THE POWER TO KNOW®

SAS® Financial Management 5.5

Customization Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2015. *SAS® Financial Management 5.5: Customization Guide*. Cary, NC: SAS Institute Inc.

SAS® Financial Management 5.5: Customization Guide

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

April 2015

Contents

1 About This Book

Overview

This chapter describes the following topics:

- for whom this book is intended
- what is in this book
- terms and documentation conventions used in this book
- related information that might be helpful

Audience

This book is intended for SAS Financial Management administrators or power users. To use the SAS Financial Management Java API, you must be familiar with Java programming. To use the SAS Financial Management Add-in API for Microsoft Excel, you must understand Microsoft Excel and Visual Basic for Applications (VBA).

What's in This Book

This book includes the following topics:

- customizing SAS Financial Management content, including dashboard content, that is displayed in the SAS Information Delivery Portal
- using the SAS Financial Management APIs, including:
  - writing SAS code that uses the SAS Financial Management Java application programming interface (API)
  - writing VBA code in Microsoft Excel that interacts with SAS Financial Management objects
  - adding custom cell actions to Microsoft Excel
  - customizing a workflow
  - writing a custom analytics stored process for use with the SAS Financial Management Add-in for Microsoft Excel
- configuring Secure Socket Layer (SSL)

Note: For information about customizing themes, see the *SAS Intelligence Platform: Middle-Tier Administration Guide*.

What’s New in This Book

The *SAS Financial Management: Customization Guide* includes the following new or updated information:

- SAS Financial Management Java API
  - In the FormSet class, the following methods have been added to support the Table Reallocation feature, which enables a user to reallocate values at the table level:
    - void reallocateAllForms
    - void reallocateSelectedForms
    - void reallocateSeletedTables
    - void reallocateSelectedFormsandTables

    The errors encountered by the four methods above are recorded in a SASSDM (PostgreSQL) database table that is named sas_table_reallc_report.
  - In the FormSet class, callStringMethod (‘getErrorDescriptions’) has been added to enable the user to retrieve the descriptions of any errors recorded in the sas_table_reallc_report table. A user can use the descriptions to create custom reports on the results.
- SAS Financial Management API for Microsoft Excel

- The FMTable class has a new function, ReallocateTable. You can use the ReallocateTable function to reallocate values at the table level. The ReallocateTable function stops at the first error.

Documentation Conventions

Directory Paths

Directory Paths Used by Previous Installations

This book uses the following documentation conventions to identify directory paths used by SAS Financial Management 5.3 and earlier installations:

Term	Refers to	Example Path
!sasroot	Path to the SAS root directory in a SAS 9.2 installation	Windows: `C:\Program Files\SAS\SASFoundation\9.2` UNIX: `/usr/local/SAS/SASFoundation/9.2`
!sasroot	Path to the SAS root directory in a SAS 9.3 installation	Windows: `C:\Program Files\SASHome\SASFoundation\9.3` UNIX: `/usr/local/SASHome/SASFoundation/9.3`
SAS-config-dir	Path to the SAS configuration directory	Windows: `C:\SAS\Config\Lev1` UNIX: `/usr/local/SAS/config/Lev1`
MySQL-install-dir	Path to the MySQL installation directory in a SAS installation prior to SAS 9.4	Windows: `C:\MySQL\bin` UNIX: `/usr/local/mysql` **Note:** As of SAS Financial Management 5.4, MySQL is no longer supported. For more information, see *SAS Financial Management: System Administration Guide.*

Directory Paths Used by a SAS Financial Management 5.4 and a SAS Financial Management 5.5 Installation

This book uses the following documentation conventions to identify directory paths that are used by SAS Financial Management 5.4 and SAS Financial Management 5.5:

Term	Refers to	Example Path
!sasroot	Path to the SAS root directory (SAS Foundation)	Windows: `C:\Program Files\SASHome\SASFoundation\9.4` UNIX: ha`/usr/local/install/SASHome/SASFoundation/9.4`
!sasinst	Path to the SAS installation directory	Windows: `C:\Program Files\SASHome` UNIX: `/usr/local/install/SASHome`
SAS-config-dir	Path to the SAS configuration directory	Windows: `C:\SAS\Config\Lev1` UNIX: `/usr/local/SAS/config/Lev1`

