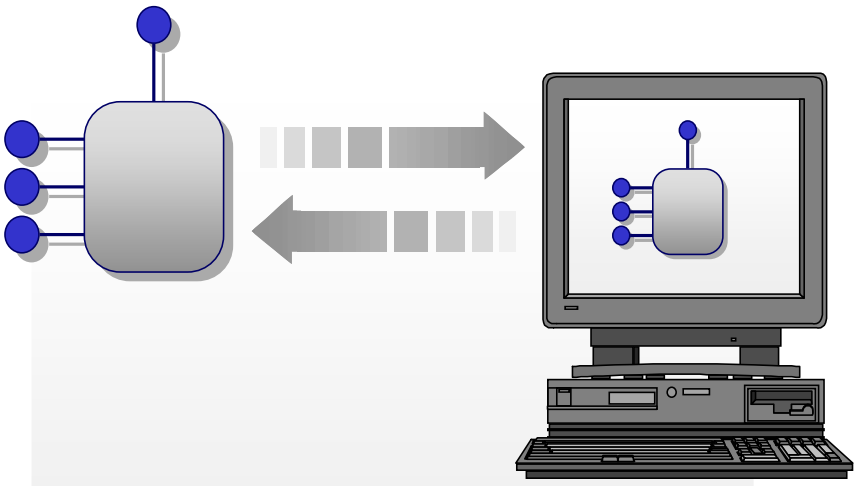
Threads and Apartments

QA-IQ



COM Programming

231

Chapter Overview

QA-IQ

- Objectives
  - Understand the purpose and nature of COM apartments
- Chapter content
  - COM threading issues
  - Apartments
  - COM threading models
  - Cross apartment marshaling
- Practical content
  - Investigating inter-thread synchronisation
- Summary

232

In this chapter we will focus on one of the most important concepts in COM - the apartment. It is through the apartment that COM addresses the potential mismatch that can occur between the concurrency requirements of clients and object implementations. COM takes the view that clients should be able to talk to objects without having to worry about such issues. Broadly speaking, it means that concurrency becomes just another implementation detail.

Certain objects may be developed without any regard to multithreaded usage, while others will have been written with the assumption of concurrent access by multiple client threads. Likewise some clients will be inherently single threaded while others will be spawning worker threads left, right and center. In this chapter we will see how the notion of an apartment addresses these issues and we will look at the different apartment types supported by COM. We will also look at the special (marshaling) steps that must be taken if you wish to pass and share interface pointers between threads.

General Threading Issues

QA-IQ

- Client code may have multiple threads
  - May create objects on many threads
  - May call object instances from many threads
- Objects may have multiple threads
  - Worker threads
  - May access object state data
  - May access global data
- Some objects are thread safe, others are not
- Some clients are thread safe, others are not
- Apartments provide the solution

233

Code is executed by *threads* of execution (a thread consists of a stack, an instruction pointer and a copy of the registers). Win32 is a *pre-emptive multi-threaded* operating system, which means that many threads can be running at the same time (on a multi-processor machine - on a single processor machine, only one thread can ever be executing code at a specific point in time).

Windows provides a thread scheduler that grants threads access to the processor(s) depending on a thread's priority. Regardless of the number of processors in a machine, this means that threads are continually being stopped and started to perform small pieces of work.

So why does Windows have multi-threading? Simply stated, when used correctly multi-threading can improve responsiveness for the user and performance for a system.

What does this all mean for the COM programmer?

Some client software will be utilising multiple threads, and may be either creating objects on different threads, or calling objects from different threads, or both. Some objects may be using multiple threads of execution to perform their work, including calling back to client sinks.

The problem here is that some objects and clients will be written using unsafe (for threading) code constructs, such as global variables or using non-atomic operations for managing reference counts.

Whether a COM object (or client) is aware of threads should be (and is) regarded as an implementation detail. COM is a nurturing component technology that will therefore attempt to protect COM objects from problems that can occur with threading.

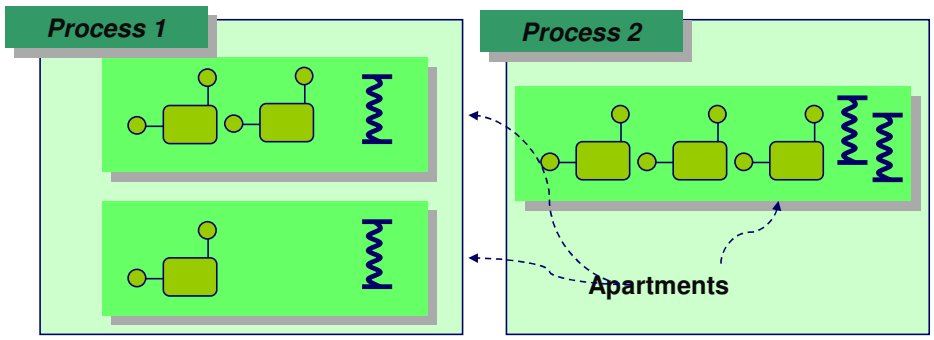
How does it do this (with a little bit of help from us programmers!)?

It provides *apartments* in which the COM objects reside and will use these apartments to provide the necessary synchronisation protection where possible.

Apartments

QA-IQ

- An "apartment" groups objects with the same concurrency constraints together
- COM processes have 1 or more apartments
- Objects live in apartments
  - An apartment can contain many objects
  - Every object is associated with exactly one apartment
  - Objects are only called by thread(s) in the object's apartment



234

An apartment is used by COM to group objects with similar concurrency constraints together. An apartment is neither a process or a thread. Rather a process using COM can contain one or more apartments and each apartment (depending upon its type) can contain one or more threads.

The key point to grasp when attempting to understand apartments, is that an object *can only ever be accessed* by a thread in its owning apartment.

As we shall see COM defines three different types of apartment. One type supports only a single thread within the apartment, while the other type supports multiple threads within the same apartment. Rather sensibly these apartment types are referred to as the "Single- Threaded Apartment" or STA, and the "Multi-Threaded Apartment" or MTA.

The third type, that appeared with Windows 2000, is called the "Thread-Neutral Apartment" or TNA and it can be thought of as a variant of the MTA, in that multiple threads can be used to service an object.

Apartment Types

QA-IQ

- **Single-Threaded Apartments (STA)**
  - **Only one thread can live in the apartment**
  - **0+ STAs per process**
  - **Method calls serialised via Windows message queue**
  - **The same thread always calls the object**
    - Object need not be thread safe
    - Object has thread affinity
- **Multi-Threaded Apartments (MTA)**
  - **Many threads can reside within the apartment**
  - **0 or 1 MTA per process**
  - **Any thread can call object at any time**
    - Object must be thread safe
    - No thread affinity
- **Thread Neutral Apartment (TNA) (Windows 2000 and above)**
  - **An apartment into which any thread can enter**

