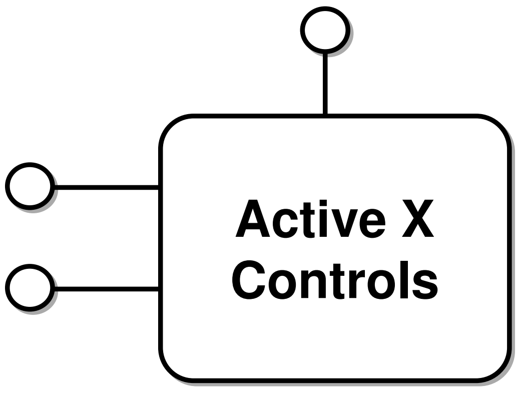
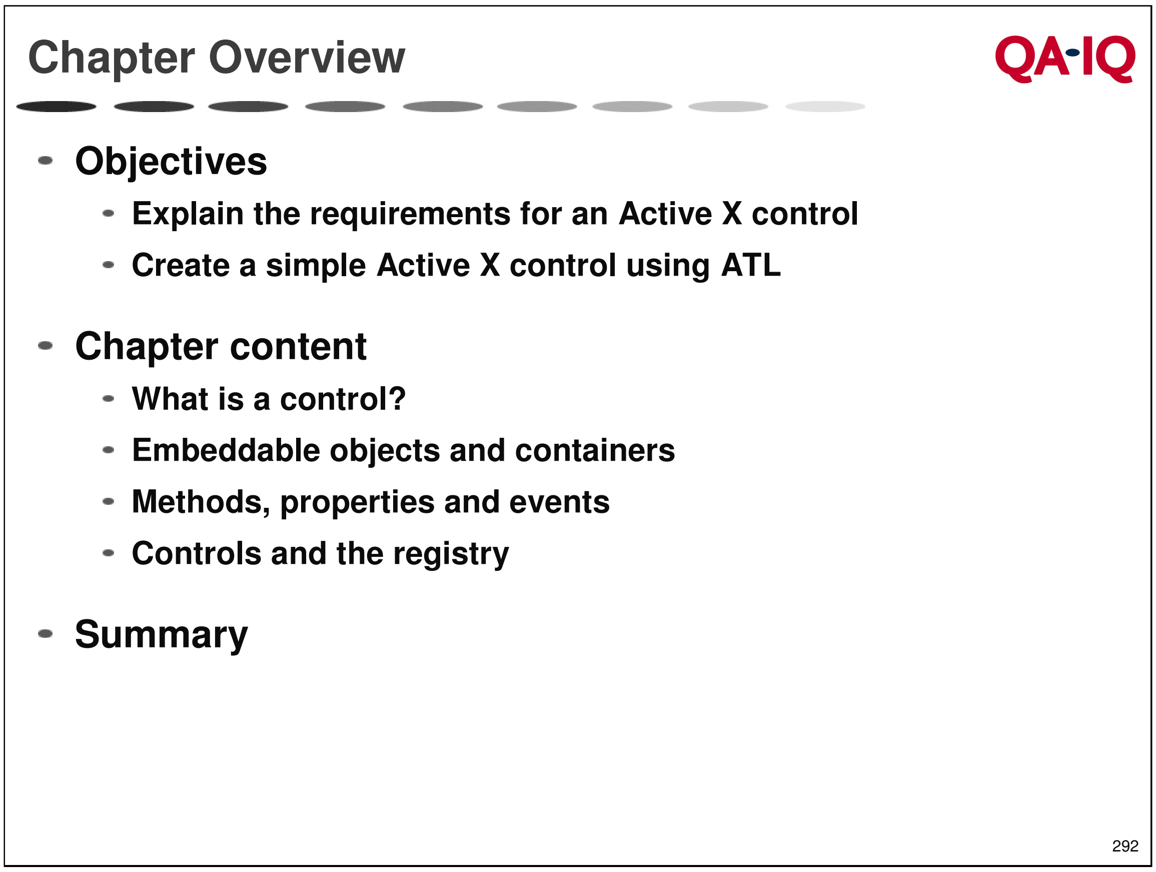
ActiveX Controls

QA-IQ



291



Chapter Overview

QA-IQ

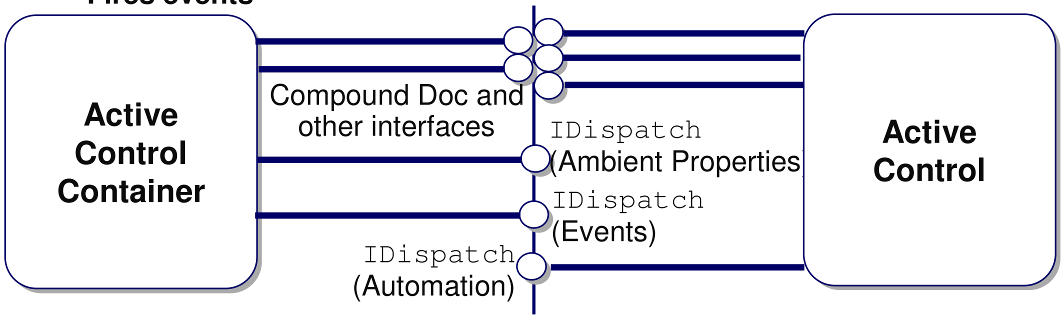
- **Objectives**
  - **Explain the requirements for an Active X control**
  - **Create a simple Active X control using ATL**
- **Chapter content**
  - **What is a control?**
  - **Embeddable objects and containers**
  - **Methods, properties and events**
  - **Controls and the registry**
- **Summary**

292

What is an Active Control?

QA-IQ

- By definition
  - Supports IUnknown and is self registering
- But typically...
  - Has a specific and dedicated user interface
    - i.e. it is embeddable
  - Has methods, and properties
    - i.e. is automatable
  - Fires events



293

An Active Control is a component object with a user interface and a programming interface; it *is* both an automation object and a compound document object. In addition, an Active Control can call two `IDispatch` interfaces exported by the container to query the container's *ambient properties* and to trigger *events*.

A control describes its characteristics through named *properties*. Properties may affect visual and behavioural characteristics. A control's properties define its "state" and some or all of these properties may be persistent. These properties are defined as part of the control's automation interface and can be set and queried by any automation controller or the control itself.

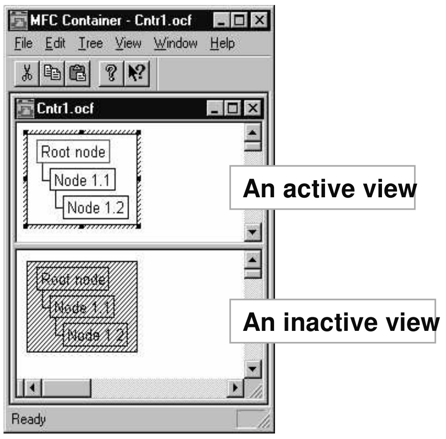
A control describes its functionality with named *methods* and *events*. Methods are functions which are defined and implemented by the control itself. Events are functions which are defined by the control but implemented by the control container. Events are implemented as an `IDispatch` interface exposed by the container. Not shown on the diagram above are the *connection point* interfaces that Microsoft had to develop to provide a mechanism for associating an event `IDispatch` to a control. If a container provides a handler for an event that is not supported by the control then the event handler is never called. If a container fails to provide an event handler then the event is ignored.

A control provides a user interface which defines how it appears when visible in a container; they are compound document objects which support in-place activation and editing. Controls are marked as *inside-out* (activated by a single-click) and will often be automatically activated when visible. Some controls may only be visible at design time.

Embeddable Objects

QA-IQ

- An object with a user interface
- Data saved by the container
  - Using structured storage



Active Object
• Displays a window
• Receives Keyboard and
• Mouse messages

Inactive Object
• Displays an image

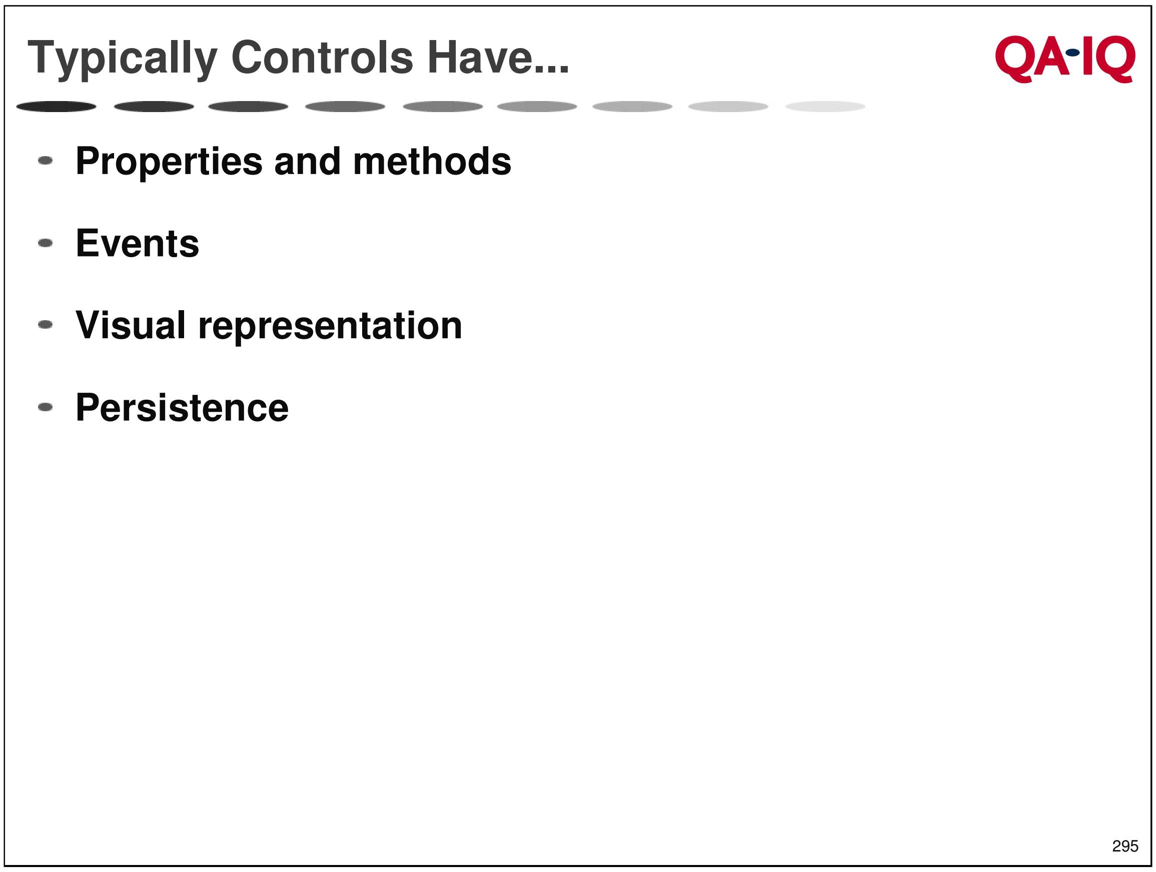
294

In Microsoft's 'Information At Your Fingertips' model embedding and linking was the core part. Embedding and Linking are also the L and E of OLE.

