SAS® 9.4 Integration Technologies: Windows Client Developer's Guide

SAS® Documentation
July 31, 2017

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2013. *SAS® 9.4 Integration Technologies: Windows Client Developer's Guide*. Cary, NC: SAS Institute Inc.

SAS® 9.4 Integration Technologies: Windows Client Developer's Guide

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2017

9.4-P5:itechwcdg

Contents

Chapter 1

Concepts

Developing Windows Clients

When developing Microsoft Windows clients, you interact with the SAS Integrated Object Model (IOM) using the Microsoft Component Object Model (COM). In all the leading programming language products under Windows, and in most Windows applications, COM is the predominant mechanism for software interoperability on the Windows platform.

For the benefit of Windows applications, SAS manifests its IOM as a COM component that uses the automation type system. Microsoft calls this type of COM component an ActiveX component or an OLE Automation server.

Interacting with SAS as an ActiveX component has the following benefits:

- SAS can be called from a wide variety of programming language environments such as Microsoft Visual Basic (including Visual Basic for Applications and VBScript), Microsoft Visual C++, Borland C++Builder, Visual Basic .NET, and Visual C# .NET, Perl, Borland Delphi, and others.

- SAS processing can be invoked from the macro language of many popular applications, including those in Microsoft Office.
- The programming language skills most commonly used to build solutions in the Windows environment can also be applied to developing solutions that involve SAS.
- Operating system-level security, configuration, and management are the same for SAS as for other applications and systems utilities.

In addition to these standard advantages for integration via COM, the SAS IOM offers a superior capability that is not commonly available to ActiveX components. This function, known as the IOM Bridge for COM, provides the ability to run the server on platforms other than Windows. Using this bridge, a Windows application can request SAS analytical processing for data on a UNIX or z/OS server and receive the results.

The exact interfacing technique used by Windows language products has evolved over the years. The initial approach that was documented in COM supported calls from Visual Basic by using an interface known as IDispatch. With IDispatch, calls into an interface go through a single method (IDispatch:Invoke), and then the appropriate implementation code is looked up at run time. This technique was compatible with early versions of Visual Basic, but was not optimal because of the amount of run-time interpretation that is involved in a method call. To improve performance, subsequent versions of Visual Basic and other languages can use v-table binding to call methods directly. Besides yielding better performance, v-table binding is also the most natural approach for COM calls from C++. The IOM implementation of ActiveX component interfaces uses the dual interface layout that provides both IDispatch and v-table binding. This dual interface gives the best performance from newer language implementations, but still supports the Dispatch technique for client languages (including VBScript and JScript) that use the older approach.

SAS®9 Integration Technologies includes a new client-side component called the object manager. The SAS 8 workspace manager is still supported, but it is recommended that you use the object manager interface in order to take advantage of the new features.

If you are using a SAS Metadata Server, then the object manager enables you to launch and manage objects on other SAS Metadata Servers, SAS Workspace Servers, SAS Stored Process Servers, and SAS OLAP Servers.

Client Requirements

The SAS Integration Technologies client for the Windows operating environment has the following software requirements:

- Windows XP, Windows Vista, Windows Server 2003, or Windows Server 2008. The client can connect to SAS using all three methods (COM, Distributed Component Object Model [DCOM], IOM Bridge for COM).
- If your client machines use the IOM Bridge for COM (a component of SAS Integration Technologies) to attach to an IOM server, then each client machine needs a valid TCP/IP configuration.
- If you are using the IOM Data Provider (a component of SAS Integration Technologies), then you need the Microsoft Data Access Components (MDAC) Version 2.1 and later on each client machine. MDAC is available from the Microsoft website.

Client Installation

Installation Methods

The SAS Integration Technologies client for the Windows operating environment can be installed in multiple ways:

- Install Base SAS for Windows. The SAS Integration Technologies client is installed with Base SAS software.
- Install an enterprise client (such as SAS Enterprise Guide). The setup program for the enterprise client component installs the SAS Integration Technologies client.
- Install the SAS Integration Technologies client by itself. The SAS Integration Technologies client is packaged into a single executable file that can be copied to Windows machines for easy installation of the SAS Integration Technologies client.

Note: If you are using the SAS Integration Technologies client with 64-bit SAS, then additional setup steps are required for IOM COM servers on 64-bit Windows. For details, see the SAS installation documentation.

Default Installation Directory

For 32–bit Windows, the default installation directory is **`C:\Program Files\SASHome\Integration Technologies`**.

For 64–bit Windows, by default SAS Integration Technologies is installed in both of the following directories:

- **`C:\Program Files\SASHome\Integration Technologies`**
- **`C:\Program Files\SASHome\x86\Integration Technologies`**

Note: If you had a previous release of SAS Integration Technologies, then the installation directory might be **`C:\Program Files\SAS\Shared Files\Integration Technologies`**.

Note: If a maintenance update has been applied for SAS Integration Technologies, then a backup copy of the installation is located within the **`C:\Program Files\SASHome\Integration Technologies\installs\`** directory structure.

Components of the SAS Integration Technologies Client

Regardless of which method is used to install the SAS Integration Technologies client, the core function is installed via the inttech.exe file. When this file executes, it unbundles and installs the following components:

- the Integration Technologies Configuration Utility (itconfig.exe)
- SAS Package Reader (SASspk.dll, with Help in sasspk.chm)
- IOM Bridge for COM (SASComb.dll)
- Object Manager (SASOMan.dll, help file in sasoman.chm)
- Workspace Manager (SASWMan.dll, help file in saswman.chm)

- SAS Stored Process Service (StoredProcessService.dll, help file in StoredProcessService.chm)
- SAS Logging Service (LoggingService.dll)
- SAS Metadata Service (SAS.Metadata.dll)
- SAS type libraries (sas.tlb with Help in SAS.chm, arbor.tlb, asp.tlb, gms.tlb, IMLPlusServer.tlb, LoggingService.tlb, mdx.tlb, mqx.tlb, ObjectSpawner.tlb, olap.tlb, omi.tlb, sascache.tlb, SASIOMCommon.tlb, sastableserver.tlb, stp.tlb, tst.tlb)
- an executable to register type libraries (RegTypeLib.exe)
- Scripto (Scripto.dll)
- the Encryption Algorithm Manager (tcpdeam.dll) and the SAS Proprietary Encryption Algorithm (tcpdencr.dll), which are used by the IOM spawner and the IOM Bridge for COM components for encrypting the communication across the network
- the Xerces library from Apache Software Foundation

Note: The SAS Integration Technologies Windows client includes software that was developed by the Apache Software Foundation. For more details, see the Apache website at www.apache.org.

Encryption Support

Windows clients (and Windows servers) that use strong encryption need additional support, which is supplied through SAS/SECURE software. SAS/SECURE software enables SAS Integration Technologies to use encryption algorithms that are available through the Microsoft Cryptographic Application Programming Interface (CryptoAPI).

If you have licensed SAS/SECURE software, you should install the SAS/SECURE client for Windows. You can install the SAS/SECURE client from the SAS Installation Kit.

The SAS/SECURE client installs the tcpdcapi.dll file that is necessary for the IOM Bridge for COM to use the CryptoAPI algorithms to communicate with the IOM server. The file is installed to the shared file area on the client.

For more information, see the online product overview for SAS/SECURE software at www.sas.com/products/secure on the SAS Products and Solutions website.

Windows Client Security

Security Contexts

Beginning in SAS®9, separate client and server security contexts within the SAS Workspace Server are not supported. All file access checks are now performed solely with the user ID under which the server was launched. The additional level of security checking, which was based on the client user ID, has been removed.

This change affects sites that were relying on two levels of access checking, one under the ServerUserID and an additional level under a ClientUserID that can be distinct. Before upgrading to SAS®9 from SAS 8, the following sites should review their security policies:

- sites that launch COM workspace servers by using the "This user" identity setting in the dcomcnfg utility
- sites that use COM+ pooling
- sites that use web applications configurations where the site administration has control of the client security settings

IOM Bridge for COM Security

Specifying the ServerUserID

When using the IOM Bridge for COM, the spawner uses the client-provided login name and password to launch the server. The server receives its ServerUserID (OS process user ID) from the login name that is provided by the client. The operating system then performs access checks by using the ServerUserID.

Specifying Encryption

The IOM Bridge for COM supports encryption of network traffic between the client and the IOM server. Weak encryption support (through the SASPROPRIETARY encryption algorithm) is available with Base SAS software. Stronger encryption requires a license to SAS/SECURE on the server machine.

On the Windows platform, the SAS/SECURE license allows the encryption algorithms that are available through the CryptoAPI to be used. On other platforms, encryption algorithms are included in the SAS/SECURE software.

Note: The SAS/SECURE Client for Windows must be installed on the Windows client machines in order to use the CryptoAPI. For more information, see "Client Installation" on page 3.

When you define a ServerDef, you can use the BridgeEncryptionAlgorithm and BridgeEncryptionLevel attributes to specify the encryption algorithm and the types of information that are encrypted. For more information, see the Object Manager package documentation (sasoman.chm.)

COM and DCOM Security

Specifying the ServerUserID

On Windows NT, the COM Service Control Manager (SCM) is responsible for launching COM and DCOM processes. The SCM reads values from the Windows registry to determine which identity to use when launching a process. These registry settings are configured by using the Windows dcomcnfg utility on the server.

The ServerUserID is set under the **Identity** tab for the properties of the SAS.Workspace application.

You can choose one of the following options for the identity to use to run the IOM server (the ServerUserID):

Interactive User
: allows the interactive user to kill the SAS server process through the task manager, because the interactive user owns the process. This is a security risk for the interactive user, because SAS will be running under this user's ID. Someone must be logged on, or the SAS process will not run.

Note: For SAS®9 and later, there is no longer an additional level of security checking that is based on the client user ID. All file access checks are performed based solely on the user ID of the interactive user. This setting is not recommended for production environments.

Launching User
: specifies the default option and is more secure than the other two identity settings. This option ensures that the ServerUserID is the same as the client who created it. This setting provides a single-signon environment for launching the server. However, it does require the use of network authentication to set up the identity of the server. The Windows NT LAN Manager (NTLM) network authentication mechanism, which is the default if any Windows NT4 machines are involved or if Windows 2000 Kerberos is not set up, cannot pass the client identity across more than one machine boundary. If you configure IOM servers to use "launching user" for DCOM connections, then the servers cannot access network files unless special Windows registry settings are adjusted on the file server. The servers also cannot deliver events back to a DCOM client unless one of the following is true:

- The client permits everyone to access its COM interfaces by setting the authentication level to Everyone.
- The client turns off authentication and authorization by setting the authentication level to None.

This user
: enables you to configure a specific user name and password to run the IOM server. This option has the same security considerations as the interactive user, but you do not have to worry about always having someone logged on. Also, the identity does not change based on who is logged on.

Note: For SAS®9 and later, there is no longer an additional level of security checking based on the client user ID. All file access checks are performed based solely on the user ID that you specify for **This user**.

Specifying Encryption

For DCOM, encryption is enabled by using an authentication level of **Packet Privacy**.

Authentication and Impersonation Levels for COM and DCOM Security

Authentication levels must be set on both the client and server machines. The stronger of the two authentication levels is then selected. Here are the available authentication levels:

None
: The client is not authenticated. It is not possible for the server to determine the identity of the caller.

Connect
: The client is authenticated when the connection is first established and never again.

Call
: The client is authenticated each time a method call is made.

Packet
: Client authentication occurs for each packet. There might be multiple network packets used for a given call.

PacketIntegrity
: Each packet is authenticated and verified to have the same content as when it was sent.

PacketPrivacy
: All data is encrypted across the wire. Note that for local COM, the authentication level appears to be **PacketPrivacy**, but no encryption actually occurs on the local machine.

For more information about authentication levels, see the Microsoft documentation.

The impersonation level that is set on the client machine determines the impersonation level to use for the connection. The impersonation level that is set on the server machine is not used. Here are the available impersonation levels:

Anonymous
: Impersonation is not allowed.

Identify
: The server is allowed to know who is calling but cannot make calls by using the credentials of the caller.

Impersonate
: The server can access resources by using the security credentials of the caller. The server cannot pass on the credentials. The IOM server attempts to impersonate the caller. This is the recommended setting.

Delegate
: The server can access resources by using the security credentials of the caller and by passing on those credentials to other servers. SAS software does not currently support this option.

Programming Examples

This Visual Basic example retrieves the ClientUserID and the ServerUserID from the IOM server.

```
Public Sub sectest()
' This example prints both the client user ID and the server user ID.
' In SAS 9, the ServerUserID and the ClientUserID will always be the same.
' Create a local COM connection.
Dim sinfo As String
Dim swinfo() As String
Dim hwinfo() As String
Dim obWSMgr As New SASWorkspaceManager.WorkspaceManager
Set obSAS = obWSMgr.Workspaces.CreateWorkspaceByServer(
"MyServer", VisibilityNone, Nothing, "", "", sinfo)
' Get the host properties.
obSAS.Utilities.HostSystem.GetInfo swinfo, hwinfo
Debug.Print "ServerUserID: " & swinfo(
SAS.HostSystemSoftwareInfoIndexServerUserID)
Debug.Print "ClientUserID: " & swinfo(
SAS.HostSystemSoftwareInfoIndexClientUserID)
obSAS.Close
End Sub
```

COM Security Considerations for Client Applications

Here are some additional points to consider when developing client applications:

- Always test your application and configuration before making sensitive information available to ensure that people who are not authorized cannot see the data.

- Security settings and performance are inversely related. In general, the stronger the security, the slower things run. Security settings are highly configurable to allow the administrator to optimize performance for the required level of security.
- No system is completely secure, even at the strongest security settings.
- For maximum security when using the IOM Bridge for COM, use a BridgeEncryptionLevel of **All** and a strong BridgeEncryptionAlgorithm, such as **RC4**. For maximum performance, use a BridgeEncryptionLevel of **None**. (In this case, the setting for the BridgeEncryptionAlgorithm is ignored.)
- In general, for maximum security with DCOM, use an impersonation level of **Impersonate** and an authentication level of **Packet Privacy**.

  In SAS®9 and later, impersonation is not applied to workspace servers. For other IOM servers, an impersonation level of **Impersonate** is required.

  In SAS®9 and later, the use of connectionless transports (mainly UDP) can cause difficulty with configuring and debugging. If your system (particularly Windows NT4) still uses a connectionless transport, then you might avoid complications by naming a connection-oriented transport (typically TCP) as the primary default.

COM Security Settings for Threaded Multi-User IOM Servers

In SAS®9, SAS Integration Technologies provides three new threaded multi-user IOM servers. Windows Component Services has an entry for each application as follows:

SASMDX.Server
: SAS OLAP Server

SASOMI.OMI
: SAS Metadata Server

StoredProcessServer.StoredProcessServer
: SAS Stored Process Server, which provides interfaces to run user-written SAS programs (stored processes) to produce HTML output

If you do not specify the NOSECURITY object server parameter, then clients are authenticated when they connect to the server.

You can set DCOM security settings for each type of server individually. Use the Windows Component Services utility to specify security settings. In the Component Services utility, after you view the properties of a server, several tabs provide controls to customize the server's security.

Identity
: controls the ServerUserID for COM launches. If you launch the server via COM (rather than via the object spawner), then the first client that connects must be a Windows client. For multi-user servers, select **This user** and specify a user ID and password under which the server will run. The server runs in its own logon session. Therefore, interactive logon and logoff activity on the same machine does not affect it. COM does not start the server with any environment variables and does not set a home directory. You should edit the SAS config file to remove environment variable references and to specify the "-sasinitialfolder" that you need at start-up.

General
: controls the minimum client authentication level that the server accepts (which is also the level that the server uses on any outward calls). **Connect** (the default) is the minimum setting for the Authentication Level. Every COM client must also do the following:

- define an authentication level of **Connect** or higher
- set an impersonation level of **Impersonate**

These client settings can be specified by the client program on each calling interface or through one of two defaulting mechanisms:

- a call to CoInitializeSecurity() in the client program
- via the machine-wide default settings in COM security configuration

The ideal client program installs and uses an AppID of its own. However, some commonly used development languages, such as Visual Basic, do not provide an easy means to install and use an AppID.

Security
: controls access, configuration, and launch permissions. These permissions can either be determined from machine-wide defaults or set up specifically for the particular IOM server application. Ensure that System is included in any of these permissions that you customize. You can use the access permissions editor to create a standard access control list to indicate who can use the server.

Location
: indicates that the application should be run on this computer.

Endpoints
: enables you to select the most used protocol. It is recommended that you use connection-oriented protocols such as TCP.

Windows Client Logging

Overview of Logging for Windows Clients

Logging for SAS Integration Technologies Windows clients is implemented by using the log4net framework from Apache. The log4net framework enables you to manage the logging for all of your Windows clients in a single logging configuration. The log4net configuration is stored in an XML file, which can be modified by using the Integration Technologies Configuration utility (ITConfig).

The log4net configuration consists of appenders and loggers. Appenders specify how the logging information is written to the log. Loggers associate a level in the object hierarchy with one or more appenders, and set the level of detail that is included in the log.

For more information about the log4net framework, see the Apache log4net site at http://logging.apache.org/log4net/.

Editing Your Logging Configuration

The logging configuration for Windows clients is stored in a file named log4netConfig.xml. If you configure logging for specific users, then the file is located the following path:

```
\Documents and Settings\user-name\Application Data\SAS\WindowsClientLogging
```

If you use a single configuration for all users, then the file is located in the following path:

```
\Documents and Settings\All Users\Application Data\SAS\WindowsClientLogging
```

To edit your configuration, perform the following steps:

1. Start the Integration Technologies Configuration utility by selecting **Start** ⇨ **Programs** ⇨ **SAS** ⇨ **Utilities** ⇨ **Integration Technologies Configuration**.
2. Select **Configure Windows Client Logging**, and then click **Next**.
3. Select whether you want to configure logging for the current user (**Current user only**), or for all users on this machine (**All users of this machine**). Click **Next**.
4. Select one of the following options:

Edit the current logging configuration file	specifies that you want to edit the current configuration.
Load default settings	specifies that you want to edit a new configuration based on the default settings.
Enable logging	if your configuration was previously disabled, then this option re-enables the current logging configuration and enables you to edit it.

   Click **Next**.
5. In the Configure Logging Appenders window, create the logging appenders that you want to use. For each appender, specify the following information:

Name	specifies the name of the appender.
Layout	specifies a formatting string that controls the output.
Type	specifies the type of appender. The appender type determines where the output is sent. For example, EventLogAppender sends output to the Windows Event Log. FileAppender writes output to a file.

