Using the Caché ActiveX Gateway

Version 2018.1
2024-11-07

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using the Caché ActiveX Gateway
PDF generated on 2024-11-07
InterSystems Caché® Version 2018.1

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1

About This Book

See the Table of Contents for a detailed listing of the subjects covered in this document.

This book is a guide to using Caché Activate to manipulate an external ActiveX object as if it were a native Caché object.

This book contains the following sections:

- Introduction
- Using Caché Activate

For general information, see *Using InterSystems Documentation*.

2

Introduction

Caché Activate gives Caché applications an easy way to interoperate with ActiveX (also known as COM) components from within a Caché server. By means of wrapper classes, ActiveX components are made available as instances of Caché object classes and can be used in the same manner as any other class. Caché Activate provides the ability to instantiate an external COM object and manipulate it as if it were a native Caché object.

Note: The terms "ActiveX" and "COM" are used interchangeably within this document.

Caché Activate is available only on platforms that support ActiveX (both 32-bit and 64-bit versions of Microsoft Windows).

Caché Activate works as follows:

1. Using the Caché Activate Wizard, you can create one or more wrapper classes. These are Caché classes that provide methods that correspond to the interface of an ActiveX component.
2. Within a Caché application, you can create an instance of an ActiveX wrapper class. Caché Activate transparently creates an instance of the appropriate ActiveX component within the same process. When you invoke the methods of the wrapper class, it automatically dispatches them to a method of the appropriate ActiveX interface.

You must exercise caution when using ActiveX components within Caché. Caché is designed to provide a safe environment for running application code. Every Caché server process runs an instance of the Caché virtual machine, is isolated from other service processes, and can handle application errors quite safely. ActiveX, unfortunately, is not a safe technology. Using ActiveX incorrectly or using poorly implemented ActiveX components can lead to memory leaks or unexpected application crashes. If you are using ActiveX components within a critical application, you should take extra care to ensure that you are using the interfaces of the component interfaces correctly and that the components have been thoroughly tested. It is a good idea to test any components using a tool such as Visual Basic before using them within your application.

2.1 Architecture

Caché Activate consists of the following components:

- The Caché Activate Wizard: This provides a simple graphical interface that lets you choose from the ActiveX components on your Caché server and automatically creates Caché wrapper classes for the components you select. The Caché Activate Wizard is accessible from the **Add-Ins** item on the **Tools** menu of the Atelier development environment. The Activate Wizard is available only on Windows systems.
- The Caché Activate Class Hierarchy: These are helper classes used by the generated wrapper classes in order to communicate with ActiveX.

- The Caché ActiveX Gateway: This is a shared library (DLL) loaded by and used by a Caché process to perform operations (loading, invoking methods, and releasing) on ActiveX components.

2.2 Overview of ActiveX / COM

The following is a simple overview of ActiveX / COM component architecture as it relates to Caché. If you intend to make use of ActiveX within your application, you should consult one of the many published works on the subject.

2.2.1 What Is a COM Object?

A COM object is a piece of code that conforms to the COM specification and provides one or more services that may be consumed by client programs. A certain class of COM objects, those which support the notion of Automation, are specially designed to be easily accessible from high-level programming languages such as VisualBasic, Delphi and now Caché. Such automation objects may be implemented as a dynamic link library and provide a simple function such as encryption of a text string or they may be full-blown executable applications such as Microsoft Excel or Microsoft Word which provide dozens of different services.

2.2.2 COM Interfaces

COM objects expose their functionality as interfaces. An interface is simply a collection of methods and properties that encapsulate some particular functionality. For example, a word processing object may provide a spell checking interface as well as a printing interface. Each implementation of a COM object is given a unique identifier in the form of a class id and each interface which it exposes also has a unique identifier referred to as an interface id. Once the class id of a particular object and the interface id of the required interface is known, it is possible for a client application to instantiate the COM object and avail itself of the services provided by the requested interface. By convention, when the name of an interface is written it is preceded by a capital "I", so the SpellCheck interface becomes ISpellCheck.