An embedding server passes two types of data to a client, it passes a graphical representation, and the 'native' data that the server needs to recreate the object. The client can display the graphical representation (usually a metafile) passed to it, this will be an 'inactive' view of the object, inactive because the server will not be running at this point.

To edit the data a user will 'activate' the server (typically by double-clicking). At this point the client will launch the server (either an EXE or a DLL), pass across the server's 'native' data which the server uses to re-create the object. The client and server then negotiate over the extent and position, the server then displays its own window inside the server (if it is UI Active as opposed to fully open).

An Active Control is (typically) an embedded object.



Typically Controls Have...

QA-IQ

- Properties and methods
- Events
- Visual representation
- Persistence

295

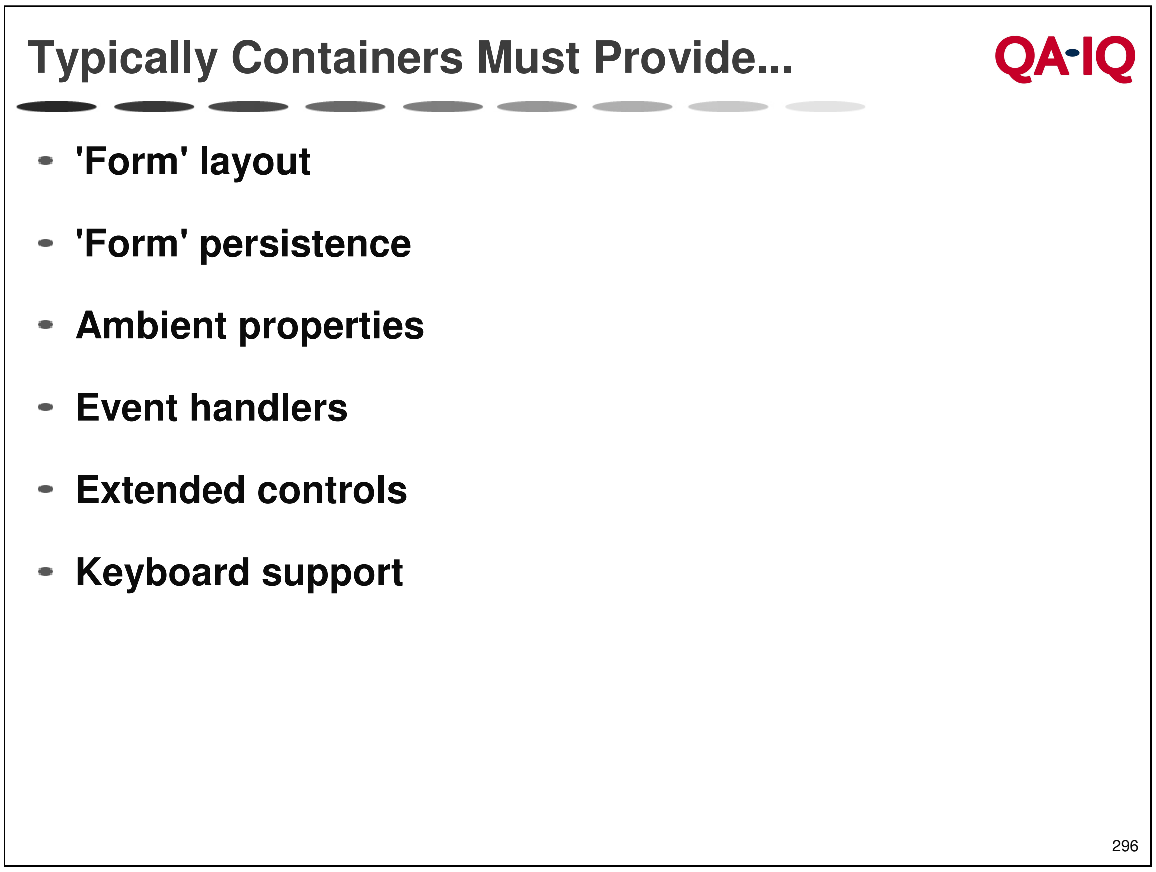
A typical control has all of the above.

As we've just mentioned a control can be an embedded object, which gives it the ability to have a user interface.

It will also be an automation object (directly implemented through `IDispatch` or as a dual interface) which gives clients access to its methods and properties.

Controls can fire 'asynchronous' events in a container.

Controls typically support two persistence interfaces, `IPersistPropertyBag` and `IPersistStreamInit`.



Typically Containers Must Provide...

QA-IQ

- 'Form' layout
- 'Form' persistence
- Ambient properties
- Event handlers
- Extended controls
- Keyboard support

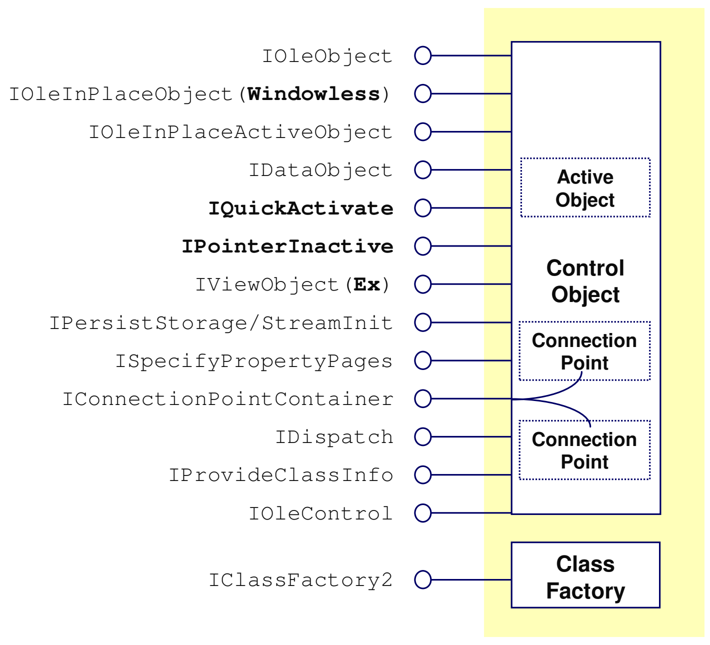
296

Controls live inside containers of which there are various types.

Embedding containers (such as word), OLE control containers (such as Visual Basic 6 or MFC 4.x) and finally Active Control Containers (such as Internet Explorer).

Control Interfaces

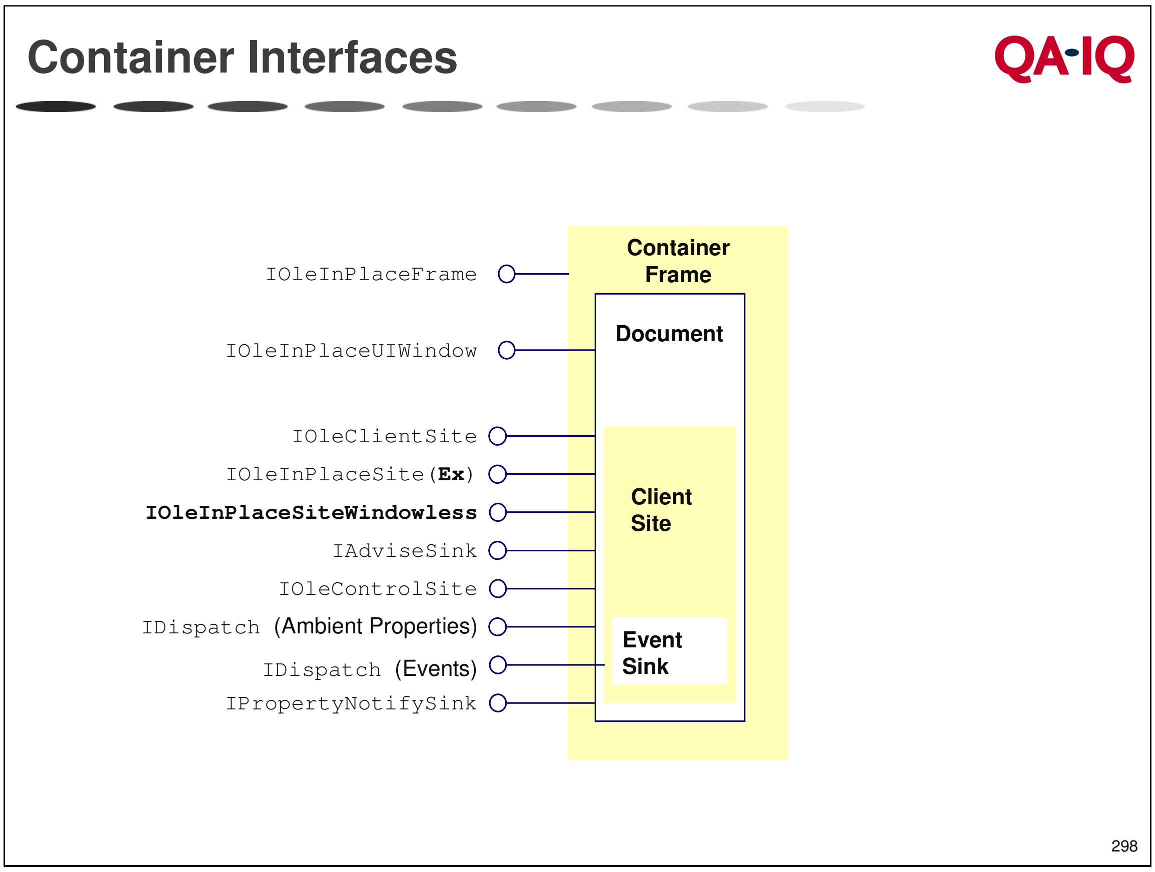
QA-IQ



297

Listed above are the interfaces a control may support.