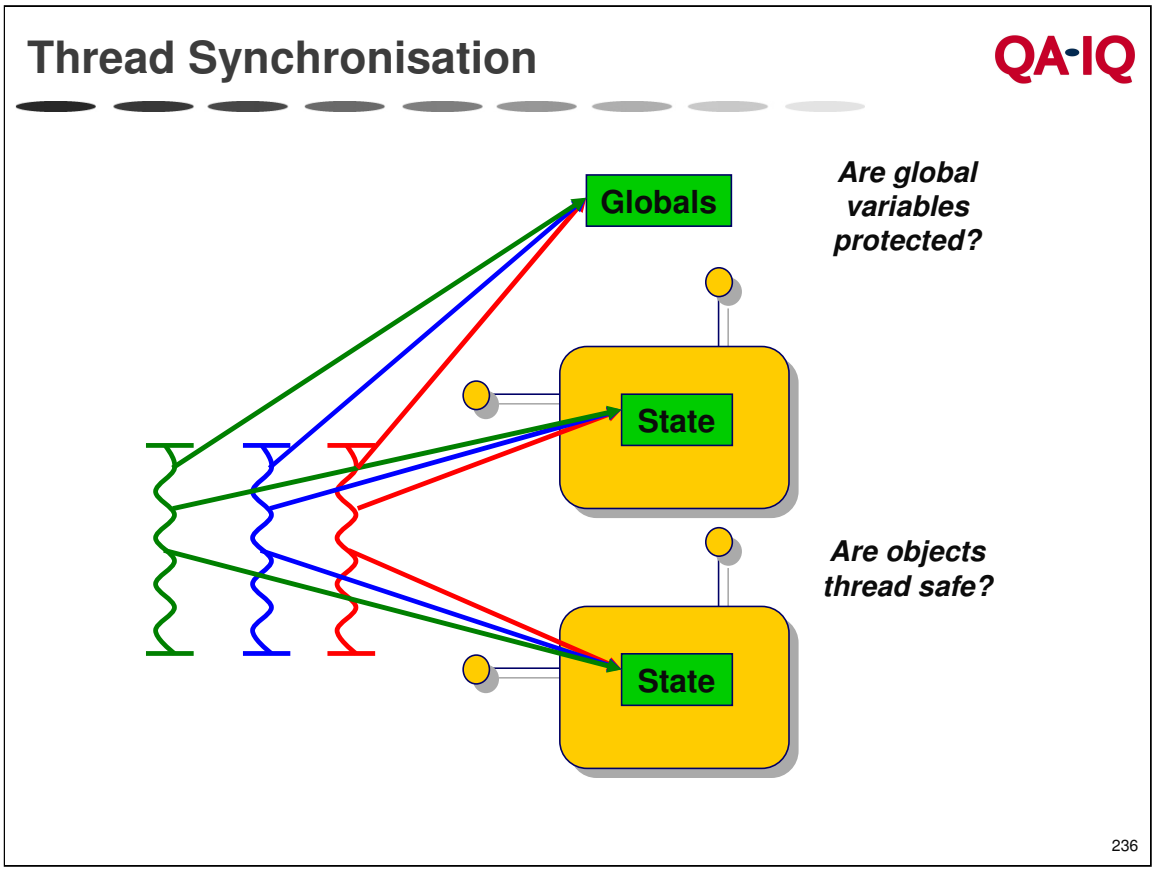
235

The STA is so called because only one thread can live in the apartment. A process using COM may contain zero, one or many STAs. The advantage of the STA, certainly from the object developer's point of view, is that COM will automatically serialise all method calls to all objects within the apartment. This means that the object implementation need not worry about concurrent method access and as a result need not protect and serialise access to instance data. Access to global data and static functions must be serialised by the component developer, however, as potentially two threads each residing in its own STA may call into these concurrently. The serialisation provided by the STA occurs using a standard Windows message queue. The downside of this approach is performance - method calls are serialised to *all* object instances living in the apartment.

Another important fact is that while it is true to say that an STA only ever contains a single thread, it is the *same thread* each time that accesses a given object. The object is said to have *thread affinity*. This is useful when writing components that create windows, such as ActiveX controls, which require thread affinity for delivery of messages.

A process can contain at most one MTA. As its name would suggest an MTA can contain multiple threads. COM will not serialise access to object instances contained within the MTA. As a result object implementations are somewhat more complex, because access to an object's instance data (in addition to static functions and global data) must be serialised by the object itself (for example using critical sections or mutexes). The big advantage of the MTA approach is performance. DCOM servers supporting many clients can reap significant benefits from being "free threaded". This is another commonly used term for an MTA-based server.

The TNA arrived with Windows 2000, and was designed to improve scalability by reducing the number of thread switches that are involved when crossing apartment boundaries. The TNA has no threads of its own, but lets threads from other apartments in to touch objects inside its apartment. Only minor changes have to be made to the calling thread to adjust its context information whilst it executes inside the TNA.



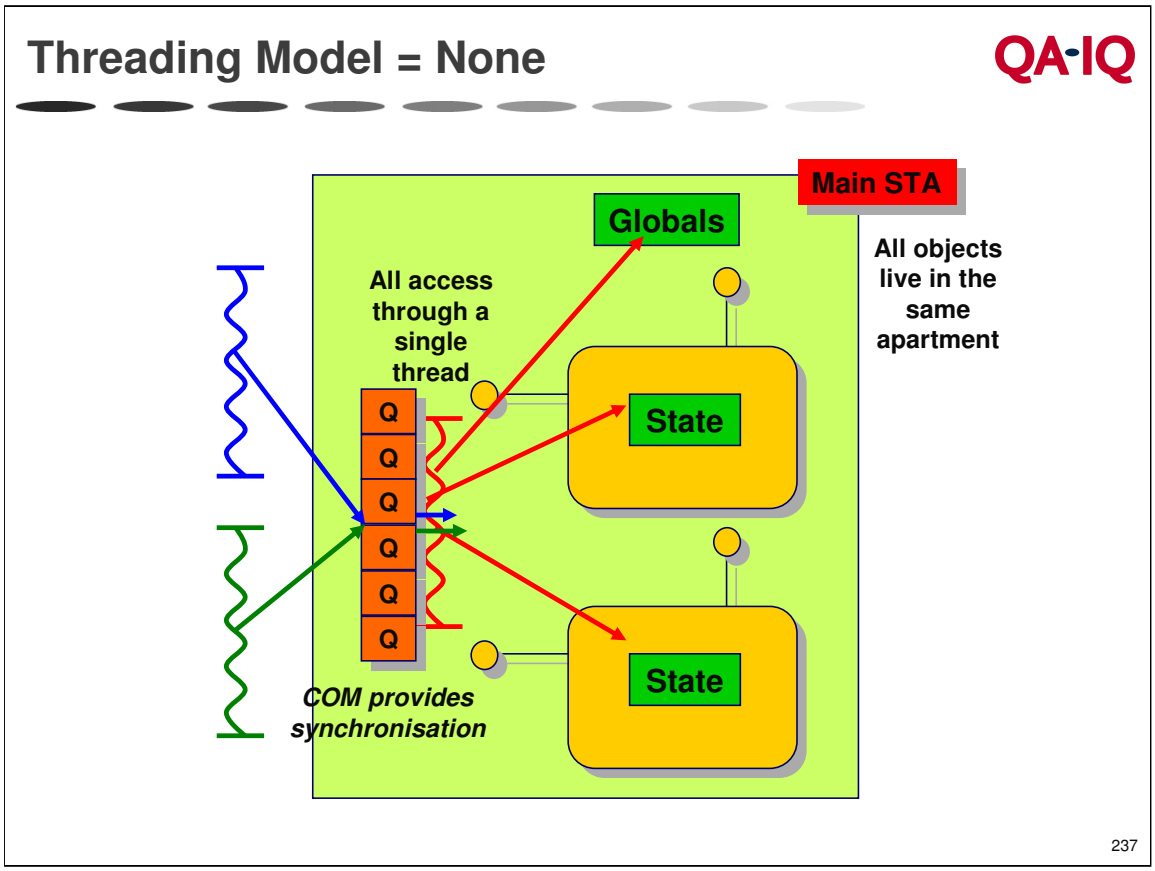
To understand why we need apartments we must first consider the issues affecting COM objects and data access. We must understand the interaction between multiple threads (code) and state (data).

Every object will have internal state information which can be accessed simultaneously (potentially) by several threads. Threads will in general modify this state information. Code in the threads is said to be thread safe if this access is achieved without one thread corrupting the changes being made by any other thread.

Thread safety can be achieved using a queuing mechanism such as Mutexes, Semaphores and Critical Sections. You can add these mechanisms to the thread code to ensure thread safety. Writing code to be thread safe is tricky but leads to very efficient access.

Alternatively you can get COM to provide the queuing mechanism. COM will use a generic queuing mechanism using the windows message queue. This leads to much less efficient access than if you code it yourself. However, the advantage with this approach is that is easy - just leave it to COM.

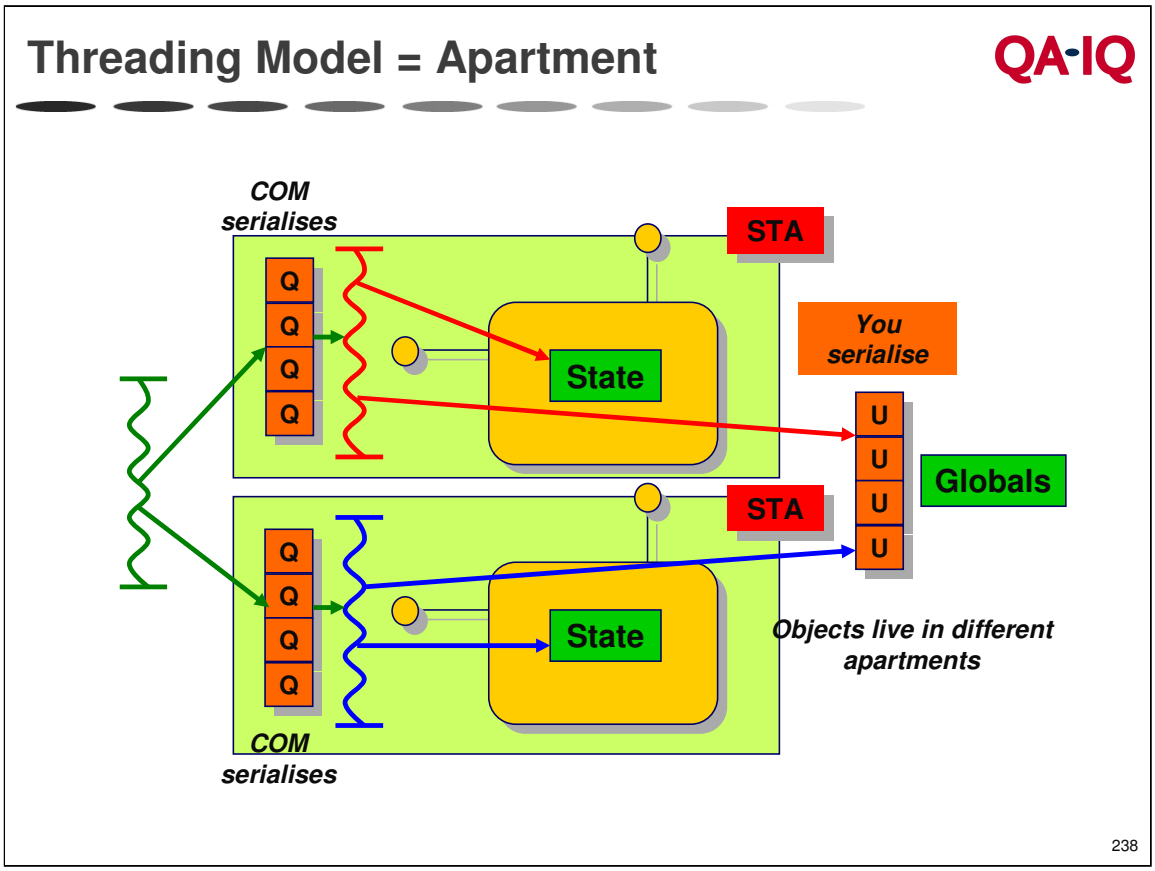
The various scenarios are discussed in the next set of slides under the heading of threading models.