2.2.3 The IDispatch Interface

Different programing languages have different internal data types which are incompatible at a binary level. For example, a Caché local variable has a completely different implementation from that of a VisualBasic string or a C++ string. This makes it difficult to call an object written in one language from another, because conversion has to be done from say, a C++ data type to a Caché variable and vice versa. To solve this problem and enable different programming languages to communicate, the notion of the VARIANT data type and the IDispatch interface was developed.

At its simplest, IDispatch provides the ability to call a method or access a property in an external COM object by specifying the name of the method or parameter and passing the appropriate arguments. Arguments are represented by a VARIANT type, which is a standardized data type that the operating system supports. This standardized type is "understood" by all programming languages that support the use of COM automation.

By creating a COM object and requesting its IDispatch interface, a client program or language, such as Caché, can easily access the functionality exposed by the object.

Although IDispatch provides a generic means to access a COM Automation object, it is really intended as a technique that a programming language uses internally to provide COM object services via the particular constructs of that language. In other words, the high-level language should abstract the details of calling IDispatch and provide programming language constructs to ease use of external objects. Ideally such COM objects should act as if they are native objects within a programming environment. In Caché Activate, the key to this is to exploit the information contained in a COM objects type library.

2.2.4 Type Libraries

Most, if not all, COM Automation objects expose their metadata, i.e., a description of the types, methods and properties, in the form of a type library. The type library may be bound into a .DLL (dynamic link library), within an executable file as a binary resource, or it may exist in a separate file with an extension such as .tlb. Within the type library, each object is identified by a class id and is known as a CoClass. A CoClass may expose at most one IDispatch-derived interface known as the default interface. (Another interface known as the source interface may or may not be present. However it is not directly callable, and can safely be ignored for now). Some objects do not implement an IDispatch-derived interface at all and consequently are not callable via the IDispatch based mechanisms.

Caché Activate exploits the metadata contained in the type library by reading and decoding the information and creating Caché classes that expose the methods and properties defined therein. A type-library may contain one or more CoClass objects and potentially many IDispatch-derived interface definitions. There may be many interfaces because, although a CoClass may not expose more than a single IDispatch derived interface as its default interface, it is free to define methods and properties that either return or are typed as interfaces. In fact, this situation is common where a single CoClass (object) may define a rich object model. Consider a word processor for instance. It may provide a default interface of IApplication, which has methods such as **AboutBox**, **Exit**, etc. It also may provide a collection of documents (IDocuments*) as property called Documents.

Note: Many COM interfaces are quite complex; they may contain hundreds of methods and may use many additional COM objects as parameters. If your application needs to use only a small subset of a specific interface, you should consider building a wrapper COM component (for example using Visual Basic) to expose only the interfaces you actually need and to pass any requests to these interfaces to the original COM component.

