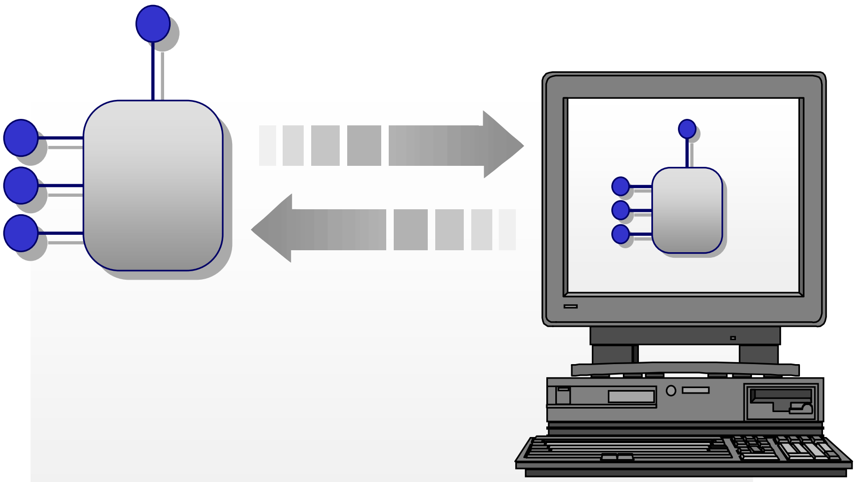
COM Fundamentals (Server)



COM Programming

55

Chapter Overview

QA-IQ

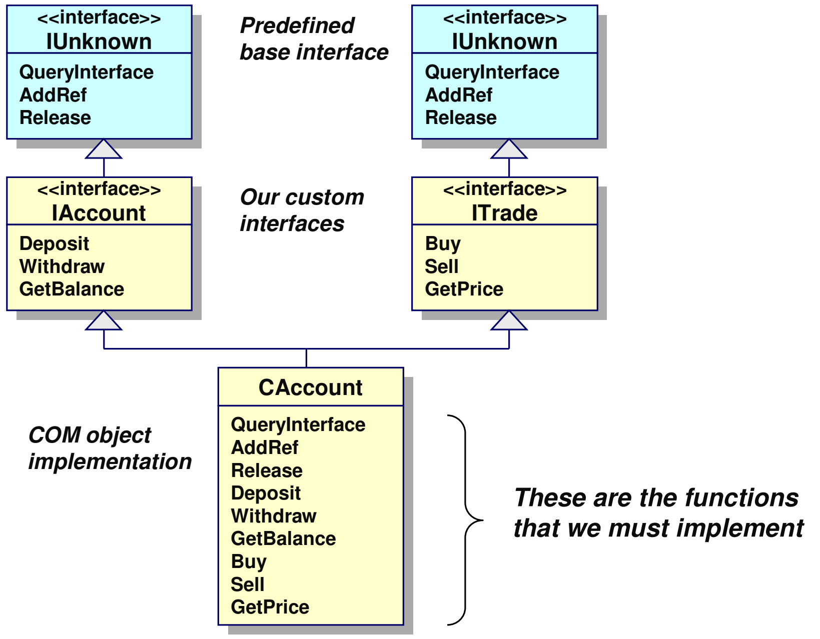
- **Objective**
  - **Introduce COM fundamentals**
- **Chapter content**
  - **QueryInterface() rules and implementation**
  - **Implementing Reference counting**
  - **The COM Service Control Manager**
  - **DLL-based COM servers**
  - **Locating components - The registry**
- **Practical content**
  - **Develop and test a simple reusable component using C++**
- **Summary**

56

Implementation of Interfaces

QA-IQ

- **Use multiple inheritance**



57

In the following slides, we will look at how to develop a simple COM object, `CAccount`, that implements our custom interfaces, `IAccount` and `ITrade`. As the class diagram shows, we can achieve this using multiple inheritance. Also, we will need to implement the functions of `IUnknown` in addition to the functions of `IAccount` and `ITrade`.

Implementing QueryInterface()

QA-IQ

- **Return pointer to requested interface or NULL**

```
// CAccount.h
class CAccount : public IAccount, public ITrade
{ ... };
```

HRESULT __stdcall

```
// CAccount.cpp
STDMETHODIMP CAccount::QueryInterface(REFIID riid, void **ppv)
{
  if (riid == IID_IAccount)
    *ppv = static_cast<IAccount*>(this);
  else if (riid == IID_ITrade)
    *ppv = static_cast<ITrade*>(this);
  else if (riid == IID_IUnknown) // would be ambiguous
    *ppv = static_cast<IUnknown*>(static_cast<IAccount*>(this));
    // cast to IUnknown via IAccount
  else {
    *ppv = 0;
    return E_NOINTERFACE;    // not supported
  }
  reinterpret_cast<IUnknown *>( *ppv )->AddRef();
  return S_OK;               // success
}
```

Can use == operator to compare IIDs

58

`QueryInterface()` allows a client to request a pointer to a specified interface. The first thing you will notice is that we've used another macro `STDMETHODIMP`, which on Win32 platforms is equivalent to `HRESULT STDMETHODCALLTYPE` where `STDMETHODCALLTYPE` equals `__stdcall`.