   You can also click **Filters** to define filters for the appender. Filters enable you to send only the output that matches certain criteria. At the bottom of the list of filters, you must add a DenyAllFilter filter to remove the output that does not match the other filter.

   Depending on the appender type that you specify, additional information might be required. For more information about the information that you can specify, see the Integration Technologies Configuration Help.

   When you are finished defining appenders, click **Next**.
6. In the Configure Loggers window, create the loggers that you want to use. For each logger, the name specifies the logging namespace for the logger. You can select the standard SAS namespaces from the drop-down list or specify a custom namespace. The root logger specifies the default logging settings for Windows clients. For each logger, specify the level of logging detail and the appenders that are associated with the logger.

   If you want a logger to exclude its contents from any parent loggers, then select **Enable Appender Additivity**. For example, if you create a logger for SAS.BI and a logger for SAS.BI.Metadata, then select **Enable Appender Additivity** for the SAS.BI.Metadata logger if you do not want its content to also appear in the SAS.BI logger.

   For more information about specifying loggers, see the Integration Technologies Configuration Help.

When you are finished defining loggers, click **Save** to save your configuration file.

Disabling Windows Client Logging

To disable logging for SAS Integration Technologies Windows clients, perform the following steps:

1. Start the Integration Technologies Configuration utility by selecting **Start** ⇨ **Programs** ⇨ **SAS** ⇨ **Utilities** ⇨ **Integration Technologies Configuration**.
2. Select **Configure Windows Client Logging**, and then click **Next**.
3. Select whether you want to configure logging for the current user (**Current user only**), or for all users on this machine (**All users of this machine**). Click **Next**.
4. Select **Disable logging**, and then click **Finish**.dd

Chapter 2

Selecting a Windows Programming Language

Selecting a Windows Programming Language

Any language that supports calling ActiveX components, also known as Object Linking and Embedding (OLE) Automation servers, should be able to make calls to IOM interfaces. That is, almost every programming language product that is available on the Windows platform can use the SAS IOM interfaces.

Microsoft designed the ActiveX components technology with a heavy bias toward meeting the needs of Visual Basic. Much of the technology is effectively a part of the Visual Basic run-time environment. In its own implementations of ActiveX components, such as in the Microsoft Office Suite, Microsoft has documented the interfaces in terms of the Visual Basic language.

Based on this convention and on the wide use of Visual Basic as a Windows programming language, we have documented the SAS IOM interfaces in terms of Visual Basic language syntax and conventions.

The .NET run time, with its family of languages including C# and VB.NET, represents the latest direction in Windows programming. ASP.NET is now the environment of choice for Windows web applications. The .NET environment also supports traditional desktop graphical user interface (GUI) clients similar to those that were developed with Visual Basic forms in Visual Basic 6 (VB6). IOM integrates fully with .NET through the use of COM Interoperability. For more information, see “Programming in the .NET Environment ” on page 23 .

Programming with Visual Basic

Referencing the Type Library

ActiveX components can contain numerous classes, each with one or more programming interfaces. These interfaces can have methods and properties. The components can also have many enumeration constants, which are symbolic names for constants that are passed or returned over the interface. For name scoping and management, each application defines its own group of these definitions into a type library. The type library is used by programming languages to check the correctness of calls to the component and is used by COM when it creates the data packet. The data packet conveys a method call from one Windows process to another and is called marshaling.

You must reference at least two type libraries to access IOM servers. One is for Base SAS software and the other is for the object manager. These type libraries are installed when you install Base SAS or when you install the SAS Integration Technologies client. However, before you can write Visual Basic code to call the Base SAS IOM interfaces, you must reference these type libraries from within your Visual Basic project.

To reference these type libraries from Visual Basic, perform the following steps:

1. On the Visual Basic menu bar, select the **References** menu item. Depending on the version of Visual Basic that you are running, this item is either under the **Project** (VB6) or under the **Tools** (Visual Basic for Applications [VBA]) menu.

   The references dialog box lists every registered type library on the system. This list is divided into two groups. The type libraries that are already referenced by the project are listed first. All the remaining registered system type libraries are listed after that in alphabetical order. The first group shows the list that can be referenced in your program. The second group helps you find additional libraries to add to the first group.

2. Find the type library labeled SAS: Integrated Object Model (IOM), and select the check box to add the reference to your project. (Simply selecting the line is not sufficient. You have to select the check box.)

3. Find the type library labeled SASObjectManager, and select its check box to add this reference to your project.

4. Click **OK** to activate these changes.

After referencing these libraries, you can use them immediately. Also, if you reopen the references dialog box, then you see that these libraries have been moved up to the top of the list with the other libraries that are referenced.

Each type library has a name, known as the library name. The library name is used in programming to qualify (or scope) all names (such as components, methods, constants, and so on) that are used within it. This name is not necessarily the same as the name of the file that contains the type library. Visual Basic looks up the names in your program by going through the referenced type libraries in the order in which they are listed in the references dialog box. If two type libraries contain the same name, then Visual Basic uses the definition from the first library in the list, unless the name is qualified with a library name.

The Base SAS IOM library name is "sas". You can use the name "sas" to qualify any identifier defined in that library. For example, the Base SAS library has a FormatService component. If another library in your application has its own identifier with the same name, then you reference the SAS component by using sas.FormatService.

The Visual Basic Development Environment

After the type library is referenced, its definitions become available to the Visual Basic Object Browser. From the Visual Basic development environment, the Object Browser component is typically available via a toolbar button, a menu selection, or the F2 key.

The Object Browser can show all referenced type libraries at one time or it can focus on any one of them. To focus on the Base SAS IOM library, select **SAS** from the drop-down list in the upper left. The left panel of the Object Browser will then show only the component classes, interfaces, and enumerations for SAS components. If you select a component (such as FileService) in the left panel, then its methods and properties are shown in the right panel. When you select an item in the right pane, more complete information (such as the parameters to pass to a method and a brief description of the method) is shown at the bottom.

Referencing a type library also activates Visual Basic IntelliSense™ for all programming language names in the library. As you begin to enter a name, like SAS.FileService, the Visual Basic editor shows you the list of possible names. As you code a method call, Visual Basic shows each parameter that you need to provide.

For VB6 and later or Microsoft Office 2000 VBA or later, the Object Browser and the Visual Basic editor are connected to the Base IOM interface Help file. In the Object Browser, if you click the ? button or press the F1 function key, you will see the Help page for the selected item. In the editor, if you press F1 with your cursor on a name, then the Help page for that name appears.

Working with Object Variables and Creating a Workspace

Your Visual Basic program's interaction with SAS begins by creating a SAS.Workspace object. This object represents a session with SAS and is functionally equivalent to a SAS Display Manager session or the execution of Base SAS software in a batch job. Using a workspace from Visual Basic requires that you declare an object variable in order to reference it. Here is an example declaration:

```
Dim obSAS as SAS.Workspace
```

This statement only declares a variable to reference a workspace. It does not create a workspace or assign a workspace to the variable. For creating workspaces, the SAS

Integration Technologies client provides the object manager. The object manager is an ActiveX component that can be used to create SAS workspaces, either locally or on any server that runs the SAS Integration Technologies product.

If the server is remote, then you can create a workspace by using either the machine name of the server (such as unx03.abccorp.com) or a logical name (such as FinanceDept) that is given to you by your system administrator. For more information about creating workspaces on remote servers, see "Using the SAS Object Manager " on page 44.

If SAS is installed on your local PC and you want to create a workspace that runs locally, then it is very easy to use the object manager to create the workspace and assign it to your object variable. The following example shows how to create a workspace and assign it to your object variable:

```
Dim obObjectFactory As New SASObjectManager.ObjectFactory
Dim obSAS As SAS.Workspace
Set obSAS = obObjectFactory.CreateObjectByServer(
"", True, Nothing, "", "")
```

By using the keyword **new**, you instruct COM to create a SAS Object Factory and assign a reference to it when the object variable (obObjectFactory in this case) is first used.

After you create the instance of the object factory, you can use it to create a workspace (SAS session) on the local machine. The object factory has a CreateObjectByServer method for creating a new workspace.

The first parameter to this method is the name of your choice. The name can be useful when your program is creating several objects and must later distinguish among them. The next parameter indicates whether to create the workspace synchronously. If this parameter equals true, then CreateObjectByServer does not return until a connection is made. If this parameter equals false, then the caller must get the created workspace from the ObjectKeeper.

The next parameter is an object to identify the desired server for the workspace. Because we are creating a workspace locally, we pass the Nothing (a Visual Basic keyword) to indicate that there is no server object. The next two strings are for a user ID and password. Again, we do not need these to create a local workspace. This method returns a reference to the newly created workspace.

To get error information about the connection, you can use the following code:

```
Dim obSAS As SAS.Workspace
Dim obObjectFactory As New SASObjectManager.ObjectFactory
obObjectFactory.LogEnabled = True
Set obSAS = obObjectFactory.CreateObjectByServer("", True, Nothing, "", "")
Debug.Print obSAS.Utilities.HostSystem.DNSName
Debug.Print "Log:"  obObjectFactory.GetCreationLog(True, True)
```

For more information about logging errors, see "SAS Object Manager Error Reporting " on page 50.

Note: Whenever you assign a reference to an object variable, the assignment uses the keyword **Set**. If you do not use **Set**, then Visual Basic attempts to find a default property of the object variable and set that default property.

You can prove that the previous example program code works by instructing the server to print its Internet Domain Name System (DNS) name. The following program creates a SAS workspace on the local computer, prints the DNS name, and then closes the newly created workspace.

```
Dim obObjectFactory As New SASObjectManager.ObjectFactory
```

```
Dim obSAS As SAS.Workspace
Set obSAS= obObjectFactory.CreateObjectByServer(
"", True, Nothing, "", "")
Debug.print obSAS.Utilities.HostSystem.DNSName
ObSAS.Close
```

As you use other features of SAS, you will need additional object variables to hold references to objects created within your SAS workspace. These are declared similarly. For example, if you want to assign SAS FILEREFs, then you might want an object variable to reference your workspace's FileService. Here is the declaration for this example:

```
Dim obFS as SAS.FileService
```

Such declarations never use the keyword **new** because doing so asks COM to create the object. Object variables for objects within the workspace always reference objects that are created by the workspace, not COM. In fact, when you use **new** in a declaration, Visual Basic IntelliSense provides a much shorter list of possibilities because it lists only those classes of objects that COM knows how to create.

The following statement uses the obFS variable to hold a reference to the FileService for the workspace that is referenced by obSAS:

```
Set obFS = obSAS.FileService
```

The assignment of an object variable must use the **Set** keyword.

Basic Method Calls

After you create an object variable and set it with an object reference, you can make calls against the object and access the object's properties.

The following example assigns a SAS libref by using the IOM DataService:

```
' Create a workspace on the local machine using the SAS Object Manager
Dim obObjectFactory As New SASObjectManager.ObjectFactory
Dim obSAS As SAS.Workspace
Set obSAS = obObjectFactory.CreateObjectByServer(
"My workspace", Nothing, "", "")
' Get a reference to the workspace's DataService.
Dim obDS as SAS.DataService
Set obDS = obSAS.DataService
' Assign a libref named "mysaslib" within the new workspace
' (This is an example of a method call that returns a value)
' Note: you must have a "c:\mysaslib" directory for this to work
Dim obLibref as SAS.Libref
Set obLibref = obDS.AssignLibref(
"mysaslib", "", "c:\mysaslib", "")
' Should print "mysaslib"
Debug.Print obLibref.Name
' De-assign the libref; this is an example of a method call that does
' not return a value
obDS.DeassignLibref obLibref.name
' Close the workspace; this is another example of a method call that
' does not return a value.
ObSAS.Close
```

This example illustrates three method calls and two property accesses.

In Visual Basic, method calls that use a return value are coded differently from those that do not. The AssignLibref call returns a reference to a new Libref object. Because the method returns a value that is being used in this call, the method's arguments must be enclosed in parentheses. The calls to DeassignLibref and Close, on the other hand, do not return a value. (Neither of these methods can return a value.) Thus, no parentheses are used following the method name, even for DeassignLibref, which passes a parameter in the method call.

When passing parameters, you must first understand whether the parameter is for input or output (also known as ByVal and ByRef in Visual Basic). Unfortunately, the Visual Basic Object Browser does not provide this information. If you are not sure which type of parameter is required, see the Help or documentation.

Input parameters can be passed by a constant value (such as **`mysaslib`** for a string parameter or 0 for a numeric parameter) or an expression. They can also be passed using a variable whose value is taken as input to the method. This technique is used in the DeassignLibref call in the example. Output parameters require the use of variables that will be updated at the time the call returns.

Understanding the data type of the parameter is just as important as understanding the direction. The data type is listed in the Object Browser. If the provided constant, variable, or expression has the same type as the parameter, then no conversion is necessary. Otherwise, Visual Basic might try to convert the value. If you do pass a different type than the method actually defines for that parameter, ensure that you understand Visual Basic's data conversion rules.

The parameters in IOM methods have a range of data types. Many are standard types such as String, Long, Short, Boolean, and Date. Others are object types defined by a type library (such as SAS.Libref in the previous example). Most parameters are designed to accept or return a single type of object. In these cases, it is best to use an object variable of that specific type. However, you can use variables that are declared with the generic object type Object. In some cases, methods can return more than one type of object, depending on the situation. For example, the GetApplication method for a SAS workspace might return different types of objects (based on the application name that was requested). It is declared to return the generic object type.

When a parameter can take some fixed set of integer values, its declaration uses an enumeration that is defined in the type library. For example, the Fileref:OpenBinaryStream method takes a parameter that indicates the open mode of the stream. The type of this parameter is SAS.StreamOpenMode. In the current version of Visual Basic, you cannot declare a variable of this type. You must use the type Long instead. The declaration is still important because it specifies the set of possible values and provides a constant for each. For example, if you want to open a stream for writing, then you would pass a constant of SAS.StreamOpenModeForWriting.

Some Visual Basic procedures and subroutines take optional parameters. When calling such routines, you often pass the parameters by specifying their name (such as **`color:="Red"`**). This feature of Visual Basic is not used by SAS IOM methods because it does not work naturally with other types of client programming environments, including Visual C++. Simply list your parameter values in order without showing their names.

Many objects have properties that can be obtained simply by naming the property. For example, libref objects have a Name property, which is used in the previous example. Some properties are read-only, but others can also be set. Such properties can be set by using a normal assignment statement. Keep in mind, however, that if the property type is an object type, then you will need to use the Set keyword with your assignment statement (as described in the previous section).

In order to keep the number of function calls to a minimum, many IOM methods have a large number of parameters and accept arrays for those parameters. Visual Basic IntelliSense technology is very helpful for keeping track of parameters as you code them.

Passing Arrays in IOM Method Calls

Overview of Arrays in IOM Calls

When SAS generates a listing, it might contain thousands of lines. In order to improve performance, IOM methods make heavy use of arrays. By returning an array of lines, the FlushListLines() method can potentially return all LIST output in one call. Similar situations occur in many other places in IOM, including reading and writing files, getting and setting options, and listing SAS librefs and filerefs.

Visual Basic arrays can be either fixed-size or dynamic. An array dimensioned with a size is fixed-size. An array dimensioned with no size (empty parentheses) or declared with the Redim statement is dynamic. For array output parameters, you must use dynamic array variables because the size that SAS will return is not known when the array variable is declared.

The following example lists all librefs in the workspace:

```
' Create a workspace on the local machine using the SAS Object Manager
Dim obObjectFactory As New SASObjectManager.ObjectFactory
Dim obSAS As SAS.Workspace
Set obSAS = obObjectFactory.CreateObjectByServer(
"My workspace", True, Nothing, "", "")
Dim obDS as SAS.DataService
Dim vName as variant
' Declare a dynamic array of strings to hold libnames
Dim arLibnames() as string
Set obDS = obSAS.DataService
' Pass the dynamic array variable to "ListLibrefs"; upon return,
' the array variable will be filled in with an array of strings,
' one element for each libref in the workspace
obDS.ListLibrefs arLibnames
' Print each name in the returned array
For Each vName in arLibnames
debug.print vName
Next vName
' Print the size of the array
debug.print "Number of librefs was: "  Ubound(arLibnames)+1
ObSAS.Close
```

In the object browser, you can tell that the **`names`** parameter for ListLibrefs is an array of strings because it is shown as follows:

```
names() as string
```

The empty parentheses indicate an array. A few of the array parameters in IOM require a two-dimensional array. The Object Browser does not distinguish the number of dimensions. You must consult the class documentation to determine the number of dimensions in an array.

After the ListLibrefs call returns, there will be one array element for each libref assigned in the workspace. The For Each statement can be used to iterate through these elements. You must use Variant as the type of the loop control variable.

In many cases, you need to know the size of the returned array. Visual Basic allows arrays to be created with a particular lower and upper bound in each dimension. In order to allow better compatibility with other client programming languages, all IOM calls require that the lower bound be zero. Input arrays are not accepted if their lower bound is not zero. Output arrays are always returned with a lower bound of zero.

The size of the array can be determined by checking its upper bound. For a one-dimensional array, you can get the upper bound by passing the array name to the Ubound function. Because the array is zero-based and Ubound returns the number of the last element, you must add one to get the size of the array.

You can get the Ubound for each dimension of a two-dimensional array by passing a dimension index. The following example illustrates this technique:

```
Dim table()
Redim table(5,2) ' 6 rows (0 through 5) and 3 columns (0 through 2)
debug.print "Number of rows: "  Ubound(table, 1)+1
debug.print "Number of columns: "  Ubound(table, 2)+1
```

When passing input arrays, you can pass a fixed size array if you know the size in advance. The following code provides an example:

```
' Create a workspace on the local machine using the SAS Object Manager
Dim obObjectFactory As New SASObjectManager.ObjectFactory
Dim obSAS As SAS.Workspace
Set obSAS = obObjectFactory.CreateObjectByServer(
"My workspace", True, Nothing, "", "")
Dim obLS as SAS.LanguageService
' Declare a fixed-size array of strings to hold input statements
Dim arSrc(2) as string
' Declare a dynamic array of strings to hold the list output
Dim arList() as string
' These arrays will return line types and carriage control
Dim arCC() as SAS.LanguageServiceCarriageControl
Dim arLT() as SAS.LanguageServiceLineType
Dim vOutLine as variant
arSrc(0) = "data a; x=1; y=2;"
arSrc(1) = "proc print;"
arSrc(2) = "run;"
Set obLS = obSAS.LanguageService
ObLS.SubmitLines arSrc
' Get up to 1000 lines of output
ObLS.FlushListLines 1000, arCC, arLT, arList
' Print each name in the returned array
For Each vOutLine in arList
debug.print vOutLine
Next vOutLine
ObSAS.Close
```

The number of input statements are known in advance, so the arSrc array variable can be declared as a fixed size. (A dynamic array could also be used.)