Terms

This book uses the following terms:

Term	Description
Data Mart	The SAS Financial Management Data Mart.
data tier	The machine on which you install the data-tier software for SAS Financial Management.
middle tier	The machine on which you installed the web application server and on which your web applications run.
metadata tier	The machine on which you installed the SAS Metadata Server. Usually, this is the same machine as the data tier.
multi-tier installation	An installation that is done on more than one machine (for example, with a data tier and a middle tier).
single-tier installation	An installation that is done on one machine. In that case, the single machine functions as both the data tier and the middle tier. Follow instructions for both the data tier and the middle tier.
staging area	The SAS Financial Management staging area.

Note:

- The name of the configuration directory and the SAS release might be different at your site.
- If your configuration is the result of a migration from the previous release of SAS Financial Management, the SASApp directory might be called SASMain

instead. For example: `C:\SAS\Config\Lev1\SASMain` instead of `C:\SAS\Config\Lev1\SASApp`. Please make the appropriate substitutions as you read this book.

- File system pathnames are typically shown with Windows separators (\); for UNIX, substitute a forward slash (/).
- Some code examples contain line breaks so that the code fits on the line. If you copy the code, remove the line breaks.

Related Documentation

SAS Financial Management

For information about installing, administrating, or migrating SAS Financial Management, see the documentation located at

http://support.sas.com/documentation/onlinedoc/fm/

Note: This site is password-restricted. You can find the user name and password in the pre-installation checklist, the Instructions.html, or by contacting SAS Technical Support at http://support.sas.com/techsup/contact/

SAS Intelligence Platform

For information about administering the SAS Intelligence Platform, see the documentation located at

http://support.sas.com/documentation/onlinedoc/intellplatform/index.html

SAS Information Delivery Portal

For information about the SAS Information Delivery Portal, see the documentation located at

http://support.sas.com/documentation/onlinedoc/portal/index.html

SAS Notes

SAS Technical Support develops SAS Notes to inform customers of issues that they need to be aware of when using SAS software. SAS Notes contain additional information about a SAS product and support fixes.

To view SAS Notes for SAS Financial Management, see the product page at

http://support.sas.com/software/products/fm/index.html

On the left side of the product page, select **Samples & SAS Notes** from the menu.

2

Content Administration

Overview

This chapter describes the following topics:

- content administration
- SAS Financial Management reports
- the SAS Information Delivery Portal
- SAS Financial Management stored processes

About Content Administration

This chapter describes ways to manage the content (other than forms) that is displayed when a user logs on to SAS Financial Management or the SAS Information Delivery Portal in a web browser:

- SAS Financial Management reports
- content that is available in the SAS Information Delivery Portal:
  - alerts

- ☐ stored processes
- ☐ SAS reports
- ☐ information maps
- ☐ a link to SAS Financial Management

For information about the Form Status dashboard that is available in a SAS BI Dashboard portlet or in the SAS BI Dashboard viewer, see Chapter 6, “Customizing the Form Status Dashboard,” on page 95.

SAS Financial Management Reports

When a user logs on to SAS Financial Management in a web browser, they can access reports from the SAS Visual Analytics home page. For detailed information about accessing a SAS Financial Management report from the SAS Visual Analytics home page, see the *SAS Financial Management: User’s Guide*.

SAS Financial Management supports the following content types:

- **Microsoft Excel spreadsheet**—Static SAS Financial Management report. You cannot modify a static report and you cannot refresh its data.
- **Financial management report for Microsoft Excel**—Dynamic SAS Financial Management report. A dynamic report is a fully functional Excel binary file.
- **PDF file**—PDF document that you open in Adobe Acrobat.

Reports automatically open in the appropriate application (Microsoft Excel or Adobe Acrobat).

For more information about working with reports, see the *SAS Financial Management: User’s Guide*. You can access the *SAS Financial Management: User’s Guide* online or from the **Help** menu of SAS Financial Management on the web.

