

## Chapter Overview



- **Objectives**
  - Use wrapper classes to simplify C++ COM clients
  - Access COM objects from Visual Basic, VBScript and Java
- **Chapter content**
  - Type libraries
  - Early binding versus late binding
  - Simple VB clients
  - Importing a type library into a C++ client
  - Using smart pointers
  - Error handling
  - Supporting rich error information
- **Practical content**
  - Generating and handling errors
- **Summary**

152

## Using Type Libraries



- **Type libraries are the key to integrating COM servers and clients written in different languages**
  - Binary tokenised form of IDL
  - Normally embedded as a resource in COM server DLL
- **Many types of clients can use them**
  - Visual Basic, Visual C++, Delphi, etc.
- **Others absolutely depend on them**
  - Visual J++, JavaScript and VBScript

153

---

One of the key elements for language integration is the type library.

As we saw in the chapter on IDL, the type library is prepared from IDL files using the MIDL compiler. Whether the client is Visual C++, Java, Visual Basic or a scripting client, the type library is fundamental to *optimal* (in some cases, mandatory) integration.

## Visual Basic Clients

QA-IQ

- **If type library is provided:**
  - Client can use any described interface - either through vtable or via IDispatch
  - Client can use either late or early binding
- **If type library is not provided:**
  - Client can use only late binding - via IDispatch::GetTypeInfo() and IDispatch::GetIDsOfNames()

154

---

If you're using Visual Basic 5.0 or later to develop a COM client, you can use any interface described in a type library irrespective of whether it's a dispinterface, dual interface or a vtable interface. Normally, you would use early binding for best performance, but you could choose to use late binding.

If a type library is not provided, then your only option is to use late binding to a dispinterface or dual interface.

## Early Binding vs. Late Binding

- **Early binding**

- **Visual Basic IDE needs to consult type library at compile time**

- Use Project->References dialog to make type library available
    - Declare object variable of specific class
    - Faster execution than late binding

```
Dim acc As SIMPLEACCOUNTLib.Account
Set acc = CreateObject("SimpleAccount.Account")
```

- **Late binding**

- **Compiled code gets type information at run time via IDispatch**

- Component must expose an IDispatch (or dual) interface
    - Declare variable of type Object
    - Slower execution than early binding and no type checking

```
Dim acc As Object
Set acc = CreateObject("SimpleAccount.Account")
```

155

Note that the Visual Basic `Object` type is the equivalent of a pointer to the `IDispatch` interface of a COM object. Also a `Set` statement calls `QueryInterface()` to request the correct interface reference.

