream starts as soon as the page is opened. The default value is false.

background

Syntax: background="*#rrggbb*"

Specifies a background color to be displayed behind the plug-in area. This uses the standard HTML format for specifying RGB color values. This background can't be seen when the plug-in video screen is displayed.

controls

Syntax: controls="true | false"

Specifies whether the player displays the VCR-style controller. The controller consists of:

- a push button that toggles between the Play and Pause icon
- a stop button
- a seek slider indicating the current stream position; you can change the stream position by dragging the slider to another position
- a volume slider indicating the current volume; you can change the volume by dragging the slider to another position

If **controls** is true, the plug-in displays the controls specified by the **controlMask** parameter. The default value is false.

Figure A–1 shows the Oracle Video Web Plug-in with both the controller and the status line visible.

Figure A–1 controls=true and controlMask="controller+statusline"

controlMask

Syntax: `controlMask="controller" [+] "statusline"`

Selects which controls appear in the plug-in. The default is the controller only. This requires you to set **controls** to true, otherwise no controls show up:

- **controls=true** displays the controller only
- **controls=true** and **controlMask= "controller+statusline"** displays both the controller and the status line
- **controls=true** and **controlMask= "statusline"** displays the status line only

height

Syntax: `height=`*`nnnn`*

Required. Specifies the height of the plug-in in pixels. If the **controls** and **border** attributes are not set, then this indicates the actual height of the plug-in. If the **border** attribute is set, then:

1. Multiply the border thickness times two.
2. Add this to the height that you want the video screen.

3. Use this number for **height**.

If **controls** is turned on:

1. If the controller is displayed, add 24 pixels.
2. If the status line is displayed, add 21 pixels.

For example, if you want a video screen that is 200 pixels high with a 10-pixel thick border and the controller displayed, multiply the border times two to get 20, add this to the desired width of the screen, 200, then add 24 pixels for the controller. This gives you a height of 244 pixels.

hidden

Syntax: `hidden="true | false"`

Specifies whether the plug-in is displayed on the page or not. If true, the plug-in is not displayed, regardless of the value of **height** and **width** attributes. The default value is false.

leftClick

Syntax: `leftClick="true | false"`

Specifies whether a left click on the video screen toggles Play and Pause. If **leftClick** is true, left clicking in the video screen has the same effect as clicking on the Play / Pause button. The default value is true.

loop

Syntax: `loop="true | false"`

Specifies whether the player loops the current stream by starting back at the beginning whenever it reaches the end of stream. The default value is false.

If you set the **playFrom** and **playTo** attributes or started playback by calling the **play()** method with **to** and **from** parameters, the stream returns to the **playFrom** position when it reaches the **playTo** position.

mediafile

Syntax: mediafile="*mediafile_url*"

Required. Specifies the protocol, server, and media file to play. You must use the this attribute in conjunction with either the **src** or **type** attribute.

For information on how these attributes work together, see "Specifying the Media File and MIME Type" on page 2-8. For information on media file specifiers, see Appendix D, "The Media File".

name

Syntax: name="*plugin_name*"

Required (if you want to call methods on the plug-in from a JavaScript function or a Java applet). Declares the public name for the plug-in instance.

In the following example, the plug-in instance is named **video1**:

```
<embed type="application/oracle-video" name="video1" width=352
height=240 mediafile="vstcp://sun:5000/mds/video/oracle1.mpi">
```

A JavaScript call to the plug-in would look like this:

```
document.video1.play();
```

playFrom

Syntax: playFrom=["beginning" | "end" | "hh:mm:ss:cc" | millisecs]

Specifies a start time for playback. If **loop** is true, then playback loops back to the point specified by this attribute; the default is to play from the beginning.

To specify a time, use one of these formats:

- The predefined string values "beginning" or "end"
- A string in the format **hh:mm:ss:cc**

  For example, **00:02:40:10** starts the stream two minutes, forty seconds, and ten hundredths of a second from the beginning of the stream (160.1 seconds).
- The number of milliseconds from the beginning of the stream

  For example, **160100** is the millisecond equivalent of 00:02:40:10 above.

playTo

Syntax: `playTo=["beginning" | "end" | "hh:mm:ss:cc" | millisecs]`

Specifies a stop time for playback. If the **loop** attribute is false or not set, the plug-in stops when it reaches this point. The default is to play until the end. If **loop** is true, the stream restarts at the **playFrom** position when the **playTo** position is reached.

To set a specified time use one of these formats:

- The predefined string values “beginning” or “end”
- A string in the format **hh:mm:ss:cc**

  For example, **00:02:40:10** starts the stream two minutes, forty seconds, and ten hundredths of a second from the beginning of the stream (160.1 seconds).
- The number of milliseconds from the beginning of the stream

  For example, **160100** is the millisecond equivalent of 00:02:40:10 above.

popupMenu

Syntax: `popupMenu="true | false"`

Specifies whether the pop-up menu appears when the user clicks the right mouse button on the player. This pop-up menu allows basic operations like play and pause, rewind to the beginning of the movie, forward to the end of the movie, and close. The default value is true.

sliderRate

Syntax: `sliderRate=`*`milliseconds`*

Specifies the increments by which you can adjust the plug-in’s seek slider. This also affects the time units displayed in the status line and when the **OviObserver.onPositionChange()** method is called.

src

Syntax: `src="`*`file`*`.mpi"`

Required (if not used, **type** must be used). The **src** attribute is recognized by all browsers. Use the **src** attribute if your browser does not support the **type** attribute (for example, Microsoft Internet Explorer and Netscape 2.*x*). The **src** attribute identifies the type of plug-in to load by the extension of the file specified. In the

case of the Oracle Video Web Plug-in, this attribute doesn't identify the actual file to load, however: the plug-in actually gets that from the **mediafile** attribute.

Note that your HTTP server must be configured to recognize the **application/oracle-video** MIME type, which are associated with files with an extension of **.mpi**.

toolTips

Syntax: `toolTips="true | false"`

Indicates whether the plug-in displays tool tips when the user moves the mouse pointer over the plug-in area.

type

Syntax: `type="application/oracle-video"`

Required (if not used, **src** must be used). Specifies the MIME type. In this case, it invokes the plug-in and prepares it to play an Oracle video file. The plug-in gets the name of the file to play from the **mediafile** attribute.

The only value you should specify for the **type** attribute is **application/oracle-video**, as shown.

> **Note:** The **type** attribute only works in Netscape 3.0 or greater. For Microsoft Internet Explorer or Netscape Navigator 2.*x* browsers, use the **src** attribute. See "Specifying the Media File and MIME Type" on page 2-8 for more information.

volume

Syntax: `volume=`*`nnn`*

Where *n* is a volume level (integer value) in the range 0-100. Normally, allow the user to set the volume with the volume slider. This attribute is useful if there are multiple clips on the same server that have varying base volume levels.

width

Syntax: width=*nnn*

Required. Specifies the width of the plug-in in pixels. If the **border** attribute is not set, then this indicates the actual width of the plug-in. If the **border** attribute is set, then:

1. Multiply the border width times two.
2. Add this to the width that you want the video screen.
3. Use this number for **width**.

For example, if you want a video screen that is 320 pixels wide with a 10-pixel thick border, multiply the border times two to get 20, add this to the desired width of the screen, 320, for a width of 340 pixels.

JavaScript Methods

From JavaScript, you can call most of the methods on the plug-in defined in the **OviPlayer** class, as described in "OviPlayer" on page A-9. The methods you can call from JavaScript are:

forward()	resume()	setLoop()
getLength()	rewind()	setPopupMenu()
getMaxPos()	setAutoStart()	setPos()
getMinPos()	setFullScreen()	setVol()
getPos()	load()	stop()
getState()	pause()	unload()
getVol()	play()	

If you want to call any of these methods through the JavaScript LiveConnect interface, call the method through the plug-in's name. For example, you have a plug-in embedded into an HTML document with the following code:

```
<embed name="video1" height=320 width=200 src="oracle.mpi" mediafile="spud.spi">
```

Call the **setLoop()** method for this instance of the plug-in like this:

```
document.video1.setLoop(1);
```

See "OviPlayer" on page A-9 for descriptions of available methods' syntax.