The threaded model of **none** in COM is effectively a degenerate case of the Apartment model. A single threaded DLL is stating it can only live in one STA (the so called 'main' STA which is defined to be the first STA initialised in a process). A single threaded DLL is stating that neither its objects nor its class factory are thread safe.

In this case all calls to all object instances (including calls to the class factory) will be serialised through the message queue.

These days you should avoid using this model, certainly in distributed applications supporting multiple clients, as performance would be intolerable. For example, if you had 10,000 instances of the object in existence, all 10,000 instances would accessed via a single thread. This is not the sort of scalability that we are looking for!

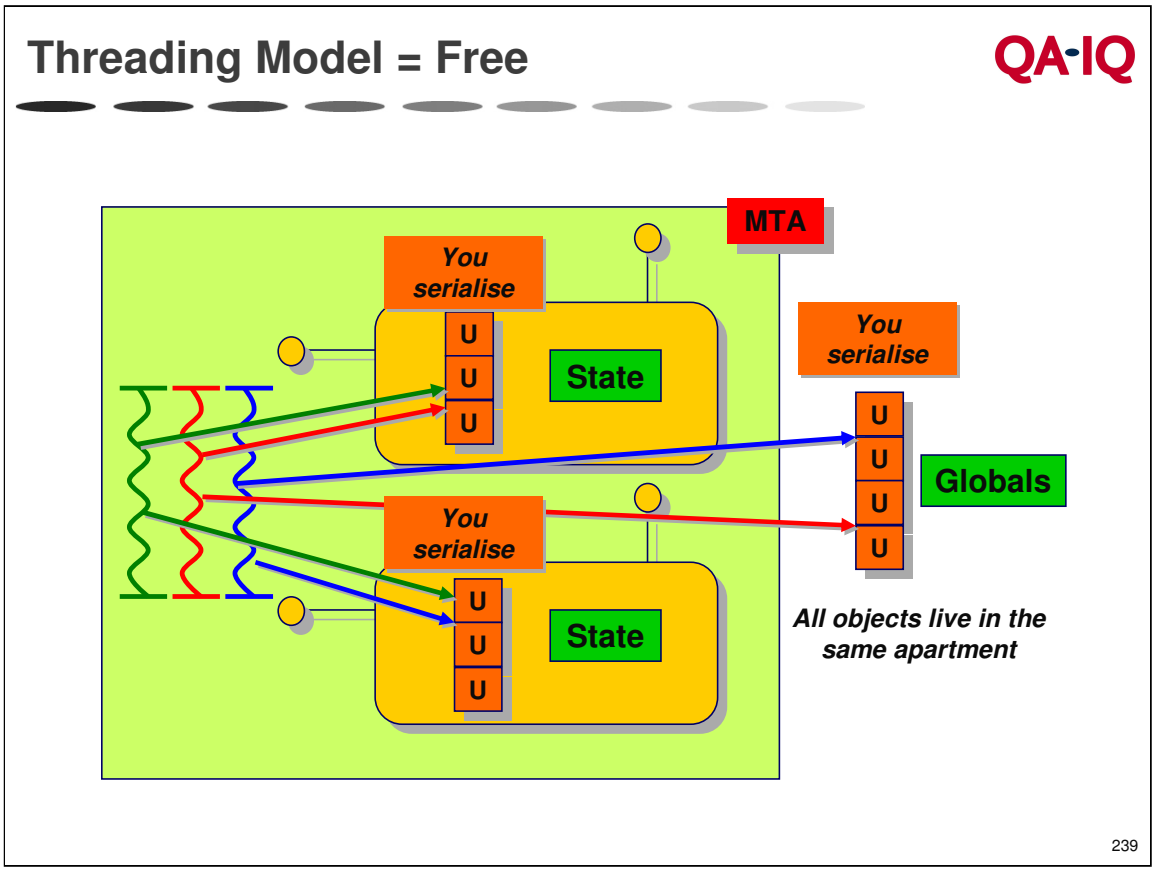


The **apartment** threading model is where COM provides the necessary synchronisation between method calls using a windows message queue. In this model each object is serviced by a dedicated thread (thread affinity) and the object, the thread and the message queue all live in the same apartment. Since there can only be one thread in this type of apartment it is often called a Single Threaded Apartment (STA).

A client specifies that a thread should use apartment threading by calling `CoInitializeEx( 0, COINIT_APARTMENTTHREADED ).`

If another thread wishes to access the object it must do so indirectly via a proxy object which will forward the method call to the queue of the STA. We will investigate proxies and inter apartment communication later in the chapter.

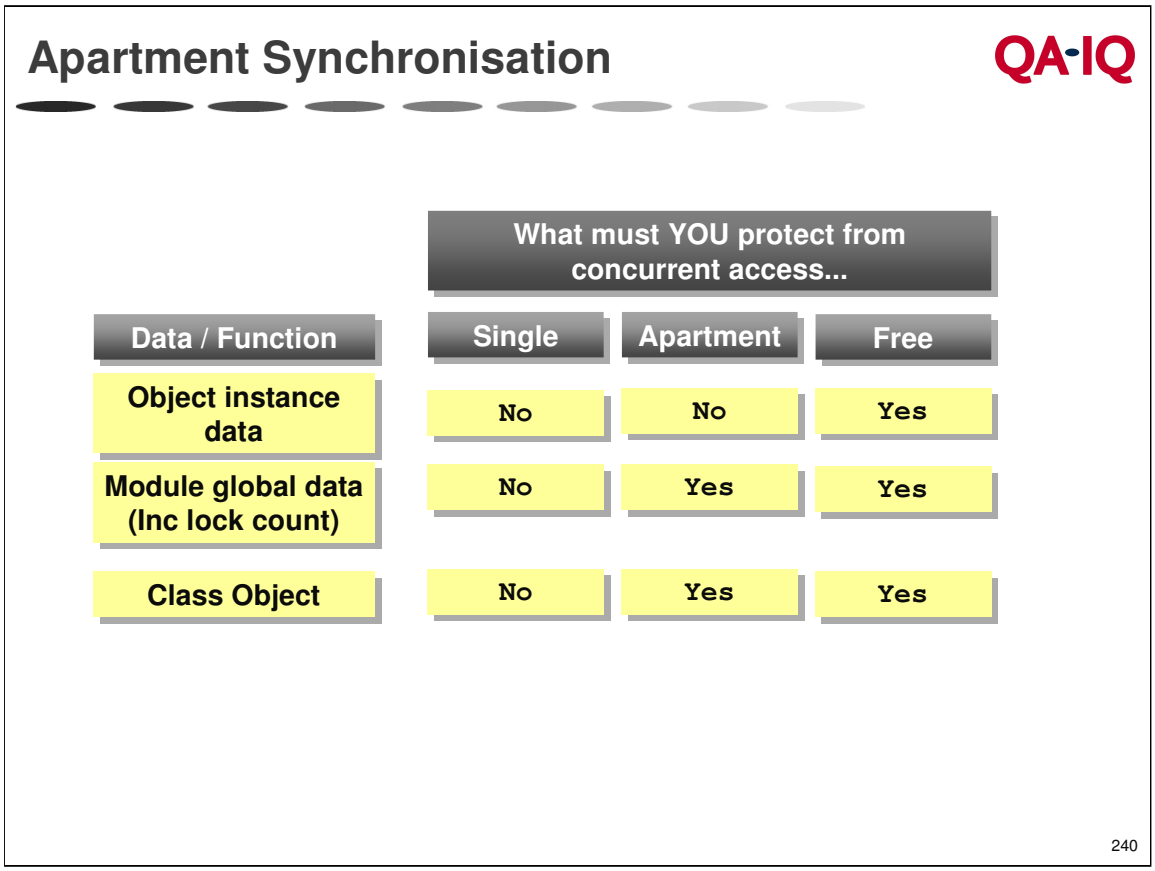
Note that the STA does not protect access to global data because this data lies outside the apartment. Be careful when writing class factories that update global object counts - you must make them thread safe yourself!



