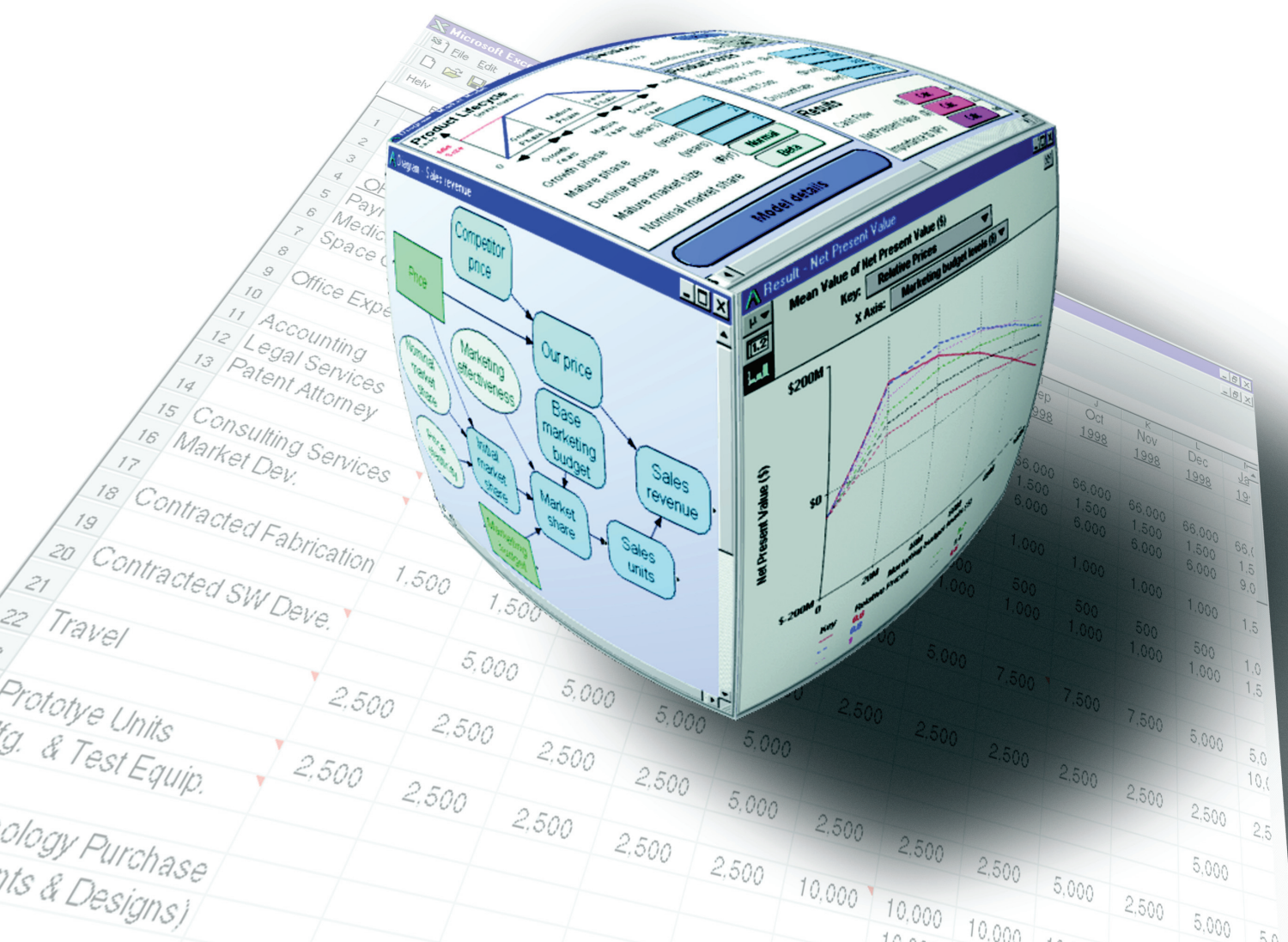




analytica[®]

Beyond the Spreadsheet

Analytica[®] Decision Engine User Guide Release 4.1



Lumina Decision Systems, Inc.
26010 Highland Way
Los Gatos, CA 95033
Phone: (650) 212-1212
Fax: (650) 240-2230
www.lumina.com



Copyright Notice

Information in this document is subject to change without notice and does not represent a commitment on the part of Lumina Decision Systems, Inc. The software program described in this document is provided under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement.

This document and the software program that it describes, the **Analytica Decision Engine**, are copyrighted 1998-2008 by Lumina Decision Systems, Inc. All rights reserved.

The Analytica Decision Engine software contains software technology licensed from Carnegie Mellon University exclusively to Lumina Decision Systems, Inc., and includes software proprietary to Lumina Decision Systems, Inc. Carnegie Mellon University and Lumina Decision Systems, Inc., make no warranties whatsoever, either expressed or implied, regarding this product, including warranties with respect to its merchantability or its fitness for any particular purpose.

Analytica is a registered trademark and Lumina Decision Systems and Intelligent Arrays are trademarks of Lumina Decision Systems, Inc.

Acknowledgements

The ADE User Guide was written by Richard Sonnenblick, Hugh Silin, Lonnie Chrisman, Max Henrion, and Richard Morgan.

Contents

About ADE	1
What is the Analytica Decision Engine?	2
Using the ADE server	2
How to use this document	3
Chapter 1: Installation	5
System requirements	6
Installing the Analytica Decision Engine files	6
Installing from the network	6
Installing from CD	6
Entering a new license code	7
Upgrading from an earlier version of ADE	7
Uninstalling ADE	7
Chapter 2: The Analytica Decision Engine Tutorial	9
Your first ADE application	10
What's next?	10
Distinguishing title from identifier	11
Creating an ADE object from within Visual Basic	12
COM vs. Automation interface	12
Opening a model with ADE	13
Retrieving objects from the Analytica model	13
Getting object attributes	14
Evaluating objects and retrieving results	14
Getting the index elements of a table	14
Getting information from CATable and CAIndex	15
Controlling formats of atomic values	16
Other ways to access tables	16
Modifying objects	16
Graphing with ADE	18
Conclusion	19
Chapter 3: Using the Analytica Decision Engine Server	21
ADE classes	22
Server class architecture	22
COM, Automation, and .NET	22
In-process vs. out-of-process	22
Typescript	23
Security permissions under IIS 5	23
The AdeTest program	23
Sample application in Excel's Visual Basic	25
Sample ASP web application	25
Using the ADE COM interface	25
From a .NET project in Visual Studio 2005	25
Releasing objects in .NET	26
From an ATL project in C++	26
Using the ADE Automation interface	27
From Visual Basic or VBScript	27
ADE typescript: command language communication	27
Errors and error handling	30
Chapter 4: Working with Models, Modules, and Files	31
Models and modules	32

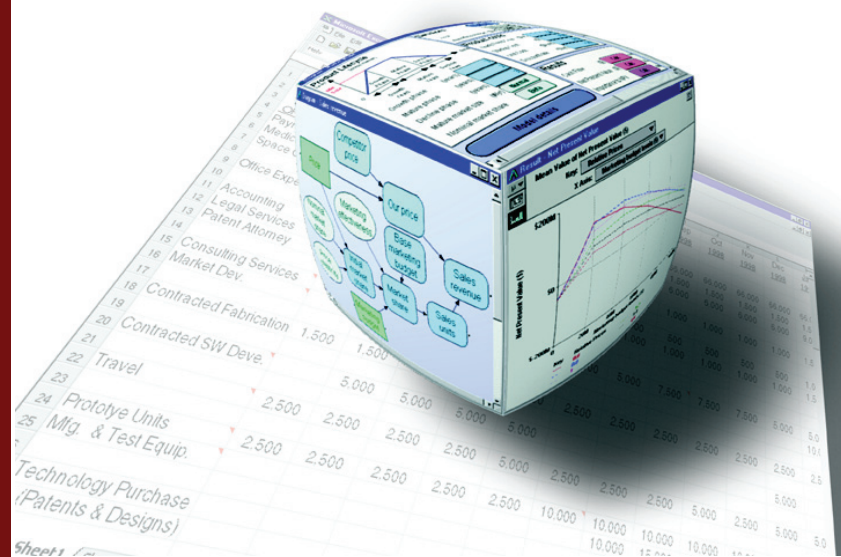
Contents

ADE objects	33
Retrieving computed results.....	34
Retrieving multi-dimensional results	35
Creating tables and setting values in tables.....	41
Adjusting how values are returned.....	44
Terminating an in-progress computation.....	46
Instantiating CAEngine using CALicense.....	47
Using the Analytica Graphing Engine	47
Chapter 5: ADE Server Class Reference	49
ADE server classes.....	50
Class CAEngine	50
Properties.....	50
Methods	51
Class CALicense.....	54
Properties.....	55
Methods	56
Class CObject.....	56
Properties.....	56
Methods	58
Class CTable	60
Properties.....	60
Methods	61
Class CIndex	65
Properties.....	65
Methods	66
Class CRenderingStyle.....	66
Properties.....	66
Appendix A: Error Codes	70
ADE error codes.....	70
API error codes	70
Index.....	73

Introduction

About ADE

This chapter provides an overview of the Analytica Decision Engine (ADE) and summarizes the information presented in this book.



What is the Analytica Decision Engine?

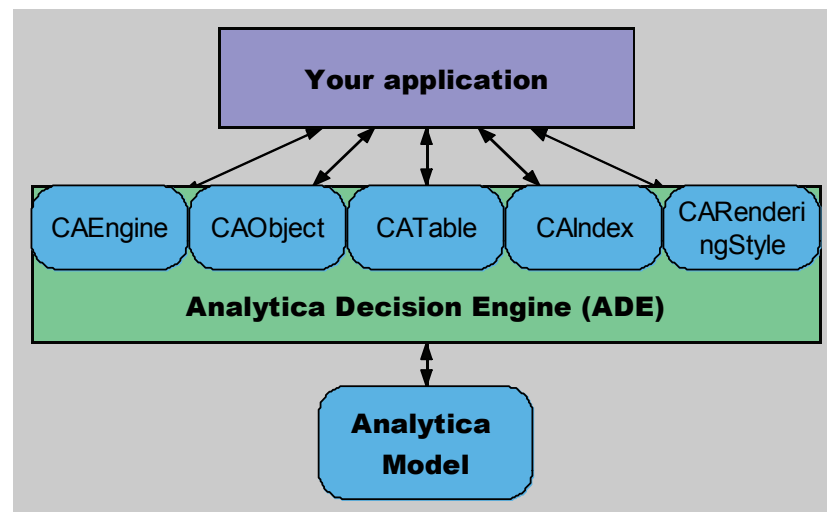
The *Analytica Decision Engine* (ADE) is a powerful COM component that helps you to access Analytica models via a program. ADE lets you run any Analytica model on a server computer. It provides an Application Programming Interface (API) through which other application programs can create, read, check, parse, evaluate, modify, and save Analytica models. For example, you can create a user interface accessible via a web browser so that users can run Analytica models as web applications. Or you can use ADE to access your Analytica model from another application that can supply inputs, run the model, and collect and display results.

Although you can use ADE to build and edit models with commands issued via the API, it is usually much more convenient to use Analytica Enterprise for this purpose (see the *Analytica Tutorial* and *Analytica User Guide* for details, including the “Analytica Enterprise” chapter of the *Analytica User Guide*). After you have an Analytica model, you can use ADE to build a custom user interface via a web browser or other application, to interface the model with another application.

ADE is provided in two forms so that it is compatible with a wide range of applications. These forms are an ActiveX in-process automation server, **Adew.dll**, and a COM local automation server, **ADE.exe**. The classes, methods, and properties exposed by these servers are accessible from any programming environment that supports the use of COM, ActiveX Automation, or .NET interfaces. Such environments include VB, VB.NET, ASP, ASP.NET, C#, Visual C/C++, J#, VB Script, and JavaScript. For example, you can use Visual Basic or C# to create graphical user interfaces (GUIs) on 32-bit Microsoft Windows platforms for your Analytica models, tailored to specific applications and specific classes of end users.

Figure shows a conceptual model of ADE. Your application makes calls to the functions exposed by the interface classes of ADE. Those functions then return information to your application. Server objects allow you to read, check, parse, evaluate, modify, and save Analytica models from within your applications.

Conceptual model of
Analytica Decision
Engine



Using the ADE server

ADE provides objects of six OLE classes: **CALicense**, **CAEngine**, **CAObject**, **CTable**, **CAIndex**, and **CARenderingStyle** ("CA" stands for Class Analytica). You use these classes to interact with your Analytica model through ADE. The **CAEngine** class contains methods and properties to open and close existing models, create new models, and access objects in your Analytica model.

It is important to distinguish these OLE object classes in ADE from the Analytica object classes. Analytica classes include chance, decision, index, objective, and variable (which we refer to collectively as **variable classes**); model, module, and library (which we refer to collectively as **module classes**); functions, and attributes. You can access Analytica objects as instances of the

CAObject class. This class provides properties and methods to get and set attributes of Analytica objects, including identifier, title, and description, as well as definition and value for variables.

You can access the value of a variable via the **ResultTable** property of class **CATable**. A **CATable** represents an Analytica **array** (also known as a **table**) so that you can get or set its individual elements (also known as **cells**). Each element can be a **number** or a **string** value (termed a **text** value in Analytica).

A **CATable** has zero or more dimensions. Zero dimensions means it is **atomic** (it has a single element). Each dimension is identified by an Analytica index, represented by the **CAIndex** class. A **CAIndex** has a name and a list of labels, numbers, or strings used to identify the rows or columns (more generally, **slices**) of the array. In Analytica, you identify dimensions of an array by name, not by order.

The **CARenderingStyle** class provides control over formatting of returned values as numbers or text.

Before your code can interact with ADE, you must create a **CAEngine** object, from which all else is obtained. You can create a **CAEngine** class directly, or you can first obtain a **CALicense** object, and then use it to create a **CAEngine**. The **CALicense** class can tell you whether your ADE license will allow you to create a **CAEngine**, and provide more detailed information about why the **CAEngine** could not be created when a failure occurs.

How to use this document

The rest of this guide includes these sections:

Chapter 1, Installation

This chapter explains the steps required to install the Analytica Decision Engine 4.1 on your Windows NT 4 (>SP 6), 2000, XP, or Vista computer.

Chapter 2, The Analytica Decision Engine Tutorial

This chapter shows you how to use the Analytica Decision Engine (ADE) from within a Visual Basic program, and steps you through building your first ADE application using Visual Basic.

Chapter 3, Using the Analytica Decision Engine Server

This chapter provides a step-by-step guide to the functionality accessible through ADE. You should read this section to get better acquainted with the classes, and their methods and properties. By using the sample code fragments presented in this section in your code, you can begin accessing information in your models from your Visual Basic applications immediately.

Chapter 4, Working with Models, Modules, and Files

This chapter contains examples of common operations and manipulations you might perform on objects in your Analytica model.

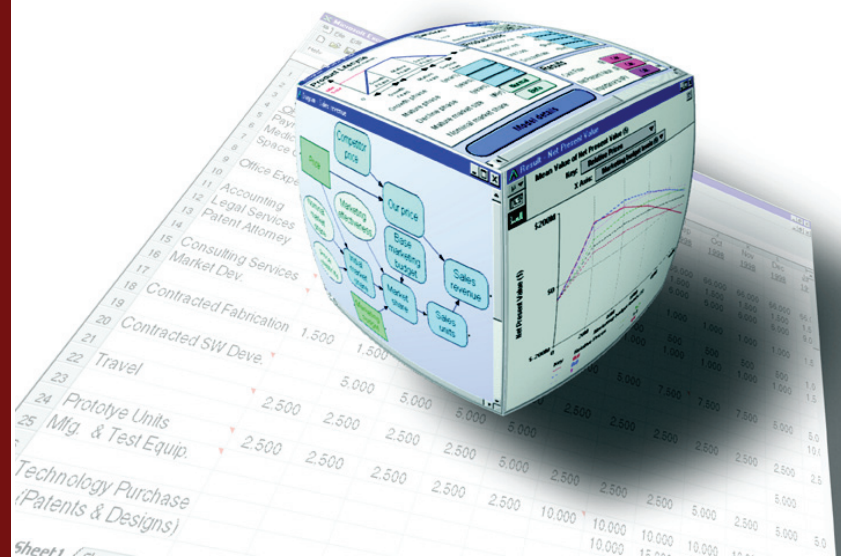
Chapter 5, ADE Server Class Reference

This chapter provides reference materials on the sixobject classes in ADE and their properties and methods, including method syntax, data types, and property access information. Refer to the information in this section after you've read through "Chapter 3, "Using the Analytica Decision Engine Server," and have specific questions about particular methods and properties.

Chapter 1 *Installation*

Installation

This chapter explains the system requirements for the Analytica Decision Engine (ADE) and describes how to install, upgrade, and uninstall ADE.



System requirements

- Windows NT 4.0 (>SP6), 2000, XP, Windows Server 2003, and Vista
- 20 MB of hard drive space (you need more space to develop your applications)
- 256 MB of RAM
- You also need a development language environment to build your application using ADE. This could be Visual Studio with VB.NET, C#, ASP.NET, VC++, or any other COM or .NET-enabled development environment. You can also use ADE from Microsoft Office Visual Basic for Applications (VBA) or from Windows Scripting Host (**CScript.exe** or **WScript.exe**), as well as other COM-enabled or ActiveX automation-enabled packages.

You can obtain all the files for installation of ADE from the ADE CD-ROM or you can download the installer from <http://www.lumina.com/ana/support/download.htm>.

The installation contains the ADE in-process automation server (**Adew.dll**), the ADE local automation server (**ADE.exe**), auxiliary files needed by ADE, this *ADE User Guide*, and example programs.

Installing the Analytica Decision Engine files

Installing from the network

Obtain an ADE 4.1 license code from Lumina. This is supplied to you, usually through e-mail, when you purchase ADE. You must complete the installation within three days after the license code is issued to you.

Download the ADE setup executable. The location of the file is provided when you receive your license code. Save the file to disk.

Run (i.e., double click) the file you just downloaded to begin the ADE installer..

Follow the instructions. Read and agree to the license agreement, select a directory for the installation, and enter your license code when prompted.

If the installer reports that your license code is stale, go to <http://lumina.com/ADE/staleLicense> to obtain a fresh code. After you obtain a fresh code, be sure to enter the license code within three days.

Installing from CD

Insert the Analytica Decision Engine CD into your CD-ROM drive.

If the installer does not automatically start, run the **setup.exe** program on the CD.

During setup, you need to select a directory for installation, to read and agree to the licensing terms, and to enter the license code supplied to you by Lumina Decision Systems when you acquired ADE.

If the installer reports that your license code is stale, go to <http://lumina.com/ADE/staleLicense> and obtain a fresh code. After you obtain a fresh code, be sure to enter the license code within three days.

After following the steps above, the following files should exist in the directory in which you installed ADE:

- **Adew.dll**
- **ADE.exe**
- **Analytica.ini**
- **Analytica.i**
- **ODBC4Analytica.dll**
- **license.txt**
- **SolverSDK.dll**

- `readme.txt`
- `solver.lic`

Two ADE manuals are installed into a subdirectory called `docs`. These are:

- `ADE User Guide.pdf` (this document)
- `ADE Scripting.pdf`

Four example programs should also have been installed in that directory underneath the examples directory. They are as follows:

Tutorial.NET — This program is referred to by the *Analytica Decision Engine Tutorial*. It is recommended that you read the *Analytica Decision Engine Tutorial* completely before writing your own programs that depend on ADE.

AdeTest — This program allows you to call or test the methods of ADE objects through a GUI. You can run `AdeTest.Exe` (in the bin directory), or you can trace through the code in the Visual Studio.NET 2005 debugger to observe each method being called.

asp_exam — This program shows you how to access ADE through a Microsoft ASP program.

excel_exam — This program shows you how to access ADE from any application with Visual Basic for Applications (VBA) support, including the Microsoft Office suite of applications.