Java Classes

OVC provides two classes to facilitate communications between Java applets, JavaScript, and the Oracle Video Web Plug-in. These classes are:

- OviPlayer
- OviObserver

OviPlayer

This section describes the public methods in the **OviPlayer** Java class. For information on controlling the Oracle Video Web Plug-in from a Java applet, see "Retrieving the OviPlayer Object" on page 2-25. The following methods are available:

advise()	getVol()	setFullScreen()
forward()	load()	setLoop()
getLength()	pause()	setPopupMenu()
getMaxPos()	play()	setPos()
getMinPos()	resume()	setVol()
getObserver()	rewind()	stop()
getPos()	setAutoStart()	unload()
getState()		

advise()

Syntax: `boolean advise(OviObserver o)`

Specifies the observer object for plug-in events. Returns true if the observer was successfully registered.

forward()

Syntax: `boolean forward()`

Forward the stream to the end of the movie or the position specified in the **playTo** property. Returns true if the forward operation was successful.

getLength()

Syntax: `int getLength()`

Returns the total length of the stream in milliseconds.

getMaxPos()

Syntax: `long getMaxPos()`

Returns the maximum stream position. If no end point has been set, either through the **playTo** attribute or the **play()** method, then **getMaxPos()** returns:

- The length of the stream, as long as the stream has a finite length; for example, a movie stored on the video server has a finite length
- Zero, if the stream is unbounded; for example, a live video feed has no set endpoint

Otherwise it returns the value of the end point.

getMinPos()

Syntax: `long getMinPos()`

Returns the minimum stream position. If no startpoint has been set, either through the **playFrom** attribute or the **play()** method, then **getMinPos()** returns zero. Otherwise it returns the value of the start point.

getObserver()

Syntax: `OviObserver getObserver()`

Returns the registered observer object.

getPos()

Syntax: `long getPos()`

Gets the current stream position in milliseconds.

getState()

Syntax: `int getState()`

Returns the current state of the player:

Table A–1 State Values for OviPlayer

Value	Description
OviPlayer.ST_EOS	Indicates the end of stream has been reached. If the **playTo** attribute was set or playback was started by calling the **play()** method with **to** and **from** parameters, playback stops at the position indicated.
OviPlayer.ST_ERROR	Indicates a non-recoverable error has occurred.
OviPlayer.ST_INIT	Indicates the player has been created and initialized; this state is also set after the **unload()** method returns.
OviPlayer.ST_PAUSED	Indicates the player is paused.
OviPlayer.ST_PLAYING	Indicates the player is playing.
OviPlayer.ST_REALIZED	Indicates the player has completed loading a stream.

When calling **getState()** through JavaScript, you can't access the symbolic representation of the return values. Instead you need to use the literal numeric values shown in Table A–2.

Table A–2 State Values for getState() JavaScript Call

Symbolic	Value
OviPlayer.ST_INIT	1
OviPlayer.ST_ERROR	2
OviPlayer.ST_REALIZED	4
OviPlayer.ST_PLAYING	5
OviPlayer.ST_PAUSED	6
OviPlayer.ST_EOS	7

getVol()

Syntax: `int getVol()`

Gets the play volume (0 to 100).

load()

Syntax: `boolean load(String mediafile)`

Loads a new stream. **mediafile** indicates the filename or asset cookie to load. Returns true if the media file was loaded successful.

After loading, if the **autoStart** attribute is false, the plug-in state is set to **OviPlayer.ST_REALIZED**. If **autoStart** is true, the stream starts playing and the plug-in state is set to **OviPlayer.ST_PLAYING**.

pause()

Syntax: `boolean pause()`

Pauses stream playback at the current position. Returns true and sets the plug-in state to **OviPlayer.ST_PAUSED** if the pause was successful.

play()

Syntax: `boolean play(), boolean play(String from, String to)`

Starts stream playback:

- The first version starts from the beginning of the stream or at the position set with **playFrom**. Playback continues until the end of the stream.
- The second version starts at the position indicated by **from** and continues to play until the position indicated by **to**.

Both methods return true if playback was successfully started. The plug-in state is set to **OviPlayer.ST_PLAYING**.

Note that you can't call **play()** while the player is paused (state set to **OviPlayer.ST_PAUSED)**, only when stopped. When the player is paused, call the **resume()** method to restart playback.

resume()

Syntax: `boolean resume()`

Resumes playback of the stream from the pause state. Returns true and sets state to set to **OviPlayer.ST_PLAYING** if playback was successfully resumed.

Note that you can't call **resume()** while the player is stopped (state **OviPlayer.ST_REALIZED)**, only when paused (state **OviPlayer.ST_PAUSED)**. When the player is stopped, call the **play()** method to start playback.

rewind()

Syntax: `boolean rewind()`

Rewind the stream to the beginning of the movie or the position specified in the **playFrom** property. Returns true if the rewind was successful.

setAutoStart()

Syntax: `void setAutoStart(boolean startflag)`

Turns the autostart feature on and off. If **startflag** is true, the stream starts automatically when loaded.

setFullScreen()

Syntax: `void setFullScreen(boolean mode)`

Sets normal and full-screen mode. Passing **mode** as true sets full-screen mode, while passing **mode** as false sets normal mode.

setLoop()

Syntax: `void setLoop(boolean loop)`

If **loop** is true, the position indicator returns to the beginning of the stream when playback reaches the end of the stream, except:

- If the **playFrom** property specifies a playback starting point other than the beginning of the stream. In that case, the position indicator returns to the **playFrom** position instead of the beginning.
- If the **playTo** property specifies a playback end point other than the end of the stream. In that case, the position indicator returns to the beginning (or **playFrom** position) when it reaches the **playTo** position.

setPopupMenu()

Syntax: `void setPopupMenu(boolean menu)`

Specifies whether the pop-up menu appears when the user clicks the right mouse button on the player. This pop-up menu allows basic operations like play and pause, rewind to the beginning of the movie, forward to the end of the movie, and close.

setPos()

Syntax: `boolean setPos(long position)`

Go to a specified stream position, indicated in milliseconds. Returns true if the position change was successful.

You should only call this method when the loaded stream is active; this means that the stream has been loaded and is currently playing or paused. **setPos()** does not function properly from the initialized or realized states (see **getState()** for information on plug-in states). Call **play()** with the **from** and **to** for initialized or realized streams.

After the call to this method, the stream maintains its current state: if playing, it continues playing, if paused, it remains paused.

setVol()

Syntax: `void setVol(int vol)`

Set the play volume (range from 0 to 100).

stop()

Syntax: `boolean stop()`

Stop playing the stream and rewind to the beginning. If a start position was specified by **playFrom** or the **from** parameter of **play()**, it's erased and the start position returns to the beginning of the stream. Returns true and sets player state to **OviPlayer.ST_REALIZED** if the stop operation was successful.

unload()

Syntax: `boolean unload()`

Unloads the stream. Returns true if the unload operation was successful.

OviObserver

The **OviObserver** interface specifies two methods you must complete when you implement the **OviObserver** interface. Because there are no implementations provided by OVC for these methods, the description of these methods explains when the methods are called. You then need to implement whatever functionality you want for these events. For more information on using the **OviObserver** interface, see "Using OviObserver" on page 2-26.

OviObserver contains the following methods:

onPositionChange()	onStop()

onPositionChange()

Syntax: `public void onPositionChange()`

Called when the stream changes position, as measured in regular increments. The default is one second, but this changes to reflect the setting of the **sliderRate** attribute.

onStop()

Syntax: `public void onStop()`

Called when the plug-in reaches the end of the stream or the **playTo** position if set.

B

Oracle Video Java Library Reference

This appendix describes the public Java classes and interfaces in the Oracle Video Java Library. You can use this library to build native Java applications that access the Oracle Video Server. You can find out how to use these classes in your applications by referring to Chapter 3, "Oracle Video Java Library".

This section contains descriptions of the following classes and interfaces:

Content

The **Content** class provides the ability to query an Oracle Video Server for a list of the available content titles in a directory of the media data store. **Content** provides only one public method, **query()**. Since this method is static, and **Content** has no data members or constants, you don't need to create **Content** objects.

You can find information on using the content classes **Content**, **ContentException**, and **ContentIter** in "Querying Available Content Titles" on page 3-26.

query()

Syntax: `static StmInfo[] query(String srvAddr, String spec, ContentIter iter)`

This method returns an array of **StmInfo** objects containing information about all of the content titles that:

- Are on the server indicated by **srvAddr**
- Meet the specification **spec** on the server indicated by **srvAddr**

Use the **ContentIter** to move through the returned objects. See "Querying Available Content Titles" on page 3-26 for more information on how to query for content titles.

ContentException

ContentException objects are thrown when a problem occurs during content query operations. **ContentException** contains the following members:

Constants	
ContentException.EX_BADPARAM	ContentException.EX_INTERNAL
ContentException.EX_BADSTATE	ContentException.EX_NOTIMPL
ContentException.EX_ERROR	ContentException.EX_UNTRANS
Data Members	
m_code	m_type
m_msg	
Methods	
toString()	

You can find information on using the content classes **Content**, **ContentException**, and **ContentIter** in "Querying Available Content Titles" on page 3-26.

ContentException Constants

All of the constants defined in the **ContentException** class are of type **int**. You can check the particular type of exception thrown by examining the **ContentException** member **m_type**, which is set to one of these constants.

ContentException.EX_BADPARAM

Indicates that one of the parameters passed to the routine was invalid in the context of that **Content** method. For instance, if you call **Content.query()** with an uninitialized **ContentIter** reference, a **ContentException** with this value is thrown.

ContentException.EX_BADSTATE

Indicates that a **Content** method was called during an invalid state.

ContentException.EX_ERROR

Indicates an error has occurred that is not covered by any other **ContentException** constants.

ContentException.EX_INTERNAL

Indicates that an internal (unexpected) error occurred. In this case, **m_code** and **m_msg** are set and should be reported to Oracle Worldwide Customer Support Services.

ContentException.EX_NOTIMPL

Indicates the caller attempted to use functionality that is not implemented in the current version of the Oracle Video Java Library.

ContentException.EX_UNTRANS

Indicates that an untranslatable error has occurred.

ContentException Data Members

The **ContentException** class contains the following public data members:

m_code

Type: int

Set to an Oracle-specific error code. This code may be useful in debugging and problem solving interactions with Oracle Worldwide Customer Support Services.

m_msg

Type: int

Contains a string containing additional information on the thrown exception.

m_type

Type: int

Indicates the type of exception thrown by a **Content** method. This is one of the **EX_*** constants described in "ContentException Constants" on page B-3.

