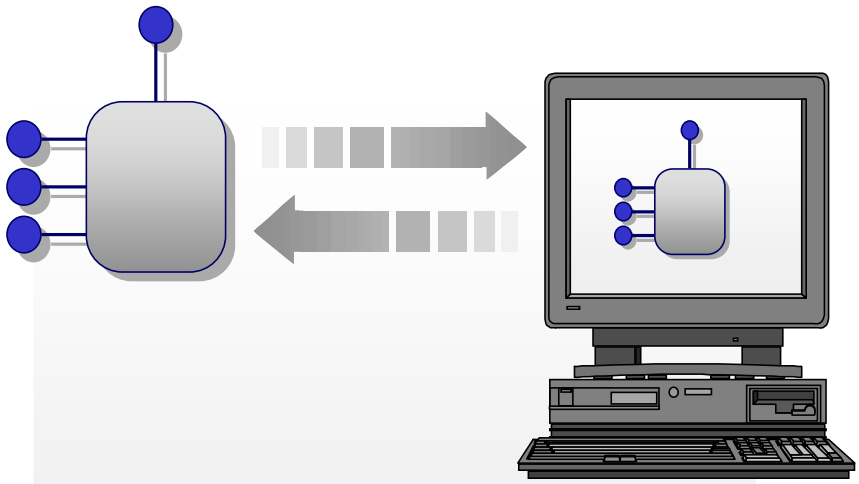
COM Fundamentals (Client)

QA-IQ



COM Programming

37

Chapter Overview

QA-IQ

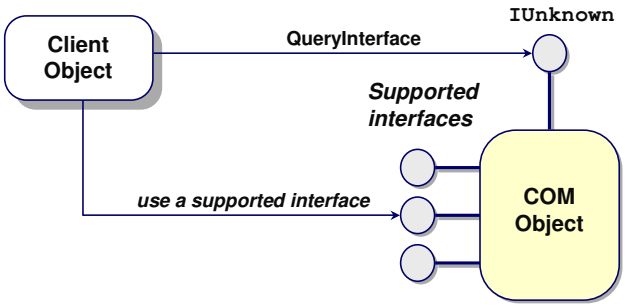
- **Objective**
  - **Introduce the fundamentals of a COM client**
- **Chapter content**
  - **Interfaces and interface identifiers**
  - **The IUnknown interface**
  - **Reference counting**
  - **HRESULTs**
- **Practical content**
  - **Test a simple reusable component using C++**
- **Summary**

38

COM Basics

QA-IQ

- **A COM object**
  - **Separates its interface(s) from its implementation**
  - **Provides multiple services through separate interfaces**
- **All COM objects implement an interface IUnknown**
  - **Includes a QueryInterface() function that clients can use to acquire pointers to COM object's other interfaces**
  - **Also includes AddRef() and Release() functions for reference counting**



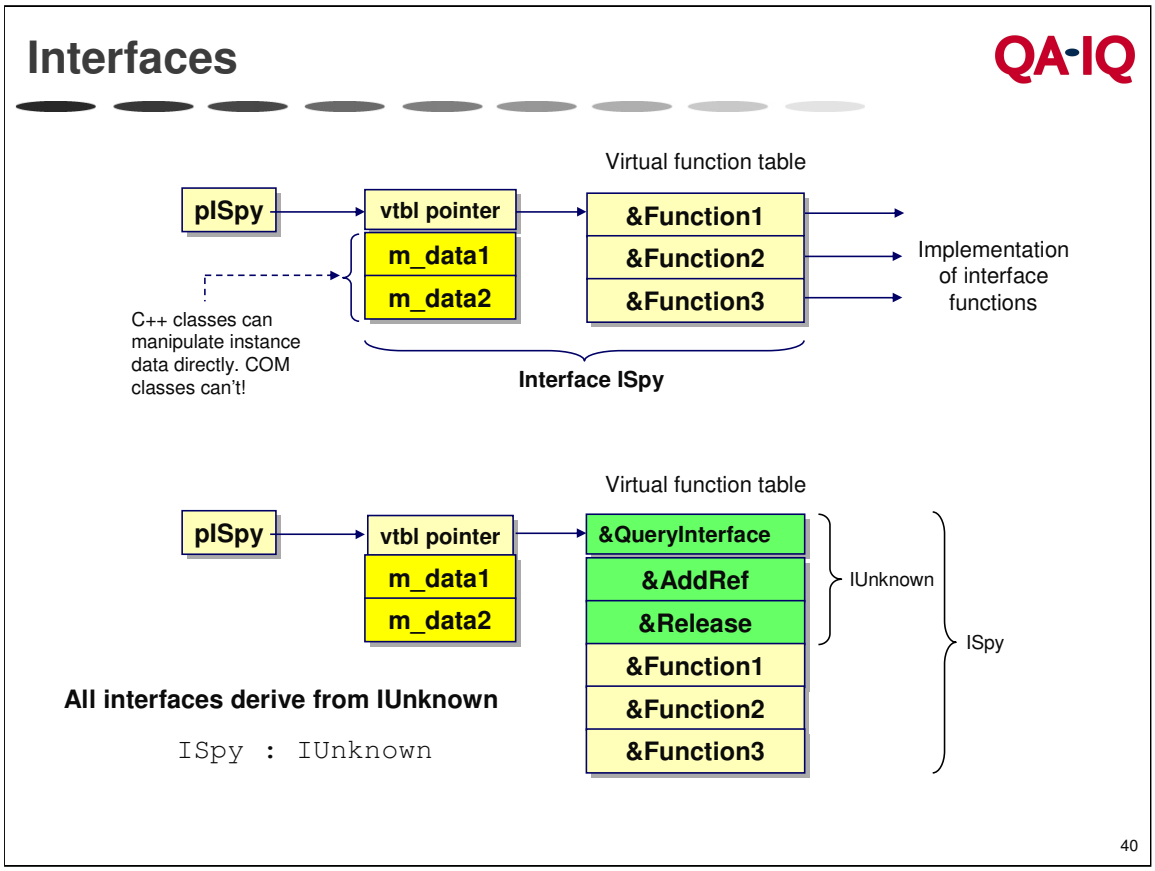
39

A "COM object" is a reusable "component", and like most COM literature, we will tend to use these terms interchangeably.