3

Using Caché Activate

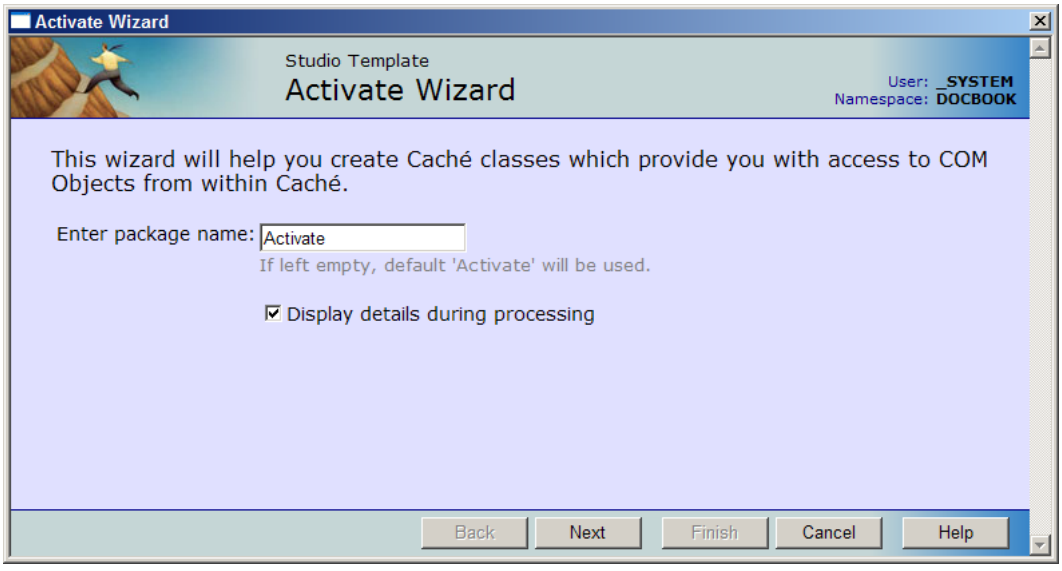
This chapter describes how to create a Caché wrapper class for an ActiveX component and how to use this wrapper class within an application.

3.1 The Caché Activate Wizard

The Caché Activate Wizard automatically creates one or more Caché wrapper classes for a given set of ActiveX interfaces.

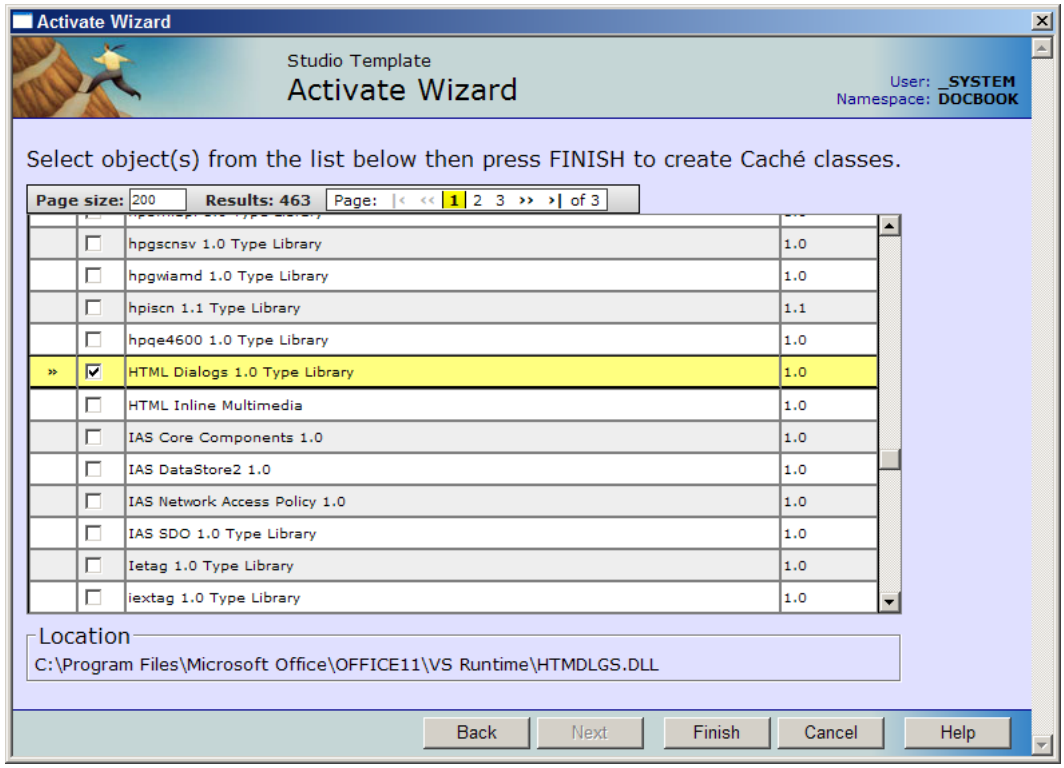
To use the Wizard:

1. Start Atelier.
2. Select a project for your application.
3. Select **Tools > Add-Ins...** from the main menu and press the **Next** button.
4. Expand the item **Standard Add-Ins** and select **Activate Wizard**.
5. Press the **Finish** button to start the Activate Wizard:



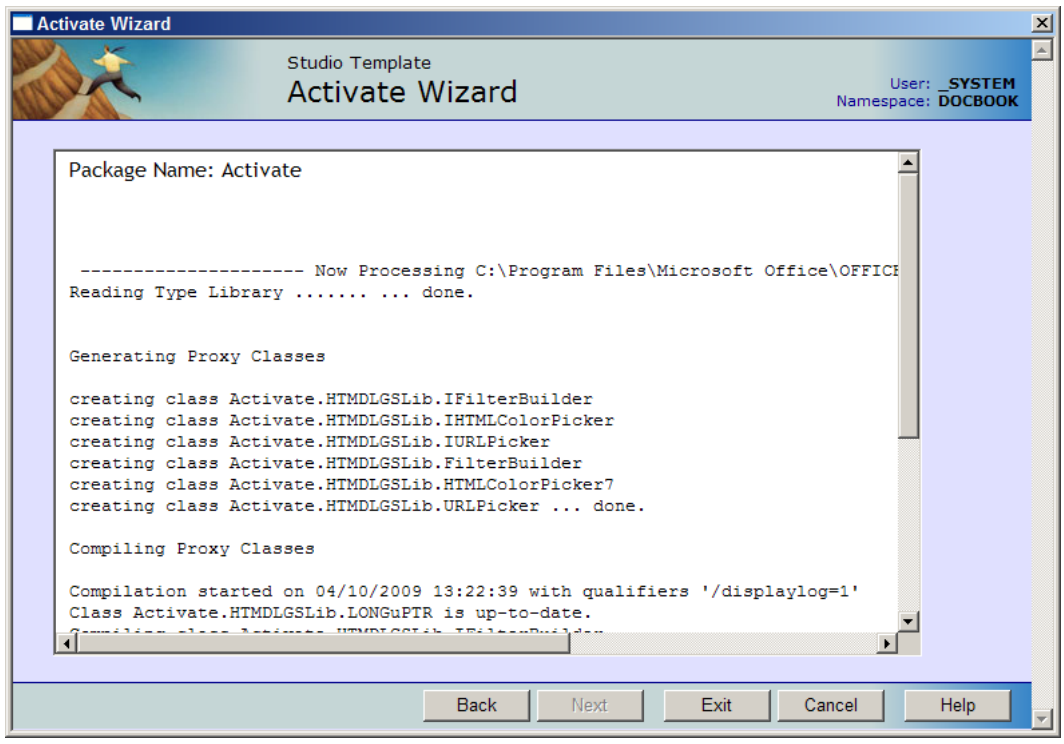
Enter the package name you wish to use for the generated classes, and press the **Next** button.

6. The Wizard displays a list of available COM interfaces (these are interfaces available on the Caché server, not the machine on which Atelier is running):



Choose one or more interfaces and press the **Next** button.

7. The Wizard automatically generates wrapper classes within the selected package and compiles them:



Note: **64–bit ActiveX Controls**

On a 64-bit version of Windows with a 64-bit cache, you can call 64-bit ActiveX controls. This will not enable a 64-bit cache to use 32-bit ActiveX controls (which is impossible due to operating system constraints). However, some companies are now releasing 64-bit versions of their existing 32-bit controls, allowing customers to migrate to 64-bit systems.

3.2 Using the Generated Wrapper Classes

The classes that are generated in Caché are proxy classes for the COM objects. Once the classes have been generated and compiled, you can then use them in Caché applications.

For example, using the Activate Wizard, you can generate wrapper classes for the Microsoft SysInfo Control, which provides some information regarding system resources.

