WMI Consumers

Consumer Application Framework

Chapter 7 of “Developing WMI Solutions” covers how developers can use C++ and COM to access the WMI management environment. This client consumer framework makes it easier to access the management environment including high-performance WMI classes.

Getting started

Discussion of this application framework assumes knowledge of C++ and the WMI consumer APIs. For more information about writing WMI management applications, read Chapter 7. The following guide will look at a few examples to demonstrate how the user this framework.

Projects that want to use this framework must include WMIConsumer.h. The framework works for both Windows 2000 and Windows XP & Server 2003. If the project is to run on Windows 2000, you need to `#define` the symbol, `_AFX_WMICONSUMER_WIN2K`, before including the WMIConsumer.h. In addition, you will need to include Platform SDK files into your project. Check the Windows 2000 client sample for more details.

The version of the framework for Windows 2000 uses the Platform SDK provided code for parsing object paths. The Windows XP and Server 2003 version uses the Operating System provided object path parser and query parser.

Connecting to WMI

Connecting to WMI is easy. Simply create an object of class `CWbemLocator` and call `Connect`. By default, `Connect` will open the root\CIMV2 namespace on the current machine. For example:

```
CWbemLocator wmi;
wmi.Connect();
```

The following example connects to the root\WMIBook namespace on the current machine.

```
CWbemLocator wmibook;
wmibook.Connect(L".", L"root\\WMIBook");
```

The `CWbemLocator` class is discussed later in the “Framework public classes” section.

Enumerating management objects

Enumerating management objects is possibly one of the most common operations. Use the class, `CWbemInstanceList`, and call `Create` to retrieve the enumeration for the management class. Here’s how to do this:

```
printf("\nEnumerating Windows services\n");
printf("%-40ls\t%-12ls\t%ls\n", L"Service name", L"Process ID", L"State");
printf("------------------------------------------------------------------\n");

CWbemInstanceList listServices(wmi);
listServices.Create(L"Win32_Service");
```

```
CWbemInstanceList::iterator itrService = listServices.begin();
for ( ; itrService != listServices.end(); itrService++)
{
   CComBSTR bstrName = BSTR(itrService->GetProperty(L"Name"));
   int nProcessID = int(itrService->GetProperty(L"ProcessId"));
   bool bStarted = bool(itrService->GetProperty(L"Started"));

   printf("%-40ls\t%-12d\t%ls\n", bstrName, nProcessID,
      bStarted ? L"Started" : L"--");
}
```

Pass the `CWbemLocator` WMI connection to the constructor of the `CWbemInstanceList` class and call `Create` passing in a class reference. Then use the STL list iterators to obtain access to each management object. Each management object is encapsulated within the class, `CWbemInstance`. This class has many useful methods including property access methods such as `GetProperty`.

The above code outputs the following:

```
Enumerating Windows services
Service name                                    Process ID      State
-------------------------------------------------------------------------------

Albd                                            1256            Started
Alerter                                         0               --
ALG                                             0               --
AppMgmt                                         800             Started
aspnet_state                                    0               --
AudioSrv                                        800             Started
AvSynMgr                                        1268            Started
BITS                                            800             Started
Browser                                         800             Started
... .. ...
.. .
```

Querying the management environment

The ability to query the management environment is a key benefit. It allows you to gain access to a set of management objects that you specifically want. Use the class, `CWbemInstanceList`, and call `Query` to retrieve the enumeration for the management query. Here's how to do this:

```
printf("\n\nPerforming query 'SELECT * FROM Win32_Service WHERE Started=true'\n");
printf("%-40ls\t%-12ls\t%ls\n", L"Service name", L"Process ID", L"State");
printf("-------------------------------------------------------------------------\n");

CWbemInstanceList queryServices(wmi);
queryServices.Query(L"SELECT * FROM Win32_Service WHERE Started=true");

CWbemInstanceList::iterator itrService2 = queryServices.begin();
for ( ; itrService2 != queryServices.end(); itrService2++)
{
   CComBSTR bstrName = BSTR(itrService2->GetProperty(L"Name"));
   int nProcessID = int(itrService2->GetProperty(L"ProcessId"));
   bool bStarted = bool(itrService2->GetProperty(L"Started"));

   printf("%-40ls\t%-12d\t%ls\n", bstrName, nProcessID,
      bStarted ? L"Started" : L"--");
}
```

Pass the `CWbemLocator` WMI connection to the constructor of the `CWbemInstanceList` class and call `Query` passing in a WQL query. Like enumerating a class's management objects, use the STL list iterators to obtain access to each management object.

