AIMMS

The COM Object User's Guide and Reference

AIMMS 3.8

July 15, 2008

AIMMS

The COM Object Reference

Paragon Decision Technology

Paragon Decision Technology B.V.
Julianastraat 30
2012 ES Haarlem
The Netherlands
Tel.: +31 23 5511512
Fax: +31 23 5511517

Paragon Decision Technology Inc.
5400 Carillon Point
Kirkland, WA 98033
USA
Tel.: +1 425 576 4060
Fax: +1 425 576 4061

Paragon Decision Technology Pte. Ltd.
80 Raffles Place
UOB Plaza 1, Level 36-01
Singapore 048624
Tel.: +65 96404182

Email: info@aimms.com
WWW: www.aimms.com

This documentation was typeset by Paragon Decision Technology B.V. using LATEX and the LUCIDA font family.

Contents

Part I

The AIMMS COM Object Example

Chapter 1

An AIMMS COM Object Example

1.1 Introduction

This chapter

AIMMS comes with an AIMMS COM object, which makes it possible to easily integrate your AIMMS models with programming languages like Visual Basic. An example of how to use this AIMMS COM object is also provided. This chapter describes how to use this example.

Assumptions

In order to be able to understand the example, you must have some knowledge of the Visual Basic language and of AIMMS.

Two example projects

The example consists of two projects, namely an AIMMS project and a Visual Basic project. The AIMMS project is called `Islands.prj` and the Visual Basic project is called `AIMMS_COM_example.vbp`. Using the AIMMS COM object, you can run, analyze and change the AIMMS project from within the Visual Basic project.

The Visual Basic example

In the Visual Basic example, various calls to functions provided by the AIMMS COM object are used. Not all functions are included in the example, but they should still give you a good idea of how the AIMMS COM object can be used. Technical details are provided as comments in the source code of the example (where necessary).

A closer look

However, before digging into the details of the AIMMS COM object, let's first have a closer look at the AIMMS `Island` project...

1.2 The AIMMS Island project

The Canary Islands

Not far from the west coast of Africa lie seven islands that are known as the Canary Islands. These islands are a part of Spain and are called Tenerife, Gran Canaria, Fuerteventura, Lanzarote, La Palma, La Gomera and El Hierro. See figure 1.1 for a map of the islands.

Figure 1.1: The Canary Islands

The airline company and its planes

Consider an airline company that wants to operate in the region. The company currently has three types of planes, all of which have different numbers of seats, fixed costs per flight and costs per flown mile. These parameters are displayed in table 1.1.

Plane types	ATR-72	Boeing 737	Fokker F-50
Number of seats	50	130	60
Cost per flight	600	1200	670
Cost per mile	20	65	28

Table 1.1: Initial plane characteristics

Environmental limitations

Due to environmental considerations, the government has put a limit on the number of daily flights of each type of airplane. The more polluting an aircraft, the less flights there are permitted per day using that particular aircraft. The initial flight limits for the three plane types are listed in table 1.2.

Limitations on the number of flights

The company wants to offer direct flights from every island to every other island at a reasonable ticket price. The government has put a limitation on the number of daily flights the company may offer between any two islands, regardless of the plane type. This means that, in order to satisfy passenger demand, the company may be forced to use a more uneconomic plane for a particular flight. The maximum number of passengers that travels between any two islands is assumed to be constant. The company always wants to transport all passengers.

Mandatory return flights

Another limitation on the schedule is the fact that if a flight from island A to island B is performed using a certain number of flights with plane type C, the same number of return flights must take place using the same plane type.

Plane types	ATR-72	Boeing 737	Fokker F-50
Flight limit	50	7	20

Table 1.2: Initial flight limits

Rotating schedule

Of course, this limitation has the desired effect that at the start of each day, the same number of planes of a certain type will be available on an island and therefore the same schedule can be performed over and over again...

The best flight schedule

A model written in AIMMS helps in determining the daily flight schedule that maximizes the profit. The initial data of the model consists of the data in table 1.1 and 1.2, along with the fixed passenger demand and the ticket prices of every possible flight. The schedule specifies how many flights with a certain plane type must be performed between a combination of islands to reach the maximum profit. For example, in the schedule which uses the initial data, the flight between Tenerife and Fuerteventura is performed three times per day; twice using the Fokker-50 and once using the ATR-72 (see the third and fourth line from below in figure 1.2).

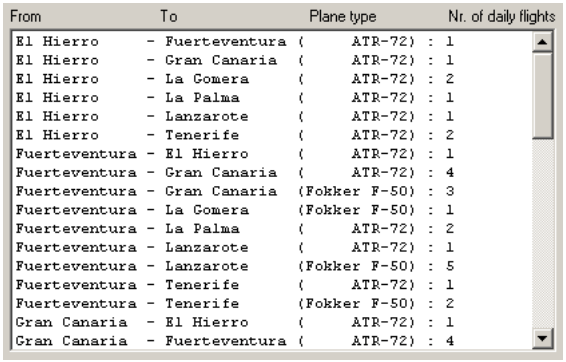

Figure 1.2: A fragment of the initial schedule

Initial profit

Using the initial data, the best schedule offers a profit of 84.404 dollars per day.

1.3 Model details

Three model sections

The `Island` model is divided into three sections (see figure 1.3):

1. `Island section`
2. `Plane section`
3. `Schedule section`

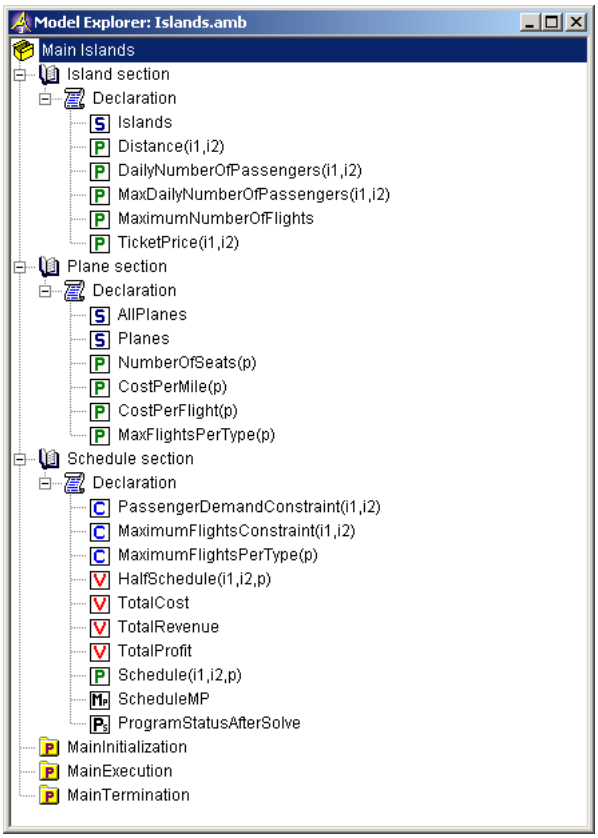

Figure 1.3: The three sections of the `Island` project

The island section

The island section contains the sets and parameters that describe the seven Canary Islands and their various relationships. The distances between all island combinations are defined here, together with the passenger demand and ticket price for each flight.

The plane section

The plane section provides all plane related information, namely the possible and actual plane types the company has in operation, together with some parameters that describe them.

The schedule section

Finally, the schedule section provides the constraints, variables and the mathematical program needed to solve the model. The schedule that will eventually be calculated is represented by the variable `schedule(i1, i2, p)`. For example, using the initial data, the value of

```
Schedule('La Palma', 'Tenerife', 'ATR-72')
```

is 6, which means that in order to reach the maximum profit every day 6 flights between La Palma and Tenerife have to be flown using the ATR-72 plane type.

Detailed comments

Every identifier in the model comes with detailed comments that describe its precise purpose. Please note the non-standard code in the MainTermination procedure that prevents the displaying of an AIMMS dialog at termination time, which would be annoying in the Visual Basic example that uses AIMMS in *hidden* mode.

Initial data

When the project starts up, the initial data is automatically loaded through the AIMMS project option `Startup case`.

1.4 The Visual Basic application

Introducing the Visual Basic application

A small example application in Visual Basic is provided with the `Island` project, to demonstrate the use of the AIMMS COM object from within Visual Basic. It consists of various windows, of which the main window is called `AIMMS COM example`. When you run the application, this window will show up automatically. To avoid errors locating the `Island` project, please start up the Visual Basic project `AIMMS_COM_example.vbp` by double clicking on the project name in the folder where the project is located, instead of first starting Visual Basic and then selecting the project.

Starting and solving

To do anything useful with the application, please click on the **Start AIMMS** button first. Doing this starts an AIMMS session with the `Island` project in the background (*hidden* mode). To verify that all went right, you can click on the **Solve Model** button to solve the model with the initial data. The **profit** field should come up with a value of 84.404 dollars. The field next to the profit-field shows the status of the mathematical program in AIMMS after solving. Please note that the time to solve the model is dependent on the machine you're using and on the solver that is used by AIMMS. When you've started AIMMS, the **Plane types** field and the **project** field are filled with the initial plane types and the full path of the `Island` project, respectively. Figure 1.4 shows the main window of the Visual Basic example, after solving the model with the initial data.

Various actions

For the sake of demonstrating some of the uses of the AIMMS COM object, various actions have been defined in the application, that can be initiated from the main window by clicking on the corresponding command buttons. Those actions are described in the next four subsections.

Learn by viewing the code

As already stated in the introduction, in order to really learn how to use the various calls to the AIMMS COM object, you should look in the subs of the Visual Basic example project. All subs have been commented properly, so studying the code should give you the necessary understanding of how to use the functions provided by the COM object in your own Visual Basic projects.

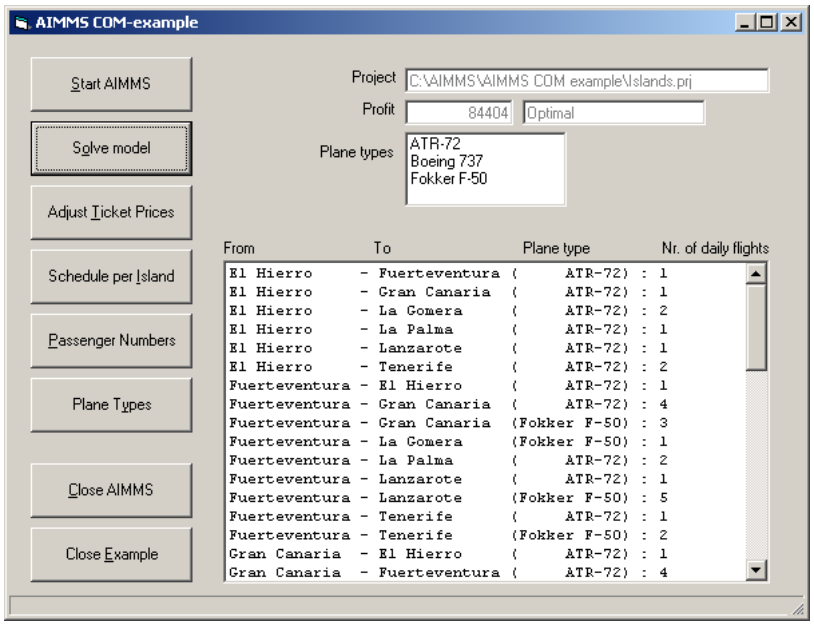

Figure 1.4: The main window of the Visual Basic project

1.4.1 Adjust ticket prices

Opening the window

Clicking on the **Adjust Ticket Prices** button brings up a window in which you can alter the ticket prices for flights that you select. Figure 1.5 shows this window.