ContentException Methods

The **ContentException** class contains the following public method:

toString()

Syntax: `String toString()`

Returns a single string consisting of all the information contained in each of the member variables.

ContentIter

ContentIter works in conjunction with the **Content** class to let you retrieve lists of available content titles from an Oracle Video Server. The process of retrieving these titles is iterative, which means that it is performed in steps, by retrieving only a few titles at a time. You use a **ContentIter** object to manage this process.

ContentIter contains the following members:

Data Members	
m_num	m_pos
Methods	
ContentIter()	

You can find information on using the content classes **Content**, **ContentException**, and **ContentIter** in "Querying Available Content Titles" on page 3-26.

ContentIter Data Members

ContentIter contains the following public data members.

m_num

Type: int

Indicates the number of titles to retrieve with each query operation. When all available titles have been retrieved, **Content.query()** sets this to the number of titles actually retrieved with the last query. **Content.query()** also sets **m_pos** to -1 when the last titles have been queried.

m_pos

Type: int

Indicates the current position in the list of available titles. **Content.query()** sets this to -1 when all available titles have been retrieved. **Content.query()** also sets **m_num** to the number of titles actually retrieved with the last query.

ContentIter Methods

ContentIter contains only one public method, its constructor.

ContentIter()

Syntax: `ContentIter(int pos, int num)`

Sets the iterator values to be used during a query operation:

- **pos** indicates the position in the list of available titles at which you want to start. For example, if your last query retrieved the first 30 titles from a particular server, you may not want to retrieve those again. You could then set **pos** to 30 before your next query.
- **num** indicates the number of titles you want to retrieve with each query operation.

Player

Player is the central object in the Oracle Video Java Library. It provides:

- A set of constants for defining the player state
- Default user interface
- Media control API
- Service methods

Unlike the **PlayerListener** interface, the **Player** interface is already implemented at a lower level in the **oracle.ovc** package and therefore cannot be implemented in your own Java application. Note that **Player** does not have a constructor: to create a **Player** object, call **PlayerFactory.getPlayer()**. See "getPlayer()" on page B-19 for more information on this method.

Player contains the following members:

Constants		
Player.ST_EOS	Player.ST_PAUSED	Player.ST_REALIZED
Player.ST_ERROR	Player.ST_PLAYING	Player.ST_UNINIT
Player.ST_INIT		

Methods		
addListener()	getStatusComp()	setFullScreen()
getControlComp()	getVisualComp()	setPos()
getInfo()	getVol()	setVol()
getPlayerUI()	load()	stateToString()
getPos()	pause()	stop()
getSelRange()	play()	term()
getState()	resume()	unload()
getStats()		

Player does not have a public constructor. To create new **Player** objects, use the **PlayerFactory.getPlayer()** method. See "PlayerFactory" on page B-19 for more information.

Player Constants

This section details the possible states of the **Player** object. All **Player** states are expressed as constants in the form **Player.ST_***.

Player State Reference

This section describes the **Player** states and the various ways a **Player** object can arrive at these states. Methods that are briefly mentioned in this section are discussed in detail in "Player Methods" on page B-8. For more information on tracking **Player** states, see "Stream and Player State" on page 3-13 and "Handling Player Events" on page 3-17.

Player.ST_EOS

Indicates the end of stream has been reached. **PlayerListener.endOfStream()** provides another method for determining when the player reaches the end of a stream.

Player.ST_ERROR

Indicates a non-recoverable error has occurred. **PlayerListener.error()** provides another method for determining when such errors occur.

Player.ST_INIT

Indicates the **Player** object has been created and initialized, without having a stream loaded. You can also encounter this state after the **unload()** method returns.

Player.ST_PAUSED

Indicates the **Player** object is paused.

Player.ST_PLAYING

Indicates the **Player** object is playing the current stream. This happens after the **play()** or **resume()** methods return. It also happens after **load()** returns if **StmOpts.m_autoStart** is **true.** See the **StmOpts** class reference on page B-26 for more information.

Player.ST_REALIZED

Indicates the player has completed loading a stream.

Player.ST_UNINIT

Indicates the player has been terminated and uninitialized. The player is in this state a very brief period of time after the **term()** method is called. Note that the **term()** method should only be called after **unload()** returns and the state is **Player.ST_INIT**. You can not reuse a **Player** object after calling **term()**.

Player Methods

Player methods break down into three general categories:

- User Interface Methods
- Media Control Methods
- Player Service Methods

There are some things that all of these methods share:

- All of the **Player** methods can throw **PlayerException** objects. You must place **Player** method calls within **try** blocks and catch any **PlayerException** objects in order for your code to compile.
- Each of the methods presented here are declared **public**.

User Interface Methods

The following methods deal with retrieving or configuring the user interface portion of the video client.

getControlComp()

Syntax: `Component getControlComp()`

Retrieves the standard controller interface component to be added to the player interface. The controller includes a Play / Pause button, a Stop button and a Seek scrollbar.

getPlayerUI()

Syntax:
```
Component getPlayerUI()
Component getPlayerUI(boolean Video, boolean Controls, boolean Status)
Component getPlayerUI(boolean Video, boolean Controls, boolean Status,
                      int width, int height)
```

Retrieves the video screen (visual component), the controller panel and the status line panel bundled together as a single component to be added to the player interface.

- The first form of this method returns the interface with all of the components.
- The second form lets you select which interface components are displayed.
- The third form, like the second, lets you select which interface components are displayed, but also lets you specify a width and height for the interface. This forces the interface component to keep a fixed width and height, even if the host window is resized.

getSelRange()

Syntax: `void getSelRange(StmPos from, StmPos to)`

Retrieves the current start and end positions of the player and puts them in the **from** and **to** parameters.

getStatusComp()

Syntax: `Component getStatusComp()`

Retrieves the standard status line interface component to be added to the player interface. This status line component includes a stream position counter and a status message text area.

getVisualComp()

Syntax: `Component getVisualComp()`

Retrieves the video screen (visual component) as a interface component to be added to the player interface.

Media Control Methods

The following methods give you control over stream playback.

getPos()

Syntax: `StmPos getPos(int fmt)`

Returns the current position of the stream in the form of a **StmPos** object. This method requires that the position format, **fmt**, be stipulated by the Java application.

The only two position formats supported by this method are **StmPos.POSFMT_TIME** and **StmPos.POSFMT_FRAMES**. You can call this method from any state in which a stream is present. See the **StmPos** class reference on page B-29 for more information.

getVol()

Syntax: `int getVol()`

Returns the current volume setting of the player as an integer in the range of 0 to 100. You can call this method from any state in which a stream is present.

load()

Syntax: `void load(String mediafile, StmOpts opts)`

Loads the media content file and prepares it for playback. You can call this method from all active states.

mediafile is a **String** object containing a media file specifier. This specifier describes the location of the media resource to be loaded. See Appendix D, "The Media File", for more information on media file specifiers.

StmOpts defines a number of stream attributes, such as auto start, start position, end position, and so on. See the **StmOpts** class reference on page B-26 for more information.

Pass a null **StmOpts** object if you want all of the following default media stream options:

- **m_autoStart=false**
- **m_loop=false**
- **m_volume=StmOpts.DEFAULT_VOL**
- **m_playFrom=beginning**
- **m_playTo=end**

For example:

```
try {
   myapp.m_player.load("/mds/video/oracle1.mpi", null);
}
catch(PlayerException e) {
   System.out.println("Failed to load default stream.");
}
```

pause()

Syntax: `void pause()`

Pauses playback, preserving the current position. The current frame will remain on screen. Can only be called from **Player.ST_PLAYING**.

play()

Syntax: `void play(), void play(StmPos from, StmPos to)`

Starts stream playback:

- The first version starts from the beginning of the stream or the position indicated by the **m_playFrom** member of the **StmOpts** object passed to **load()**. Unless interrupted, playback continues until the stream reaches its end or the position indicated by the **m_playTo** member of the **StmOpts** object passed to **load()**.
- The second version starts playback from the position indicated by the **from** parameter and continues until the stream reaches the position indicated by the **to** parameter. The **from** and **to** parameters override the start and end positions set in the **m_playFrom** and **m_playTo** members of the **StmOpts** object passed to **load()**, but do not replace them: the next call to the first version of **play()** uses the original start and end positions.

You must call this method while the player is stopped. You can't call **play()** to resume play from a paused state; in that case, call **resume()**.

resume()

Syntax: `void resume()`

Resumes playback of the stream from the current position. You must call this method while the player is paused. You can't call **resume()** to resume play from a stopped state; in that case, call **play()**.

setFullScreen()

Syntax: `void setFullScreen(boolean mode)`

Toggles player between normal and full-screen mode. Passing **mode** as **true** sets full-screen mode, while passing **mode** as **false** sets normal mode.

setPos()

Syntax: `void setPos(StmPos pos)`

Moves the stream position to the value contained in the **StmPos** object. You can call this method from any state in which a stream is present. See the **StmPos** class reference on page B-29 for more information.

setVol()

Syntax: `void setVol(int vol)`

Sets the current volume of the **Player** as an integer from 0 to 100. You can call this method from any state in which a stream is present.

stateToString()

Syntax: `String stateToString(int state)`

Converts the player state indicated by **state** into a text description contained in the returned **String** value. Use this to convert **Player** constants such as **Player.ST_PLAYING** or **Player.ST_ERROR** returned from **getState()** into text.

stop()

Syntax: `void stop()`

Stops playback and resets the stream position to the beginning of the stream. If a start position was specified in **StmOpts.m_playFrom** or the **from** parameter of **play()**, it's erased and the start position returns to the beginning of the stream. The stream rewinds to the first frame, which appears in the video screen. Note that encoding sometimes makes the first frame black. You can call this method while the player state is one of **Player.ST_PLAYING**, **Player.ST_PAUSED**, or **Player.ST_EOS**.

unload()

Syntax: `void unload()`

Releases any resources associated with the current stream. You can call this method from any state in which a stream is present.