The `IUnknown` interface is the fundamental interface that every COM object must implement, and from which every COM interface must derive. This interface is used by clients to discover, at runtime, what other interfaces a COM object provides and then to make a connection to them. This is the responsibility of the `QueryInterface()` method.

There are two other methods in `IUnknown` that allow COM objects manage their lifecycle using reference counting. The `AddRef()` and `Release()` methods increment and decrement a reference counter every time a client connects to it and releases it. When this reference count goes to zero, the object can safely delete itself from memory.

As you can see from the notation in the picture above, you cannot tell how a COM object is implemented. All that is visible are its interfaces (represented by the individual "jacks" on the object). Of course, this means that a COM object might be implemented by one (or more) C++ classes, or in Visual Basic or even in 'C' (only for the masochists amongst us).



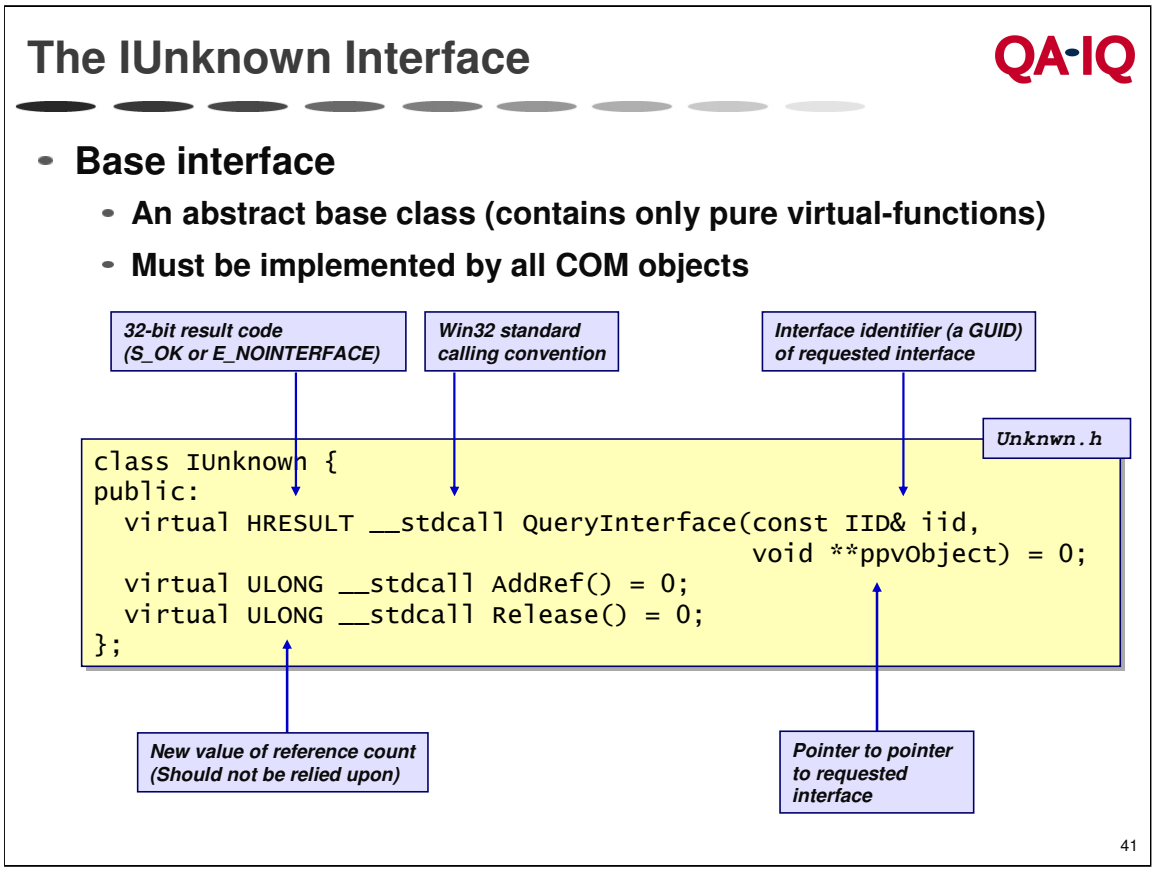
As we've seen, an interface is represented by a standard binary layout in memory. Any class wishing to implement a particular interface must be capable of constructing this binary layout. Typically in C++ this is achieved via inheritance, using abstract base classes.

For example class `CA` wishing to implement the interface `ISpy` would derive from `ISpy`:

```
class CA : public ISpy
{
    . . .
}
```

A COM interface is defined as a pointer to a pointer (the vtbl pointer) to an array of function pointers. The above illustration shows how an arbitrary interface called ISpy containing three functions would be laid out. The class implementing this interface may also contain instance data. By contrast with C++ which allows direct client manipulation of this instance data, COM explicitly prevents such access. If a COM class wishes to expose it's instance data (or *properties*) it needs to do so by providing a set of methods for manipulation - traditionally a "*getter*" and a "*setter*" method for each property (although this approach is flawed in a distributed environment due to the overhead introduced by numerous method calls).

One of the fundamental rules of COM is that all interfaces must ultimately derive from the `IUnknown` interface, which itself contains three functions. This means that every interface in existence will contain the same three functions in the first three slots of its vtable - namely `QueryInterface()`, `AddRef()` and `Release()`. These methods as we shall see are fundamental to the operation and lifecycle of a COM object.