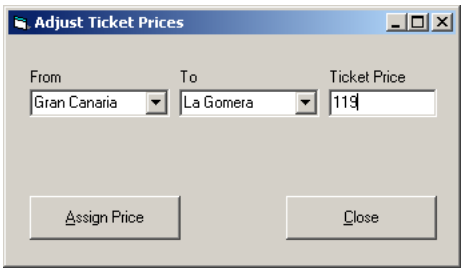

Figure 1.5: The window for adjusting ticket prices

Assigning a new price

By selecting two different islands in the **From** and **To** combo boxes, the current ticket price for the flight between those selected islands appears in the edit field **Ticket Price**. You can change this price. Click on **Assign Price** to make sure that all identifiers in the AIMMS model that rely on the ticket prices are updated.

Seeing the effects

If you like, you can change some more ticket prices before clicking on the **Close** button to return to the main window. In the main window, you can see the effects of your changes by clicking on the **solve** button. Both the schedule and the profit may have been influenced by your actions.

1.4.2 Schedule per island

Opening the window

To see just a slice of the complete schedule that you've created by solving the AIMMS model, click on the **Schedule per Island** button.

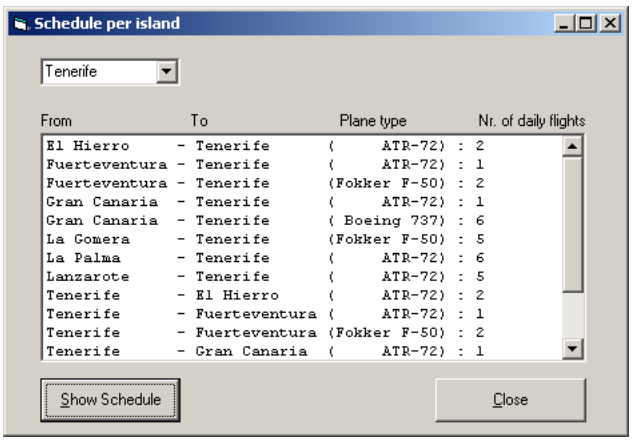

Figure 1.6: The schedule for the island of Tenerife

Displaying the schedule

In the window that pops up, you can pick an island from a list. By clicking on **Show Schedule** only those rows of the complete schedule that involve the selected island are shown. This function makes it easier to see the effects of the changes that you apply to the initial model data and at the same time demonstrates the use of selecting sliced identifiers from within Visual Basic. Figure 1.6 shows the schedule for the island of Tenerife with the initial data.

1.4.3 Passenger numbers

Opening the window

The daily number of passengers that wants to travel from or to a particular island can be changed by clicking on the **Passenger Numbers** button.

The current passenger numbers

A window appears, in which you can pick a desired island. After doing this, clicking on **Show** shows all the daily numbers of passengers that want to fly to and from the selected island. The total number of passengers involving the picked island is also displayed. Figure 1.7 shows the initial passenger numbers for flights involving the island of El Hierro.

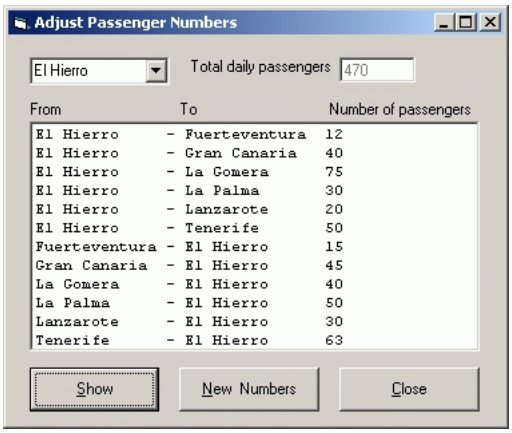

Figure 1.7: The initial passenger numbers involving flights from and to the island of El Hierro

Altering the passenger numbers

By clicking on the **New Numbers** button the computer randomly changes all displayed passenger numbers slightly, adding a value between -10 and +10 to the current number. When you've changed the passenger numbers the way you wanted, click on the **Close** button to return to the main window. You can solve the model again, using the changed passenger numbers to see the effects.

When the model becomes infeasible

In some cases the model can become infeasible. For example, if the number of passengers between two islands has grown extensively, it is possible that under the current constraints a feasible schedule cannot be determined. In that case a profit of 0 dollars is returned by the AIMMS COM object and the field next to the profit field reads **Infeasible**.

1.4.4 Plane types

Opening the window

The initial model data assumes that the company has three types of airplane in operation. You can influence this by clicking on the **Plane Types** button. By doing so, a window appears in which you can pick either an existing plane type or a new plane type from a standard list. It's also possible to introduce a completely new plane type that is not on the list. Figure 1.8 shows the window after just having introduced the `Casa CN-235` plane.

Removing or changing

When you select a plane type from the list that is already used by the company, you have two possibilities: you can *remove* this plane type (having the effect that the schedule must be recalculated using only the remaining plane types, if possible) or *change* some parameter(s) of this plane type.

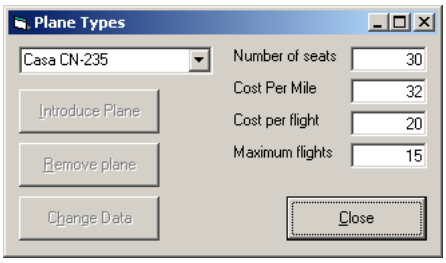

Figure 1.8: The **Plane Types** window

The plane parameters

Each type of airplane has four parameters that influence whether it's selected for certain flights or not. You can specify or change these parameters. They are:

- the number of seats,
- the cost of flying one mile,
- the static cost of performing a flight and
- the governmental limit on the number of flights allowed.

Applying the changes

When you've changed the numbers, click on the **Change Data** button to update the values in the AIMMS model. By closing the plane type window and returning to the main window, you can see the effects of your changes by solving the AIMMS model again.

Introducing a new plane

When you select a plane type from the list or type in a completely new plane type, you must fill the four parameters mentioned above in order to be able to introduce the new plane type. Again, return to the main window to see the effects of your changes by solving the AIMMS model.

Impact on solving time

Please note that changing the parameters can influence solving time considerably. It seems fastest to experiment with small changes in the existing plane types or introducing aircraft with a relatively low number of seats.

Part II

The AIMMS COM Object Reference

Chapter 2

Introduction to the AIMMS COM Object

2.1 The AIMMS COM object

The AIMMS *COM object*

The AIMMS COM object defines a number of COM interfaces, which allow you to easily integrate AIMMS projects into languages such as Visual Basic, Visual Basic .NET, or any of the other .NET languages, and scripting languages such as VBScript.

AIMMS *COM interface*

The AIMMS COM object defines four COM interfaces:

- `Aimms.Project`,
- `Aimms.Identifier`,
- `Aimms.Set`, and
- `Aimms.Procedure`

Of these four interfaces, only the `Aimms.Project` interface is registered as a real COM object, in the sense that you can create it via a call to a standard object creator function (for example through the `New` operator, or the `CreateObject` procedure in Visual Basic). The other three AIMMS COM interfaces can only be created via methods in the `Aimms.Project` interface. This may seem odd at first, but these objects correspond directly to model identifiers, and thus can only exist within the context of an `Aimms.Project` object.

Version dependent interfaces

The AIMMS COM object has been made version dependent as of AIMMS 3.6. The interpretation of this remark is as follows.

- If you are using a so-called ProgID to creating an `Aimms.Project` interface of the AIMMS COM object, the following rules apply:
  - the ProgID "`Aimms.Project.3.6`" will open the AIMMS 3.6 COM object,
  - the ProgID "`Aimms.Project.1`" will open an AIMMS COM object corresponding to AIMMS 3.5 or earlier. Which AIMMS COM version depends on which AIMMS version $\leq$ 3.5 has been installed latest.
  - the ProgID "`Aimms.Project`" will open the AIMMS COM object associated with the latest installed AIMMS version.

- If you are creating an Aimms.Project interface of the AIMMS COM object via its associated CLSID (i.e. a GUID associated with the Aimms.Project object) or as through a reference in a VB or .NET application, the following rules apply:
  - if you are referencing the CLSID of AIMMS 3.6 (directly or indirectly via a reference to the AIMMS 3.6 COM object), this will always lead to the use of the AIMMS 3.6 COM object
  - if you are referencing the CLSID of AIMMS 3.5 or lower, the COM object of the latest installed AIMMS version ≤ 3.5 will be used

The benefits for you as a developer are clear. If you want to enforce the use of the AIMMS 3.6 COM object, you can use the "Aimms.Project.3.6" ProgID or the associated CLSID. If the AIMMS COM of any AIMMS version suffices, just use the "Aimms.Project" ProgID to create an Aimms.Project interface.

Example and reference

The use of the AIMMS COM object is rather straightforward, as illustrated in the example in Chapter 1. All properties and methods corresponding to the four AIMMS COM interfaces are discussed in full detail in the subsequent chapters.

2.2 The Variant data type

Variant

Many of the properties and methods of the AIMMS COM interface make use of the Variant data type. This is a special data type that is commonly used in COM interfaces and can hold various other data types. The AIMMS COM interface uses this type to generalize the methods and functions and to avoid that methods and properties have different versions based on the data type that is used. The Variant type is used to:

- pass values from numerical identifiers, string parameters and element parameters, and
- to pass set elements either by an integer number or by string.

Identifier values

AIMMS supports three main data types for its identifiers: numerical, string and element valued. In the AIMMS COM interface this means that if you want to refer to the value of an identifier this can either be a double, a string or an integer. Therefore, instead of providing three separate methods for each type, the methods concerning identifier values use the Variant type. The context of a specific call then determines whether this Variant contains a double, a string or an integer. Although a method uses a Variant type, you can still pass the required double, string or integer directly. For example, to assign new strings to an indexed string parameter, you can pass an array of Variants containing strings, or simply pass an array of strings. At runtime, the method itself checks whether the given argument can be converted to the required data type.