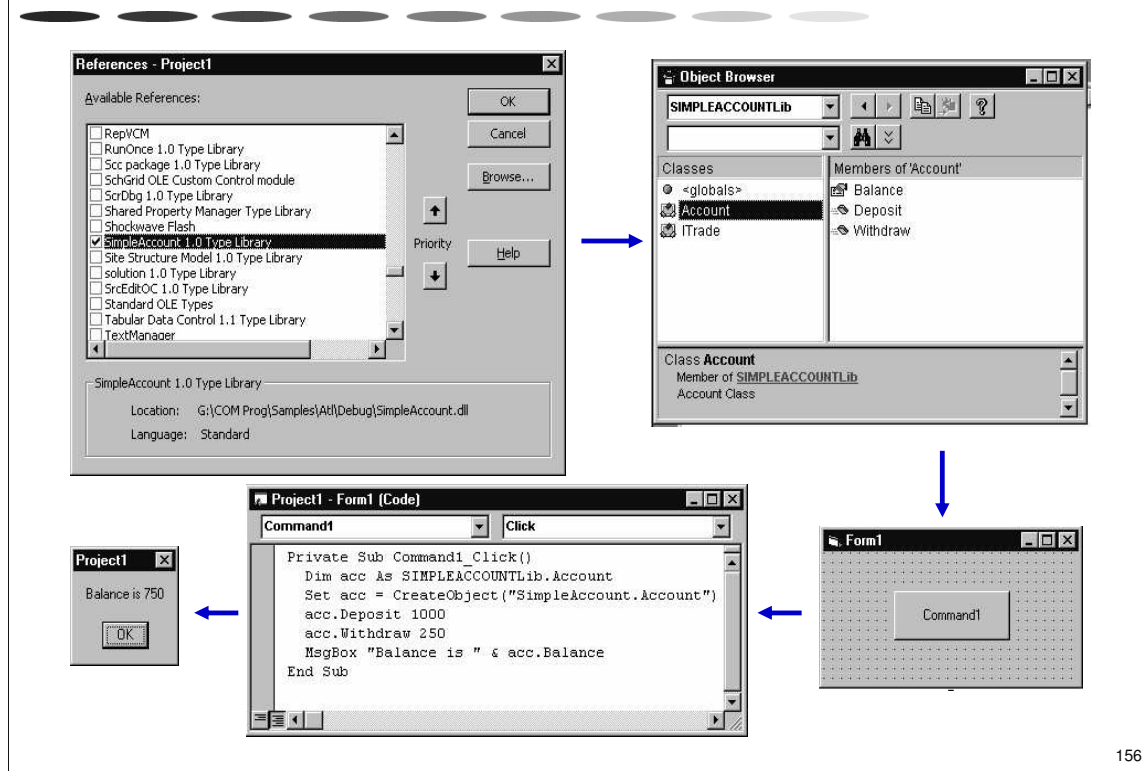
Early binding occurs when full type information is provided to the Visual Basic compiler. Full compile time checking is enabled and IntelliSense is available to the developer. This option also maximises the performance at runtime, as direct vtable calls are used.

Late binding occurs when a type library is not provided to the Visual Basic compiler. In this case, no edit time and compile time checking is available, IntelliSense is not available and the runtime performance is affected as `IDispatch` has to be used to make the calls.

Notice that both cases (early and late bound) use the same object creation function, `CreateObject()`.