Entering a new license code

If you have previously installed an earlier version of ADE and need to enter a new (different) license code, follow these steps:

1. Open a command prompt.
2. Select **Start > Run** and type `Cmd.exe`.
3. Use the `cd` command to change the directory to the same directory where you installed the earlier version of ADE.
4. Type `ADE /RegServer`.

A dialog appears prompting you enter your new license code.

Upgrading from an earlier version of ADE

ADE 4.1 has been configured to install without disturbing previously installed versions of ADE. This allows you to compare the performance and output of your application under the different versions. However, it also means that your existing applications continue to use the previous version until you have changed them to use ADE 4.1.

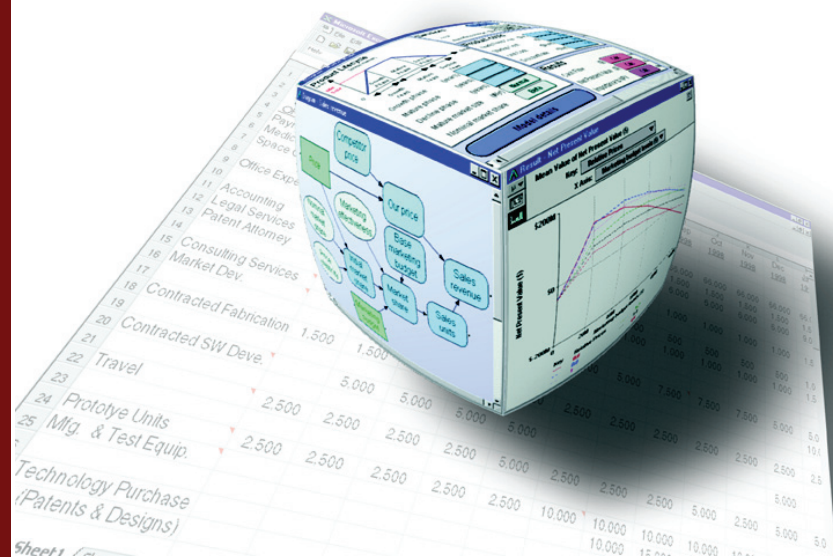
Uninstalling ADE

To uninstall ADE, select **Add/Remove Programs** in the Windows Control Panel. Scroll through the list to find “Analytica Decision Engine 4.1”. Click the **Remove** button and proceed through the uninstall wizard.

Chapter 2

The Analytica Decision Engine Tutorial

This tutorial shows you how to use the Analytica Decision Engine (ADE) from within a Visual Basic program.



Your first ADE application

First let's write a simple ADE application from scratch, just to be sure that everything is set up correctly. Follow these steps:

1. Bring up Visual Studio.NET.
2. Select **New Project**, then select the Project Type "Visual Basic Projects" and the template "Console Application." Select a project name, e.g., "FirstADEtry," and an appropriate folder location.
3. From the **Project** menu, select **Add Reference** and select the **COM** tab in the dialog. Find and select **Analytica Decision Engine Local Server 4.1 (Ade.exe)** and click **OK**.

Note: If you cannot find this entry in the list of COM servers, then ADE 4.1 is not properly installed. See "Installation" on page 5 for instruction on how to install ADE before reading further.

4. Add to the **Module1** class as follows:

```
Imports ADE
Module Module1
    Public ADE As CAEngine
    Sub Main()
        Dim FileName, ModelName As String
        FileName = "C:\Program Files\Lumina\Analytica 4.0"
        FileName &= "\Tutorial Models\Car Cost.ana"
        ADE = New CAEngine
        ModelName = ADE.OpenModel(FileName)
        If ModelName = "" Then
            Console.Write(FileName & " not found")
        Else
            Console.Write("Congratulations on opening ")
            Console.WriteLine(ModelName)
            Console.WriteLine("Press 'enter' to exit")
            Console.ReadLine()
        End If
    End Sub
End Module
```

5. Now run the program. If your program prints "Congratulations on opening Carcosts" you have just successfully written your first ADE program.

This first program did the following:

- Created a **CAEngine** automation object called **ADE** (using **new CAEngine**).
- Opened an Analytica model (using the **OpenModel** method of **CAEngine**).
- Displayed the name of the model (the return value of **OpenModel**).

We go into the details of these and other functions in the next section.

What's next?

We will not attempt to explain *all* of the features of ADE in this tutorial. These are described in the following chapters of this guide. Here, we give you the background to explore the more advanced features of ADE on your own.

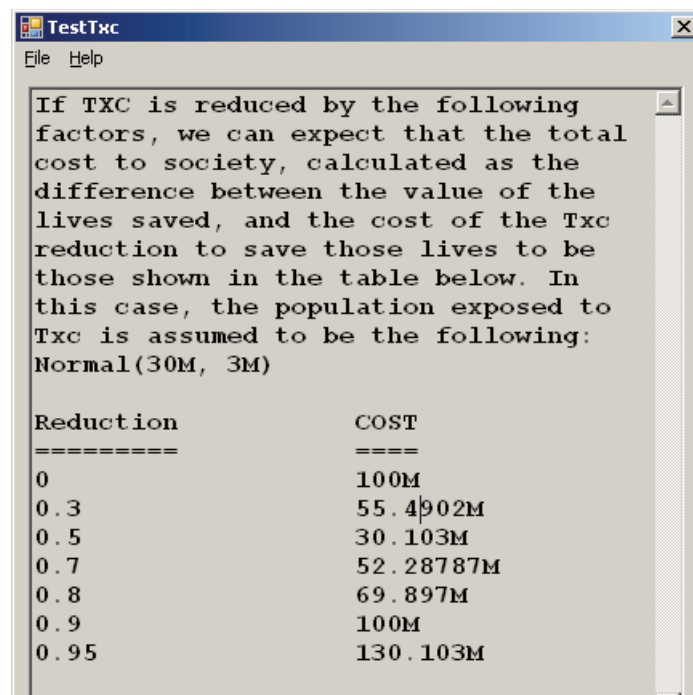
From this point, we use the example model called **Txc.ana**. You can find **Txc.ana** in the **Risk Analysis** folder under the **Example Models** folder installed with Analytica. If you cannot find it, or if you opted not to install the examples when you originally installed Analytica, there is a copy in the **Examples\Tutorial.NET** folder in the directory where you installed ADE.

The Txc model demonstrates risk-benefit analysis of reducing the emissions of the fictitious air pollutant Txc. Please open the Txc model with Analytica to see how it works.

The example Visual Basic.NET program called **TestTxc** in your **Ade Examples\Tutorial.NET** folder shows many aspects of ADE. This program creates an ADE automation object, opens the **Txc.ana** model with this object, gets the definition of the **Population Exposed** variable, evaluates the **Total Cost** variable, prints out the result of the **Total cost** variable as a table by getting at the individual components of the table, and changes the definition of the **Population Exposed** variable. It then gets the result of the **Total cost** variable again, to see what effect the change of definition for **Population Exposed** had on the **Total Cost** variable. If things are set up properly, **TestTxc** displays the window shown in "Text Txc window".

The application displays the definition of the **Population Exposed** variable ("**Normal (30M, 3M)**"), and the table associated with **Total Cost**, based on the definition of **Population Exposed**. You can change the definition of **Population Exposed** by selecting **File > Change Population Exposed** from the main menu and seeing the effect this has on the **Total Cost** table.

Text Txc window



The screenshot shows a window titled "TestTxc" with a menu bar containing "File" and "Help". The main text area contains the following text:

If Txc is reduced by the following factors, we can expect that the total cost to society, calculated as the difference between the value of the lives saved, and the cost of the Txc reduction to save those lives to be those shown in the table below. In this case, the population exposed to Txc is assumed to be the following:
Normal(30M, 3M)

Reduction	COST
0	100M
0.3	55.4902M
0.5	30.103M
0.7	52.28787M
0.8	69.897M
0.9	100M
0.95	130.103M

Distinguishing title from identifier

Whenever an ADE function requires a variable, you must pass it the **identifier** of the variable, not its **title**. This can be confusing since Analytica normally displays the titles of each variable in an influence diagram. By default, when you first create each object, Analytica automatically creates an identifier based on the title. It substitutes underscore (_) for each blank or other character in the title that is not a letter or number.

You can show the identifiers in an influence diagram by pressing **Control+y** (or by selecting **Show by identifier** from the **Object** menu). For model **Txc.ana**, you can see that the identifier of the variable titled **Population Exposed** is **Pop_exp**. It is important to use **Pop_exp** as the identifier when passing this variable to ADE functions. ADE would not be able to find the variable if you pass **Population Exposed** instead, and would return an error.

Creating an ADE object from within Visual Basic

If you haven't already, load the project called `Examples\Tutorial\TestTxc.sln` into Visual Basic.NET, and view the code for the file called `TestTxc.vb`. The code looks like this:

```
Imports ADEW
...
Public adeEngine As CAEngine

Public Sub Main()
    Dim exeDirectory, theModel As String
    Dim theModelString As String
    exeDirectory = VB6.GetPath
    theModel = exeDirectory & "..\" & "Txc.ana"
    adeEngine = New CAEngine
    ...
    theModelString = adeEngine.OpenModel(theModel)
    ...
    frmMain.DefInstance.Show()
End Sub
```

At the very top of the file, the code declares the automation object **adeEngine** as a **CAEngine** object. Using this object, we can access all of the public functions exposed by **CAEngine** (see “ADE Server Class Reference” on page 49 for a complete listing). This line then creates the **CAEngine** object.

```
adeEngine = New CAEngine
```

The **adeEngine** variable now holds our in-process **CAEngine** object.

If we want to use the local (out-of-process) server version of ADE, we can add a reference to the project to the **Analytica Decision Engine Server 4.1 COM** component and change the top line from `Imports ADEW` to `Imports ADE`.

Here is another way to obtain a new **CAEngine** object. This sequence does not require adding a reference to the project.

```
adeEngine = CreateObject("ADEW4.1.CAEngine") ' in-process
adeEngine = CreateObject("ADE4.1.CAEngine") ' out-of-process
```

To understand the pros and cons of using an in-process server versus as out-of-process (or local) server, and which automation server to use for different scenarios, see “In-process vs. out-of-process” on page 22, as well as other books related to COM servers.

COM vs. Automation interface

In the example above, we used a COM interface to call ADE. In a COM interface, the object (**CAEngine** in this case) is declared as **CAEngine**, and the compiler resolves each member function and can detect several obvious errors at compile time. In addition, Visual Studio can provide a list of methods and parameter types as tool tips as you program, which is helpful when writing programs that use ADE. COM calls are slightly faster than Automation calls, but the speed difference is not usually significant in applications of ADE. With ADE 4.1, we recommend using the COM interface if your programming language supports it.

In VB Automation, you can declare an object simply as **Object**, rather than a more specific types such as **CAEngine**, **CAObject**, and so on. When **ADE** methods are called using Automation, the methods are resolved at run time. At compile time, the compiler does not know whether your **ADE** object has a function named **OpenModel**. In VB, the syntax for calling a COM method or an Automation method is identical — the only difference is whether the object's type is declared explicitly.

In VC++ and C#, the syntax for calling COM is not the same as for Automation. In these cases, COM is much more convenient, while Automation can get rather tedious. However, some languages, including VBScript and other scripting languages, support only Automation and not COM.

Opening a model with ADE

We will now open the `Txc.ana` model, and show the main window of our application. Use the following call:

```
theModelString = adeEngine.OpenModel(theModel)
frmMain.DefInstance.Show
```

The **OpenModel** function of **CAEngine** opens the model. If successful, the variable `theModelString` contains the name of the model. Otherwise, it contains an empty string. Although we haven't done so in this example for the sake of brevity, you should check to see that the string returned from **OpenModel** isn't empty. If it is, there was an error in opening your model. You can find out what kind of error with the **ErrorCode** and **ErrorText** properties of **CAEngine** (`adeEngine.ErrorCode` and `adeEngine.ErrorText`). We will see how to use these two properties later on. For a listing of all the error codes, see Appendix A, "Error Codes" on page 70.

Retrieving objects from the Analytica model

The next step is to retrieve objects (variables, modules, functions, etc.) from our model, so that we can access their attributes (definition, title, class, etc.). Our example model (`Txc.ana`) manipulates the **Pop_exp** and **Cost** objects. In particular, it modifies **Pop_exp** to see how this effects the **Cost** object.

The **PrintAttributes** function in the file `frmMain.frm` of our `TxcTest.vbproj` (`TxcTest.sln`) project shows how to do this. This function is first called by the **Form_Load** function of `frmMain.frm`, when the application starts, to display the **Cost** table. It is also called whenever we wish to print out the current result of our **Cost** table. The function looks like this:

```
Public Sub PrintAttributes(ByRef inputIdentifier As String, ByRef
    outputIdentifier As String)
    Dim inputObject, outputObject As CObject
    Dim resultTable As CTable
    Dim definitionAttrInput As String
    inputObject = adeEngine.GetObjectByName(inputIdentifier)
    outputObject = adeEngine.GetObjectByName(outputIdentifier)
    definitionAttrInput = inputObject.GetAttribute("definition")
    resultTable = outputObject.ResultTable
    Call PrintResultTable(resultTable, inputIdentifier,
        definitionAttrInput, outputIdentifier)
    ReleaseComObject(resultTable)
    ReleaseComObject(inputObject)
    ReleaseComObject(outputObject)
End Sub
```

PrintAttributes works with the variable identifiers **Pop_exp** passed as parameter `inputIdentifier` and **Cost** passed as parameter `outputIdentifier`. It fetches the corresponding objects using the **GetObjectByName** function of **CAEngine** as follows:

```
inputObject = adeEngine.GetObjectByName(inputIdentifier)
outputObject = adeEngine.GetObjectByName(outputIdentifier)
```

If **GetObjectByName** succeeds, it returns an object of type **CAObject**. You then use the functions of **CAObject**. See "SendCommand(command)" on page 54 for a listing all **CAObject** functions. If **GetObjectByName** fails, the return value is **Nothing**. The code should check to make sure that the result from **GetObjectByName** is valid. If not, use the **ErrorCode** and **ErrorText** properties of **CAEngine** to get more information about the error. For example:

```
Set inputObject = adeEngine.GetObjectByName(inputIdentifier)
```

```

If inputObject Is Nothing Then
    MsgBox("This error from GetObjectByName occurred: "& _
        vbCrLf & adeEngine.errorCode & ":" & adeEngine.errorText)

Else
    'inputObject valid
End If

```

Getting object attributes

Each Analytica object has a set of attributes (analogous to properties), such as identifier, title, description, and class. You can use the **GetAttribute** function to obtain an attribute from an Analytica object. For example, to get the definition of **inputObject** (currently, the cost):

```
definitionAttrInput = inputObject.GetAttribute("definition")
```

In the **Txc.ana** model, the definition of **Pop_exp** is **"Normal (30M, 3M)"** which we store in **definitionAttrInput**.

Evaluating objects and retrieving results

Use **Result** or **ResultTable** methods of **CAObject** to get the value of a variable. ADE automatically evaluates the variable first, if necessary. Use the **Result** method if you are sure the result will be **atomic**, i.e., a single element. Otherwise, use **ResultTable**, which retrieves the result as an **array**. An atomic result is treated as a special case of an array, one with zero dimensions. If the value is atomic, the method **AtomicValue** returns its single value as a number or string.

By default, **Result** and **ResultTable** return the **mid** value of the result, i.e., the result of ADE evaluating it as deterministic. For a probabilistic value, set the **ResultType** property of **CAObject** to the desired uncertainty view — Mean, Sample, PDF, CDF, Confidence bands, or Statistics (see "ResultType" on page 57 for details). We get the value of **outputObject** like this:

```
resultTable = outputObject.ResultTable
```

The result is a **CATable** object, which lets us access individual elements in a table.

If you call **Result** to get an array (or table) value, it returns the array as a string, listing the indexes and elements separated by commas. It is usually easier to use **ResultTable**, so that you don't have to parse elements of the table from the string.

Getting the index elements of a table

An Analytica table has zero or more indexes. If it has one index, then it is one-dimensional; if it has two indexes, it is two-dimensional, and so on. A zero-dimensional table holds a single **atomic** (or **scalar**) value. You can use the **NumDims** function of **CATable** to get the number of dimensions (same as number of indexes) of a table. To get at the individual indexes of a table, use methods **IndexNames** and **GetIndexObject** of **CATable**.

The function **PrintResultTable** in **frmMain.frm** shows the use of these two functions. **PrintResultTable** is called from **PrintAttributes**, and does the actual work of printing the table that shows up in our **TestTxc** application (for brevity, we show only the parts of this function related to ADE).

```

Public Sub PrintResultTable(ByRef resultTable As CATable,
    ByRef inputIdentifier As String,
    ByRef definitionAttrInput As String,
    ByRef outputIdentifier As String)

    Dim theIndexName, theTableName As String
    Dim theIndexElement As String
    Dim theTableElement
    Dim theIndexObj As CAIndex

```

```

Dim numEls As Integer
Dim spaces, i As Integer
Dim lenStr As Short
Dim OutputStr As Short
Dim spaceString, underlineString As String
...
theIndexName = resultTable.IndexNames(1)
theTableName = resultTable.Name
theIndexObj = resultTable.GetIndexObject(theIndexName)
numEls = theIndexObj.IndexElements
...
For i = 1 To numEls
    theIndexElement = theIndexObj.GetValueByNumber(i)
    theTableElement = resultTable.GetDataByElements(i)
    ...
Next i
InformationPane.Text = outputString
End Sub

```

The lines of **PrintResultTable** that get an index of a table are as follows:

```

theIndexName = resultTable.IndexNames(1)
theIndexObj = resultTable.GetIndexObject(theIndexName)

```

We get the name of first index using the **IndexNames** function of **CATable**. We pass it into the **GetIndexObject** function of **CATable** to get a **CAIndex** object that represents our index. This automation object returns information about its corresponding index. If this function fails, it returns **Nothing**. In that case, use **ErrorCode** and **ErrorText** functions of **CAEngine** to find out why.

Getting information from CATable and CAIndex

PrintResultTable also shows how to get information from **CATable** and **CAIndex** objects. This code gets the index and table elements of the **Cost** table:

```

numEls = theIndexObj.IndexElements
For i = 1 To numEls
    theIndexElement = theIndexObj.GetValueByNumber(i)
    theTableElement = resultTable.GetDataByElements(i)
    ...
Next i

```

The **IndexElements** property of **CAIndex** returns the number of elements in the (first) index. The **GetValueByNumber** function of **CAIndex** gets individual index elements.

To get the individual table elements of the **Cost** table object, **resultTable**, we use the **GetDataByElements** function of **CATable**, passing in the coordinates of the element in the table.

When we retrieve an individual element of our **CATable** object (**resultTable**), we take advantage of the fact that the table is one-dimensional. Therefore, we only need to pass **GetDataByElements** a single number representing the position in our table. If we were dealing with two or more dimensions, however, we would need to pass **GetDataByElements** an array specifying the coordinates of the element of our table to retrieve. So, if we want to retrieve the element at position (4,3) of a two-dimensional table, we would write:

```

Dim W as Variant 'return element
Dim IndexPtrs(1 To 2) As Variant 'position in table
...

```

```

IndexPtrs(1) = 4
IndexPtrs(2) = 3
W = resultTable.GetDataByElements(IndexPtrs)

```

Controlling formats of atomic values

Each atomic value in a **CATable** can be a number, string, or one of a few other basic types (e.g., Null, Undefined, Reference, or Handle). These are returned as **variants**, a data structure understood by Visual Basic, specifying the type and value. The **RenderingStyle** property of **CATable** controls how the underlying Analytica value is mapped to the Visual Basic variant.

For example, it can return a numeric value as a number, or a string using the Analytica model's number format setting. If it is formatted, an option controls whether to truncate the number of digits or to return it with full precision.

In the **PrintResultTable** subroutine, located in **frmMain.vb**, the rendering style is explicitly specified:

```

resultTable.RenderingStyle.NumberAsText = True
resultTable.RenderingStyle.FullPrecision = False
resultTable.RenderingStyle.StringQuotes = 2

```

The first line specifies that numeric values should be formatted as text according to the number format associated with the result object. For example, in the program output, we see **30.103M** instead of **30102995.6639812**, which would likely be displayed if we had let Visual Basic concatenate the numeric value to our result string. In the event that a string-valued cell occurs in the result, it returns with explicit double quotes around the value. See “Class **CARenderingStyle**” on [page 66](#) for additional properties available through the **CARenderingStyle** object.