The SAS Information Delivery Portal

Overview

From the SAS Information Delivery Portal, users can access the following SAS Financial Management content:

- **Stored processes**—Standard SAS Financial Management reports located in the `/Products/SAS Financial Management/5.5 Standard Reports` folder.

  In the portal, stored processes can be added to a Collection portlet.

  Alternatively, a single stored process can be added to a SAS Stored Process portlet. In that case, the stored process is executed when a user opens the portal page or refreshes the stored process.

  For more information about stored processes, see “SAS Financial Management Stored Processes” on page 10.

- **SAS reports**—SAS reports that are generated by the Publish wizard in the SAS Financial Management Add-In for Microsoft Excel. These static reports can be opened in SAS Web Report Studio. They display current data but cannot be modified.

  In the portal, SAS reports can be added to a Collection portlet. Clicking a report opens it in SAS Web Report Studio.

- **Information maps**—Information maps that are generated by the Information Map wizard in the SAS Financial Management Add-In for Microsoft Excel. They can be opened in SAS Web Report Studio as dynamic reports.

  In the portal, information maps can be added to a Collection portlet. Clicking an information map opens it in SAS Web Report Studio.

  Note: Information maps that are created in the SAS Financial Management Add-In for Microsoft Excel cannot be opened in SAS Information Map Studio.

- **Alerts**—Alerts from workflow events and forecasting that can be viewed in an Alerts portlet. See "Alerts Portlets" on page 9.

- **Link to SAS Financial Management**—For information about the link to SAS Financial Management, see "Link to SAS Financial Management" on page 10.

- **Form Status dashboard**—For information about the Form Status dashboard, see "The Form Status Dashboard" on page 95.

Note: Users must have the necessary capabilities and security permissions to perform these tasks and view the data. For more information about capabilities, see "Assigning Groups and Roles" in the *SAS Financial Management: System Administrator's Guide*.

Creating Page Templates for the Portal

If you are a portal content administrator, you can design page templates for the portal that contain Collection portlets, Alerts portlets, and BI Dashboard portlets. You can share these templates with specific groups, and users in those groups can then add those pages to their own portal pages.

For information about becoming a portal content administrator and designing page templates, see the *SAS Intelligence Platform: Web Application Administration Guide*, available at http://support.sas.com/documentation/onlinedoc/intellplatform/index.html.

The *SAS Financial Management: User's Guide* also explains how users can add portlets to their own portal pages.

Alerts Portlets

An *alert* is a notification of an event that the user might need to respond to.

Workflow alerts are notifications of tasks that the user has to perform, such as approving a budget form. On the Preferences window of the portal, users can specify how they want to receive alert notifications. One method is via an email message. Another method is an Alerts portlet.

To add an Alerts portlet to a page, complete the following steps:

1 From the **Customize** menu, select **Edit Page** ▸ **Edit Page Content**.

2 On the Edit Page Content page, select **Add Portlets** and add a Shared Alerts portlet.

For more information about adding portlets to a page, see the online Help for the portal.

Link to SAS Financial Management

To add a link to SAS Financial Management from the portal, complete the following steps:

1 In a Collection portlet, click the **Edit Content** button.

2 In the Edit Portlet Content window, click **Add Items**.

3 In the Add Items to Portlet window, click the **Search** tab.

4 In the **Content Types** list, select **Application**.

5 Enter `Forms` as a keyword and click **Search**.

6 In the search results, select the SAS Financial Management application and add it to the portal page.

SAS Financial Management Stored Processes

Overview

The `/Products/SAS Financial Management/5.5 Standard Reports` folder contains a number of stored processes:

- two stored processes that are available for administrators (Import Users and Groups and ETL Job Status)
- several standard reports, such as Audit and Facts, that can be made available to end users
- a stored process (Form status) that is used only to generate output for the Form Process Status dashboard. It cannot be executed from the portal directly.

Note: If you open the stored process properties in SAS Management Console, you see that in many cases the **Type** is **Stored process (9.2)**. This indicates that they are compatible with SAS 9.2. If you want to modify one of these stored processes and use SAS 9.4 stored process features, you must first upgrade the stored process. For more information, see “Making Stored Processes Compatible with 9.2 and Upgrading Stored Processes” in the *SAS Stored Processes: Developer's Guide*.