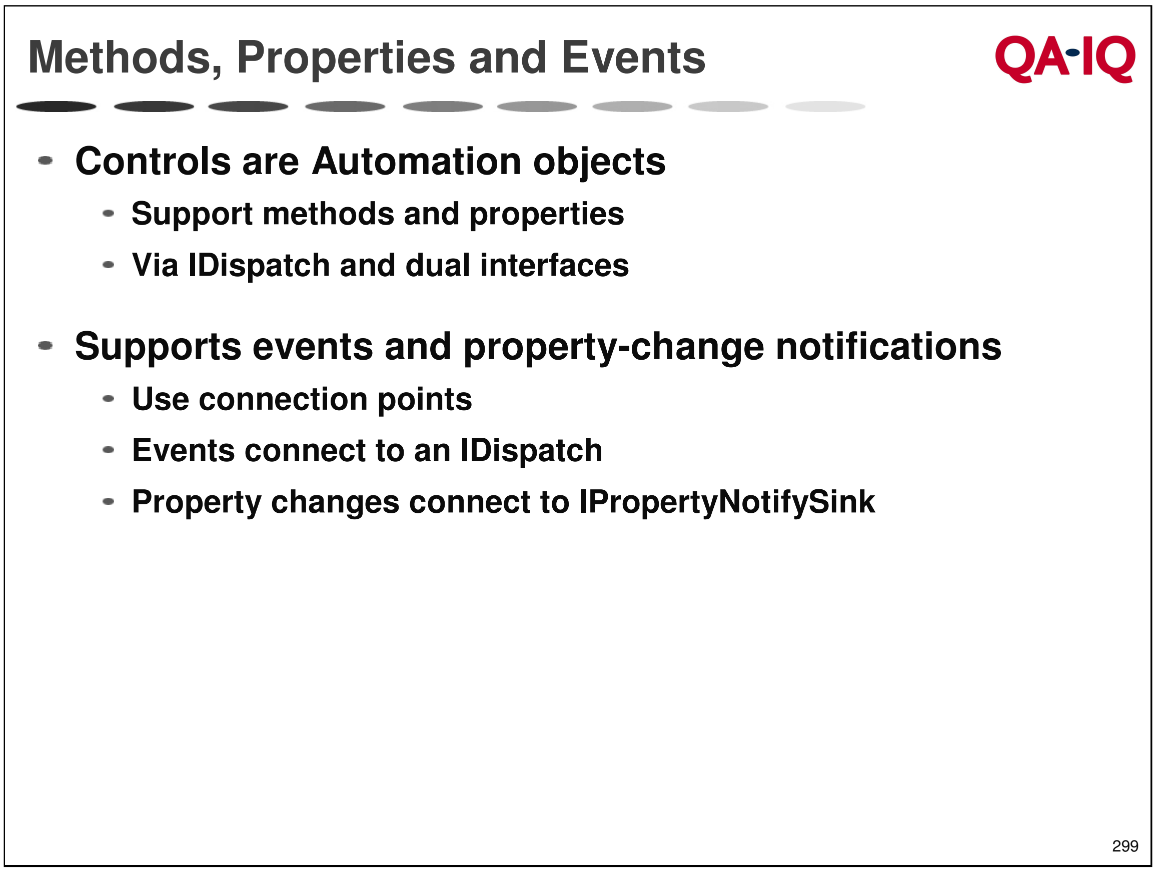
Those shown in bold are the new ActiveX interfaces.



Container Interfaces

Listed above are the interfaces a container may support.

Those shown in bold are the new ActiveX interfaces.



Methods, Properties and Events

QA-IQ

- **Controls are Automation objects**
  - **Support methods and properties**
  - **Via IDispatch and dual interfaces**
- **Supports events and property-change notifications**
  - **Use connection points**
  - **Events connect to an IDispatch**
  - **Property changes connect to IPropertyNotifySink**

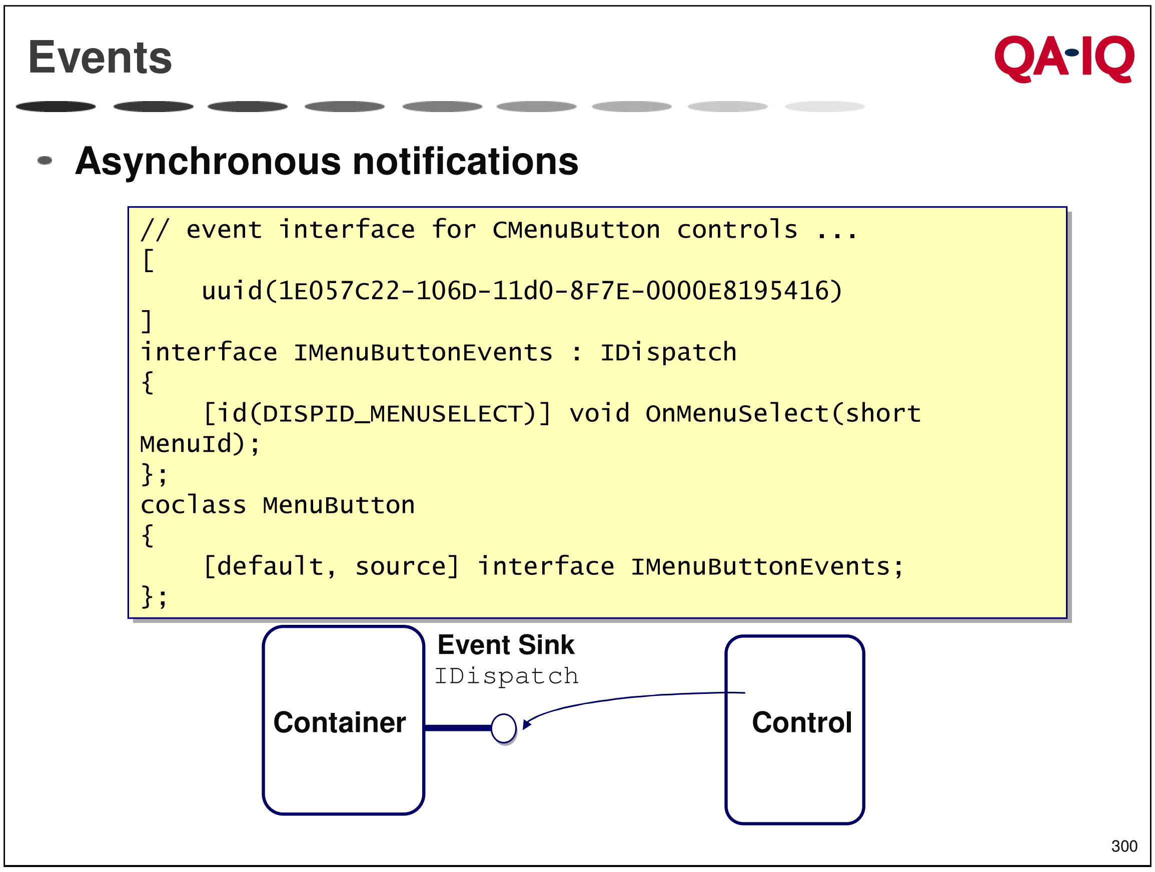
299

As we've said a number of times - controls are automation objects. Typically they support methods and properties via a dual interface.

More important is that controls have two outgoing interfaces. The first of these is for *Events*.

Events are supported through an `IDispatch` interface that is implemented on the container, and which is connected to the control via a connection point. Controls 'fire' events asynchronously to signal that something significant has happened, such as a timer control firing a timer event. Typically containers will associate user-defined code with any events they need to respond to.

The other 'outgoing' interface supported by controls is `IPropertyNotifySink`. Whereas you can think of events as being asynchronous methods, `IPropertyNotifySink` supports asynchronous property change notifications.

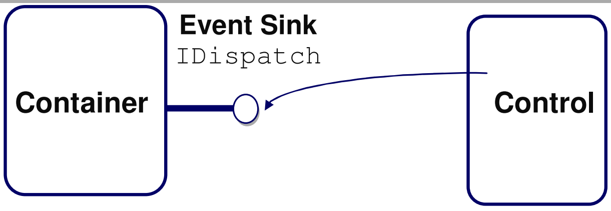


Events

QA-IQ

- **Asynchronous notifications**

```
// event interface for CMenuButton controls ...
[
    uuid(1E057C22-106D-11d0-8F7E-0000E8195416)
]
interface IMenuButtonEvents : IDispatch
{
    [id(DISPID_MENUSELECT)] void OnMenuSelect(short
MenuId);
};
coclass MenuButton
{
    [default, source] interface IMenuButtonEvents;
};
```

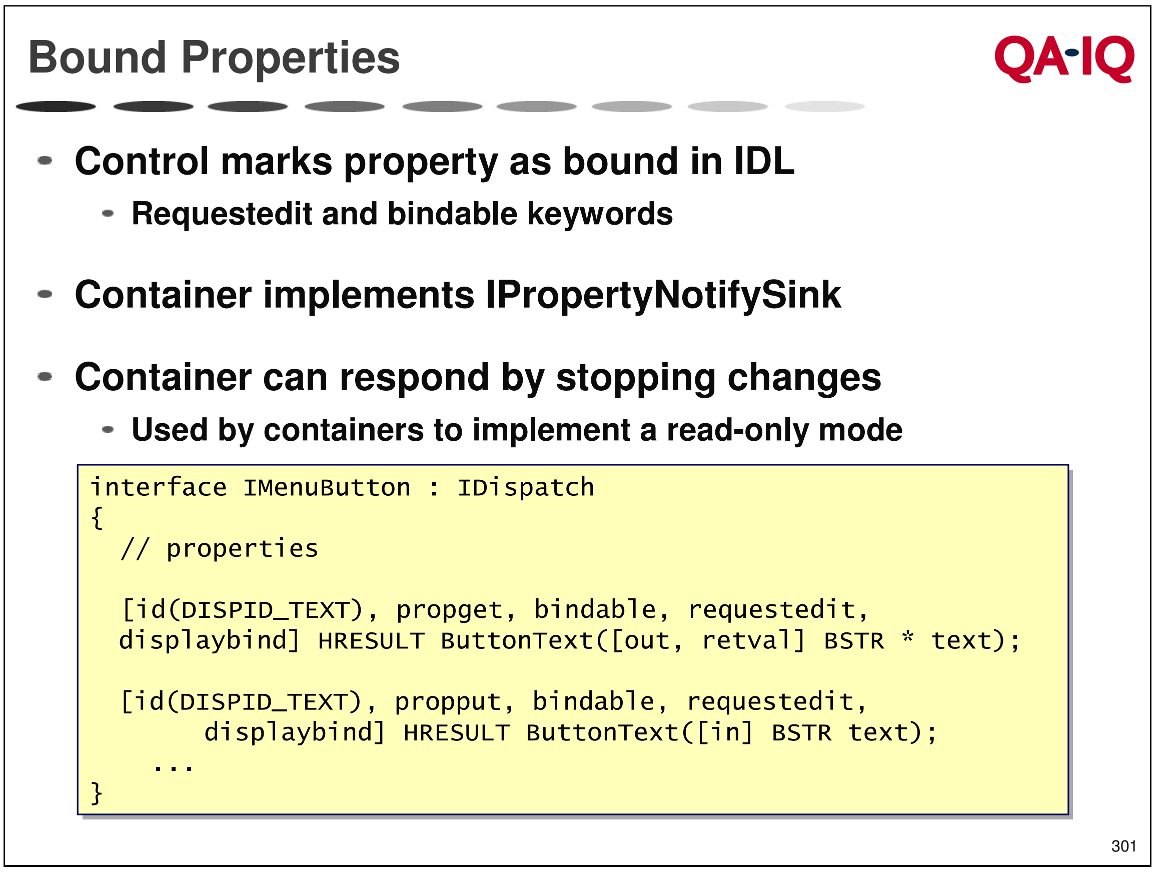


300

As was said on the previous slide, events are asynchronous notifications.

