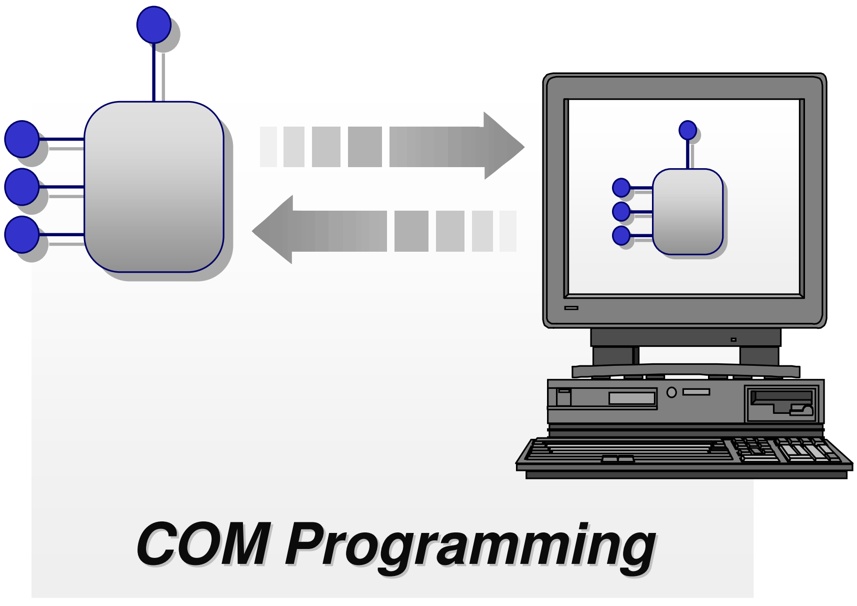
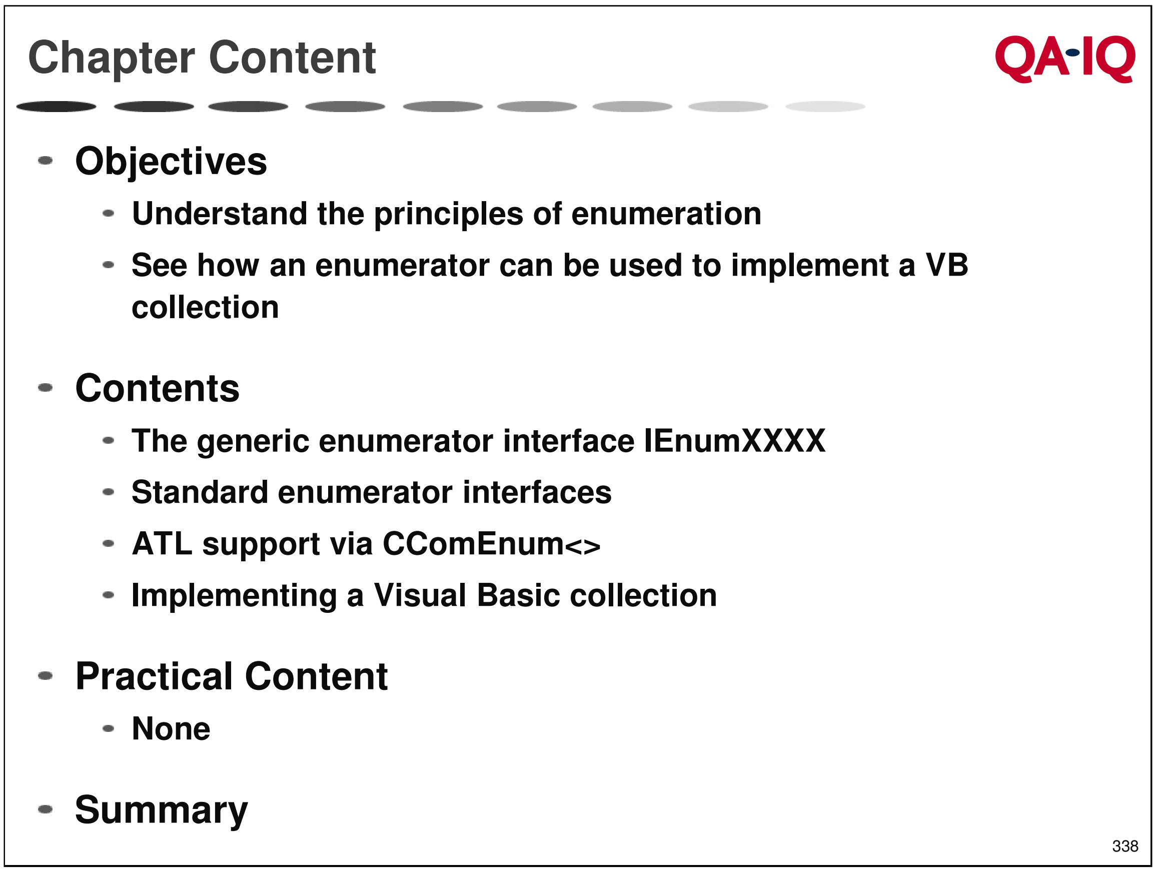
Collections and Enumerators



337



Chapter Content

QA-IQ

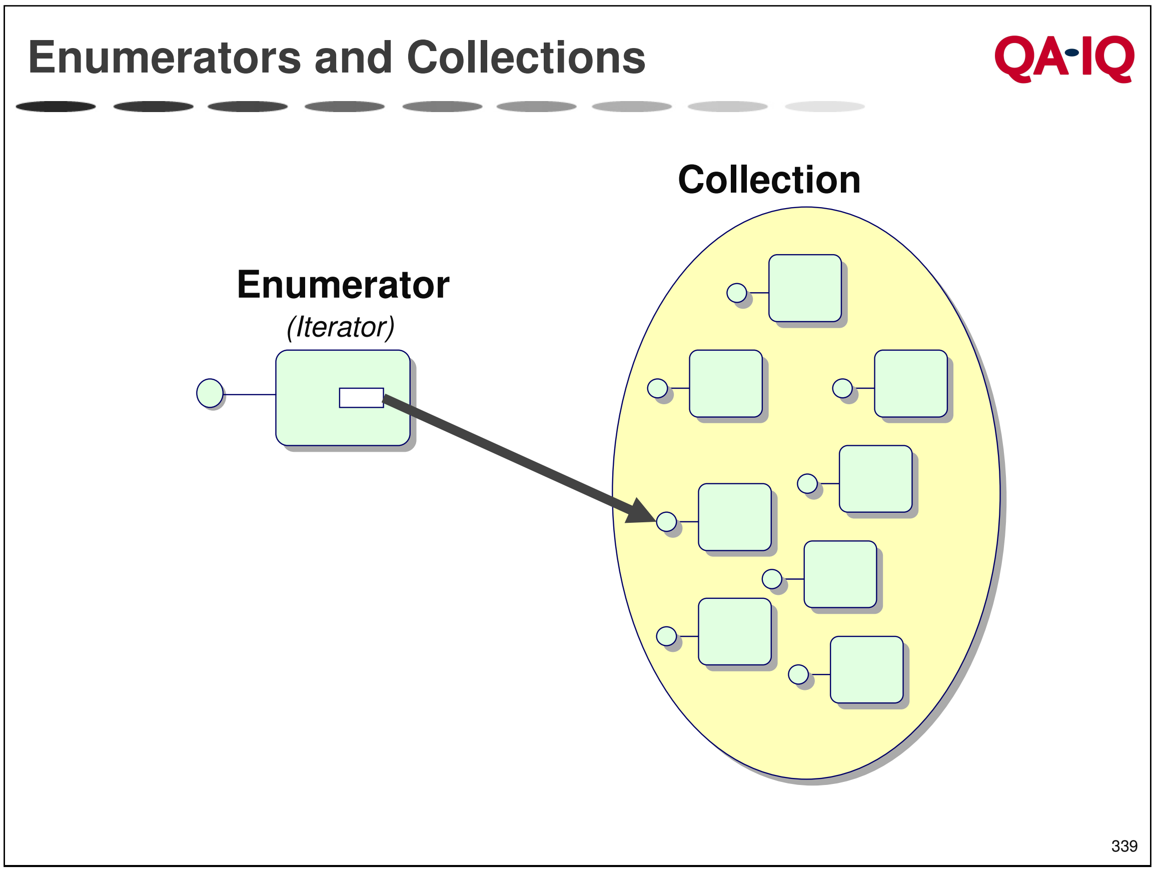
- **Objectives**
  - **Understand the principles of enumeration**
  - **See how an enumerator can be used to implement a VB collection**
- **Contents**
  - **The generic enumerator interface IEnumXXXX**
  - **Standard enumerator interfaces**
  - **ATL support via CComEnum<>**
  - **Implementing a Visual Basic collection**
- **Practical Content**
  - **None**
- **Summary**