Other ways to access tables

There are several ways to access the elements of a multi-dimensional **CATable**. Some might be more convenient in certain scenarios than others.

The first way is to use the **GetDataByElements** or **GetDataByLabels** methods of **CATable**, shown in the code example above. In this case, you supply the coordinates of the cell whose atomic value you wish to retrieve.

A second way is to use the **Slice** or **Subscript** methods of **CATable** to obtain a new **CATable** object having one less dimension. By repeatedly reducing the dimensionality, you eventually reach zero dimensions, in which case you have a single atomic value. At that point, the **AtomicValue** method of **CATable** returns this value. The **AtomicValue** method is the only way to access a scalar value (since it doesn't have a coordinate). You must use this method if you need to generate a graph image of a slice of the full result.

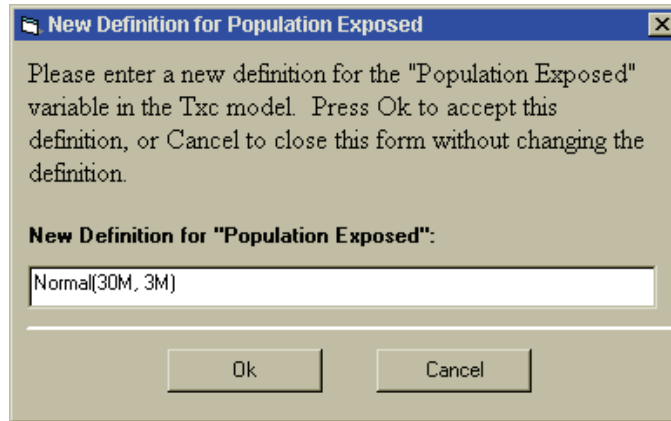
A third way is to use the **GetSafeArray** method of **CATable**, to convert the multi-dimensional array into a **safe array** (or into a .NET array). You can then manipulate the multi-dimensional array directly in VB or other .NET language. Since there is no inherent ordering to Analytica dimensions, but safe arrays and .NET arrays have an explicit ordering, you must first use the **SetIndexOrder** of **CATable** to specify the ordering of dimensions before calling **GetSafeArray**. Note that this is not necessary if you know that your array is one-dimensional.

Modifying objects

A custom application often gets input from a user or other external source to transfer into input variables in the Analytica model. You can do this either by setting the definition of an input variable, or by using a definition table.

TestTxc shows how to modify the definition of **Pop_exp**, which is a model input that effects the **Cost** result variable. To set the definition in the example, select **File > Change Population Exposed** from the main menu. A dialog appears, as shown in .

Redefining the variable definition



Enter a new definition into the field and click **Ok**. The main window displays the new value of **Cost**. The **OkButton_Click** function in **ChangeDef.frm** is called when the **Ok** button is clicked in the dialog. It modifies the definition of **Pop_exp**, and then calls the **PrintAttributes** function that prints the result of **Cost**.

The function looks like this:

```
Private Sub OkButton_Click(ByVal eventSender As System.Object,
    ByVal eventArgs As System.EventArgs) Handles OkButton.Click
    Dim errorText As String
    Dim pop_exp_Object As CAObject
    Dim errorCode As Short
    Dim errorString As String
    newDefinition = PopExposedDef.Text
    pop_exp_Object = adeEngine.GetObjectByName("pop_exp")
    pop_exp_Object.SetAttribute("definition", newDefinition)
    errorCode = adeEngine.errorCode
    If errorCode <> 0 Then
        MsgBox("This error occurred while processing your definition:
" &
            vbCrLf & vbCrLf & adeEngine.errorText)
        PopExposedDef.Focus()
    Else
        Me.Close()
        frmMain.DefInstance.PrintAttributes("Pop_exp", "Cost")
    End If
    ReleaseComObject(pop_exp_Object)
End Sub
```

This function grabs the new definition typed into the **New Definition for Population Exposed** field and sets it to the **Pop_exp** object by using the **SetAttribute** function of **CAObject** object. It then calls **PrintAttributes**, which evaluates the **Cost** object, and prints the new table.

To set a new definition for the **pop_exp** variable, we get the **CAObject** for **Pop_exp**, and set its definition to the definition typed in by the user. This is done with the following code:

```
pop_exp_Object = adeEngine.GetObjectByName("pop_exp")
pop_exp_Object.SetAttribute "definition", newDefinition
```

Whenever you call **SetAttribute**, you should check the **ErrorCode** of the **CAEngine** automation object (**adeEngine**), in case the definition is illegal.

Try entering a new definition such as **Uniform(25M,35M)** and click **Ok**. When the definition of **pop_exp** is changed, the result for **Cost** gets recomputed by ADE when **ResultTable** is next called for the **Cost** variable (when the application window is repainted).

Graphing with ADE

Using the same graphing engine used by Analytica 4.1, you can generate a chart or graph to display an array-valued or uncertain result. In ADE 4.1, you can use the **GraphToFile** and **GraphToStream** methods of **CATable**. The graphs are returned in several possible image formats, such as **image/bmp** or **image/jpeg**.

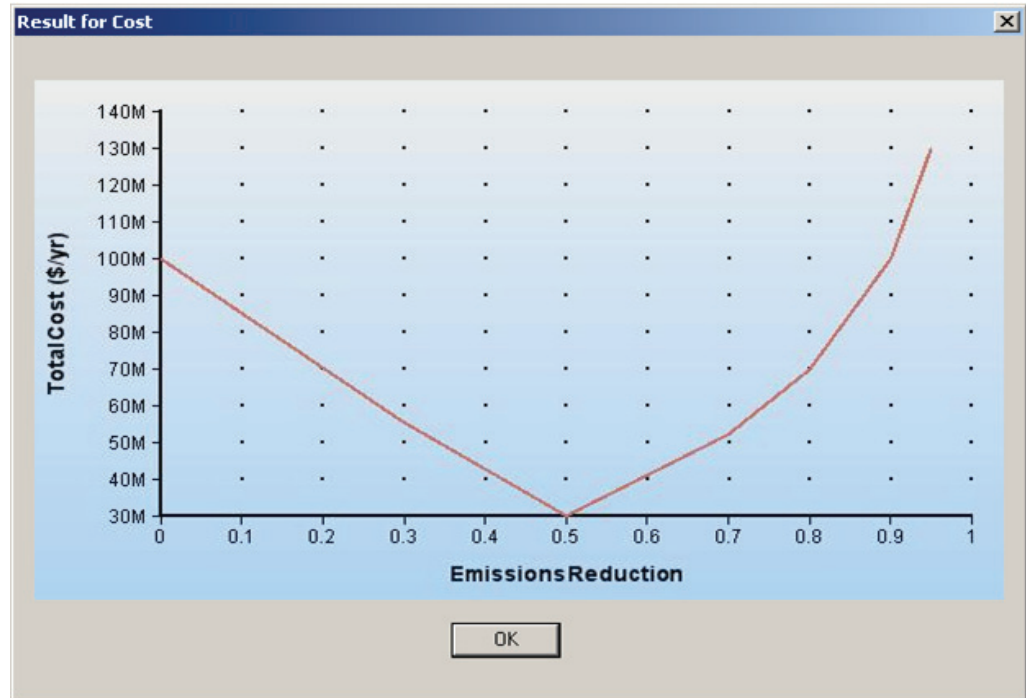
The easiest way to select from the available graphing options is to open your model with Analytica Enterprise. You can experiment with the settings for the various defaults or override selected variables to see how they look. When you've chosen the settings you want, save the model. ADE then uses these settings when producing result graphs for each variable.

For higher-dimensional results, some work might be necessary to select the slice of the result that will be plotted and the specific **pivot** (i.e., which dimensions will appear on the X-axis versus in the key). The **Subscript** or **Slice** methods of **CATable** can be used to select the particular slice to be plotted and **SetIndexOrder** can be used to control the pivot. See "Class CATable" on page 60 for details. In our Txc example, we have a one-dimensional result (**Cost**), and do not need to worry about slicing or pivoting.

Comment: The **SetIndexOrder** method for controlling the pivot is subject to change in the near future. The paragraph above will need to be revised.

In the example, the **GraphToStream** method is used to transfer the graph image directly from ADE to a user-interface method. **GraphToStream** is a bit more complicated to use than **GraphToFile**, since **GraphToFile** requires little more than a file name to write the image to. To use **GraphToStream**, we must set up a stream in memory, allow ADE to write to that stream, and then reconstitute the image from that stream. Because .NET streams are not compatible with COM streams, you need to use the **StreamConnector** class provided with ADE. The **GraphResult_Click** routine in **frmMain** shows the use of **GraphToStream**. Select the **Graph Result** menu option from the main application window, and the results appear in a graph as shown in .

Result graph



Conclusion

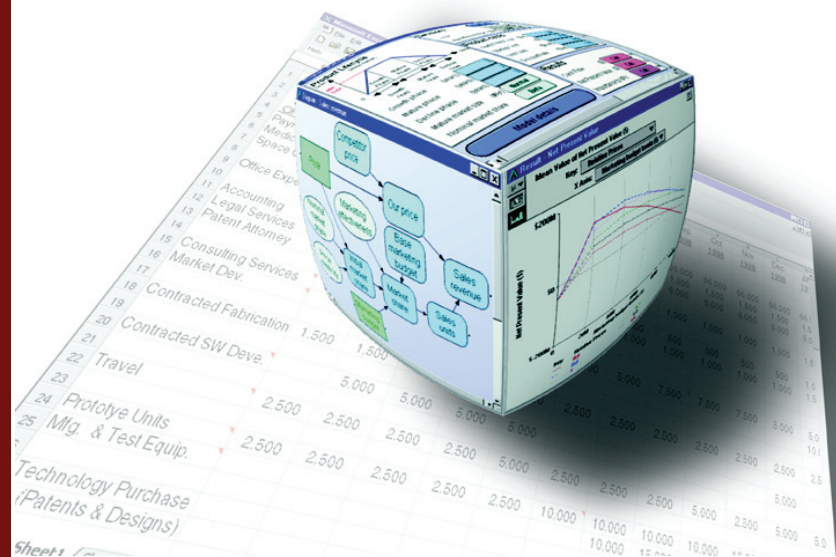
In this tutorial, we introduced several important aspects of the Analytica Decision Engine. We saw how to create the ADE server object, open a model with ADE, get at an individual object in a model, evaluate objects, access elements in a table, and modify objects in a model. But, ADE can do a lot more!

We hope that you have learned enough about the basics so that you can now explore the more advanced features on your own. We recommend that you now read the rest of this guide to learn about what ADE can do.

Chapter 3

Using the Analytica Decision Engine Server

This chapter describes the Analytica Decision Engine server classes **CAEngine**, **CALicense**, **CAObject**, **CATable**, **CAIndex**, and **CARenderingStyle**, and the server class architecture.



ADE classes

ADE uses the following six classes:

- The **CAEngine** class contains methods and properties that allow you to open and close existing models, create new models, create new Analytica objects, and access Analytica objects contained in your model.
- The **CALicense** class contains methods that allow you to instantiate a **CAEngine**, to use a special application license code, to examine certain restrictions on your ADE license, and to access details about why a **CAEngine** failed to instantiate.
- The **CAObject** class contains methods and properties that allow you to set and obtain information about the Analytica objects (such as variables or modules) that you obtain from the **CAEngine** class.
- The **CATable** class is used to examine multi-dimensional results or to view and modify multi-dimensional definition tables (also called **edit tables**).
- A **CAIndex** object provides access to one dimension of a multi-dimensional **CATable**.
- The **CARenderingStyle** class is new to ADE 4.0, and allows you to control or alter the format in which ADE returns values.

Note: “CA” in these class names stands for “Class Analytica.”

The following sections describe how to access these Analytica Server objects from Visual Basic or C#.

Server class architecture

COM, Automation, and .NET

ADE 4.1 supports two calling conventions: COM and ActiveX Automation. COM is an early-binding convention in which the methods and data types are resolved when your application code is compiled. Automation is a late-binding convention where method calls are resolved at run time. The COM convention is somewhat more efficient, although for most applications, the difference in efficiency is far overshadowed by the time required to compute your model's results.

In Visual Basic, the syntax for calling a method using COM or Automation is identical, and which interface is used depends on how you declare your objects. In other languages, such as C# or C++, the method of invocation can look quite different. In C# and C++, it is generally more convenient to use the COM interface. VBScript (used by the Windows Scripting Host and older versions of IIS ASP) supports only the Automation interface.

The COM interface can be used transparently from a .NET environment such as Visual Studio 2005. The .NET programming environment wraps COM objects with a .NET Interop object, which gives ADE interfaces the appearance of being .NET interfaces.

In ADE 3.1 and before, the Automation interface was the recommended convention; however, with the ADE 4.1 release, we now recommend the COM interface unless this is not an option in your programming environment (such as VBScript).

In-process vs. out-of-process

ADE can be launched either in-process or out-of-process. When launched in-process (ADEW), the **Adew.dll** library is loaded into your application's process space. When launched out-of-process (ADE), the **ADE.exe** server is launched and runs in a different process. Both types of server use the same class interfaces, so the choice of which type of server to use can usually be changed by editing a single line of code, i.e., the line that instantiates the **CAEngine**.

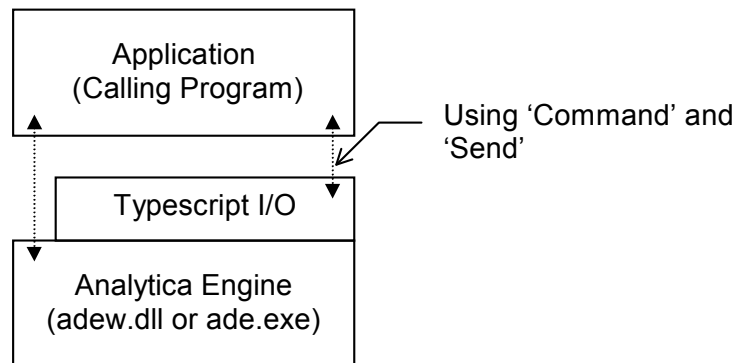
In-process servers have a slight performance advantage, but come with several restrictions. First, the **apartment** threading model of ADEW must be compatible with your application's threading model. For example, The Microsoft IIS web server (IIS 5.0 or later) does not allow you to use an apartment-threaded component under its default settings. Also, you are restricted to have only one **CAEngine** instance (and thus, only one model) in memory at any one time.

Out-of-process instances of ADE run in a different process, and can be configured to run on a different computer from your application. Because data must be “marshaled” across process boundaries, it is less efficient, but it is far more flexible than the in-process server. Your program can make use of multiple simultaneous instances of ADE, each with a separate model instance loaded. As such, the out-of-process server is almost always preferred for web applications because you can have one ADE instance for each session.

Typescript

In addition to the program interface, ADE has a fully functional command interface, known as the **typescript** language. This language is described in the **Analytica Scripting Guide**. This language allows access to all of ADE’s functionality. The API provides a more convenient, object-oriented set of functions for communication with the engine from Visual Basic and C++ applications. A calling program can use the API functions, or it can pass typescript commands directly to the typescript interface.

The Analytica Decision Engine architecture



Security permissions under IIS 5

When creating a web application that uses ADE from within Microsoft’s Active Server Pages (ASP/ASPX) under Internet Information Server (IIS), you might need to configure permission settings in order to instantiate and access the ADE COM component from your program.

When creating a web application or web service, you should use the out-of-process ADE server. When your ASPX application is executed while serving a web page request, the ADE COM component is launched and accessed from a special internal Windows account name. Even though your programs can create and access ADE when run under your account, the same access might not exist for ASP or ASPX programs. To configure security permissions so that your ASPX application can use ADE, follows these steps:

1. From the Windows Control Panel, select **Administrative Tools > Component Services**.
2. In the **DCOM Config** folder, locate “Analytica Decision Engine Local Server 4.1.”
3. Select **Properties** from the right mouse menu, and select the **Security** tab.
4. Set **Launch and Activation Permissions** to **Customize**, then click **Edit**.
5. For the user {computer_name}\ASPNET, grant local launch and local activation permissions.
6. Save these settings. You might need to reboot of the machine to finalize these changes.

When these permissions are not properly configured, a “security exception” occurs on the line of your program that attempts to instantiate the **CAEngine**.

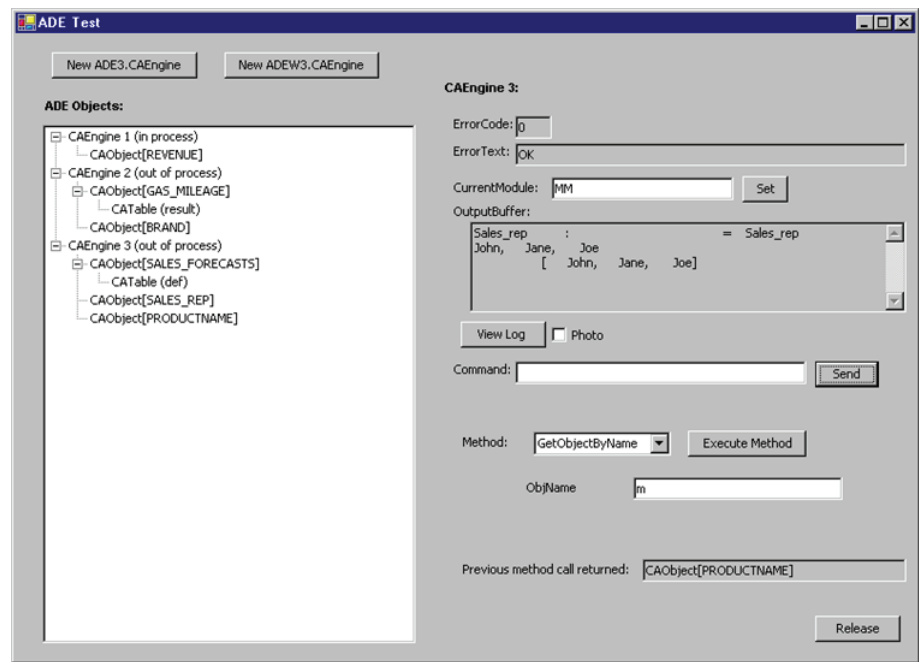
The AdeTest program

ADE 4.1 ships with a sample program called **AdeTest.exe**. The executable can be found in the **Examples/AdeTest/bin** directory. You can use **AdeTest** to exercise the functionality of either the in-process (**Adew.dll**) or the local process (**ADE.exe**) versions of ADE 4.1. Using **AdeTest**,

you can send script commands to the engine, create ADE objects, and set or call virtually any of the properties and methods of the ADE objects. If you have Visual Studio 2005 installed, you can step through the code in the Visual Studio Debugger to observe the methods being called.

shows the AdeTest program dialog. The left-hand pane shows a list of ADE objects that the program is currently holding. The right side shows details of one of those objects. In the figure, there are three **CAEngine** instances, each with a different model open. The first **CAEngine** is an in-process (**Adew.dll**) instance, while the second two are out-of-process local servers (**ADE.exe**) instances. The two buttons above the left pane can be used to create additional **CAEngine** instances, while the **Release** button at the lower-right corner of the right-hand panel releases an instance. The right-hand panel shows information about the third **CAEngine** instance. The current values for the **CAEngine** properties **ErrorCode**, **ErrorText**, **CurrentModule**, **OutputBuffer**, and **Photo** are displayed. You can execute a typescript command by typing the command into the text box and clicking the **Send** button. Or you can execute any of the method of **CAEngine** by selecting the method in the drop-down **Method** box, filling in the parameters, and clicking the **Execute Method** button.

ADE Test dialog



If you click an object in the left-hand pane, the properties for that object are displayed on the right-hand side and you can set its properties or call its methods. Thus, you can simulate a series of steps your program might execute through the graphical interface.