Set elements

To communicate set elements back and forth, the AIMMS COM offers three different modes

- using unique integer element numbers,
- using integer ordinal numbers, or
- using element names (i.e. strings).

Therefore, in the methods where elements or tuples of elements are passed, the corresponding argument is usually of type `Variant`, and special properties determine whether these `Variants` contain integers or strings. Similar as with the identifier values your may circumvent the `Variant` type and directly pass a string or integer.

2.3 Using the AIMMS COM interfaces

You can access the AIMMS COM interface from within any language or program that supports COM technology. The two most important and widely used languages are perhaps Visual Basic or any of the .NET languages, because these languages and the derived scripting languages are used frequently in spreadsheets and HTML pages.

Visual Basic and .NET languages

In this document we describe how the interface looks from a Visual Basic point of view. You can also use the AIMMS COM object from any of the .NET languages in a manner very similar to that use from within Visual Basic.

Other languages

If you are using another language (such as C/C++), you should refer to the documentation of that language to learn how to access COM objects. In that case it may be useful to know that the exact C/C++ style description of the object methods and properties are presented in the files `AimmsCOM.h` and `AimmsCOM_i.c`. Those two files are located in the `Api\AimmsCOM` folder in your AIMMS directory tree.

Chapter 3

The Aimms.Project object

Project

The Project object (or Aimms.Project), is the main object that you must create to access an AIMMS application. The object allows you to:

- open and close an existing AIMMS application,
- access the identifiers in your model for retrieving and/or assigning data, and
- run model procedures.

Object creation

The Project object is registered as a real COM object, in the sense that you can create it via a call to a standard object creator function (for example through the New operator, or the CreateObject procedure in Visual Basic). The other three objects in the AIMMS COM interface Identifier, Set and Procedure) can only be created via specific calls to the Project object. This may seem odd at first, but these objects correspond directly to model identifiers, and thus can only exist within the context of a Project object.

Example

The following example shows a small Visual Basic program, that opens an AIMMS project, runs the procedure MainExecution and then closes the project.

```
set MyProject = CreateObject( "Aimms.Project" )
MyProject.ProjectOpen "C:\Examples\Transport.prj"
MyProject.Run  "MainExecution"
MyProject.ProjectClose 0
```

This implementation assumes late binding, as in VBScript. Using early binding, the example could be implemented equivalently as follows.

```
Dim MyProject as Aimms.Project

set MyProject = New Aimms.Project
MyProject.ProjectOpen "C:\Examples\Transport.prj"
MyProject.Run  "MainExecution"
MyProject.ProjectClose 0
```

Methods and Properties

The Project object exposes the following methods and properties:

- Project.StartupMode
- Project.User
- Project.Data
- Project.LicenseServer
- Project.ConfigDir
- Project.ProjectOpen
- Project.ProjectClose
- Project.Name
- Project.ArrayIndexOffset
- Project.DefaultTuplePassMode
- Project.DefaultElementValuePassMode
- Project.AssignArray
- Project.RetrieveArray
- Project.CreateArray
- Project.AssignElementArray
- Project.RetrieveElementArray
- Project.CreateElementArray
- Project.AssignTable
- Project.RetrieveTable
- Project.GetControl
- Project.ReleaseControl
- Project.Value
- Project.Run
- Project.GetIdentifier
- Project.GetSet
- Project.GetProcedure

Project.StartupMode

Property

With this property of the Project object, you can modify the display mode of the AIMMS window. You can either set it before opening a project and thus set the initial mode of the AIMMS window, or you can modify it later on to minimize, maximize or restore the current AIMMS window.

Synopsis:

```
Int StartupMode
```

Remarks:

The property StartupMode is of type integer and can have any of the following values:

- STARTUP_NORMAL = 0 (default)
- STARTUP_MINIMIZED = 1
- STARTUP_MAXIMIZED = 2
- STARTUP_HIDDEN = 3

Project.User

Property

This property is identical to the `--user` or `-U` command line option of AIMMS. With this option you can specify the user name and optionally the password with which you want to log on to the system.

Synopsis:

```
String User
```

Remarks:

Setting the property `User` only has an effect before a call to `ProjectOpen`.

See also:

Chapter 16 of User's Guide, Calling AIMMS

Project.Data

Property

This property is identical to the `--data` or `-D` command line option of AIMMS. With this option you can specify the data manager file that you want to use within the project.

Synopsis:

```
String Data
```

Remarks:

Setting the property `Data` only has an effect before a call to `ProjectOpen`.

See also:

Chapter 16 of User's Guide, Calling AIMMS

Project.LicenseServer

Property

This property is identical to the `--license-server` or `-L` command line option of AIMMS. With this option you can specify the host name of the server machine where the AIMMS License Manager is running.

Synopsis:

```
String LicenseServer
```

Remarks:

Setting the property `LicenseServer` only has an effect before a call to the method `ProjectOpen`.

See also:

Chapter 16 of User's Guide, Calling AIMMS

Project.ConfigDir

Property

This property is identical to the `--config_dir` or `-C` command line option of AIMMS. With this option you can specify the directory where AIMMS should look for the various licensing files. By default AIMMS looks in the `CONFIG` directory below the top level AIMMS directory.

Synopsis:

```
String ConfigDir
```

Remarks:

Setting the property `ConfigDir` only has an effect before a call to the function `ProjectOpen`.

See also:

Chapter 16 of User's Guide, Calling AIMMS

Project.ProjectOpen

With this method you can open an existing AIMMS project. Depending on the StartUpMode the AIMMS window is created and the startup sequence of the project is executed.

Synopsis:

```
Long ProjectOpen(
        project_name,        ! (input) String
        [as_server]          ! (optional) Boolean
        )
```

Arguments:

project_name
: The full path name of the existing AIMMS project file that you want to open.

as_server (optional)
: This option is especially created for accessing AIMMS from within the script of an ASP file (Active Server Page). The option instructs AIMMS to avoid any unwanted output (windows or dialog boxes) to the screen.

Return value:

If the project is successfully opened the method returns 1, otherwise 0.

Remarks:

Before opening an AIMMS project, you can set the Project properties User, Data, StartUpMode, LicenseServer, and/or ConfigDir to make sure that AIMMS starts in the correct manner.

Example:

```
Dim TheProject As Aimms.Project
set TheProject = new Aimms.Project
TheProject.StartUpMode = STARTUP_MINIMIZED
TheProject.ProjectOpen "C:\Aimms\Projects\Transport.prj"
```

Project.ProjectClose

With this method you can close the AIMMS project.

Synopsis:

```
ProjectClose
```

Arguments:

None

Project.Name

This property holds the name of the project file as you have specified it in the call to ProjectOpen. *Property*

Synopsis:

```
String Name
```

Remarks:

The property Name is read only.

See also:

The method Project.ProjectOpen

Project.ArrayIndexOffset

Property

Some of the objects in the AIMMS COM interface have methods that create and allocate memory for a new array of values. Similar as in Visual Basic, these so-called SafeArrays use index numbering which starts at 0 or 1. In other words: should the first element in an array A be referenced as A(0) or A(1). By default, the arrays created by the AIMMS COM interface start at 0, the property ArrayIndexOffset changes this default.

Synopsis:

```
Long ArrayIndexOffset
```

Remarks:

By default this property is equal to 0, but you can change it to any other integer value. Usually, you either use 0 or 1. The property ArrayIndexOffset is similar to the "Option Base" statement in Visual Basic.

Project.DefaultTuplePassMode

Property

When assigning or retrieving values from indexed identifiers you must communicate tuple(s) of elements. The AIMMS COM interface allows you to specify these tuples in three different modes:

- as element numbers (unique numbers, which remain unchanged during the AIMMS session)
- as ordinal numbers (the ordinal position of the element in the corresponding set), or
- as element names (the name of the element as you see it in the model).

With this property you can specify your default mode for passing element tuples. When you create an `Identifier` or `Set` object, the value of the property `DefaultTuplePassMode` is used as the initial value of `Identifier.TuplePassMode` or `Set.CompoundTuplePassMode`, respectively.

Synopsis:

```
Int DefaultTuplePassMode
```

Remarks:

`DefaultTuplePassMode` is of type integer, and it can have one of the following defined values:

- `ELEMENT_BY_NUMBER = 0` (default)
- `ELEMENT_BY_ORDINAL = 1`
- `ELEMENT_BY_NAME = 2`

See also:

The properties `Identifier.TuplePassMode`, `Set.CompoundTuplePassMode`

Project.DefaultElementValuePassMode

Property

When you assign or retrieve a value of an element parameter, you are actually passing an element of an AIMMS set. The AIMMS COM interface allows you to specify this element in three different modes:

- as element number (a unique number, which remains unchanged during the AIMMS session)
- as ordinal number (the ordinal position of the element in the corresponding set), or
- as element name (the name of the element as you see it in the model).

With this property you can specify your default mode for passing element values. When you create an `Identifier` or `Set` object, the value of the property `DefaultElementValuePassMode` is used as the initial value of the property `Identifier.ElementValuePassMode` or `Set.ElementPassMode`, respectively.

Synopsis:

```
Int DefaultElementValuePassMode
```

Remarks:

`DefaultElementValuePassMode` is of type integer, and it can have one of the following defined values:

- `ELEMENT_BY_NUMBER = 0` (default)
- `ELEMENT_BY_ORDINAL = 1`
- `ELEMENT_BY_NAME = 2`

See also:

The properties `Identifier.ElementValuePassMode`, `Set.ElementPassMode`

Project.AssignArray

This method is a shortcut for calling the method AssignArray of an Identifier object directly from the Project object.

Synopsis:

```
AssignArray(
         identifier_name,     ! (input) String
         value_array,         ! (input) Variant array (Double/String/Integer)
         [transposed]         ! (optional) Boolean
         )
```

Arguments:

The arguments of this method are a combination of the arguments from Project.GetIdentifier and Identifier.AssignArray

Examples:

The following two pieces of code have an identical effect on the AIMMS project:

```
'Assigning an array using the Identifier object
Dim id As Aimms.Identifier
Set id = MyProject.GetIdentifier( "Transport" )
id.AssignArray NewValues


'Assigning an array directly from the Project objet
MyProject.AssignArray "Transport", NewValues
```

See also:

The methods Project.GetIdentifier and Identifier.AssignArray

Project.RetrieveArray

This method is a shortcut for calling the method RetrieveArray of an Identifier object directly from the Project object.