Free threaded clients can make COM method calls from multiple threads. Each 'free' thread lives in the Multi-threaded apartment (MTA). There is (at most) one MTA per process. A client specifies that a thread should use free threading by calling `CoInitializeEx( 0, COINIT_MULTITHREADED ).`

Calls into a free threaded object are not synchronised by COM; the object is responsible for providing it's own synchronisation where necessary. Also the object may be called by any arbitrary thread in the MTA and so must make no assumptions about which thread it will be called on.

Interface pointers passed between threads in the MTA do not need to be marshaled. This is because there is no apartment boundary being crossed. However, if an MTA thread passes an interface pointer to an STA then the pointer must be marshaled as it is going across an apartment boundary.



This slide summarises the rules presented in the previous slides.

Remember, as a COM *Server* author, it is your responsibility to select the threading model for your components, and then to code according to the rules of that threading model. The table above tells you what you need to protect with your own synchronisation code (anything with a *Yes* in a box needs synchronising).

There are also a couple of extra rules that you should consider when coding with apartments.

Firstly, a thread that calls `CoInitializeEx(0,COINIT_APARTMENTTHREADED)` can never block indefinitely, using calls such as `WaitForSingleObject()`. If you block using a thread in this way, you can deadlock your application, as no calls on COM objects in that apartment can then be processed while the thread is blocked.

Next, when coding a COM object that will reside in the MTA, you should avoid holding synchronisation locks (for example, on a mutex), if you call out of the apartment or return from a method call. Again, it is relatively easy to deadlock the application, as you never know which thread will be used to service a COM call in the MTA, and a call back in to the same object may be executed with a different thread.

It is worth remembering that COM's apartments do simplify interoperation between clients and objects that have different threading requirements, but they do not alleviate all of the problems that occur with multi-threading programming.

Take care, read widely on the subject, experiment and have fun!

Entering an Apartment

QA-IQ

- A thread calls CoInitializeEx() to enter an apartment
  - COINIT_APARTMENTTHREADED enters a new STA
  - COINIT_MULTITHREADED enters the lone MTA
  - CoInitialize(0) is equivalent to
    - CoInitializeEx(0, COINIT_APARTMENTTHREADED)
- Every thread using COM must call CoInitializeEx()
- Local-servers call CoInitializeEx() directly
- But what about in-proc servers?
  - The client process will have already called CoInitializeEx()

241

Every thread that wishes to use COM must call `CoInitializeEx()` (or its higher level `CoInitialize()` counterpart). The main reason for calling this routine is to allow the thread to enter an apartment. With `CoInitializeEx()` you can specify via its second parameter which type of apartment the thread should enter. Specifying `COINIT_MULTITHREADED` instructs the thread to enter the process's one and only MTA, while `COINIT_APARTMENTTHREADED` instructs the thread to enter a new STA.

Note that each thread calling `CoInitializeEx()` supplying `COINIT_APARTMENTTHREADED` results in the creation of a new STA - after all, they are by definition SINGLE threaded apartments.

Local EXE-based servers as we have seen call `CoInitializeEx()` as part of their start up sequence and therefore are in total control of the type of apartment being used. In-proc (DLL-based) servers on the other hand have a different problem. By the time in-proc objects are loaded their associated client process has already called `CoInitializeEx()`.

Consequently, there needs to be an another way for in-proc servers to inform COM of their concurrency constraints. Where do in-proc COM objects provide information for the SCM to read? The registry, as we shall see on the next page.

In-proc Servers

QA-IQ

- **Specify threading model through a registry key**
- **[HKCR\CLSID\<clsid>\InprocServer32]**
  - **ThreadingModel value can be one of**
    - Apartment
      Object will live in (any) STA and COM will synchronise calls
    - Free
      Object will live in MTA, and object must provide synchronisation
    - Neutral
      Object will live in TNA, and object must provide synchronisation
    - Both
      Object can live in any type of apartment
      Must follow rules for ALL apartment types
- **Absence of ThreadingModel value implies**
  - **All instances of component will be placed into the 'Main' STA**

242

In-process (DLL based) objects and out-of-process (EXE based) objects specify their threading model in different ways. DLL based objects provide a registry key under their CLSID. This tells COM what threading model the DLL server wants to run under. COM will handle any mismatch between the client threading model and the model the object server wants to use. COM currently defines 3 threading ‘models’.

A COM object marked with no threading information in the registry will have all of its instances stored in a single STA, ensuring that there is no concurrent access to any of its code. This is the original COM model for early versions of Windows.

With Apartment model threading, each thread is contained within its own Single Threaded Apartment (STA), and COM provides synchronisation. Windows NT 4.0 and Windows 95 with DCOM support, provide support for ‘Free threading’. Again, multiple threads can make COM calls. However Free threads all live in the same MultiThreaded Apartment (MTA). COM provides no synchronisation at all.

At first glance it may not be obvious why "Free" exists in addition to "Both". "Free" will force an object to be instantiated into an MTA, while "Both" will allow the object to be instantiated into the apartment of the client thread - this may be an STA, MTA or TNA. "Free" is particularly useful for objects that create worker threads. If these worker threads need access to the object, then unless the worker threads reside in the same apartment as the object then each worker thread will incur the overhead of a inter-apartment proxy. Given that the worker threads will exist within the process’s MTA, it is beneficial to ensure the object also resides in this apartment - hence ThreadingModel "Free". This benefit will only exist while the number of worker thread / object accesses exceeds the number of client objects accesses.

Setting the ThreadingModel to "Neutral" means that the object is placed in the process' single TNA. The TNA has no thread of its own, but threads from other apartments can concurrently enter the TNA to talk to objects. This offers a performance enhancement over the MTA/STA communication, which requires thread switches, but as stated, COM has to perform a context switch on the thread as it enters and leaves the TNA.

Cross Apartment Access

QA-IQ

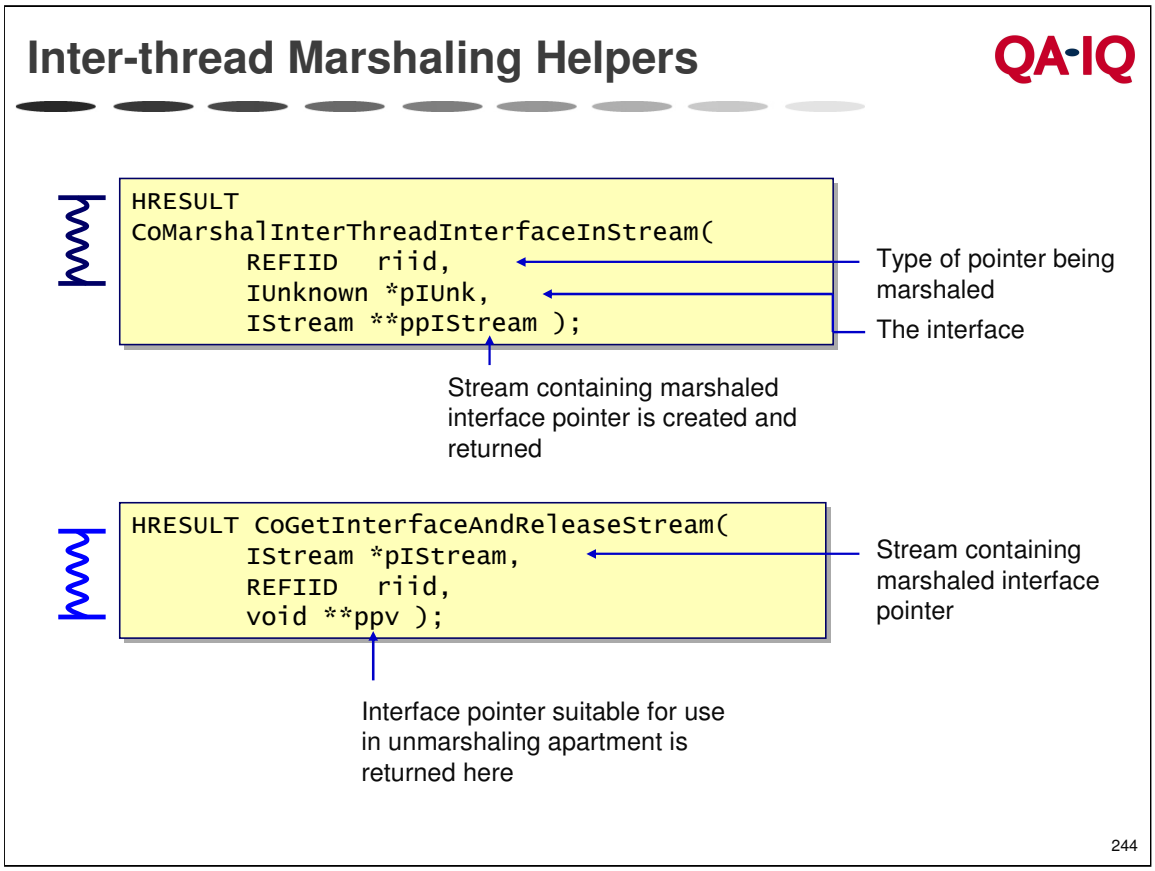
- If creating thread is in a compatible apartment, caller gets direct pointer
  - Otherwise caller receives a proxy
- Cross apartment access to an object is only possible via a proxy
- Interface pointers can not be passed directly across apartment boundaries
  - Interface pointer must be marshaled
  - Use CoMarshalInterThreadInterfaceInStream() in source thread
  - Use CoGetInterfaceAndReleaseStream() in destination thread
- Threads in the MTA can share interface pointers
  - Each thread belongs to the single MTA
  - No apartment boundaries are being crossed