Administrative Stored Processes

The `/Products/SAS Financial Management/5.5 Standard Reports` folder contains two stored processes that are typically made available only to administrators.

- **Import Users and Groups**

  Loads information about user and group membership to the SAS Financial Management Data Mart. It is equivalent to running the solnsvc_1300_load_users, solnsvc_1400_load_groups, and solnsvc_1500_load_user_x_group jobs in SAS Data Integration Studio.

  For more information about user and group membership, see “Assigning Groups and Roles” in the *SAS Financial Management: System Administrator’s Guide*.

- **ETL Job Status**

  Displays the status of jobs that are executed in SAS Data Integration Studio.

Note: To run these stored processes, administrators need ReadMetadata access to the stored process and membership in the SASSDM Users group.

Standard Reports

The `/Products/SAS Financial Management/5.5 Standard Reports` folder also contains a set of standard SAS Financial Management reports that you might want to make available to other users. To execute these stored processes, users need ReadMetadata permission for the folder or for the individual stored processes.

Note: Without ReadMetadata permission on a folder, users cannot navigate to items beneath that folder. For more information about managing folders, see “Best Practices for Managing SAS Folders” in the *SAS Intelligence Platform: System Administration Guide*, available at http://support.sas.com/documentation/onlinedoc/intellplatform/index.html.

Table 2.1 SAS Financial Management Standard Reports

Report	Description
Audit	Lists actions that have been completed. You can limit an Audit report to an object type, a user, or a range of dates. By default this stored process returns a maximum of 10,000 rows. For information about modifying the stored process, see the description of the AuditHistory class in this document.
Data Entry	Lists data records that were entered through forms using a specified model. You can limit a Data Entry report to a time period, an organization, an analysis member, or a form set.
Eliminations	Lists, for a specified model, data records for all accounts that have the Intercompany attribute but that are not specified in any intercompany balancing rule or net intercompany balancing rule. There should not be any such accounts, so this report should not list any data records. If the report does list data records, then you need to edit the rules that look for imbalances in intercompany accounts, or add more such rules. You can limit an Eliminations report to a time period, an organization, or an analysis member.

Report	Description
ETL Facts	Lists data records that have been loaded from SAS Data Integration Studio to a specified time period and analysis member within a specified cycle. You can further limit an ETL Facts report to a specified organization.
Facts	Lists data records that are associated with a specified model. You can limit a Facts report to a time period or an analysis member, and in several other ways.
ICAccounts	Lists, for a specified model, accounts that have the Intercompany attribute but that are not specified in any intercompany balancing rule or net intercompany balancing rule. You can limit an ICAccounts report to accounts that belong to a particular account type or accounts that have a particular balance type.
Intercompany	Lists, for a specified model, data records in which the account member has the Intercompany attribute and the trader member is either EXT or identical to the organization member. No records should satisfy this condition. You can limit an Intercompany report to a time period, an organization, or an analysis member.
Manual Adjustments	Lists all the currently posted manual adjustments for a specified model. You can limit a Manual Adjustments report to a time period, an organization, or an analysis member. You can also limit the report to a range of adjustment amounts.
Non Intercompany	Lists the data records for a specified model in which the account member does not have the Intercompany attribute and the trader member is not EXT or identical to the organization member. No records should satisfy this condition. You can limit a Non Intercompany report to a time period, an organization, or an analysis member.
Non Leaf	Lists all the non-leaf data records (also known as virtual-child data records) for a specified model. You can limit a Non Leaf report to a time period, an organization, or an analysis member.
Ownership Adjustments	Lists all the adjustments that are generated by the ownership rule for a specified model. You can limit an Ownership Adjustments report to a time period, an analysis member, a holding organization, or a held organization.
Ownership Methods	Lists all the ownership relations that are specified in the ownership rule for a specified model, showing the consolidation method for each relation. You can limit an Ownership Methods report to a holding organization, a held organization, or a consolidation method.
Ownership Transactions	Lists all the asset purchases and asset sales that are specified in the ownership rule for a specified model. You can limit an Ownership Transactions report to a holding organization, a held organization, or a transaction type.
Rule	Lists all the adjustments that are generated by a specified adjustment rule within a specified model. You can limit a Rule report to a time period or an analysis member.
Rules Facts	Lists all the adjustments that are generated by all the adjustment rules that are part of a specified model. You can limit a Rules Facts report to a time period, an organization, or an analysis member. You can also limit the report to a range of adjustment amounts.
Trial Balance	Lists data records that are associated with a specified model and that were loaded from the SAS Financial Management staging area. You can limit a Trial Balance report to a time period, an organization, or an analysis member.