338

In this chapter we investigate the way enumerators are used in COM and their relation to collections. Enumerators are generic and may be defined for an infinite set of types. This makes it difficult for the COM specification to tie down enumerators. In C++ we would describe enumerators using templates, but because IDL does not support templates COM uses less precise syntax. The enumerator is defined as a 'meta' interface IEnumXXXX. The XXXX has to be treated like a template argument.

The chapter shows how to implement an enumerator class with the help of the ATL CComEnum template class.

Enumerators are mainly used by Visual Basic applications and the chapter concludes by showing how to use such an enumerator.



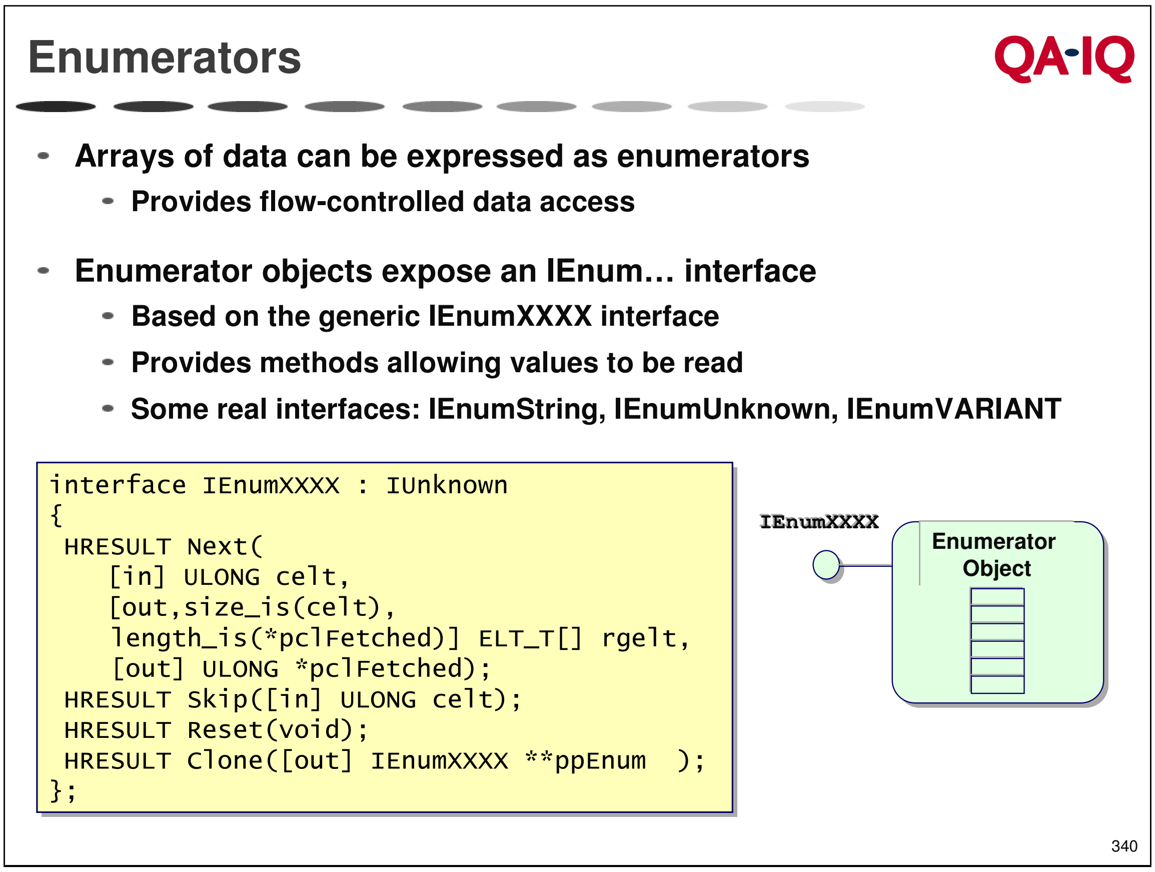
Enumerators and Collections

Before we discuss enumerators in detail it is important to emphasise the difference between enumerators and collections.

A collection is itself an object (it can be a COM object or just a C++ object) that contains other objects or pointers to objects. Collections are very useful at keeping track of objects at run time, particularly if you create lots of objects at run time.

An enumerator is an object that can *iterate* or step through a collection of objects, retrieving each object in turn. In fact, enumerators are often called iterators. Since collections can be implemented in a wide variety of ways (array, linked list, binary tree ...), the enumerator will need inside knowledge on how to iterate through the collection. Therefore enumerators are always tightly coupled to the collection they iterate.

The enumerator will always be pointing to an element in the collection. This is defined as the current position of the iterator.

