WMI Providers

Provider Application Framework

Chapter 12 of "Developing WMI Solutions" covers how developers can develop their own WMI providers. One of the most powerful features of Windows Management Instrumentation (WMI) is that it allows developers to expose their own management objects through a provider. WMI offers a lot of infrastructure that makes writing providers reasonably straightforward. The application framework described here makes writing WMI providers even easier.

Getting started

Discussion of this application framework assumes knowledge of C++ and WMI provider development. For more information about writing WMI providers, read Chapter 12. The following guide will re-implement the fruit basket WMI provider example discussed in Chapter 12.

There are two versions of this framework. One for Windows 2000 which uses the Platform SDK provided code for parsing object paths and there's code to parse WQL queries. The Windows XP and Server 2003 version uses the Operating System provided object path parser and query parser.

Let's start by creating a new project and describing the tasks you should do to get a provider up and running.

1) Create a new ATL-based project with Visual Studio 6.0.
2) Add an *ATL Class* and specify *Custom* for the interface type field.
3) Open the .idl file and remove references to the newly created interface. Then insert the sections in the following highlighted IDL code:

```
import "oaidl.idl";
import "ocidl.idl";
import "wbemprov.idl";

[
	uuid(67033001-B1D4-4074-A4C9-9DC440E412E9),
	version(1.0),
	helpstring("AfxWMIProviderSample 1.0 Type Library")
]
library AFXWMIPROVIDERSAMPLELib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	[
		uuid(BACFF311-AA23-4C5D-A769-5172894D2E74),
		helpstring("FruitBasket Class")
	]
	coclass FruitBasket
	{
		[default] interface IWbemProviderInit;
		interface IWbemServices;
```

```
        };
};
```

If you're planning for your provider to also be an event provider, then you should also add `interface IWbemEventProvider` in the coclass's interface list.

4) The stdafx.h file should define `_ATL_FREE_THREADED` instead of `_ATL_APARTMENT_THREADED` (which is added by default by the ATL class wizard). You must include the WMI Provider framework by including WMIProvider.h and define the `WMIPROVIDER_COMPONENTNAME` symbol with a unique name for your provider (note this should be #defined before including WMIProvider.h. For example:

```
#define _ATL_FREE_THREADED
#include <atlbase.h>
... .. ...
.... .
..
#define WMIPROVIDER_COMPONENTNAME L"WMIProvider_Sample"
#include <WMIProvider.h>
```

5) If you are developing an out-of-process COM server, ensure that you call `CoInitializeSecurity` in the project's main .cpp file. Out-of-process WMI provider security will be covered in another article. Do something similar to this:

```
CoInitializeSecurity( 0, -1, 0, 0, RPC_C_AUTHN_LEVEL_CONNECT,
        RPC_C_IMP_LEVEL_IMPERSONATE, 0, EOAC_NONE, 0);
```

6) Open the header file with the coclass definition. Make sure it looks something like the following. The highlighted sections show the changes you make:

```
class CFruitBasket :
        public CComObjectRoot,
        public CComCoClass<CFruitBasket,&CLSID_FruitBasket>,
        public IWbemProviderInitImpl<CFruitBasket>,
        public IWbemServicesImpl<CFruitBasket>
{
public:
        CFruitBasket() {}

        BEGIN_COM_MAP(CFruitBasket)
                COM_INTERFACE_ENTRY(IWbemProviderInit)
                COM_INTERFACE_ENTRY(IWbemServices)
        END_COM_MAP()
        ... ..
        ..
};
```

7) Setup a .mof file that registers your provider with WMI which may also optionally include the WMI management classes that the provider will expose. For more information on how to do this, go to Chapter 12.

Now we are ready to move forward by discussing each provider function and how to implement it using the framework.

Provider initialization

On occasion you may need to perform specific initialization of the provider. The template class `IWbemProviderInitImpl` provides a default implementation of the `IWbemServices::Initialize` method. To add your own initialization, override the `OnInitialize` method and ensure that it is defined as `public`.

```
void OnInitialize(LPWSTR wszUser, LONG lFlags, LPWSTR wszNamespace,
        IWbemContext *pCtx);
```

The default implementation keeps a reference to the `IWbemServices` interface provided by WMI in the member, `m_spWbemServices`. If required, `m_spWbemServices` can be used within the `OnInitialize` method.

Providing management object enumeration

Providing management object enumeration is considered *best practice*, perhaps even a requirement. As you will have gathered from Chapter 12, WMI instance providers can expose one or more management classes. All requests that are passed to the provider include the management class and it is up to the provider how it processes requests when several supported classes are involved. The most common method to solve this is to use a technique of routing requests for a specific class to a specific function. The instance provider sample in Chapter 12 used this approach and so does this provider framework. Use the `BEGIN_ENUM_INSTANCE_MAP` macro to start the mapping between management classes and the functions that will serve the requests. Each management class supported by the provider should have an entry in the map. The `ENUM_INSTANCE_ENTRY` macro is used to route management class enumeration requests to a function responsible with exposing the enumeration. Finally, the enumeration map is completed with the `END_ENUM_INSTANCE_MAP` macro. The following code highlighted in bold shows the map and the functions that will expose the enumeration.

```
class CFruitBasket :
    public CComObjectRoot,
    public CComCoClass<CFruitBasket,&CLSID_FruitBasket>,
    public IWbemProviderInitImpl<CFruitBasket>,
    public IWbemServicesImpl<CFruitBasket>
{
    .... ..
    ...
    .
public:
    BEGIN_ENUM_INSTANCE_MAP()
       ENUM_INSTANCE_ENTRY(L"SampleWinNET_Basket", OnBasketInstances)
       ENUM_INSTANCE_ENTRY(L"SampleWinNET_Fruit", OnFruitInstances)
       ENUM_INSTANCE_ENTRY(L"SampleWinNET_BasketFruitMembership",
          OnFruitBasketInstances)
    END_ENUM_INSTANCE_MAP()

private:
    void OnBasketInstances(CWbemInstanceList& instList, LPWBEMMETHODCTX ptrs);
    void OnFruitInstances(CWbemInstanceList& instList, LPWBEMMETHODCTX ptrs);
    void OnFruitBasketInstances(CWbemInstanceList& instList,
         LPWBEMMETHODCTX ptrs);
    . ...
    ..
};
```