When a method returns an object, for example, as with **CAEngine::GetObjectByName**, the returned object is added to the tree on the left as a child of the object that created it. After executing a method from a class other than **CAEngine**, it is a good idea to glance at the corresponding **CAEngine**'s panel to check the **ErrorCode**, **ErrorText**, and **OutputBuffer** properties.

The **Photo** checkbox in the Analytica window is mirrored by the **Photo** property of the **CAEngine** class. By default the **Photo** property is False, so typescript communications between the client and ADE are not copied to the Analytica log window. Setting the **Photo** property to True copies all subsequent typescript communications between the client and ADE. In Visual Basic, this would be done as follows:

```
ADE.Photo=True
ADE.Photo=False
```

Turning on the **Photo** property significantly slows down communication with ADE.

Sample application in Excel's Visual Basic

Another example program called `excel_exam` is also included in the ADE package. The program, `Analytica.xls`, in the `excel_exam` directory can be loaded into Microsoft Excel and executed as a macro. This program demonstrates the use of Visual Basic for Applications in Excel for ADE communications. This sample makes use of the local server version of ADE.

Sample ASP web application

The example in `asp_exam` demonstrates the use of ADE from an Active Server Pages web application. This application produces a hierarchical outline of your model structure in HTML. The `readme.txt` file in that directory contains instructions for configuring the web server to run the example.

When using Microsoft's ASP, we recommend that you use the local server. By using the local server (`ADE.exe`), you can ensure that each web application, or even each session, uses a different version of `ADE.exe`. Currently, there is a limitation in ADE that prevents creation of two or more in-process server objects at the same time. Therefore, if you expect to have more than one session of ADE active at one time (as is almost always the case in web-based applications), always use the local server of ADE.

Using the ADE COM interface

From a .NET project in Visual Studio 2005

From a Visual Basic, C#, J#, ASP.NET, or C++/CLR project in Visual Studio 2005, you gain access to ADE by adding a reference to it in your project. The same technique holds with slight variations in older (pre-.NET) versions of Visual Basic and several other non-Microsoft development environments.

In Visual Studio 2005, select **Add reference** or **References** from the **Project** menu, and in the dialog that appears, select the **COM** tab (in VC++ you need to click the **Add new reference** button to get to the **COM** tab). In the list of components, locate and select one of the following:

Analytica Decision Engine Local Server 4.1
Analytica Decision Engine Server 4.1

For out-of-process `ADE.exe` servers, select the **Local** server. To use `Adew.dll`, select the (non-local) server. It is also possible to add both references into a project (the `Adetest` example does this), although the need for this would be rare.

The ADE classes are exposed in the name space ADE or ADEW for the local server and in-process server, respectively. For convenience, you can add a `using` declaration to the top of your source files, like this:

```
Imports ADE      ' Visual basic
using ADE;       // C#
using namespace ADE; // C++/CLR
import ADE.*;    // J#
```

Of course, when using the in-process server you would type `ADEW` in place of `ADE` above. These declarations allow you to refer to `CAEngine`, `CAObject`, etc., in your code, rather than `ADE.CAEngine`, `ADE.CAObject`, etc., which makes it easy to convert from the local to the in-process ADE server should the need arise.

To begin using ADE, you need to obtain a first instantiation of **CAEngine**. This is done with one of the following lines:

```
dim ADE as CAEngine = new CAEngineClass      ' VB
CAEngine ADE = new CAEngineClass();          // C#, J#
CAEngine^ pAde = gcnew CAEngineClass();      // C++/CLR
```

CAEngine is the name of a particular abstract interface, while **ADEW.CAEngineClass** and **ADE.CAEngineClass** are the names of two particular object classes that implement that interface. The **CAEngineClass** object is the only object that you can create directly; all other ADE object instances are obtained by calling methods on existing objects.

To keep the use of the COM interface, always declare your variables with the class names **CAEngine**, **CAObject**, **CATable**, **CAIndex**, and **CARenderingStyle**. Avoid assigning object instances to variables declared as **System.Object**. This allows the compiler to perform early binding and type checking.

Releasing objects in .NET

In pre-.NET Visual Basic and scripting languages, the programming environment automatically ensures that COM objects are released immediately. This is not the case in VB.NET, ASP.NET, or other .NET programs. From .NET, it is important that your program explicitly releases each COM object when it is through with it. Setting a pointer to Null (or Nothing) is not sufficient, since the actual release doesn't occur until the next garbage collection.

To release a COM object from a .NET program, you need to execute code similar to the following (C# syntax shown):

```
System.Runtime.InteropServices.Marshal.ReleaseComObject(ADE);
ADE = null;
```

Releasing objects in this fashion is especially important when you are using an out-of-process COM server (e.g., **ADE.CAEngine**). In this case, the memory resources are predominantly consumed in the ADE process, not in your program's process. This can cause the ADE process to run out of memory before your program's process uses enough memory to cause an automatic garbage collection to occur. From a .NET-based web application, old **ADE.exe** processes linger long after a session has finished unless you explicitly release the **CAEngine** object.

This need to release COM objects is not unique to ADE. You must take care to release any COM object, including those provided by Microsoft, especially when those COM objects are out-of-process.

Because of this absence of deterministic destruction in .NET, it can be tedious to ensure that every COM object is released. Therefore, you might want to occasionally force an explicit garbage collection in your code, which releases all unused objects. This can be accomplished by calling:

```
System.GC.Collect();
```

From an ATL project in C++

To use ADE 4.1 from a non-.NET C++ project, place the following two lines at the top of your source file:

```
#import "ADE.exe"
using namespace ADE;
```

Or to use the in-process server, use these lines:

```
#import "Adew.dll"
using namespace ADEW;
```

You need to include the ADE home directory in your include path in the project settings, or spell out the complete path in the **#import** declaration.

Next, obtain the first instance to an ADE engine using this code:

```
CoInitialize(NULL);
CAEnginePtr pAde(__uuidof(_CAEngine));
.
.
.
CoUninitialize();
```

Colnitalize() is a Windows system call that is required before the COM system can be used.

If your project spans multiple code files, use this in each of your source files (or once in `stdafx.h`):

```
#import "ADE.exe" no_implementation
```

And then in one file only (e.g., `stdafx.cpp`), include this line:

```
#import "ADE.exe" implementation_only
```

Using the ADE Automation interface

VBScript is an example of a scripting language, usable from Windows Scripting Host (**CScript.exe** or **WScript.exe**), pre-.NET versions of Active Server Pages, Internet Explorer, and so on. JScript is another, and many other scripting OLE-Automation compliant scripting languages are available including Perl.

These scripting languages support ActiveX Automation scripting, but not COM interfaces. Using the Automation interface, ADE can be used from these, often with no additional tools beyond a simple text editor.

For ADE releases prior to 4.1, the Automation interface was the preferred convention to use. For languages that support direct COM calls, the COM convention is now recommended in ADE 4.1. Using Automation from C++ or C# is rather tedious and not covered here.

From Visual Basic or VBScript

To use the Automation interface, it is not necessary to add a reference to your Visual Basic project. The syntax here is similar in other scripting languages. In Visual Basic, the code to instantiate a **CAEngine** is:

```
dim ADE as Object
ADE = CreateObject("ADE4.CAEngine")
```

In VBScript, and some older versions of Visual Basic, the **set** keyword is required:

```
dim ADE
set ADE = CreateObject("ADE4.CAEngine")
```

For the in-process server, you send the parameter **ADEW4.CAEngine** to the **CreateObject** call.

ADE typescript: command language communication

The **Command** property and **Send** method of the **CAEngine** class allow you to use typescript commands, sent as ASCII strings to the engine, and receive the resulting output as another ASCII string. You might want to use a typescript command instead of an API method if:

- You want to perform your own parsing on ADE output (e.g., on tabular data that are output from the Analytica Decision Engine as text strings of comma-delimited text).
- No appropriate API method exists.

You perform these steps to send a typescript command to ADE:

1. Assign a text string containing the command to the **Command** property of your **CAEngine** object.

2. Use the **Send** method to send the command to the Engine. If the **Send** method returns **True**, then the command was processed without error by ADE.
3. Store the error code and error text (if the return code is nonzero). These two pieces of information are stored in the **CAEngine** properties **ErrorCode** and **ErrorText**.
4. Get the output by calling the **OutputBuffer** function in the **CAEngine** class.

Note: You can also combine the first two steps by calling `CAEngine.SendCommand(cmd)`.

These steps are demonstrated below for various programming languages. After these simple examples, subsequent examples are given using a Visual Basic syntax, but you should have no problem extrapolating the syntax to your language of choice.

In Visual Basic

```
Imports ADE

Module Module1
Sub Main()
    Dim Result, ErrT As String
    Dim ErrCode as Integer

    dim ADE as CAEngine = new CAEngineClass
    ADE.Command = "news" 'any typescript command
    dim SendCode as Boolean = ADE.Send
    If SendCode = False Then
        ErrCode = ADE.ErrorCode
        ErrT = ADE.ErrorText
    Else
        Result = ADE.OutputBuffer
    End If
End Sub
End Module
```

In VBScript

```
set ADE = CreateObject("ADE4.CAEngine")
ADE.Command = "news"
If ADE.Send = False Then
    ErrCode = ADE.ErrorCode
    ErrT = ADE.ErrorText
Else
    Result = ADE.OutputBuffer
End if
```

In C#

```
using System;
using ADE;
namespace ADE_from_Csharp
{
    class Program
    {
        static void Main()
        {
            String errT, result;
            int errCode;
            CAEngine ADE = new CAEngineClass();
            ADE.Command = "news";
            if (!ADE.Send()) {
                errCode = ADE.ErrorCode;
```

```

        errT = ADE.ErrorText;
    } else {
        result = ADE.OutputBuffer;
    }
}
}

```

In J#

```

import ADE.*;
public class Program
{
    public static void main( )
    {
        String errT, result;
        int errCode;
        ADE.CAEngine ADE = new ADE.CAEngineClass();
        ADE.set_Command("news");
        boolean sendRes = ADE.Send();
        if (!sendRes) {
            errCode = ADE.get_ErrorCode();
            errT = ADE.get_ErrorText();
        } else {
            result = ADE.get_OutputBuffer();
        }
    }
}

```

In C++/CLR

```

using namespace System;
using namespace ADE;
void main( )
{
    String ^result, ^errT;
    int errCode;
    CAEngine^ ADE = gcnew CAEngineClass();
    ADE->Command = "news";
    if (!ADE->Send()) {
        errCode = ADE->ErrorCode;
        errT = ADE->ErrorText;
    } else {
        result = ADE->OutputBuffer;
    }
}

```

In VC++ (without .NET)

```

#import "ADE.exe"
using namespace ADE;
void main( )
{
    CoInitialize(NULL);
    _bstr_t errT, result;
    int errCode;
    _CAEnginePtr pAde(__uuidof(_CAEngine));
    pAde->Command = "news";
    if (!pAde->Send()) {
        errT = pAde->ErrorText;
    }
}

```

```
        errCode = pAde->ErrorCode;  
    } else {  
        result = pAde->OutputBuffer;  
    }  
    CoUninitialize();  
}
```

Errors and error handling

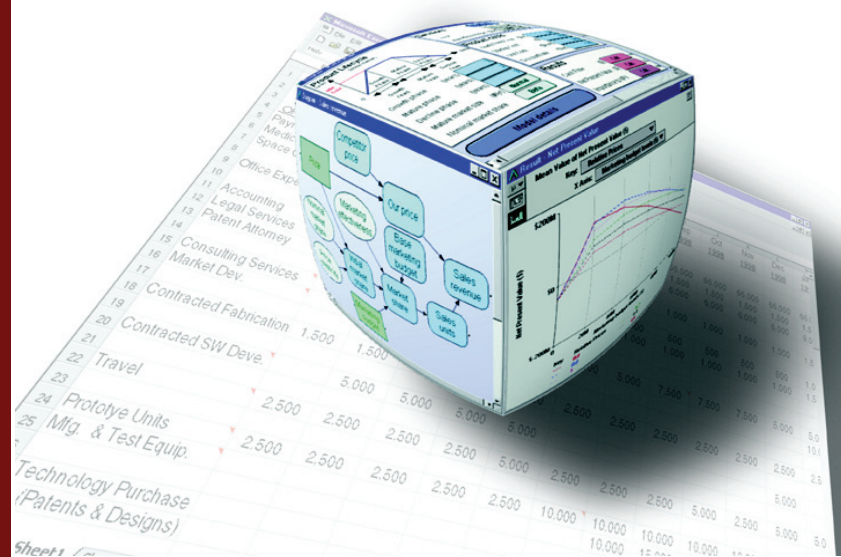
The **CAEngine** properties **ErrorCode** and **ErrorText** should be queried after any operation with ADE whenever an error is possible. Reading a value of a property from an ADE object does not change the error code. Setting the value of a property might result in an error, usually indicating an illegal value for that property. All method calls reset **ErrorCode** to zero if there is no error, or to a value indicating the error.

To get additional information on an error, check the **OutputBuffer** property of **CAEngine**. Any error messages that a user of Analytica would have seen appear in the output buffer.

Chapter 4

Working with Models, Modules, and Files

This section contains examples of common operations and manipulations you might perform on objects in your Analytica model.



Models and modules

Note: In VBScript, VBA, and pre-dot-NET versions of Visual Basic, the **Set** keyword was necessary when assigning an object to a variable. In VB.NET, the **Set** keyword is no longer necessary. The **Set** keyword is not used in the examples below.

- To create a new model:

```
If ADE.CreateModel("NewModelName") Then
    'Model successfully created
End If
```

The **CreateModel** method only requires one parameter, a string containing a model name.

- To open an existing Analytica model:

```
Dim ModName as String
ModName = ADE.OpenModel("C:\ ... \Anamodel.ana")
If ModName="" then
    ' Handle Error condition here
End if
```

If a model has already been opened, that model is closed automatically before the new model is created. If the specified filename is not legal, **OpenModel** returns an empty string. In that case, use the **ErrorCode** property of **CAEngine** to determine the cause of the error. Be aware that an **ErrorCode=2** warning is often returned even though the load is successful. For full details as to what has caused an error or warning, use the **OutputBuffer** property of the **CAEngine**. You must use the backward slash (\) for the path delimiter when using ADE. It does not support the forward slash (/).

- To add a module from a file to the currently open model:

```
Dim Merge as Boolean = True
Dim ModName as String
ModName = ADE.AddModule ("C:\...\MyLibrary.ana", Merge)
if ModName="" Then
    ' Handle error conditions here
End if
```

The **FileSpec** parameter should contain the path and filename of the module to be included. The **Merge** parameter is a Boolean variable that determines whether preexisting objects with identical names are overwritten. If **Merge=True**, then conflicting variables are overwritten. If **Merge=False** and there are conflicting variables, then the call to **AddModule** fails.

- To read a script file:

```
If ADE.ReadScript("C:\...\MyScript.ana") Then
    ' Script successfully read
End If
```

A script file can contain a list of typescript commands. Upon loading the file, the engine executes the commands contained in the file. Errors encountered while running the script file are described in the **ErrorText** property.

- To save a module (i.e., a subset of the current model) in a separate file:

```
If ADE.SaveModuleFile ("MyLibrary", "C:\...\MyLibrary.ana") Then
    ' Save succeeded
End If
```

The first parameter is the module identifier, the second is the file name.

- To save the current model in a file:

```
If ADE.SaveModel("C:\...\MyNewModel.ana") Then
    ' Save succeeded
End If
```


- To close the current model without saving:

```
If ADE.CloseModel() Then ... ' Close succeeded
```

The **CloseModel** method takes no parameters.

ADE objects

- To create a new **CAObject** object:

```
Dim ObjName As String = "NewVariable"
Dim ObjClass As String = "Variable"
Dim var As CAObject = ADE.CreateObject(ObjName, ObjClass)
```

The object name and the class of the object to be created are passed into the **CreateObject** method. Note that an identifier and not the title of the object should be used when giving the object a name. Most object-related methods use their **Identifier** attribute, not their **Title** attribute. ADE can create the following types of objects: variable, module, chance, constant, decision, index, and objective. Refer to the *Analytica User Guide* for more information on these object types.

- To delete an Analytica object from a model:

```
Dim obj as CAObject
If ADE.DeleteObject(obj) Then ... ' Successful
```
- To set the active module:

```
ObjName = "ModuleToMakeActive"
ObjClass = "Module"
Var = ADE.CreateObject (ObjName, ObjClass)
ADE.CurrentModule = Var
```

ADE uses a hierarchy to order objects. When an object is created, it is created inside the current module. By default, all objects are placed within the top-level module unless you set the **CurrentModule** property.

- To identify the current module:

```
Dim module As CAObject = ADE.CurrentModule
```
- To obtain a **CAObject** object when you know the name of an Analytica variable (this is probably the most commonly used method in ADE):

```
Dim Var As CAObject = ADE.GetObjectByName ("IdentifierInModel")
If Var Is Nothing Then
    'Analytica model associated with Ana
    'does not contain variable with
    'identifier "IdentifierInModel"
End If
```

The method **CAObject::Get** is synonymous with **GetObjectByName**.

- You can get all Analytica object attributes using the **GetAttribute** method:

```
UnitsOfVar = Var.GetAttribute ("Units")
```


 Use the **SetAttribute** method to change an attribute of an Analytica object:

```
If Var.SetAttribute ("definition","A/B") Then
    'Attribute Set Correctly
Else
    'Attribute Not Set
End If
```
- To access or rename the identifier of an object, use the **Name** property:

```
Dim oldName As String = Var.Name
Var.Name = "NewIdentifier"
```

For the full lists of object attributes, see Chapter 2 of *The Analytica Scripting Guide*.

Retrieving computed results

The **CAObject** class contains three methods that cause results to be computed and returned. The **Result** method evaluates an object in your model and returns the result as a single value. This is most useful if you know that the result is a single number or single text string. The **Result-Table** method evaluates an object in your model and returns the result as a **CATable** object. Methods and properties of the **CATable** object allow you to understand what dimensions are present and to access individual elements (cells). The **Evaluate** method processes an arbitrary expression and returns the result of parsing and evaluating that expression as a multi-dimensional **CATable**.

When retrieving results, you have control over which computation mode is used to compute the result. You can compute the deterministic mid point value, or the various probabilistic views: Mean, Sample, PDF, CDF, Statistics, or Bands. Set the **ResultType** to indicate which result type you desire (default is **Mid**).

Whether you are computing a scalar or a table, your program eventually accesses individual **atomic** values such as numbers or text strings. You can use various **RenderingStyle** settings to control the form in which these values are returned. For example, numeric values can be returned as floating point numbers, formatted strings, or full-precision string depictions. Textual strings can be returned with or without surrounding quotes.

- To evaluate and obtain a simple result (e.g., a scalar) of an object, use the **Result** method of **CAObject**:

```
Dim Obj As CAObject
Dim Result
Obj = ADE.GetObjectByName ("ObjectToEvaluate")
Result = Obj.Result
If ADE.ErrorCode = 0 Then
    'Result was successfully retrieved
Else
    'An error occurred
End If
```

By default, the **Result** property of **CAObject** retrieves the midpoint result of the object. It returns the result as a variant (or in .NET, as a **System.Object**). This method is convenient for retrieving the results of objects that evaluate to a scalar.

- To evaluate and obtain the result of an object as something other than the midpoint, use the **ResultType** property of **CATable** or **CAObject**:

```
Dim Obj As CAObject = ADE.GetObjectByName("ObjectToEvaluate")
Dim Result
Obj = ADE.GetObjectByName("ObjectToEvaluate")
Obj.ResultType = 1 ' get result as mean
Result = Obj.Result
If ADE.ErrorCode = 0 Then
    'Result was successfully retrieved as a mean
Else
    'An error occurred
End If
```

The **ResultType** property is used to indicate the type of result that **Result** should return. Possible values are 0=Mid point, 1=Mean, 2=Probabilistic Sample, 3=PDF, 4=CDF, 5=Statistics, and 6=Probability Bands. When **ResultType**>=2, the result is always a table, even if the mid and mean are scalars. See the next section for a discussion on retrieving table results.