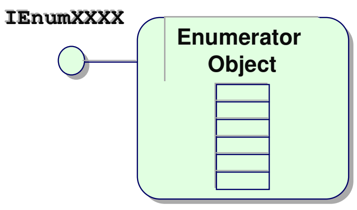


Enumerators

QA-IQ

- Arrays of data can be expressed as enumerators
  - Provides flow-controlled data access
- Enumerator objects expose an IEnum… interface
  - Based on the generic IEnumXXXX interface
  - Provides methods allowing values to be read
  - Some real interfaces: IEnumString, IEnumUnknown, IEnumVARIANT

```
interface IEnumXXXX : IUnknown
{
 HRESULT Next(
    [in] ULONG celt,
    [out,size_is(celt),
    length_is(*pclFetched)] ELT_T[] rgelt,
    [out] ULONG *pclFetched);
 HRESULT Skip([in] ULONG celt);
 HRESULT Reset(void);
 HRESULT Clone([out] IEnumXXXX **ppEnum  );
};
```

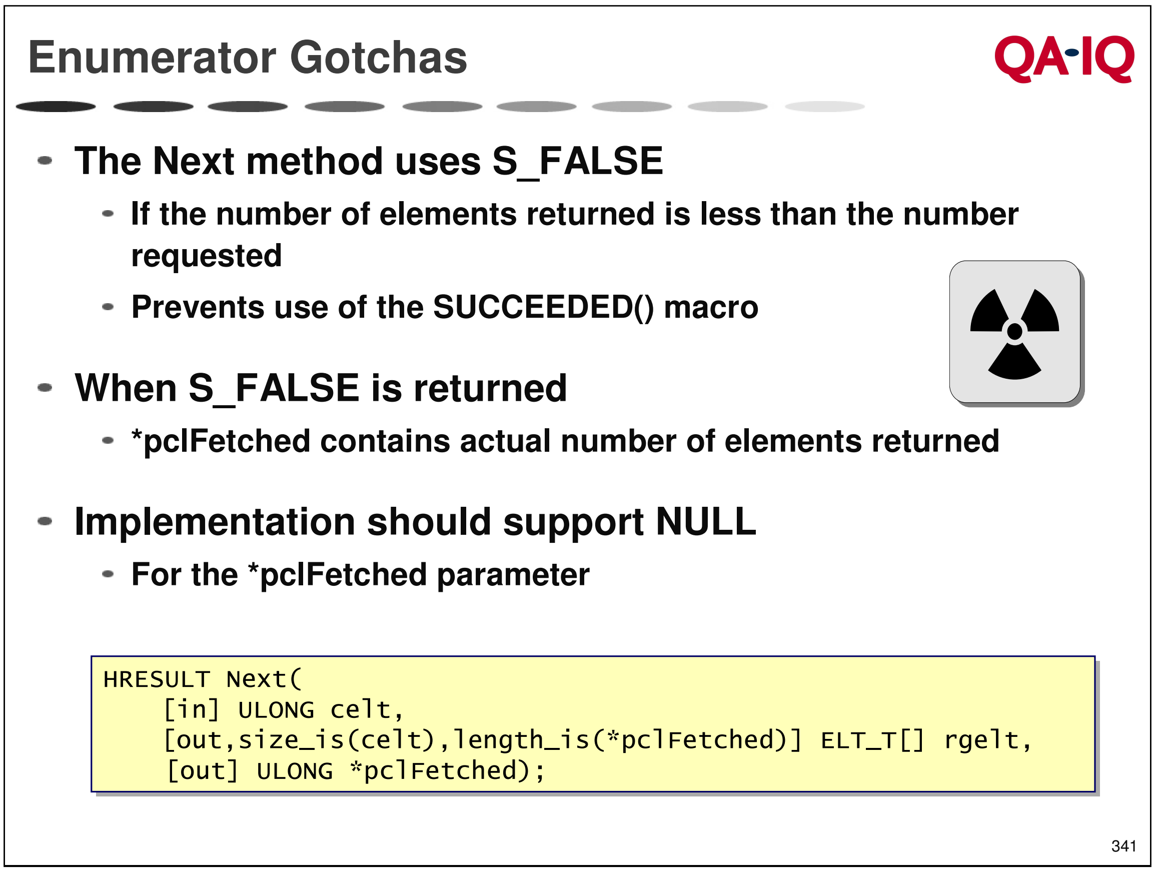


340

To allow you to enumerate the number of items of a given type that an object maintains, OLE provides a set of enumeration interfaces, one for each type of item.

To use these interfaces, the client asks an object that maintains a collection of items to create an enumerator object. The interface on the enumeration object is one of the enumeration interfaces, all of which have a name of the form **IEnum***Item_name*. The only difference between enumeration interfaces is what they enumerate — there must be a separate enumeration interface for each type of item enumerated. All have the same set of methods, and are used in the same way. For example, by repeatedly calling the **Next** method, the client gets successive pointers to each item in the collection.

COM defines a number of IEnum*XXXX* interfaces, such as IEnumString and IEnumUnknown. You can find more of these by looking the MSDN Library.



Enumerator Gotchas

QA-IQ

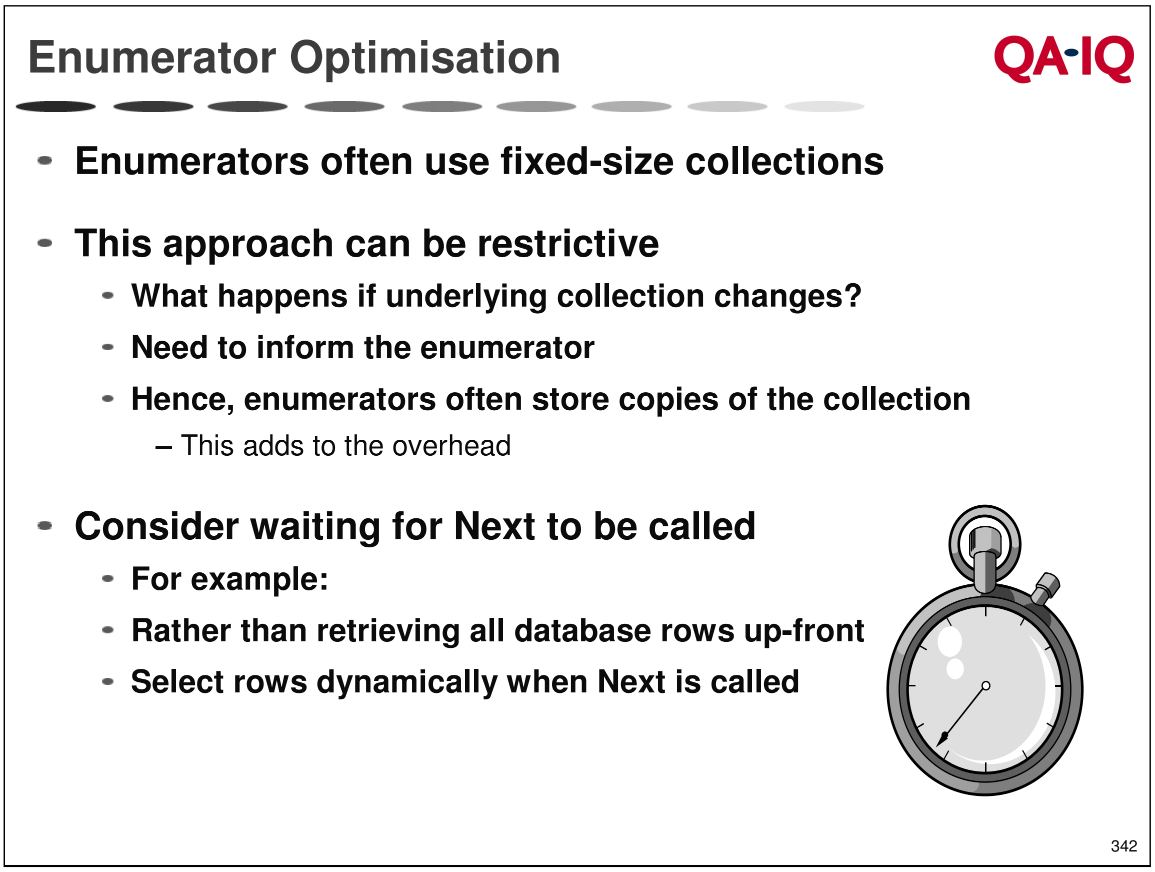
- **The Next method uses S_FALSE**
  - **If the number of elements returned is less than the number requested**
  - **Prevents use of the SUCCEEDED() macro**
- **When S_FALSE is returned**
  - ***pclFetched contains actual number of elements returned**
- **Implementation should support NULL**
  - **For the *pclFetched parameter**

```
HRESULT Next(
    [in] ULONG celt,
    [out,size_is(celt),length_is(*pclFetched)] ELT_T[] rgelt,
    [out] ULONG *pclFetched);
```