To define an event set the controls type library contains a description of a `dispinterface`. This interface is marked as 'source' in the IDL file, which means that it must be implemented by the client not by the control.

Typically a container will offer these methods up to a developer. The developer can then decide which methods they are interested in, and provide code that is called in response to the event.



Bound Properties

QA-IQ

- Control marks property as bound in IDL
  - Requestedit and bindable keywords
- Container implements IPropertyNotifySink
- Container can respond by stopping changes
  - Used by containers to implement a read-only mode

```
interface IMenuButton : IDispatch
{
  // properties

  [id(DISPID_TEXT), propget, bindable, requestedit,
  displaybind] HRESULT ButtonText([out, retval] BSTR * text);

  [id(DISPID_TEXT), propput, bindable, requestedit,
        displaybind] HRESULT ButtonText([in] BSTR text);
    ...
}
```

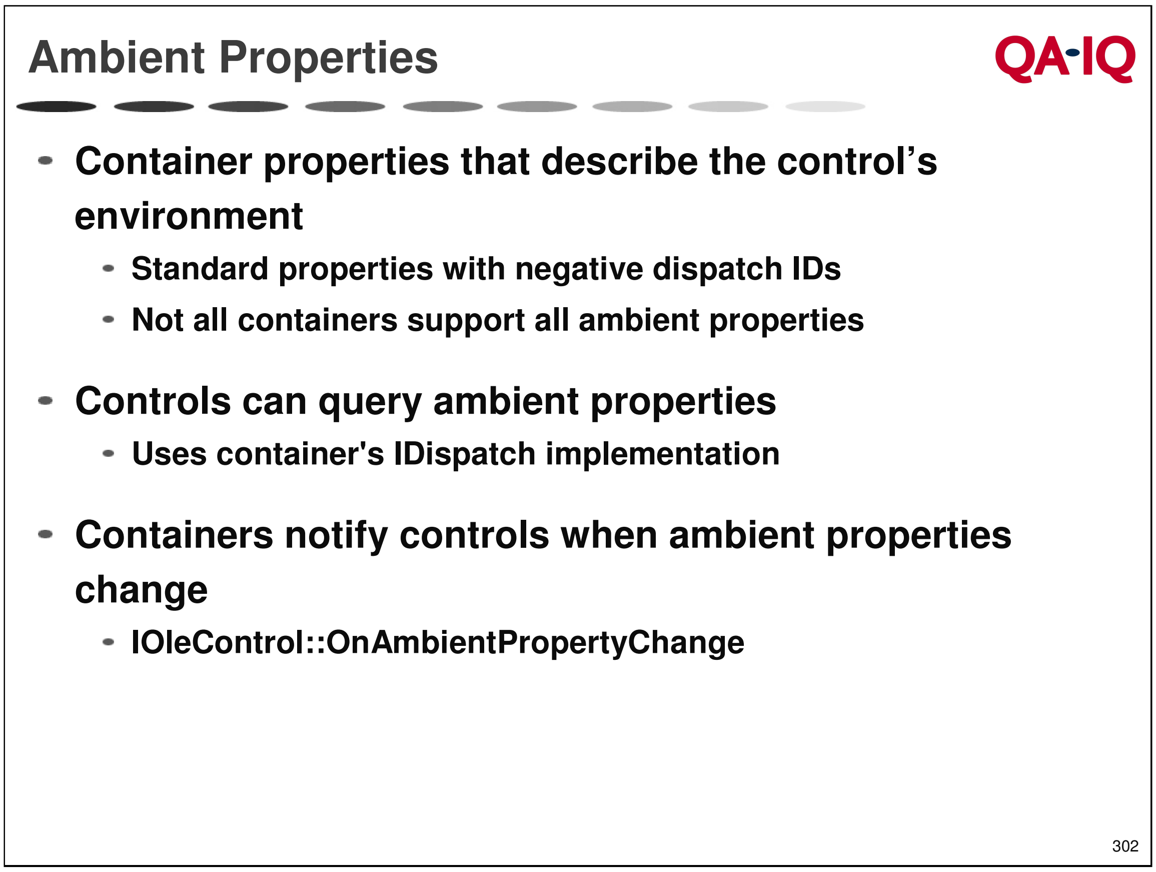
301

Asynchronous properties (also known as bound properties) are implemented in much the same way as events.

The control stores information in its type library identifying any properties about which it wants to notify the container if these properties changed.

In the IDL file (and hence the type library) a control identifies a bound property in its dispatch interface definition (either `dispinterface` or dual interface) by adding the `bindable` flag to the property definition. If the property wants to first check with the container that it can be changed then it should put the `requestedit` flag in the registry.

The interface through which a control can notify the container is `IPropertyNotifySink`, this has two methods `OnChanged()` and `OnRequestEdit()`. The control calls `OnRequestEdit()` if it is marked as `requestedit` and the control wants to check that the container is not in a 'read-only' mode. If the container returns TRUE then the control calls `OnChanged()` to notify the change.



Ambient Properties

- **Container properties that describe the control’s environment**
  - **Standard properties with negative dispatch IDs**
  - **Not all containers support all ambient properties**
- **Controls can query ambient properties**
  - **Uses container's IDispatch implementation**
- **Containers notify controls when ambient properties change**
  - **IOleControl::OnAmbientPropertyChange**

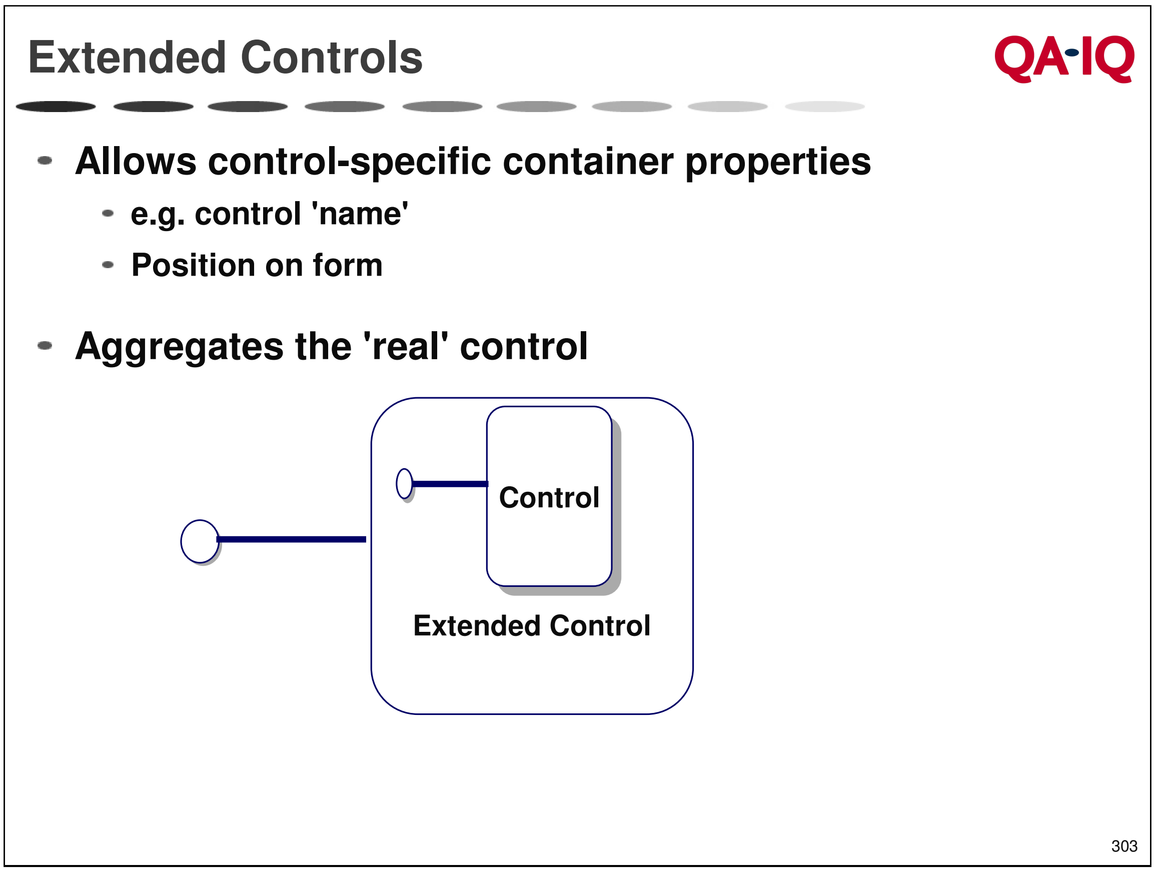
302

Controls typically use the ambient properties of their container as defaults for their own internal properties. For example, a control embedded in a word-processor document would probably use the ambient font of the surrounding text for any text it displayed itself.

Ambient properties are standard (stock) properties; their (negative) dispatch IDs and types are defined by Microsoft. Containers are not required to support any ambient properties so your control should be able to choose suitable default property values if your container does not supply any.

You can query ambient properties at any time by calling through the containers `IDispatch` interface. Most frameworks have some sort of `GetAmbientProperty()` method to which you pass the `DISPID_*` value.