In C++, a COM interface can be represented by an abstract base class that contains only pure virtual functions. Microsoft has specified a number of standard interfaces, the most important of which, `IUnknown`, is defined in Unknwn.h. As shown on the slide, IUnknown contains three pure virtual functions: `QueryInterface()`, `AddRef()` and `Release()`. Like all interface functions, these are declared `__stdcall`, which specifies the Win32 standard calling convention.

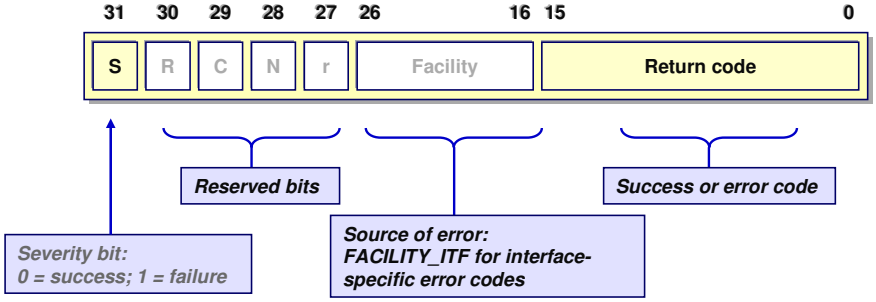
`QueryInterface()` allows a client to request a pointer to a specified interface. This function takes two arguments: a reference to an interface identifier (IID) and a pointer to a memory location to receive the returned pointer. (Because `QueryInterface()` must be able to query for any interface, this parameter is declared as `void**`.) It returns an HRESULT, a 32-bit result code, which will generally be one of two values: `S_OK` or `E_NOINTERFACE`.

The other two functions, `AddRef()` and `Release()`, are used with reference counting. Unlike, `QueryInterface()`, these functions return the new value of the reference count as a `ULONG`. However, this value maybe unreliable and should only be used for diagnostic purposes (proxy objects may not forward all calls to `AddRef()` and `Release()` to the real object).

HRESULTs

QA-IQ

- Interface functions should not throw Win32 or C++ exceptions
  - Not meaningful to all types of clients
  - Cannot cross process boundaries
- Instead, interface functions should return an HRESULT
  - 32-bit value that contains success or error code
  - AddRef() and Release() are only exceptions to this rule



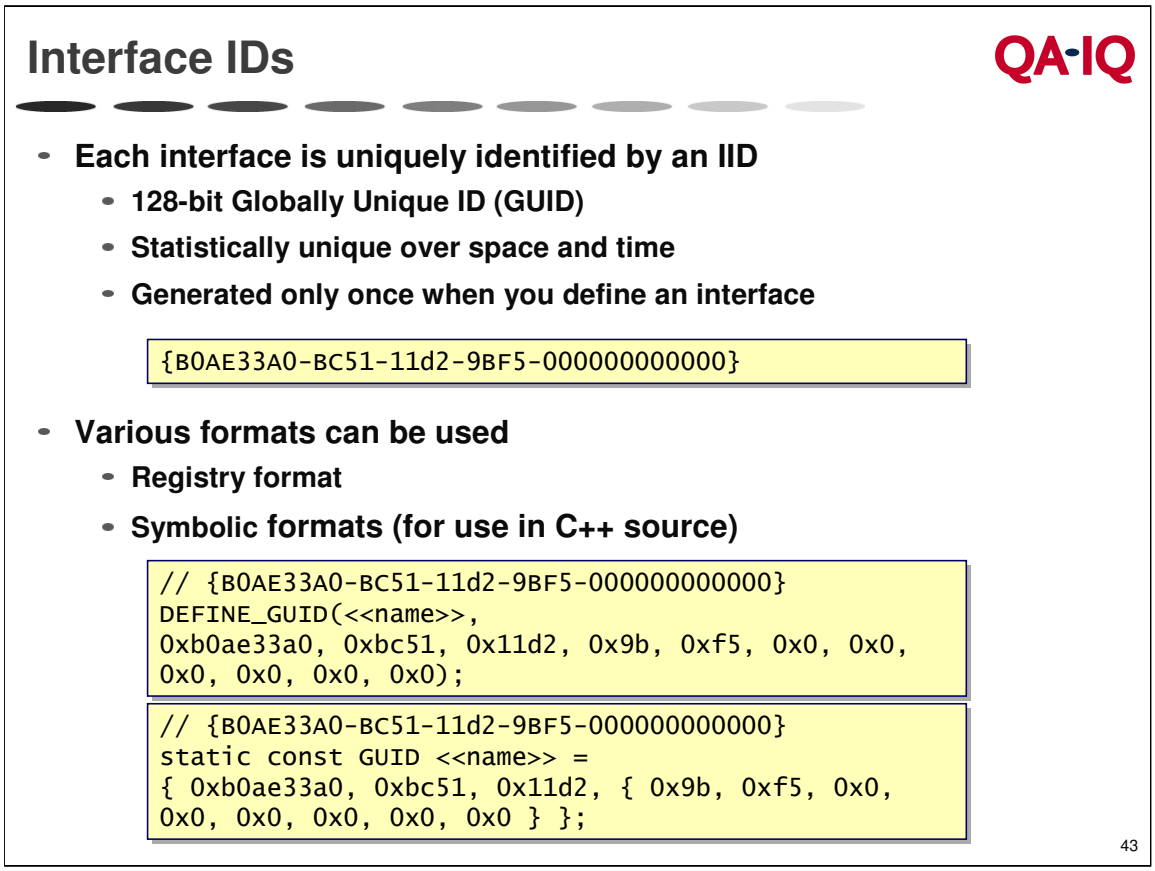
42

Interface functions are not allowed to throw Win32 or C++ exceptions, because non-C++ clients would be unable to handle them. Besides, COM does not provide a mechanism to allow raw exceptions to cross process boundaries. Therefore, all interface functions, with the exception of `AddRef()` and `Release()`, should return an HRESULT. In fact, the MIDL compiler, which we'll cover in a later chapter, will not produce marshaling code for any interface that contains a function that does not return an HRESULT.