341

The **IEnum*XXXX*::Next** method retrieves a specified number of items in the enumeration sequence. The number of items retrieved can be greater than 1. This allows the enumerator to perform block copies.

Note that if you ask for too many objects then the call will only partially succeed and the HRESULT of S_FALSE will be returned and *pclFetched contains actual number of elements returned. Most developers consider this a failure mode, but the S_FALSE return code precludes use of the SUCCEEDED macro.



Enumerator Optimisation

QA-IQ

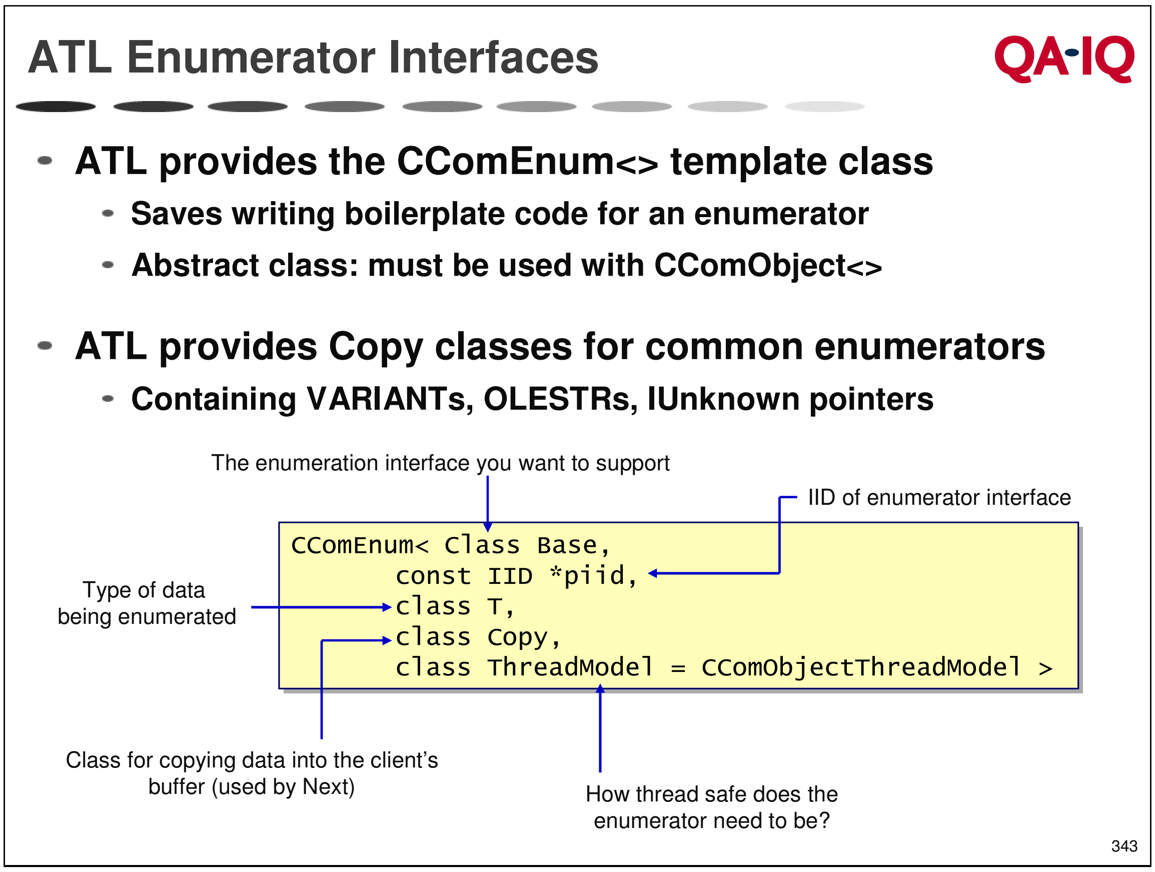
- **Enumerators often use fixed-size collections**
- **This approach can be restrictive**
  - **What happens if underlying collection changes?**
  - **Need to inform the enumerator**
  - **Hence, enumerators often store copies of the collection**
    - This adds to the overhead
- **Consider waiting for Next to be called**
  - **For example:**
  - **Rather than retrieving all database rows up-front**
  - **Select rows dynamically when Next is called**

342

Applications will often retrieve multiple elements from a collection in a single call to **Next**. The COM specification states that the enumerator must return the requested elements in an array. If the associated collection does not store its elements in an array (the normal situation) then the iterator is forced to perform additional copying of elements. This makes enumerators inefficient in practice.

Sometimes the enumerator can optimise the retrieval of elements from the associated collection and this can only help speed up the enumeration process.

Many developers prefer not to use enumerators because of these defects. However, enumerators are use extensively by Visual Basic clients and this is the real area where enumerators shine.



ATL Enumerator Interfaces

QA-IQ

- ATL provides the CComEnum<> template class
  - Saves writing boilerplate code for an enumerator
  - Abstract class: must be used with CComObject<>
- ATL provides Copy classes for common enumerators
  - Containing VARIANTs, OLESTRs, IUnknown pointers

```
CComEnum< class Base,
       const IID *piid,
       class T,
       class Copy,
       class ThreadModel = CComObjectThreadModel >
```