For some methods, you might want to pass an input array with no elements. Unfortunately, Visual Basic does not have syntax for creating either a fixed-size array or a dynamic array with zero elements. To work around this deficiency in Visual Basic, IOM methods enable you to pass an uninitialized dynamic array variable, which is the same as an array with no elements.

Visual Basic does properly handle output arrays with no elements. When an array has no elements, the Ubound() function returns `-1` and the For Each statement does not execute the body of the loop.

Using Enumeration Types in Visual Basic Programs

Older versions of Visual Basic (such as VBA in Microsoft Office 97) do not support defining variables as enumeration types. For these older versions, you must define variables as type Long. From the third example in the previous section, consider the statement:

```
Dim arCC() as long, arLT() as long
```

Here, the variables arCC and arLT are being used as arrays of LanguageServiceCarriageControl and LanguageServiceLineType, respectively. However, because this program was written to execute in a VBA (Microsoft Office 97) environment, the variables are defined as arrays of long integers.

For newer versions (such as VB6), you need to use enumeration types. To execute this program in a VB6 environment, the line would be changed to the following:

```
Dim arCC() as LanguageServiceCarriageControl
Dim arLT() as LanguageServiceLineType
```

Note that the enumeration constants themselves are available for use in both environments. Given that arCC and arLT are defined appropriately for the environment, the following statements are valid in both the old and new versions:

```
arCC(0) = LanguageServiceCarriageControlNormal
arLT(0) = LanguageServiceLineTypeSource
```

Object Lifetime

Use the Workspace.Close method to close a workspace when you are finished with it. This method also deletes all of the objects within the workspace. For some types of objects, such as streams, you might finish using the object long before you finish using the workspace. These objects typically have their own Close method that removes the object and performs other termination processing (such as closing the file against which the stream is open). The lifetime of objects that do not have a Close (or similar) method is managed by the workspace in which the object is created.

In Visual Basic, you can release the reference held by an object variable by assigning the keyword Nothing to that variable. The syntax is as follows:

```
ObStream = Nothing
```

By using this technique, you can make Visual Basic release its references to an object. Doing so does not, however, delete the object because the workspace still references the object.

As a special case, if your program releases its references to all of the objects on a SAS server, then the server deletes the workspace and all of the objects in it. This behavior is implemented to prevent excess SAS processes from being left on a machine when clients fail to close their workspaces properly. If a client program terminates abruptly or is killed by the user from the Task Manager, then COM notifies SAS at a later time. In current versions of COM, this notification takes approximately six minutes.

Exceptions

Method calls sometimes fail. When they do, Visual Basic raises an error condition. If you want to write code that responds to error conditions, then you should code **`On Error Goto Next`** after each call. If you want to centralize the code that responds to error conditions, then code an **`On Error Goto`** with a label and place your error handling code at the label.

When an error occurs, COM and Visual Basic store the error in the Err predefined variable of the type VBA.ErrObject. The two most important fields in this variable are Err.Number and Err.Description. Err.Number can be used by your program logic to determine what type of error occurred.

Err.Description provides a text description of the error. It can also provide important details about the error. For example, in the case of an error in opening a file, the description provides the name of the file that could not be opened. SAS®9 servers return the error description in XML format. This format allows multiple messages from the server to be packed into one string. The format is as follows:

```
<Exceptions>
<Exception>
<SASMessage>ERROR: First message/SASMessage>
</Exception>
<Exception>
<SASMessage>ERROR: Second message/SASMessage>
</Exception>
</Exceptions>
```

Use an XML parser to break the XML into individual messages to display to a user. You can use the V8ERRORTEXT objectserverparm to suppress this XML.

Different errors are assigned different codes. IOM method calls can return many different codes. The code that they return is listed in various enumerations whose names end in "Errors". For example, the IOM DataService defines an enumeration called DataServiceERRORS in which the constants for all of the possible errors that are returned by the DataService are defined. For more information about the common errors for a particular call, see the documentation and Help. In addition to errors that listed explicitly for the method, SAS.GenericError can always be raised.

Receiving Events

Some IOM objects can generate events. For example, the LanguageService generates the events DatastepStart, DatastepComplete, ProcStart, ProcComplete, SubmitComplete, and StepError. In the Object Browser, you can recognize objects that generate events because the event procedures are marked using a lightning bolt icon.

Your Visual Basic program can implement procedures that receive these events if you declare the interface using the WithEvents keyword.

The following declaration must be outside any procedure:

```
Dim WithEvents obLS As SAS.LanguageService
```

After you declare an interface to receive events, the Visual Basic development environment provides empty definitions for the event procedures. The following code segment is a definition that is provided for the ProcStart event, along with a line that has been added to print a debug message tracing the start of the procedure:

```
Private Sub obLS_ProcStart(ByVal Procname As String)
```

```
Debug.Print "Starting PROC: "  Procname
End Sub
```

Note that the event procedure name is of the form *object-name_event-name*.

In the routine that initializes your workspace variable, you should initialize the LanguageService variable using a statement such as the following:

```
set obLS = obSAS.LanguageService
```

After this initialization, your events procedures will be called whenever SAS fires an event.

Programming in the .NET Environment

.NET Environment Overview

The Windows .NET environment is a Microsoft application development platform that supersedes technologies such as VB6 and Active Server Pages. The .NET environment defines its own object system with the Common Language Specification (CLS). This object system is similar to Java but contains many enhancements and additional features.

The .NET environment supports many languages, including the new C# language and the latest variant of Visual Basic, VB.NET. The differences among languages are mainly in the syntax that is used to express the same CLS-defined semantics.

The C# language is popular because of its syntactic resemblance to Java and because it was designed specifically for the .NET CLS. VB.NET has a syntax that appeals to devoted Visual Basic programmers, although the requirements of the .NET environment have created significant compatibility issues for the existing VB6 code base. Most of the example code in this section was written in C#, but it should be easy to translate to other .NET languages.

Because .NET is a new environment, interoperability between .NET programs and existing programs is very important. Microsoft devoted careful attention to this and uses several technologies to provide compatibility:

- Web Services (provided via ASP.NET or .NET remoting)
  - provides connectivity to the broadest range of software implementations. Web services toolkits supporting many different environments and languages are available from a variety of vendors.
  - can leverage existing web server infrastructure to provide connectivity.
  - has a simplistic mapping into the .NET object system. Method calls cannot return an interface (only data can be returned) and interface casts are not supported.
- COM Interop
  - creates useful .NET classes for most existing COM interfaces -- especially for the Automation compatible interfaces used by IOM.
  - has some weaknesses in the mapping due to lack of information in COM. For example, specific exception types are not distinguished because they cannot be declared in a COM type library.
- .NET Remoting binary formatter

    - provides seamless interfacing when talking to another .NET remoting application. The programming paradigm is very similar to Java RMI.
    - does not support traditional executables or other environments.

- PInvoke
    - can call existing C APIs in-process.
    - requires low-level programming.

COM Interop provides the highest quality interfaces for programs that run outside the .NET environment (like SAS IOM servers). This is the approach used for IOM programming under .NET. COM Interop with IOM servers provides a .NET interface that is superior to the .NET interface that Web Service proxies. IOM supports the connectivity to UNIX and z/OS servers that is a primary goal of Web Services architectures.

.NET developers who want to implement a Web Service for use by others have a number of options:

- Write an ASP.NET Web Service and call IOM interfaces via COM Interop.
- Write a .NET remoting Web Service and call IOM interfaces via COM Interop.
- Use the new COM Web Services feature of the SAS Integration Technologies Stored Process Server. This enables you to implement a Web Service with SAS language programming. For more information, see the *SAS BI Web Services: Developer's Guide*.

IOM Support for .NET

IOM servers are easily accessible to .NET clients, and the IOM Bridge capability allows calls to servers on UNIX and z/OS platforms.

Your .NET clients will naturally require .NET classes to provide any services that they need. With COM Interop, these .NET classes directly represent the individual IOM interfaces such as IWorkspace and ILanguageService. The .NET Software Development Kit (SDK) and development environments, such as Microsoft Visual Studio .NET, provide the tlbimp (type library import) utility program to read a COM type library and create a .NET assembly that contains .NET classes that correspond to each of the COM interfaces.

It is possible for the creator of a type library to create an official .NET interop assembly for the type library. This is called the primary interop assembly for the type library. When the type library creator has not done this, developers should import the type library themselves to make an interop assembly specific to their own project.

SAS®9 does not provide a primary COM interop assembly for any of the IOM type libraries or dynamic link libraries (DLLs). Thus, the first step in developing a .NET project that will use IOM is to import the desired type libraries or DLLs such as sas.tlb (which contains the Workspace interfaces) and SASOMan.dll (which contains the SAS Object Manager interfaces).

The following commands create the COM interop assemblies for sas.tlb and SASOMan.dll. You must modify the path to your shared files folder if you did not install SAS in the default location as follows:

```
tlbimp /sysarray
"c:\Program Files\SAS\Shared Files\Integration Technologies\SAS.tlb"
tlbimp /sysarray
```

```
"c:\Program Files\SAS\Shared Files\Integration Technologies\SASOMan.dll"
```

Because the type library import process converts between two different object models, there are a number of idiosyncrasies in the process that will be explained in detail in the following sections.

Classes and Interfaces

Before looking at how .NET handles classes and interfaces, it is important to understand how COM interfaces were used in VB6, because the .NET was designed to have some continuity with the treatment of COM objects in VB6.

IOM servers provide components that have a default interface and possibly additional interfaces. All access to the server component must occur through a method in one of its interfaces. While this is the same approach used by COM, VB6 also allowed access to the default interface via the coclass (component) name. IOM type libraries use the standard COM convention—interface names have a capital "I" prefix, while coclass names do not.

Thus, while the SAS®9 SAS::Fileref component supports both the default SAS::IFileref interface and an additional SAS::IFileInfo interface, a VB6 programmer previously called SAS::IFileref methods through a variable whose type was declared with the coclass name. Here is a typical example:

```
' VB6 code
' Primary interface declared as coclass name (without the leading "I").
Dim ws as new SAS.Workspace
Dim fref as SAS.Fileref
Dim name as string
Set fref = ws.FileService.AssignFileref("myfilref", _
"DISK", "c:\myfile.txt", "", name)
' Secondary interface must use COM interface name (requires "I").
' There is no "SAS.FileInfo" because this is only an interface,
' not an object (coclass).
Dim finfo as SAS.IFileInfo
' Call a method on the default interface.
debug.print fref.FilerefName
' Obtain a reference to the secondary interface on the object.
set finfo = fref
' Make a call using the secondary interface.
debug.print finfo.PhysicalName
```

The interop assembly for an IOM type library contains the following elements:

- a .NET interface for each COM interface. For example, .NET code that is equivalent to the preceding VB6 code contains both an IFileref interface and an IFileInfo interface in the assembly.
- a .NET interface with the same name as the coclass. Thus, for the Fileref coclass, there is also a Fileref interface in the assembly. This .NET interface is called the coclass interface. The coclass interface is almost identical to the default .NET interface for the coclass. It inherits from the default COM interface's .NET interface and defines no additional members. The SAS assembly's Fileref interface (a coclass interface) defines no members itself and inherits from the IFileref interface. The coclass interface plays an additional role for components that can raise events. For more information, see “Receiving Events” on page 34.

- a .NET class named with the "Class" suffix, which is called the RCW (Run-time Callable Wrapper) class. This class is represented by the FilerefClass in the following example.

Given these substitutions, here is the equivalent C# code:

```
// C#
// Using the "coclass interface" for the workspace and fileref.
// The name without the leading "I" came from the COM coclass,
// but in .NET it is an interface.
SAS.Workspace ws2 =
new SAS.Workspace(); // instantiable interface - see below
SAS.Fileref fref;
String name;
fref = ws2.FileService.AssignFileref("myfilref", "DISK",
"c:\\myfile.txt", "", out name);
// Secondary interface must use COM interface name (requires "I")
// There is no "SAS.FileInfo" because this is just an interface,
// not an object (coclass).
SAS.IFileInfo finfo;
// Call a method on the default interface.
Trace.Write(fref.FilerefName);
// Obtain a reference to the secondary interface on the object.
finfo = (SAS.IFileInfo) fref;
// Make a call using the secondary interface.
Trace.Write(finfo.PhysicalName);
```

While the VB6 and C# programs appear to be very similar, there is a difference at a deeper level. Unlike VB6, the .NET CLS makes a distinction between interface types and class types. All of the variables declared in the C# example are interfaces. None of the variables are classes. **`ws`** and **`fref`** are the coclass interfaces because the type library importer created them from the default interface of the Workspace and Fileref coclasses. **`finfo`** is a .NET interface variable of type IFileinfo—a type that the type library importer created from the COM IFileInfo interface.

COM coclasses that have the Creatable attribute in the type library can be instantiated directly. This is illustrated in the previous example by declaration of the following workspace variable:

```
Dim ws as new SAS.Workspace
```

In order to provide further similarity with VB6, the coclass interface on a creatable coclass (the .NET interface named Workspace in our example) has a unique feature. The type library importer adds some extra .NET metadata to the interface so that it can be used for instantiation. Thus, while it would normally be invalid to try to instantiate an interface, the following statement becomes possible:

```
SAS.Workspace ws = new SAS.Workspace();
```

Because of the extra metadata added to the SAS.Workspace interface by the type library importer, the C# and VB.NET compilers change this statement to the following:

```
SAS.Workspace ws = new SAS.WorkspaceClass();
```

This is a transformation done by the compiler. If you get a compilation error with the former syntax when using a language other than C# or VB.NET, then you should try the second approach, which includes an actual class name instead of a coclass interface name.

In order to complete the emulation of the type names that are used by VB6, the type library importer makes a transformation of method parameters to use the coclass

interface where possible. Whenever a COM method signature (or attribute type) refers to a default interface for a coclass in the same type library, the .NET method (or attribute) uses the coclass interface type.

So, in the previous example, while the COM IDataService::AssignFileref() method is defined to return an IFileref interface, the .NET IFileref interface is not used. Instead, the type library importer recognizes that the COM IFileref interface is the default interface of the Fileref coclass and substitutes its coclass interface (Fileref) as the AssignFileref() return type.

With all of these transformations, the VB6 compatible view of the interface is complete. Let us review the rules as they apply to IOM programming:

- There are effectively two types of interface:
  - default interfaces such as Workspace, FileService, and Fileref use the coclass interface, which has the same name as the component (no extra leading "I").
  - additional interfaces might be used by only one component (such as SAS::IFileInfo) or might be used by more than one (such as SASIOMCommon::IServerStatus). The names of these interfaces are identical to the COM interface names (and thus use the leading "I").
- Use the .NET variables of either type as interfaces (which they are), even if the type does not have the leading "I".
- Instantiate a component such as the Workspace or the ObjectManager by using the coclass interface (such as Workspace) in C# and VB6. Other languages might require you to use an actual class name (such as WorkspaceClass).

Simple Data Types

As illustrated in the following table, most of the simple data types from VB6 have the expected equivalents in .NET programming.

Table 2.1 *Data Types*

COM	VB6	.NET	VB.NET	C#
unsigned char	Byte	System.Byte	Byte	byte
VARIANT_BOOL	Boolean	System.Boolean	Boolean	bool
short	Integer	System.Int16	Short	short
long	Long	System.Int32	Integer	int
float	Single	System.Single	Single	float
double	Double	System.Double	Double	double
BSTR	String	System.String	String	string
DATE	Date	System.DateTime	Date	System.DateTime

Almost all of the data types are exact equivalents. The most significant difference is that Long, the 32-bit integer type in COM and VB6 has become Integer in VB.NET and Int in C#.

Arrays

An IOM array is represented as a COM SAFEARRAY. The .NET type library import utility (tlbimp) can convert SAFEARRAYs in one of two ways:

- using the /sysarray option—converted to a .NET System.Array
- not using the /sysarray option—converted to a one dimensional array with a zero lower bound

This latter approach is discouraged. It does not work well with IOM type libraries because many arrays in IOM interfaces are two dimensional. However, when you use the Visual Studio .NET IDE to import a type library, Visual Studio supplies the /sysarray option.

Therefore, for IOM programming, the resulting .NET arrays are passed in terms of the generic System.Array base class, instead of using the programming language's array syntax. Furthermore, an array variable is needed for input as well as output, because IOM follows the VB6 convention of always passing arrays by reference, even for input parameters.

As an illustration of input and output arrays, here is an example of using the IOM LanguageService in C#:

```
SAS.LanguageService lang = ws.LanguageService;
string[] sasPgmLines = {
"data _NULL_; ",
" infile \'" + filenameBox.Text + "\';",
" input;" ,
" put _infile_;" ,
"run;" } ;
System.Array linesVar = sasPgmLines; // identical to type of ref parm
lang.SubmitLines(ref linesVar);
bool bMore = true;
while (bMore) {
System.Array CCs;
const int maxLines = 100;
System.Array lineTypes;
System.Array logLines;
lang.FlushLogLines(maxLines, out CCs,
out lineTypes, out logLines);
for (int i=0; ilogLines.Length; i++) {
fileBox.Text += (logLines.GetValue(i) + "\n");
}
if (logLines.Length  maxLines)
bMore = false;
}
```

The example uses the sasPgmLines C# array to set up the array value. However, C# requires the type in the reference parameter declaration be identical to (not just implicitly convertible to) the type of the argument that is being passed. Thus, the example must use the linesVar variable as the reference parameter.

A similar situation occurs with the output parameters. The parameters must be received into System.Array variables. At that point, if there are to be only a few lines where the

array is accessed, it might be most convenient to access the array through the System.Array variable, as shown in the previous example. Note that the Length attribute is primarily applicable to one-dimensional arrays. For two-dimensional arrays, the GetLength() method (which takes a dimension number) is usually needed.