## A Simple Early Binding VB Client

QA-IQ



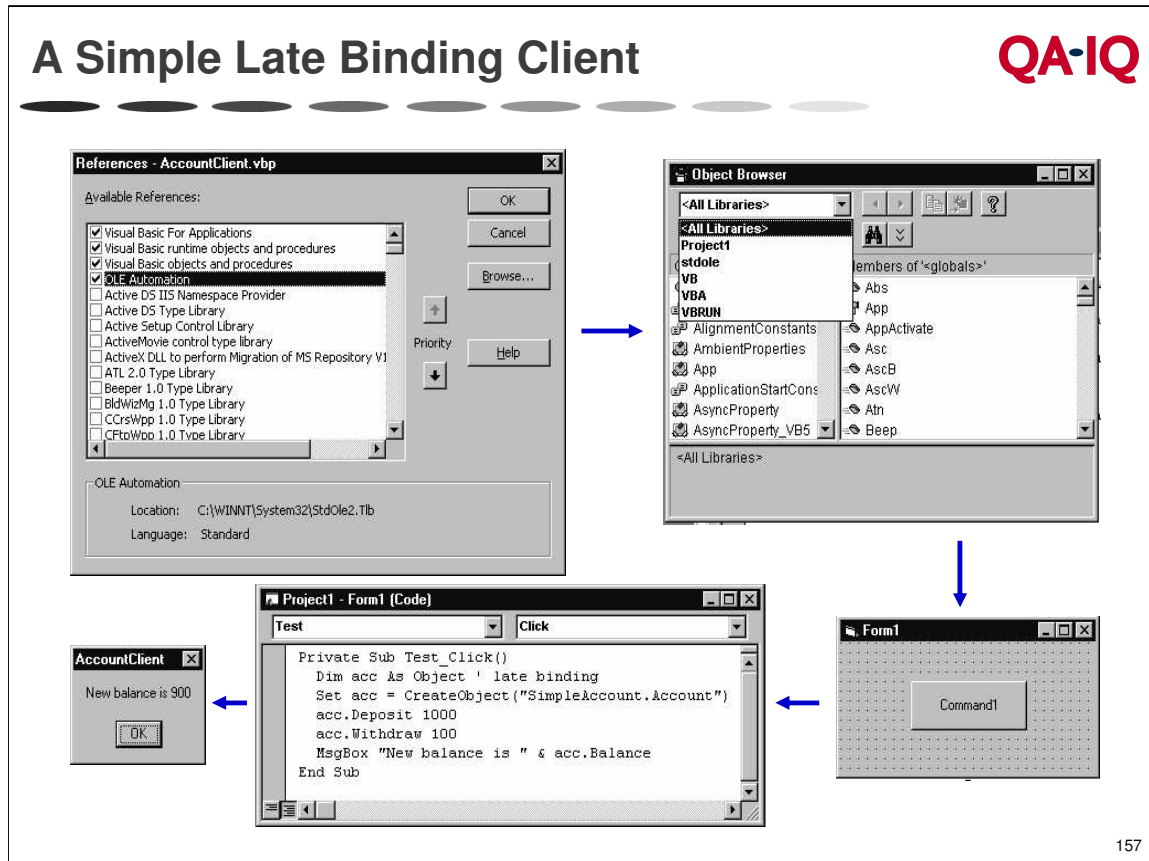
156

Remember that with early binding, the VB development environment is able to fully understand all of the COM objects as described in the type library. Consequently, you gain IntelliSense support, syntax checking and compile time code verification.

Information about your COM object is also available to the VB object browser.

## A Simple Late Binding Client

QA-IQ



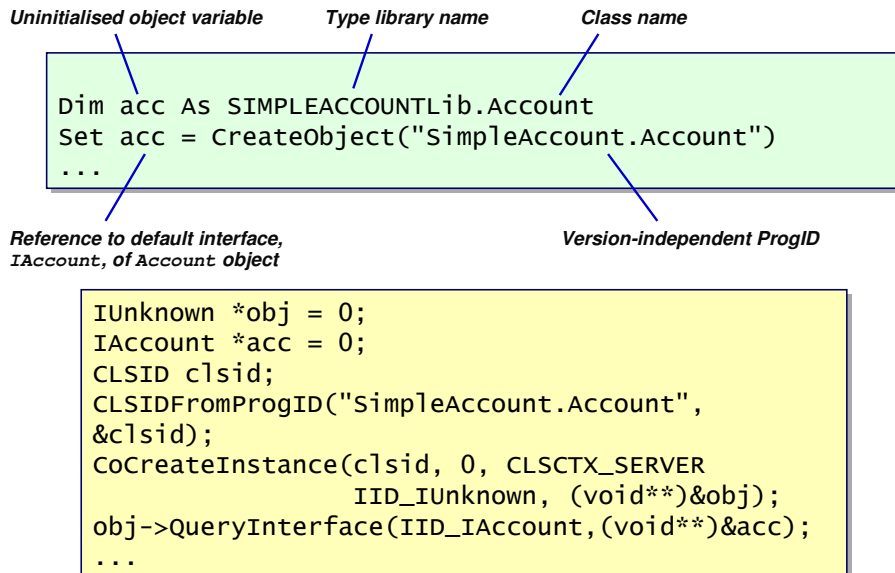
157

Late binding occurs when you do not reference the type library from within Visual Basic.

In this case, you will only be able to communicate with the COM object through its IDispatch interface, and no information is available in the VB object browser.