- To retrieve a formatted result, set properties of the object's **RenderingStyle**:

```

Dim Obj As CAObject = ADE.GetObjectByName("ObjectToEvaluate")
Dim Result
Obj.RenderingStyle.NumberAsText = true
Obj.RenderingStyle.StringQuotes = 2 ' double quotes.
Result = Obj.Result
If ADE.ErrorCode = 0 then
    ' Result was successfully returned.
End If

```

In this example, numbers are returned as formatted text using the object's number format property. Strings are returned surrounded by double quotes. So, for example, the numeric value 1.2K might be returned as the string "\$1,200.00" if the number format happens to be fixed point, two digits, with trailing zeros, thousand separators, and currency. This numeric value is returned as a text string because the **NumberAsText** property is True. The string would be returned as "\$1,200.00" with two extra double quote characters in the result string. This is controlled by the **StringQuotes** property (0 = no quotes, 1 = 'single quotes', 2="double quotes").

Retrieving multi-dimensional results

Before describing how to obtain results from table objects (arrays with one or more dimensions), let us briefly discuss the conceptual model of a table in Analytica.

An Analytica table has the following components:

- Indexes, each of which identifies a dimension of the table
- Values in the cells of the table
- Index labels, which identify the coordinates of each cell

The number of indexes determines the dimensionality of the table. So, for example, if a table contains two indexes, then the table is two-dimensional.

The number of elements in the index determines the actual number of cells in the table. Suppose table **T** is composed of two indexes, **I** and **J**. If **I** has five elements (**AA**, **BB**, **CC**, **DD**, and **EE**) and **J** has three elements (**A**, **B**, and **C**), then **T** is either a 5x3 table, or a 3x5 table, depending on your perspective.

Determining your perspective of a table is very important when working with ADE. It is up to you to tell ADE how you wish to view the table. So, for example, in the paragraph above, if you tell ADE to use index **I** first, followed by index **J**, then element 2,3 would be the element described by position **I=B**, **J=CC**. If, however, you tell ADE to use index **J** first, followed by index **I**, then element 2,3 would be described by position **I=C**, **J=BB** (note that tables in ADE are 1-based; that is, each dimension goes from 1 to **N** where **N** is the size of the index). The method called **SetIndex-Order**, described below, allows you to set the order of the indexes for your table so that you can look at the table in any way you desire.

The ADE methods are very flexible in terms of how you refer to individual elements in the table. You can either refer to the individual elements by their position number or by their label names. So, for example, you can tell ADE to give you the element at position 2,1 (2 along the first index, and 1 along the second index), or you can tell ADE to give you the element described by '**BB**', '**A**' where '**BB**' and '**A**' are label names in their respective indexes. The methods most commonly used for these types of transactions (**GetDataByElements** and **GetDataByLabels**) are described below.

As discussed in the previous section, the **Result** and **ResultType** methods are used to evaluate and obtain the result of an object. For objects that evaluate to multi-dimensional results, however, it is often inconvenient to use the **Result** method because the output would be a long comma-delimited string in the following form:

```
Table Index1...IndexN [Value1, Value2...]
```

Here, **Index1** to **IndexN** are the indexes of the table, and **Value1** to **ValueN** are the values in the table (which are filled in row by row). So, if we wanted to get at a particular element in the table after using the **Result** method, we would have to parse through the comma-delimited string returned from **Result** to get at the element of interest. Fortunately, ADE provides an ADE object of type **CATable** that provides methods to simplify the manipulation of tables.

- To evaluate and obtain the result of an object as a table use the **ResultType** method of **CAObject**:

```
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim TableResult As CATable = Obj.ResultTable
If Not (TableResult Is Nothing) Then
    'Result table was successfully retrieved
Else
    'An error occurred
End If
```

The **ResultTable** method of **CAObject** returns an automation object of type **CATable**.

CATable contains various methods that allow you to set, retrieve, and manipulate individual elements in the table. More than likely, the first thing that you want to do after retrieving the **CATable** object is to set the index order of the result table.

- To parse and evaluate an arbitrary expression, use the **Evaluate** method of **CAObject**.

```
Dim Obj As CAObject = ADE.GetObjectByName ("ContextObject")
Obj.ResultType = 2      ' Sample
Dim TableResult As CATable = Obj.Evaluate("Normal(X,Y^2) / Z")
If ADE.ErrorCode <> 0 Then
    'An error occurred
Else
    'Evaluation successful
End If
```

To use **Evaluate**, you must first obtain a **CAObject** instance. Although the expression you are evaluating might have nothing to with any specific object, the **CAObject** serves a couple of purposes. First, the **ResultType** property of the object provides a place to specify the result type that you want computed. Second, if you make use of the **NumberAsText** rendering style, the number format stored with the indicated object determines how the numbers are formatted. Often, however, the object you use is of no consequence; you can even use the top-level model object as your context object.

Comparing the previous two examples demonstrates also that there are often two ways to detect failure. The **ErrorCode** property is non-zero if an error occurred during the evaluation of a method. And for many methods, the return value is **Nothing** or **False** if it fails.

- To set the index order of a **CATable** object, use the **SetIndexOrder** method:

```
Dim Obj As CAObject
Dim TableResult As CATable
Dim IndexOrder (2) as String
Set Obj = ADE.GetObjectByName ("MultiDimObject")
Set TableResult = Obj.ResultTable
If Not (TableResult Is Nothing) Then
    'Result table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"

    If TableResult.SetIndexOrder(IndexOrder) Then
        'Index Order set successfully
    Else
        'An error occurred in setting index order
    End If
End If
```

```

        End If
    Else
        'An error occurred
    End If

```

The code above assumes that we are manipulating a two-dimensional table. We set the index order of this table so that **Index2** is the first index, and **Index1** is the second index.

In some computer languages, the first element of an array is considered position 0 (zero-based), and in other languages it is position 1 (one-based). Analytica's **Slice** function, and the ADE methods are one-based. Older versions of Visual Basic are one-based, while current versions of Visual Basic and most other modern programming languages are zero-based. In the example above, the Visual Basic array was declared and used as follows:

```

Dim IndexOrder(2) As String
IndexOrder (1) = "Index2"
IndexOrder (2) = "Index1"

```

In modern Visual Basic, this declares an array that ranges from position 0 to position 2 — an array having three elements. Because the first element was not set, it contains the special value **Empty**. ADE can recognize whether zero-based or one-based arrays are being passed to it. So, depending on your preference, it would work equally well to use a zero-based version, for example:

```

Dim IndexOrder(1) As String
IndexOrder (0) = "Index2"
IndexOrder (1) = "Index1"
ResultTable.SetIndexOrder (IndexOrder)

```

- To retrieve an element in a table by index order, use the **GetDataByElements** method:

```

Dim Obj As CAObject
Dim TableResult As CATable
Dim IndexOrder (2) As String
Dim Pos (2) As Integer
Dim Element
Obj = ADE.GetObjectByName ("MultiDimObject")
TableResult = Obj.ResultTable
If Not (TableResult Is Nothing) Then
    'Result table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    RetValue = TableResult.SetIndexOrder (IndexOrder)
    If RetValue = True Then
        'Index Order set successfully
        Pos (1) = 2
        Pos (2) = 1
        Element = TableResult.GetDataByElements (Pos)
        If ADE.ErrorCode = 0 Then
            'element retrieved successfully
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
    'An error occurred

```

End If

This code uses **GetDataByElements** to retrieve the element at position **Index2=2**, **Index1=1** and stores the result to **Element**.

- To retrieve an element in a table by index labels, use the **GetDataByLabels** method:

```
Dim Obj As CAObject
Dim TableResult As CATable
Dim IndexOrder (2) As String
Dim Pos (2) As String
Dim Element
Obj = ADE.GetObjectByName ("MultiDimObject")
TableResult = Obj.ResultTable
If Not TableResult Is Nothing Then
    'Result table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    If TableResult.SetIndexOrder(IndexOrder) Then
        'Index Order set successfully
        Pos (1) = "SomeLabelInIndex2"
        Pos (2) = "SomeLabelInIndex1"
        Element = TableResult.GetDataByLabels (Pos)
        If ADE.ErrorCode = 0 Then
            'element retrieved successfully
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
    'An error occurred
End If
```

The code above uses **GetDataByLabels** to retrieve the element at position **Index2="SomeLabelInIndex2"**, **Index1="SomeLabelInIndex1"** and stores the result to **Element**.

- To control the format of elements obtained by **GetDataByLabels**, **GetDataByElements**, **AtomicValue**, or **GetSafeArray** methods, set **CATable's RenderingStyle** properties:

```
Dim TableResult as CATable = Obj.ResultTable
Dim rs As CARenderingStyle = TableResult.RenderingStyle
rs.NumberAsText = True
rs.FullPrecision = True
rs.UndefValue = ""
rs.StringQuotes = 1
Dim Element
If TableResult.SetIndexOrder(Split("Index2;Index1", ";")) Then
    Element = TableResult.GetDataByLabels( _
        Split("SomeLabel1,SomeLabel2", ","))
    If ADE.ErrorCode=0 Then
        ' Element retrieved successfully
    End If
End If
```

- To retrieve the whole table into a Visual Basic or .NET array in one call, use the **GetSafeArray** method:

```
Dim Obj As CAObject
Dim TableResult As CATable
Dim IndexOrder (2) as String
Dim Pos (2) As Integer
Dim TheWholeTable
Obj = ADE.GetObjectByName ("MultiDimObject")
TableResult = Obj.ResultTable
If Not (TableResult Is Nothing) Then
    'Result table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    If TableResult.SetIndexOrder(IndexOrder) Then
        'Index Order set successfully
        TheWholeTable = TableResult.GetSafeArray
        If ADE.ErrorCode = 0 Then
            'table retrieved successfully
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
    'An error occurred
End If
```

The code above uses **GetSafeArray** to store the entire table in **TheWholeTable**. The elements of each dimension associated with the table returned from **GetSafeArray** are indexed 1 to **N**, where **N** is the length of the dimension. The lower bound of the safe array can be changed to zero using this code prior to calling **GetSafeArray**:

```
TableResult.RenderingStyle.SafeArrayLowerBound = 0
```

The syntax for reading a multi-dimensional result in a .NET array in C# is worth mentioning:

```
Array theWholeTable = (Array) tableResult.GetSafeArray( );
```

- To determine the number of dimensions of the table, use the **NumDims** property:
`NumDimensions = ADE.Get("MultiDimObject").ResultTable.NumDims`
- To get the index names associated with the table, use the **IndexNames** method:

```
Dim CurIndexName As String
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim TableResult As CATable = Obj.ResultTable
Dim NumDimensions As integer = TableResult.NumDims
Dim I as Integer
For I = 1 To NumDimensions
    CurIndexName = TableResult.IndexNames(I)
    MsgBox "Current index is " & CurIndexName
Next I
```

The **IndexNames** method returns the index names of the table in the order specified to **SetIndexOrder**. If **SetIndexOrder** has not been set for the **CATable**, then the default order of the indexes is returned.

- To associate **CAIndex** objects with your table, use the **GetIndexObject** method of **CATable**:

```

Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CTable = Obj.ResultTable
Dim NumDimensions As Integer = Res.NumDims
Dim CurIndexName As String = Res.IndexNames(NumDimensions)
Dim IndexObj As CAIndex = Res.GetIndexObject(CurIndexName)

```

The example above retrieved the last **CAIndex** object, with respect to the index order, from the table. The **CAIndex** object provides properties and methods that allow you to obtain information about the respective index.

- To get the number of elements in the index, use the **IndexElements** property:

```

Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CTable = Obj.ResultTable
Dim NumDimensions As Integer = Res.NumDims
Dim CurIndexName As String = Res.IndexNames (NumDimensions)
Dim IndexObj As CAIndex = Res.GetIndexObject(CurIndexName)
Dim NumElsInIndex As Integer = IndexObj.IndexElements

```

- To get an index label at the specified position in the index, use the **GetValueByNumber** method:

```

Dim I As Integer
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CTable = Obj.ResultTable
Dim NumDimensions As Integer = Res.NumDims
Dim CurIndexName As String = Res.IndexNames(NumDimensions)
Dim IndexObj As CAIndex = Res.GetIndexObject (CurIndexName)
Dim Str As String = "The elements in the index are: " & vbCrLf
For I=1 To IndexObj.IndexElements
    Str = Str & IndexObj.GetValueByNumber(I) & " "
Next I
MsgBox Str

```

- To get at the position of an index label in an index, use the **GetNumberByValue** method:

```

Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CTable = Obj.ResultTable
Dim IndexName As String = Res.IndexNames(Res.NumDims)
Dim IndexObj As CAIndex = Res.GetIndexObject (IndexName)
Dim IndexPosition As Integer
IndexPosition = IndexObj.GetNumberByValue("SomeIndexLabel")
If ADE.ErrorCode = 0 Then
    ' the index position was successfully retrieved
Else
    ' an error occurred
End If

```

- To obtain the scalar value in a zero-dimensional array:

```

Dim Res As CTable = ADE.Get("SomeObject").ResultTable
Dim x As Object
If Res.NumDims = 0 Then
    x = Res.AtomicValue
Else
    ' Handle the array case.
End If

```

Sometimes, it is not possible to know in advance whether the evaluation of an object returns

a multi-dimensional result or a scalar. In this case, use **ResultTable**. If the result happens to be a scalar, **NumDims** returns zero. In this case, the so-called “array” isn’t an array at all, but rather contains a single atomic value. It is also possible to end up with a zero-dimensional array after calling the **CATable::Slice** or **CATable::Subscript** methods. To obtain the atomic value, use the **CATable::AtomicValue** method.

- To reduce dimensionality in **Slice** and **Subscript** operations:

```
Dim PandL As CATable = ADE.Get("P_n_L_Statement").ResultTable
Dim CatIndex As CAIndex = PandL.GetIndexObject("Categories")
Dim Expenses As CATable = PandL.Subscript(CatIndex, "Expenses")
Dim Year As CAIndex = Expenses.GetIndexObject("Year")
Dim InitialExpense
InitialExpense = Expenses.Slice(Year, 1).AtomicValue
```

The **Slice** and **Subscript** methods of **CATable** return a new **CATable** object with the number of dimensions reduced by one. These methods are similar to the **Slice** and **Subscript** functions built into Analytica. **Slice** returns the Nth slice (by position) along a given dimension. **Subscript** returns the slice corresponding to a specified index value.

Creating tables and setting values in tables

We can apply the same methods described above to definition tables to retrieve values from result tables. A **definition table**, as the name suggests, is when the definition of an object is a **Table** function (also known as an **edit table** in Analytica).

The value of an Analytica variable (accessed via **ResultTable**) can be an array — not because it was defined by a definition table, but simply because it is defined as an expression or function that returns an array value.

When using an edit table, you need to pay careful attention to whether you are passing general expressions into each table cell, or just literal strings. The **RenderingStyle.GeneralExpression** property determines how string values that you send to the table are interpreted. By default, **GeneralExpression=true**, which means that if you set a cell value to the string "**Revenue**", this is an actual expression consisting of one variable identifier, and not a literal string. If you are populating a definition table with literal constants (as you might an input table to your model), you should either use **RenderingStyle.GeneralExpressions=false**, or remember to prepend and append quotation marks on all literal string values.

An object defined as a definition table does not necessarily produce the same table when **ResultTable** is called. After all, the definition table can be defined to be an array of identifiers. When **ResultTable** is called, each identifier’s result is evaluated, and a new table is produced (which would be different than the definition table). If identifiers evaluate to arrays, the result table might have more dimensions than the definition table.

- To get the definition table of an object as a **CATable**, use the **DefTable** method of **CAObject**:

```
Dim Obj As CAObject
Dim TableDef As CATable
Obj = ADE.GetObjectByName ("MultiDimObject")
TableDef = Obj.DefTable
If Not TableDef Is Nothing Then
    'Definition table was successfully retrieved
Else
    'An error occurred, or definition is not a table
End If
```

After the definition table is retrieved, we can use all the same methods described in the section above (**GetDataByElements**, **GetDataByLabels**, **SetIndexOrder**, etc.) to retrieve elements in the table and to obtain information about the indexes in the table. We can also

use the same method that we used above in determining whether the result of the object was multi-dimensional or scalar to determine whether the definition of the object is a table or scalar:

```
Dim Obj As Object
Dim TableDefinition As Object
Dim ScalarDefinition

Obj = ADE.GetObjectByName ("SomeObject")
TableDefinition = Obj.DefTable
If TableDefinition Is Nothing Then
    ScalarDefinition = Obj.GetAttribute ("definition")
    If ADE.ErrorCode = 0 Then
        'you have a scalar definition
    Else
        'an error occurred
    End If
Else
    'you have a table definition
End If
```

- To set an element in a table by index order, use the **SetDataByElements** method of **CATable**:

```
Dim Obj As CAObject
Dim TableDef As CATable
Dim IndexOrder (2) As String
Dim Pos (2) As Integer
Dim Element

Obj = ADE.GetObjectByName ("MultiDimObject")
TableDef = Obj.DefTable
If Not TableDef Is Nothing Then
    'Definition table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    If TableDef.SetIndexOrder (IndexOrder) Then
        'Index Order set successfully
        Pos (1) = 2
        Pos (2) = 1
        Element = "'ABC'" ' Notice the extra quotes

        If TableDef.SetDataByElements (Element, Pos) Then
            'element successfully set
            If TableDef.Update Then
                'model successfully updated
            Else
                'error updating def in model
            End If
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
```

```

        'An error occurred, or definition is scalar
    End If

```

This code uses **SetDataByElements** to set the element at position **Index2=2**, **Index1=1** to **Element**. Note the use of the quotes around **ABC**. Here, since **ABC** is single quoted, we are putting the string "ABC" in the table. If we instead set **Element** to "ABC", then the expression **ABC** would be placed in the table. In the latter case, **ABC** would likely be a variable. If an identifier, **ABC**, did not exist in the model, then an error would have occurred while trying to set the element in the latter case. The code then used **Update** to update the model with the new definition. It is important to note that the model containing the object is not updated until **Update** is called. Therefore, if **Update** is not called, and the result of a node that depends on this object is later calculated, the old definition of this object is still used. The other important thing to note is that **Update** functions very differently for result tables than for definition tables. For result tables, **Update** retrieves the result from the specified object again. Therefore, it overwrites any changes that were made to the object using

SetDataByElements and **SetDataByLabels**.

- To set an element in a table by index labels, use the **SetDataByLabels** method of **CATable**:

```

Dim Obj As CAObject
Dim TabDef As CATable
Dim IndexOrder (2) as String
Dim Pos (2) as String
Dim Element
Obj = ADE.GetObjectByName ("MultiDimObject")
TabDef = Obj.DefTable
If Not TabDef Is Nothing Then
    'Definition table was successfully retrieved
    TabDef.RenderingStyle.GeneralExpression = False
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    If TabDef.SetIndexOrder (IndexOrder) Then
        'Index Order set successfully
        Pos (1) = "SomeLabelInIndex2"
        Pos (2) = "SomeLabelInIndex1"
        Element = "ABC"
        If TabDef.SetDataByLabels(Element,Pos) Then
            'element set successfully
            If TabDef.Update Then
                'model successfully updated
            Else
                'an error occurred
            End If
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
    'An error occurred, or definition is scalar
End If

```

The code above uses **SetDataByLabels** to set the element at position **Index2="SomeLabelInIndex2"**, **Index1="SomeLabelInIndex1"** to **Element**. In this example, the **RenderingStyle.GeneralExpression** property was set to **False**. This

eliminates the need to explicitly include quotes around the string as was done in the previous example for **SetDataByElements**.