To understand the basics of implementing the enumeration function, let's look at a simple example:

```
void CSomeProvider::OnExampleInstances(CWbemInstanceList& instList,
                                       LPWBEMMETHODCTX ptrs)
{
    while (not end of the enumeration)
    {
       // Create management object
       LPWBEMINSTANCE pInst = CreateWbemInstance(ptrs);

       if (pInst)
       {
          // Set management object properties
          pInst->SetProperty(L"Some Property", CComBSTR(some string));
          pInst->SetProperty(L"Another Property", int(some value));
```

```
            // Add management object to collection
            instList.Add(pInst);
         }
      }
   }
```

Within the while loop, call `CreateWbemInstance` to create a management object which you intend to expose. `CreateWbemInstance` knows which type of object to create because it gets the class name from the `ptrs` parameter. The next step involves setting all the management object's properties that you want to expose. Finally, add the new management object to the collection which is immediately passed to WMI and forwarded to the client. The following code is the implementation of the `OnBasketInstances` method.

```
void CFruitBasket::OnBasketInstances(CWbemInstanceList& instList,
                                     LPWBEMMETHODCTX ptrs)
{
    USES_CONVERSION;

    // Get basket instances from basket registry key
    CRegKey regBasketEnum;
    LONG lReg = regBasketEnum.Open(HKEY_LOCAL_MACHINE,
       _T("Software\\WMIBookProv\\Basket"), KEY_ENUMERATE_SUB_KEYS);

    if (lReg == ERROR_SUCCESS)
    {
       DWORD dwRegIndex = 0;
       LONG lRegBasket = 0;

       do
       {
          // Enum the basket key entries
          WCHAR szBasketName[128];
          lRegBasket = RegEnumKeyW(regBasketEnum, dwRegIndex++,
             szBasketName, 128);

          if (lRegBasket == ERROR_SUCCESS)
          {
             // Create instance!
             LPWBEMINSTANCE pInst = CreateWbemInstance(ptrs);

             if (pInst)
             {
                // Set property
                pInst->SetProperty(L"Name", szBasketName);

                // Get other basket properties
                CComBSTR bstrBasketPath(L"Software\\WMIBookProv\\Basket\\");
                bstrBasketPath += szBasketName;

                CRegKey regBasket;
                lReg = regBasket.Open(HKEY_LOCAL_MACHINE,
                   OLE2T(bstrBasketPath), KEY_QUERY_VALUE);

                if (lReg == ERROR_SUCCESS)
                {
                   DWORD dwCapacity = 0;
                   lReg = regBasket.QueryValue(dwCapacity, _T("Capacity"));
```

```
                    if (lReg == ERROR_SUCCESS)
                    {
                       CComVariant varCapacity((int)dwCapacity);
                       varCapacity.ChangeType(VT_UI1);

                       // Set other properties
                       pInst->SetProperty(L"Capacity", varCapacity);
                    }
                 }

                 // Add instance to collection of instances!
                 instList.Add(pInst);
              }
           }
        }
        while (lRegBasket == ERROR_SUCCESS);
     }
  }
```

The `CWbemInstanceList` class is discussed later in the "Framework public classes" section. You can view the code for the method `OnFruitInstances` by checking the source code.

Let's look at the `OnFruitBasketInstances` method in more detail because this exposes an association management class. The main difference in implementation is that properties of an association class include reference properties and these are set with a call to `SetObjPathProperty` and not `SetProperty` (like in the previous example). `SetObjPathProperty` requires a `WBEMKEYREFS` array which represents the key properties of the management object reference. Each element in the array is a key property of the reference. In the fruit basket sample, both the fruit and basket management objects have one key property.

Each `WBEMKEYREFS` array element needs to know the name (`lpszKeyRefName`) of the key property; the CIM data type (`cimtypeValue`) of the key property; and finally the value (`varKeyRefValue`) of the key property.

Let's have a look at the implementation:

```
void CFruitBasket::OnFruitBasketInstances(CWbemInstanceList& instList,
                                          LPWBEMMETHODCTX ptrs)
{
    TBasketFruitMap mapFruit;
    GetEnumFruitMap(mapFruit);

    TBasketFruitMap::iterator itrFruit = mapFruit.begin();
    for ( ; itrFruit != mapFruit.end(); itrFruit++)
    {
       // Create instance!
       LPWBEMINSTANCE pInst = CreateWbemInstance(ptrs);

       if (pInst)
       {
          // Basket reference
          WBEMKEYREFS keyRef[1];
          keyRef[0].lpszKeyRefName = _T("Name");
          keyRef[0].cimtypeValue = CIM_STRING;
          keyRef[0].varKeyRefValue =
             CComVariant((*itrFruit).second.szBasketName.c_str());

          pInst->SetObjPathProperty(_T("SampleWinNET_Basket"),
             _T("Basket"), keyRef, 1);

          // Fruit reference
```

```
            keyRef[0].lpszKeyRefName = _T("Name");
            keyRef[0].cimtypeValue = CIM_STRING;
            keyRef[0].varKeyRefValue = CComVariant((*itrFruit).first.c_str());

            pInst->SetObjPathProperty(_T("SampleWinNET_Fruit"),
               _T("Fruit"), keyRef, 1);

            instList.Add(pInst);
         }
      }
   }
```

The first parameter to `SetObjPathProperty` is the name of the referenced management class. The second parameter is the name of the reference property. The third parameter is the array of `WBEMKEYREFS` structures and the last parameter indicates how many elements there are in the array.

Providing management object retrieval

Providing the ability to retrieve a specific management object is considered *best practice*, perhaps even a requirement. It's very irritating when you cannot browse a specific management object in CIM Studio because the provider only exposes the enumeration. To provide management object retrieval, use the `BEGIN_OBJECT_INSTANCE_MAP` macro to start the mapping between management classes and the functions that will serve the requests. Each management class supported by the provider should have an entry in the map. The `OBJECT_INSTANCE_ENTRY` macro is used to route management class retrieval request to a function responsible with exposing it. Finally, the map is completed with the `END_OBJECT_INSTANCE_MAP` macro. The following code highlighted in bold shows the map and the functions that will expose the retrieved management object.

```
class CFruitBasket :
    ... ... ..
    .. ...
{
public:
    BEGIN_OBJECT_INSTANCE_MAP()
       OBJECT_INSTANCE_ENTRY(L"SampleWinNET_Basket", OnBasketObject)
       OBJECT_INSTANCE_ENTRY(L"SampleWinNET_Fruit", OnFruitObject)
       OBJECT_INSTANCE_ENTRY(L"SampleWinNET_BasketFruitMembership",
          OnFruitBasketObject)
    END_OBJECT_INSTANCE_MAP()

private:
    void OnBasketObject(LPWBEMINSTANCE pInstance, long lFlags);
    void OnFruitObject(LPWBEMINSTANCE pInstance, long lFlags);
    void OnFruitBasketObject(LPWBEMINSTANCE pInstance, long lFlags);
    . ...
    ..
};
```