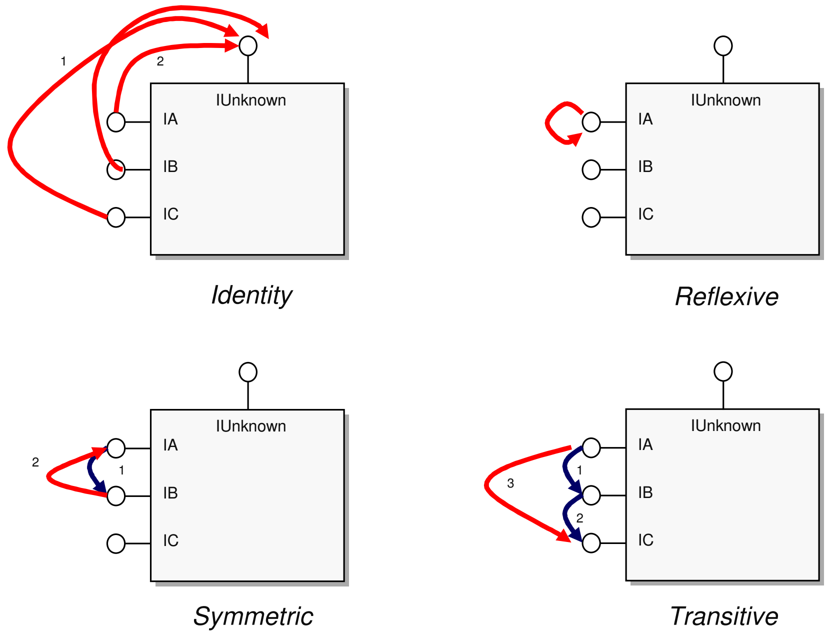
There are various ways we could implement `QueryInterface()`, but the simplest is to use an if … else if … statement to compare the requested IID riid with each supported interface, i.e. `IAccount`, `ITrade` and `IUnknown`. As shown on the slide, it's possible to compare IIDs using the == operator because of the overloaded == operator and inline GUID functions defined in objbase.h. If there's a match, we then use the `static_cast<>` operator to cast the this pointer to the appropriate type; otherwise we return a null pointer as required by the COM specification.

Note that when a pointer to the `IUnknown` interface is requested, we must statically cast to either `IAccount` or `ITrade`. A static cast to `IUnknown` would be ambiguous because `IUnknown` is a common base class, which means that there will be multiple copies of its member functions in the COM object's vtable.

Note that the pointer to a COM object's `IUnknown` interface establishes its identity, so we must always be consistent. In other words, elsewhere in the implementation of our COM object, we must remember that the pointer to `IUnknown` is equivalent to the pointer to `IAccount` and not to to the pointer to `ITrade` (because, of course, these pointers will be different).

If our COM object implemented many different interfaces, a better way to implement `QueryInterface()` might be use a hash table or a map.

QueryInterface() Rules



59

The implementation of QueryInterface() must conform to the rules of identity, consistency, reflexivity, symmetry and transitivity.

The pointer to an COM object's IUnknown interface establishes its identity. Therefore, if a client acquires a pointer to a COM object's IUnknown interface, subsequent calls to QueryInterface() requests for that object's IUnknown pointer must return the same value, no matter through which interface QueryInterface() is called. This allows a client to determine whether two IUnknown pointers refer to the same COM object by simply comparing their values.

QueryInterface() must be consistent. If a call to QueryInterface() for a specific interface succeeds the first time, it must succeed on all subsequent calls. Conversely, if it fails the first time, it must fail on all subsequent calls.

QueryInterface() must be reflexive. If a client acquires an interface pointer through QueryInterface(), subsequent calls for the same interface must succeed.

QueryInterface() must be symmetric. If a client calls QueryInterface() on one interface to acquire a pointer to a second interface, a subsequent call to QueryInterface() through the second interface to acquire a pointer to the first interface must succeed.

QueryInterface() must be transitive. If a client calls QueryInterface() on one interface to acquire a pointer to a second interface, and then calls QueryInterface() through the second interface to acquire a pointer to third interface, a single call to QueryInterface() through the first interface to acquire a pointer to the third interface must also succeed.