The input parameter in the previous example shows how to interchange arrays with System.Array objects. The output parameter can also be used. If the output parameter is used, then the **`while`** loop might be changed as follows:

```
while (bMore) {
System.Array CCs;
const int maxLines = 100;
System.Array lineTypes;
System.Array logLinesVar;
string []logLines;
lang.FlushLogLines(maxLines, out CCs,
out lineTypes, out logLinesVar);
logLines = (string [])logLinesVar; // explicit conversion
for (int i=0; ilogLines.Length; i++) {
fileBox.Text += (logLines[i] + "\n");
}
if (logLines.Length  maxLines)
bMore = false;
}
```

The primary benefit of using this **`while`** loop is the ability to use normal array indexing syntax when accessing the element within the **`for`** loop. Also, the assignment from logLinesVar to LogLines required a string[] cast, because the conversion from the more general System.Array to the specific array type is an explicit conversion.

The examples in this section use **`for`** loops to illustrate array indexing with each type of array declaration. In practice, simple loops can be expressed more concisely using the C# FOREACH statement. Here is an example:

```
foreach (string line in loglines){
filebox.Text += (line + "\n");
}
```

Another alternative, which avoids the use of two variables per array, is to use the Array.CreateInstance() method here as illustrated with the optionNames variable:

```
Array optionNames, types, isPortable, isStartupOnly, values,
errorIndices, errorCodes, errorMsgs;
optionNames=Array.CreateInstance(typeof(string),1);
optionNames.SetValue("MLOGIC",0);
iOS.GetOptions(ref optionNames, out types, out isPortable,
out isStartupOnly, out values, out errorIndices,
out errorCodes, out errorMsgs);
```

Enumerations

Many IOM methods accept or return enumeration types, which are declared in the IOM type library. In the preceding **`while`** loop, the CCs variable is actually an array of enumeration. The type library importer creates a .NET enumeration type for each COM enumeration in the type library.

Here is an elaboration of the LanguageService example that uses output enumeration to determine the number of lines to skip:

```
while (bMore) {
```

```
System.Array CCVar;
const int maxLines = 100;
System.Array lineTypes;
System.Array logLinesVar;
string []logLines;
SAS.LanguageServiceCarriageControl []CCs;
lang.FlushLogLines(maxLines, out CCVar,
out lineTypes, out logLinesVar);
logLines = (string [])logLinesVar;
CCs = (LanguageServiceCarriageControl [])CCVar;
for (int i=0; i<logLines.Length; i++) {
// Simulate some carriage control with newlines.
switch (CCs[i]) {
case LanguageServiceCarriageControl.
LanguageServiceCarriageControlNewPage:
fileBox.Text+="\n\n\n";
break;
case LanguageServiceCarriageControl.
LanguageServiceCarriageControlSkipTwoLines:
fileBox.Text += "\n\n";
break;
case LanguageServiceCarriageControl.
LanguageServiceCarriageControlSkipLine:
fileBox.Text += "\n";
break;
case LanguageServiceCarriageControl.
LanguageServiceCarriageControlOverPrint:
continue; // Don't do overprints.
}
fileBox.Text += (logLines[i] + '\n');
}
if (logLines.Length < maxLines)
bMore = false;
}
```

COM does not provide scoping for enumeration constant names, either with respect to the enumeration or with respect to the interface to which an enumeration might be related. Thus, the name for each constant in the type library must contain the name of the enumeration type in order to avoid potential clashes. .NET, on the other hand, does provide scoping for constant names within enumeration names and requires them to be qualified. This combination of factors results in the very long and repetitive names in the preceding example.

IOM places loose constants in enumerations that end in CONSTANTS. An enumeration simply named CONSTANTS includes constants for the entire type library. Some interfaces have their own set. For example, LibrefCONSTANTS contains the constants for the Libref interface. Enumerations that contain the word INTERNAL are collections of constants that are not documented for customer use.

Exceptions

The COM type library importer maps all failure HRESULTs into the same .NET exception: System.Runtime.InteropServices.COMException. This exception is thrown whenever an IOM method call fails. The specific type of failure is determined by its HRESULT code, which is found in the ErrorCode property of the exception.

Types of errors fall into two broad categories: system errors and application errors. The system error codes are standard to all Windows programming languages. The following table shows common error codes in IOM applications.

Table 2.2 *Error Codes*

HRESULT Value	Symbolic Name	Description
0x8007000E	E_OUTOFMEMORY	The server ran out of memory.
0x80004001	E_NOTIMPL	The method is not implemented.
0x80004002	E_NOINTERFACE	The object does not support the requested interface.
0x80070057	E_INVALIDARG	You passed an invalid argument.
0x80070005	E_ACCESSDENIED	You lack authorization to complete the request.
0x80070532	HRESULT_FROM_WIN32(ERROR_PASSWORD_EXPIRED)	The supplied password is expired.
0x8007052E	HRESULT_FROM_WIN32(ERROR_LOGON_FAILURE)	Either the user name or the password is invalid.
0x800401FD	CO_E_OBJNOTCONNECTED	You tried to call an object that no longer exists.
0x80010114	RPC_E_INVALID_OBJECT	You tried to call an object that no longer exists (equivalent to CO_E_OBJNOTCONNECTED).

In principle, these exceptions can be returned from any call, although exceptions that are related to a password occur only when connecting to a server.

The second broad category of errors consists of those that are specific to particular IOM applications. These errors are documented by enumerations in the type library. The IOM class documentation lists which of these (if any) can be returned from a particular method call. The type library typically has one ERRORS enumeration for errors that are relevant to more than one interface. It also has another enumeration for each interface that has its own set of errors. For example, there is a DataServiceERRORs enumeration for the IDataService interface.

Besides the ErrorCode property, there are several other fields. The most important of these is the ErrorMessage. SAS®9 and later servers return an entire list of messages. Because COM does not support a chain of exceptions, these messages are returned as an XML-based list in the ErrorMessage field.

If you want to present an attractive error message to your user, you need to parse the error message by using an XML parser.

The following code fragment shows error handling for a libref assignment call. This code makes an extra check for an invalid pathname error and illustrates a very simple reformatting of the error message field by using XSL.

```
bool bPathError;
string styleString =
"" +
"<xsl:stylesheet xmlns:xsl= " +
"\"http://www.w3.org/1999/XSL/Transform\" version=\"1.0\">" +
"<xsl:output method=\"text\"/>" +
"<xsl:template match=\"SASMessage\">" +
"[<xsl:value-of select=\"@severity\"/>] " +
"<xsl:value-of select=\".\"/> " +
"</xsl:template>" +
"</xsl:stylesheet>";
try
{
ws.DataService.AssignLibref(nameField, engineField,
pathField, optionsField);
}
catch (COMException libnameEx) {
switch ((DataServiceERRORS)libnameEx.ErrorCode)
{
case DataServiceERRORS.DataServiceNoLibrary:
bPathError = true;
break;
}
try
{
// Load the style sheet as an XmlReader.
UTF8Encoding utfEnc = new UTF8Encoding();
byte []styleData = utfEnc.GetBytes(styleString);
MemoryStream styleStream = new MemoryStream(styleData);
XmlReader styleRdr = new XmlTextReader(styleStream);
// Load the error message as an XPathDocument.
byte []errorData = utfEnc.GetBytes(libnameEx.Message);
MemoryStream errorStream = new MemoryStream(errorData);
XPathDocument errorDoc = new XPathDocument(errorStream);
// Transform to create a message.
StringWriter msgStringWriter = new StringWriter();
XslTransform xslt = new XslTransform();
xslt.Load(styleRdr);
xslt.Transform(errorDoc,null, msgStringWriter);
// Return the resulting error string to the user.
errorMsgLabel.Text = msgStringWriter.ToString();
errorMsgLabel.Visible = true;
}
catch (XmlException)
{
// Accommodate SAS V8-style error messages with no XML.
errorMsgLabel.Text = libnameEx.Message;
errorMsgLabel.Visible = true;
}
}
```

The error text for the COM exception will be XML only if you are running against a SAS®9 server. If your client program runs against a SAS 8 server or against SAS®9

servers with the V8ERRORTEXT object server parameter, which suppresses the XML in error messages, then the construction of the XPathDocument will throw an exception. The previous example catches this exception and returns the error message unchanged.

Accessing SAS Data with ADO.NET

SAS Providers for OLE DB includes an OLE DB provider for use with the SAS Workspace. This provider makes it possible for your program to access SAS data within an IOM Workspace using ADO.NET's OLE DB adapter. This technique is particularly important in IOM Workspace programming because it is often the easiest way to get results back to the client after a SAS PROC or DATA step.

The following example shows how to copy an ADO.Net data set into a SAS data set:

```
' This method sends the given data set to the provided workspace, and
' assigns the WebSvc libref to that input data set
Private Sub SendData(ByVal obSAS As SAS.Workspace,
ByVal inputDS As DataSet)
' Take the provided data set and put it in a fileref in SAS as XML
Dim obFileref As SAS.Fileref
Dim assignedName As String
' Filename websvc TEMP;
obFileref = obSAS.FileService.AssignFileref(
"WebSvc", "TEMP", "", "", assignedName)
Dim obTextStream As SAS.TextStream
obTextStream = obFileref.OpenTextStream(
SAS.StreamOpenMode.StreamOpenModeForWriting, 2000)
obTextStream.Separator = " "
obTextStream.Write(""<?xml version=""1.0"" standalone=""yes"" ?>")
obTextStream.Write(inputDS.GetXml())
obTextStream.Close()
' An ADO.Net data set is capable of holding multiple tables, schemas,
' and relationships. This sample assumes that the ADO.Net data set
' only contains a single table whose name and columns fit within SAS
' naming rules. This would be an ideal location to use XMLMap to
' transform the schema of the provided data set into something that
' SAS may prefer.
' Here, the default mapping is used. Note that the LIBNAME statement
' uses the fileref of the same name because we did not specify a file.
' Using the IOM method is cleaner than using the Submit because an
' error is returned if there is a problem making the assignment
obSAS.DataService.AssignLibref("WebSvc", "XML", "", "")
' obSAS.LanguageService.Submit("libname webSvc XML;")
End Sub
```

The following example shows how to copy a SAS data set into an ADO.Net data set:

```
' Copy a single SAS data set into a .NET data set
Private Function GetData(ByVal obSAS As SAS.Workspace, ByVal
sasDataset As String) As DataSet
Dim obAdapter As New System.Data.OleDb.OleDbDataAdapter("select * from "
& sasDataset, "provider=sas.iomprovider.1; SAS Workspace ID=" &
obSAS.UniqueIdentifier)
Dim obDS As New DataSet()
' Copy data from the adapter into the data set
obAdapter.Fill(obDS, "sasdata")
GetData = obDS
```

```
End Function
```

For more information about using OLE DB with IOM, see "Using the SAS IOM Data Provider" on page 63 .

Object Lifetime

In native COM programming, reference counting, whether it is explicitly written or managed by smart pointer wrapper classes, can require careful attention to detail. The VB6 run-time environment did much to alleviate that, and .NET's garbage collection has a similar simplifying effect. When programming with Interop, COM objects are normally released via garbage collection. You can release the objects at any time by making a sufficient number of calls to System.Runtime.InteropServices.Marshal.ReleaseComObject().

With most IOM objects, the exact timing of releases is unimportant. For example, in the workspace, the various components maintain references among themselves so that they are not destroyed, even if the client releases them. The workspace as a whole shuts down (and is disconnected from its clients) when the client calls its Close() method. This also causes the process to shut down in the typical (not pooled) situation where there is only one workspace in the process. Releases only become significant when all COM objects for the workspace hierarchy are released. When all objects are released, the objects cannot be used again, and the Close() method cannot be called. Therefore, the server shuts down. If you do not call Close(), then the SAS process (sas.exe) does not terminate until .NET runs garbage collection.

Receiving Events

When a COM component raises events, the type library provides helper classes to make it easy to receive those events by using delegates and event members, which are the standard .NET event handling mechanisms.

The first (and often the only) event interface that is supported by a component is included in the coclass interface. You can add event listener methods to the event members of the coclass interface, which are then called when the IOM server raises the events.

Here is an example of collecting SAS Language events:

```
private void logDSStart() {
progress.Text += "[LanguageService Event] DATASTEP start.\n";
}
private void logDSComplete() {
progress.Text +=
"[LanguageService Event] DATASTEP complete.\n";
}
private void logProcStart(string procName) {
progress.Text += "[LanguageService Event] PROC " +
procName + " start.\n";
}
private void logProcComplete(string procName) {
progress.Text += "[LanguageService Event] PROC " +
procName + " complete.\n";
}
private void logStepError() {
progress.Text += "Step error.\n";
}
```

```
private void logSubmitComplete(int sasrc) {
progress.Text +=
"[LanguageService Event] Submit complete return code: " +
sasrc.ToString() + ".\n";
}
// Event listeners use the LanguageService coclass interface.
// The Language Service also includes events for the default event interface.
SAS.LanguageService lang = ws.LanguageService;
lang.DatastepStart += new
CILanguageEvents_DatastepStartEventHandler(this.logDSStart);
lang.DatastepComplete += new
CILanguageEvents_DatastepCompleteEventHandler(
this.logDSComplete);
lang.ProcStart += new
CILanguageEvents_ProcStartEventHandler(this.logProcStart);
lang.ProcComplete += new
CILanguageEvents_ProcCompleteEventHandler(this.logProcComplete);
lang.StepError += new
CILanguageEvents_StepErrorEventHandler(this.logStepError);
lang.SubmitComplete += new
CILanguageEvents_SubmitCompleteEventHandler(
this.logSubmitComplete);
// Submit source, clear the list and log, etc...
// Stop listening.
// The "new" operator here is confusing.
// Event removal does not really care about the particular
// delegate instance. It just looks at the identity of the
// listening object and the method being raised. Getting a
// new delegate is an easy way to gather that together.
SAS.LanguageService lang = ws.LanguageService;
lang.DatastepStart -= new
CILanguageEvents_DatastepStartEventHandler(this.logDSStart);
lang.DatastepComplete -= new
CILanguageEvents_DatastepCompleteEventHandler(
this.logDSComplete);
lang.ProcStart -= new
CILanguageEvents_ProcStartEventHandler(this.logProcStart);
lang.ProcComplete -= new
CILanguageEvents_ProcCompleteEventHandler(this.logProcComplete);
lang.StepError -= new
CILanguageEvents_StepErrorEventHandler(this.logStepError);
lang.SubmitComplete -= new
CILanguageEvents_SubmitCompleteEventHandler(
this.logSubmitComplete);
```

IOM event interfaces begin with CI (not just "I"), because they are not dual interfaces. They are, instead, ordinary COM vtable interfaces, which makes them easier for some types of clients to implement.

COM Interop also has support for components that raise events from more than one interface. In this case, you must add your event handlers to the RCW interface (such as LanguageServiceClass in the previous example). Currently, there are no SAS IOM components with a publicly documented interface that is not the default.

Using VBScript

Overview of IOM Interfaces in VBScript

The preceding sections provide many examples of using the IOM interfaces in a full Visual Basic environment (or in a VBA environment like Microsoft Word and Excel).

You can also use the IOM interfaces from a VBScript environment. VBScript is a commonly used scripting language that is available in Active Server Pages (ASP), Dynamic HTML (DHTML), Microsoft Outlook 97 and later, and Windows Scripting Host.

Scripto is an ActiveX DLL that has been developed by SAS in order to provide workarounds for the following common VBScript limitations:

- Method calls in VBScript are limited to a single return value of type Variant.
- Arrays can contain Variants only.
- There is no support for reading or writing binary files.

Note: Scripto is provided with SAS Integration Technologies so that developers who choose to use VBScript can effectively use the SAS automation interfaces. However, Scripto is not specific to SAS and can be used with other automation interfaces.

The performance of VBScript is often slower than a Visual Basic application for several reasons. First, VBScript provides only late binding. It uses IDispatch instead of v-table calls. Also, VBScript is interpreted, not compiled. In addition, the way that VBScript invokes methods (InvokeMethod) causes additional overhead because all parameters must be copied twice to get them in the proper format.

The slower performance of VBScript is especially evident in the case of safe arrays in which SAS expects to contain the actual type of element in the array (such as string or an enumeration value). VBScript expects it to contain only VARIANTS, which in turn contain the appropriate type of element. InvokeMethod takes care of this conversion. However, it produces an additional copy of the array.

Scripto does not address these performance issues. We recommend that if performance is an issue, consider something other than a scripting language for your implementation. We also recommend that you only use Scripto when calling methods whose signatures require it.

Two components are implemented by the Scripto DLL: Scripto and StreamHelper.

The Scripto Interface: IScripto

IScripto is the single interface to the Scripto component. It provides two methods:

SetInterface *IUnknown*
: The SetInterface method is used to specify the IOM interface that contains the method that you want to invoke. The interface must be set before using InvokeMethod. The interface specified must support IDispatch, and that IDispatch implementation must support type information. If either of these is not true, SetInterface returns E_INVALIDARG(&H80070057). An instance of the Scripto component can handle only a single interface at a time. Although you can create multiple instances of Scripto that handle with a different interface, you typically need only a single instance of Scripto. Switching between interfaces does not consume a

significant amount of system resources. SetInterface performs an AddRef function on the interface that is specified. You can release this reference when you are finished with the interface using the following statement:

```
SetInterface Nothing
```

Also, when you release IScripto, it releases any interface that Scripto still references. After you set an interface, you can still make calls on it without using Scripto.

Return-Parameter InvokeMethod(*methodName, params*)
The InvokeMethod method invokes the desired IOM interface method through the Scripto DLL. This method can be invoked only after the interface has been set using SetInterface. The *methodName* parameter is a string that contains the name of the method to invoke on the interface that has been set. The *params* parameter is an array whose elements contain the parameters for the method that is to be invoked. The order of the parameters in the array must be in reverse order of how they are specified in the method signature. For example, the first parameter in the method signature must be the final parameter in the array. The array must have exactly the same number of elements as there are parameters to the method. Otherwise, InvokeMethod returns DISP_E_BADPARAMCOUNT(&H8002000E).

Note: If the method has a parameter that uses the [retval] modifier, then this parameter is returned as the return value to InvokeMethod and you do not need an element in the params array for it.

If the method that you invoke returns a value, then InvokeMethod returns the value in the *Return-Parameter*.

Note: If the method returns a reference to an object, then be sure to use the Set keyword. For example:

```
Set obServer = obScripto.InvokeMethod(_ "CreateObjectByServer", ServerDefParms)
```

The StreamHelper Interface

The StreamHelper interface contains three methods that are related to working with the SAS BinaryStream (available through the SAS FileService). These methods are useful only when working with SAS; these methods make calls on IBinaryStream to read and write binary data.