Synopsis:

```
RetrieveArray(
         identifier_name,   ! (input) String
         value_array,       ! (output) Variant array (Double/String/Integer)
         [sparse],          ! (optional) Boolean
         [transposed]       ! (optional) Boolean
         )
```

Arguments:

The arguments of this method are a combination of the arguments from Project.GetIdentifier and Identifier.RetrieveArray

Examples:

The following two pieces of code have an identical effect on the AIMMS project:

```
'Retrieving an array using the Identifier object
Dim id As Aimms.Identifier
Dim CurrentValues(4,4) As Variant
Set id = MyProject.GetIdentifier( "Transport" )
id.RetrieveArray CurrentValues


'Retrieving an array directly from the Project objet
Dim CurrentValues(4,4) As Variant
MyProject.RetrieveArray "Transport", CurrentValues
```

See also:

The methods Project.GetIdentifier and Identifier.RetrieveArray

Project.CreateArray

This method is a shortcut for calling the method CreateArray of an Identifier object directly from the Project object.

Synopsis:

```
VariantArray CreateArray(
                identifier_name,       ! (input) String
                [sparse],              ! (optional) Boolean
                [transposed]           ! (optional) Boolean
                )
```

Arguments:

The arguments of this method are a combination of the arguments from Project.GetIdentifier and Identifier.CreateArray.

Return value:

The method CreateArray returns a newly allocated array of Variant values. Whether the indices in this array start at 0 or 1 depends on the value of the property ArrayIndexOffset.

Examples:

The following two pieces of code have an identical effect on the AIMMS project:

```
'Creating an array using the Identifier object
Dim id As Aimms.Identifier
Set id = MyProject.GetIdentifier( "Transport" )
CurrentValues = id.CreateArray


'Creating an array directly from the Project objet
CurrentValues = MyProject.CreateArray( "Transport" )
```

See also:

The methods Project.GetIdentifier, Identifier.CreateArray and the property Project.ArrayIndexOffset.

Project.AssignElementArray

This method is a shortcut for calling the method AssignElementArray of an Set object directly from the Project object.

Synopsis:

```
AssignElementArray(
            set_name,         (input) String
            element_array,    (input) Variant array (String/Integer)
            [mode]            (optional) ReplaceMode
            )
```

Arguments:

The arguments are a combination of the arguments from Project.GetSet and Set.AssignElementArray

Examples:

The following two pieces of code have an identical effect on the AIMMS project:

```
'Assigning an array using the Set object
Dim s As Aimms.Set
Set s = MyProject.GetSet( "Cities" )
s.AssignElementArray MyCityNames


'Assigning an array directly from the Project objet
MyProject.AssignElementArray "Cities", MyCityNames
```

See also:

The methods Project.GetSet and Set.AssignElementArray

Project.RetrieveElementArray

This method is a shortcut for calling the method RetrieveElementArray of an Set object directly from the Project object.

Synopsis:

```
RetrieveElementArray(
            set_name,         ! (input) String
            element_array     ! (output) Variant array (String/Integer)
            )
```

Arguments:

The arguments are a combination of the arguments from Project.GetSet and Set.RetrieveElementArray

Examples:

The following two pieces of code have an identical effect on the AIMMS project:

```
'Retrieving an array using the Set object
Dim s As Aimms.Set
Set s = MyProject.GetSet( "Cities" )
s.RetrieveElementArray CurrentCityNames


'Retrieving an array directly from the Project objet
MyProject.RetrieveElementArray "Cities", CurrentCityNames
```

See also:

The methods Project.GetSet and Set.RetrieveElementArray

Project.CreateElementArray

This method is a shortcut for calling the method CreateElementArray of an Set object directly from the Project object.

Synopsis:

```
VariantArray CreateElementArray(
                         set_name      ! (input) String
                         )
```

Arguments:

The arguments are a combination of the arguments from Project.GetSet and Set.CreateElementArray

Return value:

The method CreateElementArray returns a newly allocated array containing elements.

Examples:

The following two pieces of code give the same result:

```
'Creating an array using the Set object
Dim s As Aimms.Set
Set s = MyProject.GetSet( "Cities" )
CurrentCityNames = s.CreateElementArray


'Creating an array directly from the Project objet
CurrentCityNames = MyProject.CreateElementArray( "Cities" )
```

See also:

The methods Project.GetSet and Set.CreateElementArray

Project.AssignTable

This method is a shortcut for calling the method AssignTable of an Identifier object directly from the Project object.

Synopsis:

```
AssignTable(
         identifier_name,     ! (input) String
         row_tuples,          ! (input) Variant array (String/Integer)
         column_tuples,       ! (input) Variant array (String/Integer)
         value_array          ! (input) Variant array (Double/String/Integer)
         )
```

Arguments:

The arguments of this method are a combination of the arguments from Project.GetIdentifier and Identifier.AssignTable

Examples:

The following two pieces of code have an identical effect on the AIMMS project:

```
'Assigning a table using the Identifier object
Dim id As Aimms.Identifier
Set id = MyProject.GetIdentifier( "Transport" )
id.AssignTable Rows, Columns, CurrentValues


'Assigning a table directly from the Project objet
MyProject.AssignTable "Transport", Rows, Columns, CurrentValues
```

See also:

The methods Project.GetIdentifier and Identifier.AssignTable

Project.RetrieveTable

This method is a shortcut for calling the method RetrieveTable of an Identifier object directly from the Project object.

Synopsis:

```
RetrieveTable(
           identifier_name,    ! (input) String
           row_tuples,         ! (input) Variant array (String/Integer)
           column_tuples,      ! (input) Variant array (String/Integer)
           value_array,        ! (input) Variant array (Double/String/Integer)
           [sparse],           ! (optional) Boolean
           [row_mode],         ! (optional) RowColumnMode
           [column_node]       ! (optional) RowColumnMode
           )
```

Arguments:

The arguments of this method are a combination of the arguments from Project.GetIdentifier and Identifier.RetrieveTable

Examples:

The following two pieces of code produce the same results:

```
'Retrieving a table using the Identifier object
Dim id As Aimms.Identifier
Set id = MyProject.GetIdentifier( "Transport" )
id.RetrieveTable Rows, Columns, CurrentValues


'Retrieving a table directly from the Project objet
MyProject.RetrieveTable "Transport", Rows, Columns, CurrentValues
```

See also:

The methods Project.GetIdentifier and Identifier.RetrieveTable

Project.GetControl

Whenever an AIMMS project runs in a multi-threaded environment, synchronization of the execution and data passing requests becomes of the utmost importance. By default, the AIMMS COM interface will make sure that no two execution or data requests initiated from different threads are dealt with simultaneously. In fact, any request checks whether it can get exclusive control and if not, waits for an infinite time until another thread releases its control. This works fine except for two cases:

- You want to wait only for a specified amount of time, and if this time elapses, cancel the operation.
- You want to execute a sequence of requests, without any intervention from other threads.

In both cases you should try and get the control explicitly, execute the request(s) and then release the control.

Synopsis:

```
Long GetControl(
               [timeout]       ! (optional) Long
               )
```

Arguments:

timeout (optional)
: The number of milliseconds to wait if the control cannot be obtained immediately. If this timeout argument is omitted, then the method will wait for an infinite amount of time. A timeout value of 0 means that the method will only obtain the control if it does not have to wait for it.

Return value:

If the control is obtained successfully the method returns 1, otherwise it returns 0.

Remarks:

Any successfull call to the method `GetControl` *must* be followed by a call to `ReleaseControl`. Failure to do so will result in a situation where other threads cannot get the control.

See also:

The method `Project.ReleaseControl`

Project.ReleaseControl

This method releases the control over AIMMS that was obtained via a call to `GetControl`. Any successfull call to `GetControl` *must* be followed by a call to this method.

Synopsis:

```
Release
```

Arguments:

None

See also:

The method `Project.GetControl`

Project.Value

Property

This property is a shortcut for referencing the Value property of a Scalar object directly from the Project object. Note that Value is implemented as a property and not as a method. In a Visual Basic environment, this makes assignment and retrieval of a scalar value much easier.

Synopsis:

```
Variant Value(
            identifier_name         ! (input) String
            )
```

Arguments:

identifier_name
: The name of the scalar identifier(-slice). This argument is directly related to the single argument of Project.GetIdentifier.

Return value:

The property is of type Variant, which contains either a double, string or integer depending on the type of the identifier.

Examples:

The following two pieces of code give an identical result:

```
'Assigning a value using the Identifier object
Dim id As Aimms.Identifier
Set id = MyProject.GetIdentifier( "FuelCost" )
id.Value = 2.50


'Assigning a scalar value directly from the Project object
MyProject.Value( "FuelCost" ) = 2.50
```

Project.Run

This method is a shortcut for calling the method Run of a Procedure object directly from the Project object.

Synopsis:

```
Long Run(
        procedure_name,         ! (input) String
        [var_args]              ! (optional) variable number of arguments
        )
```

Arguments:

The arguments of this method are a combination of the arguments from Project.GetProcedure and Procedure.Run

Examples:

The following two pieces of code give the same result:

```
'Running a procedure using the Procedure object
Dim proc As Aimms.Procedure
Set proc = MyProject.GetProcedure( "MainExecution" )
proc.Run


'Running a procedure directly from the Project objet
MyProject.Run "MainExecution"
```

See also:

The methods Project.GetProcedure and Procedure.Run

Project.GetIdentifier

This method creates a new Identifier object for a given identifier in the model or slice thereof.

Synopsis:

```
Aimms.Identifier GetIdentifier(
                              id_name,        ! (input) String
                              )
```

Arguments:

id_name
: The name of an identifier (slice) in the model. For example: "Transport" or "transport('Amsterdam',j)". In the latter example you may omit the free index j and specify "Transport('Amsterdam',)".

Return value:

A newly created Identifier object.

Remarks:

In Visual Basic you must use the 'Set' statement to assign the object to an object variable. See the example below:

Example:

```
Dim MyIdentifier As Aimms.Identifier
Set MyIdentifier = MyProject.GetIdentifier( "Transport" )
```

See also:

The Aimms.Identifier object.

Project.GetSet

This method creates a new Set object for a given set in the model.

Synopsis:

```
Aimms.Set GetScalar(
                  set_name      ! (input) String
                  )
```

Arguments:

set_name
The name of a set in the model.

Return value:

A newly created Set object.

Remarks:

In Visual Basic you must use the 'Set' statement to assign the object to an object variable. See the example below:

Example:

```
Dim MySet As Aimms.Set
Set MySet = MyProject.GetSet( "Cities" )
```

See also:

The Aimms.Set object.

Project.GetProcedure

This method creates a new Procedure object for a given procedure in the model.

Synopsis:

```
Aimms.Procedure GetProcedure(
                            proc_name       ! (input) String
                            )
```

Arguments:

proc_name
: The name of a procedure in the model. If the procedure has arguments then you should not include these arguments in the name.

Return value:

A newly created Procedure object.

Remarks:

In Visual Basic you must use the 'Set' statement to assign the object to an object variable. See the example below:

Example:

```
Dim MyProc As Aimms.Procedure
Set MyProc = MyProject.GetProcedure( "MainExecution" )
```

See also:

The Aimms.Procedure object.