An HRESULT is not a handle; it's simply a ULONG (i.e. a 32-bit value) that contains a success or error code in a structured format. As shown on the slide, it contains five individual bits (S, R, C, N and r), an 11-bit facility code and a 16-bit return code. The most important bit is the S or Severity bit because it indicates success (zero) or failure (one). The R, C, N and r bits are reserved.

The facility code identifies the source (e.g. subsystem) of the HRESULT and is used to classify return codes or to identify the subsystem that returned the HRESULT. Microsoft has defined a number of standard facility codes such as FACILITY_WIN32 (Win32 API), FACILITY_RPC (RPC calls) FACILITY_DISPATCH (IDispatch interface) and FACILITY_ITF (custom interface). The return code (bits 0 to 15) define the actual success or error code.

Non-Microsoft developers are not allowed to define new facility codes, so error codes returned by functions of custom (i.e. non-standard) interfaces must all use the FACILITY_ITF facility code. Furthermore, the associated return code must be in the range 0x200 to 0xFFFF to avoid conflicting with Microsoft-defined codes in the range 0x0000 to 0x01FF.



Anybody can develop a new COM interface, so it's clearly impossible to uniquely identify a COM interface just by its textual name. For example, if we develop a new interface called IAccount, how are we be sure that this name doesn't clash with the name of an existing interface defined by another developer, or one that may be defined in the future? Therefore, to avoid the need for a central authority for dispensing interface IDs, a COM interface is uniquely identified by a 128-bit number called a GUID.

A GUID, which stands for Globally Unique Identifier, can be generated, without reference to any central authority, using an algorithm defined by the Open Software Foundation (OSF), originally for DCE RPCs. This algorithm takes the MAC address of the network card in the machine on which the GUID is generated, the current time (in 100 ns intervals since 00:00:00, 15 October 1582) and a random seed to generate a 128-bit (16 byte) number that is statistically unique over space and time. If the machine doesn't have a network card, another algorithm to used to generate a unique number for *that machine only*. Since a GUID is a 128-bit number, it can be represented in various formats as shown on the slide. In the Windows Registry, and in documentation, it is represented as a hex string in the form {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}. For example, the IID of `IUnknown` is defined as {00000000-0000-0000-0000-000000000046}. The other formats are designed to represent a GUID symbolically in C++ source code. GUID is a structure that is defined in wtypes.h as:

```
typedef struct _GUID {
  DWORD Data1;
  WORD Data2;
  WORD Data3;
  BYTE Data4[ 8 ];
} GUID;
```

Using QueryInterface()

QA-IQ

- **Use IID to specify required interface**
  - **IID_IAccount is a variable of type IID (GUID)**

```
IAccount *pIAccount = 0;

HRESULT hr = pIUnkown->QueryInterface( IID_IAccount,
                                       (void**) &pIAccount );
```

Important to remember
Must pass in address of interface pointer
Have to cast interface pointer to a void**

44

As every interface inherits from `IUnknown`, `QueryInterface()` can be called on an instance of any interface pointer.

An IID is used to specify the required interface. This IID will either be an instance of an IID included in the project or will be generated using the `__uuidof()` Microsoft extension to C++, the use of which will be seen later.

As the name implies an interface pointer is a pointer and thus the address of the variable must be passed in if the contents of the pointer are to be changed. The variable must also be cast to a `void**` to allow `QueryInterface()` to be so generic.

When using QueryInterface(), it is imperative that the type of the interface pointer matches the IID that is passed in. QueryInterface() can be the source of subtle and hard to find bugs, caused when

(a) the IID doesn't match the type of interface pointer, or

(b) instead of passing in the address of an interface pointer, the pointer is passed.

Good client coding practices, such as initialising your interface pointers to 0, can help to overcome some of these problem, but in the end you will need to rely on careful client coding to help. We will see one mechanism, smart pointers, that will help to overcome this problem later in the course.

Reference Counting

QA-IQ

- **COM object determines its own lifetime**
  - **Maintains a reference count**
  - **Incremented by a call to AddRef() or a successful call to QueryInterface()**
  - **Decremented by a call to Release()**
  - **When all reference counts drop to zero, deletes itself**

```
interface IUnknown {
  STDMETHOD(QueryInterface)(REFIID riid, void **ppv) PURE;
  STDMETHOD_(ULONG, AddRef)() PURE;
  STDMETHOD_(ULONG, Release)() PURE;
};
```

- **Clients of COM objects must follow the rules:**
  - **Call AddRef() after duplicating an interface pointer**
  - **Call Release() when an interface is no longer required**
  - **Failure to follow these rules can result in memory leaks!**

45

A COM object must manage its own lifetime by maintaining a "usage" or reference count. This count must be incremented when a client obtains another pointer to an interface and decremented when such a pointer is no longer required. A zero reference count therefore indicates that none of the interfaces are in use, so the object can then safely destroy itself.

Of course, the COM object relies totally on clients to keep it informed about their usage of its interfaces. This is the reason for the `AddRef()` and `Release()` methods in `IUnknown`. Whenever a client duplicates an interface pointer, it must call `AddRef()`. Similarly, whenever a client has finished with an interface pointer, it must call `Release()`. Failure to follow these basic rules will result in memory faults, or worse still, memory leaks, which can be very difficult to track down.

To conform to the COM specification, the COM object must also increment the reference count automatically before it returns an interface pointer to a client, for example, as a result of a call to `QueryInterface()`. Therefore, clients rarely need to call `AddRef()`.

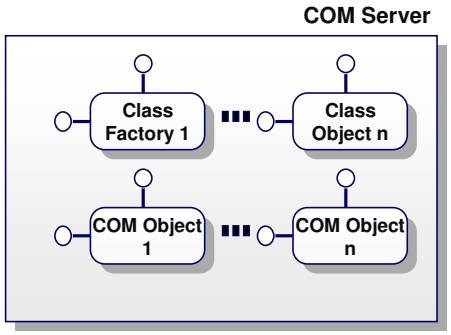
Although, from the client's point of view, a reference count is maintained on each of a COM object's interfaces, a single reference count (for the entire COM object) may actually be implemented.