WriteBinaryFile *IBinaryStream, fileName*
copies the entire contents of the given IBinaryStream into the specified file. This method can be useful for copying a SAS ResultPackage from the SAS server to the machine where the VBScript is running. It can also be used to copy GIF images. The *IBinaryStream* parameter that is passed in must have already been opened with permission to read.

ReadBinaryFile *fileName, IBinaryStream*
reads the entire contents of the file into the SAS BinaryStream. This method does the reverse of what WriteBinaryFile does. The *IBinaryStream* parameter that is passed in must be open for permission to write. Sending binary data to SAS can be useful if you want to include a binary file in a SAS package.

arrayOfBytes ReadBinaryArray (*IBinaryStream, numBytes*)
reads the specified number of bytes from the *IBinaryStream* parameter and returns them in the *arrayOfBytes* array. If the value of the *numBytes* parameter is 0, then the entire contents of the BinaryStream is returned in the array. The array is a variant containing a safe array of bytes. This array can be passed directly to the Response.BinaryWrite method of Microsoft ASP.

Programming Examples

- This programming example illustrates using the Scripto component to reverse the order of parameters when invoking a method. This example assumes that the method has been defined with the following Interface Definition Language (IDL) code:

  ```
  HRESULT MethodToInvoke([in]param1, [out]param2, [out]param3)
  ```

  To call this method with Scripto, use the following code:

  ```
  Dim f(2) 'An array of 3 VARIANTS
  obScripto.SetInterface myInterface
  f(2) = param1
  obScripto.InvokeMethod "MethodToInvoke", f
  param3 = f(0) ' Note that the order of parameters is reversed
  param2 = f(1)
  ```

- The next example uses the IOM LanguageService to submit SAS language statements to the IOM server. It then uses Scripto to invoke the FlushLogLines method of the LanguageService. Using Scripto provides two important advantages over invoking the method directly from VBScript:

  - It converts the three arrays that are returned from the method from arrays of long integers to arrays of variants. (Note that VBScript can use arrays that contain elements of data type variant only.)
  - It allows the VBScript application to receive more than one return parameter from the method that is invoked. (Note that when you invoke a method directly from VBScript, you are limited to a single return value.)

  ```
  set obSAS = CreateObject("SAS.Workspace.1.0")
  Set obScripto = CreateObject("SASScripto.Scripto")
  obSAS.LanguageService.Submit "proc options;run;"
  obScripto.SetInterface obSAS.LanguageService
  ' This example uses Scripto to invoke the FlushLogLines method
  ' instead of invoking the method directly from VBScript
  ' as shown in the following statement:
  ' obSAS.LanguageService.FlushLogLines 1000, carriageControls,_
  ' linetypes, logLines
  Dim flushLogLinesParams(3)
  ' Note that the FlushLogLines method takes 4 parameters:
  ' 1) numLinesRequested (in) Specifies an upper limit on the
  ' number of lines to be returned.
  ' 2) carriageControls (out) An array that indicates carriage
  ' control for each line returned.
  ' 3) lineTypes (out) An array that indicates the line type
  ' for each line returned.
  ' 4) logLines (out) Contains the SAS log output lines
  ' retrieved by this call.
  flushLogLinesParams(3) = 1000
  obScripto.InvokeMethod "FlushLogLines",_
  flushLogLinesParams
  ' flushLogLinesParams(0) now has logLines
  ' flushLogLinesParams(1) now has lineTypes
  ' flushLogLinesParams(2) now has carriageControls
  ' Print the first line
  wscript.echo flushLogLinesParams(0)(0)
  ```

The CreateObject() call in this example creates a server object by providing a COM ProgID. The ProgID is a string form of the COM class identifier (CLSID). In this case, the ProgID describes a SAS session that provides the Workspace interface. By explicitly specifying the major and minor version 1.0 as a suffix to the ProgID SAS.Workspace, you can ensure compatibility with all versions of the SAS Integration Technologies IOM server. If you are using a SAS®9 SAS Integration Technologies client and you do not specify a version suffix to the ProgID, then you cannot connect to a SAS 8 COM server.

Note: Two-level version suffixes were not available in previous versions of the SAS Integration Technologies Windows client.

- The next example illustrates using ReadBinaryArray with Microsoft Active Server Pages. Assume that a SAS program has already run that uses SAS/GRAPH to write a GIF image to a fileref called **`img`**. This example then uses the FileService to read that file back to the browser, with no intermediate files on the web server.

```
set obFref = obSAS.FileService.UseFileref("img")
set obBinaryStream = obFref.OpenBinaryStream(1)
' SAS.StreamOpenModeForReading=1
set obStreamHelper = CreateObject("SASScripto.StreamHelper")
response.contenttype = "image/gif"
response.binarywrite obStreamHelper.ReadBinaryArray(_
obBinaryStream, 0)
```

- The next example shows how to use WriteBinaryFile to copy a SAS ResultPackage. Demo.sas is a stored procedure that creates a ResultPackage in an archive.

```
obSAS.LanguageService.StoredProcessService.Execute
"demo.sas",_
parameters
```

The following lines move the package that was just created from the SAS server to the local machine:

```
Set obRemotePackageFileRef = obSAS.FileService.AssignFileref(_
"", "", "./demo.spk", "", "")
```

To open a binary stream on the SAS server for the package file, use the following:

```
Set obRemotePackageBinaryStream = _
obRemotePackageFileRef.OpenBinaryStream(1)
'StreamOpenModeForReading
```

Because VBScript cannot handle binary files, use the Scripto object to write the binary file on the local machine:

```
Set obStreamHelper = Server.CreateObject("SASScripto.StreamHelper")
obStreamHelper.WriteBinaryFile obRemotePackageBinaryStream,_
"c:\myfile.spk"
```

Programming with Visual C++

All SAS IOM interfaces are designed to work well with Microsoft Visual C++.

The documentation is written in terms of Visual Basic. This means that the default interface of a component is listed as if it were the interface of the COM object (coclass) itself, even though in pure COM terms, the default interface is just one of many interfaces implemented by the object. Thus, a C++ programmer programs to the

IWorkspace interface, even though the Visual Basic documentation shows the methods as belonging to the Workspace object. This section discusses issues with which Visual C ++ programmers must be concerned (in addition to the material covered in "Programming with Visual Basic" on page 14).

All IOM interfaces implemented by SAS are COM dual interfaces. Therefore, methods and property get and set routines can be called as direction entry points with positional parameters. Although IOM interfaces implement an IDispatch interface at the beginning of each v-table, this is only for compatibility with older OLE Automation controllers. Visual C++ programs should not make calls by using IDispatch::Invoke; but should instead call through the v-table entry for the specific method that they want to call. The ClassWizard-generated wrappers for IDispatch (with COleDispatchDriver) should not be used in IOM programming. This feature of Visual C++ is now useful only for interfaces that contain only IDispatch.

All event (also called source) IOM interfaces are COM custom interfaces. This means that callers to the Advise method should pass interfaces that derive only from IUnknown, not IDispatch. All parameters to the event interfaces are only in parameters, which means that none of the interfaces support the ability to return data to SAS through the event interface.

To use the IOM interface in your Visual C++ program, you should import the IOM interface type library. Here is an example:

```
#import "sas.tlb"
```

In order for this to work, you must ensure that the type library directory is listed in your include path.

The import statement causes everything in the type library to be placed in a namespace. The fully qualified name for IWorkspace is SAS::IWorkspace. Also see the Using directive in Visual C++, and the **`-no_namespace`** attribute on the import statement.

When you import a type library, the Visual C++ compiler creates a comprehensive set of definitions that are specific to that type library by using the helper classes in COM compiler support (as defined through <comdef.h>). The helper classes perform many useful functions, including the following:

- provide smart pointers for interface references
- map COM HRESULTs to C++ exceptions
- use helper classes for BSTRs
- create wrapper functions that return the IDL-defined return value instead of an HRESULT
- provide create instance helpers

Programming with the Visual C++ COM compiler support is almost as easy as calling the functions in Visual Basic.

Unfortunately, as of Visual C++ version 6, the COM compiler support is lacking in one important area. There are no wrapper functions for handling safe arrays. You must deal with the OLE Automation safe array API directly.

Dealing with dimensions in this API requires care. You must be particularly careful if you are dealing with two-dimensional arrays. The APIs that deal with safe arrays take a dimension number that is 1-based. In a two-dimensional array, the rows are indexed in dimension 1 and the columns by dimension 2. When you create an input array by using SafeArrayCreate(), the bounds are also passed in this order (the row bounds are passed in rgsabound[0] and the column bounds are passed in rgsabound[1]). Do not be confused by the ordering that you see when you display a safe array structure in the debugger.

Finally, keep in mind that for IOM method calls, lower bounds must always be zero.

Chapter 3

Using the Object Manager

Using the SAS Object Manager

SAS Object Manager Overview

The SAS Object Manager is a component that executes on the client machine and it is used to create and manage objects on IOM servers. The object manager can use server definitions from a metadata server, or define the servers in the code. When using a metadata server, the object manager can use IOM server definitions that are administered separately from the application. Using a metadata definition enables client applications to connect to a server simply by using the server name. The definition for this server can change as required without affecting the application.

The object manager can create a SAS object in one of these ways:

- through local COM if the SAS server runs on the same machine as the client
- through DCOM if the SAS server runs on another machine that supports DCOM
- through the IOM Bridge for COM (SASComb.dll) if the SAS server runs on another machine that does not support COM or DCOM functionality (z/OS or UNIX [Solaris, HP-UX IPF, HP 64, AIX, ALX, Linux])
- through the IOM Bridge for COM (SASComb.dll) if the SAS server runs on Windows

With the SAS Object Manager, you can perform the following tasks:

- launch SAS objects, such as SAS workspaces
- select between running SAS objects
- retrieve definitions from a metadata server

The object manager can be used in Visual Basic, C, C++, and VBScript (with the help of Scripto) programs.

The object manager can also be used with the .NET framework by using COM Interop.

Code Reference

The reference documentation for the SAS Object Manager is shipped with the object manager as online Help in the file sasoman.chm.

This file is located in SAS Integration Technologies installation directory. See “Default Installation Directory” on page 3.

Creating an Object

Object Factory Classes

The object manager uses the ObjectFactory and ObjectFactoryMulti2 classes to create objects. These classes contain the same components, but ObjectFactory is a singleton object and ObjectFactoryMulti2 is not a singleton object. A process can use only one instance of a singleton object at a time.

If you are using pooling, then the ObjectFactory class is recommended. If you are not using pooling, then the ObjectFactoryMulti2 class is recommended.

The CreateObjectByLogicalName and CreateObjectByServer methods are available from both of the object factory classes.

CreateObjectByLogicalName Method

This method creates a SAS object from a logical name. When you use CreateObjectByLogicalName, you must define metadata for your connections. Each connection should have a logical name associated with it. You can then use the logical name to make a connection. This technique enables you to change the machines where SAS is running without modifying your code.

Here is the signature for this method:

```
Set obSAS = obObjectFactory.CreateObjectByLogicalName(Name,
Synchronous, LogicalName, LoginReference)
```

CreateObjectByLogicalName requires the following parameters:

Name
specifies the name that will be associated with the object. The name can be useful when your program is creating several objects and must later distinguish among them.

Synchronous
indicates whether a connection is synchronous or asynchronous. The synchronous connection (TRUE) is most frequently used and is the simplest connection to create. If this parameter equals TRUE, then CreateObjectByLogicalName does not return until a connection is made. If this parameter equals FALSE, then the caller must get the created object from the ObjectKeeper.

LogicalName
provides the logical name that is associated with the server. The logical name that you specify must match the server's logical name that is defined on the metadata server.

LoginReference
specifies a reference to an identity object on the metadata server. If you specify a null value for LoginReference, then a value is retrieved from the metadata server based on your current connection to the metadata server. In most cases, you should use a null value. If you are using a COM or DCOM server connection, then the Windows integrated security is used and this parameter is ignored. If you are using an IOM Bridge connection, then the login reference is looked up on the metadata server in order to find the matching login object, which defines the user name and password to use. The lookup is performed based on the authentication domain of the server. A

user can have different logins for different security domains. The login reference is an object ID that points to an identity. You associate the identity with one or more login objects, but each identity can have only one login for each domain.

The CreateObjectByLogicalName method creates a new object by performing the following steps:

1. Creates a list of all ServerDefs that define the provided logical name.
2. Selects the first ServerDef in the list. (The first definition can vary depending on the metadata server.)
3. Locates a LoginDef that matches both the DomainName (from the ServerDef) and the provided LoginReference.
4. Attempts to create a connection for each MachineDNSName in the ServerDef, until a successful connection is made. The results are recorded in the log.

   If a connection to SAS could not be established, then the next ServerDef is tried and the process is repeated from step 3.

   If no object has been created after going through all the host names in each ServerDef and all the ServerDefs that match the logical name, then an error is returned. Use the GetCreationLog method to check for errors. For more details, see "SAS Object Manager Error Reporting " on page 50.

   If an object is created and the value for the Synchronous parameter is FALSE, then the new interface is added to the object keeper.

The name that is passed to the method can be used in other IObjectKeeper methods to determine which object to locate or remove. The name is also set on IObjectKeeper->Name before this method returns. The SAS Object Manager never looks at IObjectKeeper->Name, so a client could change the name after calling Create. The xmlInfo is defined only when this method returns an IObjectKeeper. For more information, see "SAS Object Manager Error Reporting " on page 50.

Note: The object keeper should be notified when an object is no longer in use (using the RemoveObject method) if the object was created with an asynchronous connection or if the object was explicitly added to the object keeper (by using the AddObject method).

The following example creates a workspace object that is named TestWorkspace by using the logical name Test Server. The null loginReference indicates that the identity that is associated with the current metadata server login is used to find a Login object.

```
Dim obObjectFactory As New SASObjectManager.ObjectFactory
Dim obSAS As SAS.Workspace
' This assumes that either your metadata configuration files are stored in
' the default location, or that the location of your metadata configuration
' files is stored in the registry. If this is not the case, you can specify
' the location of your configuration files by calling
' obObjectFactory.SetMetadataFile(systemFile, userFile, false)
Set obSAS = obObjectFactory.CreateObjectByLogicalName(
"TestWorkspace", True, "Test Server", "")
```

CreateObjectByServer Method

This method creates an object from a ServerDef object instead of a logical name. It also accepts the actual user ID and password instead of a login reference.

Here is the signature for this method:

```
Set obSAS = obObjectFactory.CreateObjectByServer("Name", Synchronous,
obServerDef, "Username", "Password")
```

The CreateObjectByServer method requires the following parameters:

Name
specifies the name that will be associated with the object. The name can be useful when your program is creating several objects and must later distinguish among them.

Synchronous
indicates whether a connection is synchronous or asynchronous. The synchronous connection is most frequently used and is the simplest connection to create. If this parameter equals TRUE, CreateObjectByServer does not return until a connection is made. If this parameter equals FALSE, the function returns a null value immediately. When a connection is made, the object is stored in the ObjectKeeper.

obServerDef
specifies a Server Definition object.

Username
specifies the user name that will be used for authentication on the server.

Password
specifies the password that will be used for authentication on the server.

This method attempts to connect to each of the hosts listed in the provided ServerDef, one at a time, until either a successful connection is made or all hosts have failed. The Username and Password parameters are not required for ServerDefs that use the COM protocol.

The following example creates a new workspace object named TestWorkspace:

```
Dim obObjectFactory As New SASObjectManager.ObjectFactory
Dim obSAS As SAS.Workspace
Dim obServer As New SASObjectManager.ServerDef
obServer.MachineDNSName = "RemoteMachine.company.com"
obServer.Protocol = ProtocolBridge
obServer.Port = 8591
Set obSAS = obObjectFactory.CreateObjectByServer("TestWorkspace", True,
obServer, "myUserName", "myPassword")
```

SAS Object Manager Interfaces

The principal interfaces of the SAS Object Manager are as follows:

IObjectFactory
provides collection methods for ServerDefs, LoginDefs, LogicalNameDefs, SAS objects, and ObjectPools. It also provides methods to establish connections to SAS servers and to define metadata sources.

IObjectKeeper
stores an interface and can retrieve it later, possibly from another thread. The IObjectKeeper interface is also used as a rendezvous point for objects that are created asynchronously from the ObjectFactory.

ILoginDef
creates and manipulates login definitions (LoginDefs). A LoginDef is needed only for connections that use the IOM Bridge for COM.

ILoginDefs
: contains the standard collection methods Count, _NewEnum, Add, Item, and Remove where the key is the LoginDefName. It also supports one additional method: CheckAccess.

IServerDef
: specifies how to connect to the server. For a local or DCOM connection, only the Name and Hostname values are needed. The IOM Bridge for COM requires Protocol to be set to ProtocolBridge. Port and ServiceName can be used with the IOM Bridge for COM.

IServerDefs
: contains the standard collection methods Count, _NewEnum, Add, Item, and Remove where the key is the ServerDefName. It also supports one additional method: CheckAccess.

IObjectPools
: can create, enumerate, locate, and remove ObjectPool objects.

IObjectPool
: configures parameters for an ObjectPool.

IPooledObject
: notifies a pool when the associated SAS object can be returned to the pool.

The reference documentation for using the SAS Object Manager interfaces is shipped with the SAS Object Manager in the sasoman.chm Help file.

Using a Metadata Server with the SAS Object Manager

Connecting to Metadata

Before you can use server metadata, you must first connect to a metadata server by using a metadata configuration file. For more information, see "Metadata Configuration Files" on page 49.

If you created your configuration files by using the ITConfig utility, then the SAS Object Manager connects to the metadata server automatically when you call a method or interface that requires metadata.

If your configuration files are not in the default location and the location is not stored in the Windows registry, then you must specify the location by using the SetMetadataFile method. The SetMetadataFile method has three parameters: the full path to the system configuration file, the full path to the user configuration file (optional), and a flag that indicates whether to store the file path values in the registry.

Note: An application should call the SetMetadataFile method one time only. To create a new metadata server connection, use the CreateOMRConnection method.

Note: The user ID that is used to log on to SAS is determined when the object is launched. After the object is launched, the user ID cannot be changed.

Metadata Caching

When a metadata server connection is established, the metadata is read from the server and stored by the SAS Object Manager in a cache. The cached metadata is used when you call CreateObjectByLogicalName, ServerDefs, or LoginDefs.

You can use the SetRepository method to refresh the metadata from the current metadata repository or read metadata from a new repository. For details about the SetRepository method, see the SAS Object Manager reference documentation (sasoman.chm).