Chapter 4

The Aimms.Identifier Object

The Identifier object allows you to access any scalar or multidimensional identifier in the AIMMS project. The Identifier object offers several properties and methods to retrieve and assign data and various identifier characteristics.

Object creation

Because an identifier only exists within the context of an AIMMS project, you cannot create an Identifier object directly using a standard COM object creator function. Instead, you must create an Identifier object via a specific call to the Aimms.Project object, namely Project.GetIdentifier.

Example

The following example shows a small Visual Basic program, that opens an AIMMS project, creates an Identifier object, and retrieves the current values for that identifier.

```
Dim MyProject As Aimms.Project
Dim Demand As Aimms.Identifier

'create project object, and open the project file
set MyProject = New Aimms.Project
MyProject.ProjectOpen "C:\Examples\Transport.prj"

'create identifier object for the AIMMS identifier 'demand'
set Demand = MyProject.GetIdentifier( "demand" )

'retrieve the current demand data
CurValues = Demand.CreateArray
```

Methods and Properties

The Identifier object exposes the following methods and properties:

- Identifier.FullDimension
- Identifier.SlicedDimension
- Identifier.Default
- Identifier.Card
- Identifier.Name
- Identifier.ReadOnly
- Identifier.ElementValuePassMode
- Identifier.TuplePassMode
- Identifier.OrdinalsOffset
- Identifier.Value
- Identifier.AssignArray

- Identifier.RetrieveArray
- Identifier.CreateArray
- Identifier.Empty
- Identifier.Update
- Identifier.Cleanup
- Identifier.GetElementRangeSet
- Identifier.GetCallDomain
- Identifier.ValueNext
- Identifier.Reset
- Identifier.AssignSparseValues
- Identifier.AssignTable
- Identifier.RetrieveTable

Identifier.FullDimension

Property

This property holds the dimension of the identifier according to its declaration in the AIMMS model. Whether the identifier is sliced has no effect on this number.

Synopsis:

```
Int FullDimension
```

Remarks:

The property `FullDimension` is read-only.

Example:

```
Dim tr As Aimms.Identifier
Set tr = MyProject.GetIdentifier( "transport('Amsterdam',)" )

' the full dimension equals 2
MsgBox tr.FullDimension
```

See also:

The property `Identifier.SlicedDimension`

Identifier.SlicedDimension

This property holds the sliced dimension of the identifier as it was specified in the call to Project.GetIdentifier. *Property*

Synopsis:

```
Int SlicedDimension
```

Remarks:

The property SlicedDimension is read-only.

For most methods of the Identifier object that involve element tuples or other dimension aspects, you must handle the object as if it is an identifier with dimension equal to SlicedDimension.

Example:

```
Dim tr As Aimms.Identifier
Set tr = MyProject.GetIdentifier( "transport('Amsterdam',)" )

If tr.SlicedDimension < tr.FullDimension Then
    MsgBox "The identifier is sliced."
End If
```

See also:

The property Identifier.FullDimension

Identifier.Default

Property

This property holds the default value of the identifier as it is specified in the declaration of the identifier in AIMMS. When you are retrieving sparse data of the identifier, any value that is equal to this default will not be included.

Synopsis:

```
Variant Default
```

Remarks:

The property `Default` is read-only and is of type `Variant`. This special data type is used frequently in a COM context and can hold various other data types such as integer, double and string. AIMMS uses this `Variant` type in a similar way to pass values of numerical identifiers, string parameters or element parameters.

Identifier type	ElementValuePassMode	Variant type
Parameter or Variable	-	Double or Integer
String Parameter	-	String
Element Parameter	ELEMENT_BY_NUMBER	Integer
	ELEMENT_BY_ORDINAL	Integer
	ELEMENT_BY_NAME	String

Table 4.1: `Variant` types used for various AIMMS identifiers

Example:

```
Dim tr As Aimms.Identifier
Set tr = MyProject.GetIdentifier( "transport('Amsterdam',)" )

' usually the default of an identifier equals 0
MsgBox tr.Default
```

See also:

The property `Identifier.SlicedDimension`

Identifier.Card

Property

The property Card provides the cardinality of the identifier(-slice). The cardinality of an identifier is defined as the total number of nondefault data values.

Synopsis:

```
Int Card
```

Remarks:

The property Card is read-only, and its value may change whenever new values are assigned to the identifier.

See also:

The property Identifier.Default

Identifier.Name

The property Name holds the name of the AIMMS identifier. Any domain or slicing information is excluded from the name. *Property*

Synopsis:

```
String Name
```

Remarks:

The property Name is read-only.

See also:

The property Project.GetIdentifier

Identifier.ReadOnly

Property

The property ReadOnly indicates whether you are allowed to make modifications to the data of the identifier. If in the AIMMS model the identifier is declared with a Definition, or if the identifier is not part of the predefined set AllUpdatableIdentifiers, then modifications are not allowed.

Synopsis:

```
Boolean ReadOnly
```

Remarks:

You cannot change the value of the property ReadOnly.

Identifier.ElementValuePassMode

Property

When you assign or retrieve a value of an element parameter, you are actually passing an element of an AIMMS set. The AIMMS COM interface allows you to specify this element in three different modes:

- as element number (a unique number, which remains unchanged during the AIMMS session)
- as ordinal number (the ordinal position of the element in the corresponding set), or
- as element name (the name of the element as you see it in the model).

With the property ElementValuePassMode you can specify how you want to pass the values for the corresponding identifier. Initially, this property inherits its value from Project.DefaultElementValuePassMode.

Synopsis:

```
Long ElementValuePassMode
```

Remarks:

This property only applies if the underlying identifier is an element parameter. It can have any of the following defined values:

- ELEMENT_BY_NUMBER = 0
- ELEMENT_BY_ORDINAL = 1
- ELEMENT_BY_NAME = 2

Example:

```
Dim CurCity As Aimms.Identifier
Dim OrdNumber As Integer
Set CurCity = MyProject.GetIdentifier( "CurrentCity" )

' assiging a value using the element name
CurCity.ElementValuePassMode = ELEMENT_BY_NAME
CurCity.Value = "Amsterdam"

' retrieving the same value as an ordinal number
CurCity.ElementValuePassMode = ELEMENT_BY_ORDINAL
OrdNumber = CurCity.Value
```

Identifier.TuplePassMode

Property

When assigning or retrieving values from indexed identifiers you must communicate tuple(s) of elements. The AIMMS COM interface allows you to specify these tuples in three different modes:

- as element numbers (unique numbers, which remain unchanged during the AIMMS session)
- as ordinal numbers (the ordinal position of the element in the corresponding set), or
- as element names (the name of the element as you see it in the model).

With the property TuplePassMode you can specify how you want to pass the values for the corresponding identifier. Initially, this property inherits its value from Project.DefaultTuplePassMode.

Synopsis:

```
Long TuplePassMode
```

Remarks:

The property TuplePassMode is integer valued and can have any of the following defined values:

- ELEMENT_BY_NUMBER = 0
- ELEMENT_BY_ORDINAL = 1
- ELEMENT_BY_NAME = 2

Example:

```
Dim CurCity As Aimms.Identifier
Dim tuple(1) As String
Set CurCity = MyProject.GetIdentifier( "Transport" )

tuple(0) = "Amsterdam"
tuple(1) = "Rotterdam"
CurCity.TuplePassMode = ELEMENT_BY_NAME
CurCity.Value( tuple ) = 25.0
```

Identifier.OrdinalsOffset

Property

When passing set elements, the AIMMS COM allows you to reference these elements by their ordinal position within the set. With the property `OrdinalsOffset` you can indicate whether the first element in the set is referenced as ordinal number 0 or ordinal number 1.

Synopsis:

```
Int OrdinalsOffset
```

Remarks:

This property has an effect on:

- the passing of element tuples, if the property `TuplePassMode` is set to `ELEMENT_BY_ORDINAL`, and
- the passing of element parameter values, if the property `ElementValuePassMode` is set to `ELEMENT_BY_ORDINAL`.

See also:

The properties `Identifier.TuplePassMode` `Identifier.ElementValuePassMode`.

Identifier.Value

Property

With the property Value you can set or retrieve a single scalar value in the identifier. If the underlying identifier is not a scalar of itself, you must specify the element tuple that you want to access.

Synopsis:

```
Variant Value(
              [tuple]     ! (optional input) Variant array (String/Integer)
             }
```

Arguments:

tuple (optional)
: This array specifies the element tuple that you want to access (for data retrieval or assignment). The size of the array should match with the SlicedDimension of the identifier. Depending on the property TuplePassMode, this array should contain integers or strings. If SlicedDimension equals 0, then you may omit this argument.

Return value:

The value is given as a Variant. Table 4.1 illustrates how this data type is used in combination with the various AIMMS identifier types.

Example:

```
Dim CurCity As Aimms.Identifier
Dim TotCost As Aimms.Identifier
Dim tuple(1) As String

' assigning a single value in a 2-dimensional identifier
Set CurCity = MyProject.GetIdentifier( "Transport" )
tuple(0) = "Amsterdam"
tuple(1) = "Rotterdam"
CurCity.TuplePassMode = ELEMENT_BY_NAME
CurCity.Value( tuple ) = 25.0

' getting the value of a scalar identifier
Set TotCost = MyProject.GetIdentifier( "TotalCost" )
x = TotCost.Value
```

See also:

The properties Identifier.TuplePassMode and Identifier.SlicedDimension.

Identifier.AssignArray

With this method you can assign a (multi-dimensional) array of values to the identifier. The data values in the array will completely overwrite the current values of the identifier(-slice).

Synopsis:

```
AssignArray(
         value_array,       ! (input) Variant array (double/string/integer)
         [transposed]       ! (optional) Boolean
         )
```

Arguments:

value_array
: The array containing the new values for the identifier. The array dimensions should match the `SlicedDimension` of the identifier. The type of values in the array depend on the type of the identifier. For example: an array of string values for a string parameter, an array of doubles for a numeric parameter, and for an element parameter a type that matches the `ElementValuePassMode`. In all cases the array may be provided as an array of `Variants`. For example, for a string parameter you can either provide an array of strings, or an array of Variants containing strings.

transposed (optional)
: This argument determines whether the method expects the values for a multi-dimensional identifiers in row-wise or column-wise order. For the default value (FALSE), values must be passed in row-wise order.

Example:

```
Dim id As Aimms.Identifier
Dim random_vals(3,3) As Variant
Set id = MyProject.GetIdentifier( "Transport" )
For i=0 To 3
   For j=0 To 3
       random_vals(i,j) = Rnd
   Next j
Next i
id.AssignArray random_vals
```

See also:

The methods `Project.AssignArray`

Identifier.RetrieveArray

With this method you can retrieve an array of values from the identifier.