243

Having gone to the trouble of ensuring that objects are always instantiated into an apartment of the correct type based on the object's concurrency constraints, things would go horribly wrong if COM allowed you to pass an interface pointer directly from one STA-based thread to another (say via a global variable), or from an MTA-based thread to an STA-based thread or vice-versa. This would violate the rules of COM, as it would now be quite possible for more than one thread to call an STA-based object - something that the object would not be expecting! In order to ensure that chaos does not prevail, COM mandates that interface pointers can not be passed directly from one apartment to another but must be *marshaled* en-route.

COM provides low level APIs (`CoMarshalInterface()` and `CoUnmarshalInterface()`) to handle this task. However, these API's tend only to be used internally by COM when interface pointers are implicitly marshaled, say as a result of an activation request or as a result of passing an interface pointer across an apartment boundary as a method parameter.

As an application developer, the main time that you will need to concern yourself about explicitly marshaling an interface pointer is when you want to share an interface pointer between threads (assuming for the moment that these are not MTA based threads). To achieve this, COM provides slightly higher level (and somewhat longer named!) APIs, called `CoMarshalInterThreadInterfaceInStream()` and `CoGetInterfaceAndReleaseStream()`, designed specifically for the task of passing interface pointers between threads in the same process. These APIs wrap up the creation and release of an `IStream` object in which the marshaled interface pointer is placed, followed by a call to `Co(Un)MarshalInterface`.

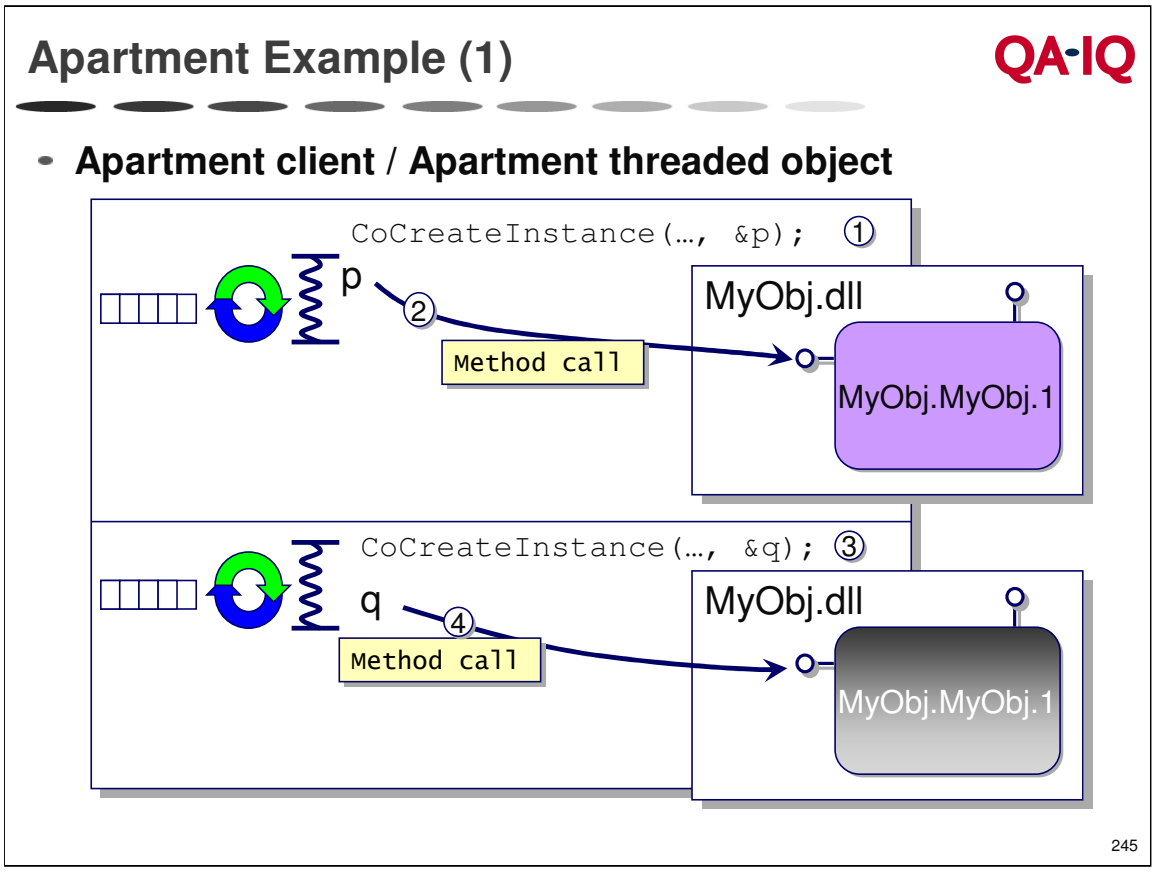


The `CoMarshalInterThreadInterfaceInStream()` API is designed to allow you to easily marshal an interface pointer from one thread to another in the same process. The stream pointer returned (which points to the stream object containing the marshaled interface pointer) is guaranteed to work correctly regardless of which thread accesses it (this is contrary to most 'standard' interface pointers, which of course require marshaling!).

`CoMarshalInterThreadInterfaceInStream()` is a light weight wrapper function around `CoMarshalInterface()`. You could imagine that the pseudo code for this function would look something like:

```
HRESULT CoMarshalInterThreadInterfaceInStream( REFIID riid,
            IUnknown *pIItf,
            IStream **ppIStream )
{
 // Create a stream object stored in memory
 // The stream interface pointer will be released when
CoGetInterfaceAndReleaseStream
 // is called.
 hr = CreateStreamOnHGlobal( 0, TRUE, ppIStream );
 if ( SUCCEEDED( hr ) )
 {
 // Now actually marshal the interface pointer into the stream
 hr = CoMarshalInterface( *ppIStream,
         riid,
         pIItf,
         MSHCTX_INPROC,
         0,
         MSHLFLAGS_NORMAL );
 }
 return hr;
}
```

`CoGetInterfaceAndReleaseStream()` simply calls `CoUnmarshalInterface()` and then releases the stream object. Bear in mind that due to the `MSHLFLAGS_NORMAL` flag passed to `CoMarshalInterface()`, the unmarshal will only be able to occur *once*.