For threaded applications, using the metadata cache in the SAS Object Manager causes performance issues. The CreateObjectByOMR method enables you to read metadata from the server without caching the metadata.

Object Definitions

There are three definitions that are useful for defining IOM objects. These definitions can be created either in the source code or on the metadata server:

Server definition (ServerDef)
: must be created before an IOM server can be launched using the SAS Object Manager. The server definition can either be loaded from a metadata server or created dynamically. The server definition is independent of the user. Server definitions include a *Logical Name* attribute.

Login definition (LoginDef)
: contains user-specific information, including user name, password, and domain. Login definitions are a convenience and are not required for creating a connection to an IOM server. They provide a mechanism for storing persistent definitions of user names and passwords.

  LoginDefs also allow multiple definitions for the same user on different security domains. For example, you could use one user name and password on z/OS and a different one for UNIX.

Logical name definition (LogicalNameDef)
: corresponds to a logical server name on the SAS Metadata Server.

For each type of definition, a corresponding container interface enables you to manage the definitions. For example, ServerDef objects are managed by using the ServerDefs interface. You can use the Item method to retrieve definitions by name.

Metadata Configuration Files

Metadata configuration files contain information about how to access the SAS Metadata Server.

You can create metadata configuration files in the following ways:

- using the METACON command in SAS
- using the SAS Integration Technologies configuration utility (ITConfig)
- using the WriteConfigFile method

The metadata configuration file contains information about how to access the metadata server and might also contain user and password information for connecting to the

metadata server. ITConfig enables you to create a configuration file without a user name and password, so that users can specify their own credentials.

Use one of the methods previously listed to generate the metadata configuration file. The configuration file can contain the following connection information:

- port
- machine
- encryption
- repository name
- user name
- password
- domain

For more information about using the ITConfig utility to generate metadata configuration files, see the ITConfig utility Help.

For more information about using the METACON command to generate metadata configuration files, see the Help for the Metadata Server Connections window.

For more information about using WriteConfigFile, see the SAS Object Manager reference (sasoman.chm).

SAS Object Manager Error Reporting

Overview of Error Reporting

If a call succeeds in obtaining an object, then the method returns S_OK. However, there are many reasons an error might occur. Sometimes an error might occur even before the connection attempt is made. Here is an example of such an error:

```
Requested logical name was not found
```

Other errors can occur during the connection attempt. Examples include the following:

- **`Invalid userid/password`**
- **`Couldn't connect to a SAS server`**
- **`Invalid hostname`**
- **`Server configuration error`**

Errors can occur even though a successful connection is established. Errors that occur during connection can be reported through the logging mechanism. To enable logging, set the ObjectFactory.LogEnabled property to TRUE. When LogEnabled equals TRUE, both successful and failed connection attempts are recorded in the log. To obtain error information, call GetCreationLog as follows:

```
ObjectFactory.GetCreationLog(erase,allThreads)
```

Here are the parameters for GetCreationLog:

erase
indicates whether to erase all of the accumulated logging information. Values are TRUE or FALSE.

allThreads
indicates the threads for which to obtain connectionAttempts log information. Here are values for the allThreads parameter:

FALSE
Obtain log information for your current thread.

TRUE
Obtain log information for all threads in the process.

Error XML

The error information returned through XML allows applications to fix detected problems. Applications can be used to fix these errors by parsing the XML and possibly providing a user interface or sending a message to an administrator to have the errors fixed.

The following example is an output from GetCreationLog. The first attempt establishes an IOM Bridge connection to the SAS Metadata Server, and the second attempt establishes a COM connection to a workspace server. Note that the sasport and sasmachinednsname are not reported for COM connections.

```
<connectionAttempts>
<connectionAttempt>
<description>Connected.</description>
<status>0x0</status>
<saslogin>myDomain\myLogin</saslogin>
<sasmachinednsname>myMachine</sasmachinednsname>
<sasport>1235</sasport>
<sasclassid>2887E7D7-4780-11D4-879F-00C04F38F0DB</sasclassid>
<sasprogid>SASOMI.OMI.1</sasprogid>
<sasserver>SAS Metadata Server</sasserver>
<threadid>3324</threadid>
<name>OMR</name>
</connectionAttempt>
<connectionAttempt>
<description>Connected.</description>
<status>0x0</status>
<saslogin></saslogin>
<sasmachinednsname>myOtherMachine</sasmachinednsname>
<sasclassid>440196D4-90F0-11D0-9F41-00A024BB830C</sasclassid>
<sasprogid>SAS.Workspace.1</sasprogid>
<sasserver>myOtherMachineCOM - Workspace Server</sasserver>
<threadid>3324</threadid>
</connectionAttempt>
</connectionAttempts>
```

The following error message shows a failed IOM Bridge connection attempt. Note that the port number, machine name, and login are listed for IOM Bridge connections.

```
<connectionAttempts>
<connectionAttempt>
<description>Could not establish a connection to the SAS server on
the requested machine. Verify that the SAS server has been
started with the -objectserver option or that the SAS
spawner has been started. Verify that the port Combridge is
attempting to connect to is the same as the port SAS (or the
spawner) is listening on./description>
```

```
<status>0x8004274d/status>
<saslogin>Username/saslogin>
<sasmachinednsname>machineName/sasmachinednsname>
<sasport>1234/sasport>
<sasclassid>440196D4-90F0-11D0-9F41-00A024BB830C/sasclassid>
<sasprogid>SAS.Workspace.1/sasprogid>
<sasserver>myWorkspace - Workspace Server/sasserver>
<threadid>3324/threadid>
</connectionAttempt>
</connectionAttempts>
```

SAS Object Manager Code Samples

Using CreateObjectByServer to Make a Connection

Overview of Using CreateObjectByServer to Make a Connection

Samples 1 through 5 show how to create a connection by specifying server parameters (machine, protocol, port) directly in the source code.

Samples 1, 2, and 3 show synchronous connections. The synchronous connection is the most frequently used and is the simplest connection to create.

Samples 4 and 5 use asynchronous connections. The advantage of making an asynchronous connection is that the client application can continue to execute code while the connection is being established. An application that interacts with multiple SAS servers might create asynchronous connections to all of the servers at the same time.

When making asynchronous connections, the ObjectKeeper maintains a reference to the created object until you call ObjectKeeper.RemoveObject. This call is not shown in the code for samples 4 and 5.

The object factory adds the newly created object to the ObjectKeeper as soon as the connection is established. You can then use the ObjectKeeper to get the connection.

Sample 1: Make a Connection to a Local Workspace Using COM

```
Dim obObjectFactory As New SASObjectManager.ObjectFactoryMulti2
Dim obSAS As SAS.Workspace
Set obSAS = obObjectFactory.CreateObjectByServer(
"myName", True, Nothing, "", "")
```

Sample 2: Make a Connection to a Remote Workspace Using DCOM

```
Dim obObjectFactory As New SASObjectManager.ObjectFactoryMulti2
Dim obSAS As SAS.Workspace
Dim obServer As New SASObjectManager.ServerDef
obServer.MachineDNSName = "RemoteMachine.company.com"
Set obSAS = obObjectFactory.CreateObjectByServer(
"myName", True, obServer, "", "")
```

Sample 3: Make a Connection to a Remote Workspace Using IOM Bridge

```
Dim obObjectFactory As New SASObjectManager.ObjectFactoryMulti2
```

```
Dim obSAS As SAS.Workspace
Dim obServer As New SASObjectManager.ServerDef
obServer.MachineDNSName = "RemoteMachine.company.com"
obServer.Protocol = ProtocolBridge
obServer.Port = 6903
Set obSAS = obObjectFactory.CreateObjectByServer("myName", True, obServer,
"myUserName", "myPassword")
```

Sample 4: Start Multiple Connections Asynchronously

```
Dim obObjectFactory As New SASObjectManager.ObjectFactoryMulti2
Dim obObjectKeeper As New SASObjectManager.ObjectKeeper
Dim obSAS As SAS.Workspace
Dim obSAS2 As SAS.Workspace
Dim obServer As New SASObjectManager.ServerDef
Dim obServer2 As New SASObjectManager.ServerDef
obServer.MachineDNSName = "MachineA.company.com"
obServer.Protocol = ProtocolBridge
obServer.Port = 6903
obObjectFactory.CreateObjectByServer "myName", False, obServer,
"myUsername", "myPassword"
obServer2.MachineDNSName = "MachineB.company.com"
obServer2.Protocol = ProtocolBridge
obServer2.Port = 6903
obObjectFactory.CreateObjectByServer "myName2", False, obServer2,
"myUsername", "myPassword"
' Note that the first parameter here matches the first parameter in the
' call to CreateObjectByServer
Set obSAS = obObjectKeeper.WaitForObject("myName", 10000)
Set obSAS2 = obObjectKeeper.WaitForObject("myName2", 10000)
```

Sample 5: Listen for Events Using Asynchronous Connections

```
Public WithEvents obObjectKeeperEvents As SASObjectManager.ObjectKeeper
Private Sub Form_Load()
Dim obObjectFactory As New SASObjectManager.ObjectFactoryMulti2
Dim obObjectKeeper As New SASObjectManager.ObjectKeeper
Dim obSAS As SAS.Workspace
Dim obSAS2 As SAS.Workspace
Dim obServer As New SASObjectManager.ServerDef
Dim obServer2 As New SASObjectManager.ServerDef
Set obObjectKeeperEvents = obObjectKeeper
obServer.MachineDNSName = "MachineA.company.com"
obServer.Protocol = ProtocolBridge
obServer.Port = 6903
obObjectFactory.CreateObjectByServer "myName", False, obServer,
"myUsername", "myPassword"
obServer2.MachineDNSName = "Machineb.company.com"
obServer2.Protocol = ProtocolBridge
obServer2.Port = 6903
obObjectFactory.CreateObjectByServer "myName2", False, obServer2,
"myUsername", "myPassword"
End Sub
Private Sub obObjectKeeperEvents_ErrorAdded(ByVal objectName As String,
ByVal errInfo As String)
Debug.Print "Error creating " & objectName & ": " & errInfo
End Sub
```

```
Private Sub obObjectKeeperEvents_ObjectAdded(ByVal objectName As String,
ByVal objectUUID As String)
Debug.Print "Added object " & objectName & ": " & objectUUID
Dim obSAS As SAS.Workspace
Set obSAS = obObjectKeeperEvents.GetObjectByUUID(objectUUID)
Debug.Print obSAS.UniqueIdentifier
End Sub
```

Using CreateObjectByLogicalName to Make a Connection

The CreateObjectByLogicalName method creates a SAS object from a logical name definition. When you use CreateObjectByLogicalName, you must define server metadata for your connections and associate a logical name with the connection. This technique enables you to administratively change the machines where SAS is running without modifying your source code.

The following sample shows how to make a connection to a logical server.

```
Dim obObjectFactory As New SASObjectManager.ObjectFactoryMulti2
Dim obSAS As SAS.Workspace
' This assumes that your metadata configuration files are in the default
' location, or that the location of your configuration files is stored
' in the registry. If this is not the case, or you want each application
' on a given machine to use its own metadata repository, then you should
' call
' obObjectFactory.SetMetadataFile systemFile, userFile, false
Set obSAS = obObjectFactory.CreateObjectByLogicalName("myName", True,
"LogicalName", "LoginReference")
```

Using Connection Pooling

Overview of Pooling

Pooling enables you to create a pool of connections to IOM servers. The connection pool can be shared between multiple connection requests within the same process, and the connections can be reused. Pooling improves the efficiency of connections between clients and servers because clients use the connections only when they need to process a transaction.

When to Use Pooling

Pooling is most useful for applications that require the use of an IOM server for a short period of time. Because pooling reduces the wait that an application incurs when establishing a connection to SAS, pooling can reduce connection times in environments where one or more client applications make frequent but brief requests for IOM services. For example, pooling is useful for web applications, such as Active Server Pages (ASP).

Pooling is least useful for applications that acquire an IOM server and use the server for a long period of time. A pooled connection does not offer any advantage to applications that use connections for an extended period of time.

Note: Pooling can be used only with SAS Workspace Servers.

What Type of Pooling to Use

The SAS Object Manager supports two different pooling mechanisms: SAS Integration Technologies pooling and COM+ pooling. The most noticeable difference between the two mechanisms is the way in which the pools are administered.

For Windows clients, you can choose between SAS Integration Technologies and COM+ pooling. For information, see "Choosing SAS Integration Technologies or COM+ Pooling" on page 55 .

Note: Java applications and COM applications cannot share the same pool, but they can share the administration model that is used for the pools.

Using the Pooled Connection Object

In both COM+ pooling and SAS Integration Technologies pooling, the ObjectManager represents a pooled connection with the PooledObject COM object. The PooledObject COM object has the 'SASObject' property, which holds the interface pointer to the object actually being pooled. It also has the ReturnToPool() method to allow the object to be reused.

When a PooledObject object is obtained, the calling application keeps a reference to the PooledObject object for as long as it needs to use the object. You can use ReturntoPool to release the PooledObject connection and return the associated connection to the pool. Returning the PooledObject connection to the pool also causes the object to be cleaned up so that no further calls can be made on the object.

For more information about the client-side coding for pooling, see the Help for the SAS Object Manager (sasoman.chm).

Using Puddles

Each pool can have any number of puddle objects. The metadata server administrator can partition a pool of connections into several puddles to allow users to have different permissions. For example, one user might be granted access to a puddle that can access summary data sets within SAS. Another executive user might be granted access to a different puddle that can access more detailed data.

Choosing SAS Integration Technologies or COM+ Pooling

The SAS Object Manager supports two different pooling mechanisms: SAS Integration Technologies and COM+ pooling. The main differences between the two pooling mechanisms are as follows:

- the way in which the pools are administered
- the way the programmer makes the call to get the pooled connection (workspace)

COM+ connection pooling has the following limitations:

- COM+ pooling cannot be used on Windows NT.

- COM+ does not allow multiple pooling configurations on the same machine (Windows 2000 only).

You might choose SAS Integration Technologies pooling instead of COM+ if any of the following are true:

- you use Windows NT
- you want to use multiple pools on the same machine
- you want to use the security mechanism that is available in SAS Integration Technologies pooling
- you want to use the same administration model for Java applications and COM applications
- you want to specify the pooling parameters in the source code

Note: Java applications and COM applications cannot share the same pool. However, for SAS Integration Technologies pooling, they can share the administration model that is used for the pools.

Note: You cannot use SAS Integration Technologies and COM+ pooling configurations on the same machine.

For information about setting up SAS Integration Technologies pooling, see "Using SAS Integration Technologies Pooling " on page 56 .

For information about setting up COM+ pooling, see "Using COM+ Pooling " on page 59 .

Using SAS Integration Technologies Pooling

Overview of SAS Integration Technologies Pooling

SAS Integration Technologies pooling uses the SAS Object Manager's implementation as a COM singleton object so that only a single instance of the ObjectFactory component is created in any given process. This mechanism makes the same pools available to all callers in the same process.

Note: You define a pool by a logical name. When using ObjectPools.CreatePoolByLogicalName, a single pool can contain connections from multiple servers and multiple logins. When using ObjectPools.CreatePoolByServer, a pool consists of objects from a single server and a single login.

To implement pooling in your application, perform the following steps:

1. Create the pool (use CreatePoolByServer or CreatePoolByLogicalName). You need to create the pool one time only, usually when the application is started. If you try to create a pool by using a logical name that has already been used in a pool, then you will receive an error.
2. Use GetPooledObject to get a PooledObject object from the pool. The PooledObject is a wrapper around the SASObject that is being pooled. This PooledObject wrapper is necessary to notify the pooling code when you are finished using the pooled workspace. When you use pooling, keep a reference to the PooledObject for as long as you keep a reference to the object.
3. Use the object for processing, such as running a stored process and receiving output from SAS.

4. Use ReturnToPool to return the PooledObject object to the pool so it can be used again.
5. Release the object. In Visual Basic, you can release these objects by either letting them go out of scope or by calling **`set obPooledObject = Nothing`**.

   The pool continues to run until either your process exits or you call Shutdown() on each pool. Releasing your reference to ObjectManager does not release the pool.

For example code, see "Example Code Using SAS Integration Technologies Pooling" on page 61 .

In an ASP application, you can create a pool in one of these two ways:

- in the Application_OnStart callback in the global.asa
- in the code that calls to get the PooledObject

When a pool is running, methods and properties are available on the SASObjectManager.ObjectPool object to look at statistics for the pool and to shut down the pool.

Supplying Pooling Parameters Directly in the Source Code

To specify the pooling parameters in the source code, perform the following steps:

1. Create both a Server object and a Login object.
2. Fill out the relevant properties.
3. Pass the objects to ObjectManager.ObjectPools.CreatePoolByServer.

The following properties are used to configure SAS Integration Technologies pooling. You specify some of the properties on the LoginDef object and some of the properties on the ServerDef object.

ServerDef.MaxPerObjectPool
: specifies the maximum number of servers that should be created from the provided ServerDef object. A good starting place for this number is the number of CPUs that are available on the machine that is running SAS.

ServerDef.RecycleActivationLimit
: specifies the number of times that a workspace is used before the process that it is running in is replaced. You can use this property to cap memory leaks or non-scalable resource usage. A value of 0 means to never recycle the processes.

ServerDef.RunForever
: specifies whether unallocated connections are allowed to remain alive indefinitely. The value for this property must be either TRUE or FALSE. If the value is FALSE, then unallocated live connections will be disconnected after a period of time (specified by the value of ServerDef.ShutdownAfter). If the value is TRUE (the default value), then unallocated live connections remain alive indefinitely.

ServerDef.ShutdownAfter
: specifies the number of minutes that an unallocated live connection will wait to be allocated to a user before shutting down. This property is optional, and it is ignored if the value of ServerDef.ServerRunForever is TRUE. The value must not be less than 0, and it must not be greater than 1440 (the number of minutes in a day). The default value is 3. If the value is 0, then a connection returned to a pool by a user is disconnected immediately unless another user is waiting for a connection from the pool.

LoginDef.MinSize
specifies the minimum number of workspaces for this LoginDef that are created when the pool is created.

LoginDef.MinAvail
specifies the minimum number of available workspaces for this LoginDef. Note that MaxPerObjectPool is never exceeded.

LoginDef.LogicalName
determines which set of LoginDefs in LDAP to use in the pool. This property is used only when a pool is created by using CreatePoolByLogicalName.

For more information about the client-side coding for pooling, see the online Help shipped with the Object Manager.