- To set the whole table in one call, use **PutSafeArray** and **Update**:

```
Dim Obj As CAObject
Dim TableResult, TableDef As CTable
Dim RetValue
Dim TheWholeTable

Obj = ADE.GetObjectByName ("MultiDimObject")
TableDef = Obj.DefTable
TableResult = Obj.ResultTable
TheWholeTable = TableResult.GetSafeArray
'make changes to TheWholeTable

...
RetValue = TableDef.PutSafeArray (TheWholeTable)
If RetValue = True Then
    'table successfully put
    RetValue = TableDef.Update
    If RetValue = True Then
        'model successfully updated
    End If
End If
```

- To create a whole table from scratch, use **CreateDefTable**:

```
Dim Obj As CAObject
Dim RetValue
Dim IndexLabs (2) As String

Obj = ADE.CreateObject ("MyNewTable", "Variable")
IndexLabs (1) = "I"
IndexLabs (2) = "J"
RetValue = Obj.CreateDefTable (IndexLabs)
If RetValue = True Then
    'a table indexed by I and J has successfully
    'been created. We are assuming that I and J
    'already exist
Else
    'an error occurred when creating the table
End If
```

The code above created a definition table indexed by **I** and **J**. The table is dimensioned according to the size of **I** and **J**. All the cells in the table are initially set to 0. The user can then call **DefTable**, and then use **SetIndexOrder**, **SetDataByElements**, **SetDataByLabels**, **PutSafeArray**, and **Update** to put values into the table. Note that the function **CreateDefTable** is very rarely used in an ADE program. After all, it is much easier to create an object in Analytica than it is in ADE.

Adjusting how values are returned

Analytica models can contain several different data types for values in attributes or in the cells of a table or index. Data types include floating point numbers, textual strings, the special values **Undefined** and **Null**, references, and handles to other objects. When these data types are

returned and ultimately mapped to data types in the programming language you are using, you might want or need to alter how the values are returned. You can do so using **RenderingStyle**.

The **CAObject**, **CATable**, and **CAIndex** objects all contain a property called **RenderingStyle**, which returns a **CARenderingStyle** object. Properties of the rendering style can be changed to alter how values are returned. You can also control whether safe arrays returned by Analytica are 1-based or 0-based. These settings impact **CAObject::GetAttribute** and **CAObject::Result**; they also impact **CATable::GetDataByElements**, **CATable::GetDataByLabels**, **CATable::AtomicValue**, **CATable::GetSafeArray**, and **CAIndex::GetValueByNumber**.

When transferring values to cells in a **DefTable**, you can also control whether the cells are populated by literal strings and values, or by general expressions. This is controlled by the **GeneralExpression** property of **CARenderingStyle**.

The **DefaultRenderingStyle** and **DefaultDefTableRenderingStyle** properties of **CAEngine** can be set once just after the **CAEngine** has been instantiated to set the rendering style globally. For example, if you always use zero-based arrays, this can be specified once.

- Retrieving numeric values as numbers:

```
obj.RenderingStyle.NumberAsText = False
Dim x As Double = obj.Result
```

Numeric values are returned as numbers by default, so unless **NumberAsText** is set to **True** at some point, there is no need to specify this explicitly.

- Retrieving numeric values as formatted strings:

```
Dim TableResult As CATable = Obj.Evaluate("1 / 3 * 10 ^ 6")
TableResult.RenderingStyle.NumberAsText = True
TableResult.RenderingStyle.FullPrecision = False
Dim s As String = TableResult.AtomicValue
```

The number format associated with **Obj** is used to format the numeric value. A suffix style with four digits returns "333.3K" for this example.

- Retrieving numeric values as strings with no loss of precision:

```
Dim TableResult As CATable = Obj.Evaluate("1 / 3 * 10 ^ 6")
TableResult.RenderingStyle.NumberAsText = True
TableResult.RenderingStyle.FullPrecision = True
Dim s As String = TableResult.AtomicValue
```

Analytica continues to use the number format associated with **Obj**, but the significant digits is increased to avoid any loss in precision. So, suffix, exponential, fixed point, and percent formats are not truncated. If a date, integer, or Boolean format is used, some truncation might still occur. In the example, the return value would be 333.333333333333.

- Retrieving string results without quotation marks:

```
tab.RenderingStyle.StringQuotes = 0
```

- Retrieving string results with explicit quotation marks:

```
tab.RenderingStyle.StringQuotes = 1 ' for single quotes
tab.RenderingStyle.StringQuotes = 2 ' for double quotes
```

- Using a custom value for **Undefined**:

```
ADE.DefaultRenderingStyle.UndefValue = ""
```

By default, the special value **Undefined** is returned as a the special Windows variant type **Empty**. There are some scripting languages that cannot deal with the **Empty** data type, so if you encounter this problem, you might want to change this value.

- Setting the cells of a definition table to string values:

```
defTab.RenderingStyle.GeneralExpression = False
defTab.SetDataByElements("A & B", inds)
```

In this example, the indicated table cell is set to the string value "A & B". When this table is evaluated, this cell's result is a string containing the five characters "A & B".

- Setting the cells of a definition table to expressions:

```
defTab.RenderingStyle.GeneralExpression = True
defTab.SetDataByElements("A & B", inds)
```

Here the cell is set to the expression "A & B". When this table is evaluated, the variable named **A** and the variable **B** is evaluated, and their results are coerced to strings and concatenated by the & operator.

You can set table cells to literal strings with **GeneralExpression=True**, but you must embed explicit quotations marks in the expression. For example:

```
defTab.RenderingStyle.GeneralExpression = True
defTab.SetDataByElements("'A & B'", inds)
GeneralExpression=True by default.
```

Terminating an in-progress computation

Methods such as **CAObject.ResultTable**, **CAObject.Result**, or **CAObject.Evaluate** may initiate lengthy evaluations that could tie up your application. Some models may take many minutes, or even several hours, to compute the result. Normally, your call to **ResultTable** does not return until the result is fully computed.

You may want to provide your end-user with the capability to abort the current computation, such as when a "break" key or button is pressed. Alternatively, you may want to terminate the current computation if your application is suddenly being terminated while a lengthy computation is still processing, so that ADE will also terminate.

To implement the ability to abort a computation, you need to have some form of multi-threading in your application. The thread that calls ADE will be occupied waiting for ADE to return, so some other thread in your application must detect the request to terminate the computation. When this request is detected, the second thread communicates this request to ADE through a global system event object. You must obtain the information required to locate this global event object from ADE *before* you have started the length computation. Each **CAEngine** instance has its own event object with a unique global name, but the same event object applies to the same **CAEngine** instance for the entire lifetime of the **CAEngine** instance.

To obtain the event object (using Windows Platform SDK calls):

```
Dim abortEventName As String
Dim abortHandle as Handle ' This needs to be global
ade.SendCommand("GetProcessInfo('Abort Event Object')")
abortEventName = ade.OutputBuffer
abortHandle = OpenEvent(EVENT_MODIFY_STATE, FALSE, abortEventName )
...
' Launch the computation
tab = obj.ResultTable
```

In a separate thread that detects a request to terminate the computation, the following code sends that request to ADE (using Windows Platform SDK calls):

```
SetEvent(abortHandle)
```

Note that **OpenEvent** and **SetEvent** are Windows Platform SDK calls, not part of ADE. Also notice that the **abortHandle** variable must be a global that is accessible from both threads. However, the second thread does not require access to the ADE objects themselves. Alternatively, the **abortEventName** can be shared and the second thread can call **OpenEvent**.

Once the event has been flagged, ADE will back-out of the current computation and return from the current method. A small delay may be experienced while ADE backs out of the current computation. Once it has returned, the **CAEngine.ErrorCode** is set to 78 ("Computation aborted"). Your application may continue using the same ADE instance as it would have prior to the call that was aborted.

Instantiating CAEngine using CALicense

To start using ADE, the first thing you need to do is instantiate a **CAEngine** object. You have the option of instantiating your **CAEngine** object directly, or you can instantiate a **CALicense** object first and use it to instantiate your **CAEngine** object. Once you've successfully obtained a **CAEngine** object, it doesn't matter which of these two methods you used to get it. The difference is that if you use a **CALicense** object to instantiate your **CAEngine**, more information about why it failed is available. In particular, if the failure is due to a license code problem, your application has an opportunity to handle the error gracefully.

If you are creating an ADE-based application that are to be redistributed to many end-users, who would otherwise not have an ADE license, you will need to make arrangements with Lumina to license ADE for distribution with your application. This form of licensing requires your application to provide a special type of license code to ADE, called an application license code, since your end-users will usually not have a license code of ADE on their computer. Note that a license to redistribute ADE in this fashion is not included as part of the standard ADE license agreement, and additional license fees apply. To use an application license code, you must use a **CALicense** to instantiate your **CAEngine**, since your application will use the **CALicense** object to supply the license code to Analytica.

The steps for instantiating a **CAEngine** using a **CALicense** are as follows, e.g.,:

```
' In VB
Imports ADE
...
Dim license As CALicense = New CALicense
Dim ade As CAEngine = license.NewCAEngine
If (ade Is Nothing) Then
    ' substitute error handling routine here
    ReportError(license.ErrorText)
End If

// In C#
Using ADE;
...
CALicense license = new CALicense();
CAEngine ade = license.NewCAEngine();
if (ade==null) {
    ReportError( license.ErrorText );
}
```

If you have an application license code, insert a call to **SetApplicationLicenseCode**:

```
license = new CALicense
license.SetApplicationLicenseCode("1234-5678-ABCD-EFGH-IJKL")
ade = license.NewCAEngine
```

Using the Analytica Graphing Engine

When you have a **CATable** result with at least one dimension, you can obtain a graph of the result as an image. One use of this is to embed graphs as JPEG images in a web page that uses ADE on the back end.

Obtaining the graph of a result requires the following steps:

1. Select the appropriate graph settings, such as chart type, axis range settings, colors, fonts, and so on. The easiest way is to open the model in Analytica Enterprise, and select the settings you want for each variable using the **Graph Setup** dialog.

The graph template you create using the **Graph Setup** dialog is stored in the **GraphSetup**

attribute of the object. You can copy the **GraphSetup** attribute from an existing variable if you need to change the style template.

2. From ADE, obtain a **CATable** with the result to be graphed.
3. Set the **GraphWidth** and **GraphHeight** properties of the **CATable** object to indicate the desired size of the graph in pixels.
4. If your result has more than two dimensions, call **Slice** or **Subscribe** to reduce the dimensionality to the desired dimensionality for the plot (usually one dimension if there is no key or two dimensions if there is a key).
5. If you have more than one dimension, call **SetIndexOrder** to select the desired pivot for the graph.
6. If you are sending the graph to an output stream, obtain a Windows **IStream** interface to the stream. If you have a .NET Stream (**System.io.Stream**), you need to use a wrapper class (see below).
7. Call either the **GraphToStream** or **GraphToFile** methods of **CATable**, depending on where you want the graph written to. The graph can be created in different MIME types (e.g., **image/JPEG**).

If you are able to view the result graph in Analytica with no slicers, then steps 4 and 5 are unnecessary.

- Writing a result graph to a file:

```
Dim res As CATable = obj.ResultTable
res.GraphWidth = 640
res.GraphHeight = 400
If res.GraphToFile("C:\Temp\Result.JPG", "image/JPEG") Then
    ' success
End If
```

- Dynamically generating a result graph from an ASP.NET web page:

```
<%
Response.ContentType = "image/JPEG"
Dim varName As String = Request.QueryString("var")
Dim ADE As CAEngine = Session("ADE") ' assume existing session
Dim res As CATable = ADE.Get(varName).ResultTable
res.GraphWidth = 640
res.GraphHeight = 400
Dim stream As StreamConnector = _
    new StreamConnector( Response.OutputStream)
If res.GraphToStream( stream, "image/JPEG") Then
    ' success
End If
%>
```

In this example code, **Response**, **Request**, and **Session** are Active Server Page objects. The HTTP in the client browser would contain a tag like this:

```

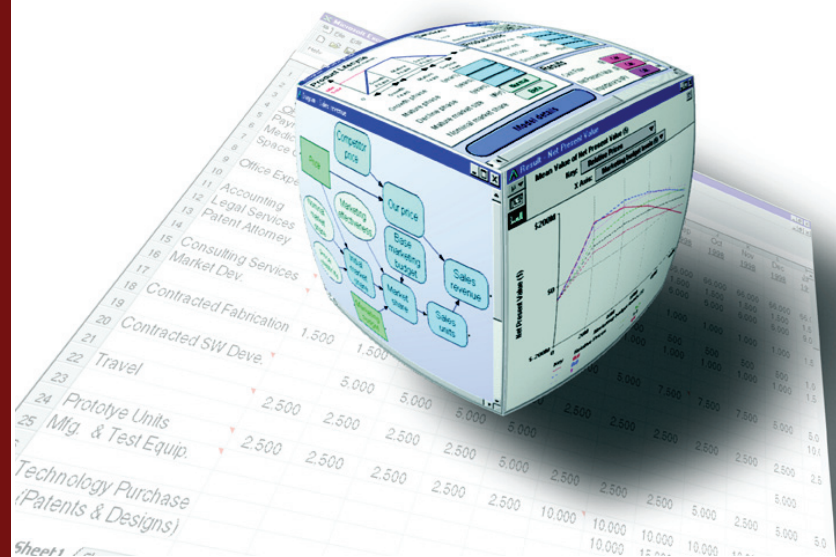
```

When Microsoft introduced .NET, they did not make the base **Stream** class interface in .NET compatible with the **IStream** interface in Windows. Because of this, it is necessary to create a stream wrapper that implements the **IStream** interface around the .NET **Stream** before passing it to **GraphToStream**. This wrapper, class **StreamConnector**, is included in the example **AdeTest**. To use the example above, add the file **StreamConnector.vb** to your project.

Chapter 5

ADE Server Class Reference

This chapter lists the properties and methods for the six ADE server classes **CAEngine**, **CALicense**, **CAObject**, **CATable**, **CAIndex**, and **CARenderingStyle**.



ADE server classes

The five ADE server classes are **CAEngine**, **CAObject**, **CATable**, **CAIndex**, and **CARenderingStyle**. They are listed below in that order, with a complete description of the properties and methods of each class.

Class CAEngine

Properties

Command

Description	Sets a typescript language command for execution by the ADE Automation Server. The Send method causes Command to be sent to ADE for execution. For the list of typescript commands, see the <i>Analytica Scripting Guide</i> .
Data type	string
Access	read/write
Usage	<code>ADE.Command = "value obj1"</code>

CurrentModule

Description	The currently open module.
Data type	CAObject
Access	read/write
Usage	<code>Dim Obj As Object</code> <code>Set Obj = ADE.CurrentModule</code>
Remarks	Newly created objects are placed into CurrentModule ; so, you should set CurrentModule before creating any new objects. Setting CurrentModule = Nothing means that no module is open, so all new objects are created in the top-level Module or Model of the currently opened model.
API errors	44- " Module could not be set"

DefaultDefTableRenderingStyle

Description	The default rendering style controlling how definition table values are transferred to and from ADE. All definition tables returned from CAObject::DefTable inherits these settings when they are first created.
Data type	CARenderingStyle
Access	read/write
Usage	<code>ADE.DefaultRenderingStyle.GeneralExpression = false</code>

DefaultRenderingStyle

Description	The default rendering style controlling how result values are returned from ADE. All CAObject instances inherit this rendering style when they are created.
Data type	CARenderingStyle
Access	read/write
Usage	<code>ADE.DefaultRenderingStyle.StringQuotes = 2</code>

ErrorCode

Description	Returns the error code generated by the last communication with the Analytica Decision Engine Server. The property ErrorCode should be checked after setting and retrieving critical CAEngine properties and calling CAEngine methods. An ErrorCode of zero indicates the last action was successful.
Data type	integer
Access	read
Usage	<pre>Dim x As Integer x = ADE.ErrorCode</pre>

ErrorText

Description	Short text explanation of error from ErrorCode .
Data type	string
Usage	<pre>Dim x As String x = ADE.ErrorText</pre>
Access	read

Log

Description	A record of all commands sent to the ADE typescript and the results received from those commands, when the Photo property is true.
Data type	string
Usage	<pre>Dim x As String x = ADE.Log</pre>
Access	read

OutputBuffer

Description	A text string buffer that contains the result of the last typescript (i.e., using the Command property and Send method) interaction with the ADE.
Data type	string
Usage	<pre>Dim x As String x = ADE.OutputBuffer</pre>
Access	read

Photo

Description	When Photo is True, ADE records all typescript commands and results into the Log property.
Data type	boolean
Access	read/write
Usage	<pre>ADE.Photo = True</pre>
Remarks	Setting Photo property to True slows down computation speed of the engine.

Methods

AddModule(fileName, merge)

Description	Adds a module from file <i>fileName</i> into the CurrentModule . The <i>merge</i> parameter currently has no effect and should be set to True.
--------------------	---

Parameters *fileName* – string
merge – Boolean

Return value *ModuleName* – string

Usage `ModName = ADE.AddModule ("C:\MYMOD\MYMOD.ANA", True)`

API errors 39 – "Module could not be found"

CloseModel

Description Closes the model.

Usage `ADE.CloseModel`

CreateObject(objName, objClass)

Description Creates a new Analytica object with identifier *objName* and class *objClass* in the **CurrentModule** and returns it as a **CAObject**.

Parameters *objName* – string
objClass – string

Return value **CAObject**

Usage `Dim obj As CAObject
Set obj = ADE.CreateObject ("NewVar", "Chance")`

Remarks *objClass* can be one of the following values: **Decision**, **Variable**, **Chance**, **Constant**, **Index**, **Module**, **Objective**, **Determ**, **Alias**, or **Formnode**.

API errors 40 – "Object could not be created"
41 – "Invalid name for object"
42 – "Object name already in use"
48 – "Invalid object class"

CreateModel(modelName)

Description Creates a new Analytica model with identifier *modelName*.

Parameters *modelName* – string

Return value Boolean (success or failure)

Usage `boolval = ADE.CreateModel ("MyNewModel")`

API errors 45 – "Model could not be created"

DeleteObject(obj)

Description Deletes **CAObject** *obj* from the current model.

Parameters *obj* – **CAObject**

Usage `Dim Obj As CAObject
Set Obj = ADE.GetObjectByName ("ObjToDelete")
ADE.DeleteObject (Obj)`

API errors 41 – "Invalid object"

GetObjectByName(objName)

Get

Description Returns an object of type **CAObject** for an existing Analytica object with identifier *objName*.

Parameters *objName* – string

Return value **CAObject**

Usage `Dim Obj As CAObject`
`Set Obj = ADE.GetObjectByName ("MyObject")`
`Set Obj = ADE.Get("MyObject") 'alternate equiv. form`

API errors 41 – "Invalid name for object"

OpenModel

Description Reads a model from a disk file and opens it as the current model.

Parameters **FileSpec** – string (the filename containing the model)

Return value **ModelName** – string (actual model name)

Usage `modName = ADE.OpenModel ("C:\TMP\MYMODEL.ANA")`

Remarks Failure should be detected by checking whether the return value is "", not by checking for a zero **ErrorCode**. It is possible that some errors or warnings might occur during loading, and is thus reflected in the **ErrorCode**, **ErrorText**, and **OutputBuffer** properties, even though the load was successful.

API errors 2 – Warning (but load was successfully completed)
 3 – Lexical error (load was only partially successful)
 4 – Statement error (load was only partially successful)
 39 – "Model could not be found"

ReadScript(filePath)

Description Reads an Analytica script file and executes it.

Parameters **filePath** – string

Usage `ADE.ReadScript ("C:\TMP\SCRIPT.MOD")`

API errors 39 – "Script file could not be found"

ResetError

Description Resets the error code, error text string associated with the error code, and the output buffer to default values. This function is normally used internally, but could be useful in other circumstances as well.

Usage `ADE.ResetError`

SaveModel(filePath)

Description Saves the model to file **filePath**.

Parameters **filePath** – string

Usage `ADE.SaveModel ("C:\TMP\CHANGES.ANA")`

API errors 46 – "Model could not be saved"
 49 – "There is no model to save"

SaveModuleFile(modName, filePath)

Description Saves module with identifier **modName** into file **filePath**.