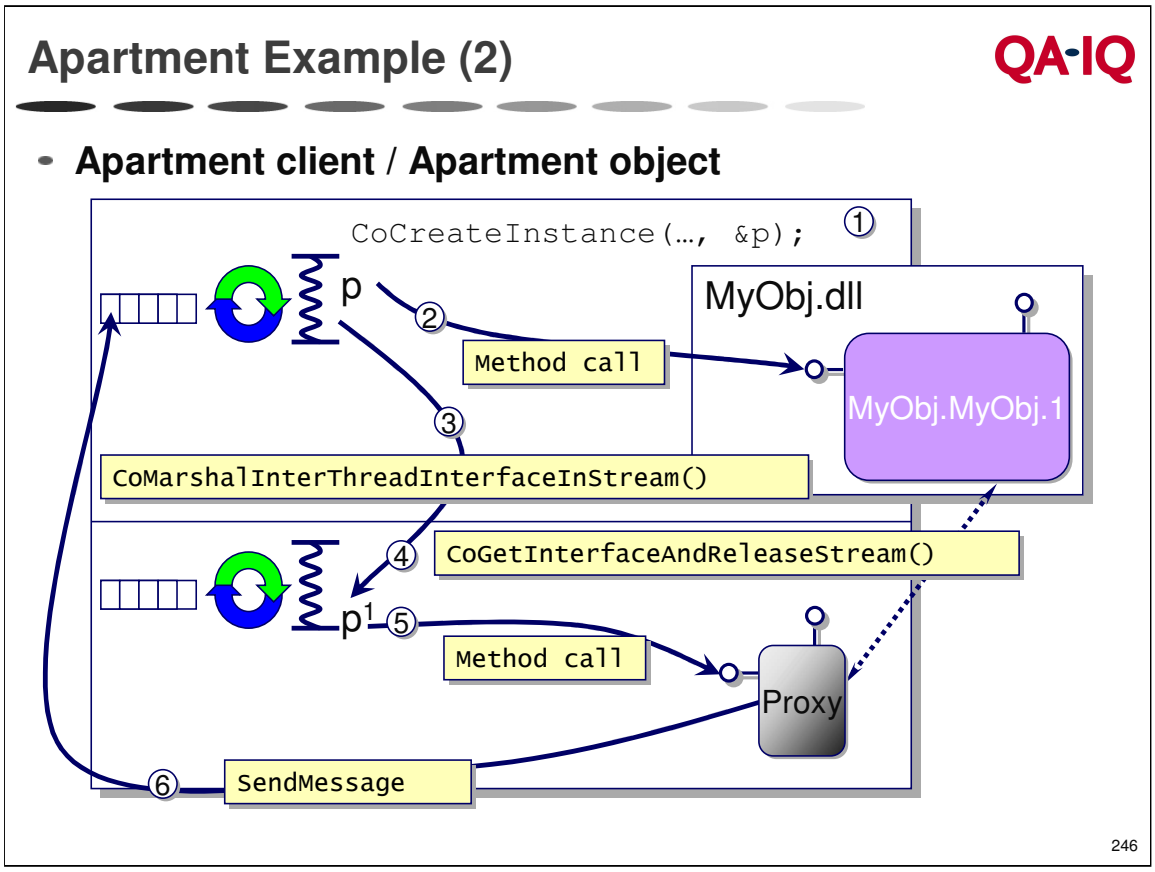
This diagram illustrates the sequence of events that occur when an Apartment model client has two apartments each of which creates an Apartment-threaded DLL based object.

The client's first thread calls `CoCreateInstance()`. COM sees that the client thread is Apartment model and that the object has specified in the registry that it is Apartment model. Because the object and the client have specified that same threading model, COM knows it can set up a direct connection between the two, so it loads the DLL, creates the object and gives the client an interface pointer, p, to the object. The client then makes method calls on the object using the interface pointer, p.

The client's second thread has also called `CoInitializeEx( 0, COINIT_APARTMENTTHREADED )` and has thus created a second STA. This thread also calls `CoCreateInstance()` to create an instance of the same class of object. COM sees that the object DLL is marked as Apartment, and so creates the second object in the second apartment and gives the client's second thread an interface pointer, q, to the object. The client then makes method calls on the object using the interface pointer, q.

Note that both threads can make method calls on their objects simultaneously. This is because each object instance is only being called from one thread. Consequently, the object itself does not need to be thread-safe*, although the class factory for the object *will* need to be as several creation requests may occur simultaneously.

* What this means, in effect, is that items such as the reference count for an individual object will not need to be thread-safe. However, if the code inside the object adjusts a global variable (such as the overall object count within a DLL server), then this code must be made thread-safe.



This diagram illustrates the sequence of events that occur when an Apartment model client (one that has called `CoInitializeEx(0,COINIT_APARTMENTTHREADED)`, creates an Apartment model DLL based object and passes the resulting interface pointer to a second apartment.

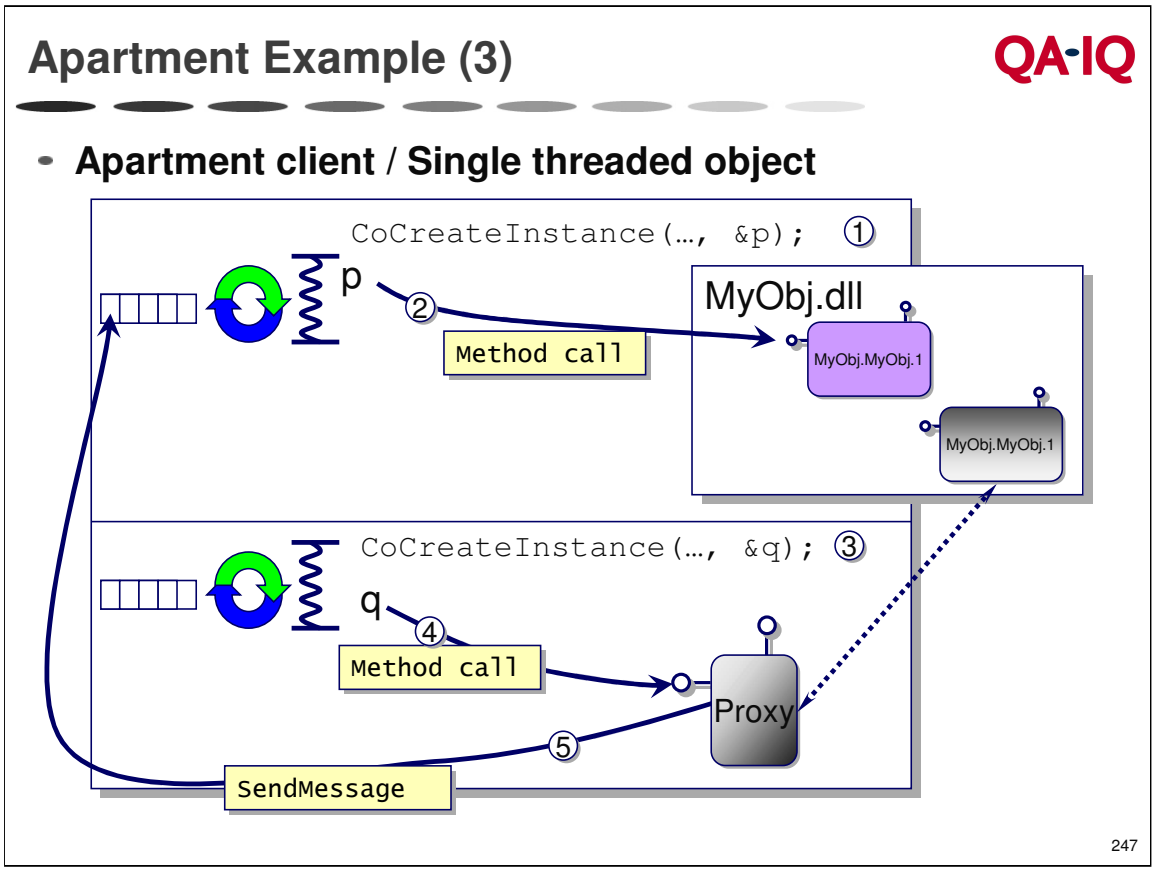
First the client calls `CoCreateInstance()`. COM sees that the client thread is Apartment model and that the object has specified in the registry that it is Apartment model. Because the object and the client have specified that same threading model, COM knows it can set up a direct connection between the two, so it loads the DLL, creates the object and gives the client an interface pointer, p, to the object. The client then makes method calls on the object using the interface pointer, p.

The client then passes the interface pointer, p, to a second thread, that has also called `CoInitializeEx(0,COINIT_APARTMENTTHREADED)` and has thus created a second STA. In order to pass the interface pointer safely, the source thread must call `CoMarshalInterThreadInterfaceInStream()`.

The reference is retrieved using the `CoGetInterfaceAndReleaseStream()` API. This API creates a proxy and gives the destination apartment a pointer to the proxy. The proxy has a connection back to the message queue of the thread which created the original object.

When the second thread makes a method call using p1, the call is actually made on the proxy, which sends* a windows message to the creating threads message queue. When the creating thread next services its message queue the method call will be forwarded to the actual object.

*In order to avoid deadlock situations, when a thread in an STA calls out to another STA apartment, the thread does not call the `SendMessage()` function itself. Instead, a separate channel thread sends the message, which frees the STA thread up to respond to any incoming requests into its own apartment.