Administering Pooling

When using a SAS Metadata Server with the SAS Object Manager, you create a pool by specifying a logical name that matches a logical server name on the SAS Metadata Server.

The administrator can associate puddles with the pooled logical server name and administer pooling and puddle parameters by using SAS Management Console. For more information, see the *SAS Intelligence Platform: Application Server Administration Guide*.

The authentication and authorization checking in SAS Integration Technologies pooling enables you to create a pool that contains connections that have been authenticated using different user IDs. This capability allows the access to sensitive data to be controlled on the server machine instead of the middle tier.

Checking is performed only in pools that were created with CreatePoolByLogicalName where the checkCredentialsOnEachGet parameter is set to TRUE.

Authentication is performed by using the user ID and password to authenticate a new connection to a SAS Metadata Server. The pool is searched for a puddle whose access group has the authenticated user as a member.

The GetPooledObject method authenticates the user by performing the following steps:

1. Binds to the SAS Metadata Server by using the credentials that are provided to GetPooledObject.
2. If that bind fails, then GetPooledObject returns an error. If that bind is successful, then it is released and is not used. The bind is connected only to authenticate the credentials. Authorization is then performed against the set of identities in the puddle:
   - If a match is not found, then ERROR_ACCESS_DENIED is returned (0x80004005).
   - Otherwise, a pooled object is returned when one becomes available.

Pooling authentication enables credentials to be used by people who do not have permission to read the credentials directly.

Using COM+ Pooling

Creating a Pooled Object

To obtain a PooledObject object, create a new PooledObject object by using COM. The COM+ interceptors detect the new object and manage the pooled objects. The PooledObject object has been written to support the COM+ pooling mechanism.

For an example, see "Example Code Using COM+ Pooling" on page 61 .

Administering COM+ Pooling

To administer a COM+ pool, use the COM+ Component Services administrative tool, which is a standard Microsoft Management Console (MMC) plug-in that is shipped with Windows 2000.

You can configure COM+ pools in these two different ways:

- library application -- each process has its own pool
- server application -- the pool is shared by all processes on the same machine

To create a new COM+ server application, perform the following steps:

1. Start the Component Services administrative tool by selecting **Start** ⇨ **Programs** ⇨ **Administrative Tools** ⇨ **Component Services**.
2. Expand **Component Services** ⇨ **Computer** ⇨ **COM+ Applications**. Right-click **COM+ Applications** and select **New** ⇨ **Application**. This starts a wizard. Click **Next**.
3. Select **Create an Empty Application**.
4. Enter a name of your choice, and select **Server application** as the Activation Type. Click **Next**.
5. Specify an identity, and click **Next**.
6. Click **Finish**.

To add the PooledObject component to the application, perform the following steps:

1. Expand the application that you previously created in order to see **Components and Roles**.
2. Right-click **Components** and select **New Component**, which brings up a wizard.
3. Click **Next**, and then select **Import components that are already registered**.
4. Select **SASObjectManager.PooledObject.1**, and then click **Next** and **Finish**.

To administer the pooling properties, perform the following steps:

1. Right-click **SASObjectManager.PooledObject.1** under the **Components** node of the application that you created, and select **Properties** from the pop-up menu.
2. Select the **Activation** tab.
3. Select the **Enable object pooling** check box. Enter the properties.

4. (Optional) You can also enter a constructor string, which enables you to specify which machine SAS should run on. For more information, see "Constructor Strings" on page 60 .

   Note: If you do not specify a constructor string, then the SAS Object Manager creates workspaces on the local machine by using COM. It is necessary to configure a metadata server with pooling metadata only if you specify a logicalName.

Constructor Strings

The constructor string is a single string that specifies the parameters that are used to initially create the pool. The object manager requires that the constructor string contain a set of name and value pairs, with the names separated from the values by an equal sign (=), and the pairs separated by a semicolon (;). If no parameters are specified, then a pool that consists of SAS servers running on the local machine is created. You should never use quotation marks (") in the constructor string. The constructor string contains the only attributes that are specific to SAS.

If errors occur when creating a workspace, then the object manager writes entries to the Event Log.

Constructor Parameters Used to Connect with Metadata

The following parameters are required when you use a constructor string to connect to SAS with metadata:

logicalName
: specifies which sasServer objects to use when creating objects in the pool.

referenceDN
: specifies the login to use for authentication. This parameter is necessary only for an IOM Bridge connection. For the SAS Metadata Server, specifying only the logicalName parameter sets the value of referenceDN as the identity of the user who is specified in the metadata configuration file.

Here is an example of a valid constructor string:

```
logicalname=pooltest;referencedn=A5YEODSG.AE00005L
```

Constructor Parameters Used to Connect without Metadata

The following parameters are required when you use a constructor string to connect to SAS without metadata:

classIdentifier
: specifies the class ID number. For example, **2887E7D7-4780-11D4-879F-00C04F38F0DB** specifies a SAS Metadata Server. The default value is **440196D4-90F0-11D0-9F41-00A024BB830C**, which specifies a SAS Workspace Server.

machineDNSName
: specifies the name of the machine to connect to.

protocol
: can be either com or bridge.

port
: specifies the port number of a server to connect to. This parameter should be specified only for an IOM Bridge connection.

serviceName
: used to resolve a TCP/IP service to a port number. This parameter should be specified only for an IOM Bridge connection. Only one of the port or serviceName parameters should be specified.

loginName
: specifies the user ID to use when connecting to a SAS server. This parameter should be specified only for an IOM Bridge connection.

password
: defines the password that authenticates the loginName. This parameter should be specified only for an IOM Bridge connection.

Pooling Samples

Example Code Using COM+ Pooling

```
' Create the pooled object
Dim obPooledObject As New SASObjectManager.PooledObject
Dim obSAS As SAS.Workspace
Set obSAS = obPooledObject.SASObject
' Test the object
Debug.Print obSAS.Utilities.HostSystem.DNSName
' Return the object to the pool
set obSAS=Nothing
obPooledObject.ReturnToPool
```

Example Code Using SAS Integration Technologies Pooling

```
Set obServer = CreateObject("SASObjectManager.ServerDef")
Set obLogin = CreateObject("SASObjectManager.LoginDef")
Set obObjectFactory = CreateObject("SASObjectManager.ObjectFactory")
' Define a ServerDef and LoginDef for the pool
obServer.MachineDNSName = "localhost"
' For VBScript, set obServer.Protocol to 2 instead of ProtocolBridge
obServer.Protocol = ProtocolBridge
obServer.Port = 8591
obServer.ShutdownAfter = 1
obServer.RunForever = False
obServer.RecycleActivationLimit = 10000
obLogin.LoginName = "mydomain\myuserid"
obLogin.Password = "myPassword"
' Create the pool
Set obPool = obObjectFactory.ObjectPools.CreatePoolByServer(
"myPool", obServer, obLogin)
' Create a pooled object
Set obPooledWorkspace = obPool.GetPooledObject("", "", 10000)
Set obSAS = obPooledWorkspace.SASObject
' Test the object
```

```
Debug.Print obSAS.Utilities.HostSystem.DNSName
' Return the object to the pool
Set obSAS = Nothing
obPooledWorkspace.ReturnToPool
Set obPooledWorkspace = Nothing
```

Chapter 4

Using the SAS IOM Data Provider

Using the SAS IOM Data Provider

The SAS IOM Data Provider is an OLE DB data provider that supports access to SAS data sets that are managed by SAS Integrated Object Model (IOM) servers. Although an object server is a scriptable interface to SAS, manipulating data in that server context is best accomplished with the IOM Data Provider.

OLE DB is a set of interfaces that evolved from the Microsoft Open Database Connectivity (ODBC) data access interface. OLE DB interfaces provide a standard by which applications can uniformly access data that is located over an enterprise's entire network and stored in a variety of formats (such as SAS data sets, database files, and nonrelational data stores). OLE DB interfaces can provide much of the same functionality that is provided by database management systems. In addition, OLE DB for Online Analytical Processing (OLAP) extends the OLE DB interfaces to provide support for multidimensional data stores.

Through the OLE DB interfaces, the IOM Data Provider adds the following functionality to consumer programs:

- simultaneous user updates
- SQL processing
- a choice of either exclusive access (member-level lock) or multiple user access (record-level lock) to SAS data files, selectable on a per-rowset open basis
- access to SAS data files on SAS 8 and later SAS IOM servers
- integrated SAS formatting services, which are the included core set of SAS formats that are used when reading or modifying data
- use of basic OLE DB schema rowsets, which enable consumers to obtain metadata about the data source that they use
- support for random access by using the ADO adOpenDynamic cursor type and recordset bookmarks

OLE DB data providers, including the IOM Data Provider, can be accessed through the low-level OLE DB interfaces by using Visual C++. Alternatively, they can be accessed through the higher-level ActiveX Data Objects (ADO) using Visual C++, VBScript, Java, JScript, or Visual Basic. In .NET programming languages like C# and VB.NET,

the IOM Data Provider can be used via the ADO.NET OLE DB data provider. For examples using ADO.NET, see "Accessing SAS Data with ADO.NET" on page 33 .

The IOM Data Provider is documented in the *SAS Providers for OLE DB: Cookbook*. This cookbook, which applies to all four SAS Data Providers, contains ADO examples written in Microsoft Visual Basic as well as OLE DB examples written in Microsoft Visual C++. You can either apply the examples directly or modify them to fit your needs.

Note: When you create an IOM Data Provider connection, you specify the unique identifier of a workspace object. The IOM Data Provider can locate the specified workspace only if it is stored in either a WorkspaceManager object or an ObjectKeeper object. Depending on the interface that you use, store your workspace as follows:

- If you use the object manager, then create an ObjectKeeper object and add your workspace to it manually by using the ObjectKeeper's **`AddObject`** function. When you have finished using the IOM Data Provider, remove the workspace by using the **`RemoveObject`** function.
- If you use the workspace manager, then specify **`VisibilityProcess`** when you create your workspace. When you specify this attribute, the workspace manager stores the workspace automatically. When you have finished using the IOM Data Provider, remove the workspace by using the **`RemoveWorkspace`** function.

Chapter 5

Using the Workspace Manager

Using the Workspace Manager

Note: It is recommended that you use the Object Manager interface in order to use the new features that are available with SAS®9 SAS Integration Technologies. The Workspace Manager is provided for legacy programs.

The SAS IOM components are arranged in a hierarchy, and the root of the hierarchy is a workspace object. A workspace represents a session with the SAS System, and is functionally equivalent to a SAS Display Manager session or the execution of Base SAS software in a batch job.

The SAS Workspace Manager is a component that executes on the client machine, and it is used to create and manage workspaces on IOM servers. The Workspace Manager uses IOM server definitions that are administered separately from the application. This

enables, for example, a client application to connect to a server simply by using a logical name. The definition for this server can change as required without affecting the application.

The Workspace Manager can create a workspace in one of these three ways:

- through local COM if the SAS Server runs on the same machine as the client
- through DCOM if the SAS Server runs on another machine that supports DCOM
- through the IOM Bridge for COM (SAScomb.dll) if the SAS Server runs on another machine that does not support COM/DCOM functionality (UNIX [Solaris, HP-UX, AIX] or z/OS)

With the Workspace Manager, you can perform the following tasks:

- launch workspaces
- select between running workspaces
- share IWorkspace pointers within a process
- access a workspace from within web pages
- use ADO within a workspace
- store and retrieve definitions by using a flat file

The Workspace Manager can be used from Visual Basic, C, C++, and VBScript (with the help of Scripto).

Launching IOM Servers

Definitions

You can specify parameters for the definitions in either of these two locations:

- source code
- an LDAP Data Interchange Format (LDIF) file

The following three definitions can be created to assist in launching an IOM Server:

Server definition (ServerDef)
: must be created before an IOM Server can be launched with the workspace manager. The server definition can either be loaded from persistent storage (an LDIF file), or created dynamically. A ServerDef includes a Logical Name attribute. The server definition is independent of the user.

Login definition (LoginDef)
: contains user-specific information such as a user name and password. Login definitions are a convenience and are not required for creating a connection to an IOM server. They provide a mechanism for storing persistent definitions of user names and passwords. LoginDefs also allow multiple definitions for the same user on different security domains. For example, you could use one user name and password on MVS and a different one for UNIX. This flexibility is also possible without the use of a login definition, but the user must enter the user name and password each time a server is launched.

Logical name definition (LogicalNameDef)
: allows a description to be associated with each logical name used in a server definition. Logical name definitions are not used to launch a server. However, a logical name is required to launch a server when using the login definition.

These definitions can be stored in a file on the local system.

Finding Definitions

The SAS Workspace Manager can access status information that is stored in the Windows registry. This status information consists of two search specifications for finding launch information:

- a per-user file for local computer storage
- a file that is shared for all users of the local machine

This information is designed to make it easy to find launch information in a standard location. Also, applications can immediately list launch definitions that are found in the standard location.

LDIF File

The LDIF format is the standard for the interchange of LDAP data. The SAS Workspace Manager has the ability to read Login and Server definitions from LDIF files.

LDIF specifies that each object definition start with a distinguished name (DN), but the full DN of any object is not known. The part that is not known is replaced with $SUFFIX$ to allow administrators to use an automated search and replace mechanism should they want to import a file into LDAP.

This format is also supported by the object spawner.

Security Considerations

The user ID that is used to log on to SAS will be determined when the workspace is launched. Once launched, the user ID cannot be changed.

Stored passwords are not encrypted.

The information in a file is restricted only by the permissions on the file. If you are concerned about security, then you might not want to use files to store LoginDefs.

Administering the SAS Workspace Manager

For the SAS Workspace Manager, administration tasks include creating and saving LoginDefs, ServerDefs, and LogicalNameDefs. There are several alternatives for the administration of definitions. Definitions can be administrated directly through the IWorkspaceManager interfaces. These interfaces can write to LDIF files. Note that the workspace manager will read or write only the attributes that are of interest to it. If you are using files, then any existing definitions or attributes that are not used by the workspace manager are deleted, so caution should be used. Additional attributes that are defined for these objects are ignored.

A text editor is used to modify the LDIF files.

Note that it is not necessary to perform any administration at all. Definitions can be created, used, then discarded without ever being persisted.

Workspace Manager Interfaces

Overview of Workspace Manager Interfaces

The principal interfaces of the workspace manager are as follows:

IWorkspaceManager
: provides collection methods for ServerDefs, LoginDefs, LogicalNameDefs, SAS Workspaces, and WorkspacePools. IWorkspaceManager is the top level interface.

IWorkspaces
: is used to create, enumerate, locate, and remove running workspace instances. The only workspaces that can be manipulated through IWorkspaces are those that are created from IWorkspaces.

ILoginDef
: enables you to create and manipulate login definitions (LoginDefs). A loginDef is needed only for connections that use the IOM Bridge for COM. SSPI, the default NT security, is used for COM and DCOM connections.

ILoginDefs
: contains the standard collection methods Count, _NewEnum, Add, Item, and Remove where the key is the loginDefName. It also supports one additional method: CheckAccess.

IServerDef
: determines how to connect to the server. For a local or DCOM server, only the Name and Hostname values need be filled out. The IOM Bridge for COM requires Protocol to be set to ProtocolBridge. Port and ServiceName can also be used with the IOM Bridge for COM.

IServerDefs
: contains the standard collection methods Count, _NewEnum, Add, Item, and Remove, where the key is the serverDefName. It also supports one additional method: CheckAccess.

IWorkspacePools
: creates, enumerates, locates, and removes WorkspacePool objects.

IWorkspacePool
: configures parameters for a WorkspacePool.

IPooledWorkspace
: notifies a pool when the associated workspace can be returned to the pool.

Creating a Workspace

CreateWorkspaceByLogicalName Method

This method creates a SAS Workspace from a logical name definition. Here is the signature for this method:

```
CreateWorkspaceByLogicalName([in]BSTR name,
[in] enum Visibility visibility, [in]BSTR logicalName,
```

```
[in]BSTR referenceDN, [out]BSTR *xmlInfo, [out, retval]IWorkspace
**newWorkspace)
```

The CreateWorkspaceByLogicalName method creates a new workspace by performing the following steps:

1. Creates a list of all ServerDefs that define the provided logicalName.
2. Selects the first serverDef in the list.
3. Locates a LoginDef that matches both the DomainName (from the ServerDef) and the provided referenceDN.
4. Attempts to create a connection, once for each MachineDNSName in the serverDef, adding the results (either success or failure) into the xmlInfo.

   If a connection to SAS could not be established, then the next serverDef is tried and the process is repeated from step 3.

   If no Workspace has been created after going through all of the host names in each serverDef and all of the serverDefs that match the given logicalName, then an error is returned (SEE_XML). Also, the table of connectionAttempts is returned in Err.Description (and if IWorkspaces.UseXMLInErrorInfo is true).

   If a Workspace is created, then the IWorkspace is added to an internal list in the WorkspaceManager. In this case, the errorString will still show all failed attempts, if any.

The name that is passed to the method can be used in other IWorkspaces methods to determine which workspace to locate or remove. The name is also set on IWorkspace->Name before this method returns. The SAS Workspace Manager does not ever look at IWorkspace->Name, so a client could change the name after calling Create. The xmlInfo parameter is defined only when this method returns an IWorkspace. For more information, see “Error Reporting” on page 70 .

Note: The workspace manager should be notified when a workspace is no longer in use so that the workspace can be removed from the internal list and the reference that the workspace manager holds on the Workspace can be released.

CreateWorkspaceByServer Method

This method creates a workspace from a ServerDef object instead of a logical name. It also accepts the actual user name and password instead of a reference to a LoginDef. This method is preferred over CreateWorkspaceByLogicalName since it does not require user name and password pairs to be written to a file or a network directory.

Here is the signature for this method:

```
CreateWorkspaceByServer([in]BSTR workspaceName, [in] enum
SASWorkspaceManagerVisibility visibility, [in]IServerDef *serverDef,
[in]BSTR userName, [in]BSTR password, [out]BSTR *xmlInfo,
[out, retval]IWorkspace **newWorkspace)
```

This method attempts to connect to each of the hosts listed in the provided serverDef, one at a time, until either a successful connection is made or all hosts have failed. The user name and password are not required for serverDefs that specify ProtocolCom.