To understand the basics of implementing the retrieval function, let's look at a simple example:

```
void CSomeProvider::OnExampleObject(LPWBEMINSTANCE pInstance, long lFlags)
{
    // Get key properties from managment object request
    CComBSTR bstrKeyProp1 = BSTR(pInstance->GetProperty(L"Some Property"));
    CComBSTR bstrKeyProp2 = BSTR(pInstance->GetProperty(L"Another Property"));

    bool bFound = false;
    //
    // TODO: Find object within your own local cache. Set bFound accordingly...
    //
```

```
    if (bFound)
    {
       // Set other management object properties
       pInstance->SetProperty(L"My Property 1", varProp1);
       pInstance->SetProperty(L"My Property 2", varProp2);
       pInstance->SetProperty(L"My Property 3", varProp3);
    }
    else
    {
       // Object not found, throw this exception which will get caught
       // by the framework.
       WMIThrowHRESULT(WBEM_E_NOT_FOUND);
    }
}
```

The management object to be exposed is already created by the framework and is passed into the function via the `pInstance` parameter. The next step involves getting the key properties so that the existence of the management object can be identified. If it does exist, set the rest of the management object's properties. If it doesn't exist, throw a `WBEM_E_NOT_FOUND` HRESULT exception. On leaving the routing function, the management object is immediately passed to WMI and is forwarded to the client. The following code is the implementation of the `OnBasketObject` method.

```
void CFruitBasket::OnBasketObject(LPWBEMINSTANCE pInstance, long lFlags)
{
    // Get key property, this contains the value for which object is required
    bool bFound = false;
    CComBSTR bstrName = BSTR(pInstance->GetProperty(L"Name"));

    // Build registry path
    CComBSTR bstrBasketPath(L"Software\\WMIBookProv\\Basket\\");
    bstrBasketPath += bstrName;

    CRegKey regBasket;
    LONG lReg = regBasket.Open(HKEY_LOCAL_MACHINE, OLE2T(bstrBasketPath),
       KEY_QUERY_VALUE);

    if (lReg == ERROR_SUCCESS)
    {
       // Registry entry exists for basket
       bFound = true;

       // Get other basket properties
       DWORD dwCapacity = 0;
       lReg = regBasket.QueryValue(dwCapacity, _T("Capacity"));

       if (lReg == ERROR_SUCCESS)
       {
          CComVariant varCapacity((int)dwCapacity);
          varCapacity.ChangeType(VT_UI1);

          // Set property!!
          pInstance->SetProperty(L"Capacity", varCapacity);
       }
    }

    if (!bFound)
       WMIThrowHRESULT(WBEM_E_NOT_FOUND);
}
```

The CWbemInstance class pointed to by LPWBEMINSTANCE is discussed later in the "Framework public classes" section. You can view the code for the methods OnFruitObject and OnFruitBasketObject by checking the source code. OnFruitBasketObject makes use of CWbemObjectPath to parse the reference object path and is also discussed in more detail in the "Framework public classes" section.

Providing management object creation and update

Depending on the management classes that a provider exposes, some may need the ability to create and update management objects. Use the BEGIN_PUT_INSTANCE_MAP macro to start the mapping between management classes and the functions that will serve the requests. Each management class supported by the provider should have an entry in the map. The PUT_INSTANCE_ENTRY macro is used to route a management class *put* request to a function responsible with persisting the change. Finally, the map is completed with the END_PUT_INSTANCE_MAP macro. Note that if a management class does not support creation or update operations, it should use the PUT_INSTANCE_ENTRY_NOTSUPPORTED macro. If this is not done the return code from the provider will indicate that the class is invalid, which is not the case. The following code highlighted in bold shows the map and the functions that will provide management object creation and update operations.

```
class CFruitBasket :
    ... ... ..
    .. ...
{
public:
    BEGIN_PUT_INSTANCE_MAP()
       PUT_INSTANCE_ENTRY(L"SampleWinNET_Basket", OnBasketPut)
       PUT_INSTANCE_ENTRY(L"SampleWinNET_Fruit", OnFruitPut)
       PUT_INSTANCE_ENTRY_NOTSUPPORTED(L"SampleWinNET_BasketFruitMembership")
    END_PUT_INSTANCE_MAP()

private:
    void OnBasketPut(LPWBEMINSTANCE pInstance, long lFlags);
    void OnFruitPut(LPWBEMINSTANCE pInstance, long lFlags);
    void OnFruitBasketObject(LPWBEMINSTANCE pInstance, long lFlags);

    HRESULT UpdateBasket(LPWBEMINSTANCE pInstance);
    HRESULT UpdateFruit(LPWBEMINSTANCE pInstance, LPCWSTR lpszFruitRegPath);
    void GetEnumFruitMap(TBasketFruitMap& map);
    bool FindFruitInMap(LPCWSTR lpszFruitName, TBasketFruitMap& map);
    . ...
    ..
};
```

To understand the basics of implementing management object creation and update operations, let's look at a simple example:

```
void CSomeProvider::OnExamplePut(LPWBEMINSTANCE pInstance, long lFlags)
{
    if (lFlags == WBEM_FLAG_CREATE_OR_UPDATE)
    {
       CComBSTR bstrKeyProp = BSTR(pInstance->GetProperty(L"Some Property"));
       bool bFound = FindMgmtObject(bstrKeyProp);

       if (bFound)
          ; // Update management object
       else
          ; // Create management object
    }
    else if (lFlags & WBEM_FLAG_UPDATE_ONLY)
    {
```

```
            CComBSTR bstrKeyProp = BSTR(pInstance->GetProperty(L"Some Property"));
            bool bFound = FindMgmtObject(bstrKeyProp);

            if (bFound)
               ; // Update management object
            else
               WMIThrowHRESULT(WBEM_E_NOT_FOUND);
         }
         else if (lFlags & WBEM_FLAG_CREATE_ONLY)
         {
            CComBSTR bstrKeyProp = BSTR(pInstance->GetProperty(L"Some Property"));
            bool bFound = FindMgmtObject(bstrKeyProp);

            if (bFound)
               WMIThrowHRESULT(WBEM_E_ALREADY_EXISTS);
            else
               ; // Update management object
         }
         else
         {
            WMIThrowHRESULT(WBEM_E_INVALID_PARAMETER);
         }
      }
```

There are three flags that determine how the provider should process the changes of the *put* operation. These are `WBEM_FLAG_CREATE_OR_UPDATE`, `WBEM_FLAG_UPDATE_ONLY` and `WBEM_FLAG_CREATE_ONLY`. The meaning of each flag should be intuitive. Note that WMI HRESULTs are thrown for certain conditions where the request should fail. As you will see in the next code sample, you may need to throw other HRESULTs depending on what type of *put* operations the provider supports.