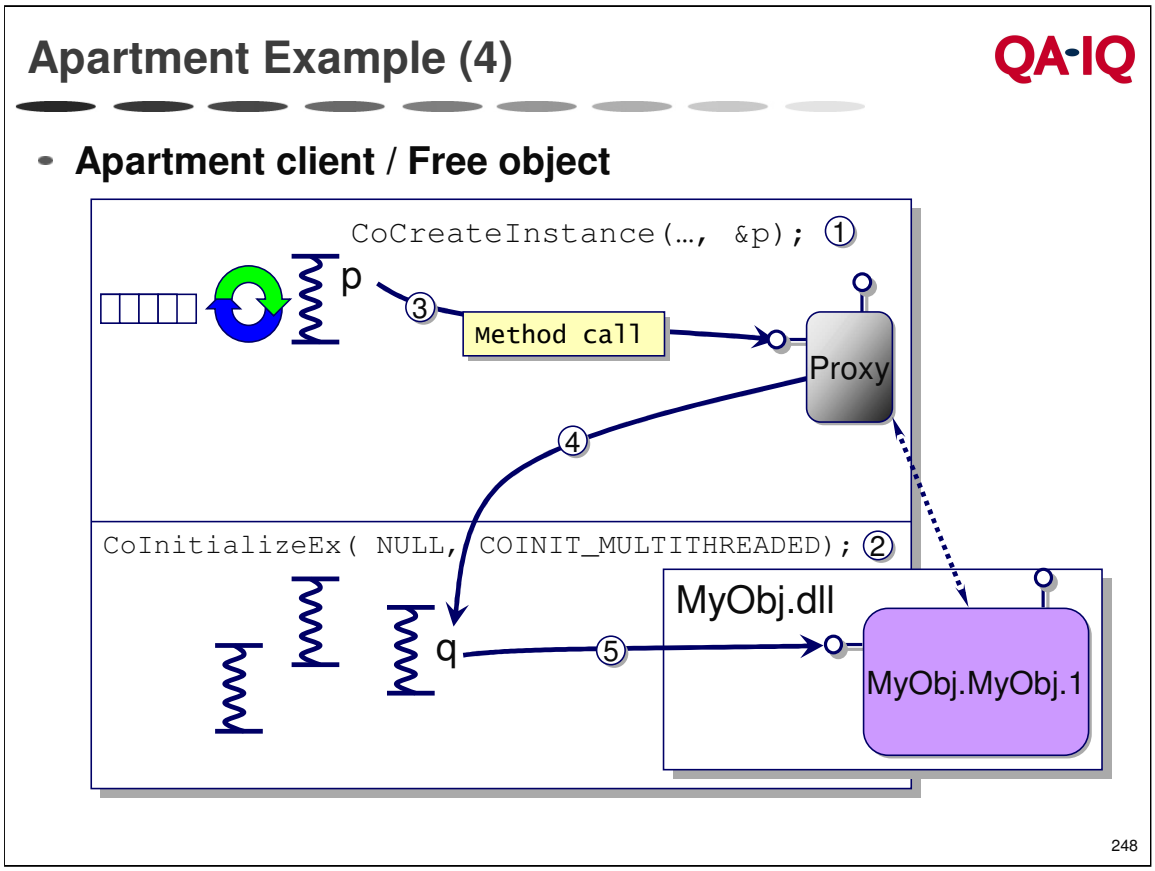


This diagram illustrates the sequence of events that occur when an Apartment model client has two apartments each of which creates a Single-threaded DLL based object. An Single model DLL is stating that it cannot cope with being called on different threads at all, so all object instances must live on the same thread.

First the client calls `CoCreateInstance()`. COM sees that the client thread is Apartment model and that the object has no threading model. COM loads the DLL into the apartment, creates the object and returns an interface pointer to the client. The client then makes method calls using the interface pointer p.

The client's second thread has also called `CoInitializeEx(0, COINIT_APARTMENTTHREADED)`, and has thus created a second STA, also calls `CoCreateInstance()` to create an instance of the same class of object. COM sees that the object DLL has no threading model AND that it is already loaded into the first STA. It forwards the creation request to the first STA and the second STA gets back an interface pointer, q, to a proxy. This proxy is a proxy to the second object instance and has a connection to the message queue of the first STA. Method calls made through q are sent by the proxy to the message queue of the first STA.

This means that ALL method calls on ALL object instances of MyObj.MyObj.1 are serialised.

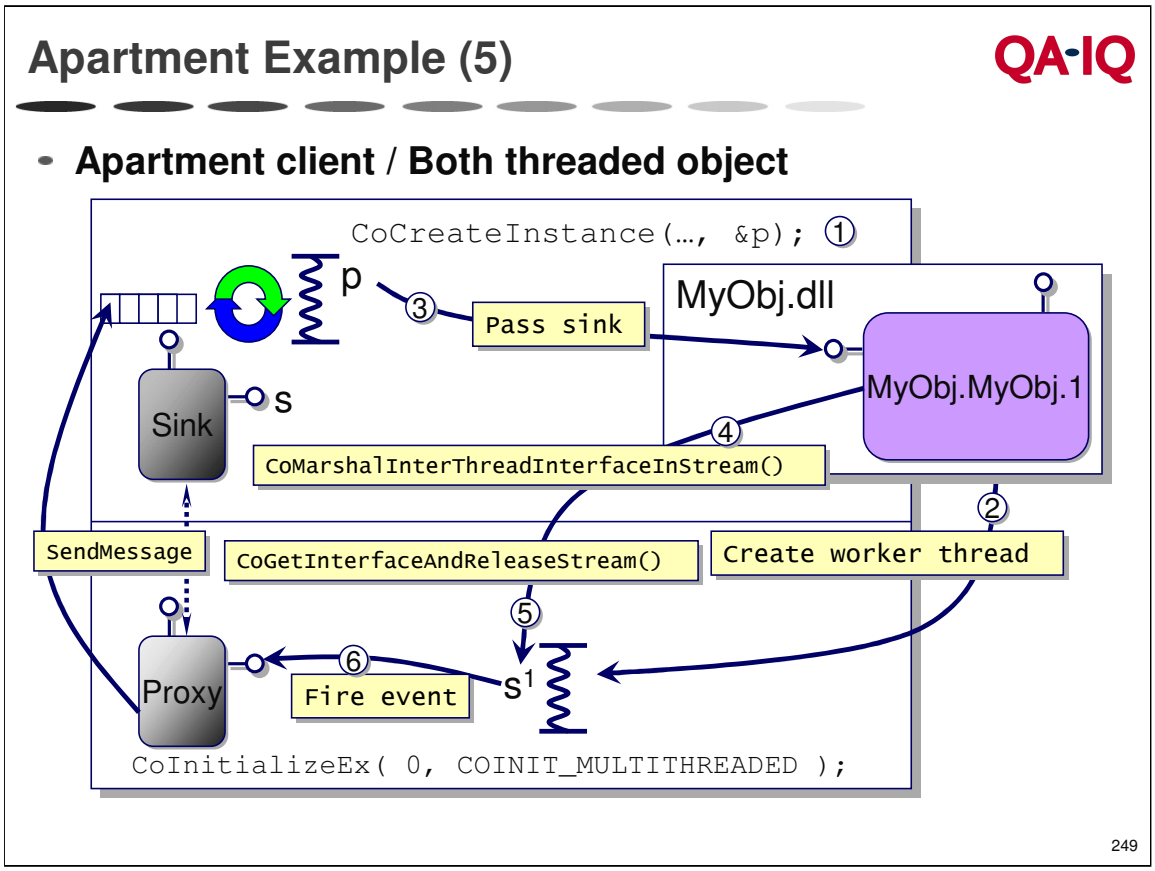


This diagram shows the sequence of events that occur when an Apartment model client creates a Free-threaded DLL based object. A Free threaded DLL is stating that it cannot live in an STA, it must live in the Multi-threaded Apartment (MTA).

First the client calls `CoCreateInstance()`. COM sees that the client thread is Apartment model and that the object has marked itself in the registry as being Free threaded. Because of this incompatibility, COM creates a new thread on the client's behalf. This thread calls `CoInitializeEx(0, COINIT_MULTITHREADED)`. COM then creates the object on this new thread and the apartment client thread gets back a proxy to the object.

The client then makes a method call, which occurs on the proxy, the proxy forwards the call to the MTA. A thread in the MTA is then used to call the object.

The thread which calls the object may not be the same thread that created the object. Remember, any thread in the MTA can be used to service a call on an object that lives in the MTA.



This diagram shows the sequence of events that occurs when an Apartment model client creates an object marked as `ThreadingModel=Both`. The object then creates a worker thread and the client then passes a sink pointer to the object*.

Firstly, the client calls `CoCreateInstance()` and COM sees that the object can live in either the MTA or an STA and so creates the object in the STA. As part of its initialisation code, the object creates a worker thread in the MTA.