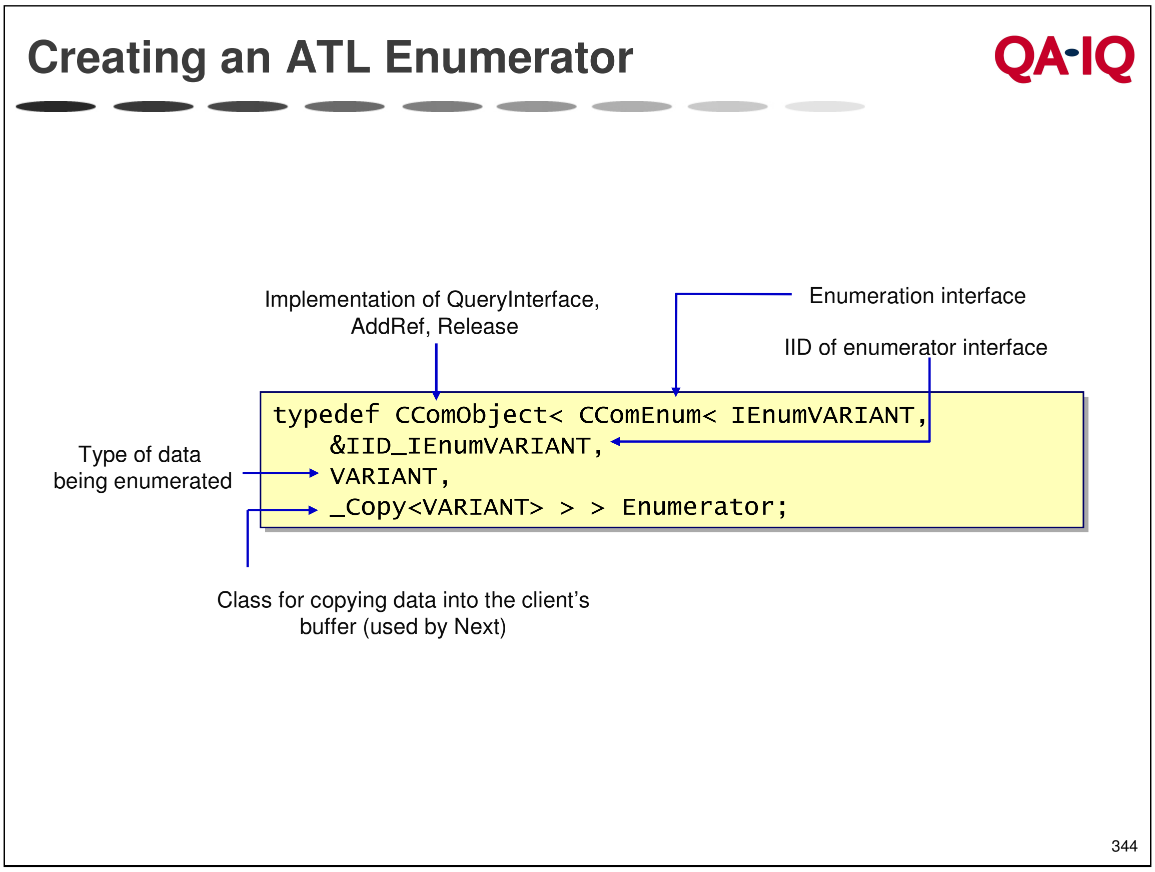
The enumeration interface you want to support
IID of enumerator interface
Type of data being enumerated
Class for copying data into the client's buffer (used by Next)
How thread safe does the enumerator need to be?

343

As enumeration interfaces are all the same except for the underlying data, their implementation can be standardised. As a result, ATL provides a template class CComEnum<>. This template class assumes you've stored your data as an array.

CComEnum derives from CComEnumImpl which actually provides an implementation of the Next, Skip, Reset and Clone methods.

Note that CComEnum<> is an abstract class; it does not implement QueryInterface, AddRef and Release. Therefore you must use CComEnum<> as a parameter of CComObject<> (see next slide).



Creating an ATL Enumerator

```
typedef CComObject< CComEnum< IEnumVARIANT,
    &IID_IEnumVARIANT,
    VARIANT,
    _Copy<VARIANT> > > Enumerator;
```

QA-IQ
Implementation of QueryInterface, AddRef, Release
Enumeration interface
IID of enumerator interface
Type of data being enumerated
Class for copying data into the client's buffer (used by Next)
344

Since CComEnum is an abstract class, it is used as a parameter to CComObject. The CComObject class provides implementations for QueryInterface, AddRef and Release. It is usually best to define a typedef to simplify notation as shown above.

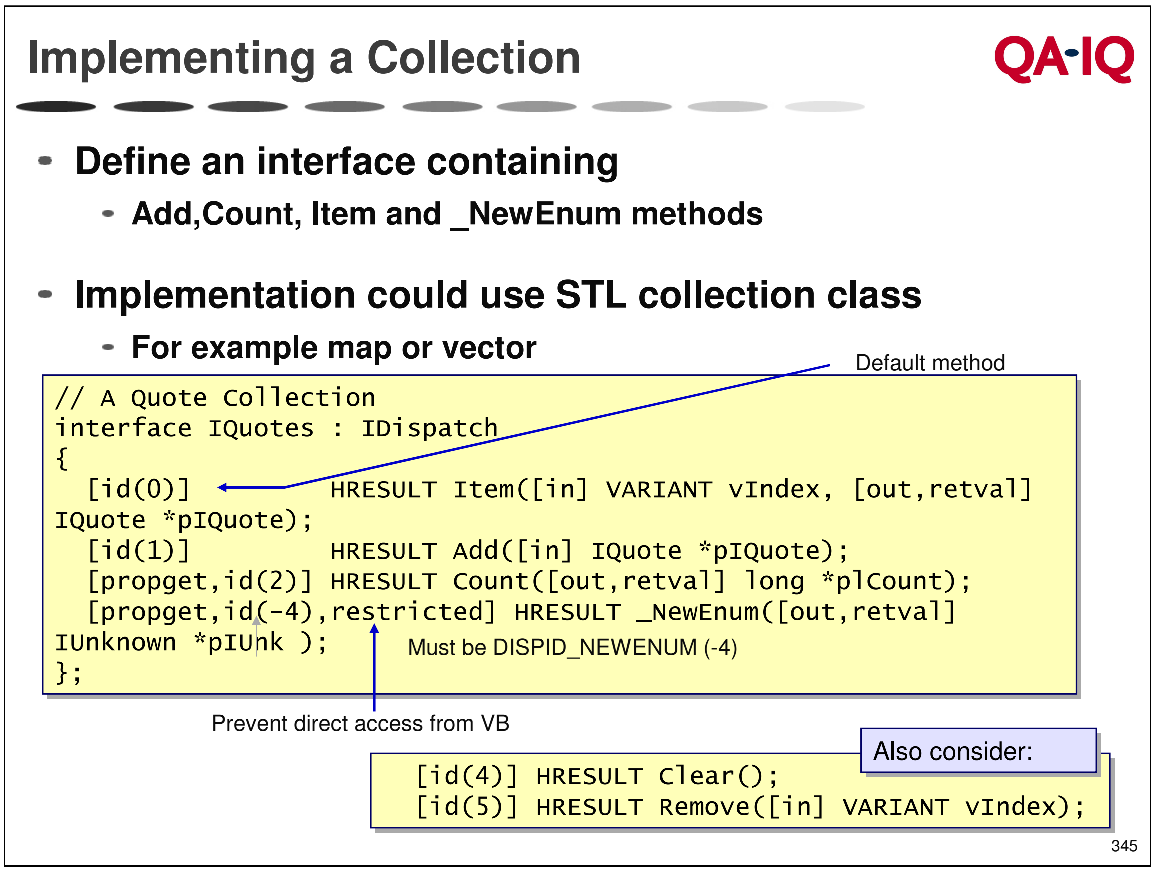
To create an enumerator at run time we use the new operator:

```
Enumerator* pEnumerator = new Enumerator;
pEnumerator->AddRef();
```

Before a CComEnum enumerator can be used you must initialise it. This step is not required with ordinary enumerators but is essential with the ATL defined enumerators:

```
HRESULT hr = pEnumerator->Init(&array[0],
&array[size], 0, AtlFlagTakeOwnership);
```

Notice that the first 2 parameters specify the underlying array representation of the data in the enumerator. You will have to populate this array with data from the collection. The remaining parameters are used to specify who has custody of the retrieved data. In this example we assume the data is copied from the collection and therefore the enumerator has custody of the copies (AtlFlagTakeOwnership).