The management object to be created or updated is contained in the `pInstance` parameter. The provider should get all the properties it cares about from the management object and perform the appropriate tasks to persist the changes. On leaving the routing function, either the management object is persisted or the operation failed. The following code is the implementation of the `OnBasketPut` method.

```
      void CFruitBasket::OnBasketPut(LPWBEMINSTANCE pInstance, long lFlags)
      {
         if (lFlags == WBEM_FLAG_CREATE_OR_UPDATE)
         {
            CComBSTR bstrBasketName = BSTR(pInstance->GetProperty(L"Name"));
            bool bFound = FindBasket(bstrBasketName);

            if (bFound)
            {
               HRESULT hr = UpdateBasket(pInstance);

               if (FAILED(hr))
                  WMIThrowHRESULT(hr);
            }
            else
            {
               WMIThrowHRESULT(WBEM_E_UNSUPPORTED_PUT_EXTENSION);
            }
         }
         else if (lFlags & WBEM_FLAG_UPDATE_ONLY)
         {
            CComBSTR bstrBasketName = BSTR(pInstance->GetProperty(L"Name"));
            bool bFound = FindBasket(bstrBasketName);
```

```
            if (bFound)
            {
               HRESULT hr = UpdateBasket(pInstance);

               if (FAILED(hr))
                  WMIThrowHRESULT(hr);
            }
            else
            {
               WMIThrowHRESULT(WBEM_E_NOT_FOUND);
            }
         }
         else if (lFlags & WBEM_FLAG_CREATE_ONLY)
         {
            WMIThrowHRESULT(WBEM_E_UNSUPPORTED_PUT_EXTENSION);
         }
         else
         {
            WMIThrowHRESULT(WBEM_E_INVALID_PARAMETER);
         }
      }
```

Notice the use of the WMI HRESULT `WBEM_E_UNSUPPORTED_PUT_EXTENSION` to indicate that the requested operation failed. The next code sample shows the function, `UpdateBasket`, which persists the properties from the management object passed in by the WMI client. The code in bold highlight shows an example of transferring the values from properties to the *local cache* (in this case, the registry).

```
HRESULT CFruitBasket::UpdateBasket(LPWBEMINSTANCE pInstance)
{
    HRESULT hr = WBEM_S_NO_ERROR;

    // Get the basket that is required for the update
    CComBSTR bstrBasketName = BSTR(pInstance->GetProperty(L"Name"));

    // Build the registry path and open the basket key
    CComBSTR bstrBasketPath(L"Software\\WMIBookProv\\Basket\\");
    bstrBasketPath += bstrBasketName;

    CRegKey regBasket;
    LONG lReg = regBasket.Open(HKEY_LOCAL_MACHINE, OLE2T(bstrBasketPath),
       KEY_WRITE);

    // Get other properties
    CComVariant varCapacity;
    pInstance->GetPropertyVariant(L"Capacity", &varCapacity);

    lReg = regBasket.SetValue(DWORD(V_UI1(&varCapacity)), _T("Capacity"));

    if (lReg != ERROR_SUCCESS)
       hr = WBEM_E_PROVIDER_FAILURE;

    return hr;
}
```

Providing management object deletion

Like creating and updating management objects, depending on the type of classes that the provider exposes, some management classes may support delete operations. Use the

BEGIN_DELETE_INSTANCE_MAP macro to start the mapping between management classes and the functions that will serve the requests. Each management class supported by the provider should have an entry in the map. The DELETE_INSTANCE_ENTRY macro is used to route a management class *delete* request to a function responsible with deleting the management object. Finally, the map is completed with the END_DELETE_INSTANCE_MAP macro. Note that if a management class does not support delete operations, it should use the DELETE_INSTANCE_ENTRY_NOTSUPPORTED macro. If this is not done the return code from the provider will indicate that the class is invalid, which is not the case. The following code highlighted in bold shows the map and the only supported routing function, OnBasketDelete.

```
class CFruitBasket :
    ... ... ..
    .. ...
{
public:
    BEGIN_DELETE_INSTANCE_MAP()
       DELETE_INSTANCE_ENTRY(L"SampleWinNET_Basket", OnBasketDelete)
       DELETE_INSTANCE_ENTRY_NOTSUPPORTED(L"SampleWinNET_Fruit")
       DELETE_INSTANCE_ENTRY_NOTSUPPORTED(L"SampleWinNET_BasketFruitMembership")
    END_DELETE_INSTANCE_MAP()

private:
    void OnBasketDelete(LPWBEMINSTANCE pInstance, long lFlags);
    . ...
    ..
};
```

To understand the basics of implementing a deletion function, let's look at a simple example:

```
void CSomeProvider::OnExampleDelete(LPWBEMINSTANCE pInstance, long lFlags)
{
    CComBSTR bstrKeyProperty = BSTR(pInstance->GetProperty(L"Some Property"));

    // TODO: Find and delete management object from local cache

    if (operation failed)
    {
       WMIThrowHRESULT(WBEM_E_PROVIDER_FAILURE);
    }
}
```

The management object to be deleted is passed by the framework through the pInstance parameter. The provider's next step is to obtain the key properties and locate the management object within its own *local cache* and remove it. If delete operation fails, the provider should throw a WBEM_E_PROVIDER_FAILURE HRESULT exception. On leaving the routing function, either the management object has been removed or the request failed. The following code is the implementation of the OnBasketDelete method.

```
void CFruitBasket::OnBasketDelete(LPWBEMINSTANCE pInstance, long lFlags)
{
       USES_CONVERSION;

       CComBSTR bstrBasketPath(L"Software\\WMIBookProv\\Basket");

       // Check basket exists?
       CRegKey regBasket;
       LONG lReg = regBasket.Open(HKEY_LOCAL_MACHINE, OLE2T(bstrBasketPath),
              KEY_WRITE);

       if (lReg == ERROR_SUCCESS)
```

```
        {
                CComBSTR bstrBasketName = BSTR(pInstance->GetProperty(L"Name"));

                // Delete basket from registry
                lReg = regBasket.RecurseDeleteKey(OLE2T(bstrBasketName));
        }

        if (lReg != ERROR_SUCCESS)
        {
                WMIThrowHRESULT(WBEM_E_PROVIDER_FAILURE);
        }
}
```