Parameters **modName** – string
filePath – string

Return value Boolean (success or failure)

Usage `b = ADE.SaveModuleFile("Function_lib", "C:\TEMP\NEWMOD.ANA")`

API errors 41 – "Invalid name for object"
 46 – "Module could not be saved"

Send

Description	Sends the string contained in the Command property as a command to be executed by ADE. See the <i>Analytica Scripting Guide</i> for details of commands and syntax.
Return value	Boolean (success or failure)
API errors	1 – "Unimplemented" 2 – "Warning" 3 – "Lexical error" 4 – "Statement error" 5 – "Expression error" 6 – "Execution error" 7 – "System error" 8 – "Fatal error" 9 – "Undefined variable error" 10 – "Aborted"

SendCommand(command)

Description	<p>Sends the string command property as a typescript command to be executed by ADE. See the <i>Analytica Scripting Guide</i> for details of commands and syntax.</p> <p>SendCommand is a single method that is faster way to execute a typescript command than using the Send method. Using Send requires these two statements to execute a command.</p> <pre>ade.Command = "profile Val" b = ade.Send()</pre> <p>This can be done with a single SendCommand statement.</p> <pre>b = ade.SendCommand("profile Val")</pre>
Return value	Boolean (success or failure)
API errors	1 – "Unimplemented" 2 – "Warning" 3 – "Lexical error" 4 – "Statement error" 5 – "Expression error" 6 – "Execution error" 7 – "System error" 8 – "Fatal error" 9 – "Undefined variable error" 10 – "Aborted"

Class CALicense

A **CALicense** object can be obtained directly, e.g.:

```
license = new CALicense;
```

or

```
license = CreateObject("CALicense")
```

The **CALicense** provides a method for instantiating a **CAEngine**, and provides details about such an instantiation may have failed which would otherwise not be available if you instantiate the CAEngine directly. It also provides some information about limitations in your license to use ADE.

A **CALicense** instance can be obtained even if your ADE license code is invalid, expired, etc. This makes it possible for your application to say "failed because".

Properties

AvailableLicenseInstances

Description	Number of additional ADE instances that can still be instantiated using the current license, beyond those that are already running.
Data type	integer
Access	read
Usage	<code>nAvail = license.AvailableLicenseInstances</code>
Remarks	Your license to use ADE may or may not limit the number of instances of ADE that can be simultaneously active on the same computer. For licenses that do impose a limit, this property how many additional ADE instances (equivalent to the number of CAEngine instances) can be created on this computer within the limit imposed by the license.

CanUseOptimizer

Description	Indicates whether your ADE license allows your models to make use of the Optimizer. If you don't have this ability, then models that use LpDefine , QpDefine or NlpDefine functions cannot perform those optimizations within ADE. This flag makes it possible to detect that limitation before actually starting your application.
Data type	boolean
Access	read
Usage	<code>if (license.CanUseOptimizer){ ... }</code>

ErrorCode

Description	<p>Status code (reason for failure) from the previous method call. In particular, after calling the CALicense.NewCAEngine method, this provides information about the cause of the failure.</p> <p>Descriptive text for any error code can be obtained using the ErrorCode property. Some codes that could encountered (this is not comprehensive) include:</p> <p>71 : "The maximum number of CAEngine instances allowed are already in use"</p> <p>72: "Invalid license code"</p> <p>73: "Stale license code"</p> <p>74: "Expired license code"</p> <p>75: "No ADE license code is present"</p> <p>76: "Not an application license code" (from SetApplicationLicenseCode method)</p>
Data type	integer
Access	read
Usage	<code>errCode = license.ErrorCode</code>

ErrorText

Description	Status text describing the reason for a failure in the previous method call. In particular, after calling the CALicense.NewCAEngine method, this provides information about the cause of the failure.
Data type	string
Access	read
Usage	<code>errDescription = license.ErrorText</code>

MaxLicenseInstances

Description	Total number of ADE instances allowed on current license, including any currently running.
Data type	integer
Access	read
Usage	<code>nAvail = license.MaxLicenseInstances</code>
Remarks	Your license to use ADE may or may not limit the number of instances of ADE that can be simultaneously active on the same computer. For licenses that do impose a limit, this property provides access to what that number is (which will be greater than 0). If you have no limit, this is zero.

Methods

NewCAEngine

Description	Attempts to create an instance of CAEngine . If this is not possible, perhaps due to a limitation in your ADE license, the ErrorCode and ErrorText properties are set to indicate the reason for the failure.
Parameters	<i>none</i>
Return value	CAEngine , or Null upon failure
Usage	<code>ade = license.NewCAEngine</code>
API errors	See description for ErrorCode property.

SetApplicationLicenseCode

Description	Specifies an ADE license code that is to be used for the instantiation of a CAEngine when CALicense.CAEngine is called. When this is used, a valid ADE license code does not need to be stored in the system registry, and ADE will run according to the licensed features of the specified license code, which might be different from the license code used when ADE was installed (if any). This only accepts "ADE application license codes". The license code you use when you install ADE is not an application license code. Application license codes are obtained through special arrangement with Lumina for applications that have been licensed through Lumina for redistribution.
Parameters	<i>licenseCode</i> – string
Return value	CAEngine , or Null upon failure
Usage	<code>license.NewCAEngine("ABCD-1234-EFGH-I5J6-KLMN-7890")</code>
API errors	See description for ErrorCode property.

Class CAObject

Properties

ClassType

Description	Contains the type of the Analytica object.
Data type	string
Access	read/write
Usage	<code>classType = obj.ClassType</code>

Remarks ADE currently supports the following types of Analytica objects: decision, chance, constant, index, module, and variable.

DefinitionType

Description Provides quick information about how a variable is defined.

Data type integer

Access read

Return value -1 = Definition not parsed (e.g., no definition)
 0 = General expression
 1 = Edit table
 2 = Prob Table
 3 = Determ Table
 4 = Sub-Table
 5 = List
 6 = List of Labels
 7 = Choice

Usage `defType = obj.DefinitionType`

MethodEvaluationTime

Description A maximum time limit for evaluations, in milliseconds. If exceeded in any given call, the computation aborts. This may cause calculations by the methods **Result**, **ResultTable**, or **Evaluate** to terminate before the computation is complete.

Data type integer

Access read/write

Usage `obj.MethodEvaluationTime = 30000 /*ms*/`
`tab = obj.ResultTable`
`if (ade.ErrorCode == 77) { /*timeout occurred*/ }`

Name

Description Contains the name given to the Analytica object.

Data type string

Access read/write

Usage `obj.Name = "NewName"`

API errors 41 – "Invalid name for object"

RenderingStyle

Description Contains a **CARenderingStyle** object that controls how data is returned from Analytica. This property is inherited from the **DefaultRenderingStyle** property of **CAEngine** when the object is first instantiated. Its settings control how data is returned from **CAObject::Result** and **CAObject::GetAttribute**. Also, the settings are inherited by any **CATable** created from the object by the **Evaluate** or **ResultTable** methods.

Data type **CARenderingStyle**

Access read/write

Usage `obj.RenderingStyle.StringQuotes = 0`

ResultType

Description Specifies the treatment of uncertainty in the value obtained using the **Result** or **ResultTable** properties:

- 0 – Mid value (default)
- 1 – Mean
- 2 – Sample
- 3 – PDF
- 4 – CDF
- 5 – Statistics
- 6 – Confidence Bands

Data type short
Access read/write
Usage `CAObject.ResultType = 1`

Methods

CreateDefTable(indexList)

Description Creates an input table object in the definition attribute of the specified Analytica object with dimension specified by the *indexList*. The **IndexList** parameter must contain an array of identifiers of existing index variables (identical in form to the **IndexNames** method of class **CATable**). The number of indexes in *indexList* determines the number of dimensions of the table. One index can be **Self**, meaning that one of the dimensions are indexed by the **Indexvals** attribute of this variable. should be one of the entries in the array. Initially, the input table object's array is filled with null elements, which can be changed using the **SetDataByElements** and **SetDataByLabels** methods of the class **CATable**.

Parameters *indexList* – array of strings

Return value Boolean (success or failure)

Usage `Var.CreateDefTable (IndexList)`

API errors 25 – "Subscripts cannot be accessed"
 26 – "Lower bound of subscript array inaccessible"
 27 – "Upper bound of subscript array inaccessible"
 28 – "Must specify at least one element in table"
 32 – "Index object not found"

DefTable

Description Gives object of class **CATable** containing the input table for an Analytica variable, or Nothing if the variable was not defined as an input table.

Data type **CATable**

Access read/write

Usage `Dim tab As CATable`
`Set tab = obj.DefTable`

API errors 34 – "Definition table not found"

Evaluate(expr)

Description Parses and evaluates an Analytica expression **expr**.

Parameters **expr** – string

Return value **CATable**

Usage `dim tab As CATable = _`
`obj.Evaluate("Sum(Revenue,Division)")`

API errors 35 – "Attribute could not be retrieved"

GetAttribute(attribName)

Description	Gets the value of attribute attribName of the object.
Parameters	attribName – string
Return value	variant
Usage	<code>x = obj.GetAttribute("definition")</code>
API errors	35 – "Attribute could not be retrieved"

PictureToFile(fileName, mimeType)

Description	Causes ADE to retrieve CAObject object's picture, if any, and save it to file fileName , in the format specified by mimeType .
Return value	Boolean (success)
Parameters	fileName – string mimeType – string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png")
Usage	<code>obj = ade.Get("Pi1"); bSuccess = obj.PictureToFile(@"C:\Temp\myPict.jpg", "image/jpeg")</code>

PictureToStream(stream, mimeType)

Description	Causes ADE to retrieve CAObject object's picture, if any, and send it to the stream specified by stream , in the format specified by mimeType .
Return value	Boolean (success)
Parameters	stream – string MimeType – string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png")
Usage	<code>dim outStream as MyStreamWrapper = _ new MyStreamWrapper(Response.OutputStream) bSuccess = obj.PictureToStream(outStream, "image/jpeg")</code>

Result

Description	Gives the result value of the object as a single value (not a table). ADE first evaluates the value if it has not already been evaluated. If the value is an array, it returns it as a string of comma-delimited elements.
Data type	variant
Access	read
Usage	<code>Dim x x = obj.Result</code>
API errors	37 – "Could not retrieve result"

ResultTable

Description	Gives the value of the object as a CATable , with zero or more dimensions, or Nothing if the object cannot be evaluated. ADE first evaluates the variable if necessary.
Data type	CATable
Access	read
Usage	<code>Dim res As CATable Set res = obj.ResultTable</code>
API errors	38 – "Could not get result table"

SetAttribute(attribName, value)

Description	Sets the attribute attribName of the object to value . It returns true if successful or false if not.
Parameters	attribName – string value – Variant
Return value	Boolean
Usage	<code>bool = obj.SetAttribute ("definition","A/B")</code>
API errors	36 – "Attribute could not be set"

Class CTable

Properties

GraphHeight

Description	Controls the height of the graph image returned by GraphToStream or GraphToFile .
Data type	integer (number of pixels)
Access	read/write
Usage	<code>tab.GraphHeight = NumDims</code>

GraphWidth

Description	Controls the width of the graph image returned by GraphToStream or GraphToFile .
Data type	integer (number of pixels)
Access	read/write
Usage	<code>x = tab.TableType</code>

NumDims

Description	The number of dimensions of the table (zero if it is a scalar with no dimensions).
Data type	short
Access	read
Usage	<code>x = tab.NumDims</code>

RenderingStyle

Description	Contains a CARenderingStyle object that controls how atomic values are interpreted when transferred to and from table cells. Definition tables inherit this property from the DefaultDefaultableRenderingStyle property of CAEngine . Result table inherits this property from the CAObject that created the table.
Data type	CARenderingStyle
Access	read/write
Usage	<code>tab.Renderingstyle.NumberAsText = true</code>

ResultType

Description	Contains the type of result that was computed, and controls the type of result computed if the table is updated. Possible value are the same as for CAObject::ResultTable .
Data type	short

Access read/write
Usage `x = tab.ResultType`

TableType

Description This property holds the type of the table ("D" for a definition table, and "V" for a result table)
Data type string
Access read
Usage `x = tab.TableType`

Methods

AtomicValue

Description Retrieves the scalar value from a zero-dimensional **CATable** object. A zero-dimensional table results when a result is not an array, or when you call **Slice** or **Subscript** on a one-dimensional array.
Return value variant
Parameters none
Usage `x = tab.AtomicValue`

GetDataByLabels(indexLabels)

Description Retrieves the value of an input table cell according to **indexLabels**, which specify the label for each index of the table in order.
Return value variant
Parameters Values of indexes (*Variant*); the number of elements in *Variant* must be equal to **NumDims**.
Usage `IndexLabs (1) = 3`
`IndexLabs (2) = "green"`
`W = Var.DefTable.GetDataByLabels (IndexLabs)`
 Or
`IndexLabs (1) = 3`
`IndexLabs (2) = "green"`
`W = Var.ResultTable.GetDataByLabels (IndexLabs)`
 If the table has only one dimension, the parameter need not be an array:
`W = Var.ResultTable.GetDataByLabels ("green")`
API errors 24 – "Subscripts must be an array of variants"
 25 – "Subscripts cannot be accessed"
 26 – "Lower bound of subscript array inaccessible"
 27 – "Upper bound of subscript array inaccessible"
 28 – "Must specify at least one element in table"
 30 – "Position does not exist"

GetDataByElements

Description Retrieves the value of input table cell according to index values.
Return value variant
Parameters Index values (variant), number of elements in the variant must be equal to **NumDims**.
Usage `IndexPtrs (1) = 1`
`IndexPtrs (2) = 2`

```

W = Var.DefTable.GetDataByElements (IndexPtrs)
Or
IndexPtrs (1) = 1
IndexPtrs (2) = 2
W = Var.ResultTable.GetDataByElements (IndexPtrs)
If the table is one-dimensional, then an array is not needed:
W = Var.ResultTable.GetDataByElements (1)

```

API errors

- 24 – "Subscripts must be an array of variants"
- 25 – "Subscripts cannot be accessed"
- 26 – "Lower bound of subscript array inaccessible"
- 27 – "Upper bound of subscript array inaccessible"
- 28 – "Must specify at least one element in table"
- 29 – "Position specified is out of bounds"
- 30 – "Position does not exist"
- 31 – "Illegal Position Specified In Table"

GetIndexObject

Description Retrieves an index object by its name

Return value **CAIndex**

Parameters index name: string

Usage

```

dim AI as CAIndex
Set AI = AObj.GetIndexObject (IndexName)

```

Remarks If **ObjName** is not valid the method returns **Nothing**.

API errors 32 – "Index object not found"

GetSafeArray

Description Retrieves the **CATable** as a safe array (i.e., a Visual Basic array). The ordering of the dimensions is controlled by the **SetIndexOrder** method. The elements of each dimension are indexed 1 to N, where N is the size of each index.

Return value Array

Usage

```

dim Var As CAObject
Dim curTable
curTable = Var.GetSafeArray

```

GraphToFile(fileName, mimeType)

Description Creates a graph image of the data contained in the **CATable** object formatted using the **mimeType** and writes it to file **fileName**. It uses attribute settings for the **CAObject** from which the **CATable** was obtained to control graph settings, uncertainty settings, and number format. The **GraphWidth** and **GraphHeight** properties control the size of the graph image in pixels.

Return value Boolean (success)

Parameters

- fileName** – string
- mimeType** – string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png")

Usage

```

tab.GraphWidth = 350
tab.GraphHeight = 200
b = tab.GraphToFile( "C:\data\trends.jpg", "image/bmp")

```

GraphToStream(stream, mimeType)

Description Creates a graph image of the data contained in the **CATable** object formatted using the **mimeType**, and writes it to **stream**, a Windows IStream.

Note: Note: An IStream is not interchangeable with a .NET **System.IO.Stream** object (including a **Response.OutputStream** object in ASP.NET). A wrapper class is necessary for converting between these. The size of the image in pixels is controlled by the **GraphWidth** and **GraphHeight** properties of the table.

Return value Boolean (success)

Parameters **stream** – string
MimeType – string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png")

Usage

```
tab.GraphWidth = 350
tab.GraphHeight = 200
dim outStream as MyStreamWrapper = _
new MyStreamWrapper(Response.OutputStream)
b = tab.GraphToStream( outStream, "image/jpeg")
```

IndexName(IndexNumber)

Description Takes one parameter, **IndexNumber**, and returns the name of the corresponding index for **CATable**.

Return value string

Parameters **IndexNumber** – short int

Usage `dim string as indexTitle = Var.DefTable.IndexName (1)`

IndexNames

Description Returns the names of the indexes of the table as an array. Use indexes in this order when using the **GetDataBy** and **SetDataBy** methods to get or set elements of the table. You can change the order with the **SetIndexOrder** method.

Data type string array with dimension from 1 to object's **NumDims** property

Access read

Usage `Dim names(k) of String = Var.DefTable.IndexNames (k)`

PutSafeArray

Description Replaces the current table represented by this object with another table of the same dimensions.

Return Value **CATable**

Parameters The table (Visual Basic array) that replaces the current table.

Usage

```
Dim Var As Object
Dim TheArray
TheArray = Var.GetSafeArray
Var.PutSafeArray (TheArray)
```

API errors 24 – "Subscripts must be an array of variants"
 50 – "Safe-array has incorrect size or number of dimensions"

SetDataByLabels

Description Sets the value of an input table cell according to its index labels.

Return value Boolean (success or failure)

Parameters	Cell value (Variant), values of indexes (Variant), number of elements in this variant must be equal to NumDims .
Usage	<pre> IndexVals (1) = 3 IndexVals (2) = 'green' RetVal= Var.DefTable.SetDataByLabels (W, IndexVals) </pre> <p>If the table is one-dimensional, then an array is not needed:</p> <pre> W = Var.DefTable.SetDataByLabels (W, "green") </pre>
API errors	24 – "Subscripts must be an array of variants" 25 – "Subscripts cannot be accessed" 26 – "Lower bound of subscript array inaccessible" 27 – "Upper bound of subscript array inaccessible" 28 – "Must specify at least one element in table" 30 – "Position does not exist" 31 – "Illegal position specified in table"

SetDataByElements(value, indexVals)

Description	Sets the input table cell specified by <i>indexVals</i> to value. <i>indexVals</i> must contain a label for each index of the table, so the number of labels must equal NumDims . If the table has just one dimension, <i>indexVals</i> can be the label for that one index rather than an array of labels.
Return value	Boolean (success or failure)
Parameters	value: Variant, <i>indexVals</i> : Variant
Usage	<pre> IndexPtrs (1) = 1 IndexPtrs (2) = 2 RetVal = Var.DefTable.SetDataByElements (W, IndexPtrs) </pre> <p>If the table is one-dimensional, then an array is not needed:</p> <pre> W = Var.DefTable.SetDataByElements (W, 1) </pre>
API errors	24 – "Subscripts must be an array of variants" 25 – "Subscripts cannot be accessed" 26 – "Lower bound of subscript array inaccessible" 27 – "Upper bound of subscript array inaccessible" 28 – "Must specify at least one element in table" 29 – "Position specified is out of bounds" 31 – "Illegal position specified in table" 51 – "Element position is non-numeric"

SetIndexOrder(indexNames)

Description	Sets the order of the indexes in the table to the order of <i>indexNames</i> , which must contain the names of only and all the indexes of the table, assuming it has more than one index. This order determines the order used by SetDataByElements , SetDataByLabels , GetDataByElements , and GetDataByLabels to access a cell in a table.
Return value	Boolean (success or failure)
Parameters	<i>indexNames</i> – array of strings
Usage	<pre> IndexVals (1) = 'X' IndexVals (2) = 'Y' RetVal = Var.DefTable.SetIndexOrder (W, IndexVals) </pre>
API errors	24 – "Subscripts must be an array of variants" 26 – "Lower bound of subscript array inaccessible" 27 – "Upper bound of subscript array inaccessible" 28 – "Must specify at least one element in array" 52 – "Specified name is not an index of the array"

Slice(indexObj, n)

Description	Returns the <i>n</i> th slice of the table over index <i>indexObj</i> . The result is a new CATable object with one fewer dimensions than the table to which it is applied.
Parameters	<i>indexObj</i> : CAObject <i>n</i> : Integer – the 1-based slice position along index
Return value	CATable
Usage	<pre>dim In1 as CAIndex = tab.GetIndexObject("In1") dim subTab as CATable = tab.Slice(In1, 1)</pre>

Subscript(indexObj, label)

Description	Returns a slice of a table for which index <i>indexObj</i> is equal to <i>label</i> . It returns a new CATable object with one fewer dimension than the original table.
Parameters	<i>indexObj</i> : CAObject <i>label</i> : variant – the label in index
Return value	CATable
Usage	<pre>dim In1 as CAIndex = tab.GetIndexObject("In1") dim subTab as CATable = tab.Slice(In1, "SomeLabel")</pre>

Update

Description	<i>For definition tables:</i> Updates an existing input table in the definition attribute of an Analytica object. Use this method after setting one or more SetDataBy methods to direct the API to send the new table data to the Analytica Decision Engine Server. <i>For result tables:</i> Retrieves an updated version of the result table from the Analytica Decision Engine Server.
Return value	Boolean (success or failure)
Usage	<code>Var.DefTable.Update</code>
Remarks	Use the CreateDefTable method to replace the current definition attribute of an Analytica object with an input table.