The Caché Activate Wizard creates the following classes for the SysInfo COM object:

- Activate.SysInfoLib.ISysInfo — An abstract interface class that defines the methods and properties which the ISysInfo interface provides. It cannot be instantiated.

  Among others it has a calculated property called BatteryLifePercent along with corresponding get and set methods for that property.

- Activate.SysInfoLib.SysInfo — This is a concrete class that inherits from the ISysInfo class. It contains the code that finds and instantiates the external COM object and maintains a "connection" to that object. You use this concrete class to manipulate the external object. When the object is closed, the external COM object is closed (released) also.

3.2.1 Example: Accessing a Property

Here is an example that uses the SysInfo wrapper object to obtain the remaining battery life percentage for a laptop computer:

ObjectScript

```
Set obj = ##Class(Activate.SysInfoLib.SysInfo).%New()
Write obj.BatteryLifePercent,!
Set obj = ""
```

The object is created in the same manner as any other within Caché. The BatteryLifePercent property is written out and finally the object is closed.

3.2.2 Example: Enumerating COM Interfaces

The Caché Activate Wizard enumerates the type libraries on a Caché Server by using a COM object called TL.dll (or TL64.dll on 64-bit systems. The file is placed in the <CacheRoot>\Bin directory and automatically registered during Caché installation). The Caché classes that are generated from this object are preloaded into the %Activate.TLLib package.

These classes consists of:

- %Activate.TLLib.IUtils — an abstract interface class that has a single property, libraries of type ILibraries. Use this property to retrieve the ILibraries interface for enumerating the type libraries on the system.
- %Activate.TLLib.ILibraries — an abstract interface class that exposes the Count and Item properties. Use these properties to enumerate the type libraries on the system.

- %Activate.TLLib.Utils — a concrete subclass that expresses the IUtils interface. Instantiate this class to access the Libraries property

Here is an example ObjectScript method that enumerates the type libraries on the system by using these classes. A concrete instance of the Utils class is created and the objlibs property is retrieved. Notice that the Item property is called via the **ItemGet** method, because Caché does not currently support calculated, indexed properties:

Class Definition

```
Class MyApp.ActivateTest
{
// ...

/// Demonstrate COM object Access and provide type library enumeration  ;
ClassMethod ListTypeLibs() {
    Set objUtils = ""
    Set objLibs = ""
    Set $ZT = "tlerr"
    Set objUtils = ##class(%Activate.TLLib.Utils).%New()
    Set objLibs = objUtils.Libraries
    For i = 1:1:objLibs.Count {
       Set tld = objLibs.ItemGet(i)
       // tld is a | delimited string
       Write !, $Piece(tld,"|"), !, $Piece(tld,"|",2), !, $Piece(tld,"|",3), !!
    }

xit      ; Exit point
    If objLibs'="" Set objLibs = ""
    If objUtils'="" Set objUtils = ""
    Quit

tlerr      ; Exception handler
    Set $ZT = ""
    Goto xit
}
}
```

3.2.3 Special Considerations for Properties

As shown in the previous example, in COM, some *properties* have parameters. Furthermore, some objects have what is known as a "default property," which means you can reference that property without specifying its name explicitly.

For example, collections (as in the previous example) always have the Count and Item property. You will note that the Item property is (obviously) not a method but that it *does* take an argument. An Item property is often the default property of a collection. Consider an example with Microsoft Excel. If we have a collection of workbooks, then in Visual Basic, we can access a specific workbook by name in this manner:

```
Application.Workbooks("Sheet1")
```

Although we are accessing the Item called "Sheet1", Item is not explicitly referenced. What the code is really doing is calling:

```
Application.Workbooks.Item("Sheet1")
```