## A Closer Look at CreateObject()

- This Visual Basic code is equivalent to the following C++ code



158

If a type library is available, you can also use the `New` keyword to create a new object and assign an object reference. For example, the following statement is equivalent to the Visual Basic code shown on the slide:

```
Dim acc = SIMPLEACCOUNTLib.Account
Set acc = New SIMPLEACCOUNTLib.Account
```

The only obvious difference is that the `New` keyword uses the CLSID obtained from the type library, whereas `CreateObject()` uses `CLSIDFromProgID()` to obtain the CLSID from the Registry. The advantages of `CreateObject()` are that it can support multiple versions (using a `VersionIndependentProgID`) and that it allows you to specify a remote machine name. Using the `New` keyword is more secure, and avoids name conflicts in the registry.

**Warning:** If you develop COM objects in VB itself, you must be aware that when the COM object code is in the same project as the client VB code, the use of the `New` keyword will result in VB bypassing the COM runtime.

There is also another way to declare a COM object in Visual Basic.

```
Dim acc as New SIMPLEACCOUNTLib.Account
```

However, with this use of the `New` keyword, the VB runtime doesn't actually create an `Account` object, and assign the returned object reference to the `acc` variable, until the first time a method or property is accessed through `acc`. This can impact performance, because every time the VB runtime encounters such a variable, it must test whether or not an object reference has already been assigned to that variable. Therefore, this method of declaring variables is not recommended.



## Accessing Another Interface

- **Declare variable of interface type and set it to point to object**

```
Dim acc AS SIMPLEACCOUNTLib.Account
Set acc = CreateObject("SimpleAccount.Account")
acc.deposit 1000
...
...
Dim itrade AS SIMPLEACCOUNTLib.ITrade
Set itrade = acc
...
```

*Creates object of type Account  
Returns reference to default  
interface, IAccount*

*Declares variable of type ITrade*

*Obtains reference to ITrade  
interface  
(calls acc.QueryInterface (...))*

- **Call methods of second interface through new variable**

```
MsgBox "Buy price is " & itrade.BuyPrice
itrade.Buy 5
...
MsgBox "New balance is " & acc.Balance
Set itrade = Nothing
...
```

*Releases reference to ITrade  
interface*

159

Using a Set statement to assign an interface reference to an object reference is the C++ equivalent of calling `QueryInterface()`.

## A Simple VBScript Client

- **Components that support IDispatch can also be called by scripting-language clients**
  - Specify CLSID in OBJECT tag of HTML document
  - Can also produce a script for Windows Scripting Host (WSH)

```
<OBJECT ID="acc"
  CLASSID="CLSID:81B8FDDD-FB37-11D2-9C81-000000000000">
</OBJECT>
<SCRIPT LANGUAGE="VBScript">
  acc.Deposit 1000
  acc.Withdraw 50
  Document.Write "Current balance is " & acc.Balance
</SCRIPT>
```

AccountClient.html

```
Dim acc
Set acc = CreateObject("SimpleAccount.Account")
acc.Deposit 1000
acc.Withdraw 50
MsgBox "New balance is " & acc.Balance
```

AccountClient.vbs

160

VBScript is supported by Internet Explorer (IE), Internet Information Server (IIS), Visual Studio and Windows Scripting Host (WSH). WSH is supplied with Windows 98, Windows 2000 and Windows XP.

For Windows 95 and Windows NT 4.0, it can be downloaded from <http://msdn.microsoft.com/scripting>. It is also a component of Windows NT 4.0 Option Pack.

The above WSH script can be run from the command line by typing:

```
>cscript AccountClient.vbs
```

Alternatively, you can simply double-click on the file in Windows Explorer.

Notice the variable declaration in VB Script. All variables in the scripting environment are `VARIANTs`, which means that the variable can hold automation compatible data types including `IUnknown` and `IDispatch` pointers, but no custom interfaces.