The above code outputs the following:

```
Performing query 'SELECT * FROM Win32_Service WHERE Started=true'
Service name                                    Process ID      State
-------------------------------------------------------------------------------

Albd                                            1256            Started
AppMgmt                                         800             Started
AudioSrv                                        800             Started
AvSynMgr                                        1268            Started
BITS                                            800             Started
Browser                                         800             Started
cccredmgr                                       1280            Started
CryptSvc                                        800             Started
Dhcp                                            800             Started
... .. ...
.. .
```

As you can see, all the services listed have been started.

Retrieving a specific management object

Many management applications may know in advance which management objects they need. Use the class, `CWbemInstance`, and call `GetObject` to retrieve the management object. Here's how to do this:

```
printf("\n\nGetting management object 'Win32_Share.Name=\"C$\"'\n");
printf("-------------------------------------------------------------------------\n");

CWbemInstance inst(wmi);
inst.GetObject(L"Win32_Share.Name=\"C$\"");

CComBSTR bstrObjectPath = BSTR(inst.GetProperty(L"__PATH"));
CComBSTR bstrPath = BSTR(inst.GetProperty(L"Path"));

printf("'%ls' has path '%ls'\n", bstrObjectPath, bstrPath);
```

Pass the `CWbemLocator` WMI connection to the constructor of the `CWbemInstance` class and call `GetObject` passing in the object path of the requested management object. Use the `GetProperty` method to get the values of the properties.

The above code outputs the following:

```
Getting management object 'Win32_Share.Name="C$"'
-------------------------------------------------------------------------------

'\\COLEXP\root\CIMV2:Win32_Share.Name="C$"' has path 'C:\'
```

Calling management methods

Many management classes have useful methods that you may want to call in your management applications. The framework has a very useful class, `CWbemMethod` that makes it easy to setup and execute method calls. This includes setting up in-parameters and retrieving the method's return value and other out-parameters. Here's an example of how to call a static management class method. It creates a share of the directory C:\temp:

```
printf("\n\nCalling method 'Win32_Share::Create'\n");
printf("-------------------------------------------------------------------------\n");
```

```
CWbemMethod method(wmi, L"Win32_Share", L"Create");

method.SetInParam(L"Name", CComBSTR(L"Temp"));
method.SetInParam(L"Path", CComBSTR(L"C:\\temp"));
method.SetInParam(L"Type", int(0));

printf("Creating shared directory 'C:\\temp'\n");
method.Execute();

int nCreateRetVal = method.GetReturnValue();

printf("'Win32_Share::Create' returned '%d'\n", nCreateRetVal);
```

Pass the `CWbemLocator` WMI connection to the constructor of the `CWbemMethod` class and also supply the class reference and method name. Then call `SetInParam` to setup the method's in-parameters. To execute the method call `Execute`. Finally, call `GetReturnValue` and `GetOutParam` to get the method's return value and out-parameters respectively.

The above code outputs the following:

```
Calling method 'Win32_Share::Create'
------------------------------------------------------------------------------

Creating shared directory 'C:\temp'
'Win32_Share::Create' returned '0'
```

The next example demonstrates how to call a management object method (i.e. not a static class method). The main difference between the last example and the next is that instead of passing a class reference to `CWbemMethod`, you pass an object path of the management object. This example removes the share created earilier:

```
printf("\n\nCalling method 'Win32_Share::Delete'\n");
printf("------------------------------------------------------------------------\n");

// Note the object reference as apposed to a class reference!
CWbemMethod method2(wmi, L"Win32_Share.Name=\"Temp\"", L"Delete");

printf("Removing shared directory 'C:\\temp'\n");
method2.Execute();

int nDeleteRetVal = method2.GetReturnValue();

printf("'Win32_Share::Delete' returned '%d'\n", nDeleteRetVal);
```

The above code outputs the following:

```
Calling method 'Win32_Share::Delete'
------------------------------------------------------------------------------

Removing shared directory 'C:\temp'
'Win32_Share::Delete' returned '0'
```