Player Service Methods

The following methods give you access to the internals of the **Player** object, such as player state and configuring the listener object.

addListener()

Syntax: void addListener(PlayerListener listener)

Registers the listener object as a **PlayerListener**. The listener is notified of specific events occurring in the **Player** object through the **PlayerListener** interface methods.

This example shows a simple class that implements the **PlayerListener** interface, creates a **Player** object, and adds itself as a listener to the **Player** object.

```
public class myClass implements PlayerListener {
   Player m_player = null;
   myClass app = null;

   public static void main(String args[]) {
       app = new myClass();

       m_player = PlayerFactory.getPlayer();
       m_player.addListener(app);
       m_player.load(args[0], null);
       m_player.play();
       }

   // PlayerListener methods
   public void stateChange(int newState) {
       System.out.println("Player state change to: " + newState);
       }
   public void error(int code, String msg) {
       System.out.println("Error! Code: " + code);
       }
   public void endOfStream() {
       System.out.println("End of stream reached!");
       }
   }
```

Note: A Java application or object must implement the **PlayerListener** interface to be registered as a listener object. See the **PlayerListener** interface reference on page B-20 for more information.

getInfo()

Syntax: `StmInfo getInfo()`

Returns a **StmInfo** object containing information about the current stream. You can call this method from any state in which a stream is present. See the **StmInfo** class reference on page B-21 for more information.

getState()

Syntax: `int getState()`

Returns the current state of the **Player** object. The return value of this method can be any of the values described in "Player Constants" on page B-7. You can call this method from any **Player** state.

getStats()

Syntax: `StmStats getStats()`

Returns a **StmStats** object containing information concerning the current playback states and statistics of the stream. You can call this method from any state in which a stream is present. See the **StmStats** class reference on page B-34 for more information.

term()

Syntax: `int term()`

Terminates the internal Oracle Video Client processes but does not release the **Player** object in the Java context. If necessary, this method calls **unload()** before terminating the player. This method should be called prior to exiting applications that use the Oracle Video Java Library.

Note: If you want an application to free the **Player** object in the Java context and continue running, call this method, set the **Player** object to **null**, and call the Java garbage collection service.

Setting the **Player** object to **null** allows the Java garbage collector to immediately release all resources associated with the object. For example, this code sample terminates the **Player**, sets the object reference to **null**, then calls the Java garbage collector:

```
try {
   app.m_player.term();
}
catch(PlayerException e){
   System.out.println("term() failed.");
}

app.m_player = null;
System.gc();
```

PlayerApplet

PlayerApplet is a simple Java applet that supports the same syntax and features as the Oracle Video Web Plug-in, which is described in Chapter 2, "Oracle Video Web Plug-in" and Appendix A, "Oracle Video Web Plug-in Reference". Note that the applet is not signed, so it may not function in all browsers. You can find out more about using **PlayerApplet** in "Using the PlayerApplet Class" on page 3-31.

PlayerApplet recognizes the same attributes to the **<APPLET>** tag that the Oracle Video Web Plug-in recognizes for the **<EMBED>** tag. For information on these attributes, see "<Embed> Attributes" on page A-1.

PlayerException

The **PlayerException** class represents exceptions thrown specifically by the methods of the **Player** class. *All* public methods of the **Player** class throw **PlayerException** exception objects.

The **PlayerException** object's **m_type** member represents the exception type thrown. This is indicated by a constant of the format **EX_*** (the constants used are described in "PlayerException Constants" on page B-17). The **PlayerException** object's **m_code** property is set to an Oracle-specific error code useful for debugging purposes. The **m_msg** property is a **String**, which may contain additional information about the exception thrown.

PlayerException contains the following members:

Constants	
PlayerException.EX_BADPARAM	PlayerException.EX_INTERNAL
PlayerException.EX_BADSTATE	PlayerException.EX_NOTIMPL
PlayerException.EX_ERROR	PlayerException.EX_UNTRANS
Data Members	
m_code	m_type
m_msg	
Methods	
toString()	

PlayerException Constants

All of the constants defined in the **PlayerException** class are of type **int**. You can check the type of exception thrown by examining **m_type**, which is set to one of these constants.

PlayerException.EX_BADPARAM

Indicates that one of the parameters passed to the routine was invalid in the context of that **Player** method. For instance, if you call **Player.setVol()** with an integer value greater than 100 or less than 0, a **PlayerException** with this value is thrown.

PlayerException.EX_BADSTATE

Indicates that a **Player** method was called during an invalid state. For instance, calling **Player.pause()** while loading a stream causes this **PlayerException** to be thrown.

PlayerException.EX_ERROR

Indicates an error has occurred that is not covered by any other **PlayerException** constants.

PlayerException.EX_INTERNAL

Indicates that an internal (unexpected) error occurred. In this case, **m_code** and **m_msg** are set and should be reported to Oracle Worldwide Customer Support Services.

PlayerException.EX_NOTIMPL

Indicates the caller attempted to use functionality that is not implemented in the current version of the Oracle Video Java Library.

PlayerException.EX_UNTRANS

Indicates that an untranslated error has occurred. To translate the error, call the **toString()** method.

PlayerException Data Members

The **PlayerException** class contains the following public data members:

m_type

Type: `int`

Indicates the type of exception thrown by a **Player** method. This is one of the **EX_*** constants described in "PlayerException Constants" on page B-17.

m_code

Type: `int`

Set to an Oracle-specific error code. This code may be useful when debugging and solving OVC and Oracle Video Java Library problems with Oracle Worldwide Customer Support Services.

m_msg

Type: `int`

Contains a string containing additional information on the thrown exception.

PlayerException Methods

The **PlayerException** class contains the following public method:

toString()

Syntax: `String toString()`

Returns a single string consisting of all the information contained in each of the member variables.

PlayerFactory

The **PlayerFactory** class provides a platform-independent means of creating new **Player** objects. You must call the methods of **PlayerFactory** class to create **Player** objects, rather than calling a **Player** constructor method.

PlayerFactory contains the following members:

Methods		
createPlayer()	getPlayer()	getPlayerFactory()

PlayerFactory Methods

As with the members of the **Player** class, all the methods presented here are declared **public** and can throw a **PlayerException** object.

createPlayer()

Syntax: `Player createPlayer()`

Instantiates a **Player** object for use by the calling Java application. **createPlayer()** throws a **PlayerException** object if a **Player** cannot be instantiated.

getPlayer()

Syntax: `static synchronized Player getPlayer()`

Returns a new **Player** object appropriate to the current platform. This is the easiest and most common method of instantiating new **Player** objects. This method is equivalent to calling **PlayerFactory.getPlayerFactory().createPlayer()**.

Here is an example of calling **getPlayer()**:

```
try {
   myapp.m_player = PlayerFactory.getPlayer();
}
catch(PlayerException e) {
   System.out.println("getPlayer() failed...");
}
```

getPlayerFactory()

Syntax: `static synchronized PlayerFactory getPlayerFactory()`

Returns the default **PlayerFactory** object for the current platform, which can then be used to instantiate appropriate **Player** objects with the **createPlayer()** method. Throws a **PlayerException** object if a factory is not available for the platform.

PlayerListener

The **PlayerListener** interface can be implemented by Java applications or objects that wish to be notified of state changes that occur in the **Player** object.

To receive notification from a **Player** object, the Java application or object must:

- Properly register itself as a **PlayerListener** by calling **Player.addListener()**; see "Player Service Methods" on page B-13 for more details about this method.
- Implement all of the **PlayerListener** methods described in this section

PlayerListener contains the following members:

Methods		
endOfStream()	error()	stateChange()

PlayerListener Methods

All **PlayerListener** methods must be implemented if you want a **Player** object to notify your Java application or object each time an event occurs. Once properly registered, each of these methods is invoked as the corresponding event occurs. For information on registering an object as a listener, see “Handling Player Events” on page 3-17.

error()

Syntax: `void error(int code, String msg)`

Indicates a known, non-recoverable error has occurred in the **Player** object. **code** is an error code specific to the Oracle Video Client in the format OVC-*xxxx*. **msg** describes the error.

After this method is invoked, the player is in the state **Player.ST_ERROR**. This is a non-recoverable state: the player must be unloaded using the **Player.unload()** method.

endOfStream()

Syntax: `void endOfStream()`

Indicates the player has reached the end of the current stream.

stateChange()

Syntax: `void stateChange(int newState)`

Indicates the **Player** object has changed state.