## Simplifying a Visual C++ Client



- **Use `#import` directive to "import" type library from .tlb, .dll, .ocx or .exe file**
- **Compiler generates two "header" files that contain**
  - **A wrapper for each interface function ...**
    - Allows property methods to be accessed as member variables
    - Directly return [retval] parameters
    - Throw `com_error` exceptions instead of returning `HRESULTS`
  - **... plus a number of wrapper classes**
    - Allow `_variant_t` and `_bstr_t` types to be used instead of `VARIANTs` and `BSTRs`
  - **... plus a smart pointer for each interface pointer**
    - Based on `_com_ptr_t<>` template class
    - Automatically make calls to `CoCreateInstance()`
    - Responsible for all of `IUnknown` functionality

161

If you're using Visual C++ 5.0 or later, you can greatly simplify COM client code by using the `#import` directive to "import" a specified type library. The type library can be contained in any file that can be read with a call to `LoadTypeLib()`, which means it can be a .tlb, .dll, .ocx, or a .exe file.

When you compile a file that contains a `#import` directive, the compiler basically converts the type library contents into C++ source code. In fact, the compiler generates two "header" files: a primary header file that has the same base name as the type-library file with a .tlh extension, and a secondary header file that is similar except that it has a .tli extension. Both files are written to the output directory (e.g. debug for debug builds). The secondary header file is `#include'd` in the primary header file, and the primary header file is automatically included in the compilation as if were `#include'd`.

Together the header files contain a number of wrapper functions and classes, as stated on the slide above. The primary header file also contains a smart pointer for each interface pointer. These smart pointers are specialisations of the `_com_ptr_t<>` template class. They can be used just like normal pointers, but they also take responsibility for making calls to `CoCreateInstance()` and for implementing `QueryInterface()`, `AddRef()` and `Release()`.

In short, they make client-side COM programming both easier and safer, but you should be aware that they will generate client-side exceptions when method calls fail.

## Primary Header File (1)



*Contains definitions for wrapper classes*

*Associates a GUID with IAccount, which can be retrieved using \_\_uuidof()*

```
#include <comdef.h>

namespace SIMPLEACCOUNTLib {

// Forward references and typedefs
struct /* coclass */ Account;
struct __declspec(uuid("81b8fddc-fb37-11d2-9c81-000000000000"))
/* dual interface */ IAccount;

// Smart pointer typedef declarations
_COM_SMARTPTR_TYPEDEF(IAccount, __uuidof(IAccount));

// continued on next slide
...
}
```

SimpleAccount.tlh

*Expanded by compiler to:*

```
typedef _com_ptr_t<_com_IID<IAccount, __uuidof(IAccount)> > IAccountPtr;
which defines a new smart-pointer class called IAccountPtr
```

162

The above slide shows the first part of the primary header file that was generated when we put a `#import "..\Debug\SimpleAccount .dll"` into our client source code (see slide on page 14). Note the start of a namespace called `SIMPLEACCOUNTLib`, which is the name of the type library as specified in the IDL file. We can also see the use of `__declspec(uuid())` to assign a GUID to the coclass `Account`.

Also shown is the use of the `_COM_SMARTPTR_TYPEDEF` macro. This is expanded by the compiler to declare a smart-pointer class called `IAccountPtr`.

## Primary Header File (2)

```

...
// Type library items
struct __declspec(uuid("81b8fddd-fb37-11d2-9c81-000000000000"))
Account;
    // [ default ] interface IAccount

struct __declspec(uuid("81b8fddc-fb37-11d2-9c81-000000000000"))
IAccount : IDispatch {
    // Property data
    __declspec(property(get=GetBalance)) float Balance;

    // wrapper methods for error handling
    HRESULT Deposit ( float amount );
    HRESULT Withdraw ( float amount );
    float GetBalance ( );

    // Raw methods provided by interface
    virtual HRESULT __stdcall raw_Deposit ( float amount ) = 0;
    virtual HRESULT __stdcall raw_Withdraw ( float amount ) = 0;
    virtual HRESULT __stdcall get_Balance ( float * pval ) = 0;
};


// wrapper method implementations
#include "...\\debug\\SimpleAccount.tli"
} // namespace SIMPLEACCOUNTLib