These rules enable COM to optimise network communications, by caching interface pointers on proxies, reducing the number of round trips across the wire. Therefore, we can confidently call QueryInterface() without worrying unduly about performance

Implementing AddRef() & Release()

QA·IQ

- **Use Win32 APIs for thread-safe increment/decrement**

```
// CAccount.h
class CAccount : public IAccount, public ITrade {
  private:
    long m_lRefCount;
};
```

Initialised to zero by constructor

```
// CAccount.cpp
...
STDMETHODIMP_(ULONG) CAccount::AddRef() {
  //return ++m_lRefCount;  // not thread-safe
  return InterlockedIncrement(&m_lRefCount);
}

STDMETHODIMP_(ULONG) CAccount::Release() {
  //if (--m_lRefCount == 0) // not thread-safe
  if (InterlockedDecrement(&m_lRefCount) == 0) {
    delete this;
    return 0;    // object no longer exists!
  }
  return m_lRefCount;
}
```

ULONG STDMETHODCALLTYPE

60

As shown on the slide, the implementation of reference counting in a COM object can be very simple. We simply need to add a member variable to hold the reference count, which will be initialised to zero by the constructor, and override the `AddRef()` and `Release()` functions of `IUnknown`. We've used another macro `STDMETHODIMP_`, which is defined in objbase.h as follows:

```
#define STDMETHODIMP_(type) type STDMETHODCALLTYPE
```

There are a couple of points to note. First, because the reference count is a `long`, which is not atomic, we've used the Win32 functions `InterlockedIncrement()` and `InterlockedDecrement()` to perform a thread-safe increment and decrement, respectively. Second, in the `Release()` function, we can't return the value of `m_lRefCount` after deleting the object, because, of course, the object no longer exists.

As mentioned on the previous slide, `QueryInterface()` must also increment the reference count as follows:

```
STDMETHODIMP CAccount::QueryInterface( REFIID riid,
                                       void **ppv )
{
  if( iid == IID_IAccount )
    *ppv = static_cast<IAccount*>(this);
  ...
  ...
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}
```

The Service Control Manager (SCM)

- A single DLL or EXE can contain many coclasses
  - For efficiency
- All coclasses loaded on demand by the SCM
  - Service Control Manager
- The SCM locates (via registry) and loads components
  - After handing an interface pointer back to client
  - It drops out of the picture

61

COM classes and their associated class objects may be packaged into DLLs or EXEs.

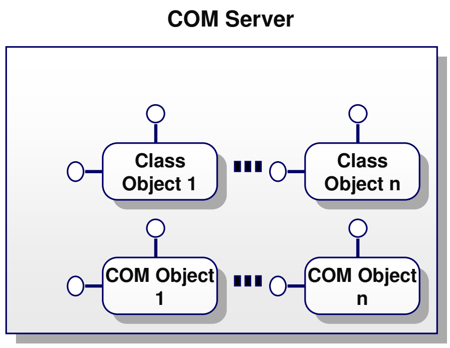
DLLs and EXEs containing COM classes are not loaded directly by the client. This is the responsibility of the COM class loader - the COM Service Control Manager (pronounced "scum").

The SCM uses CLSID configuration information contained in the COM database (currently the registry) to locate the host DLL or EXE and then loads it appropriately. The SCM contacts the class object for the requested CLSID, and returns its interface pointer back to the client.

At this stage the SCM has done its job and drops out of the picture. From here on in, the client talks directly to the class object (or COM object if `CoCreateInstance()` was used) via its newly acquired interface.

Packaging Components

- One or more COM objects can be implemented as:
  - A DLL (in-proc) server, or
  - An EXE (local) server
- Each class of COM object has an associated class object
  - Implements IClassFactory
  - Creates instance of COM object
- To implement a DLL server:
  - Implement the class object
  - Implement the exported DLL functions
  - Generate a GUID to identify the COM object's class
  - Build the DLL
  - Register the COM object with the Windows Registry



62

One or more COM objects can be implemented as a DLL. Since a DLL executes in the context of a client process, a DLL server is also known as an in-proc (*in process*) server.

COM objects can also be served in the context of a separate process or EXE. If this EXE executes on the same machine as the client, it is also known as a local server; otherwise it is known as a remote server. In this chapter, we will look only at DLL servers; we will look a EXE servers in a later chapter. However, the only differences between the implementation of a DLL server and an EXE server are in the registration of its COM objects and the loading and unloading of the server itself; no changes are required to the COM object's implementation.