Caché distinguishes between method call and property reference by the presence or absence of parentheses. This means that it interprets "person.Name" as a property and "person.RaiseSalary()" as a method. This makes default properties awkward because, unlike Visual Basic, Caché does not have the ability to define a default parameter nor the ability to do a property reference while passing parameters. For example, Caché cannot support the following Visual Basic syntax that has an implicit reference to an property:

```
Workbooks("Sheet1") ' Implicit reference to Item property
```

Neither can Caché support the following syntax, where Item is a property:

```
Workbooks.Item("Sheet1") ' Item is a property!
```

This does not work, because the Caché Interpreter considers Item to be a method. To work around this difference in the languages, use the following syntax:

```
Workbooks.ItemGet("Sheet1")
```

This works because ItemGet is the method that retrieves the Item property.

3.3 Exception Handling

Any COM object may raise an exception as the result of some operation, be it a method call or a property set/get. When an exception is raised, the exception is propagated into Caché via the *ZTrap* mechanism. The calling code will receive an error with the error code <ZACTX> and the local variable %objlasterror will contain a complete textual description of the error. Programmers should plan for this error and take action accordingly.

3.3.1 Example: Exception Handling

Here is an example of using a COM object which retrieves files by FTP. The object is created and the CurrentDirectory property is queried. The COM object throws an exception because it is not valid to try to determine the current directory until the FTP connection has been made. We will try this from a Caché command line (terminal session):

ObjectScript

```
 Set obj = ##Class(Activate.RETRIEVERLib.FtpRetriever).%New()
 Write obj.CurrentDirectory
```

In this case, this will throw an error:

```
<ZACTX>CurrentDirectoryGet+4^Activate.RETRIEVERLib.FtpRetriever.1
```

The error code associated with the <ZATCX> error should be in the local variable *%objlasterror*. We can retrieve the complete text of the error message by calling **$system.OBJ.DisplayError**:

ObjectScript

```
 Do $system.OBJ.DisplayError(%objlasterror)
```

Which will result in the following output:

```
ERROR #1101: Com Exception: '-2147220888 Ftp Retriever Connection must
be established before attempting this operation'
```

3.4 %Activate.IDispatch and %Activate.GenericObject

Some COM objects do not come with a type library or you may find that the return type of a method or a property type of a COM object is just an IDispatch interface. How do you call methods and access properties for such objects?

Caché Activate provides two classes which assist with this problem, %Activate.IDispatch and %Activate.GenericObject.

Many COM objects are identified by what is called a "ProgId", a string usually consisting of a library/object name which can be used to identify an object. In Visual Basic there is a **CreateObject** call which takes a Progid and returns an object reference which can be used to manipulate the object. Caché provides a **CreateObject** method too, as a class Mmethod of the %Activate.GenericObject class. Here is how it is used:

3.4.1 Example: Using CreateObject

Using the same Microsoft SysInfo object as above, we instantiate the object via its ProgId. Because the object is generic, that is, we have no type information for this object when instantiated in this manner, we must call the generic methods from the IDispatch interface which get and set properties and invoke methods by name:

ObjectScript

```
Set obj = ##Class(%Activate.GenericObject).CreateObject("SYSINFO.SysInfo")
Write obj.GetProperty("BatteryLifePercent")
Set obj = ""
```

3.5 Monikers

COM provides an alternative way of instantiating an object indirectly by using what is known as a moniker as a substitute for the ProgId. Visual Basic provides the **GetObject** call which takes a moniker and returns an object reference which can be used to manipulate the object. Caché provides a **GetObject** method as a Class Method in the %Activate.GenericObject class. Here is how it is used:

3.5.1 Example: Using GetObject

Here a moniker that accesses the LDAP protocol of the Active Directory service. It is used to return a reference to a collection of nodes which represents users in the current domain. The count of users is written out and the object closed:

ObjectScript

```
Set obj = ##Class(%Activate.GenericObject).GetObject("LDAP://CN=USERS")
Write obj.Count()
Set obj = ""
```

3.6 The Become Method

Sometimes a type library specifies a method or a property which has a return type of the generic IDispatch interface. This can be very inconvenient because what you get is, in effect, an instance of %Activate.IDispatch on which you are forced to use generic methods (such as **GetProperty**) in order to get and set properties and invoke methods. If you *know* the interface that it really should be (from documentation or otherwise), then you can call the **Become** method on an instance of %Activate.IDispatch object and retrieve the new (now typed) interface. The **Become** method takes the name of a class as its argument. Effectively, %Activate.IDispatch becomes an instance of the class name you pass to the method. **Become** will throw an exception if the object you call does not support the new typed interface.