Class CAIndex

Properties**IndexElements**

Description	Returns the number of elements in the index.
Data type	integer
Access	read
Usage	<code>x = theIndex.IndexElements</code>

Name

Description	Contains the name given to the Analytica index.
Data type	string
Access	read/write
Usage	<code>theName = theIndex.Name</code>

API errors 41 – "Invalid name for object"

RenderingStyle

Description Contains a **CARenderingStyle** object that controls how values are returned from **GetValueByNumber**.

Data type **CARenderingStyle**

Access read/write

Usage `theIndex.RenderingStyle.StringQuotes = 1`

Methods

GetNumberByValue

Description Returns the position of an index label in an index.

Parameters **Value** – variant

Return value integer

Usage `n = theIndex.GetNumberByValue (Value)`

API errors 22 – "Value not found in index"

GetValueByNumber

Description Returns the index label at the specified position in the index.

Parameters **Number** – integer

Return value variant

Usage `w = theIndex.GetValueByNumber (Number)`

API errors 23 – "Illegal position in index"

Class CARenderingStyle

Properties

GeneralExpression

Description Determines how string values are interpreted when they are written to a definition table. When True (default), a string is taken to be an expression, which must parse to be a valid expression. When False, a string value is stored as a literal string. For example, the value "Pi" would be interpreted as the identifier **Pi** when **GeneralExpression** is true, and would thus evaluate to 3.141592654, but the same string would be interpreted as a literal character string when **GeneralExpression** is false, and would evaluate to two-character textual string.

Data type Boolean

Access read/write

Usage `deftab.RenderingStyle.GeneralExpression = false`

FullPrecision

Description When True, a numeric value is rendered as text (see the **NumberAsText** property below) and has the maximum number digits needed to represent the number at full precision (usually 16). If False, a loss of precision does not occur.

Data type Boolean

Access read/write

Usage

```
dim res As CTable = obj.Evaluate("sqrt(2)")
res.RenderingStyle.NumberAsText = true
res.RenderingStyle.FullPrecision = false
dim s As String = res.AtomicValue ' returns "1.414"
res.renderingStyle.FullPrecision = true
s = res.AtomicValue ' returns "1.4142135623731"
```

NumberAsText

Description Controls whether numeric results or numeric table cell definitions are returned as floating point numbers or as formatted strings. When true, the number format for the current object controls how the number is formatted as a string (except that the **FullPrecision** property can override the number of digits in the number format).

Data type Boolean

Access read/write

Usage `deftab.RenderingStyle.NumberAsText = true`

ReferenceAsText

Description Some Analytica expressions can evaluate structured value (such as a tree) containing references. This property controls whether references are returned as **CTable** objects (containing the de-referenced value), or rendered as text. By default, they are returned as **CTable** objects. Note that in Analytica, a reference is treated as atomic, even though its dereferenced value can be array valued.

Data type Boolean

Access read/write

Usage

```
obj.SetAttribute("definition", "\ (A^t)")
def res as CTable = obj.Evaluate("\ (A^t)")
res.RenderingStyle.ReferencesAsText = false
def derefdVal as CTable = res.AtomicValue
```

SafeArrayLowerBound

Description The lower bound for safe arrays returned by **CTable::GetSafeArray**. Default is 1.

Data type Integer

Access read/write

Usage `ADE.DefaultRenderingStyle.SafeArrayLowerBound = 0`

StringQuotes

Description Controls whether textual values are returned with explicit quotation marks surrounding a string, according to its value:

- 0 = no quotes around strings, e.g., 0.25
- 1 = single quotes, e.g., '0.25'
- 2 = double quotes, e.g., "0.25"

When **NumberAsText** is true, with no quotation marks around string values, the numeric value 0.25 and the string containing the four characters "0.25" would be indistinguishable. If you want to be able to distinguish them, set the value of **StringQuotes** to 1 or 2.

Data type Short:

- 0 = no quotes

1 = 'single quotes'
2 = "double quotes"

Access read/write

Usage `deftab.RenderingStyle.StringQuotes = 2`

UndefValue

Description This property specifies the value returned when the Analytica value is **Undefined**. By default, ADE returns the special value variant **Empty** — for example, when **GetAttribute** is applied to an attribute that was not set. Some scripting languages cannot manipulate the value **Empty**. In this case, you can set **UndefValue** to something more convenient, such as Null or the empty string.

Data type Variant

Access read/write

Usage `ADE.DefaultRenderingStyle.UndefValue = Null`

HandleFormat

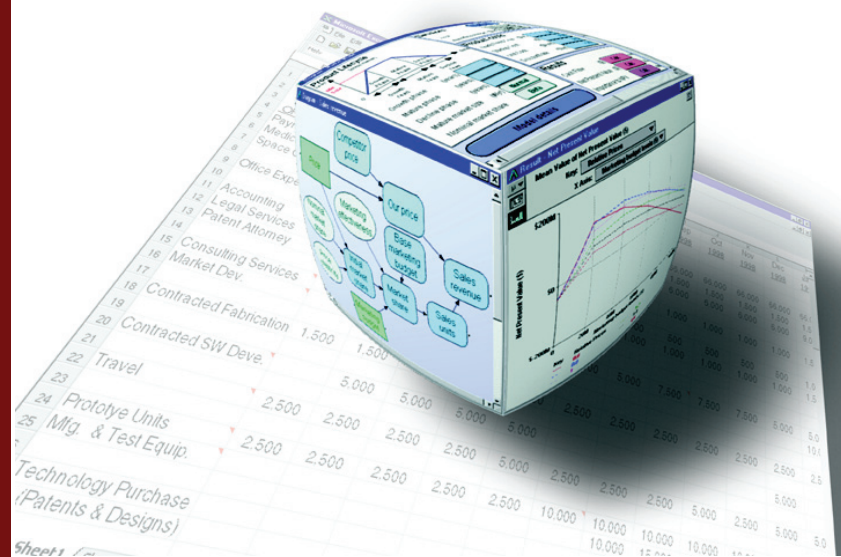
Description Controls how a **Handle** (a pointer to an Analytica object) is returned. A **Handle** can occur in a definition cell of a table if that definition consists of a single identifier. There are also some rare Analytica expressions that can produce a **Handle** in a result. If **HandleFormat** = 0, it returns the identifier of the Analytica object as a string; if 1, a **CAObject** for the Analytica object; or if 2, the title of the object as a string.

Data type Short:
0 = identifier
1 = **CAObject**
2 = title

Access read/write

Usage `deftab.RenderingStyle.HandleFormat = 1`

Appendix A



Error Codes

This appendix provides you with the complete list of ADE error messages.

ADE error codes

Error Code	Meaning/Error Text
0	"OK"
1	"Unimplemented"
2	"Warning"
3	"Lexical error"
4	"Statement error"
5	"Expression error"
6	"Execution error"
7	"System error"
8	"Fatal error"
9	"Undefined variable error"
10	"Aborted"
11	"Analytica expired — contact Lumina"
12	<i>not currently used</i>
13	<i>not currently used</i>
14	<i>not currently used</i>
15	<i>not currently used</i>
16	<i>not currently used</i>
17	<i>not currently used</i>
18	
19	<i>not currently used</i>
20	"Analytica is uninitialized"

API error codes

Error Code	Meaning/Error Text
21	"Bad parameter passed"
22	"Value not found in index"

Error Code	Meaning/Error Text
23	"Illegal position in index"
24	"Subscripts must be an array of variants"
25	"Subscript cannot be accessed"
26	"Lower bound of subscript array inaccessible"
27	"Upper bound of subscript array inaccessible"
28	"Must specify at least one element in table"
29	"Position specified is out of bounds"
30	"Position does not exist"
31	"Illegal position specified in table"
32	"Index object not found"
33	"Illegal index number specified"
34	"Definition table not found"
35	"Attribute could not be retrieved"
36	"Attribute could not be set"
37	"Could not retrieve result"
38	"Could not get result table"
39	"Module/Model/Script could not be found"
40	"Object could not be created"
41	"Invalid name for object"
42	"Object name already in use"
43	"Current module could not be retrieved"
44	"Module could not be set"
45	"Model could not be created"
46	"Model/Module could not be saved"
47	"Illegal command"
48	"Invalid object class"
49	"There is no model in memory to save"
50	"Safe-array has incorrect size or number of dimensions"
51	"Element position is non-numeric"
52	"Specified name is not an index of the array"
53	"Insufficient memory"
54	"Unrecognized MIME type"
55	"Value is not atomic"
56	"Operation allowed only on a result CTable, not on a definition table"
57	"First param is not an IStream**"

Error Code	Meaning/Error Text
58	"Subscript array contains the wrong number of elements There should be one element for each dimension"
59	"CAtable is not associated with an object, so it cannot be updated"
60	"A result table is read-only"
61	"Expression could not be parsed"
62	"Error evaluating expression"
63	"GraphWidth and GraphHeight must be positive"
64	"Value is atomic (not an array) To get value, use AtomicValue method"
65	"Index value could not be computed"
66	"No picture stored with object"
67	"Internal picture format not supported"
68	"Filename too long"
69	"Result cannot be graphed"
70	"Definition is Hidden"
71	The maximum number of CAEngine instances allowed by license are already in use
72	Invalid license code
73	Stale license code
74	Expired license code
75	No ADE license code is present. Re-install ADE
76	Not an application license code
77	Method Evaluation Time Limit exceeded
78	Computation aborted

Symbols

_ (underscore) 11

A

AddModule method 51

ADE.exe

launching 22

vs. Adew.dll 2

AdeTest program

about 7

dialog 24

working with 23

Adew.dll

loading 22

vs. ADE.exe 2

application license codes 56

applications, writing 10

arbitrary expression, parsing 36

architecture, server class 22

arrays

about 3

first element 37

obtaining scalar value 40

retrieving tables 39

asp_exam program

about 7

using 25

ASP.NET pages, generating graphs 48

atomic values

about 3

controlling formats 16

rendering 34

tables 14

AtomicValue method 61

attributes

about 14

getting 2, 14, 33

setting 2

Automation interface

C++ and C# 27

using 27

Visual Basic and VBScript 27

vs. COM interface 12, 22

AvailableLicenseInstances property 55

C

C#

calling conventions 22

sending typescript commands 28

C++

calling conventions 22

sending typescript commands 29

using the COM interface 26

CAEngine class

about 2, 22

instantiating 26, 27

methods 51–54

obtaining objects 12

Index

- opening models 13
- properties 50
- CALIndex class
 - about 3, 22
 - getting information 15
 - methods 66
 - properties 65
- CALicense
 - about 3
- CALicense class
 - about 54
 - methods 56
 - properties 55
- calling conventions 22
- CanUseOptimizer property 55
- CAObject class
 - about 2, 22
 - computing results 34
 - creating objects 33
 - methods 58–60
 - obtaining objects 33
 - properties 56
- CARenderingStyle class
 - about 3, 22
 - properties 66–68
- CATable class
 - about 3, 22
 - atomic value formats 16
 - dimensions 3
 - getting information 15
 - methods 61–65
 - properties 60
- CD, installation files 6
- cells
 - about 3
 - literal strings 46
 - populating 45
 - setting values 45
- Class Analytica 2
- classes
 - CAEngine, *see* CAEngine class
 - CALIndex, *see* CALIndex class
 - CAObject, *see* CAObject class
 - CARenderingStyle, *see* CARenderingStyle class
 - CATable, *see* CATable class
 - five main 2, 22
 - module 2
 - OLE object 2
 - reference 49
- ClassType property 56
- CloseModel method 52
- COM interface
 - projects in C++ 26
 - releasing in .NET 26
 - using 25
 - vs. Automation interface 12, 22
- Command property 50
- CreateDefTable method 58
- CreateModel method 52
- CreateObject method 52
- CurrentModule property 50
- D**
- data types 44
- DefaultDefTableRenderingStyle property 50
- DefaultRenderingStyle property 50
- definition tables
 - about 41
 - creating from scratch 44
 - getting 41
 - setting cell values 45
 - using 41
- DefinitionType property 57
- DefTable method 58
- DeleteObject method 52
- deterministic values
 - computing 34
 - obtaining 14
- dimensions
 - determining for tables 14
 - reducing 41
- E**
- edit tables
 - see also* definition tables
 - ordering elements 42
 - using 41
- elements
 - controlling format 38
 - first in arrays 37
 - getting the number in an index 40
 - retrieving 15
 - retrieving by index labels 38
 - retrieving by index order 37
 - setting by index labels 43
 - setting by index order 42
- ErrorCode property 51
- ErrorCode property, of CALicense 55
- errors
 - codes and descriptions 70–72
 - handling 30
 - script files 32
- ErrorText property 51
- ErrorText property, of CALicense 55
- Evaluate method 58
- excel_exam program
 - about 7
 - using 25
- expressions, parsing 36
- F**
- FullPrecision property 66
- G**
- GeneralExpression property 66
- Get method 52
- GetAttribute method 59
- GetDataByElements method 61

- GetDataByLabels method 61
- GetIndexObject method 62
- GetNumberByValue method 66
- GetObjectByName method 52
- GetSafeArray method 62
- GetValueByNumber method 66
- GraphHeight property 60
- graphs
 - dimensions 48, 60
 - generating from ASP.NET pages 48
 - obtaining 47
 - selecting options 18
 - working with 18, 47
 - writing to files 48
- GraphToFile method 62
- GraphToStream method 63
- GraphWidth property 60

H

- HandleFormat property 68

I

- identifiers
 - accessing 33
 - distinguishing from titles 11
 - renaming 33
 - showing in influence diagrams 11
- index labels
 - getting at a specified position 40
 - getting position 40
 - setting elements by 43
- IndexElements property 65
- indexes
 - getting from tables 14
 - getting information 15
 - names associated with tables 39
 - number of elements 40
 - number of table cells 35
 - setting order 36
 - table dimensionality 35
- IndexName method 63
- IndexNames method 63
- influence diagrams, showing identifiers 11
- in-process vs.out-of-process servers 22
- installation
 - example programs 7
 - from CD 6
 - from network 6
 - list of ADE files 6
 - manuals 7
 - obtaining files 6
 - prerequisites 6
 - removing ADE 7
 - system requirements 6
 - upgrades from earlier versions 7

J

- J#, sending typescript commands 29
- JScript, using the Automation interface 27

L

- language requirements 6
- licenses
 - entering new code 7
 - obtaining 6
 - refreshing stale code 6
- local servers 12
- Log property 51

M

- MaxLicenseInstances property 56
- MethodEvaluationTime property 57
- mid values
 - computing 34
 - obtaining 14
- models
 - building and editing 2
 - closing 32
 - closing without saving 33
 - graphing options 18
 - opening 13, 32
 - retrieving objects 13
 - saving in files 32
- modules
 - adding to models 32
 - classes 2
 - identifying current 33
 - saving 32
 - setting as active 33
- multi-dimensional results 35

N

- Name property
 - CAIndex class 65
 - CAObject class 57
- .NET
 - releasing objects 26
 - retrieving tables 39
 - stream wrappers 48
 - using the COM interface 25
- NewCAEngine method 56
- NumberAsText property 67
- numbers 3
- NumDims property 60

O

- objects
 - accessing or renaming identifiers 33
 - associating with tables 39
 - creating CAObject 33
 - creating in Visual Basic 12
 - defined as definition tables 41
 - deleting from models 33
 - evaluating 14
 - getting attributes 14, 33
 - getting information 15
 - modifying 16
 - obtaining 33
 - releasing in .NET 26

Index

- retrieving from models 13
 - working with 33
- OLE classes 2
- OpenModel method 53
- operating system requirements 6
- out-of-process servers
 - using 12
 - vs. in-process 22
 - web applications 23
- OutputBuffer property 51

P

- permissions
 - configuring 23
 - security exceptions 23
 - under IIS 5 23
- Photo property 51
- PictureToFile method 59
- PictureToStream method 59
- pivots, about 18
- probabilistic values
 - computing 34
 - obtaining 14
- projects, adding references 25
- properties, about 14
- PutSafeArray method 63

R

- RAM requirements 6
- ReadScript method 53
- ReferenceAsText property 67
- references, adding to projects 25
- rendering style 45
- RenderingStyle property
 - CAIndex class 66
 - CAObject class 57
 - CATable class 60
- requirements, system 6
- ResetError method 53
- result graphs, *see* graphs
- Result method 59
- results
 - as tables 36
 - atomic 14
 - computation mode 34
 - formatted 34
 - multi-dimensional 35
 - objects other than midpoints 34
 - retrieving 14, 34
 - setting type 34
 - simple 34
- ResultTable method 59
- ResultType property
 - CAObject class 57
 - CATable class 60

S

- SafeArrayLowerBound property 67
- SaveModel method 53

- SaveModuleFile method 53
- scalar values
 - computing 34
 - in tables 14
- script files
 - reading 32
 - typescript commands 32
- security permissions 23
- Send method 54
- SendCommand method 54
- servers
 - class architecture 22
 - comparison 2
 - in-process vs. out-of-process 22
 - local 12
 - out-of-process 12
- SetApplicationLicenseCode method 56
- SetAttribute method 60
- SetDataByElements method 64
- SetDataByLabels method 63
- SetIndexOrder method 64
- Slice method 65
- slices
 - about 3
 - selecting 18
- stale license code, refreshing 6
- stream wrappers for .NET 48
- StringQuotes property 67
- strings 3
- Subscript method 65
- system requirements 6

T

- tables
 - about 3
 - access methods 16
 - Associating objects 39
 - components 35
 - computing 34
 - conceptual model 35
 - creating 41
 - definition, *see* definition tables
 - determining features 35
 - dimensions 14
 - edit, *see* edit tables
 - getting index elements 14
 - getting information 15
 - index names 39
 - number of dimensions 39
 - retrieving elements by index labels 38
 - retrieving elements by index order 37
 - retrieving into arrays 39
 - setting elements by index labels 43
 - setting elements by index order 42
 - setting in one call 44
 - setting values 41
- TableType property 61
- TestTxc program 11
- text values 3

- titles, distinguishing from identifiers 11
- Tutorial.NET program 7
- TXC.ana model 10
- typescript language
 - about 23
 - C# code 28
 - C++/CLR code 29
 - J# code 29
 - script files 32
 - sending commands to ADE 27
 - using 27
 - VBScript code 28
 - VC++ code 29
 - Visual Basic code 28

U

- UndefValue property 68
- underscore character 11
- uninstall process 7
- Update method 65
- upgrades from earlier versions 7

V

- values
 - adjusting return 44
 - methods of retrieving 45
 - mid 14
 - numeric 45
 - probabilistic 14
 - quotation marks 45
 - rendering style 45
 - setting in tables 41
- variables
 - accessing values 3
 - classes 2
 - getting values 14
 - identifiers 11
 - redefining 17
 - titles 11
- variants 16
- VarTermFormat property 68
- VBScript
 - calling conventions 22
 - sending typescript commands 28
 - using the Automation interface 27
- VC++, sending typescript commands 29
- Visual Basic
 - array positions 37
 - calling conventions 22
 - creating ADE objects 12
 - retrieving tables 39
 - sending typescript commands 28
 - TestTxc program 11
 - using the Automation interface 27
- Visual Studio, adding references 25

W

- Windows System Installer (WSI) 6