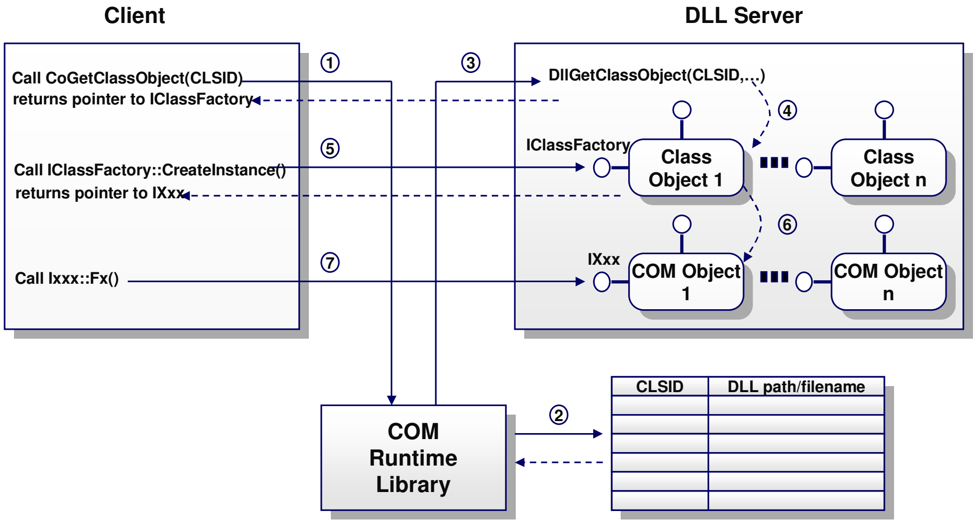
Irrespective of whether a COM object is served by a DLL or an EXE, each class of publicly-creatable COM object requires an associated class object, as shown in the diagram above.

A class object's sole purpose is create objects of the COM class with which it is associated. A class object is often referred to as a class factory, because the vast majority of them implement the interface `IClassFactory`.

Creating a COM Object (DLL)

QA-IQ

- **Each COM object's class has a class ID (CLSID)**
- **System database maps CLSID to DLL location**
  - **Implemented in Registry on Windows 9x/NT/2000**



63

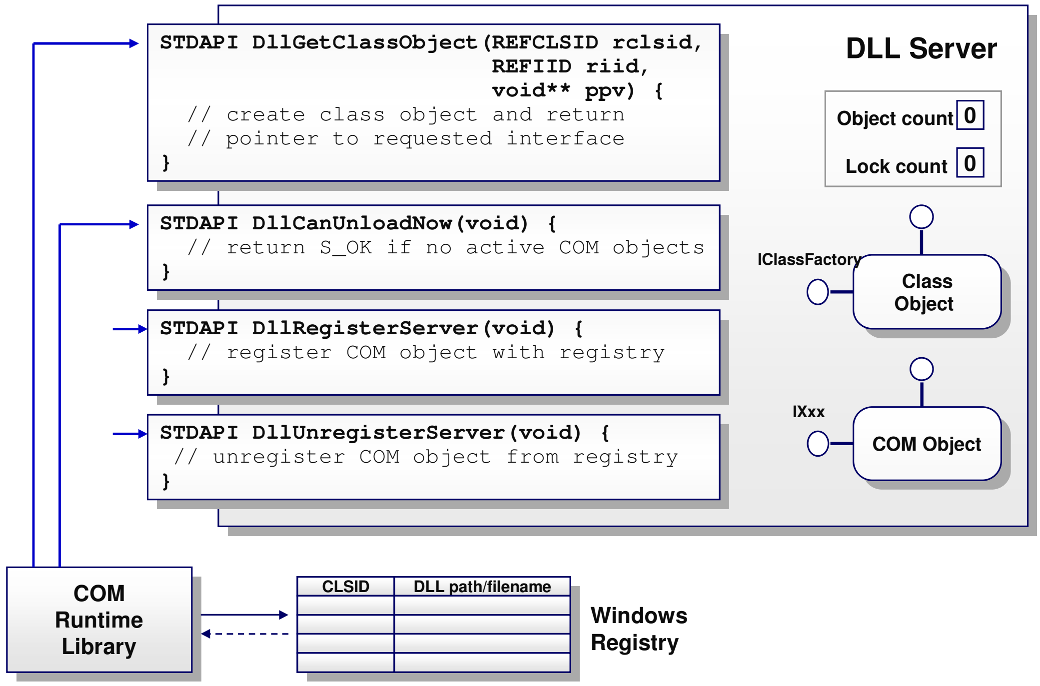
COM uses GUIDs not only to uniquely identify interfaces, but also to identify COM classes. Each publicly-creatable COM class therefore has a unique class ID or CLSID. Information about the classes of COM objects is held in a system database, which on Windows 9X/NT is the Windows Registry, and this information includes a mapping of CLSIDs to DLL (or EXE) locations.

The diagram above illustrates the process of implementing a COM object as a DLL server.