Your control receives notifications whenever the container changes an ambient property. The actual notification calls the `OnAmbientPropertyChange()` method of the `IOleControl` interface



Extended Controls

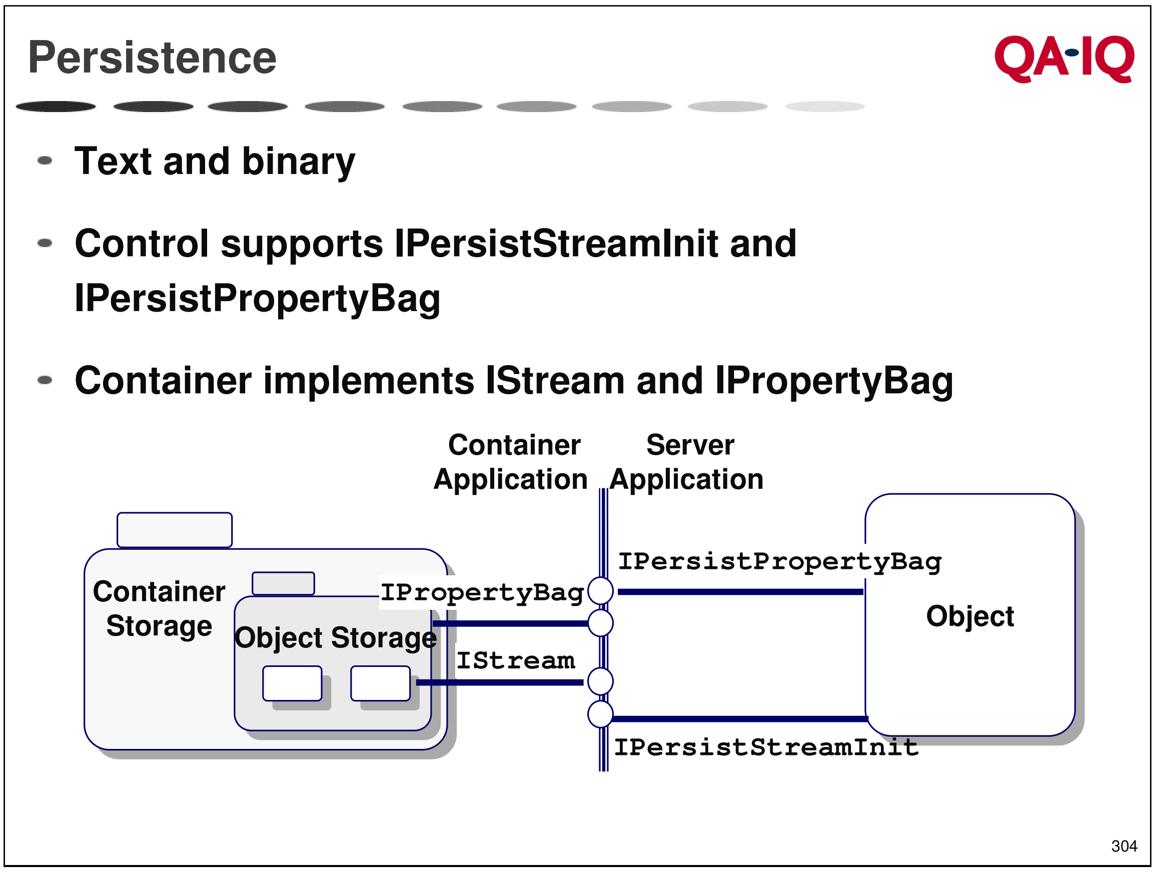
QA-IQ

- Allows control-specific container properties
  - e.g. control 'name'
  - Position on form
- Aggregates the 'real' control

Control
Extended Control
303

Containers need the ability to hold information about controls that apply to all controls. This information is expressed as ambient properties. But a control also needs to hold information about controls that apply to uniquely to each control, good examples of this are the controls ‘name’ and the controls position.

To implement this a container will aggregate the control providing an implementation of the controls `IDispatch`. This means that a container doesn’t need to be concerned whether it is getting an extended property or a property of the control itself, it is all accessed through one `IDispatch`.



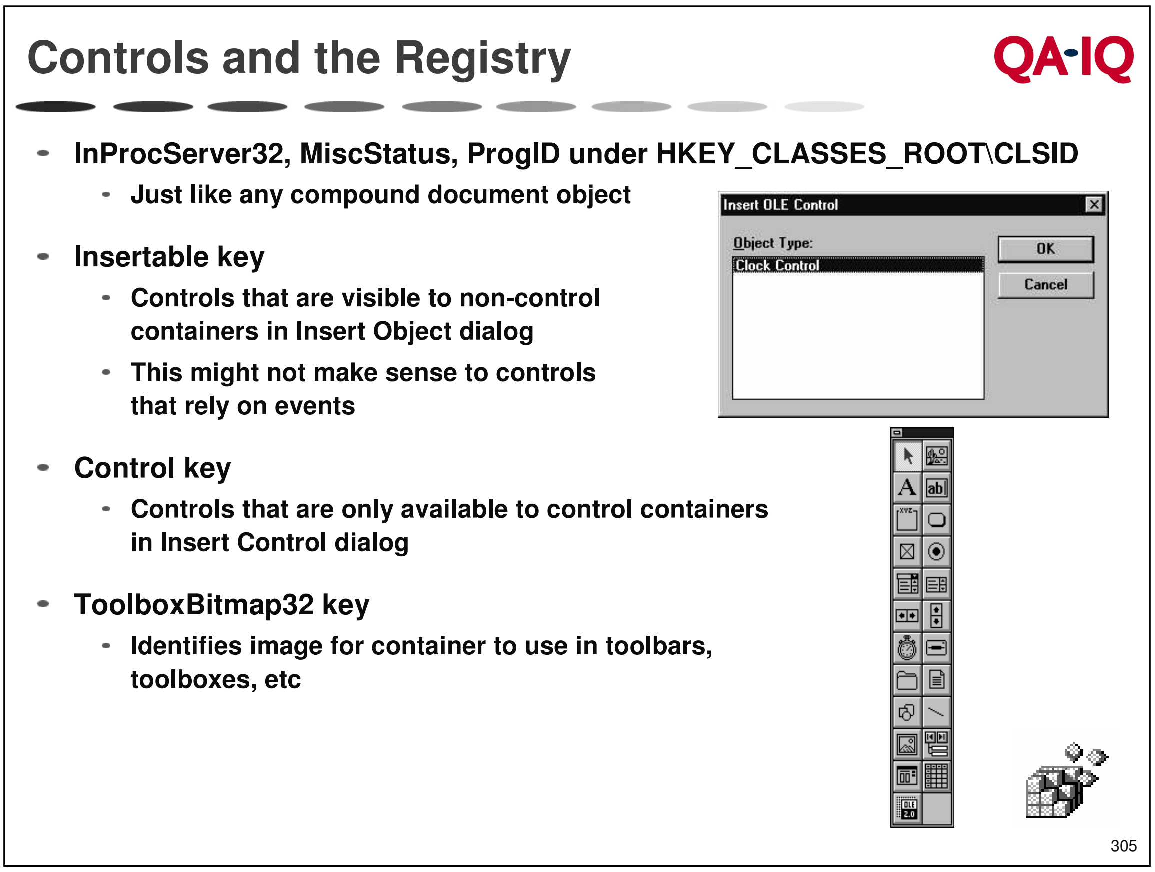
Persistence

- Text and binary
- Control supports IPersistStreamInit and IPersistPropertyBag
- Container implements IStream and IPropertyBag

Container Application
Server Application
Container Storage
Object Storage
IPropertyBag
IStream
IPersistPropertyBag
Object
IPersistStreamInit
304

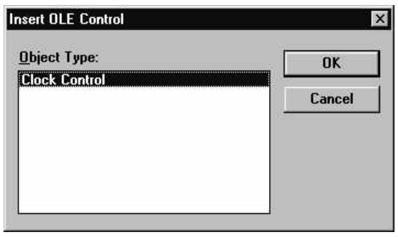
Controls need to support persistence so that at design-time containers can save away the 'design' state of the control and at run-time containers can load the saved state.

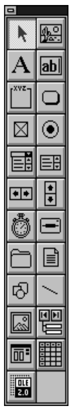
To manage persistence controls typically implement two interfaces, `IPersistPropertyBag` provides 'text' persistence and `IPersistStreamInit` offers 'binary' persistence.



Controls and the Registry

- InProcServer32, MiscStatus, ProgID under HKEY_CLASSES_ROOT\CLSID
  - Just like any compound document object
- Insertable key
  - Controls that are visible to non-control containers in Insert Object dialog
  - This might not make sense to controls that rely on events
- Control key
  - Controls that are only available to control containers in Insert Control dialog
- ToolboxBitmap32 key
  - Identifies image for container to use in toolbars, toolboxes, etc





305

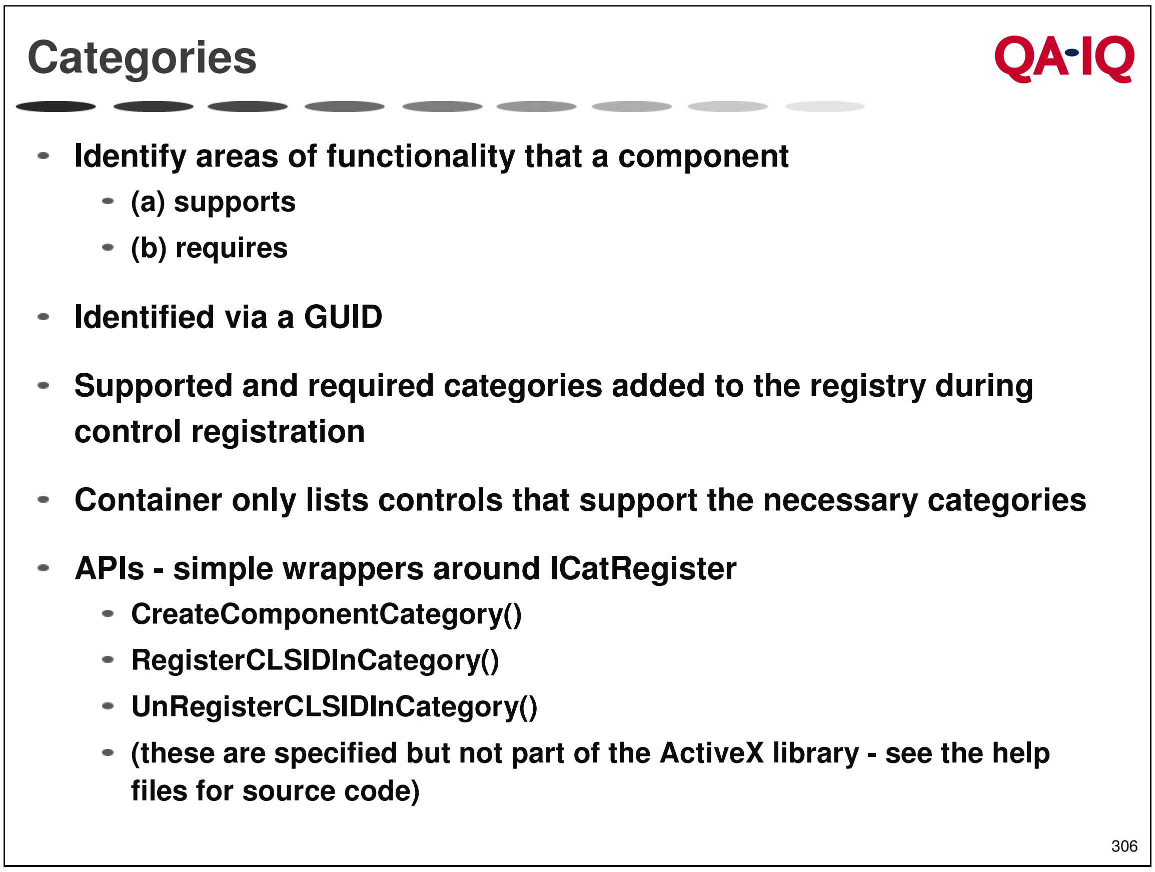
The registry entries for a control are like those of any other embeddable object: the control registers a ProgID key under which it stores its CLSID, and under the CLSID key it stores its *InProcServer32* entry along with "verbs", "MiscStatus", etc. However, there are a few keys that are of specific interest to containers.