Clients should not depend on return values from `AddRef()` and `Release()`.

Class Factories

QA-IQ

- A COM Class (or coclass), is a named body of code
  - That implements one or more COM interfaces
- All coclasses are identified by a CLSID (GUID)
- All coclasses have an associated "class factory"
  - Often implementing IClassFactory
  - These are often referred to as "class objects"



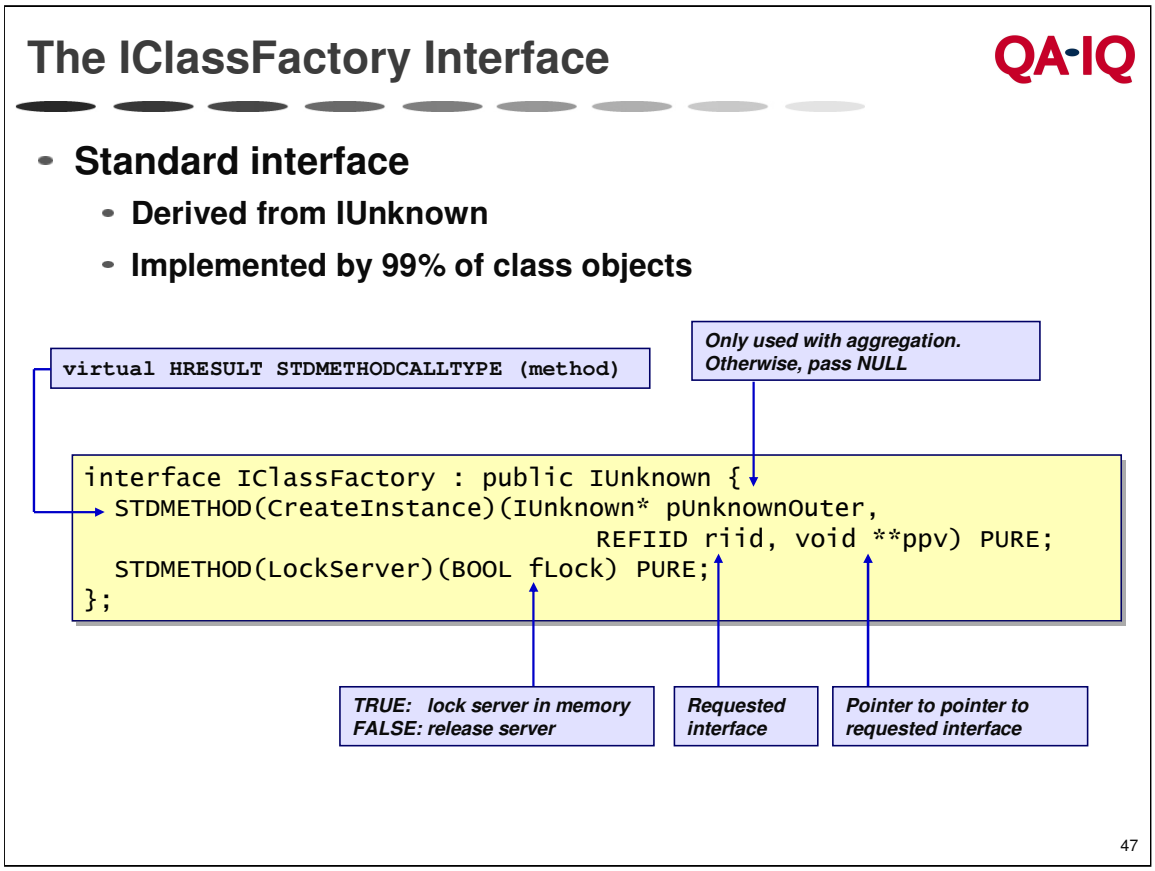
46

A COM class or *coclass* is a named concrete implementation of one or more COM interfaces. COM classes are identified via a GUID referred to as the CLSID (pronounced "class_ID").

Each and every coclass has an associated class object which is responsible for creating instances of the associated class i.e. creating COM objects. Class objects are often referred to as class factories - the manufacturing analogy explains the name.

A large number of COM class objects also implement an interface called IClassFactory - an interface which is discussed subsequently. When a class object implements IClassFactory it can be referred to as a class factory, but strictly, the correct term is class object.

The class object's purpose in life is to create new instances of the COM object. This abstraction enables both client developers, the COM runtime and other runtime engines (such as VB) that protect client developers from the underlying complications of COM, to work with many different types of COM objects.