```

*SimpleAccount.tlh*

*Allows Balance to be treated like a member variable*

*Secondary header file*



163

The above slide shows the second part of the primary header file. Notice that the declaration for `IAccount` includes a property, wrapper methods for error handling, and raw methods. The `__declspec(property(...))` declaration allows the client code to treat `Balance` as though it were a member variable of the `Account` class (i.e. as you would in Visual Basic). The wrapper methods, which are defined in the secondary header file `SimpleAccount.tli` (see below), include code to throw an `_com_error` exception if an `HRESULT` returned by the corresponding raw method indicates an error. Note that the raw methods are the only pure virtual functions and are therefore the only methods in the interface `IAccount`. The fact that they have been renamed by the compiler is irrelevant, because this renaming has not changed the layout of the vtable, which defines the interface.

The contents of the secondary header file, `SimpleAccount.tli`, is as follows:

```

// interface IAccount wrapper method implementations
inline HRESULT IAccount::Deposit(float amount) {
    HRESULT _hr = raw_Deposit(amount);
    if (FAILED(_hr)) _com_issue_errorex(hr, this, __uuidof(this));
    return _hr;
}

inline HRESULT IAccount::Withdraw(float amount) {
    HRESULT _hr = raw_Withdraw(amount);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
    return _hr;
}

inline float IAccount::GetBalance() {
    float _result;
    HRESULT _hr = get_Balance(&_result);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
    return _result;
}

```

## Using #import and Smart Pointers



```
#import "..\Debug\SimpleAccount.dll"
using namespace SIMPLEACCOUNTLib;
CoInitialize( 0 );
IAccountPtr spAccount( __uuidof(Account) );
spAccount->Deposit( 10000.00 );
float balance = spAccount->Balance;
_tprintf( _T("Opening balance is %.2f\n"), balance);
ITradePtr spTrade = spAccount;
float buyPrice = spTrade->BuyPrice;
_tprintf( _T("Buy price is %.2f\n"), buyPrice);
spTrade->Buy( 5 );
balance = spAccount->Balance;
_tprintf( _T("Final balance is %.2f\n"), balance);

spAccount = 0;
spTrade = 0;
CoUninitialize();
```

Generated header files are automatically included

Namespace has same name as type library

Creates object and smart pointer

Can treat property like a member variable

Calls `QueryInterface()` for smart pointer to `ITrade`

Automatically calls `Release()`

164

Here is an example of how we could use `#import` and smart pointers to simplify the code of a client of our Account object. This object has two interfaces: `IAccount` and `ITrade`. For clarity, we have deleted the error checking code, but we will look at this topic in some detail towards the end of the chapter.

Notice the use of a `using namespace` directive. This obviates the need to refer to the smart pointers, etc., by their full name, e.g. we can use `IAccountPtr` rather than `SIMPLEACCOUNTLib::IAccountPtr`.

After calling `CoInitialize()` to initialise the COM runtime, we use one of the `_com_ptr_t<>` constructors to create the smart pointer object `spAccount`. This constructor takes the CLSID of the component, which we obtain by using `__uuidof()` on coclass `Account`. Note that this constructor creates an instance of the specified coclass, and then queries it for the appropriate interface. If it fails, it throws an `_com_error` exception.

Then, we call the method `IAccount::Deposit()` through the smart pointer `spAccount` before getting the opening balance. Notice how we can access the `Balance` property as though it were a member variable of the `Account` object.

Now, we're ready to start trading, but to do this, we need a pointer to the `Account` object's `ITrade` interface. This is achieved simply by declaring a new smart pointer of type `ITradePtr` and initialising it with the value of our existing smart pointer. The overloaded assignment operator in the `_com_ptr_t<>` class calls `QueryInterface()` for us. If it fails, it throws a `_com_error` exception.

When we've finished, notice that we set the smart pointers to `NULL` before calling `CoUninitialize()`. In this particular case, we are closing down the COM libraries before the destructors for the two smart pointers. Given that the destructors will therefore call `Release()` after the COM objects' code has been forcibly ejected, we need to explicitly release the objects before `CoUninitialize()`.