The *Insertable* key may be registered under the control's ProgID as well as under its CLSID, but is not required for all controls. Existing compound document containers check for the *Insertable* key in either location in order to populate their *Insert Object* dialog boxes. If a control includes this key then it indicates that it can be inserted into non-control container applications. Some controls, like timers, which are practically useless without their events, would not show themselves to regular container applications.

If a control only wants to appear to control containers, then it does not include the *Insertable* key but instead includes a key named *Control* registered only under the CLSID key. Containers can use *Control* to populate an *Insert Control* dialog box.

Some controls that do not want to be exposed to any container, other than those that have hard-coded the control's CLSID, will have neither *Insertable* nor *Control* entries in the registry. This is increasingly likely when components can be connected together in such a way that most are non- graphical "line of business" objects.

The *ToolboxBitmap32* key defines a DLL path and a resource identifier, similarly to the *DefaultIcon* entry for compound documents which allows a container to extract an iconic representation of a compound document object. These new keys allow a container to extract a 16*16 button face image for each registered control such that the container can create a toolbar or toolbox populated with controls, such as that from Visual Basic shown above.



Categories

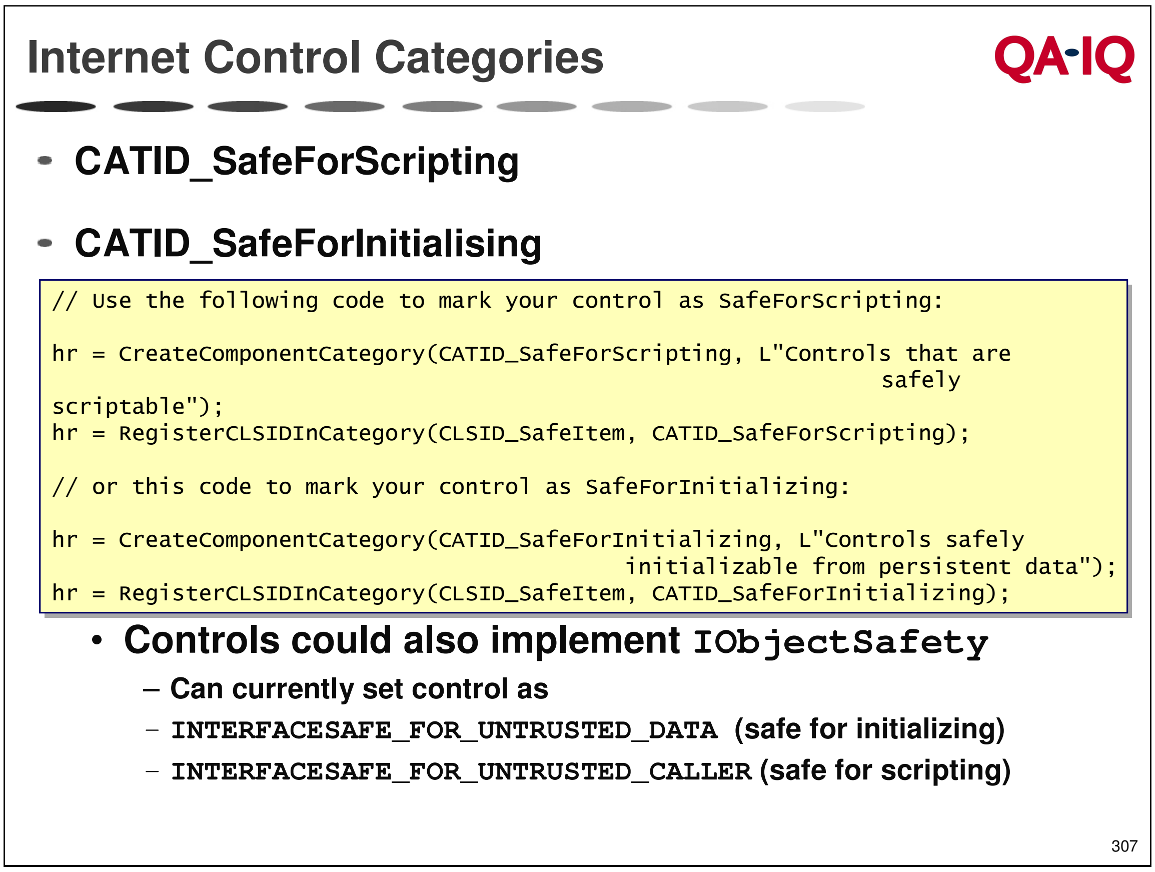
- Identify areas of functionality that a component
  - (a) supports
  - (b) requires
- Identified via a GUID
- Supported and required categories added to the registry during control registration
- Container only lists controls that support the necessary categories
- APIs - simple wrappers around ICatRegister
  - CreateComponentCategory()
  - RegisterCLSIDInCategory()
  - UnRegisterCLSIDInCategory()
  - (these are specified but not part of the ActiveX library - see the help files for source code)

306

COM component categories allow a control to state what it can do and what support it needs from a container via entries in the registry. If a container does not want to or is not able to support a specific area of functionality, (for example *databinding*) , the container will not wish to insert controls that list *databinding* as a required category in order to do their work. Component Categories allow areas of functionality to be identified, so that the control container can avoid those controls that have that area as a requirement. Component Categories are specified separately as part of OLE and are not specific to the Active Control architecture, the specification for component categories includes a set of APIs for manipulation of the component category registry keys.

Categories are created and maintained using the `ICatRegister` interface. This provides methods to register and unregister categories, to register and unregister required categories and to register and unregister implemented categories.

The APIs listed above are wrappers around some of the `ICatRegister` methods; these are not part of the official ActiveX library yet but the source code for the functions can be found in the ActiveX help file and in the samples



Internet Control Categories

QA-IQ

- CATID_SafeForScripting
- CATID_SafeForInitialising

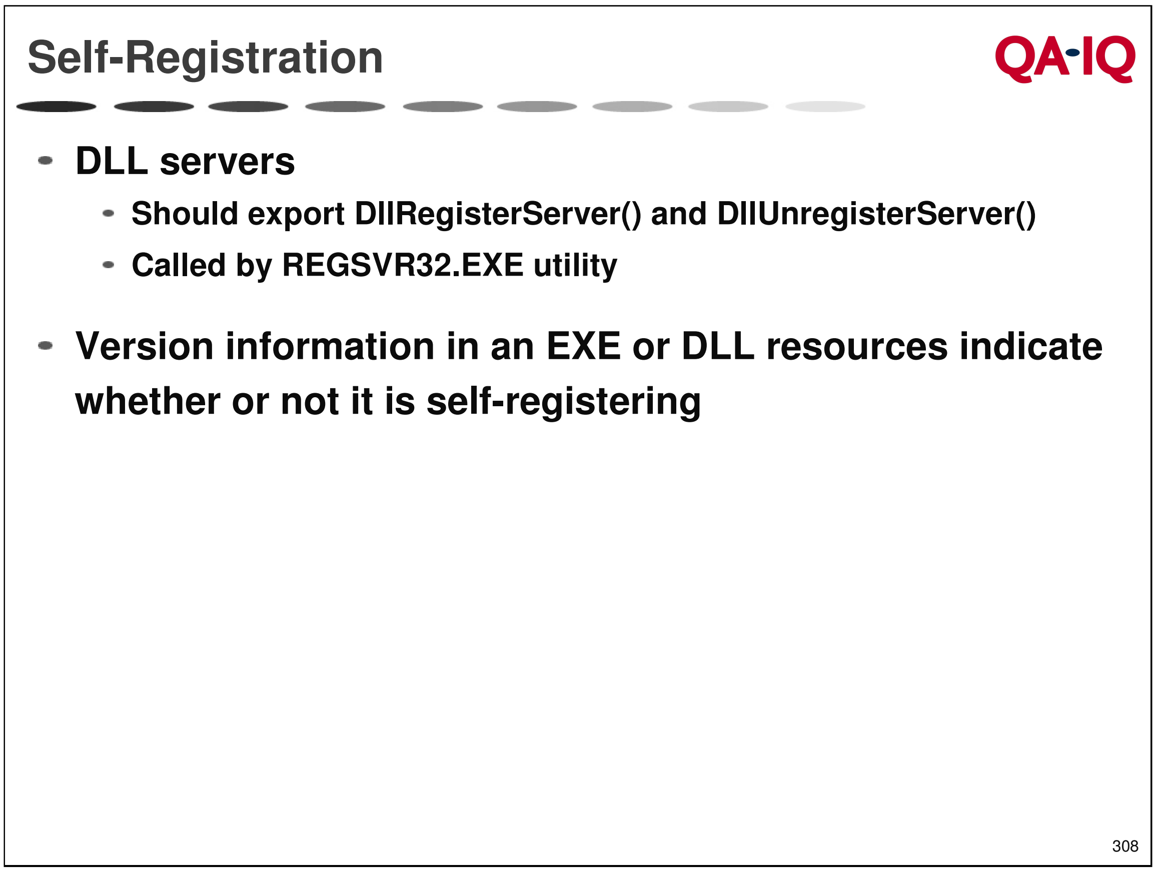
```
// Use the following code to mark your control as SafeForScripting:

hr = CreateComponentCategory(CATID_SafeForScripting, L"Controls that are
                                                             safely
scriptable");
hr = RegisterCLSIDInCategory(CLSID_SafeItem, CATID_SafeForScripting);

// or this code to mark your control as SafeForInitializing:

hr = CreateComponentCategory(CATID_SafeForInitializing, L"Controls safely
                                          initializable from persistent data");
hr = RegisterCLSIDInCategory(CLSID_SafeItem, CATID_SafeForInitializing);
```

- Controls could also implement `IObjectSafety`
  - Can currently set control as
  - `INTERFACESAFE_FOR_UNTRUSTED_DATA` (safe for initializing)
  - `INTERFACESAFE_FOR_UNTRUSTED_CALLER` (safe for scripting)

307

Internet controls have two specific 'safety' categories - CATID_SafeForScripting and CATID_SafeForInitializing. These tell a container (such as Internet Explorer) that these controls can be safely scripted and can be safely initialised.

If a control implements `IObjectSafety` then a container can use this to ask the control what its safety options are, and to ask the control to set its safety options. The only two options currently supported are `INTERFACESAFE_FOR_UNTRUSTED_DATA` and `INTERFACESAFE_FOR_UNTRUSTED_CALLER`, which correspond to safe for initializing and safe for scripting.