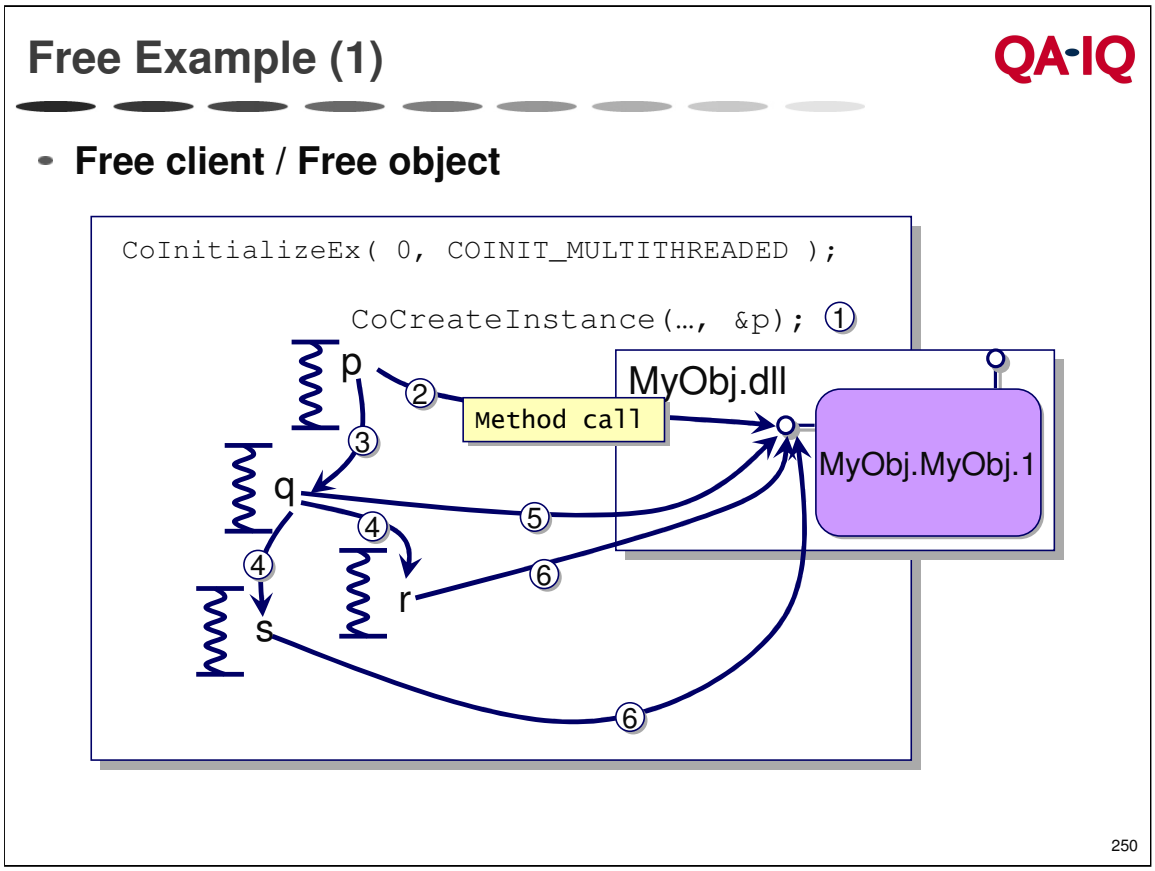
The client then passes a sink interface to the object. The object is expected to call the client sink when appropriate.

The object passes the sink interface to the worker thread. Because this involves passing the interface across an apartment boundary (STA to MTA) the object has to marshal the interface using `CoMarshalInterThreadInterfaceInStream()`.

The worker thread retrieves the reference using the standard `CoGetInterfaceAndReleaseStream()`, which results in a proxy being created in the MTA. This is a proxy to the sink and has a connection back to the STA message queue. When an event occurs, the worker thread calls the sink pointer on the proxy, resulting in a message being sent to the message queue of the STA.

* It should be noted that any DLL based COM object that creates a worker thread should normally be marked `ThreadingModel = Free` so that the object can be accessed directly from its worker thread.

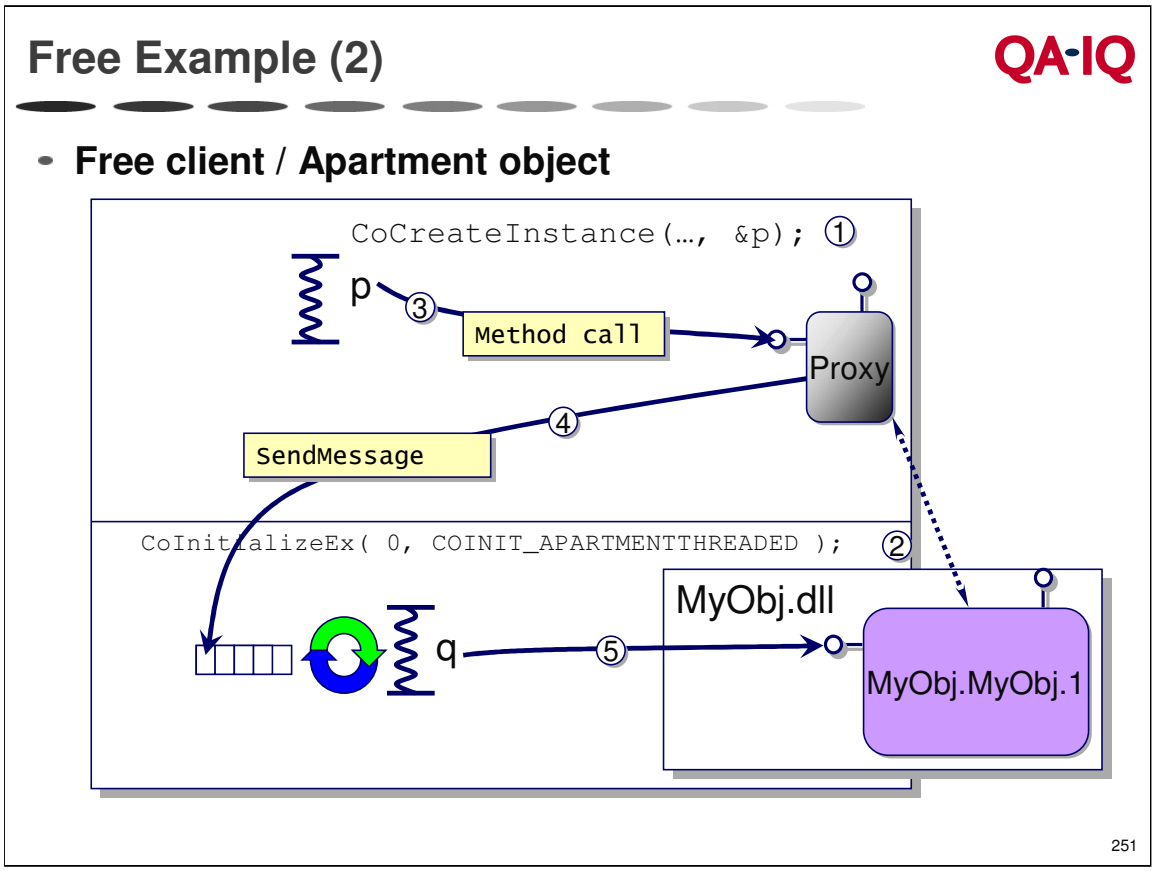
There is also one other consideration to be borne in mind. Starting a background thread in a COM object does contain some element of risk, as the code for the thread (which is held in the DLL) will be ejected from the process in response to a call to `CoUninitialize()`, leaving the thread in a disastrous state. For a fuller discussion on starting background threads (and other COM issues), see *Effective COM* by Don Box et al.



This diagram shows the sequence of events that occurs when a free threaded client, one that has called `CoInitializeEx(0, COINIT_MULTITHREADED)`, creates a free threaded DLL based object (one that has marked itself in the registry as `ThreadingModel=Free`). The client then passes the interface pointer to other threads that have called `CoInitializeEx(0, COINIT_MULTITHREADED)`.

First the client calls `CoCreateInstance()` and COM sets up a direct connection between client and object because the threading models are compatible. The client then makes method calls on the object. The interface pointer is then passed to other threads in the client and these threads also make method calls on the object. The object may end up servicing multiple method calls simultaneously.

A free threaded object is stating that it requires no synchronisation from COM. It will perform any necessary synchronisation where appropriate.



This diagram shows the sequence of events that occur when a free threaded client creates an Apartment threaded DLL object.

The client thread calls `CoCreateInstance()` and COM sees that the client and the object have incompatible threading models. COM creates a STA for the DLL object to live in and returns a proxy to the client thread.

The client then makes calls on the object, which actually occur on the proxy. The proxy sends a message to the message queue of the STA and the object method gets called. Since all method calls on the object are going via the message queue of the STA, there is no danger of the object being called re-entrantly*.

*In normal use. Re-entrancy is possible on an object running in an STA, if it makes calls on a COM object outside of its own apartment, or if the object retrieves and dispatches messages during a method call.

For more information on the subject of re-entrancy in STA, see the article

"INFO: Descriptions and Workings of OLE Threading Models ID: Q150777"

in the MSDN library.

Summary

QA-IQ

- Objects live in apartments
  - An apartment groups together objects with like concurrency constraints
  - Objects in MTAs and STAs can only be accessed by thread(s) in the owning apartment
  - COM currently supports STAs and MTAs
  - COM(+) under Windows 2000 provides the TNA
- STAs provide synchronisation via message queues
  - Useful for GUI based components
- MTAs provide no built in synchronisation
  - Best for high load, server components
- TNA can offer scalability/performance improvements
  - But use ThreadingModel = Both where possible

252

Apartments seem complex and confusing when you first meet them. Remember that they exist to manage *concurrency* issues between clients and objects. If COM detects that the client and the object have different threading requirements, then COM will attempt to overcome those differences by placing the COM object in an apartment that will enforce those requirements.

Current (Windows 2000) versions of COM support three apartment types (MTA, STA and TNA), although Windows NT 4 and Windows 9x only support the first two.