Synopsis:

```
RetrieveArray(
          value_array,        ! (output) Variant array (double/string/integer)
          [sparse],           ! (optional) Boolean
          [transposed]        ! (optional) Boolean
          )
```

Arguments:

value_array
: The array, that on return contains the current values of the identifier. The array dimensions should match the (sliced) dimension of the identifier. The type of values in the array depend on the type of the identifier. For example: an array of string values for a string parameter, an array of doubles for a numeric parameter, and for an element parameter a type that matches the `DefaultElementValuePassMode`. You may also provide an array of `Variants`, in that case AIMMS will fill these `Variants` with the corresponding data type.

sparse (optional)
: If this argument is set to TRUE, and the value_array is an array of `Variant` values, then when the identifier has default values, these elements are not passed as the default value, but as an 'empty' variant. This is especially usefull in spreadsheets, so that default values are not shown as zeros or empty strings, but as empty cells.

transposed (optional)
: This argument determines whether the method fills the array for a multi-dimensional identifiers in row-wise or column-wise order. For the default value (FALSE), values are passed according to a row-wise order.

Example:

```
'Retrieving an array using the Identifier object
Dim id As Aimms.Identifier
Dim CurVals(3,3) As Variant
Set id = MyProject.GetIdentifier( "Transport" )
id.RetrieveArray CurVals
```

See also:

The methods `Project.RetrieveArray`

Identifier.CreateArray

This method creates and retrieves an array with the current values of the identifier. The method is similar to RetrieveArray except that CreateArray does not require you to provide the value array. Instead the array is allocated by the method itself, and returned as the return value of the method.

Synopsis:

```
VariantArray CreateArray(
                    [sparse],         ! (optional) Boolean
                    [transposed]      ! (optional) Boolean
                    )
```

Arguments:

sparse (optional)
: If this argument is set to TRUE, then when the identifier has default values, these elements are not passed as the default value, but as an 'empty' variant. This is especially usefull in spreadsheets, so that default values are not shown as zeros or empty strings, but as empty cells.

transposed (optional)
: This argument determines whether the method fills the array for a multi-dimensional identifiers in row-wise or column-wise order. For the default value (FALSE), values are passed according to a row-wise order.

Return value:

A newly created array of Variants, containing the current values of the identifier. The dimension and sizes of the created array exactly match with the sliced dimension of the identifier and the cardiality of the domain sets. Whether the first element of the array starts at position 0 or 1 depends on the value of the property Project.ArrayIndexOffset.

Example:

```
'Retrieving an array using the Identifier object
Dim id As Aimms.Identifier
Dim CurVals() As Variant
Set id = MyProject.GetIdentifier( "Transport" )
CurVals = id.CreateArray
' Now, CurVals is a 2-dimensional array
```

See also:

The methods Project.CreateArray and Identifier.RetrieveArray

Identifier.Empty

This method empties the identifier(slice). In other words, all values are set to the default value of the identifier. Afterwards, the cardinality of the identifier is 0.

Synopsis:

```
Empty
```

Arguments:

None

See also:

The properties `Identifier.Card` and `Identifier.Default`

Identifier.Update

This method updates the values of the *entire* identifier, regardless whether the object refers to only a slice of the data.

Synopsis:

```
Update
```

Arguments:

None

See also:

The UPDATE statement

Identifier.Cleanup

This method cleans up the data of the *entire* identifier, regardless whether the object refers to only a slice of the data. The cleanup operation removes all inactive data of the identifier, i.e. the data elements that are no longer part of the domain of the identifier.

Synopsis:

```
Cleanup
```

Arguments:

None

See also:

The CLEANUP statement

Identifier.GetElementRangeSet

This method creates a new Set object for the range set of an element parameter.

Synopsis:

```
Aimms.Set GetElementRangeSet
```

Arguments:

None

Return value:

A newly created Aimms.Set object, that refers to the range set of the element parameter.

Remarks:

This method only applies if the underlying identifier is an element parameter in AIMMS.

See also:

The Aimms.Set object (see Chapter 5).

Identifier.GetCallDomain

This method returns a Set object for the n-th free domain index of the identifier.

Synopsis:

```
Aimms.Set GetCallDomain(
                        n      ! (input) Integer
                        )
```

Arguments:

n

The sequence number of the domain index for which you want to create a Set object. The numbering starts at 1, and only the indices in the sliced domain are counted.

Return value:

A newly created Aimms.Set object, that refers to the set of the n-th sliced domain index.

Example:

```
Dim id As Aimms.Identifier
Dim s1 As Aimms.Set
Dim s2 As Aimms.Set

'create a 2-dimensional slice of parameter x
set id = MyProject.GetIdentifier( "x(i,'j0',k)" )

'get a Set object for index i
set s1 = id.GetCallDomain( 1 )

'get a Set object for index k (the second dimension in the slice)
set s2 = id.GetCallDomain( 2 )
```

See also:

The Aimms.Set object (see Chapter 5).

Identifier.ValueNext

This method retrieves the next non-default value of the identifier. The method ValueNext can be used sequentially to retrieve all the non-default values of the identifier.

Synopsis:

```
Boolean ValueNext(
              tuple_array,     ! (output) Variant array (String/Integer)
              value            ! (output) Variant (Double/String/Integer)
              )
```

Arguments:

tuple_array
: The size of this array must match with the sliced dimension of the identifier. The type of the array elements depend on the property TuplePassMode and is either a string or an integer. If the method successfully retrieves the next value, then this array is filled with the corresponding tuple.

value
: If the next value exists, it is returned in this argument. The type of Variant depends on the type of the identifier. See also Table 4.1.

Return value:

ValueNext returns TRUE if there is a next value, otherwise it returns FALSE. In the latter case, the contents of both arguments tuple_array and value is undefined.

Remarks:

To access all non default values of an identifier, you first need to call the method Reset and then call ValueNext repeatedly, until it returns FALSE.

Example:

```
Dim id As Aimms.Identifier
Dim tuple(1) As Variant
Dim t As Variant
Set id = MyProject.GetIdentifier( "Transport" )
id.TuplePassMode = ELEMENT_BY_NAME
id.Reset
exists = id.ValueNext( tuple, t )
While exists
    MsgBox "(" + tuple(0) + "," + tuple(1) + ")=" + Str(t)
    exists = id.ValueNext( tuple, t )
Wend
```

See also:

The methods Project.AssignArray

Identifier.Reset

This method resets the cursor, that is used internally in the method `ValueNext`. After a call to `Reset`, the first call to `ValueNext` will return the first non-default value in the identifier(slice), or return FALSE if the identifier(slice) is empty.

Synopsis:

```
Reset
```

Arguments:

None

See also:

The method `Identifier.ValueNext`

Identifier.AssignSparseValues

With this method you can assign values for a number of explicitly given element tuples. The element tuples that are not referenced keep their current values.

Synopsis:

```
AssignSparseValues(
          nr,               ! (input) integer
          tuple_array,      ! (input) Variant array (String/Integer)
          value_array       ! (input) Variant array (Double/String/Integer)
          )
```

Arguments:

nr
: The number of values that you want to assign.

tuple_array
: A two-dimensional array of size: nr × SlicedDimension. The array should contain nr tuples. Depending on the value of TuplePassMode, the elements in these tuples are passed as integers or strings.

value_array
: A array of size nr. For each tuple defined in tuple_array, value_array holds the corresponding new value. The type of this value depends on the type of the identifier.

See also:

The properties Identifier.Value and Identifier.AssignArray

Identifier.AssignTable

With this method you can assign values to the identifier using a tabular format of rows and columns. For both the rows and columns you specify the (tuples of) elements for which you want to assign the data.

Synopsis:

```
AssignTable(
      row_tuples,         ! (input) Variant array (String/Integer)
      column_tuples,      ! (input) Variant array (String/Integer)
      value_array         ! (input) Variant array (Double/String/Integer)
      )
```

Arguments:

row_tuples
: A two-dimensional matrix of size $m \times r$. In this matrix every row contains a tuple of elements (or a single element in case $r = 1$).

column_tuples
: A two-dimensional matrix of size $c \times n$. In this matrix every column contains a tuple of elements (or a single element in case $c = 1$).

value_array
: A two-dimensional array of size $m \times n$, containing the new values for the identifier.

Remarks:

In the simplest form the sliced dimension of the identifier is 2, so that both the rows and columns are represented by single elements. The first index is presented in the rows and the second index in the columns. In that case both r and c are 1 and the tuples arrays can be seen as 1-dimensional arrays. The table below illustrates this for the case where m=3 and n=4:

(1×4)

j1	j2	j3	j4

i1
i2
i3

(3×1)

11	12	13	14
21	22	23	24
31	32	33	34

(3×4)

In a general situation the sliced dimension is split in two parts: the first r indices are presented in the rows, and the remaining c indices in the columns. The example below illustrates this for a sliced dimension of 4, both r and c are 2, and m=3 and n=4:

(2×4)

k1	k1	k3	k4
l1	l2	l3	l1

i1	j1
i2	j3
i2	j4

(3×2)

11.11	11.12	11.33	11.41
23.11	23.12	23.33	23.41
24.11	24.12	24.33	24.41

(3×4)

If $r = 1$ or $c = 1$, the `row_tuples` resp. `column_tuples` does not have to be a 2-dimensional matrix. Instead you can specify it as a 1-dimensional array.

See also:

The methods `Project.AssignTable` `Identifier.RetrieveTable`

Identifier.RetrieveTable

With this method you can retrieve the values of the identifier using a tabular format of rows and columns. For both the rows and columns you pass matrices that (on return) contain the element tuples. You can either specify which rows and/or columns you want to retrieve, or let the method itself fill in the rows and columns.

Synopsis:

```
RetrieveTable(
       row_tuples,         ! (input or output) Variant array (String/Integer)
       column_tuples,      ! (input or output) Variant array (String/Integer)
       value_array,        ! (output) Variant array (Double/String/Integer)
       [sparse],           ! (optional) Boolean
       [row_mode],         ! (optional) Integer
       [column_mode]       ! (optional) Integer
       )
```

Arguments:

row_tuples
: A two-dimensional matrix of size $m \times r$. In this matrix every row contains a tuple of elements (or a single element in case $r = 1$).

column_tuples
: A two-dimensional matrix of size $c \times n$. In this matrix every column contains a tuple of elements (or a single element in case $c = 1$).

value_array
: A two-dimensional array of size $m \times n$, to hold the values of the identifier. The type of values in the array depend on the type of the identifier. For example: an array of string values for a string parameter, an array of doubles for a numeric parameter, and for an element parameter a type that matches the `DefaultElementValuePassMode`. You may also provide an array of `Variants`, in that case AIMMS will fill these `Variants` with the corresponding data type.

sparse (optional)
: If this argument is set to TRUE, and the value_array is an array of `Variant` values, then when the identifier has default values, these elements are not passed as the default value, but as an 'empty' variant. This is especially useful in spreadsheets, so that default values are not shown as zeros or empty strings, but as empty cells.

row_mode
: An integer that indicates how the `row_tuples` array should be interpreted. The value should be one of the table below.

column_mode

An integer that indicates how the col_tuples array should be interpreted. The possible values are the same as for row_mode.