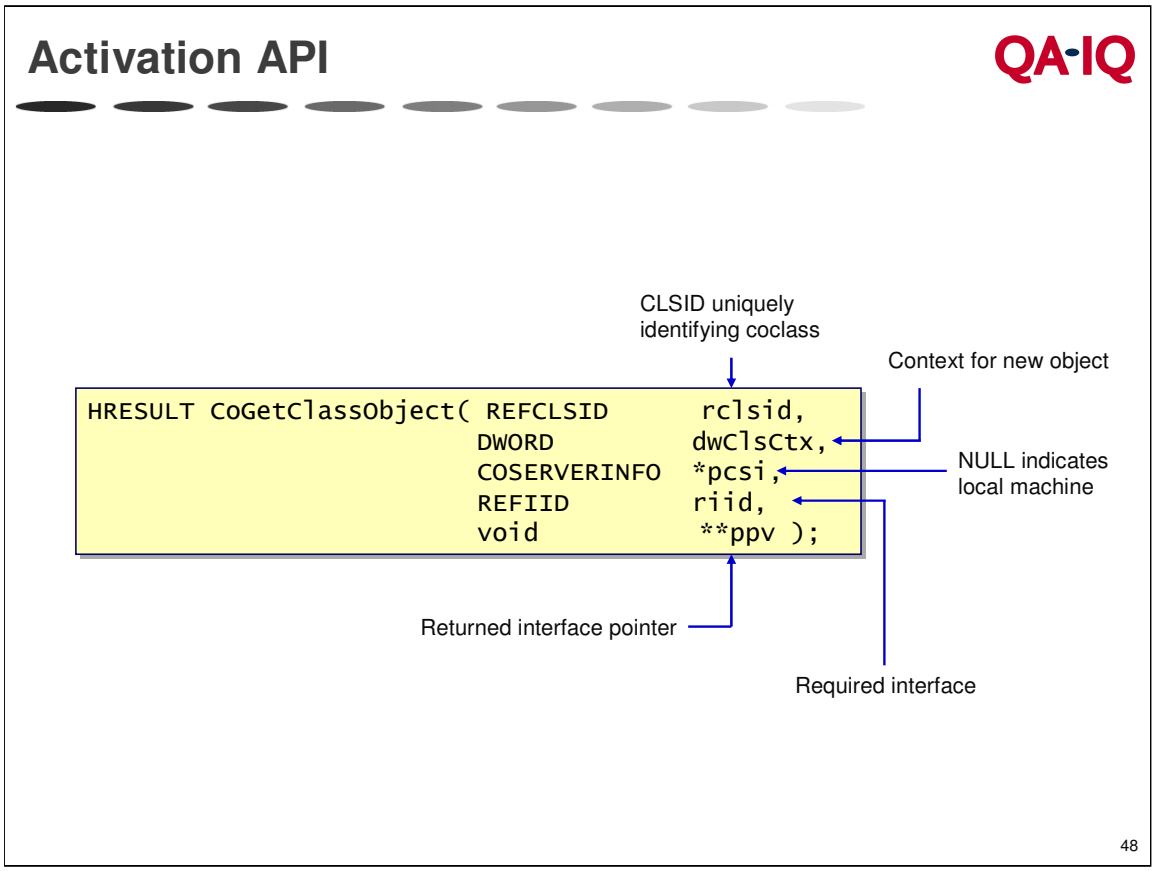


`IClassFactory` is a standard interface, which is derived from `IUnknown`, and contains two additional member functions: `CreateInstance()` and `LockServer()`, as shown on the slide. Like IUnknown, it is declared in Unknwn.h.

`CreateInstance()` creates an instance of a COM object with a specific CLSID, queries it for the requested interface in riid, and returns the interface pointer in ppv. If the COM object is contained within an outer COM object (a technique for reusing classes, known as aggregation), the pointer to the `IUnknown` interface of the outer COM object must be passed in `pUnknownOuter`. If aggregation is not being used, `NULL` is passed for this parameter.

Note that `CreateInstance()` does not have a CLSID parameter, because, of course, a class factory can only create instances of the one and only COM object with which it is associated.

`LockServer()` allows a client to lock a DLL or EXE server in memory until it has finished with it. It takes a single boolean parameter, `fLock`, which if true locks the server in memory, and if false releases it.



COM provides a number of activation APIs. Here we will just look at the first of these, `CoGetClassObject()`.

This lower level API returns an interface pointer to a class object. You supply a CLSID identifying the required COM class, a context (specifying the desired locality of the server - local, remote etc), an interface identifier (this time identifying the interface required from the *class object*) and as output the interface is returned via the `ppv` parameter.

If you require any other interface than `IClassFactory` such as some other type of custom interface to a class object, you must call `CoGetClassObject`() directly.

The other advantage to calling this API directly, is that you can cache the class object interface pointer. If multiple instances of a particular COM class are to be created, this allows you can repeatedly call the `IClassFactory::CreateInstance`() method without necessitating an additional round trip to the object to retrieve the class object interface. This can be a significant saving, particularly in a distributed environment.

The other parameter that `CoGetClassObject`() supports (`COSERVERINFO`) allows you to specify the identity of a remote machine on which you would like the class instantiated. It also allows you to make the activation request using a nominated security principal - normally COM will perform the activation using the identity of the client process.

Using CoGetClassObject

QA-IQ

```
CoInitialize( 0 );  // initialise COM runtime

IClassFactory *pIFactory = 0;
HRESULT hr = CoGetClassObject( CLSID_Account,
                               CLSCTX_INPROC_SERVER,
                               NULL,
                               IID_IClassFactory,
                               (void**)&pIFactory );
if( SUCCEEDED( hr ) ) {
  // create the COM object
  IAccount *pIAccount = 0;
  hr = pIFactory->CreateInstance( NULL, // no aggregation
                                  IID_IAccount,
                                  (void**)&pIAccount );
  pIFactory->Release();  // finished with class factory
  if( SUCCEEDED( hr ) ) {
     // Do something with our IAccount interface pointer
     ...
     pIAccount->Release();
  }
}
CoUninitialize();
```