Self-Registration

- **DLL servers**
  - **Should export DllRegisterServer() and DllUnregisterServer()**
  - **Called by REGSVR32.EXE utility**
- **Version information in an EXE or DLL resources indicate whether or not it is self-registering**

308

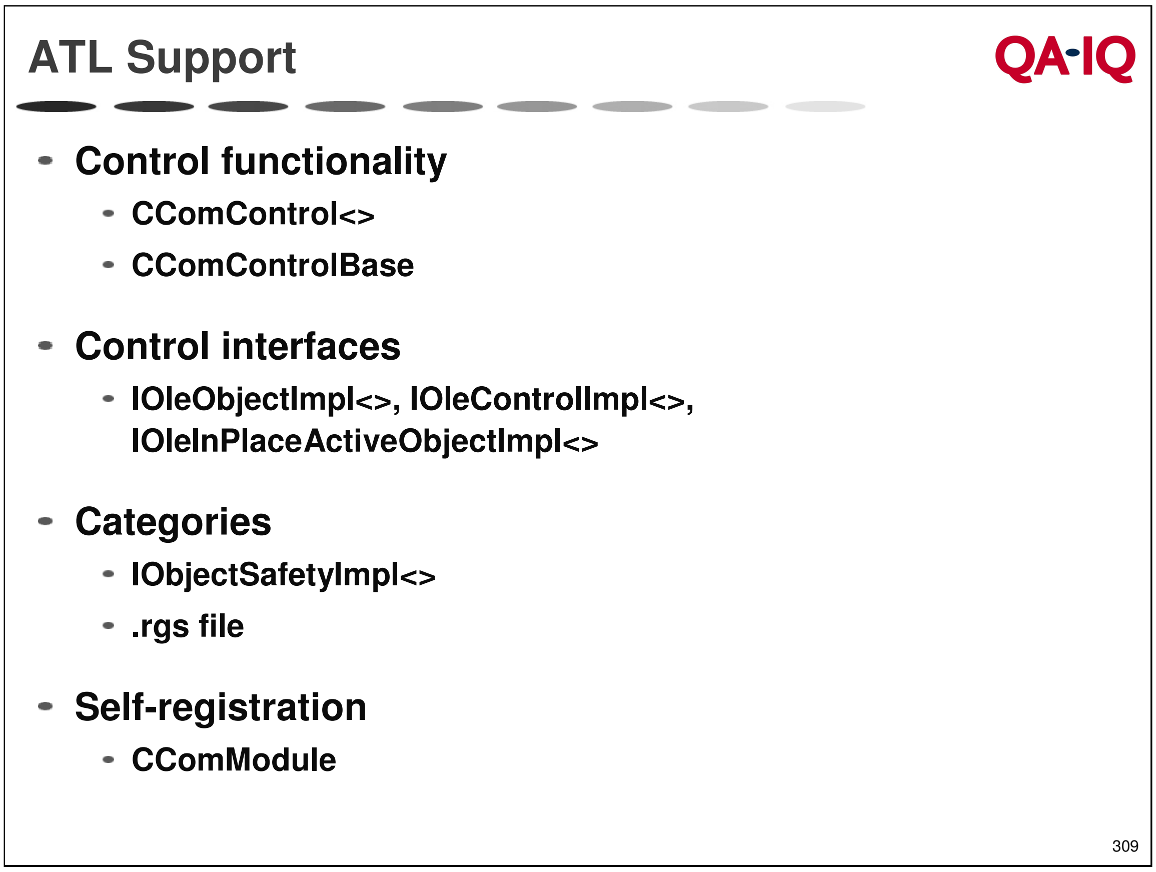
All COM servers have to be registered before they will work. EXE servers should do so every time they are run. Self-registration alleviates the need for a separate .REG file and allows the registry information to be wrapped up in the object implementation. The object can also dynamically update the path entries based on where the module is found.

For EXE servers, the OLE control extensions specify two command-line flags: `/REGSERVER` and `/UNREGSERVER`. If an EXE is launched with either of these flags it is instructed to perform the registration or deregistration and terminate. Automatic deregistration is becoming more and more important in order to ensure that registry information is clean and consistent when software is removed from the system

A COM DLL server that supports self-registration exports the functions `DllRegisterServer()` and `DllUnregisterServer()` to add/update or remove registry information for all the classes that it implements. These functions are called by the REGSVR32.EXE utility.

A container or installation program need only verify that control supports self-registration in order to call the appropriate functions exported from the DLL to get the control to (de)register itself. Ideally, we should be able to determine whether a control supports self-registration without actually loading the control, so version information, which is included in .EXEs and .DLLs as a special resource type is used. A new string *OleSelfRegister* under the *StringFileInfo* section of the `VS_VERSION_INFO` structure determines whether the control supports self-registration. The functions `GetFileVersionInfoSize()`, `GetFileVersionInfo()`, `VerQueryValue()` can be used to manipulate version information.

A control container should provide some user interface (usually something like the *File Open* dialog with a different caption and a *Register* button) with which users can locate object server implementations files.



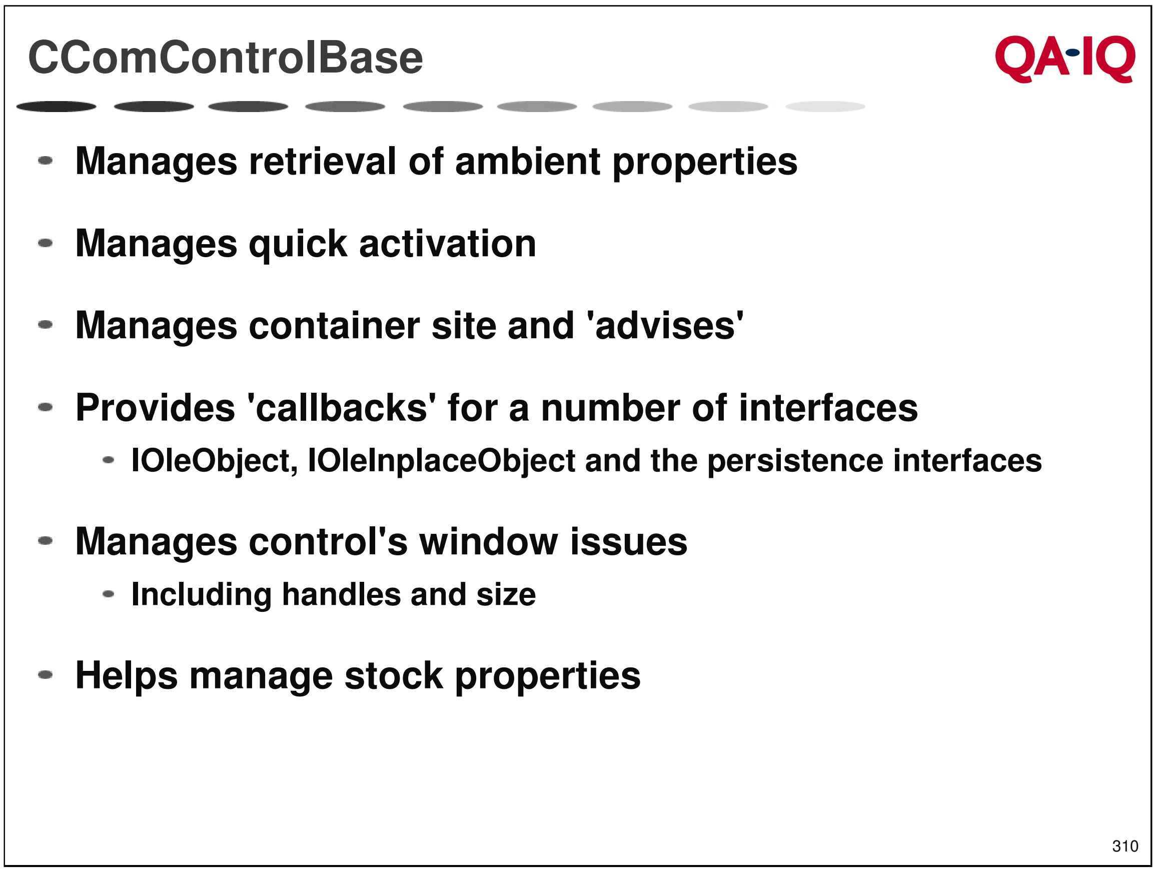
ATL Support

- **Control functionality**
  - **CComControl<>**
  - **CComControlBase**
- **Control interfaces**
  - **IOleObjectImpl<>, IOleControlImpl<>, IOleInPlaceActiveObjectImpl<>**
- **Categories**
  - **IObjectSafetyImpl<>**
  - **.rgs file**
- **Self-registration**
  - **CComModule**

309

ATL provides full support for control functionality. This is provided through `CComControlBase` and `CComControl<>`. All control objects, whether full controls or IE controls, derive from `CComControl`. Category support is provided by `IObjectSafetyImpl` or through the registry script file.

Self-registration is supported through `CComModule`.



CComControlBase

QA-IQ

- **Manages retrieval of ambient properties**
- **Manages quick activation**
- **Manages container site and 'advises'**
- **Provides 'callbacks' for a number of interfaces**
  - **IOleObject, IOleInplaceObject and the persistence interfaces**
- **Manages control's window issues**
  - **Including handles and size**
- **Helps manage stock properties**