SPARSE_OUTPUT	0	In this mode the tuples array (row or column) is regarded as output, and is thus filled by the method. The method will search for non-default values and only fill the array with tuples for which a non-default value exists.
DENSE_OUTPUT	1	In this mode the tuples array (row or column) is regarded as output, and is thus filled by the method. The method will fill the tuples array will all possible tuples, using the entire content of the domain sets.
USER_INPUT	2	In this mode the tuples array (row or column) is regarded as input, and the method will only retrieve values for the tuples that are given in this array.
NON_EXISTING	3	If the row_mode is set to NON_EXISTING, then the entire element tuple should be given in the column_tuples array (and vice-versa).

Table 4.2: Values for row_mode and column_mode

Remarks:

The dimensions of the three arrays row_tuples, column_tuples and value_array should match in a similar was as for Identifier.AssignTable.

See also:

The methods Project.RetrieveTable Identifier.AssignTable

Chapter 5

The Aimms.Set Object

The Set object allows you to access a set in the AIMMS project. The Set object offers several properties and methods to retrieve or modify the contents of the set.

Object creation

Because a set only exists within the context of an AIMMS project, you cannot create a Set object directly using a standard COM object creator function. Instead, you must create a Set object via a specific call to the Aimms.Project object, namely Project.GetSet. In addition, you can create a Set object via methods of the Identifier object, namely GetCallDomain and GetElementRangeSet.

Example

The following example shows a small Visual Basic program, that opens an AIMMS project, creates a Set object, and retrieves the current contents of that set.

```
Dim MyProject As Aimms.Project
Dim Cities As Aimms.Set

'create project object, and open the project file
set MyProject = New Aimms.Project
MyProject.ProjectOpen "C:\Examples\Transport.prj"

'create set object for the AIMMS identifier 'Cities'
set Cities = MyProject.GetSet( "Cities" )

'retrieve the current set content
AllCities = Cities.CreateElementArray
```

Methods and Properties

The Set object exposes the following methods and properties:

- Set.Card
- Set.Dimension
- Set.ElementPassMode
- Set.Name
- Set.OrdinalsOffset
- Set.ReadOnly
- Set.AddElement
- Set.AssignElementArray
- Set.RetrieveElementArray

- Set.CreateElementArray
- Set.DeleteElement
- Set.ElementToName
- Set.ElementToOrdinal
- Set.OrdinalToName
- Set.OrdinalToElement
- Set.NameToElement
- Set.NameToOrdinal
- Set.RenameElement
- Set.CompoundAddElementTuple
- Set.CompoundDimension
- Set.CompoundElementToTuple
- Set.CompoundTupleToElement
- Set.CompoundTuplePassMode

Set.Card

The property Card provides the cardinality of the set. The cardinality of a set is defined as the total number of elements in the set. *Property*

Synopsis:

```
Int Card
```

Remarks:

The property Card is read-only. Its value changes whenever elements are added to or removed from the set.

Set.Dimension

The dimension of a set is defined as follows: *Property*

- the dimension of a simple set is 1,
- the dimension of a relation is the dimension of the Cartesian product of which the relation is a subset
- the dimension of a compound set is 1, and
- the dimension of an indexed set is the dimension of the index domain of the set plus 1.

Synopsis:

```
Int Dimension
```

Set.ElementPassMode

Property

The AIMMS COM interface allows you to reference set elements in three different modes:

- as element numbers (unique numbers, which remain unchanged during the AIMMS session)
- as ordinal numbers (the ordinal position of the element in the corresponding set), or
- as element names (the name of the elements as you see it in the model).

With the property ElementPassMode you can specify how you want to pass the elements of the corresponding set. Initially, this property inherits its value from Project.DefaultElementValuePassMode.

Synopsis:

```
Long ElementPassMode
```

Remarks:

The property ElementPassMode is integer valued and can have any of the following defined values:

- ELEMENT_BY_NUMBER = 0
- ELEMENT_BY_ORDINAL = 1
- ELEMENT_BY_NAME = 2

Example:

```
Dim Cities As Aimms.Set
Set Cities = MyProject.GetIdentifier( "Cities" )

Cities.TuplePassMode = ELEMENT_BY_NAME
Cities.AddElement "Amsterdam"
```

Set.Name

The property Name holds the name of the AIMMS set. *Property*

Synopsis:

```
String Name
```

Remarks:

The property Name is read-only.

See also:

The property Project.GetSet

Set.OrdinalsOffset

Property

When passing set elements, the AIMMS COM allows you to reference these elements by their ordinal position within the set. With the property `OrdinalsOffset` you can indicate whether the first element in the set is referenced as ordinal number 0 or ordinal number 1.

Synopsis:

```
Int OrdinalsOffset
```

Remarks:

This property has an effect on:

- the passing of elements, if `ElementPassMode` equals `ELEMENT_BY_ORDINAL`, and
- the conversion of elements between number, ordinal and string representations.

Example:

```
Dim Cities As Aimms.Set
Dim Name0, Name1 As String
Set Cities = MyProject.GetIdentifier( "Cities" )

Cities.OrdinalsOffset = 0
Name0 = Cities.OrdinalToName( 0 )

Cities.OrdinalsOffset = 1
Name1 = Cities.OrdinalToName( 1 )

' Name0 and Name1 are the same
MsgBox Name0 + "=" + Name1
```

See also:

The methods `Set.OrdinalToName`, `Set.OrdinalToElement`, `Set.NameToOrdinal`, `Set.ElementToOrdinal` and `Set.ElementPassMode`.

Set.ReadOnly

Property

The property ReadOnly indicates whether you are allowed to make modifications to the contents of the set. If in the AIMMS model the set is declared with a Definition, or if the set is not part of the predefined set AllUpdatableIdentifiers, then modifications are not allowed.

Synopsis:

```
Boolean ReadOnly
```

Remarks:

You cannot change the value of the property ReadOnly.

Set.AddElement

With the method AddElement you can add single elements to the set.

Synopsis:

```
AddElement(
          element_name,      ! (input) String
          [recursive]        ! (optional) Boolean
          )
```

Arguments:

element_name
: The name of the element that you want to add to the set.

recursive (optional)
: This boolean indicates whether the method should automatically add the new element to each of its super sets. If you set this to FALSE, then if you add an element to a subset, this element must already exist in the superset.

Remarks:

This function will fail when the element to be added is already in the set.

See also:

The methods Set.DeleteElement

Set.AssignElementArray

With this method you can fill the set with (new) elements. Optionally you can replace the entire existing contents of the set with the new elements, or merge the new elements with the existing elements.

Synopsis:

```
AssignElementArray(
          element_array,      ! (input) Variant array (String/Integer)
          [mode]              ! (optional) Long
          )
```

Arguments:

element_array
: An array containing the names of the elements that must be added to the set.

mode (optional)
: Possible values: REPLACE or MERGE. If mode is set to REPLACE (the default setting), then existing elements in the set that are not in the given array are removed from the set, and new elements are added. If mode is set to MERGE, then all the new elements in the array are added to the existing elements.

Example:

```
Dim Cities As Aimms.Set
Dim DutchCities(3) As String
Set Cities = MyProject.GetSet( "Cities" )
DutchCities(0) = "Amsterdam"
DutchCities(1) = "Rotterdam"
DutchCities(2) = "The Hague"
DutchCities(3) = "Utrecht"
Cities.AssignElementArray DutchCities
```

See also:

The methods Set.AddElement

Set.RetrieveElementArray

With this method you can retrieve an array containing all the elements in a set.

Synopsis:

```
RetrieveElementArray(
          element_array       ! (output) Variant array (String/Integer)
          )
```

Arguments:

element_array
: An array, that on output contains references to all the elements in the set. Whether the array is of type String or Integer depends on the property ElementPassMode.

Example:

```
Dim Cities As Aimms.Set
Dim CurrCities() As String
Set Cities = MyProject.GetSet( "Cities" )
ReDim CurrCities( Cities.Card-1 )
Cities.ElementPassMode = ELEMENT_BY_NAME
Cities.RetrieveElementArray CurrCities
```

See also:

The method Set.CreateElementArray

Set.CreateElementArray

This method creates and retrieves an array with the current elements of the set. The method is similar to RetrieveElementArray except that CreateElementArray does not require you to provide the value array. Instead, the array is allocated by the method itself, and returned as the return value of the method.

Synopsis:

```
VariantArray CreateElementArray
```

Arguments:

None

Return value:

A newly created array of Variants, containing the current elements of the set. The size of the created array exactly matches with the cardiality of set. Whether the first element of the array starts at position 0 or 1 depends on the value of the property Project.ArrayIndexOffset.

Example:

```
Dim Cities As Aimms.Set
Dim CurrCities() As Variant
Set Cities = MyProject.GetSet( "Cities" )
Cities.ElementPassMode = ELEMENT_BY_NAME
CurrCities = Cities.CreateElementArray
```

See also:

The methods Set.RetrieveElementArray

Set.DeleteElement

With the method DeleteElement you can delete an element from the set.

Synopsis:

```
DeleteElement(
            element_ref       ! (input) Variant (String/Integer)
            )
```

Arguments:

element_ref
: A reference to the element that you want to delete. The type of this argument depends on the current value of ElementPassMode and is either a string or integer.

Remarks:

Besides removing a single element using the method DeleteElement, you can implicitly remove elements using the method AssignElementArray in REPLACE mode.

See also:

The methods Set.AddElement, Set.AssignElementArray, Set.ElementPassMode.

Set.ElementToName

With the method ElementToName you can convert from an element number to the name of that set element.

Synopsis:

```
String ElementToName(
                    name         ! (input) String
                    )
```

Arguments:

number
The unique element number.

Return value:

ElementToName returns the name of the given element.

See also:

The methods Set.NameToElement, Set.NameToOrdinal, Set.ElementToOrdinal, Set.OrdinalToElement and Set.OrdinalToName.

Set.ElementToOrdinal

With the method ElementToOrdinal you can convert from the ordinal number of a set element to its element number. In other words, you can retrieve the element number of the n-th element. The element number is a unique number of the set element, which remains unchanged during the AIMMS session.

Synopsis:

```
Integer ElementToOrdinal(
                        number        ! (output) Integer
                        )
```

Arguments:

number
The element number of a set element

Return value:

ElementToOrdinal returns the ordinal number of the given element.

Remarks:

The conversion from element to ordinal number uses the current value of the property OrdinalsOffset.

See also:

The methods Set.OrdinalToElement, Set.OrdinalToName, Set.ElementToName, Set.NameToOrdinal, Set.NameToElement, and Set.OrdinalsOffset.