One of the states of which **stateChange()** is notified is **Player.ST_EOS** (end of the current stream). This performs the same function as the **endOfStream()** method, providing another way of detecting the end of a stream. For a complete list of **Player** states and their descriptions, refer to "Player Constants" on page B-7.

StmInfo

The **StmInfo** class represents static information about the current stream. **StmInfo** objects are returned by the **Player.getInfo()** method.

StmInfo contains the following members:

Constants		
StmInfo.CSTAT_DISK	StmInfo.CSTAT_LOCALFILE	StmInfo.CSTAT_TERMINATED
StmInfo.CSTAT_FEED	StmInfo.CSTAT_NETWORK	StmInfo.CSTAT_UNKNOWN
StmInfo.CSTAT_TAPE	StmInfo.CSTAT_ROLLING	

Data Members		
m_aspect	m_contType	m_name
m_asset	m_createTime	m_proto
m_bitrate	m_desc	m_size
m_bytes	m_fps	m_url
m_contStat	m_msecs	

Methods	
contStatToString()	toString()

StmInfo Constants

All of the constants defined in the **StmInfo** class are of type **int**. You can check the container status of the current stream by examining the **StmInfo** member **m_contStat**, which is set to one of these constants. The container status indicates what type of container the current stream is using.

StmInfo.CSTAT_DISK

Indicates the current stream originates from a file stored on disk on the Oracle Video Server.

StmInfo.CSTAT_FEED

Indicates the current stream comes from a one-step encode feed.

StmInfo.CSTAT_LOCALFILE

Indicates the current stream originates from your local file system.

StmInfo.CSTAT_NETWORK

Indicates the current stream comes from a network feed, such as multicast broadcast or other wide-band network feed.

StmInfo.CSTAT_ROLLING

Indicates the current stream comes from a rolling live feed.

StmInfo.CSTAT_TAPE

Indicates the current stream originates from a file stored in the Hierarchical Storage Manager on the Oracle Video Server.

StmInfo.CSTAT_TERMINATED

Indicates the current stream comes from a terminated feed.

StmInfo.CSTAT_UNKNOWN

Indicates the current container status is unknown.

StmInfo Data Members

The **StmInfo** class contains the following public data members:

m_aspect

Type: `int`

Contains the video aspect ratio.

So that the aspect ratio, which usually contains a decimal portion, can be contained in an **int** instead of a **float**, the value of **m_aspect** actually contains the aspect ratio multiplied by 1,000. For example, the aspect ratio 1.6 would be represented as 1600.

m_asset

Type: `String`

Contains the logical content asset cookie name.

m_bitrate

Type: `int`

The total bit rate of the current stream. This includes the audio component, the video component, and the container overhead.

m_bytes

Type: `long`

File size in bytes of the stored content. This is zero if the stream is unbounded, such as with a live video feed.

m_contStat

Type: `int`

Indicates the content status of the stream. This is one of the **StmInfo** constants of the form **CSTAT_***. You can convert the integer value contained in this property into a **String** description by calling the **contStatToString()** method.

m_contType

Type: String

Contains the container type of the stream.

m_createTime

Type: Date

Contains the date and time the content was created, contained in a Java **Date** object.

m_desc

Type: String

Contains a text description of the stream.

m_fps

Type: int

Contains the frame rate of the current video component. The format is:

fps x 1,000

where *fps* is frames per second. To find the actual frames per second, divide this by 1,000.

m_msecs

Type: int

File size in milliseconds of the stored content. This is zero if the stream is unbounded, such as with a live video feed.

m_name

Type: String

Contains the name of the stream.

m_proto

Type: `String`

Contains the transport protocol of the stream.

m_size

Type: `Dimension`

Contains the source input size.

m_url

Type: `String`

Contains the media file specifier used in the **Player.load()** method. See Appendix D, "The Media File" for a description of media file specifiers.

StmInfo Methods

The **StmInfo** class contains the following public methods:

contStatToString()

Syntax: `static String contStatToString(int cstat)`

Converts the stream container status indicated by **cstat** into a text description contained in the returned **String** value. Use this to convert the **m_contStat** property and **StmInfo** constants such as **StmInfo.CSTAT_DISK** or **StmInfo.CSTAT_FEED** into text.

toString()

Syntax: `String toString()`

This routine returns a single string consisting of all the information contained in each of the **StmInfo** member variables.

StmOpts

The **StmOpts** class consists of various media stream options. Modification of these options allows you to customize attributes of the stream playback.

> **Note:** The **StmOpts** object is passed to the **Player.load()** method. Typically, if default values are desired, pass a null **StmOpts** object when loading a stream. Alternatively, create an object of this class that uses some of the default values and customizes other values.

See the code sample provided on page B-9 for an example of passing a null default **StmOpts** object to the **Player.load()** method. See the code sample on page B-29 for an example of passing a modified **StmOpts** object to the **Player.load()** method.

StmOpts contains the following members:

Constants		
StmOpts.DEFAULT_VOL		
Data Members		
m_autoStart	m_loop	m_popup
m_img	m_playFrom	m_volume
m_leftClick	m_playTo	
Methods		
StmOpts()		

StmOpts Constants

All constants in the **StmOpts** class are of type **int**.

StmOpts.DEFAULT_VOL

This constant indicates the default player volume. Setting **StmOpts.m_volume** to this when calling **Player.load()** leaves the volume unchanged.

StmOpts Data Members

The **StmOpts** class contains the following public data members:

m_autoStart

Type: `boolean`

Specifies whether playback starts automatically. If **true**, the player starts playback as soon it loads the stream. This is slightly faster than calling **Player.load()** then **Player.play()**.

The default value is **false**, meaning that you must explicitly call **Player.play()** to begin playback.

m_img

Type: `String`

Contains the name of an image file to be displayed when the video screen is blank.

m_leftClick

Type: `boolean`

Specifies whether a left click on the video screen toggles Play and Pause. If **m_leftClick** is **true**, left clicking in the video screen has the same effect as clicking on the Play / Pause button. The default value is **true**.

m_loop

Type: `boolean`

If **true**, the player, upon reaching the end of the stream, returns to the beginning of the stream and continues playback from that point. If start and end points were set, then the player returns from the end point to the start point and continues playback.

The default value is **false** (no looping).

m_playFrom

Type: StmPos

Sets the starting position of the stream. The default for this member is the beginning of the stream. Internally this is represented as a **StmPos**(**StmPos.POSFMT_BEGINNING**) object.

m_playTo

Type: StmPos

Sets the ending position of the stream. Playback stops when this position is reached. The default value for this member is the end of the stream. Internally this is represented as a **StmPos**(**StmPos.POSFMT_END**) object.

m_popup

Type: boolean

Specifies whether the pop-up menu appears when the user clicks the right mouse button on the player. This pop-up menu allows basic operations like play and pause, rewind to the beginning of the movie, forward to the end of the movie, and close. The default value is true.

m_volume

Type: int

Contains the audio volume setting for the stream. This value is an integer in the range of 0 - 100. The default value for this member is **StmOpts.DEFAULT_VOL**, which leaves the volume at its current setting.

StmOpts Methods

The **StmOpts** class contains the following public methods:

StmOpts()

Syntax: `StmOpts()`

The default constructor for the **StmOpts** class. Sets the **StmOpts** member fields to their default values. To override an option, set the member value explicitly before calling the **Player.load()** method.

Here is an example of overriding a **StmOpts** default member value:

```
//instantiate a default StmOpts object and change the m_playFrom position
StmOpts myStmOpts = new StmOpts();
myStmOpts.m_playFrom = 5000;

// Load the stream with the customized StmOpts object
try {
   myapp.m_player.load("/mds/video/oracle1.mpi", myStmOpts);
}
catch(PlayerException e) {
   System.out.println("Failed to load customized stream.");
}
```

StmPos

The **StmPos** class represents stream position information in various formats. This class is used to specify stream positions as passed to the **Player** with the **Player.setPos()** method, or as received from the **Player** with the **Player.getPos()** method. **StmPos** objects are also utilized as member variables in the **StmOpts** class.

StmPos contains the following members:

Constants		
StmPos.POSFMT_BEGINNING	StmPos.POSFMT_DEFAULT	StmPos.POSFMT_FRAMES
StmPos.POSFMT_CURRENT	StmPos.POSFMT_END	StmPos.POSFMT_TIME
Data Members		
m_fmt	m_val	
Methods		
StmPos()	fromString()	toString()

StmPos Constants

StmPos constants specify the possible formats for the stream position. These constants are used when instantiating **StmPos** objects. There are two forms of the **StmPos()** constructor:

- **StmPos(int fmt)**
- **StmPos(int fmt, long val)**

The constants presented in this section are used for the **int fmt** parameter in both versions of the constructor. Which version of the constructor you use depends on what you specify for **fmt**. This is discussed both in this section and in "StmPos Methods" on page B-31.

StmPos.POSFMT_BEGINNING

Indicates that the desired stream position is the beginning of the stream. Use the **StmPos(int fmt)** version of the constructor to specify the **StmPos.POSFMT_BEGINNING** constant.

StmPos.POSFMT_CURRENT

Indicates that the desired stream position is the current position (that is, don't change the stream position). Use the **StmPos(int fmt)** version of the constructor to specify the **StmPos.POSFMT_CURRENT** constant.