49

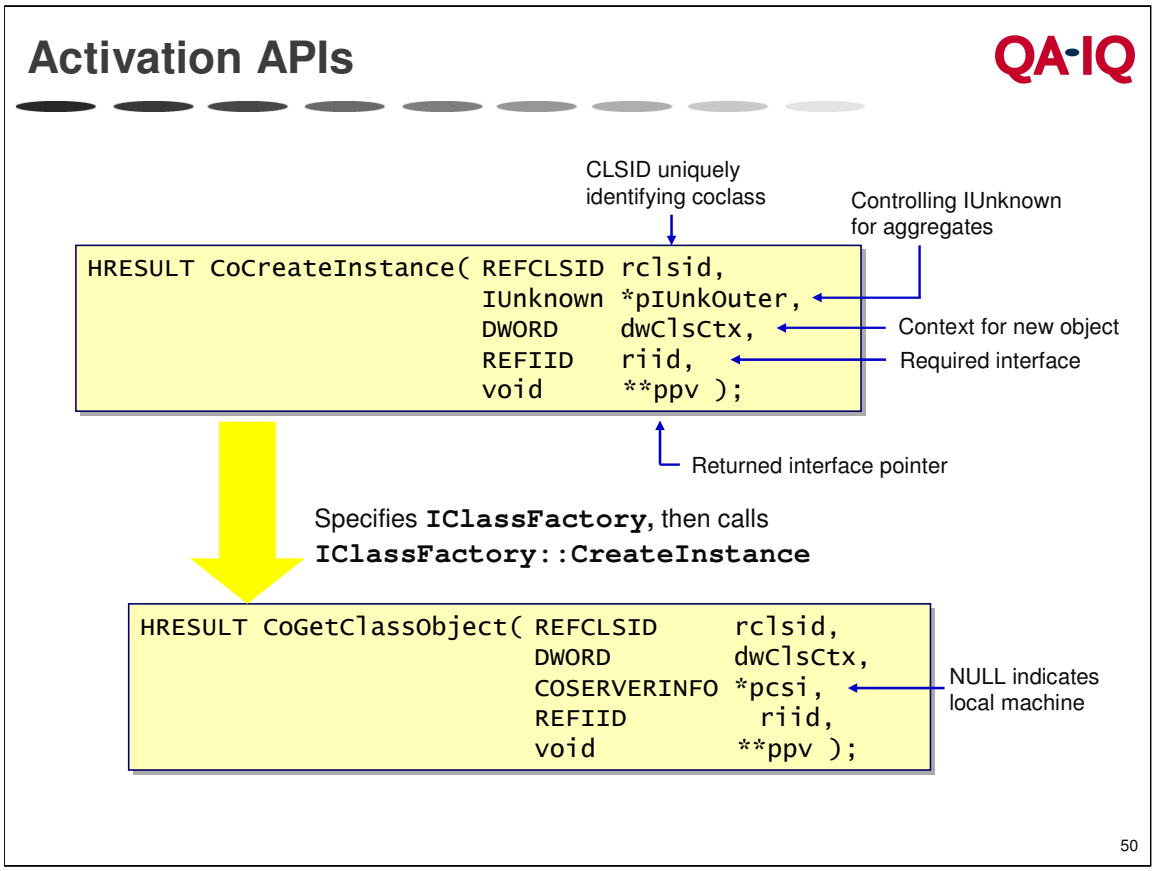
First, a client must initialise the COM library by calling CoInitialize(). This function takes a single parameter, which is no longer used with 32-bit Windows, so it must be NULL.

Next, we call CoGetClassObject() to get a pointer to the IClassFactory interface of the class factory of our Account class. CoGetClassObject() takes five parameters: the CLSID of the desired COM object, the execution context (inproc, local or remote), a COSERVERINFO pointer (only used with DCOM), the requested interface, and an address for the returned pointer.

Then, if CoGetClassObject() succeeds, we call the CreateInstance() function of the IClassFactory interface of the Account class factory. As previously discussed, CreateInstance() creates an instance of a COM object with a specific CLSID (i.e. the CLSID passed to CoGetClassObject()), queries it for the requested interface in riid (I.e. IAccount), and returns the interface pointer in pIAccount. At this point, we've finished with the class factory, so we release its IClassFactory interface.

Finally, if CreateInstance() succeeds, we can use the returned pointer to call functions of the COM object's IAccount interface, such as Deposit() and Withdraw().

Finally, each call to CoInitialize() must be balanced with a call to CoUninitialize(). Also, if the COM object supports any OLE functions such as compound documents, drag-and-drop, in-place activation, etc, OleInitialize() and OleUninitialize() should be used instead of CoInitialize() and CoUninitialize().



The activation API most often used (particularly by programmers relatively new to COM) is `CoCreateInstance`(). This API creates an uninitialised instance of a specified COM class and returns an interface pointer of the specified type.

Under the covers, `CoCreateInstance`() will locate the class object using the same techniques as those employed by `CoGetClassObject`(). It will then use the class object to create an instance using the `CreateInstance`() method from the `IClassFactory` interface.

Implementing a COM Client

QA-IQ

```
CoInitialize( 0 );  // initialise COM runtime

// Create Object
IUnknown *pIUnk = 0;
HRESULT hr = CoCreateInstance( CLSID_Account,
                               NULL,
                               CLSCTX_INPROC_SERVER,
                               IID_IUnknown,
                               (void**)&pIUnk );
if( SUCCEEDED( hr ) ) {
  // query for required interface
  IAccount *pIAccount = 0;
  hr = pIUnk->QueryInterface( IID_IAccount,
                              (void**)&pIAccount );

  pIUnk->Release();  // finished with IUnknown
  if( SUCCEEDED( hr ) ) {
    // Do something with our IAccount pointer
    ...
    pIAccount->Release();
  }
}
CoUninitialize();
```

51

The revised client code shows the use of `CoCreateInstance`() to create the named object. `CoCreateInstance`() is simpler to use than `CoGetClassObject`(), as it both locates the class object and makes an instance of the object in one call.

So when should you use `CoCreateInstance`() as opposed to `CoGetClassObject`()?

Most COM developers will use `CoCreateInstance`(), or it's extended version `CoCreateInstanceEx`() to make objects, because the call is easy to use, and protects the developer from having to manage the class object reference. `CoGetClassObject`() should be used when the object exposes a custom class creation interface, or when the client wants to create multiple objects of the same type.

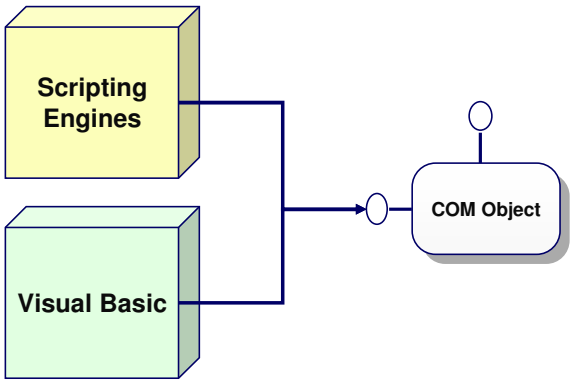
The other question that needs addressing is "Why do we ask for `IUnknown` during the call to `CoCreateInstance()`?".