## Error Handling



- **All interface methods should return an HRESULT**
  - 32-bit value that contains success or error code
  - Should be tested by C/C++ clients after each call
  - Converted to an exception by Visual Basic and Java clients
  - Can only provide a limited amount of information
- **Automation object can provide "rich error-information"**
  - EXCEPINFO structure returned by IDispatch::Invoke() can include source of error, error description, name of help file, etc
- **How can a vtable method return similar information?**
  - Especially if method is part of a dual interface!
  - Solution is to use an error object

165

---

We are now going to turn our attention to how a COM object can provide "rich error-information" to a client, and how a C++ client can obtain such information. Note that these techniques are applicable to both Automation and non-Automation objects.

## Error Objects



- **An error object must implement two interfaces:**
  - ICreateErrorInfo for setting error information
  - IErrorInfo for getting error information
- **Error information can include:**
  - IID of the interface that generated the error
  - ProgID of error source
  - Error description
  - Name of associated help file, etc.
- **COM object that is capable of generating error objects must implement ISupportErrorInfo interface**

```
interface ISupportErrorInfo : IUnknown
{
    HRESULT InterfaceSupportsErrorInfo([in] REFIID riid);
}
```

Client specifies IID

Object must return either S\_OK or S\_FALSE

166

To return rich error information to a client, a COM object must create an error object to encapsulate such information. An error object must implement two interfaces, ICreateErrorInfo and IErrorInfo, which are defined as follows:

```
interface ICreateErrorInfo : IUnknown {
    HRESULT SetGUID([in] REFUID rguid);
    HRESULT SetSource([in] LPOLESTR szSource);
    HRESULT SetDescription([in] LPOLESTR szDescription);
    HRESULT SetHelpFile([in] LPOLESTR szHelpFile);
    HRESULT SetHelpContext([in] DWORD dwHelpContext);
}

interface IErrorInfo : IUnknown {
    HRESULT GetGUID([out] GUID *pGUID);
    HRESULT GetSource([out] BSTR *pBstrSource);
    HRESULT GetDescription([out] BSTR *pBstrDescription);
    HRESULT GetHelpFile([out] BSTR *pBstrHelpFile);
    HRESULT GetHelpContext([out] DWORD *pdwHelpContext);
}
```

To indicate that it is capable of generating error objects, a COM object must also implement the ISupportErrorInfo interface, which defines a single method as shown on the slide. A client will query a COM object for this interface to see whether it is capable of providing rich error information.

The error object that is provided by COM has a very short lifetime that is managed by the COM plumbing. Calling another method on a COM object can reset it. Consequently, after creating an error object, the COM object should return control to the client as quickly as possible. In addition, as soon as the client receives a failure HRESULT, it should check for the existence of the COM error object immediately.



## Providing Error Information (1)

- **Check Support ISupportErrorInfo box in Attributes tab of ATL Object Wizard**
  - Adds ISupportErrorInfo to the list of base classes
  - Implements the ISupportErrorInfo method

```
class ATL_NO_VTABLE CAccount :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAccount, &CLSID_Account>,
    public ISupportErrorInfo,
    public IAccount
{ ...
```

```
STDMETHODIMP CAccount::InterfaceSupportsErrorInfo(REFIID riid) {
    static const IID* arr[] = {
        &IID_IAccount
    };
    for (int i=0; i < sizeof(arr) / sizeof(arr[0]); i++) {
        if (InlineIsEqualGUID(*arr[i],riid))
            return S_OK;
    }
    return S_FALSE;
}
```

Array of interfaces that support error information

167

With ATL, implementing the `ISupportErrorInfo` interface is as easy as remembering to check the *Support ISupportErrorInfo* box in the *Attributes* tab of the *ATL Object Wizard*. This results in some additions to the generated code as highlighted (in bold) above. If you forget to check the box, you can make the same additions manually.