StmPos.POSFMT_DEFAULT

Indicates that the desired stream position is the default position. The default position is the beginning of the stream for all types of streams except for unbounded streams. For unbounded streams, the default position is the end of the stream. Use the **StmPos(int fmt)** version of the constructor to specify the **StmPos.POSFMT_DEFAULT** constant.

StmPos.POSFMT_END

Indicates that the desired stream position is the end of the stream. Use the **StmPos(int fmt)** version of the constructor to specify the **StmPos.POSFMT_END** constant.

StmPos.POSFMT_FRAMES

Indicates that the desired stream position is expressed as the number of frames from the beginning of the stream. Use the **StmPos(int fmt, long val)** version of the constructor to specify the **StmPos.POSFMT_FRAMES** constant.

StmPos.POSFMT_TIME

Indicates that the desired stream position is expressed as a time in milliseconds from the beginning of the stream. Use the **StmPos(int fmt, long val)** version of the constructor to specify the **StmPos.POSFMT_TIME** constant.

StmPos Data Members

The **StmPos** class contains the following public data members:

m_fmt

Type: `int`

The format of the position field. This is limited to one of the **StmPos** constants.

m_val

Type: `long`

The format-specific stream position value. This field is only used for the time value formats of **StmPos.POSFMT_TIME** and **StmPos.POSFMT_FRAMES**.

StmPos Methods

There are three ways of instantiating objects of the **StmPos** type: two **StmPos** constructors and a **fromString()** method. Each of these ways results in the instantiation of a new **StmPos** object, and each has its appropriate usage.

All of the methods presented here are declared **public** within the **StmPos** class definition. Unlike some other classes in the Oracle Video Java Library, only the constructors, not all of the **StmPos** methods, throw the **PlayerException** object.

StmPos()

Syntax: `StmPos(int fmt), StmPos(int fmt, long val)`

Creates a new **StmPos** object:

- The first constructor specifies only the **m_fmt** format field of the **StmPos** object and can be used when the **m_val** field is not required by the object. **m_val** is not required when you instantiate objects of the following formats:
  - **StmPos.POSFMT_BEGINNING**
  - **StmPos.POSFMT_END**
  - **StmPos.POSFMT_CURRENT**
  - **StmPos.POSFMT_DEFAULT**

  In these cases, the positional information for the object is contained in the format field. A **PlayerException** object is thrown if the format type passed requires **m_val** to be specified.
- The second constructor sets both the format and the value of a stream's positional information. You can only use this constructor when instantiating **StmPos** objects of the formats **StmPos.POSFMT_TIME** or **StmPos.POSFMT_FRAMES**.

  Note that this constructor expects **val** to be in milliseconds. For instance, the constructor interprets a long value of 4000 to mean 4 seconds (4000 milliseconds).

See Table B–1 for additional **StmPos** object constructor information.

fromString()

Syntax: `static StmPos fromString(String pos)`

This third method of instantiating **StmPos** objects can be used to convert an input string into a **StmPos** object. Because it is possible to instantiate any of the **StmPos** object formats with this method, this method is the most safe and versatile way of instantiating **StmPos** objects.

See Table B–1 for the available input string formats.

Table B–1 Positional Values and StmPos constructor equivalents

Position Value	Equivalent StmPos Constructor
milliseconds	`StmPos(StmPos.POSFMT_TIME, timeVal)`
"hh:mm:ss:cc"	`StmPos.fromString("hh:mm:ss:cc")`
frame_num	`StmPos(StmPos.POSFMT_FRAMES, frameVal)`
"beginning"	`StmPos(StmPos.POSFMT_BEGINNING)` `StmPos.fromString("beginning")`
"current"	`StmPos(StmPos.POSFMT_CURRENT)` `StmPos.fromString("current")`
"end"	`StmPos(StmPos.POSFMT_END)` `StmPos.fromString("end")`
"default"	`StmPos(StmPos.POSFMT_DEFAULT)` `StmPos.fromString("default")`

toString()

Syntax: `static String toString()`

This method converts a **StmPos** object into a **String**. See Table B–1 for the possible returned string formats.

StmStats

The **StmStats** class contains dynamic information about both the playback states and statistics of the current media stream. This object is returned from the **Player.getStats()** method.

StmStats contains the following members:

Constants		
StmStats.STM_CONTROL	StmStats.STM_IDLE	StmStats.STM_PLAYING
StmStats.STM_ENDED	StmStats.STM_PAUSED	StmStats.STM_STALLED
Data Members		
m_bps	m_drops	m_minTime
m_cnsState	m_fBytes	m_pkts
m_curFrame	m_fps	m_prdState
m_curTime	m_maxTime	m_rBytes
Methods		
stateToString()	toString()	

StmStats Constants

These constants represent the current network stream state, not the state that is returned by the **Player.getState()** method.

StmStats.STM_CONTROL

Indicates that the stream is in the middle of a network transaction.

StmStats.STM_ENDED

Indicates that the end of stream has been reached.

StmStats.STM_IDLE

Indicates that the stream is idle.

StmStats.STM_PAUSED

Indicates that the stream is paused.

StmStats.STM_PLAYING

Indicates that the stream is playing.

StmStats.STM_STALLED

Indicates that the stream has stalled.

StmStats Data Members

The **StmStats** class contains the following public data members:

m_bps

Type: `int`

Indicates the average bit rate in bits per second (bps) over the life of the current stream.

m_cnsState

Type: `int`

Playback state of the current media stream. This is an integer value limited to one of the **StmStats** listed constants, such as **StmStats.STM_CONTROL** or **StmStats.STM_PLAYING**.

m_curFrame

Type: `long`

Current frame that appears on the video screen. The number of frames starts with 0 at the beginning or **playFrom** position and increments with each successive frame shown.

m_curTime

Type: `long`

Indicates the current position in the stream, in hundredths of a second.

m_drops

Type: int

Indicates the number of packet drops since playback started. Unreliable network protocols, such as UDP, tend to drop some packets.

m_fBytes

Type: int

Indicates the number of free bytes in the client cache.

m_fps

Type: int

Indicates the average frame rate in frames per second (fps) over the life of the current stream.

m_maxTime

Type: long

Indicates the latest seekable time in the stream, in hundredths of a second.

m_minTime

Type: long

Indicates the earliest seekable time in the stream, in hundredths of a second.

m_pkts

Type: int

The number of packets received since stream playback began.

m_prdState

Type: `int`

Playback state of the current media stream. This is an integer value limited to one of the **StmStats** listed constants, such as **StmStats.STM_CONTROL** or **StmStats.STM_PLAYING**.

m_rBytes

Type: `int`

Indicates the number of bytes in the client cache ready for consumption.

StmStats Methods

The **StmStats** class contains the following public methods:

stateToString()

Syntax: `static String stateToString(int state)`

Converts the stream state indicated by **state** into a text description contained in the returned **String** value. Use this to convert **StmStats** constants such as **StmStats.STM_PLAYING** or **StmStats.STM_ENDED** into a text description. The **m_cnsState** and **m_prdState** properties both use these constants.

toString()

Syntax: `String toString()`

Returns a single string consisting of all the information contained in each of the member variables.

C

Oracle Video ActiveX Control Reference

This appendix contains the reference information for the \. It can be used to embed the Oracle Video Client in applications that support the Microsoft ActiveX standard. You can find out how to use these classes in your applications by referring to Chapter 4, “Oracle Video ActiveX Control”.

This section contains descriptions of the following topics:

Methods

The Oracle Video ActiveX Control provides the following methods:

Forward()	ImportStreamAs()	SetPos()
GetInfo()	Load()	SetVol()
GetPos()	Pause()	ShowInfoDialog()
GetStats()	Play()	ShowStatsDialog()
GetVol()	Resume()	Stop()
ImportFileAs()	Rewind()	Unload()

Note: The descriptions in this section use Visual Basic syntax. For examples that show how Oracle Power Objects can use the Oracle Video ActiveX Control, see the Oracle Power Objects sample code in Chapter 4, "Oracle Video ActiveX Control".

Forward()

Syntax: `Forward`

Forward the video to the end or the position set by the **PlayTo** property.

GetInfo()

Syntax:
```
GetInfo(streamName As String,
        url As String,
        title As String,
        fmtType As String,
        extType As String,
        createTimeStr As String,
        byteLength As Long,
        bitrate As Long,
        presRate As Long,
        capabilities As Long,
        milliseconds As Long,
        frameHeight As Integer,
        frameWidth As Integer,
        aspectRatio As Long,
        frameRate As Long,
        contentStatus As Integer,
        protocol As Integer)
```

This method returns information about the current media file.

Important: Make sure you declare all of the variables that you pass as parameters, and initialize all of the strings to null strings, before calling **GetInfo()**.

If the call to **GetInfo()** succeeds, it sets the variables passed as parameters to the indicated data and returns 1. Otherwise **GetInfo()** returns 0 and the parameters do not contain valid information about the stream. You should make sure you check the return value before relying on the data contained in the variables.