1. A client calls the COM library function, `CoGetClassObject()`, specifying the CLSID of the COM object to create.
2. The COM runtime looks up the CLSID in the Registry. If it finds an in-proc server entry, it loads the specified DLL.
3. The COM runtime calls the `DllGetClassObject()` function in the DLL server, specifying the CLSID of the COM object to create.
4. If the class object has not already been instantiated, `DllGetClassObject()` creates an instance of the class object that corresponds to the specified CLSID.* Then, `DllGetClassObject()` queries the class object for a pointer to its `IClassFactory` interface, which is returned to the client via the COM runtime.
5. The client calls the `CreateInstance()` function of the `IClassFactory` interface, specifying the IID of the interface Ixxx.
6. `CreateInstance()` uses the new operator to instantiate the COM object and immediately queries it for a pointer to its IXxx interface, which it returns to the client. Finally, `CreateInstance()` releases the class object.
7. The client can now call a function of the COM object's IXxx interface.

* It is normal to create a class object statically, in which case there is no to instantiate it.

Implementing a DLL Server



64

A DLL server must export two functions, `DllGetClassObject()` and `DllCanUnloadNow()` as shown in the diagram above. The other two functions, `DllRegisterServer()` and `DllUnregisterServer()` are optional, and are only used to support self registration.

Notice that all of these functions are declared as STDAPI, which expands to `extern "C" HRESULT __stdcall` through the following definitions in objbase.h:

```
#define EXTERN_C extern "C"
#define STDAPICALLTYPE __stdcall
#define STDAPI EXTERN_C HRESULT STDAPICALLTYPE
```

In order to build the DLL, we'll also need to provide a definition file that lists the exported functions, as follows:

```
LIBRARY AccountComponent.dll

EXPORTS
  DllCanUnloadNow        @1   PRIVATE
  DllGetClassObject      @2   PRIVATE
  DllRegisterServer      @3   PRIVATE
  DllUnregisterServer    @4   PRIVATE
```

The use of ordinals is optional, whereas the PRIVATE keyword prevents the linker from putting the function's name in the import library, which is not required because COM dynamically load DLL servers using a function called `CoLoadLibrary()`.

DllGetClassObject()

- **Returns pointer to requested class-factory interface**

```
// account.h

extern "C" const CLSID CLSID_Account;
```

```
// account.cpp
...
static CAccountFactory g_AccountFactory;
...
STDAPI DllGetClassObject( REFCLSID rclsid,
                          REFIID riid,
                          void** ppv )
{
  if( CLSID_Account == rclsid )
  {
    // return pointer to requested interface
    return g_AccountFactory.QueryInterface( riid, ppv );
  }
  *ppv = 0;
  return CLASS_E_CLASSNOTAVAILABLE;
}
```

```
extern "C" HRESULT
__stdcall
```

Normally
`IID_IClassFactory`

65

`DllGetClassObject`(), which is called by the COM runtime in response to a client call to `CoGetClassObject`() or `CoCreateInstance`(), takes three parameters.

The first parameter specifies the class ID of the COM object to create, which is important to a DLL server that supports more than one class of COM object.

The second parameter specifies the interface ID of the class-factory interface requested by the client. Normally, this would be `IID_IClassFactory`, although it may be any custom interface ID.

The third parameter is the placeholder for the return of the requested pointer.

The DLL server in our simple example supports only one class of COM object, so if the specified class ID isn't CLSID_Account, we set the returned pointer to 0 (as required by the COM specification) and return the strangely-named HRESULT code CLASS_E_CLASSNOTAVAILABLE.

If the client has specified a CLSID that we are implementing, we simply query the class object for the specified interface and return the returned pointer and HRESULT code.

Subsequently, this pointer will be returned to the caller of `CoGetClassObject`().

Loading and Unloading the DLL

QA-IQ

- **COM runtime library loads DLL server dynamically**
  - **Calls CoLoadLibrary(), etc**
- **It can also unload DLL server dynamically**
  - **If DllCanUnloadNow() returns S_OK, it calls CoFreeLibrary()**
- **DllCanUnloadNow() should only return S_OK when:**
  - **There are no active COM objects in the DLL**
    - i.e. global object count is zero
  - **Server has not been locked by call to IClassFactory::LockServer()**
    - i.e. global lock count is zero

66

The COM runtime loads a DLL server dynamically on demand, for example, in response to client calling `CoGetClassObject`(). The COM runtime can also unload a DLL server, but only if the DLL confirms that it's safe to do so, that is, it does not have any active COM objects and the global lock-count is zero.* The DLL must therefore track the number of active COM objects by maintaining a global object-count that is incremented in a COM object's constructor and decremented in its destructor.

Incidentally, note that class objects should not be included in this object count.