However, you should bear in mind that you are really making a commitment to providing error information (and to be honest, you really *should* be). Therefore, before returning any failure `HRESULTS`, make sure that you set the error information.

## Providing Error Information (2)

- In object implementation, use one of overloaded `Error()` functions inherited from `CCoClass`

```
STDMETHODIMP CAccount::Withdraw(float amount)
{
    HRESULT hr = S_OK;
    if( m_balance >= amount )
    {
        m_balance -= amount;
    }
    else
    {
        hr = Error( _T("You cannot withdraw this amount\n"),
                    IID_ICAccount,
                    E_FAIL );
    }
    return hr;
}
```

168

To return rich error information to the client, you simply call one of the overloaded `CCoClass::Error()` functions before returning a failure `HRESULT`. `Error()` eventually calls `AtlReportError()`, which in turn calls `CreateErrorInfo()` to create an error object. After setting the specified error information through the `ICreateErrorInfo` interface, `AtlReportError()` calls `SetErrorInfo()` to assign the error object to the current thread of execution.

For more information on the `Error()` functions, see the Visual C++ documentation, `atlcom.h` and `atlbase.h`.

## Obtaining Error Information (1)

QA-IQ

- In Visual C++ client, catch any `_com_error` exceptions
  - `_com_error` class encapsulates `HRESULT` and any associated `IErrorInfo` object

```
#import "..\Debug\SimpleAccount.dll" no_namespace
```

*Must use #import directive*

```
try
{
    IAccountPtr spAccount( __uuidof(Account) );
    spAccount->Deposit( 100.00 );
    spAccount->Withdraw( 5000.00 );
    float balance = spAccount->Balance;
    _tprintf( _T("Final balance is %.2f\n"), balance );
}
catch( _com_error e )
{
    _tprintf( _T("Error in %s:\n %s\n"),
              (LPCTSTR) e.Source(),
              (LPCTSTR) e.Description() );
}
```

*Must use smart pointer*

*May throw \_com\_error exception*

*Calls GetSource() and GetDescription() methods of associated IErrorInfo object*

169

On the client side, we've already seen how the wrapper methods, which were generated by the compiler as a result the `#import` statement, throw an `_com_error` exception if an `HRESULT` indicates an error. For example, here is the wrapper method for the `IAccount::Withdraw()` method as defined in `SimpleAccount.tli`:

```
inline HRESULT IAccount::Withdraw(float amount) {
    HRESULT _hr = raw_Withdraw(amount);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this,
    __uuidof(this));
    return _hr;
}
```

The `_com_error` class encapsulates the `HRESULT` error code and any associated `IErrorInfo` object. So, to obtain the error information, we can simply catch `_com_error` exceptions, and then call appropriate methods of the `_com_error` object, as shown on the slide.

Note also that we create the smart pointer within the try block, because the constructor of the smart pointer template class `_com_ptr_t<>` can also throw an `_com_error` exception.

## Obtaining Error Information (2)

- In a Visual Basic client, provide an error handler

```
Private Sub Test_Click()  
    On Error GoTo Errhandler  
    Dim acc As SIMPLEACCOUNTLib.Account  
    Set acc = CreateObject("SimpleAccount.Account")  
    acc.Deposit 1000  
    acc.Withdraw 10000 ' should cause an error  
    MsgBox "New balance is " & acc.Balance  
    Exit Sub  
  
Errhandler:  
    Dim str As String  
    str = "Error in " + Err.Source + vbCrLf  
    str = str + Err.Description + vbCrLf  
    MsgBox str  
End Sub
```



170

In Visual Basic, you can obtain the rich error information simply by providing an error handler. All the implementation details are hidden from you.

## Summary



- **Visual Basic clients can use both early and late binding**
- **Scripting-language clients can use only late binding**
- **Visual C++ clients can be simplified by using #import and smart pointers**
- **Error objects allow a non-Automation server to return rich error information to a client**

171