Providing management query optimization

WMI allows providers to optimize the resulting collection of management objects they return to WMI. Providers do not have to fully parse the WQL query and return the exact required set of management objects to WMI. WMI will do this which is very useful when dealing with complex queries and it also allows for a quicker development time to implement this optimization. To provide management query optimization, use the `BEGIN_QUERY_INSTANCE_MAP` macro to start the mapping between management classes and the functions that will serve the requests. Each management class supported by the provider should have an entry in the map. The `QUERY_INSTANCE_ENTRY` macro is used to route management class retrieval request to a function responsible with exposing it. Finally, the query map is completed with the `END_QUERY_INSTANCE_MAP` macro. Note that if a management class does not support query optimization, it should use the `QUERY_INSTANCE_ENTRY_NOTSUPPORTED` macro. If this is not done the return code from the provider will indicate that the class is invalid, which is not the case. The following code highlighted in bold shows the map and the functions that will expose the query optimization filtered management objects.

```
class CFruitBasket :
    ... ... ..
    .. ...
{
public:
    BEGIN_QUERY_INSTANCE_MAP()
       QUERY_INSTANCE_ENTRY(L"SampleWinNET_Basket", OnBasketQuery)
       QUERY_INSTANCE_ENTRY_NOTSUPPORTED(L"SampleWinNET_Fruit")
       QUERY_INSTANCE_ENTRY_NOTSUPPORTED(L"SampleWinNET_BasketFruitMembership")
    END_ENUM_INSTANCE_MAP()

private:
    void OnBasketQuery(CWbemQuery& query, CWbemInstanceList& instList,
       LPWBEMMETHODCTX ptrs);
    . ...
    ..
};
```

Implementing the query optimization function involves limiting the number of management objects and properties returned to WMI. The optimization can include; only retrieving property values that have been requested. i.e. `SELECT Prop1, Prop2 FROM CIM_SomeClass WHERE PropA="Somevalue"`. Where the only properties that are requested are `Prop1` and `Prop2`. However you should not forget to include `PropA` because WMI will need this information to process more complex queries. The more common optimization is to retrieve a subset of the relevant management objects. That is, evaluating the portion of the query "`PropA="Somevalue""` where `Somevalue` defines the scope of the query and will have a huge impact on performance. Supporting both forms of optimization will increase the performance of the provider. If the query is too complex, always result to retrieving all the management objects and passing them to WMI. WMI will handle the query for you. Let's look at an incomplete example of what the function would look like:

```
void CFruitBasket::OnBasketQuery(CWbemQuery& query,
                                 CWbemInstanceList& instList,
                                 LPWBEMMETHODCTX ptrs)
{
    if (the basket exists)
    {
       //   Add basket instance to instList
       //    [Also, only get properties that are requested in either the
       //     property list or the properties mentioned after the WHERE clause]
    }
    else
    {
       // Otherwise, the query is too complex for us so return all
       // instances to WMI and let it process the query
       OnBasketInstances(instList, ptrs);
    }
}
```

The query is passed in the `query` parameter. The management object enumeration is returned to WMI through the `instList` parameter.

Providing management class and object methods

A WMI provider can also service WMI method requests. To provide method implementations for management classes and objects, use the `BEGIN_METHOD_MAP` macro to start the mapping between management method and the functions that will serve the request. Each management method supported by the provider should have an entry in the map. The `METHOD_ENTRY` macro is used to route management methods based on class name and method name to a function responsible with exposing it. Finally, the management method map is completed with the `END_METHOD_MAP` macro. The following code highlighted in bold shows the map and the functions that will expose the one and only management method.

```
class CFruitBasket :
    ... ... ..
    .. ...
{
public:
    BEGIN_METHOD_MAP()
       METHOD_ENTRY(L"SampleWinNET_Fruit", L"AddFruitToBasket",
          OnFruitAddFruitToBasket)
    END_METHOD_MAP()

private:
    void OnFruitAddFruitToBasket(LPWBEMINSTANCE pInstance,
       LPCWSTR lpszObjectPath, CWbemMethod& rMethod, long lFlags);
    . ...
    ..
};
```

To understand the basics of implementing a management method function, let's look at a simple example:

```
void CSomeProvider::OnExampleMethod(LPWBEMINSTANCE pInstance,
    LPCWSTR lpszObjectPath, CWbemMethod& rMethod, long lFlags)
{
    WMIFunc();

    bool bRetVal = true;

    try
```

```
        {
           // Get the method's in-parameters
           CComBSTR bstrInParam1 = BSTR(rMethod.GetInParam(L"InParam1"));
           CComBSTR bstrInParam2 = BSTR(rMethod.GetInParam(L"InParam2"));

           // TODO: Do something...

           // Set the method's out-parameters
           rMethod.SetOutParam(L"OutParam1", L"Some value");
           rMethod.SetOutParam(L"OutParam2", int(2));
        }
        WMICatch()

        // Set method's return value
        rMethod.Return(bRetVal);
    }
```