The DLL must also track the calls to the `LockServer()` function of the `IClassFactory` interface of a class factory using a global lock-count as discussed previously. A client can call `IClassFactory::LockServer()` to prevent a DLL from being unloaded even when it doesn't have any active COM objects. This is useful for performance reasons, because having acquired a pointer to the `IClassFactory` interface of a class factory, a client can create multiple instances of a COM class without the overhead of reloading the DLL if it has been subsequently been unloaded.

Before the COM runtime unloads a DLL server in response to a client calling `CoFreeUnusedLibraries()`, it checks that it's OK to do so by calling the DLL's `DllCanUnloadNow()` function. All this function needs to do is to return `S_OK` if both the object count and the lock count are zero, or `S_FALSE` if they are not. In fact, since both counts determine the lifetime of the DLL, they could be, and often are, combined into a single count.

* The COM runtime functions `CoLoadLibrary()` and `CoFreeLibrary()` are similar to their Win32 counterparts, except that a DLL loaded through `CoLoadLibrary()` can be freed by calls to `CoFreeUnusedLibraries()` and `CoUnitialize()`, as well as `CoFreeLibrary()`.

DllCanUnloadNow()

QA-IQ

- **Return S_OK if both DLL and object counts are zero**

```
// account.h
class CAccount : public IAccount {
public:
  CAccount() : m_lRefCount(0) {
    InterlockedIncrement(&g_lObjCount);
  }

  ~CAccount() {
    InterlockedDecrement(&g_lObjCount);
  }
};
```

Increment object count in constructor

Decrement object count in destructor

```
// account.cpp
long g_lObjectCount = 0;  // global object count
long g_lLockCount = 0;

STDAPI DllCanUnloadNow(void)
{
  if( 0 == g_lLockCount && 0 == g_lObjectCount )
    return S_OK;
  return S_FALSE;
}
```

67

Here is our implementation of `DllCanUnloadNow()`. As mentioned on the previous slide, we maintain a global object count that is incremented in a COM object's constructor and decremented in its destructor. The global lock count is incremented and decremented by `LockServer()`.

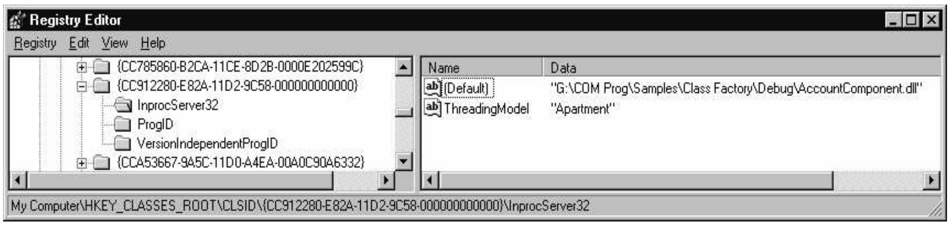
If both the object count and the lock count are zero, `DllCanUnloadNow()` returns `S_OK`; otherwise, it returns `S_FALSE`.

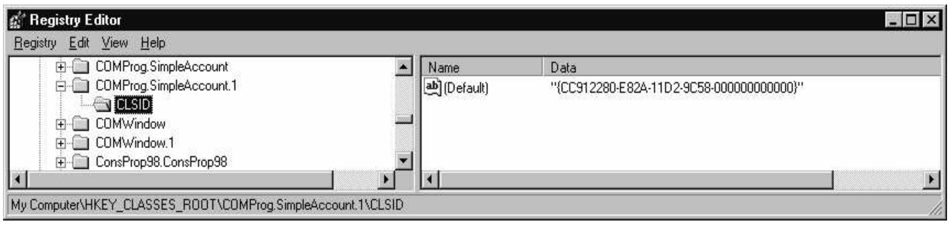
The use of global variables, as shown above, is normally replaced with a wrapper class that exposes methods to increment or decrement a private counter. This prevents the somewhat ugly use of `extern`ing global variables.

Finding COM Objects

QA-IQ

- **COM object must be registered in Registry**
  - **Bare minimum is mapping of CLSID to DLL/EXE location**
  - **Can also provide mapping of "user-friendly" ProgID to CLSID**





68

A COM object must be registered in the Registry under HKEY_CLASSES_ROOT. If the COM object is implemented as a DLL server, an entry is required under the [HKEY_CLASSES_ROOT\CLSID\<clsid>\InprocServer32] key - which specifies the full path and filename of the DLL, where <clsid> is the CLSID of the COM object. This CLSID will be used by a C++ client when it creates a COM object, either by calling `CoGetClassObject()` or `CoCreateInstance()`.

Some programming languages, such as Visual Basic, refer to COM objects using programmatic IDs or ProgIDs instead of CLSIDs. A ProgID is a "user-friendly" name that is mapped in the Registry to a CLSID (directly under HKEY_CLASSES_ROOT). The format of a ProgID is:

<program name>.<COM object name>.<version number>

A COM class usually has a second VersionIndependentProgID that maps to the CLSID of the latest version of the COM object installed on the system. The format of a VersionIndependentProgID is simply:

<program name>.<COM object name>

Of course, ProgIDs are not guaranteed to be unique, so there's always the chance of a name clash.