All COM objects must implement `IUnknown`, so we can guarantee that if a COM object can be made, we will obtain an interface pointer to it. From here we can then query the object for the desired interface. This can be useful when we are writing a client that supports multiple versions of a COM object. Using `QueryInterface()` to find out which version is installed is much quicker than attempting to create the object many times.

Other Clients

QA-IQ

- COM is about component development and reuse
- Consequently, COM must support non C++ clients
  - e.g. Visual Basic, VB Script and JavaScript
- These environments provide excellent support for COM

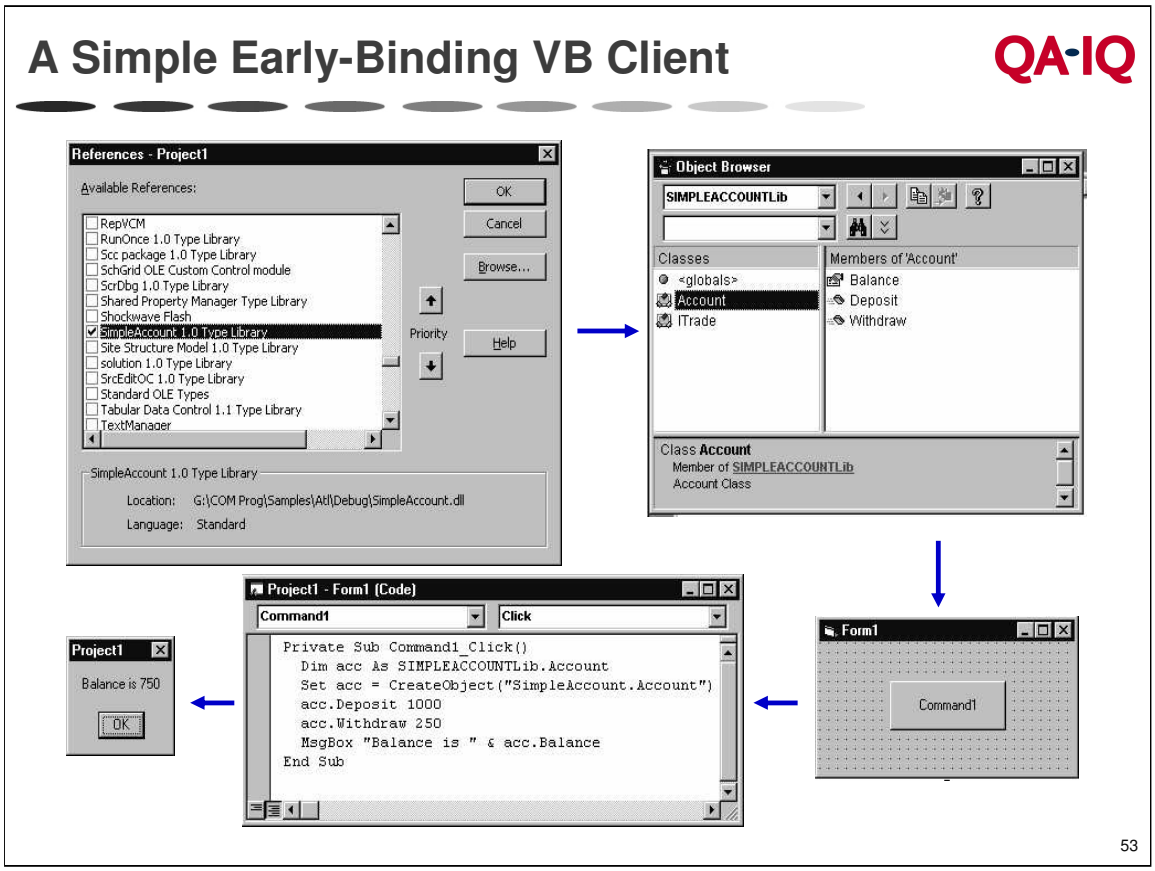


52

COM programming is not just about C++. Many different programming environments support COM, with one of the very best being Visual Basic.

Each of these handles COM differently, as we will see when we start to look at language integration later in the course.

The next page shows us one way of calling COM objects from Visual Basic.



If you're not familiar with Visual Basic:
Start Visual Basic, Select Standard EXE project from New Project dialog, then click Open, Select Project->References..., check the SimpleAccount 1.0 Type Library, and click OK

Select View->ObjectBrowser and then select SIMPLEACCOUNTLib type library from the Project/Library drop-down list box. Note that the Account class and the ITrade interface are indicated, but the IAccount interface is not. This is because the IAccount interface was marked as default in the coclass description of the Account coclass in the IDL, and is therefore hidden by Visual Basic.

Drop a CommandButton on Form 1

Select View->Code

Select Command1 from the Object drop-down list

Enter the following code:

```
Dim acc As SIMPLEACCOUNTLib.Account
Set acc = CreateObject("SimpleAccount.Account")
acc.Deposit 1000
acc.Withdraw 250
MsgBox "Balance is " & acc.Balance
```

Select Run->Start

Test the component by clicking the Command1 button

We'll talk more about early binding (as opposed to late binding) and type libraries later in the course.

Summary

QA-IQ

- IUnknown, the COM base interface, defines 3 functions
  - QueryInterface(), AddRef() and Release()
- QueryInterface() allows a client to request a pointer to a specified interface
- AddRef() and Release() to control objects lifetime
  - AddRef() after replicating a interface pointer
  - Release() when it is no longer needed
- Every COM interface is identified by an IID
  - A 128-bit GUID unique over space and time
- All interface functions, should return an HRESULT
  - 32-bit value that contains success or error code
  - Use CoCreateInstance() to create the object

54