The management method implementation first obtains the in-parameters for instruction about what to do. This is followed by the actual task which is later followed by setting the management method's out-parameters including the return value. The following code is the implementation of the `OnFruitAddFruitToBasket` method.

```
    void CFruitBasket::OnFruitAddFruitToBasket(LPWBEMINSTANCE pInstance,
        LPCWSTR lpszObjectPath, CWbemMethod& rMethod, long lFlags)
    {
        USES_CONVERSION;
        WMIFunc();

        bool bRetVal = false;

        try
        {
           // Get the method's in-parameters
           CComBSTR bstrBasketName = BSTR(rMethod.GetInParam(L"BasketName"));
           CComBSTR bstrFruitName = BSTR(rMethod.GetInParam(L"FruitName"));

           // Snsure that we have both parameters
           if (bstrBasketName.Length() > 0 && bstrFruitName.Length() > 0)
           {
              // Helper function to create a map of all the basket and
              // fruit items
              TBasketFruitMap mapFruit;
              GetEnumFruitMap(mapFruit);

              if (FindFruitInMap(bstrFruitName, mapFruit))
              {
                 // We can't add the fruit item it is already added to a basket
                 hr = WBEM_E_ALREADY_EXISTS;
              }
              else
              {
                 // Fruit name not used. Can create and assign the item of fruit.
                 wstring szBasketPath(L"Software\\WMIBookProv\\Basket\\");
                 szBasketPath += bstrBasketName;

                 // Create the basket if it doesn't already exist
                 CRegKey regBasket;
                 LONG lReg = regBasket.Create(HKEY_LOCAL_MACHINE,
                    W2CT(szBasketPath.c_str()), NULL, REG_OPTION_NON_VOLATILE,
```

```
                KEY_WRITE);

            if (lReg == ERROR_SUCCESS)
            {
               // Add Fruit item to basket
               wstring szFruitPath(szBasketPath);
               szFruitPath += L"\\";
               szFruitPath += bstrFruitName;

               // Create and assign item of fruit
               CRegKey regFruit;
               LONG lReg = regFruit.Create(HKEY_LOCAL_MACHINE,
                  W2CT(szFruitPath.c_str()), NULL, REG_OPTION_NON_VOLATILE,
                  KEY_WRITE);

               if (lReg == ERROR_SUCCESS)
                  bRetVal = true;    // Used in WMI methods return value
            }
         }
      }
   }
   WMICatch()

   // Set method's return value
   rMethod.Return(bRetVal);
}
```

Framework public classes

The WMI Provider Framework consists of a number of C++ templates (Table 1) that help to quickly implement instance, method and event WMI providers. Table 2 lists the classes that are used in the C++ templates and provide easy access to management objects, paths, methods, etc… Table 3 lists the functions that may be of use during a providers execution.

Table 1: Provider COM interface C++ template classes

C++ template class	Comment
`IWbemProviderInitImpl`	This C++ template class implements the `IWbemProviderInit` COM interface. It provides basic provider initialization.
`IWbemServicesImpl`	This C++ template class implements the `IWbemServices` COM interface. It provides basic provider implementation for all provider instance and method operations. Implementation's of each type of operation is routed through macros to C++ functions responsible for servicing the requests.
`IWbemInternalEventProviderImpl`	This C++ template class implements the `IWbemEventProvider` COM interface. It provides an implementation that allows the provider to fire events from within other operations of the instance or method provider.

Table 2: Provider C++ classes

C++ class	Comment
`CWbemObjectPath`	A helper class that parses and creates object paths.
`CWbemQuery`	A helper class to parse WQL and SQL management queries. This helps to break down a query when implementing provider query optimization.
`CWbemInstance`	This class represents a management object.
`CWbemInstanceList`	This class represents a collection of management objects.
`CWbemMethod`	This class makes it easy to set and get method parameters when implementing a method provider.

`CWbemVariantCast`
 	
`WBEMKEYREFS`
	A helper class to cast a VARIANT into a C++ equivalent type.
	
	This is a structure used to represent a key property in an object path.

Table 3: Provider C++ helper functions

Helper functions	Comment
`GetThisComputerName`	Returns the current computer name.
`GetClientBlanketName`	Returns the name of the user (principle) making the DCOM call. This function is useful when implementing provider routing methods to determine who is making the request.

IWbemProviderInitImpl template class

This class provides a basic provider initialization implementation. The derived class should override `OnInitialize` for specific provider initialization. The `m_spWbemServices` member is the provider's management services interface provided by WMI. Here is the public interface of the template class:

```
template <class T>
class IWbemProviderInitImpl : public IWbemProviderInit
{
public:
    CComPtr<IWbemServices> m_spWbemServices;

    // You can derive from this in your base class
    void OnInitialize(LPWSTR wszUser, LONG lFlags, LPWSTR wszNamespace,
        LPWSTR wszLocale, IWbemContext *pCtx)
    {

    }

    STDMETHOD(Initialize)(LPWSTR wszUser, LONG lFlags,
        LPWSTR wszNamespace, LPWSTR wszLocale,
        IWbemServices *pNamespace, IWbemContext *pCtx,
        IWbemProviderInitSink *pInitSink);
};
```

Initialize method

```
STDMETHOD(Initialize)(LPWSTR wszUser, LONG lFlags,
    LPWSTR wszNamespace, LPWSTR wszLocale,
    IWbemServices *pNamespace, IWbemContext *pCtx,
    IWbemProviderInitSink *pInitSink)
```

This method adds a reference to the management services provided by WMI and stores it in the `m_spWbemServices` member. The implementation of `Initialize` calls `OnInitialize`. This is where you should add your own provider initialization.

IWbemServicesImpl template class

This template class implements the routing of management requests to the appropriate method handler. The methods listed in the following class are the ones implemented by the template class. All other methods return WBEM_E_NOT_SUPPORTED. Here is the public interface of the template class:

```
template <typename T>
class IWbemServicesImpl : public IWbemServices
{
public:
    STDMETHOD(GetObjectAsync)(const BSTR ObjectPath, long lFlags,
        IWbemContext *pCtx, IWbemObjectSink *pResponseHandler);

    STDMETHOD(PutInstanceAsync)(IWbemClassObject *pInst, long lFlags,
        IWbemContext *pCtx, IWbemObjectSink *pResponseHandler);

    STDMETHOD(DeleteInstanceAsync)(const BSTR ObjectPath, long lFlags,
        IWbemContext *pCtx, IWbemObjectSink *pResponseHandler);

    STDMETHOD(CreateInstanceEnumAsync)(const BSTR Class, long lFlags,
        IWbemContext *pCtx, IWbemObjectSink *pResponseHandler);

    STDMETHOD(ExecQueryAsync)(const BSTR QueryLanguage, const BSTR Query,
        long lFlags, IWbemContext *pCtx, IWbemObjectSink *pResponseHandler);
```

```
        STDMETHOD(ExecMethodAsync)(const BSTR strObjectPath, const BSTR strMethodName,
            long lFlags, IWbemContext *pCtx, IWbemClassObject *pInParams,
            IWbemObjectSink *pResponseHandler);

        LPWBEMINSTANCE CreateWbemInstance(LPWBEMMETHODCTX ptrs);
    };
```

GetObjectAsync method

This method is implemented by using the following macros to route the request to a member function.

```
BEGIN_OBJECT_INSTANCE_MAP()
    OBJECT_INSTANCE_ENTRY(L"Sample_Class", OnObject)
END_OBJECT_INSTANCE_MAP()
```

The member function prototype is:

```
void OnObject(LPWBEMINSTANCE pInstance, long lFlags);
```

The requested management object specified using key properties is passed in through `pInstance` and it is the provider's responsibility to set the remaining non-key properties of the management object.