The parameters contain the following information after a successful **GetInfo()** call:

Parameter	Description
streamName	The unique name of the stream
url	The URL or asset cookie of the stream
title	The stream title or description (not unique)
extType	File extension; for example, **mpi**, **mpg**, or **osf**
fmtType	A readable string version of **extType**
createTimeStr	Time at which the stream was created
byteLength	Length of the stream in bytes
bitrate	Average bit rate over the life of the stream, in bits per second (bps)
presRate	Present bit rate
capabilities	Capabilities of the stream; this is a reserved parameter
milliseconds	Total length of video in milliseconds (ms); this is always 0 for unbounded streams
frameHeight	Default height of the video frame
frameWidth	Default width of the video frame
aspectRatio	Aspect ratio of the video
frameRate	Current frame rate of the video, expressed in frames per second
contentStatus	Indicates the status of the content stream. Possible values are: ■ 0: Unknown ■ 1: Stream is from a local file ■ 2: Stream is stored on disk on the server ■ 3: Stream is stored on Hierarchical Storage Manager (HSM) on the server ■ 4: Stream is a server feed ■ 5: Stream is unbounded (that is, a rolling feed with no determinate end) ■ 6: Stream is a wide network feed, such as multicast

Parameter	Description
protocol	Network protocol by which the stream is being delivered. ■ 0: Protocol is unknown ■ 1: Stream is a local file (that is, there is no network protocol) ■ 2: Delivery is through UDP ■ 3: Delivery is through TCP ■ 4: Delivery is through HTTP ■ 5: Delivery is through ATM

GetPos()

Syntax: `GetPos(posfmt as integer)`

Returns the current stream position. The **posfmt** parameter defines the format used to query the position. Possible values are:

Value	Description
1	Returns the current position in milliseconds
2	Returns the current position as the number of frames from the beginning

```
pos = ovcax1.GetPos(1)
```

Returns the current position. Returns 0 if:

- the call fails, or
- the video is at the beginning but hadn't been played (such as when first loaded)

GetStats()

Syntax:
```
GetStats(dropPkts as long,
         rcvdPkts as long,
         freeBytes as long,
         readBytes as long,
         streamState as long,
         streamSize as long,
         curPos as long,
         minPos as long,
         maxPos as long)
```

Returns statistics about the current stream.

Note: Make sure you declare all of the variables that you pass as parameters before calling **GetStats()**.

If the call to **GetStats()** succeeds, it sets the variables passed as parameters to the indicated data and returns 1. Otherwise **GetStats()** returns 0 and the parameters do not contain valid information about the stream. Make sure you check the return value before relying on the data contained in the variables. **GetStats()** returns 0 if no video is loaded.

The parameters contain the following information after a successful **GetStats()** call:

Parameter	Description
dropPkts	Indicates the number of packets dropped since stream creation
rcvdPkts	Indicates the number of packets received since stream creation
freeBytes	Indicates the number of free bytes in the local cache
readBytes	Indicates the number of bytes in the cache ready for decoding
streamState	Indicates the current stream state; valid values are described in the **State** property on page C-13
streamSize	Contains the overall size of the stream in milliseconds
curPos	Indicates the current stream position in milliseconds
minPos	Indicates the minimum stream position (same as **PlayFrom**) in milliseconds
maxPos	Indicates the maximum stream position (same as **PlayTo**) in milliseconds

GetVol()

Syntax: `GetVol() as integer`

Returns the current volume setting of the player as an integer from 0 to 100.

ImportFileAs()

Syntax: `ImportFileAs() as Boolean`

Presents a standard dialog window from which the user can pick a video stored on the local machine. When the user chooses a video and clicks the OK button, this method:

- Sets the **Mediafile** property to the name of the selected file
- Call **Load()** to establish a link between the Oracle Video Client software and the video file; this process does not take as long as preparing video from a server
- Returns 1

If the user clicks the **Cancel** button or if the call fails, **ImportFileAs()** returns 0.

This sample code could be associated with a button that automatically plays the video the user selected.

```
Private Sub Command1_Click()
  ovcax1.ImportFileAs
  ovcax1.Play
End Sub
```

ImportStreamAs()

Syntax: `ImportStreamAs(server_addr as String) as Boolean`

Presents a file dialog window from which the user can pick an play a video located on the Oracle Video Server specified by the **server_addr** parameter. You can pass a server name in the **server_addr** parameter or pass a null string (““). The null string indicates that this method should use the default server.

When the user chooses a video and clicks the OK button, this method:

- Sets the **Mediafile** property to the name of the selected file
- Call **Load()** to establish a link between the Oracle Video Client software and the selected file
- Returns 1

If the user clicks **Cancel** or if the call fails, **ImportStreamAs()** returns 0.

Load()

Syntax: `Load(media_url as String) as Boolean`

Loads the movie specified by the **media_url** parameter. If the **Load()** method is successful, the **Mediafile** property is set to **media_url**. **Load()** returns 0 if the call fails or if **media_url** is invalid or unspecified. Otherwise **Load()** returns 1.

Pause()

Syntax: `Pause`

Pauses playback, preserving the current position. You can only call this method while the player is in the play state. The current frame remains displayed.

Play()

Syntax: `Play`

Starts stream playback. If the properties **PlayFrom** and **PlayTo** are not specified, the playback starts at the beginning of the stream and continues until the end. Otherwise playback starts at the position specified by **PlayFrom** and finishes at the position set by **PlayTo**. See the descriptions of **PlayFrom** and **PlayTo** for proper format information.

You can only call this method while the player is stopped. To start playing while paused, call the **Resume()** method.

Resume()

Syntax: `Resume`

Resumes playback. This call can only be invoked from the paused state.

Rewind()

Syntax: `Rewind`

Rewinds the movie to the beginning or the position set by the **PlayFrom** property.

SetPos()

Syntax: `SetPos(posfmt as integer, streampos as long)`

Seeks the stream to the position specified by the **streampos** parameter. The **posfmt** parameter specified the format used to set the position. The possible values for **posfmt** are the same as those used for **GetPos()**.

SetVol()

Syntax: `SetVol(volume as integer)`

Sets the volume of the player as an integer from 0 (inaudible) to 100 (maximum).

ShowInfoDialog()

Syntax: `ShowInfoDialog`

This method displays a dialog box showing information about the current media file. See **GetInfo()** for a description of the various statistics displayed.

ShowStatsDialog()

Syntax: `ShowStatsDialog`

This method displays a dialog box showing the current stream statistics. See **GetStats()** for a description of the various statistics displayed.

Stop()

Syntax: `Stop`

Pauses playback, resetting the current position to the beginning of the stream. If a start position was specified by the **PlayFrom** property, it's erased and the start position returns to the beginning of the stream. This call can only be called from the playing state. The current frame remains displayed.

Unload()

Syntax: `Unload`

Frees all resources allocated within the call to the **Load()** method. If you call this method while a stream is playing, the video is stopped and unloaded, and the video rectangle is blanked.

Properties

In addition to the standard Visual Basic or Oracle Power Objects object properties, the Oracle Video ActiveX Control uses these properties to control aspects of the video, its playback, and its external appearance. All properties are writable, unless identified as “read-only”.

AutoStart	Loop	ShowControls
BorderStyle	Mediafile	ShowPositionAndStatus
EnableLeftClick	PlayFrom	State
EnablePopup	PlayTo	TimerFrequency
IsLoaded		

All of these properties except for those that are read-only can be set when embedding the Oracle Video ActiveX Control into an HTML document through the use of **<param>** tags with the **<object>** block (between the **<object>** and **</object>** tags). The **<param>** tags have three elements:

- The **<param>** tag itself
- The property name, such as **AutoStart** or **Mediafile**:

  `name="propName"`
- The property value; for string or numeric properties, this should just be the string or value, while for boolean properties, you should specify 1 for true or on and 0 for false or off

  `value="value"`

For example, to set the media file specifier, the **<param>** tag would look something like this:

```
<param name="Mediafile" value="vstcp://server:5000/video.mpi">
```

You can also set the values of the writable properties in the Designer component of application development tools like Visual Basic and Oracle Power Objects. The procedure for this depends on the tool being used.

AutoStart

Type: Boolean

Specifies whether stream playback begins automatically once the stream is finished loading. If true, video playback starts as soon as the load completes. The default value is 0.

BorderStyle

Type: Boolean

Specifies whether there is a border around the video screen. The border is one pixel wide. Possible values are:

- 0 specifies no border; this is the default
- 1 specifies that the border should be displayed

EnableLeftClick

Type: Boolean

Specifies whether a left click on the video screen toggles Play and Pause. If **EnableLeftClick** is 1, left clicking in the video screen has the same effect as clicking on the Play / Pause button. The default value is 1.

Note: If **EnableLeftClick** is 1, the control does not return the **LeftClick** event since it's handled by the video screen.

EnablePopup

Type: Boolean

If **EnablePopup** is 1, the pop-up menu appears when the user right-clicks the video area. The default value for **EnablePopup** is 1.

Note: If **EnablePopup** is 1, the control does not return the **RightClick** event since it's handled by the popup.

IsLoaded

Type: `Boolean`

A read-only property that indicates whether the Oracle Video ActiveX Control has a stream loaded.

Loop

Type: `Boolean`

Specifies whether the movie is to be rewound when the end or the position specified by the **PlayTo** property is reached. If **PlayFrom** is specified, the movie is rewinded at the **PlayFrom** position instead of the beginning.

Mediafile

Type: `String`

Contains a file on the local file system or the media file specifier of a file on the Oracle Video Server. See Appendix D, “The Media File” for the complete syntax for media file specifiers.