Updating management objects

Many management applications may have the requirement to update existing management objects. This next example shows how this can easily be done using the framework. The principle is to get a management object and change one of its properties and commit is back to WMI. The following example retrieves a fruit basket management object (discussed in Chapter 12) and changes the property, `Capacity`.

```
printf("\n\nUpdate management object\n");
printf("------------------------------------------------------------------------\n");
```

```
CWbemInstance instUpdate(wmibook);
instUpdate.GetObject(L"SampleWinNET_Basket.Name=\"New York\"");

// Get a property and change it
int nCapacity = BYTE(instUpdate.GetProperty(L"Capacity"));

printf("'Capacity' is '%d'\n", nCapacity);
nCapacity++;
printf("'Capacity' is now '%d'\n", nCapacity);

// You may have to explicitly set a variant with the desired VT type
CComVariant varUpdateValue(nCapacity);
varUpdateValue.ChangeType(VT_UI1);

// Set new property value
instUpdate.SetProperty(L"Capacity", varUpdateValue);

printf("Updating management object\n");
instUpdate.UpdateInstance();
```

The `Capacity` property is of type `uint8`. The `CComVariant` constructor takes a C++ integer and sets-up a VARIANT of type `VT_I4`. From Table 7.1 in Chapter 7, you will be able to determine the VARIANT to WMI data type mapping that is needed. For a `uint8` data type we need to setup a VARIANT of type `VT_UI1`. This is what is done next and the VARIANT is passed to `SetProperty`. Finally, the management object is updated by calling `UpdateInstance`. The other methods of the `CWbemInstance` class are discussed later in the "Framework public classes" section.

The above code outputs the following:

```
Update management object
------------------------------------------------------------------------------

'Capacity' is '214'
'Capacity' is now '215'
Updating management object
```

Retrieving high-performance management objects

High-performance management classes are discussed in detail in Chapter 7. They can be accessed from both the `IWbemServices` and `IWbemRefresher` interfaces. The `IWbemRefresher` interface provides a means to access the class using techniques that contribute to high performance. The `IWbemRefresher` interface can be a bit laborious to use. Fortunately, the framework makes it a bit easier to access high-performance classes using a refresher. The following example gets a *process* performance counter instance and outputs some of the class's properties (performance counters):

```
printf("\n\nGetting a single high performance refresher object\n");
printf("%-40ls\t%-12ls\t%ls\n", L"Process", L"Process ID", L"Thread Count");
printf("-------------------------------------------------------------------\n");

CWbemRefresher refresh(wmi);

LPCWSTR lpszPerfObjPath = L"Win32_PerfRawData_PerfProc_Process.Name=\"lsass\"";

// Add path of hi-perf object
refresh.AddObjectByPath(lpszPerfObjPath);

refresh.Refresh();

// Display result
```

```
wstring szProcessName;
refresh.GetProperty(lpszPerfObjPath, L"Name", szProcessName);

DWORD dwProcessID = 0;
refresh.GetProperty(lpszPerfObjPath, L"IDProcess", dwProcessID);

DWORD dwThreadCount = 0;
refresh.GetProperty(lpszPerfObjPath, L"ThreadCount", dwThreadCount);

unsigned __int64 ulPrivateBytes = 0;
refresh.GetProperty(lpszPerfObjPath, L"PrivateBytes", ulPrivateBytes);

printf("%-40ls\t%-12d\t%d\n", szProcessName.c_str(), dwProcessID, dwThreadCount);
```

Pass the `CWbemLocator` WMI connection to the constructor of the `CWbemRefresher` class. The `IWbemServices` interface encapsulated in `CWbemLocator` object will be used by `AddObjectByPath` where the object path refers to a high-performance object referred to by the WMI connection's namespace. `AddObjectByPath` adds high-performance objects to the refresher. To get and update the high-performance object's values call, `Refresh`. Property values are obtained from the high-performance object by calling `GetProperty`. `GetProperty` is overloaded to retrieve strings, DWORDs and 64-bit integers.

The above code outputs the following:

```
Getting a single high performance refresher object
Process                                          Process ID      Thread Count
-------------------------------------------------------------------------------

lsass                                            568             21
```