It should be noted that the COM runtime ONLY uses the CLSID registry data. ProgIDs are converted by the language runtime into a CLSID *before* it passes the request through to the COM runtime.

Registration Functions

QA-IQ

- **DllRegisterServer() and DllUnregisterServer() allow COM object to be self-registering**
  - **Requirement for Active X controls**
  - **Usually called by setup program**
  - **Can also be called from RegSvr32 utility**

```
C:\>regsvr32 G:\COM Prog\Samples\Class Factory\Debug\AccountComponent.dll
```

- **Implementation requires use of Win32 registry API (!)**
  - **As an interim measure, you can describe required entries in a registration file for use with RegEdit**

account.reg

```
REGEDIT4

HKEY_CLASSES_ROOT\CLSID\{CC912280-E82A-11d2-9C58-000000000000}]
@="COM Programming, Simple Account Example"

[HKEY_CLASSES_ROOT\CLSID\{CC912280-E82A-11d2-9C58-000000000000}\
InprocServer32]
@="G:\\COM Prog\\Samples\\Class Factory\\Debug\\AccountComponent.dll"
```

69

Most COM objects should be self-registering, which means that a DLL server must export two functions that can be called by an installer (i.e. setup) program. These functions must be called `DllRegisterServer()` and `DllUnregisterServer()`, respectively. During development, you can use the RegSvr32 tool, which is supplied with many development tools including Visual Studio.

If you are familiar with the Win32 registry API, implementing `DllRegisterServer()` and `DllUnregisterServer()` is straightforward, but rather tedious. However, it's boilerplate code, so if you have a copy of either Inside COM or Inside DCOM on your bookshelf, you should find a source file called registry.cpp on its accompanying CD.

ATL provides a way to automate the generation of code for `DllRegisterServer()` and `DllUnregisterServer()` from a simple script, as we'll discover in a later chapter, but for the time being, we'll create a simple registration file for use with RegEdit. An example of such a file is shown on the slide.

To add this information to the registry, simply double-click on the file in Windows Explorer, or type in the command: regedit account.reg.

Implementing a COM Server

QA-IQ

```
struct IAAA : public IUnknown
{
    virtual HRESULT stdcall f1() = 0;
    virtual HRESULT stdcall f2() = 0;
    virtual HRESULT stdcall f3() = 0;
};
```

```
DllGetClassObject()    { ... }
DllCanUnloadNow ()     { ... }
DllRegisterServer()    { ... }
DllUnregisterServer()  { ... }
```

```
class MyFactory : public IClassFactory
{
public:
    STDMETHOD(QueryInterface) {...}
    STDMETHOD_(ULONG, AddRef) {...}
    STDMETHOD_(ULONG, Release) {...}
    STDMETHOD(CreateInstance) {...}
    STDMETHOD(LockServer) {...}
};
```

```
class MyObject : public IAAA
{
public:
    STDMETHOD(QueryInterface)
{...}
    STDMETHOD_(ULONG, AddRef)
{...}
    STDMETHOD_(ULONG,
Release) {...}
    STDMETHOD(f1) {...}
    STDMETHOD(f2) {...}
    STDMETHOD(f3) {...}
private:
    // state
};
```

The implementation of a COM server can be divided into three separate areas:

1) Implementations of the 4 exported functions:

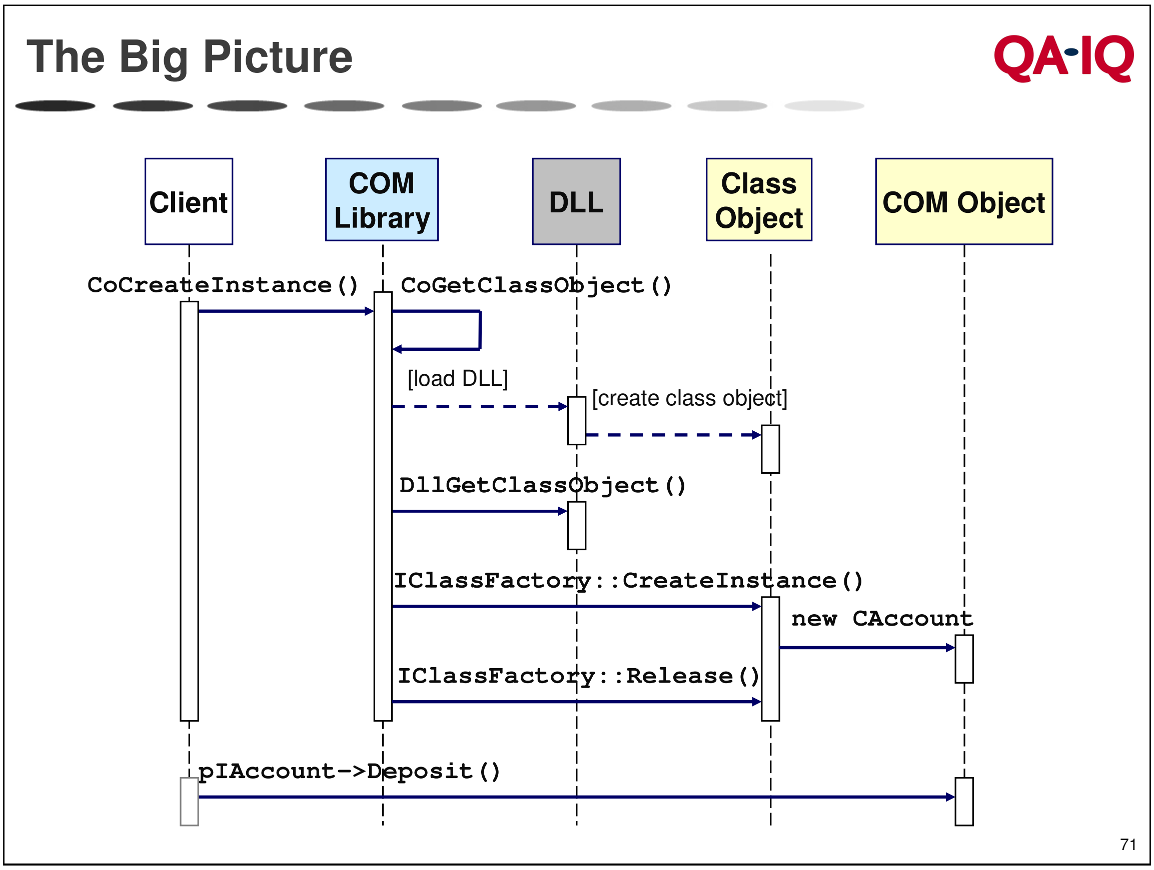
```
DllGetClassObject()    { ... }
DllCanUnloadNow ()     { ... }
DllRegisterServer()    { ... }
DllUnregisterServer()  { ... }
```

2) Implementation code for each interface supported. In the example above only one interface (IAAA) is implemented. The code for each of the methods for this interface and any interfaces from which it derives must be provided. Here we must implement the 3 methods of `IUnknown` and the 3 additional methods of `IAAA`:

```
STDMETHOD(QueryInterface) {...}
STDMETHOD(AddRef) {...}
STDMETHOD(Release) {...}
STDMETHOD(f1) {...}
STDMETHOD(f2) {...}
STDMETHOD(f3) {...}
```

3) Implementation code for the class object. Most class objects will implement the `IClassFactory` interface (2 methods) which in turn is derived from `IUnknown` (3 methods); a total of 5 methods to be implemented:

```
STDMETHOD(QueryInterface) {...}
STDMETHOD(AddRef) {...}
STDMETHOD(Release) {...}
STDMETHOD(CreateInstance) {...}
STDMETHOD(LockServer) {...}
```



In the sequence diagram above, we can clearly see the key interactions between the COM library, the DLL server, the class object and the COM object itself, when a client calls `CoCreateInstance()`. These interactions are as follows:

A client calls the COM library function, `CoCreateInstance()`, specifying the CLSID of the COM object to create and the IID of the requested interface.

`CoCreateInstance()`, in turn, calls `CoGetClassObject()`, which looks up the CLSID in the Registry. If it finds an *InprocServer32* entry, the COM runtime loads the specified DLL, which creates the class object (in a data segment).

The COM runtime calls the DLL's `DllGetClassObject()` function in the DLL server. `DllGetClassObject()` queries the class object for a pointer to its `IClassFactory` interface, which is returned to the COM runtime.

The COM runtime calls the `CreateInstance()` function of the `IClassFactory` interface, specifying the IID of the interface `IAccount`.

`CreateInstance()` uses the `new` operator to create an instance of the `CAccount` class and immediately queries it for a pointer to its `IAccount` interface, which is subsequently returned to the caller of `CoCreateInstance()`. Finally, the COM runtime releases the class object.

Using the pointer returned from `CoCreateInstance()`, the client calls the `Deposit()` function of the `CAccount` object's `IAccount` interface.

Summary

QA-IQ

- QueryInterface() allows a client to request a pointer to a specified interface
- A COM object must implement reference counting
  - Clients of COM objects must follow the rules
- COM objects have associated class objects
  - These must be implemented in the server
- Activation is handled by the COM SCM
  - With a little help from a few registry entries!

72