PlayFrom

Type: `String`

Indicates the position from which to start playback when Play, Rewind, or Loop actions are performed. You can specify the value for this property in one of three formats:

Format	Description
Literal	There are three literal values you can use for **PlayFrom**: ■ "beginning" indicates that the start of play should be at the beginning of the file or stream ■ "end" indicates that the start of play should be at the beginning of the file or stream. This may seem contradictory: how can playback begin at the end? The answer lies in the nature of real-time streaming video. The end of the stream doesn't mean the end of the movie, just the end of what has already been streamed. Basically it indicates that playback should resume where it ended. ■ "default" indicates play should start at the default starting point for the stream.
time	In the format "hh:mm:ss:cc". For example, "00:02:40:10" means to start the movie at the "second minute fortieth second ten hundredth of a second" from the beginning.
position	A single string value that indicates the position in milliseconds. For example, "10000" indicates that the position from which to begin is 10 seconds (10,000 milliseconds) from the beginning.

PlayTo

Type: `String`

Indicates the position where playback should be stopped when **Play** or **Forward** actions are performed. You can use the same formats for this as for **PlayFrom**.

ShowControls

Type: Boolean

Specifies whether the playback controls are visible or hidden. This is the master property. When **ShowControls** is 1, the controls selected by **ShowPositionAndStatus** are visible; when 0, all controls are hidden. The default value is 1.

ShowPositionAndStatus

Type: Boolean

Specifies whether the position indicator and status line are visible or hidden when the **ShowControls** property is 1. The default value is 1.

State

Type: Integer

A read-only property that indicates the state of the control. Possible values are:

State	Indicates the player...
0	Has no video loaded.
1	Has loaded a stream, but not locked resources for playback
2	Has loaded a stream and locked the resources necessary for playback
3	The player is currently playing a stream
4	The player is paused
5	The player has reached the end of the stream
6	The player has encountered an error

TimerFrequency

Type: Integer

Specifies the interval in milliseconds at which the position indicator (such as the time counter and seek slider) are updated in the Oracle Video ActiveX Control. Reasonable values are from 100 to 1000 milliseconds.

> **Note:** Ordinarily, do not set the interval to anything less than 100 milliseconds because the multiple screen repaints necessary to update the position informations can seriously degrade video rendering performance.

Events

Events are messages that the Oracle Video ActiveX Control sends to your application. Your application can then perform specific actions in response to these events. For example, the **Stopped** event is sent to your application when the video stops playing.

The following events are defined by the Oracle Video ActiveX Control:

Completed	PlayStarted	RightClick
LeftClick	Resumed	Stopped

Completed

Sent when the video stream reaches its end or the position indicated by the **PlayTo** property, if set.

In response to **Completed**, you might want to call **Load()** to prepare the next stream (if you know what it is) so that it's ready to play when the user wants it.

LeftClick

Sent when the user clicks the left mouse button while the cursor is over the video screen portion of the Oracle Video ActiveX Control. This is only sent when a stream is loaded.

> **Note:** This event is not sent when the **EnableLeftClick** property is 1, since the **LeftClick** event is handled by toggling Play and Pause.

PlayStarted

Sent when the video is started in response to a call to the **Play()** method.

> **Note:** When a paused video is resumed, the Oracle Video ActiveX Control does not send a corresponding **PlayStarted** event. Instead it sends **Resumed**.

Resumed

Sent when video playback is resumed in response to a call to the **Resume()** method.

RightClick

Sent when the user clicks the right mouse button while the cursor is over the video screen portion of the Oracle Video ActiveX Control. This is only sent when a stream is loaded.

> **Note:** This event is not sent when the **EnablePopup** property is 1, since the **RightClick** event is handled by opening the pop-up menu.

Stopped

Sent when the video is stopped, in response to a call to the **Stop()** method, or in response to a **Pause()** method.

In response to the **Stopped** event, you might want to prepare the next stream (if you know what it is) so that it's ready to play when the user wants it.

<Object> Attributes and Parameters

HTML defines a number of standard attributes that you can use in an **<object>** statement to specify the behavior of the browser and how the browser displays your embedded control. Only a subset of these standard attributes affect the Oracle Video ActiveX Control. These include:

ClassID	ID
Height	Width

The Oracle Video ActiveX Control itself recognizes a number of parameters that affect the behavior of the embedded control itself. These set the control's properties and use the same names as the properties. See "Properties" on page C-9 for more information on setting the control properties.

ClassID

ActiveX control class identifier. This should *always* be the same value:

```
CLSID:547A04EF-4F00-11D1-9559-0020AFF4F597
```

If you're creating your HTML page by hand, you'll have to type this value in manually. If you're using a composition tool such as Microsoft FrontPage or ActiveX Control Pad, this is usually inserted automatically when you place the control on the page.

You can find this value in the Windows registry by searching under HKEY_CLASSES_ROOT for **ovcax.ovcax**. This contains a single entry named **CurVer**, which points to the entry for the current version of the control. Find that entry, which in turn contains a single entry, the class ID.

Height

Indicates the height in pixels to be reserved to display the control.

ID

A string identifier used to refer to the embedded control. Use this when you have more than one control or object in your document or when you want to script an object.

Width

Indicates the width in pixels to be reserved to display the control.

D

The Media File

This appendix describes how to specify which media resource to play. The full description of the media resource you want to play, including transport protocol, server, and location, is referred to as the *media file specifier*. The media resource can be an MDS file or a logical content asset cookie.

Using the Oracle Video Web Plug-in, the **mediafile** attribute of the **<embed>** tag is a media file specifier. With the Oracle Video Java Library, the **Player**.**load()** method takes a media file specifier as a parameter. In the Oracle Video ActiveX Control, the **Mediafile** property contains the media file specifier of the currently loaded stream.

mediafile Syntax

mediafile = protocol:\|\|server:port\|media_resource

The media file specifier is similar to an URL: it specifies the protocol with which the resource should be transferred, the machine and port that contains the resource, and the location of the resource within the server's virtual directory structure. The format of the various parts of the media file specifier is:

- *protocol:* This determines the protocol used to transmit media data (not the protocol of the Media Net connection). If *protocol* is specified, three forward slashes must appear between *protocol*: and the MDS path and filename or the logical content asset cookie.

OVC supports two protocols:

- **vsudp**: This specifies the UDP protocol, which is the default protocol. You should generally use UDP since it's more efficient than TCP. UDP does not retransmit dropped packets. AVI and WAV formats are not designed to handle dropped data, so they must be rewrapped into OSF files to be sent using the UDP protocol.
- **vstcp**: This specifies the TCP protocol. TCP tracks and retransmits dropped data. As noted earlier, AVI and WAV formats don't handle dropped data, so they must be rewrapped into OSF or transmitted using TCP.

- *server:port* The server name and port number of the Media Net address server. If the server name and port number are not specified, the default Media Net address (configured through the Oracle Video Client Settings utility: choose **Start** | **Programs** | **Oracle Video Client** | **Oracle Video Client Settings**) is used.
- *media_resource* The MDS file or the logical content asset cookie you want to play from the Oracle Video Server.

Logical Content Asset Cookies

Logical content asset cookies lets you specify a logical content asset file, which allows you to group multiple chunks of content into a single logical entity.

> **Note:** Before you can use logical content asset cookies, the Oracle Video Server must be configured for logical content. See your system administrator or the *Oracle Video Server Administrator's Guide* for more information.

If there are spaces in your logical content asset cookie's name, use quotation marks to enable the system to see the words after the space. For example, suppose you have a cookie name like this:

```
vstcp://sunserver:5000/vscontsrv:oracle video
```

You must enclose this in quotes for the system to get the whole name:

```
"vstcp://sunserver:5000/vscontsrv:oracle video"
```

If you don't enclose this in quotes, the system sees only this:

```
vstcp://sunserver:5000/vscontsrv:oracle
```

mediafile Examples

Here are some typical examples of possible *mediafile* values. The first example of each type uses an MDS file, while the second example uses a logical content asset cookie.

In this example, **vstcp** specifies the TCP protocol, **sunserver:5000** specifies the server and port (temporarily overriding the OMN_ADDR environment variable), **/mds/video/oracle1.mpi** specifies the MDS file, and **vscontsrv:oracle1** specifies the logical content asset cookie.

```
vstcp://sunserver:5000/mds/video/oracle1.mpi
vstcp://sunserver:5000/vscontsrv:oracle1
```

In this example, **vstcp** specifies the TCP protocol, **sunserver** specifies the server and the default port 5000 (again, temporarily overriding the OMN_ADDR environment variable), **/mds/video/oracle1.mpi** specifies the MDS file, and **vscontsrv:oracle1** specifies the logical content asset cookie.

```
vstcp://sunserver/mds/video/oracle1.mpi
vstcp://sunserver/vscontsrv:oracle1
```

In this example, **vstcp** specifies the TCP protocol, **/mds/video/oracle1.mpi** specifies the MDS file, and **vscontsrv:oracle1** specifies the logical content asset cookie. The default server and port are used. Note that OMN_ADDR must be set for this to work. Since the protocol is specified, three forward slashes are required.

```
vstcp:///mds/video/oracle1.mpi
vstcp:///vscontsrv:oracle1
```

In this example, the default UDP protocol is used as well as the default server and port. Since the protocol is not specified, only one forward slash is required. Note that OMN_ADDR must be set for this to work.

```
/mds/video/oracle1.mpi
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W