Implementing a Collection

QA-IQ

- Define an interface containing
  - Add,Count, Item and _NewEnum methods
- Implementation could use STL collection class
  - For example map or vector

Default method

```
// A Quote Collection
interface IQuotes : IDispatch
{
  [id(0)]          HRESULT Item([in] VARIANT vIndex, [out,retval]
IQuote *pIQuote);
  [id(1)]          HRESULT Add([in] IQuote *pIQuote);
  [propget,id(2)] HRESULT Count([out,retval] long *plCount);
  [propget,id(-4),restricted] HRESULT _NewEnum([out,retval]
IUnknown *pIUnk );
};
```

Must be DISPID_NEWENUM (-4)

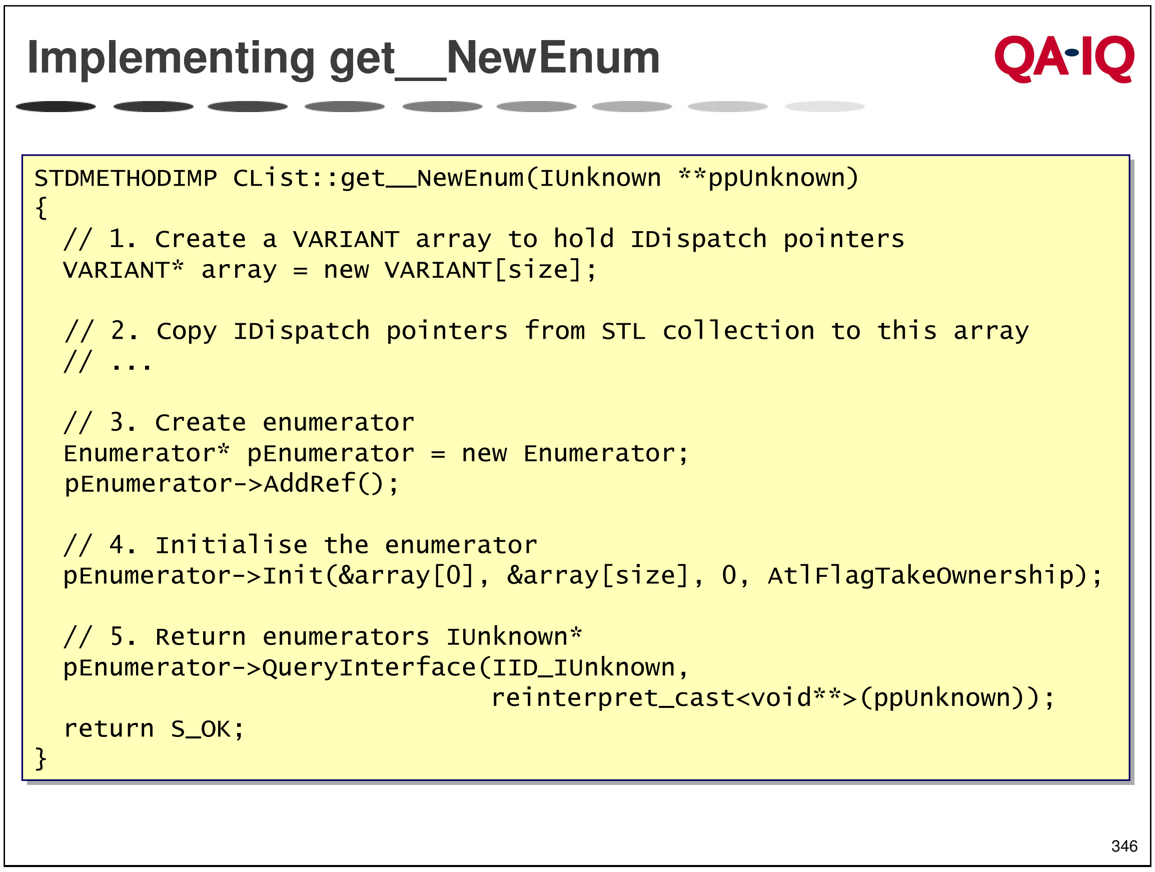
Prevent direct access from VB

Also consider:

```
[id(4)] HRESULT Clear();
[id(5)] HRESULT Remove([in] VARIANT vIndex);
```

345

To use your enumerator you will need to define a collection as a COM class. The class will define a collection interface. As a minimum, a collection interface must provide a **Count** property that returns the number of items in the collection, an **Item** property that returns an item from the collection based on an index, and a **_NewEnum** property that returns an enumerator for the collection. Optionally, collection interfaces can provide **Add** and **Remove** methods to allow items to be inserted into or deleted from the collection, and a **Clear** method to remove all items.



Implementing get__NewEnum

```
STDMETHODIMP CList::get__NewEnum(IUnknown **ppUnknown)
{
  // 1. Create a VARIANT array to hold IDispatch pointers
  VARIANT* array = new VARIANT[size];

  // 2. Copy IDispatch pointers from STL collection to this array
  // ...

  // 3. Create enumerator
  Enumerator* pEnumerator = new Enumerator;
  pEnumerator->AddRef();

  // 4. Initialise the enumerator
  pEnumerator->Init(&array[0], &array[size], 0, AtlFlagTakeOwnership);

  // 5. Return enumerators IUnknown*
  pEnumerator->QueryInterface(IID_IUnknown,
                              reinterpret_cast<void**>(ppUnknown));
  return S_OK;
}
```

346

The most complicated method of a collection interface is the get_NewEnum method. It is also the most important. To implement this method follow the procedure below:

1. Create a VARIANT array to hold an array of IDispatch pointers:

```
VARIANT* array = new VARIANT[size];
```

2. Copy IDispatch pointers from the associated collection (e.g. a Map) to this array:

```
VARIANT* pVariant = &array[0];
for(ITERATOR i = theMap.begin(); i != theMap.end(); ++i)
{
 IBook* pBook = i->second;
 IDispatch* pDispatch;
 pBook->QueryInterface(IID_IDispatch,
reinterpret_cast<void**>(&pDispatch));
 pVariant->vt = VT_DISPATCH;
 pVariant->pdispVal = pDispatch;
 pVariant++;
}
```