PutInstanceAsync method

This method is implemented by using the following macros to route the request to a member function.

```
BEGIN_PUT_INSTANCE_MAP()
    PUT_INSTANCE_ENTRY(L"Sample_Class", OnPut)
    PUT_INSTANCE_ENTRY_NOTSUPPORTED(L"Sample_OtherClass")
END_PUT_INSTANCE_MAP()
```

The member function prototype is:

```
void OnPut(LPWBEMINSTANCE pInstance, long lFlags);
```

The new or updated management object is passed in through `pInstance` and it is the provider's responsibility to persist the changes depending on the flags passed in by `lFlags`. If the provider supports a management class, but not for this operation, it should declare it using, `PUT_INSTANCE_ENTRY_NOTSUPPORTED`.

DeleteInstanceAsync method

This method is implemented by using the following macros to route the request to a member function.

```
BEGIN_DELETE_INSTANCE_MAP()
    DELETE_INSTANCE_ENTRY(L"Sample_Class", OnDelete)
    DELETE_INSTANCE_ENTRY_NOTSUPPORTED(L"Sample_OtherClass")
END_DELETE_INSTANCE_MAP()
```

The member function prototype is:

```
void OnDelete(LPWBEMINSTANCE pInstance, long lFlags);
```

The management object to be deleted is passed in through `pInstance` and it is the provider's responsibility to read the key properties and remove the object. If the provider supports a management class, but not for this operation, it should declare it using, `DELETE_INSTANCE_ENTRY_NOTSUPPORTED`.

CreateInstanceEnumAsync method

This method is implemented by using the following macros to route the request to a member function.

```
BEGIN_ENUM_INSTANCE_MAP()
    ENUM_INSTANCE_ENTRY(L"Sample_Class", OnInstances)
END_ENUM_INSTANCE_MAP()
```

The member function prototype is:

```
void OnInstances(CWbemInstanceList& instList, LPWBEMMETHODCTX ptrs);
```

Management objects are returned through `instList` and the current context is passed through `ptrs`. The context is always used when making calls back to WMI from within a provider. The context may also have additional context provided by the client/caller that the provider can understand.

ExecQueryAsync method

This method is implemented by using the following macros to route the request to a member function.

```
BEGIN_QUERY_INSTANCE_MAP()
    QUERY_INSTANCE_ENTRY(L"Sample_Class", OnQuery)
    QUERY_INSTANCE_ENTRY_NOTSUPPORTED(L"Sample_OtherClass")
END_ENUM_INSTANCE_MAP()
```

The member function prototype is:

```
void OnQuery(CWbemQuery& query, CWbemInstanceList& instList, LPWBEMMETHODCTX ptrs);
```

The management query is passed in through `query` and the `CWbemQuery` class can be used to parse it. Management objects are returned through `instList` and the current context is passed through `ptrs`. The context is always used when making calls back to WMI from within a provider. The context may also have additional context provided by the client/caller that the provider can understand. If the provider supports a management class, but not for this operation, it should declare it using, `QUERY_INSTANCE_ENTRY_NOTSUPPORTED`.

ExecMethodAsync method

This method is implemented by using the following macros to route the request to a member function.

```
BEGIN_METHOD_MAP()
    METHOD_ENTRY(L"Sample_Class", L"SomeMethod", OnSomeMethod)
END_METHOD_MAP()
```

The member function prototype is:

```
void OnSomeMethod(LPWBEMINSTANCE pInstance, LPCWSTR lpszObjectPath,
    CWbemMethod& rMethod, long lFlags);
```

If the method pertains to a management object, then the provider should get the key properties from `pInstance` to identify which object the method needs to execute against. If you need to the object path, then you can get at it directly through `lpszObjectPath`. The method's in and out parameters and be obtained and set through `rMethod`.

CreateWbemInstance method

```
LPWBEMINSTANCE CreateWbemInstance(LPWBEMMETHODCTX ptrs);
```

This helper method creates an instance of the same type as that identified by the provider's context. The caller is responsible for releasing the return value.

IWbemInternalEventProviderImpl template class

This template class is only for use within a WMI instance of method provider. On many occasions, a provider may want to fire an event while it is processing a request. The provider should call `FireEvent` whenever it is ready to fire an event. Here is the public interface of the template class:

```
template <class T>
class IWbemInternalEventProviderImpl : public IWbemEventProvider
{
public:
       CComPtr<IWbemObjectSink> m_spEventSink;

public:
       STDMETHOD(ProvideEvents)(IWbemObjectSink *pSink, long lFlags);
```

```
        void FireEvent(CWbemInstance& event);
};
```

FireEvent method

```
void FireEvent(CWbemInstance& event);
```

This method fires an event within an instance or method provider. The event to be fired should be passed in `event`. Let's look at an example of how you can do this to fire a new fruit basket `__InstanceCreationEvent` event.

```
//
// Basket just created! Fire event
//
CWbemInstance event;
pInstance->CreateInstanceOfClass(L"__InstanceCreationEvent", event);

// Set target instance management object
CWbemInstance eventTargetInst;
pInstance->CreateInstanceOfClass(L"SampleWin2K_Basket", eventTargetInst);
eventTargetInst.SetProperty(L"Name", bstrBasketName);
eventTargetInst.SetProperty(L"Capacity", int(0));

CComPtr<IWbemClassObject> spTargetInst;
eventTargetInst.GetInstance(&spTargetInst);

CComPtr<IUnknown> spUnk;
spTargetInst.QueryInterface(&spUnk);
CComVariant varTargetInst(spUnk);

// Add target instance to event
event.SetProperty(L"TargetInstance", varTargetInst);

// Fire management event
FireEvent(event);
```

If you are planning to fire events within a provider, remember to inherit from it. For example:

```
class CFruitBasket :
    public CComObjectRoot,
    public CComCoClass<CFruitBasket,&CLSID_FruitBasket>,
    public IWbemProviderInitImpl<CFruitBasket>,
    public IWbemServicesImpl<CFruitBasket>,
    public IWbemInternalEventProviderImpl<CFruitBasket>
{
public:
    CFruitBasket() {}
    ... ... ..
    .. ...
    ..
};
```

Don't forget to setup the .mof file to register your provider as an event provider. For more information on how to do this, go to Chapter 12.