Retrieving high-performance management class enumerations

The `IWbemRefresher` interface can also be used to retrieve enumerations of high-performance classes and is also a bit laborious to use. Let's have a look at how enumerations are retrieved using a refresher in the framework. The following example gets all the *process* performance instances and outputs some details:

```
printf("\n\nGetting high performance refresher objects (enumeration)\n");
printf("%-40ls\t%-12ls\t%ls\n", L"Process", L"Process ID", L"Thread Count");
printf("-------------------------------------------------------------------------\n");

refresh.AddEnumeration(L"Win32_PerfRawData_PerfProc_Process");

refresh.Refresh();

CWbemRefresherEnumObjects listRefreshObjects;
refresh.GetEnumeration(L"Win32_PerfRawData_PerfProc_Process", listRefreshObjects);

CWbemRefresherEnumObjects::iterator itrProcess = listRefreshObjects.begin();
for ( ; itrProcess != listRefreshObjects.end(); itrProcess++)
{
   REFRESHER_OBJECT& object = *itrProcess;

   wstring szProcessName;
   refresh.GetProperty(object, L"Name", szProcessName);

   DWORD dwProcessID = 0;
   refresh.GetProperty(object, L"IDProcess", dwProcessID);

   DWORD dwThreadCount = 0;
```

```
        refresh.GetProperty(object, L"ThreadCount", dwThreadCount);

        printf("%-40ls\t%-12d\t%d\n", szProcessName.c_str(), dwProcessID, dwThreadCount);
    }
```

Enumerations can be added to the refresher using `AddEnumeration` and passing in the class reference. Access to each high-performance object can be accessed using the STL iterators. Each high-performance object is represented using the `REFRESHER_OBJECT` structure which holds information that makes it possible to access the properties using high-performance techniques (i.e. using the `IWbemObjectAccess` interface's 32-bit property handles).

The above code outputs the following:

```
Getting high performance refresher objects (enumeration)
Process                                          Process ID      Thread Count
-------------------------------------------------------------------------------

Idle                                             0               1
System                                           4               50
smss                                             332             3
csrss                                            436             13
winlogon                                         460             21
services                                         556             17
lsass                                            568             21
svchost                                          748             10
svchost                                          800             84
... .. ...
.. .
```

Framework public classes

The WMI Consumer Framework consists of a number of C++ classes (Table 1) that provide easy access to management objects, paths, methods, etc…

Table 1: Consumer C++ classes

C++ class	Comment
`CWbemSecurity`	A class that has static methods to setup DCOM security settings.
`CWbemLocator`	Manages a connection to a WMI namespace.
`CWbemObjectPath`	A helper class that parses and creates object paths.
`CWbemInstance`	This class represents a management object.
`CWbemInstanceList`	This class represents a collection of management objects.
`CWbemMethod`	This class makes it easy to setup method calls to WMI.
`CWbemVariantCast`	A helper class to cast a VARIANT into a C++ equivalent type.
`CWbemRefresher`	This class makes it easy to access high-performance classes.

CWbemSecurity class

This class provides helper functions to initialize management applications with the correct security context. Here is the public interface of the class:

```
class CWbemSecurity
{
public:
```

```
        static HRESULT InitializeSecurity(DWORD dwAuthLevel = RPC_C_AUTHN_LEVEL_CONNECT,
           DWORD dwImpLevel = RPC_C_IMP_LEVEL_IMPERSONATE);

        static HRESULT SetProxyBlanket(LPUNKNOWN lpUnk,
           DWORD dwAuthLevel = RPC_C_AUTHN_LEVEL_CALL,
           DWORD dwImpLevel = RPC_C_IMP_LEVEL_IMPERSONATE);
};
```

InitializeSecurity method

```
        static HRESULT InitializeSecurity(DWORD dwAuthLevel = RPC_C_AUTHN_LEVEL_CONNECT,
            DWORD dwImpLevel = RPC_C_IMP_LEVEL_IMPERSONATE);
```

This method initializes the security requirements of the process. If the consumer is implemented in a DLL, then you should call `SetProxyBlanket` instead.

SetProxyBlanket method

```
        static HRESULT SetProxyBlanket(LPUNKNOWN lpUnk,
            DWORD dwAuthLevel = RPC_C_AUTHN_LEVEL_CALL,
            DWORD dwImpLevel = RPC_C_IMP_LEVEL_IMPERSONATE);
```

This method sets the proxy's security requirements for an interface. If the consumer is implemented in a DLL, then you will more than likely be required to call this function.