3. Create the enumerator:

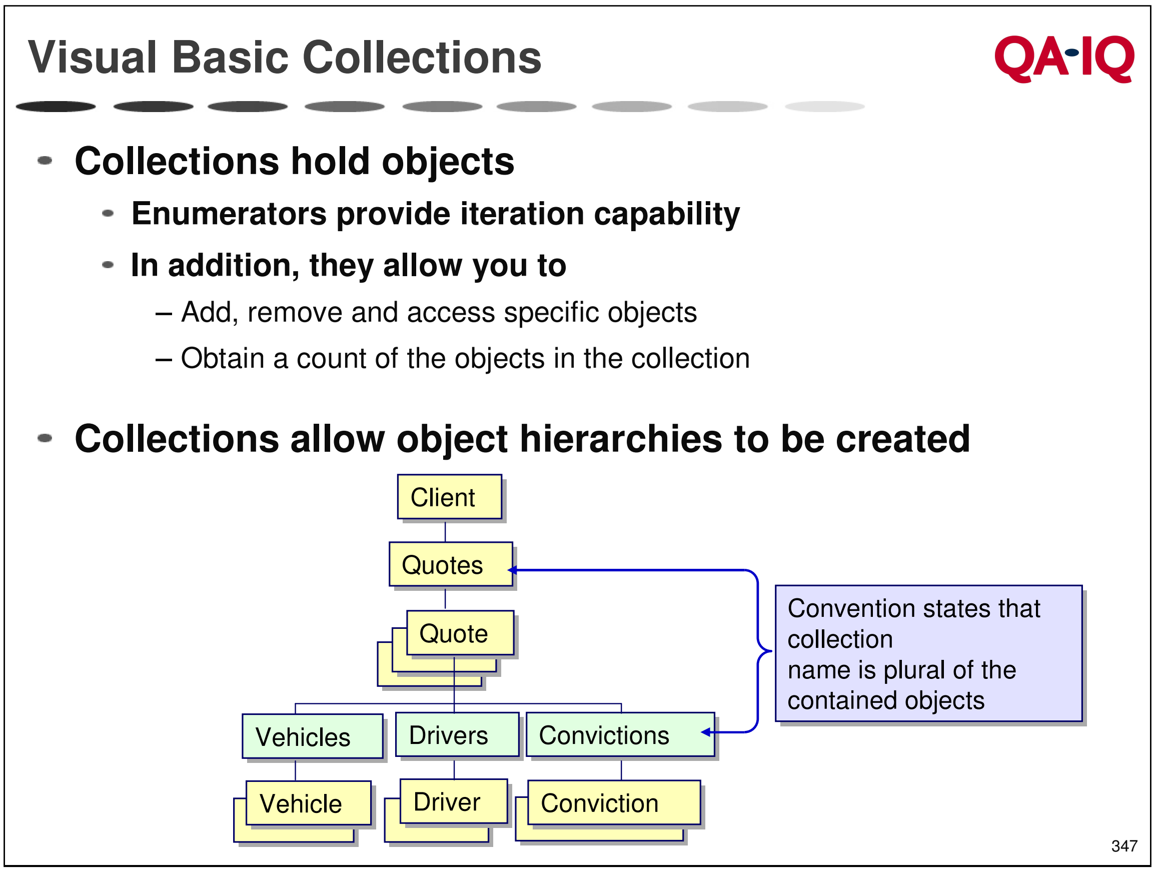
```
Enumerator* pEnumerator = new Enumerator;
pEnumerator->AddRef();
```

4. Initialise the enumerator:

```
pEnumerator->Init(&array[0], &array[size], 0,
AtlFlagTakeOwnership);
```

5. Return the enumerator's IUnknown* pointer to the client:

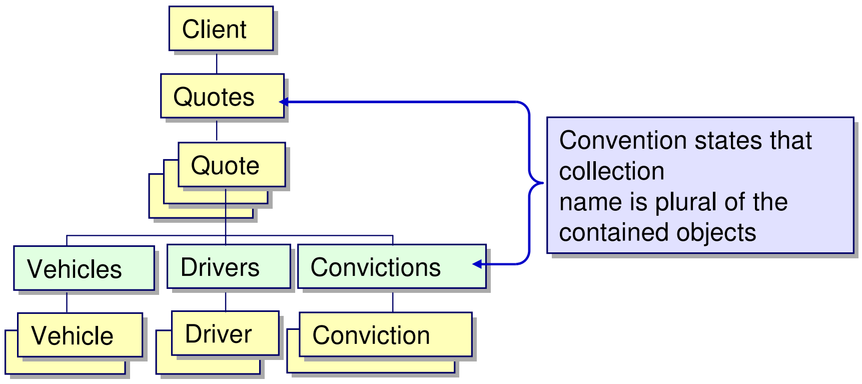
```
pEnumerator->QueryInterface(IID_IUnknown,
reinterpret_cast<void**>(ppUnknown));
```



Visual Basic Collections

QA-IQ

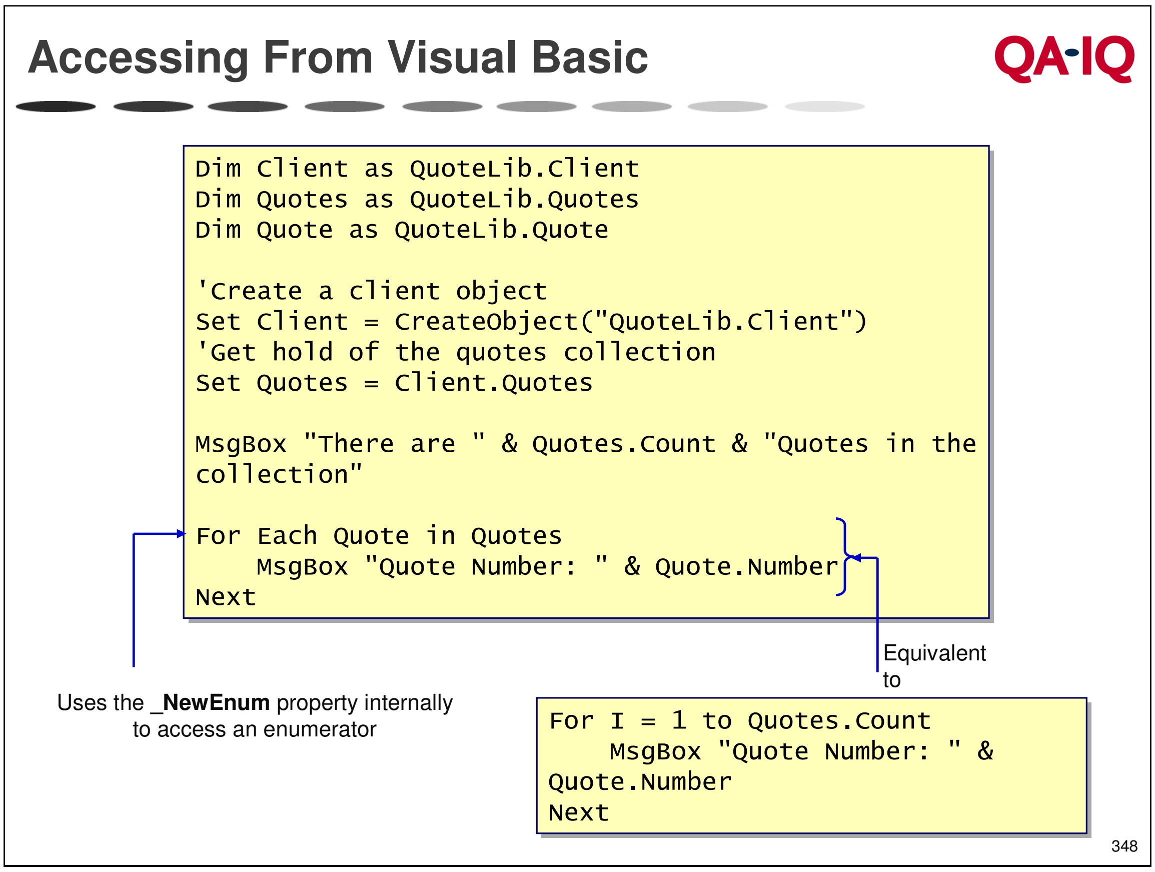
- **Collections hold objects**
  - **Enumerators provide iteration capability**
  - **In addition, they allow you to**
    - – Add, remove and access specific objects
    - – Obtain a count of the objects in the collection
- **Collections allow object hierarchies to be created**