Set.OrdinalToName

With the method OrdinalToName you can convert from the ordinal number of a set element to its element name. In other words, you can retrieve the name of the n-th element.

Synopsis:

```
String OrdinalToName(
                    ordinal      ! (input) Integer
                    )
```

Arguments:

ordinal
The ordinal number of an element.

Return value:

OrdinalToName returns the name of the given element.

Remarks:

The conversion from ordinal number to name uses the current value of the property OrdinalsOffset.

See also:

The methods Set.NameToOrdinal, Set.NameToElement, Set.OrdinalToElement, Set.ElementToName, Set.ElementToOrdinal and Set.OrdinalsOffset.

Set.OrdinalToElement

With the method OrdinalToElement you can convert from the ordinal number of a set element to its element number. In other words, you can retrieve the element number of the n-th element. The element number is a unique number of the set element, which remains unchanged during the AIMMS session.

Synopsis:

```
Integer OrdinalToElement(
                        ordinal      ! (input) Integer
                        )
```

Arguments:

ordinal
The ordinal number of an element.

Return value:

OrdinalToElement returns the unique element number for the given ordinal number.

Remarks:

The conversion from ordinal to element number uses the current value of the property OrdinalsOffset.

See also:

The methods Set.ElementToOrdinal Set.ElementToName, Set.OrdinalToName, Set.NameToOrdinal, Set.NameToElement, and Set.OrdinalsOffset.

Set.NameToElement

With the method NameToElement you can convert from the name of a set element to its element number.

Synopsis:

```
Integer NameToElement(
                    name         ! (input) String
                    )
```

Arguments:

name
The name of the set element.

Return value:

NameToElement returns the unique element number of the given element name.

See also:

The methods Set.ElementToName, Set.ElementToOrdinal, Set.NameToOrdinal, Set.OrdinalToElement and Set.OrdinalToName.

Set.NameToOrdinal

With the method NameToOrdinal you can convert from the name of a set element to the ordinal number of the element.

Synopsis:

```
Integer NameToOrdinal(
                    name         ! (input) String
                    )
```

Arguments:

name
The name of the set element.

Return value:

NameToOrdinal returns the ordinal number of the given element.

Remarks:

The conversion from name to ordinal number uses the current value of the property OrdinalsOffset.

See also:

The methods Set.OrdinalToName, Set.OrdinalToElement, Set.NameToElement, Set.ElementToName, Set.ElementToOrdinal and Set.OrdinalsOffset.

Set.RenameElement

With the method RenameElement you can rename an element in the set.

Synopsis:

```
RenameElement(
            element_ref,      ! (input) Variant (String/Integer)
            new_name,         ! (input) String
            )
```

Arguments:

element_ref
: A reference to the element that you want to rename. The type of this argument depends on the current value of ElementPassMode and is either a string or integer.

new_name
: The new name for the element.

Set.CompoundAddElementTuple

The method CompoundAddElementTuple adds a new element to a compound set. The new element is given as a tuple of elements.

Synopsis:

```
CompoundAddElementTuple(
          tuple_array,      ! (input) Variant array (String/Integer)
          number,           ! (output) Integer
          [recursive]       ! (optional) Boolean
          )
```

Arguments:

tuple_array
: An array with a size equal to CompoundDimension. This array contains the tuple of individual elements that you want to add to the compound set. The type of the array should match with CompoundTuplePassMode and is either integer or string.

number
: On return this arguments holds the element number of the newly added compound element.

recursive (optional)
: If this argument is set to TRUE, the new compound element is automatically added to all supersets of the compound set. If set to FALSE, the element is only added to the set itself. In the latter case the set must either be a root compound set or the element must already exist in the superset.

Remarks:

The method CompoundAddElementTuple only applies to compound sets in the AIMMS model. The method returns with an error if you call this function for other type of sets.

Example:

```
Dim routes As Aimms.Set
Dim newtuple(1) As Variant
Dim newelement As Variant
set routes = MyProject.GetSet( "routes" )
newtuple(0) = "Amsterdam"
newtuple(1) = "Rotterdam"
routes.TuplePassMode = ELEMENT_BY_NAME
routes.ElementPassMode = ELEMENT_BY_NAME
routes.CompoundAddTupleElement newtuple, newelement
MsgBox "Added: " + newelement
' result: "Added: ('Amsterdam','Rotterdam')"
```

See also:

The methods Set.CompoundTupleToElement, Set.CompoundDimension

Set.CompoundDimension

Property

The property Dimension of a compound set is per definition equal to 1. If you want to retrieve the dimension of a compound set as a relation, then you must use the special property CompoundDimension.

Synopsis:

```
Int CompoundDimension
```

Remarks:

The property CompoundDimension is read-only and only applies to compound sets in the AIMMS model. For other sets, this property will always return 0.

Set.CompoundElementToTuple

The method CompoundElementToTuple converts a compound element to the corresponding tuple of domain set elements.

Synopsis:

```
CompoundElementToTuple(
              elem,           ! (input) Variant (String/Integer)
              tuple_array     ! (output) Variant array (String/Integer)
              )
```

Arguments:

elem
: A reference to an element in the compound set. The type of this argument depends on the property ElementPassMode and is either a string or integer.

tuple_array
: An array with a size equal to CompoundDimension. On output this array contains the tuple of individual domain set elements. The type of the array matches with CompoundTuplePassMode and is either integer or string.

Remarks:

The method CompoundElementToTuple only applies to compound sets in the AIMMS model. You will get an error if you call this function for other type of sets.

See also:

The methods Set.CompoundTupleToElement

Set.CompoundTupleToElement

The method CompoundTupleToElement converts a tuple of elements to the corresponding element in a compound set.

Synopsis:

```
CompoundTupleToElement(
            tuple_array,      ! (input) Variant array (String/Integer)
            elem,             ! (output) Variant (String/Integer)
            )
```

Arguments:

tuple_array
: An array with a size equal to CompoundDimension. This array contains the tuple of individual elements that you want to convert. The type of the array should match with CompoundTuplePassMode and is either integer or string.

elem
: On successfull return this arguments holds a reference to an existing element in the compound set. The type of this argument depends on the property ElementPassMode and is either a string or integer.

Remarks:

The method CompoundTupleToElement only applies to compound sets in the AIMMS model. The method returns with an error if you call this function for other type of sets.

See also:

The methods Set.CompoundElementToTuple

Set.CompoundTuplePassMode

Property

The AIMMS COM interface allows you to reference set elements in three different modes:

- as element numbers (unique numbers, which remain unchanged during the AIMMS session)
- as ordinal numbers (the ordinal position of the element in the corresponding set), or
- as element names (the name of the elements as you see it in the model).

With the property CompoundTuplePassMode you can specify how you want to pass the domain set elements of a compound tuple. Initially, this property inherits its value from Project.DefaultTuplePassMode.

Synopsis:

```
Long CompoundTuplePassMode
```

Remarks:

The property CompoundTuplePassMode is integer valued and can have any of the following defined values:

- ELEMENT_BY_NUMBER = 0
- ELEMENT_BY_ORDINAL = 1
- ELEMENT_BY_NAME = 2

The property CompoundTuplePassMode only applies to compound sets in the AIMMS model. For other sets, changing this property will not have any effect.

Chapter 6

The Aimms.Procedure Object

The Procedure object allows you run procedures within the AIMMS project. The Procedure object offers a small number of properties and methods to retrieve information about the arguments and to actually run the procedure. If your procedure does not require arguments and if you do not have to run the procedure multiple times, then you do not have to use the Procedure object. Instead, you can use the Run method of the Project object directly.

Object creation

Because a procedure only exists within the context of an AIMMS project, you cannot create a Procedure object directly using a standard COM object creator function. Instead, you must create a Procedure object via a specific call to the Aimms.Project object, namely Project.GetProcedure.

Example

The following example shows a small Visual Basic program, that opens an AIMMS project, creates a Procedure object, and runs that procedure.

```
Dim MyProject As Aimms.Project
Dim MainExec As Aimms.Procedure

'create project object, and open the project file
set MyProject = New Aimms.Project
MyProject.ProjectOpen "C:\Examples\Transport.prj"

'create procedure object for the AIMMS procedure 'MainExecution'
set MainExec = MyProject.GetProcedure( "MainExecution" )

'run it
MainExec.Run
```

Methods and Properties

The Procedure object exposes the following methods and properties:

- Procedure.Run
- Procedure.NumberOfArguments
- Procedure.ArgumentInOutType
- Procedure.ArgumentStorageType

Procedure.Run

With the method Run you can execute a procedure within the AIMMS model. If the procedure expects arguments, these arguments can be filled in too.

Synopsis:

```
Long Run(
        [var_args]              ! (optional) variable number of arguments
        )
```

Arguments:

var_args (optional)
A variable number of arguments, corresponding with the arguments of the procedure in the AIMMS model.

Return value:

The method Run returns the value that is returned from the AIMMS procedure. The return value of a procedure in AIMMS is optional, and is usually omitted. In that case, the default return value 0 is used.

Example:

The following two pieces of code give the same result:

```
'Running a procedure using the Procedure object
Dim proc As Aimms.Procedure
Set proc = MyProject.GetProcedure( "MainExecution" )
proc.Run
```

See also:

The methods Project.GetProcedure Project.Run

Procedure.NumberOfArguments

This property gives the number of arguments the procedure expects according to its declaration in AIMMS. *Property*

Synopsis:

```
Int NumberOfArguments
```

Remarks:

The property NumberOfArguments is read-only.

Procedure.ArgumentInOutType

Property

This property indicates whether the n-th argument is an input argument, an output argument or both input and output.

Synopsis:

```
Integer ArgumentInOutType(
                        n       ! (input) Integer
                        )
```

Arguments:

n

The sequence number of the argument. The first argument must be referred to using $n = 1$.

Remarks:

The property ArgumentInOutType is integer valued and can have any of the following defined values:

- ARG_INPUT = 1
- ARG_OUTPUT = 2
- ARG_INOUT = 3

If an argument is of type ARG_OUTPUT or ARG_INOUT, you must make sure that you pass your argument *by reference*.

Procedure.ArgumentStorageType

Property

This property indicates the data type of the n-th argument of the AIMMS procedure.

Synopsis:

```
Integer ArgumentStorageType(
                          n       ! (input) Integer
                          )
```

Arguments:

n

The sequence number of the argument. The first argument must be referred to using $n = 1$.

Remarks:

The property ArgumentStorageType is integer valued and can have any of the following defined values:

- STORAGE_HANDLE = 0
- STORAGE_DOUBLE = 1
- STORAGE_INTEGER = 2
- STORAGE_BINARY = 3
- STORAGE_STRING = 4

If the storage type is STORAGE_HANDLE, you must pass a reference to an Aimms.Identifier object.

Part III

Appendices

Index