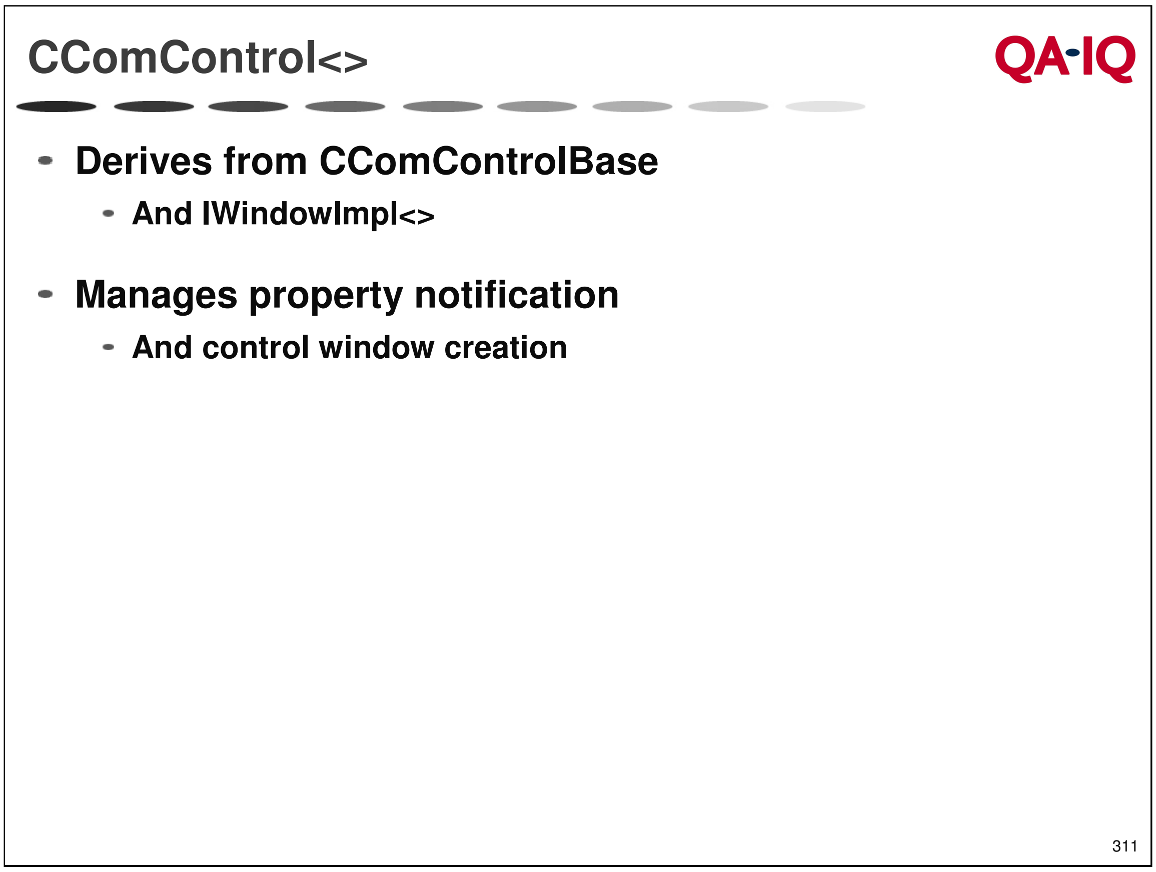
310

`CComControlBase` is the base class needed for creating Active Controls using ATL. It provides many helper methods, including a set of `GetAmbientxxx()` methods for retrieving ambient properties.

It also provides 'callback' functions for a number of interfaces, including `IOleObject`, `IOleInPlaceObject`, `IPersistStreamInit` and `IPropertyBag` as well as `IQuickActivate`.

It maintains the controls window handle (if there is one) and the current state of the control via a set of bit flags (e.g. `m_bWndLess`) that are all initialised to FALSE in the constructor.

It also contains a set of place holders for stock properties (e.g. `BSTR m_bstrText`) which are overridden whenever support for a stock property is added to the control.



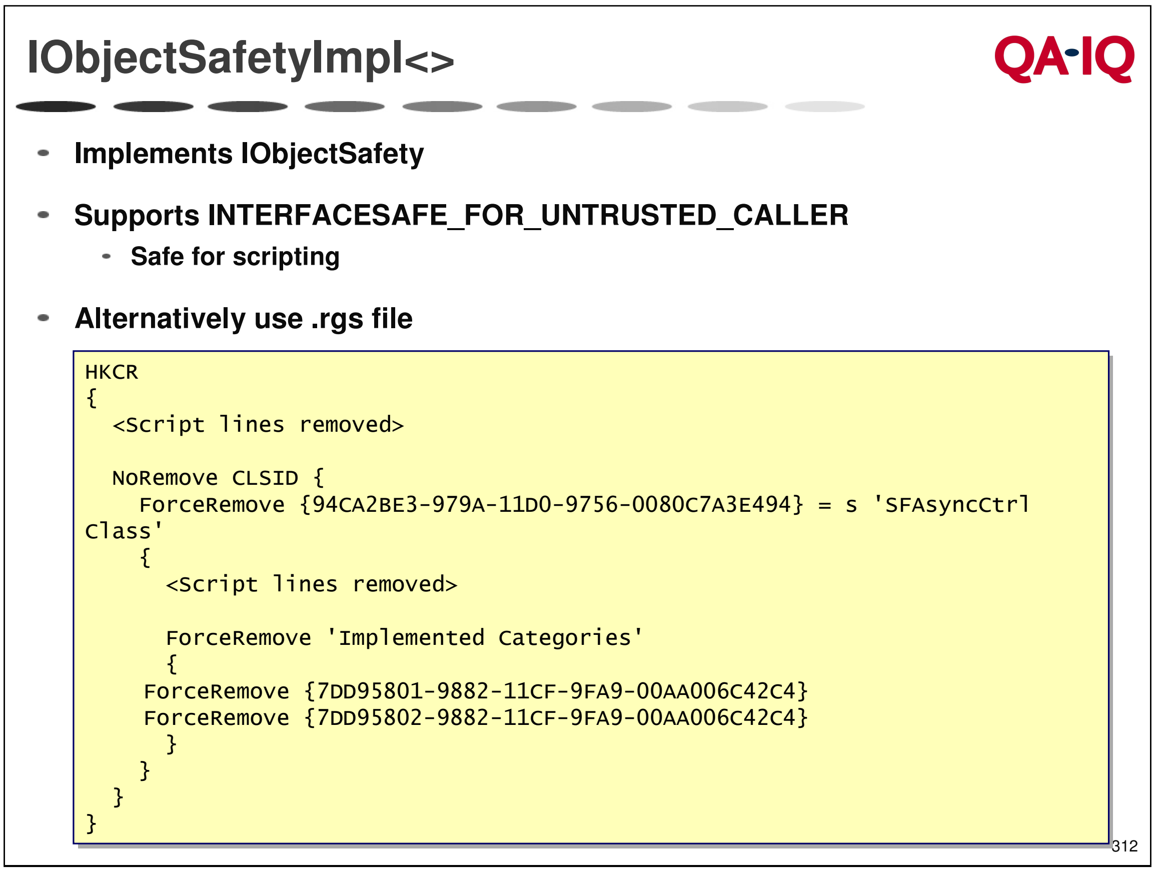
CComControl<>

- **Derives from CComControlBase**
  - **And IWindowImpl<>**
- **Manages property notification**
  - **And control window creation**

311

`CComControl<>` derives from `CComControlBase` and provides overrides for pure virtual functions in that class.

It specifically manages property notifications by supply `FireOnRequestEdit()` and `FireOnChanged()` functions and control creation (via the `CreateControlWindow()`) function.



IObjectSafetyImpl<>

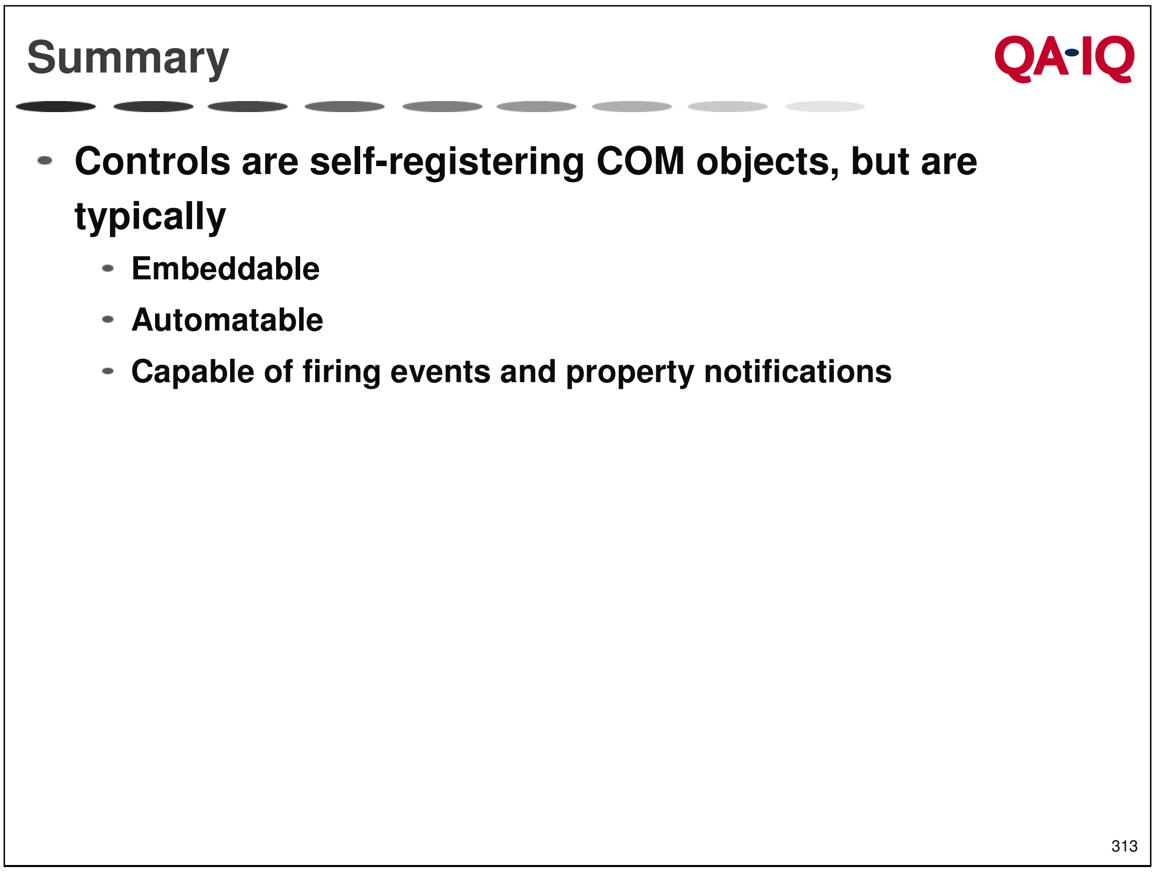
QA-IQ

- Implements IObjectSafety
- Supports INTERFACESAFE_FOR_UNTRUSTED_CALLER
  - Safe for scripting
- Alternatively use .rgs file

```
HKCR
{
  <Script lines removed>

  NoRemove CLSID {
    ForceRemove {94CA2BE3-979A-11D0-9756-0080C7A3E494} = s 'SFAsyncCtrl
Class'
    {
      <Script lines removed>

      ForceRemove 'Implemented Categories'
      {
    ForceRemove {7DD95801-9882-11CF-9FA9-00AA006C42C4}
    ForceRemove {7DD95802-9882-11CF-9FA9-00AA006C42C4}
      }
    }
  }
}
```

312

`IObjectSafetyImpl<>` provides a default implementation of `IObjectSafety`. The default implementation supports the "Safe for scripting" CATID; `INTERFACESAFE_FOR_UNTRUSTED_CALLERS` for the IDispatch interface. To support other CATIDs or interface IDs an override for `GetInterfaceSafetyOptions()` and `SetInterfaceSafetyOptions()`. Another way of supporting categories would be to use the .rgs file. The registry script file could be amended to include registry entries for the required categories.



Summary

QA-IQ

- Controls are self-registering COM objects, but are typically
  - Embeddable
  - Automatable
  - Capable of firing events and property notifications

313