Client
Quotes
Quote
Vehicles
Drivers
Convictions
Vehicle
Driver
Conviction
Convention states that collection
name is plural of the contained objects

347

Visual BAsic makes extensive use of collections and enumerators. All of the Microsoft Office suite exposes a hierarchy of objects as an object model. To access specific objects in the hierarchy you must use collections and enumerators.

One very useful convention to remember when using collections is to pluralise the name of an object when you refer to a collection of these objects. Thus in the above slide, **Quote** refers to a single COM object, but **Quotes** refers to a collection of these objects.



Accessing From Visual Basic

```
Dim Client as QuoteLib.Client
Dim Quotes as QuoteLib.Quotes
Dim Quote as QuoteLib.Quote

'Create a client object
Set Client = CreateObject("QuoteLib.Client")
'Get hold of the quotes collection
Set Quotes = Client.Quotes

MsgBox "There are " & Quotes.Count & "Quotes in the
collection"

For Each Quote in Quotes
    MsgBox "Quote Number: " & Quote.Number
Next
```

Uses the **_NewEnum** property internally to access an enumerator

Equivalent to

```
For I = 1 to Quotes.Count
    MsgBox "Quote Number: " &
Quote.Number
Next
```

348

To conclude the chapter we will look at a Visual Basic client that uses enumerators and collections. Recall that the main reason we use enumerators is for this type of client.

The code is using a type library called **QuoteLib** which defines COM objects and collections. The code:

```
    Dim Client as QuoteLib.Client
    Dim Quotes as QuoteLib.Quotes
    Dim Quote as QuoteLib.Quote
```

creates three COM interface pointers. The **Quotes** pointer points at a collection object. The **Client** object is created with:

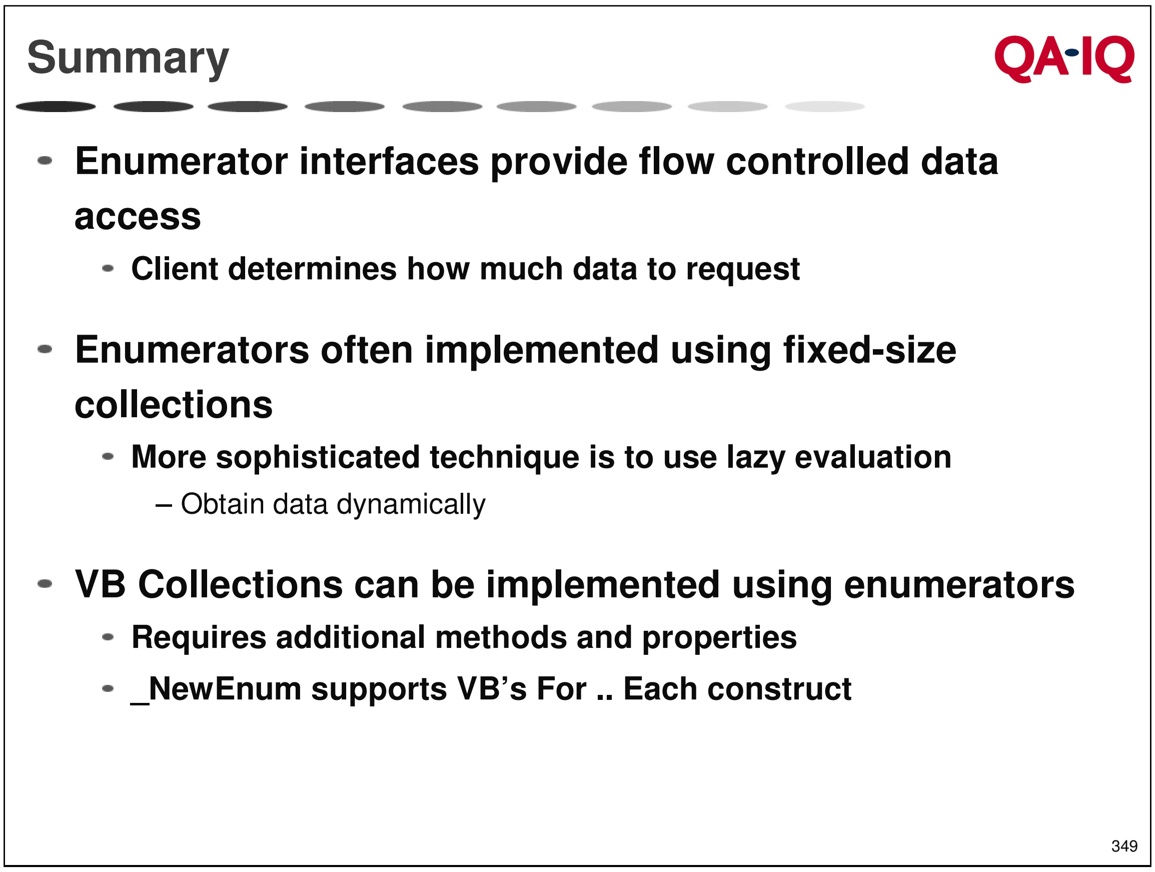
```
    Set Client = CreateObject("QuoteLib.Client")
```

and the quotes collection (already created elsewhere) is retrieved:

```
    Set Quotes = Client.Quotes
```

Finally, the **Quotes** enumerator is used to iterate through the Quotes collection:

```
    For Each Quote in Quotes
        MsgBox "Quote Number: " & Quote.Number
    Next
```



Summary

QA-IQ

- **Enumerator interfaces provide flow controlled data access**
  - **Client determines how much data to request**
- **Enumerators often implemented using fixed-size collections**
  - **More sophisticated technique is to use lazy evaluation**
    - – Obtain data dynamically
- **VB Collections can be implemented using enumerators**
  - **Requires additional methods and properties**
  - **_NewEnum supports VB's For .. Each construct**

349