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);
};
```

CWbemQuery class

This class helps you to parse management queries. The implementation of this class is limited under Windows XP/Server 2003 because it uses the query facilities provided by WMI. Refer to MSDN for more information. The Windows 2000 version provides many more methods to extract details of the query.

```
class CWbemQuery
{
public:
    CWbemQuery(LPCWSTR lpszWQLQuery, LPCWSTR lpszQueryLang = L"WQL");

    // ** NOTE!! This is a weak reference!!!
    void GetParsedQuery(SWbemRpnEncodedQuery** ppEncodedQuery);

    void ExtractClass(wstring& rszClass);
};
```

CWbemInstance class

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

```
class CWbemInstance
{
public:
    CWbemInstance();
    CWbemInstance(IWbemClassObject* pInst);
    CWbemInstance(LPWBEMMETHODCTX ptrs);

    void Create(IWbemClassObject* pInst);
    void Create(LPCWSTR lpszClassName, LPCWSTR lpszNamespace = L"root\\CIMV2");
    void Create(LPWBEMMETHODCTX ptrs, LPCWSTR lpszNamespace = L"root\\CIMV2");

    void CreateInstanceByKeyPath(LPCWSTR lpszObjectPath, CWbemInstance& rInst);
    void CreateFakeInstanceByKeyPath(LPCWSTR lpszObjectPath, CWbemInstance& rInst);
    void CreateInstanceOfClass(LPCWSTR lpszClass, CWbemInstance& rInst);

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

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

        void SetObjPathProperty(LPCWSTR lpszClassName, LPCWSTR lpszPropName,
           WBEMKEYREFS ppKeyRefs[], int nCount);

        void Indicate();

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

Create method

These methods setup a `CWbemInstance` object.

```
void Create(IWbemClassObject* pInst);
void Create(LPCWSTR lpszClassName, LPCWSTR lpszNamespace = L"root\\CIMV2");
void Create(LPWBEMMETHODCTX ptrs, LPCWSTR lpszNamespace = L"root\\CIMV2");
```

The first method basically attaches an existing management object to the class and adds a reference. The second method creates a new instance of a specific class and attaches to it. The third method creates a new instance of the class specified by the context and attaches to it.

CreateInstanceByKeyPath method

This helper method performs an `IWbemServices::GetObject` call and attaches to the returned management object.

```
void CreateInstanceByKeyPath(LPCWSTR lpszObjectPath, CWbemInstance& rInst);
```

Pass the object path into the `lpszObjectPath` parameter and if successful, the management object is returned through `rInst`.

CreateFakeInstanceByKeyPath method

This helper method creates an in-memory instance of the management object represented by the object path. The key properties in the object path are set as properties on the newly created instance. This is useful when you want an `IWbemClassObject` of the supposed management object.

```
void CreateFakeInstanceByKeyPath(LPCWSTR lpszObjectPath, CWbemInstance& rInst);
```

Pass the object path into the `lpszObjectPath` parameter and if successful, the fake management object is returned through `rInst`.

CreateInstanceOfClass method

This helper method creates a new instance of a specific class.

```
void CreateInstanceOfClass(LPCWSTR lpszClass, CWbemInstance& rInst);
```

Pass the class reference into the `lpszClass` parameter and if successful, the new instance is returned through `rInst`.

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.

SetObjPathProperty method

This helper method helps you set properties that are management object references.

```
void SetObjPathProperty(LPCWSTR lpszClassName, LPCWSTR lpszPropName,
    WBEMKEYREFS ppKeyRefs[], int nCount);
```

The name of the management class that will be referenced is passed through `lpszClassName` and the name of the property in the attached instance (i.e. the class that contains the reference) is passed through `lpszPropName`. For each key property of the class passed in through `lpszClassName` must be setup. The number of array elements in `ppKeyRefs` is passed through `nCount`. The following is an extract from some code described earlier in this article.

```
// Basket reference
WBEMKEYREFS keyRef[1];
keyRef[0].lpszKeyRefName = L"Name";
keyRef[0].cimtypeValue = CIM_STRING;
keyRef[0].varKeyRefValue =
    CComVariant((*itrFruit).second.szBasketName.c_str());

pInst->SetObjPathProperty(L"SampleWinNET_Basket", L"Basket", keyRef, 1);
```

Indicate method

This method is used to pass the management object to WMI.

```
void Indicate();
```

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.

CWbemInstanceList class

This class holds a collection of management objects. Calling `Add` will include the management object specified through `pInst` to the collection. If an object of the class is setup with a WMI provided sink, `Add` also calls `Indicate` to pass the management object to WMI. `DeleteAll` removes all the management object references from the collection.

```
class CWbemInstanceList : public std::list<LPWBEMINSTANCE>
{
public:
       CWbemInstanceList(IWbemObjectSink* pResponse = NULL);
       ~CWbemInstanceList();

       void Add(CWbemInstance* pInst);
       void DeleteAll();
};
```

CWbemMethod class

This class represents a method. You can get the method's in-parameters by calling `GetInParam` and the out-parameters can be set using `SetOutParam`. Use `Return` to set the methods return value.

```
class CWbemMethod
{
public:
	CWbemMethod();
	CWbemMethod(LPWBEMMETHODCTX ptrs, LPCWSTR lpszMethodName,
		IWbemClassObject* pInParams);

	CWbemVariantCast GetInParam(LPCWSTR lpszPropName);

	template <typename ValType>
	void SetOutParam(LPCWSTR lpszPropName, ValType value);

	template <typename ValType>
	void Return(ValType value);
};
```

ExecMethodAsync method

This method is implemented by using the following macros to route the request to a member function.

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();
};
```

WBEMKEYREFS class

Although not for public use, this class allows the provider's control callback function to set current provider state and also set session handles in shared memory. This is then accessed by the `CTraceEvent` class when tracing events. This implementation design allows DLLs and COM components to fire events for the tracing session held by the provider.

```
struct WBEMKEYREFS
{
	LPCWSTR lpszKeyRefName;
	CIMTYPE cimtypeValue;
	CComVariant varKeyRefValue;
};
```

GetThisComputerName function

This returns the name of the current machine through `rszName`.

```
inline bool GetThisComputerName(wstring& rszName);
```

GetClientBlanketName function

This returns the name of the principle/user who is accessing the provider through `szPrincipleName`.

```
inline void GetClientBlanketName(wstring& szPrincipleName)
```