3.7 Events

Some COM components have the ability to fire events during the processing of a method. The events are grouped into an event or "source" interface given a name. For example, given a COM object called MyClass, the interface may be called "MyClassEvents" or in the case of a COM object created with Visual Basic "__MyClass".

Caché Activate provides for the event handling via two classes: %Activate.RegisterEvents and %Activate.HandleEvents. If a COM object generates events, the generated Caché class will inherit from the %Activate.RegisterEvents interface class. This adds two methods **%RegisterHandler** and **%UnRegisterHandler**. In addition to the regular COM object proxy class, another class is generated which represents the Event interface. This will inherit from %Activate.HandleEvents and implements the **%Advise** and **%UnAdvise** methods as well methods to handle specific events as defined by the event interface.

3.7.1 Example: Using COM Events

An example may make things clearer. Suppose we have a hypothetical COM object which does an FTP transfer. As well as implementing methods such as **Connect**, **Close**, and **Download**, the object implements an Event interface which expresses a single method, **BytesTransferred**. Following a successful connection and initiation of a download, the FTP object will fire the "BytesTransferred" Event after each 1 kilobyte of data that it has downloaded. The Event will be represented by a **BytesTransferred** method which has two parameters, an integer, *Bytes* and a boolean, *Cancel* which is passed by reference. When the Event fires, the **BytesTransferred** method will be called passing the current value of the arguments, *Bytes* and *Cancel*. These values are then available for processing. Typically the *Bytes* argument will be displayed via the user interface. Because the *Cancel* argument has been passed by reference, its value may be set and returned to the COM object which fired the event. In this instance setting *Cancel* to True (-1 for COM) will indicate to the COM object that the current operation should be interrupted and the call to **Download** should return immediately. If the download completes normally, the call to **Download** will return control to the caller and no more events will be fired. In Caché, the FTP COM object would be represented by a generated class such as Activate.SomeLibrary.FTP and the event interface by the class Activate.SomeLibrary.FTPEvents.

This example would look something like this. First an instance of the FTP object would be created:

ObjectScript

```
Set FTP = ##Class(Activate.SomeLibrary.FTP).%New()
```

We want to handle events so we create an instance of an event handler:

ObjectScript

```
Set FTPHandler = ##Class(Activate.SomeLibrary.FTPEvents).%New()
```

Before events can be handled the event handler must be registered with the object that actually fires the events, so we call:

ObjectScript

```
Do FTP.%RegisterHandler(FTPHandler)
```

Now we connect and do a download:

ObjectScript

```
Do FTP.Connect("ftp.intersys.com")
Do FTP.Download("/public/somefile.txt")
```

During the download the following method would be called on the Activate.SomeLibrary.FTPEvents class:

Class Definition

```
Class MyApp.Test
{
//...

Method BytesTransferred(Bytes As %Integer,Cancel As %Boolean)
{
    //...
}
}
```

Note: It is up the developer to actually implement the **BytesTransferred** method by editing the Activate.SomeLibrary.FTPEvents class directly or preferably by subclassing the class and providing the implementation in the subclass.

Following the download, we do not want events to be handled anymore so we unregister the handler:

ObjectScript

```
Do FTP.%UnRegisterHandler(FTPHandler)
```

and tidy up:

ObjectScript

```
Set FTPHandler = ""
Set FTP = ""
```