Error Reporting

If the call succeeds in obtaining a workspace, then the method returns S_OK. However, there are many reasons an error might occur. Sometimes an error can occur even before the connection attempt is made. Here is an example of such an error:

```
Requested logical name was not found
```

Other errors can occur during the connection attempt. Here are some examples of other errors:

- Invalid user ID/password
- Could not connect to a SAS server
- Invalid host name
- Server configuration error

Errors can occur even though a successful connection is established. Errors that occur when connecting are reported either through the IErrorInfo mechanism (if no SAS workspace is created) or returned in the errorString parameter (if a SAS workspace is created). The Err.Description, IErrorInfo->Description() in C, and errorString both use the same XML format.

If UseXMLInErrorInfo is set to false (it defaults to true), then Err.Description contains only a single error string, which refers to the last connection attempted.

The error information that is returned through XML allows applications to fix the problems that are detected. Here is an example error, in which a successful connection is made, but the first attempt failed:

```
<connectionAttempts>
<connectionAttempt>
<sasservername>MyServer</sasservername>
<machineDNSName>MyServer.MyCompany.com</machineDNSName>
<saslogin>MyUserID</saslogin>
<status>0x8004274d</status>
<description>Could not establish a connection to the SAS
server on the requested machine. Verify that the SAS server has
been started with the -objectserver option or that the SAS
spawner has been started. Verify that the port Combridge is
attempting to connect to is the same as the port SAS (or the
spawner) is listening on.</description>
</connectionAttempt>
<connectionAttempt>
<sasserver>MyServer2</sasserver>
<machineDNSName>MyServer2.MyCompany.com</machineDNSName>
<saslogin>MyUserID</saslogin>
<status>0x0</status>
<description>Connected</description>
</connectionAttempt>
</connectionAttempts>
```

Using Workspace Pooling

Overview of Workspace Pooling with the Workspace Manager

Workspace pooling enables you to create a pool of connections to IOM servers that are shared and reused among multiple client applications. This feature can dramatically reduce connection times in environments where one or more client applications make frequent but brief requests for IOM services. This is typically the case, for example, in Active Server Pages (ASP) applications. Pooling reduces the wait that an application incurs when establishing a connection to SAS and is, therefore, most efficient when used in such applications. A pooled workspace does not offer significant advantages in applications that use workspaces for an extended period of time.

The Workspace Manager supports two different pooling mechanisms: COM+ pooling and SAS Integration Technologies pooling. The most noticeable difference between the two mechanisms is the way in which the pools are administered.

In both COM+ pooling and SAS Integration Technologies pooling, the workspace manager represents a pooled workspace with the PooledWorkspace COM object. This object has a single property: workspace. When PooledWorkspace object is obtained, the calling application keeps a reference to the PooledWorkspace object for as long as it needs to use the workspace. Releasing the PooledWorkspace object returns the connection to the pool and causes the workspace to be cleaned up. It is impossible to make new calls to the workspace after the associated PooledWorkspace is released.

COM+ Pooling

Creating a Pooled Object

To obtain a PooledWorkspace object, creates a new PooledWorkspace object by using COM. The COM+ interceptors detect the new object and manage the pooled objects. The PooledWorkspace object has been written to support the COM+ pooling mechanism.

For an example, see “Example Code Using COM+ Pooling” on page 77 .

Administering COM+ Pooling

To administer a COM+ pool, use the COM+ Component Services administrative tool, which is a standard Microsoft Management Console (MMC) plug-in that is shipped with Windows 2000.

You can configure COM+ pools in these two different ways:

- library application — each process has its own pool
- server application — the pool is shared by all processes on the same machine

To create a new COM+ server application, perform the following steps:

1. Start the Component Services administrative tool by selecting **Start** ⇨ **Programs** ⇨ **Administrative Tools** ⇨ **Component Services**.
2. Expand **Component Services** ⇨ **Computer** ⇨ **COM+ Applications**. Right-click **COM+ Applications** and select **New** ⇨ **Application**. This starts a wizard. Click **Next**.
3. Select **Create an Empty Application**.

4. Enter a name of your choice, and select **Server application** as the Activation Type. Click **Next**.
5. Specify an identity, and click **Next**.
6. Click **Finish**.

To add the PooledWorkspace component to the application, perform the following steps:

1. Expand the application that you previously created in order to see **Components and Roles**.
2. Right-click **Components** and select **New Component**, which brings up a wizard.
3. Click **Next**, and then select **Import components that are already registered**.
4. Select **SASWorkspaceManager.PooledWorkspace.1**, and then click **Next** and **Finish**.

To administer the pooling properties, perform the following steps:

1. Right-click **SASWorkspaceManager.PooledWorkspace.1** under the **Components** node of the application that you created, and select **Properties** from the pop-up menu.
2. Select the **Activation** tab.
3. Select the **Enable object pooling** check box. Enter the properties.
4. (Optional) You can also enter a constructor string, which enables you to specify which machine SAS should run on. For more information, see "Constructor Strings" on page 72 .

Constructor Strings

The constructor string is a single string that specifies the parameters that are used to initially create the pool. The workspace manager requires that the constructor string contain a set of name and value pairs, with the names separated from the values by an equal sign (=), and the pairs separated by a semicolon (;). If no parameters are specified, then a pool that consists of SAS servers running on the local machine is created. You should never use quotation marks (") in the constructor string. The constructor string contains the only attributes that are specific to SAS.

If errors occur when creating a workspace, then the workspace manager writes entries to the Event Log.

Constructor Parameters

The following parameters are required when you use a constructor string:

logicalName
specifies which sasServer objects to use when creating objects in the pool.

referenceDN
specifies the login to use for authentication. This parameter is necessary only for an IOM Bridge connection. For the

machineDNSName
specifies the name of the machine to connect to.

protocol
can be either com or bridge.

port
specifies the port number of a server to connect to. This parameter should be specified only for an IOM Bridge connection.

serviceName
used to resolve a TCP/IP service to a port number. This should be specified only for an IOM Bridge connection. Only one of the port or serviceName parameters should be specified.

loginName
specifies the user ID to use when connecting to a SAS server. This should be specified only for an IOM Bridge connection.

password
defines the password that authenticates the loginName. This should be specified only for an IOM Bridge connection.

Here is an example of a valid constructor string:

```
logicalname=pooltest;protocol=com
```

SAS Integration Technologies Pooling

Choosing SAS Integration Technologies Pooling

You might choose SAS Integration Technologies pooling instead of COM+ pooling if any of the following are true:

- COM+ is not available
- you want multiple pools on the same machine
- you want to use the security mechanism available in SAS Integration Technologies pooling
- you want to use the same administration model for Java applications and COM applications
- you want to specify the pooling parameters in the source code

Java applications and COM applications cannot share the same pool, but they can share the administration model used for the pools.

A pool is defined by a logical name. When using WorkspacePools.CreatePoolByLogicalName, a single pool can contain Workspaces from multiple Servers and multiple logins. When using WorkspacePools.CreatePoolByServer, a pool will consist of Workspaces from a single Server and a single Login.

The authentication and authorization checking in SAS Integration Technologies pooling enables you to create a pool that contains SAS Workspaces that have been authenticated by using different user IDs. This capability allowing the access to sensitive data to be controlled on the server machine instead of the middle tier. Checking is performed only in pools that were created with CreatePoolByLogicalName where the checkCredentialsOnEachGet parameter is set to true.

Using SAS Integration Technologies Pooling

The Pooling takes advantage of the WorkspaceManager being a COM singleton object (which means that only a single instance of the WorkspaceManager component is created in any given process). In other words, every time a call to CoCreateInstance (which is what Visual Basic's CreateObject and dim x as New object use) is made in the

same process, the same WorkspaceManager object is returned. Therefore, the same pools are available to all callers in the same process.

To create and use a pool of workspaces, perform the following steps:

1. Create the Pool (use CreatePoolByServer or CreatePoolByLogicalName). This step needs to be performed only once, usually when the application is started. You will get an error if you try to create a pool using a logical name that has already been used in a pool.
2. Get a PooledWorkspace object from the pool (use GetPooledWorkspace). The PooledWorkspace is really just a wrapper around the SAS.Workspace object that is being pooled. This wrapper is necessary to notify the pooling code when you are finished using the pooled workspace. In pooling, you will want to keep a reference to the pooled workspace for as long as you keep a reference to the workspace.
3. Use the workspace. For example, you might run a stored process and retrieve the output from SAS.
4. Release the workspace and the PooledWorkspace. In Visual Basic, you can release these objects by either letting them go out of scope or by calling:

   ```
   set obPooledWorkspace = Nothing
   ```

   The pool will continue to run until either your process exits or you call Shutdown() on each pool. Simply releasing your reference to the WorkspaceManager is not enough.

In an ASP application, there are two ways the pool can be created: in the Application_OnStart callback in the global.asa, or in the code that calls to get the PooledWorkspace. The following code is written in Visual Basic; some changes are needed to write it in VBScript.

Once a pool is running, methods and properties are available on the SASWorkspaceManager.WorkspacePool object to look at statistics about the pool to and shut down the pool.

For an example, see “Example Code Using SAS Integration Technologies Pooling” on page 78 .

Administration

Pool parameters can be specified in these two places:

- in an LDIF-formatted file. You can edit the file in a text editor. The WorkspaceManager.Scope property must be either ScopeLocal or ScopeUser. The location of the file is determined by WorkspaceManager.FilePath.
- in source code. Create both a sasServer object and a sasLogin object, and then fill out the relevant properties. Pass those properties to WorkspaceManager.WorkspacePools.CreatePoolByServer. The following line of code specifies the maximum number of workspaces that can be created in a pool:

  ```
  obServer.MaxPerWorkspacePool = 6
  ```

The following parameters are used to configure SAS Integration Technologies pooling. The parameters are specified in either the serverDef or loginDef object.

ServerDef.MaxPerWorkspacePool
: specifies the maximum number of servers that should be created from the provided ServerDef object. A good starting place for this number is the number of CPUs that are available on the machine that is running SAS.

ServerDef.RecycleActivationLimit
: specifies the number of times that a workspace is used before the process that it is running in is replaced. You can use this parameter to cap memory leaks or non-scalable resource usage. A value of 0 means to never recycle the processes.

ServerDef.RunForever
: specifies whether unallocated server connections remain alive indefinitely. The value must be either true or false. If the value is false, then unallocated live connections will be disconnected after a period of time (specified by the value of ServerDef.ShutdownAfter). If the value is true (the default value), then unallocated live connections remain alive indefinitely. This property is optional.

ServerDef.ServerShutdownAfter
: specifies the number of minutes that an unallocated live connection will wait to be allocated to a user before shutting down. This property is optional, and it is ignored if the value of ServerDef.ServerRunForever is true. The value must not be less than 0, and it must not be greater than 1440 (the number of minutes in a day). The default value is 3. If the value is 0, then a connection returned to a pool by a user is disconnected immediately unless another user is waiting for a connection from the pool.

LoginDef.AllowedClientDN
: is a variant property that can contain a single string or an array of strings. The DNs can be specific UserDNs or Groups. This is used to specify who is allowed to access a workspace that is created with the associated user name and password.

LoginDef.MinSize
: specifies the minimum number of workspaces using this LoginDef that are created when the pool is created.

LoginDef.MinAvail
: specifies the minimum number of available workspaces for this LoginDef. Note that MaxPerWorkspacePool is never be exceeded.

LoginDef.LogicalName
: determines which set of LoginDefs in LDAP to use in the Pool. This parameter is used only when a pool is created by using CreatePoolByLogicalName.

Code Samples

Using Local SAS with ADO

```
Dim obSAS As SAS.Workspace
Dim obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager
Private Sub Form_Load()
Dim obConnection As New ADODB.Connection
Dim obRecordSet As New ADODB.Recordset
Dim errorString As String
Set obSAS = obWorkspaceManager.Workspaces.CreateWorkspaceByServer(
"MyWorkspaceName", VisibilityProcess, Nothing, "", "",
errorString)
obSAS.LanguageService.Submit "data a; x=1; y=100; output; x=2;
y=200; output; run;"
obConnection.Open "provider=sas.iomprovider.1; SAS Workspace ID="
+ obSAS.UniqueIdentifier
```

```
obRecordSet.Open "work.a", obConnection, adOpenStatic,
adLockReadOnly, adCmdTableDirect
obRecordSet.MoveFirst
Debug.Print obRecordSet(1).Value
End Sub
Private Sub Form_Unload(Cancel As Integer) ' If we don't close SAS, the
' SAS process might run forever
If Not (obSAS is Nothing) Then
obWorkspaceManager.Workspaces.RemoveWorkspace obSAS
obSAS.Close
End If
End Sub
```

Using Remote SAS with ADO and No Persisted ServerDefs or LoginDefs

```
Dim obSAS As SAS.Workspace
Dim obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager
Private Sub Form_Load()
Dim obConnection As New ADODB.Connection
Dim obRecordSet As New ADODB.Recordset
Dim obServerDef As New SASWorkspaceManager.ServerDef
Dim xmlString As String
obServerDef.Port = ObjectServerPort
obServerDef.Protocol = ProtocolBridge ' Multiple hostNames can be provided
' for failover; order is unreliable.
obServerDef.MachineDNSName = "myServer.myCompany.com"
Set obSAS = obWorkspaceManager.Workspaces.CreateWorkspaceByServer(
"MyWorkspaceName", VisibilityProcess, obServerDef, "MyUserID",
"MyPassword", xmlString)
obSAS.LanguageService.Submit "data a; x=1; y=100; output; x=2; y=200;
output; run;"
obConnection.Open "provider=sas.iomprovider.1; SAS Workspace ID="
+ obSAS.UniqueIdentifier
obRecordSet.Open "work.a", obConnection, adOpenStatic, adLockReadOnly,
adCmdTableDirect
obRecordSet.MoveFirst
Debug.Print obRecordSet(1).Value
End Sub
Private Sub Form_Unload(Cancel As Integer)
If not (obSAS is Nothing) Then
obWorkspaceManager.Workspaces.RemoveWorkspace obSAS
obSAS.Close
End If
End Sub
```

Parsing the Returned XML

This example illustrates how to parse the XML that is returned when an error occurs:

```
Dim obSAS As SAS.Workspace
Dim obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager
Private Sub Form_Load()
Dim errorXML As String
On Error GoTo CREATEERROR
```

```
' Establish File parameters
Set obSAS = obWorkspaceManager.Workspaces.CreateWorkspaceByLogicalName(
"MyWorkspaceName", VisibilityProcess, "", "", errorXML)
' errorXML only gets returned when there is a successful connection,
' otherwise err.description gets the XML
If (Len(errorXML) > 0) Then ParseXML errorXML
CREATEERROR:
If (Err.Number > SASWorkspaceManager.Errors.SEE_XML) Then
GoTo TROUBLE
End If
DISPLAYXML:
ParseXML Err.Description
GoTo NOTROUBLE
TROUBLE:
Debug.Print Str(Err.Number) + ": " + Err.Source + ": " + Err.Description
NOTROUBLE:
End Sub
Private Sub ParseXML(xml As String)
' Another option would be to create some XML to make this look nice.
' Write the XML to a file so it can be parsed
' Form2 simply has a WebBrowser control in it
Dim f As New Scripting.FileSystemObject
Dim fname As String
Dim tstream As Scripting.TextStream
fname = f.BuildPath(f.GetSpecialFolder(TemporaryFolder), f.GetTempName)
Set tstream = f.OpenTextFile(fname, ForWriting, True)
tstream.Write ("html>body>xml id=""combridgeOutput"">")
tstream.Write (xml)
tstream.Write ("/xml>table datasrc=""#combridgeOutput"">thead>
th>sasServer/th>th>sasLogin/th>")
tstream.Write ("th colspan=50>machineDNSName/th>th colspan=20>
port/th>th colspan=40>status/th>")
tstream.Write ("th colspan=200>description/th>/thead>tbody>tr>
td>span datafld=""sasserver"">")
tstream.Write ("/span>/td>td>span
datafld=""saslogin"">/span>/td>
td colspan=50>")
tstream.Write ("span datafld=""sasmachinednsname"">/span>/td>
td colspan=50>span datafld=""sasport"">")
tstream.Write ("/span>/td>td colspan=40>span datafld=""status"">
/span>/td>td colspan=200>")
tstream.Write ("span
datafld=""description"">/span>/td>/tr>/tbody>
/table>/body>/html>")
tstream.Close
Form2.WebBrowser1.Navigate "file://" + fname
Form2.Show
End Sub
```

Example Code Using COM+ Pooling

This example illustrates how to use the COM+ pooling feature:

```
' Either of these lines will create a PooledWorkspace using COM+
dim obPooledWorkspace as new SASWorkspaceManager.PooledWorkspace
set obPooledWorkspace =
```

```
CreateObject("SASWorkspaceManager.PooledWorkspace")
' Note that from this point on, the code is the same in both the COM+
' and the Integration Technologies pooling.
Dim obSAS as SAS.Workspace
Set obSAS = obPooledWorkspace.Workspace
' Now use obSAS.
Debug.Print obSAS.UniqueIdentifier
'When done, release the obPooledWorkspace to return it to the pool.
Set obPooledWorkspace = Nothing
```

Example Code Using SAS Integration Technologies Pooling

This example illustrates how to use the SAS Integration Technologies pooling feature:

```
' Get the pool. If it doesn't exist, create it
Dim obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager
Dim obPool As SASWorkspaceManager.WorkspacePool
On Error Resume Next
' Assume the pool already exists
Set obPool = obWorkspaceManager.WorkspacePools("MyLogical")
On Error GoTo 0
If (obPool Is Nothing) Then
' The pool doesn't exist; create it
Dim obServer As New SASWorkspaceManager.ServerDef
Dim obLogin As New SASWorkspaceManager.LoginDef
obServer.Protocol = ProtocolBridge
obServer.Port = 4321 ' A random number
obServer.MachineDNSName = "MyMachine.MyCompany.com"
obLogin.LoginName = "MyLogin"
obLogin.Password = "MyPassword"
Set obPool = obWorkspaceManager.WorkspacePools.CreatePoolByServer(
"MyLogical", obServer, obLogin)
End If
' Now the pool is created, get a Workspace from it:
Dim obPooledWorkspace As SASWorkspaceManager.PooledWorkspace
set obPooledWorkspace = obPool.GetPooledWorkspace("", "", 1000)
' Note that from this point on, the code is the same in both the COM+
' and the Integration Technologies pooling.
Dim obSAS As SAS.Workspace
Set obSAS = obPooledWorkspace.Workspace
' Now use obSAS.
Debug.Print obSAS.UniqueIdentifier
'When done, release the obPooledWorkspace to return it to the pool.
Set obPooledWorkspace = Nothing
```

Index

P

R

S

T

V

W

X