CWbemLocator class

This class represents a connection to a WMI namespace. The implementation of `Connect` uses the `CWbemSecurity::SetProxyBlanket` function to set the security settings supplied by `dwAuthLevel` and `dwImpLevel` parameters. The `IWbemServices` interface can be obtained through `GetServices` and `GetServicesWeakReference`. Here is the public interface of the template class:

```
class CWbemLocator
{
public:
    IWbemServices* GetServicesWeakReference();
    void GetServices(IWbemServices** ppServices);

    void Connect(LPCWSTR lpszServer = L".", LPCWSTR lpszNamespace = L"root\\CIMV2",
       LPCWSTR lpszUser = NULL, LPCWSTR lpszPwd = NULL,
       DWORD dwAuthLevel = RPC_C_AUTHN_LEVEL_CALL,
       DWORD dwImpLevel = RPC_C_IMP_LEVEL_IMPERSONATE);
};
```

CWbemObjectPath class

This class helps you to parse management object paths. The meaning of the methods in the class below should be intuitive and easy to understand. The Windows 2000 version of this class defers slightly but produces the same functionality.

```
class CWbemObjectPath
{
public:
    CWbemObjectPath(LPCWSTR lpszPath);
    ~CWbemObjectPath();

    void GetClassName(CComBSTR& bstrClass);

    bool IsKeyValid(DWORD dwKeyIndex);
    bool GetKey(DWORD dwKeyIndex, CComBSTR& bstrKeyName);
    bool GetKeyVariant(DWORD dwKeyIndex, CComVariant& varKeyValue);
```

```
        bool GetKeyVariant(LPCWSTR lpszPropName, CComVariant& varPropValue);

        // Helper to get at the implementation provided by the WMI
        void GetObjectPath(IWbemPath** ppPath);
        void GetObjectPathKeyList(IWbemPathKeyList** ppKeyList);
    };
```

CWbemInstance class

This class includes common operations that you might want to perform with a management object. It is used by the framework and has some useful helper functions. Here is the public interface of the class:

```
    class CWbemInstance
    {
    public:
        CWbemInstance() { }
        CWbemInstance(CWbemLocator& loc) { SetLocator(loc); }
        CWbemInstance(IWbemClassObject* pInstance) :
           m_spInstance(pInstance) { }
        CWbemInstance(CWbemLocator& loc, IWbemClassObject* pInstance) :
           m_spInstance(pInstance) { SetLocator(loc); }

        void SetInstance(IWbemClassObject* pInstance);
        void GetInstance(IWbemClassObject** ppInstance)
        IWbemClassObject* GetInstanceWeakReference();

        void SetLocator(CWbemLocator& loc);

        void Create(LPCWSTR lpszClassName);

        HRESULT GetObject(LPCWSTR lpszObjectPath, long lFlags = 0);

        template <class ValType>
        void SetProperty(LPCWSTR lpszPropName, ValType value);
        void SetPropertyVariant(LPCWSTR lpszPropName, VARIANT* pValue);

        CWbemVariantCast GetProperty(LPCWSTR lpszPropName);
        void GetPropertyVariant(LPCWSTR lpszPropName, VARIANT* pValue);

        void PutInstance(long lFlags = WBEM_FLAG_CREATE_ONLY);
        void UpdateInstance(long lFlags = WBEM_FLAG_UPDATE_ONLY);

        void DeleteInstance(LPCWSTR lpszObjPath, long lFlags = 0);
    };
```

GetInstance and GetInstanceWeakReference methods

These methods return the attached `IWbemClassObject` instance.

```
    void GetInstance(IWbemClassObject** ppInstance);
    IWbemClassObject* GetInstanceWeakReference();
```

Calling `GetInstance` adds a reference and `GetInstanceWeakReference` doesn't.

Create method

This spawns a new instance of the class passed through the `lpszClassName` parameter.

```
    void Create(LPCWSTR lpszClassName);
```

GetObject method

This retrieves the management object from WMI. The management object path is supplied through `lpszObjectPath`. For detail of the flags (`lFlags`) that can be passed, refer to Chapter 7.

```
HRESULT GetObject(LPCWSTR lpszObjectPath, long lFlags = 0);
```

GetProperty and GetPropertyVariant methods

These methods return the value of a management object property.

```
CWbemVariantCast GetProperty(LPCWSTR lpszPropName);
void GetPropertyVariant(LPCWSTR lpszPropName, VARIANT* pValue);
```

The first method returns the property value through `CWbemVariantCast` which casts VARIANT data types to C++ data types. The second method returns the raw variant value for the property. This is useful for dealing with SAFEARRAY-based VARIANTs.

SetProperty and SetPropertyVariant methods

These methods set the value of a management object property.

```
template <class ValType>
void SetProperty(LPCWSTR lpszPropName, ValType value);
void SetPropertyVariant(LPCWSTR lpszPropName, VARIANT* pValue);
```

The first method sets the property value by using `CComVariant`'s constructor. This enables you to use C++ data types for many of the properties you're likely to come across. The second method sets a raw variant value for the property. This is useful for dealing with SAFEARRAY-based VARIANTs.

PutInstance and UpdateInstance methods

Both of the following methods call the `IWbemServices` interface method, `PutInstance`. The only difference is the default flag passed through the `lFlags` parameter.

```
void PutInstance(long lFlags = WBEM_FLAG_CREATE_ONLY);
void UpdateInstance(long lFlags = WBEM_FLAG_UPDATE_ONLY);
```

DeleteInstance method

This method removes the management object referred to by the object path, `lpszObjPath`. For detail of the flags (`lFlags`) that can be passed, refer to Chapter 7.

```
void DeleteInstance(LPCWSTR lpszObjPath, long lFlags = 0);
```

CWbemInstanceList class

This class holds a collection of management objects. Calling `Create` will retrieve the management object enumeration for the class specified by the `lpszClass` parameter. `Query` will perform a management WQL query and return the results into the collection. For detail of the flags (`lFlags`) that can be passed, refer to Chapter 7.

```
class CWbemInstanceList : public std::list<CWbemInstance>
{
public:
	CWbemInstanceList(CWbemLocator& loc);
	virtual ~CWbemInstanceList();

	void Create(LPCWSTR lpszClass, long lFlags = WBEM_FLAG_SHALLOW);

	void Query(LPCWSTR lpszQuery, long lFlags = 0L);
};
```

CWbemMethod class

This class represents a method. You can set the method's in-parameters by calling `SetInParam` and get the out-parameters using `GetOutParam`. The method can be executed using `Execute` and it's return value can be obtained with `GetReturnValue`.

```
class CWbemMethod
{
public:
	CWbemMethod(CWbemLocator& loc, LPCWSTR lpszObjectRef,
		LPCWSTR lpszMethodName);

	void Execute();

	template <class ValType>
	void SetInParam(LPCWSTR lpszParamName, ValType value);

	CWbemVariantCast GetOutParam(LPCWSTR lpszParamName);
	CWbemVariantCast GetReturnValue();
};
```

CWbemVariantCast class

This class converts VARIANTs to C++ data types. The class below shows which data types are converted.

```
class CWbemVariantCast
{
public:
	CWbemVariantCast(CComVariant& v, bool bTakeOwnership = false);
	CWbemVariantCast(const VARIANT& v);

	operator int();
	operator long();
	operator UINT();
	operator DWORD();
	operator BYTE();
	operator short();
	operator bool();
	operator BSTR();
};
```

CWbemRefresher class

The following class represents a *refresher* which is used to get high-performance management classes. For more information about high-performance classes, refer to Chapter 7.

A high-performance object can be added the refresher using `AddObjectByPath` and high-performance class enumerations can be added with `AddEnumeration`. `Refresh` can be called to update the property values of each high-performance object added to the refresher.

```
class CWbemRefresher
{
public:
	CWbemRefresher();
	CWbemRefresher(CWbemLocator& locator);

	// Added in case the appropriate constructor cannot be called.
	void SetWbemServices(CWbemLocator& locator);

	void AddObjectByPath(LPCWSTR lpszPath);
```

```
        void AddEnumeration(LPCWSTR lpszClass);

        void GetProperty(LPCWSTR lpszPath, LPCWSTR lpszProperty,
               DWORD& dwValue);
        void GetProperty(LPCWSTR lpszPath, LPCWSTR lpszProperty,
               unsigned __int64& qwValue);
        void GetProperty(LPCWSTR lpszPath, LPCWSTR lpszProperty,
               wstring& szValue);

        void GetProperty(REFRESHER_OBJECT& object, LPCWSTR lpszProperty,
               DWORD& dwValue);
        void GetProperty(REFRESHER_OBJECT& object, LPCWSTR lpszProperty,
               unsigned __int64& qwValue);
        void GetProperty(REFRESHER_OBJECT& object, LPCWSTR lpszProperty,
               wstring& szValue);

        void GetEnumeration(LPCWSTR lpszClass,
               CWbemRefresherEnumObjects& listObjects);

        void Refresh(WBEM_REFRESHER_FLAGS Flags = WBEM_FLAG_REFRESH_AUTO_RECONNECT);
};
```

GetProperty methods

There are two sets of overloaded `GetProperty` methods. The first uses an object path (`LPCWSTR lpszPath`) to reference each high-performance object. And the second uses a structure (`REFRESHER_OBJECT`) which contains the details of the high-performance object. The second version is used when dealing with refresher enumerations.