The Form status stored process, also in this folder, is used only to generate output for the Form Process Status dashboard. You cannot directly execute the Form status stored process how you can other stored processes. For more information about the Form Status stored process, see Chapter 6, “Customizing the Form Status Dashboard,” on page 95.

Saving a New or Customized Stored Process to the Correct Location for Migration

Note: You can complete the following configuration when adding a new stored process or by editing an existing stored process.

To ensure that stored processes that have been customized or created at a customer site migrate properly, you must save them to `C:\SAS\Config\Lev1\SASApp\SASEnvironment\FinancialManagement\SASCode\Jobs`.

To configure the properties of a customized stored process to ensure that the stored process migrates properly, complete the following steps:

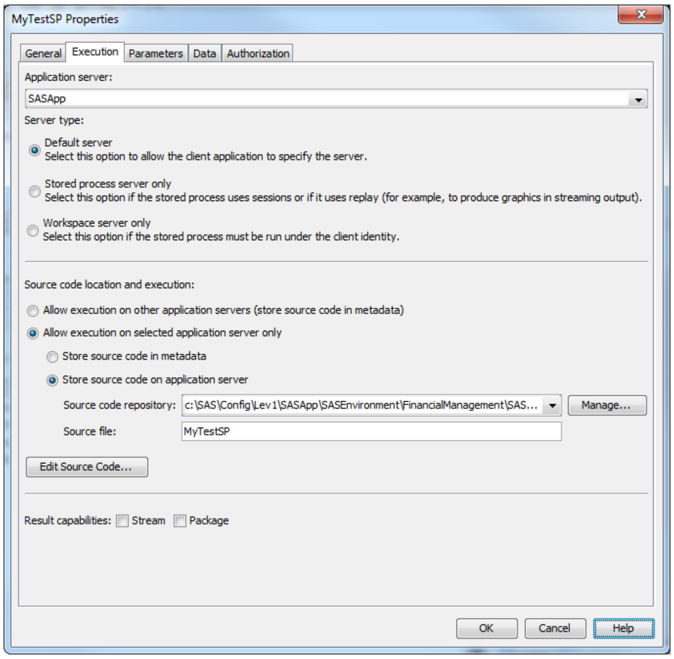
1 In SAS Management Console, select the **Folders** tab.

2 In the Folders tree, select **Products** ▸ **SAS Financial Management** and the name of the folder that contains the new or customized stored process.

3 Double-click on the name of a customized stored process. The Properties window for the stored process is displayed.

4 Select the **Execution** tab to display the execution properties for the stored process.

Display 2.1 Stored Process Properties Window — Execution Properties



Application server: SASApp

Server type:

[✓ Default server] Select this option to allow the client application to specify the server.

Stored process server only Select this option if the stored process uses sessions or if it uses replay (for example, to produce graphics in streaming output).

Workspace server only Select this option if the stored process must be run under the client identity.

Source code location and execution:

Allow execution on other application servers (store source code in metadata)

[✓ Allow execution on selected application server only]

Store source code in metadata

[✓ Store source code on application server]

Source code repository: c:\SAS\Config\Lev1\SASApp\SASEnvironment\FinancialManagement\SAS...

Source file: MyTestSP

Result capabilities: Stream Package

5 Click the **Manage** button located to the right of the **Source code repository** field. The Manage Source Code Repositories dialog box is displayed.

6 Click **Add**. The Add Source Code Repository dialog box is displayed.

7 In the **Path** field, enter the following path:

```
C:\SAS\Config\Lev1\SASApp\SASEnvironment\FinancialManagement
\SASCode\Jobs
```

CAUTION! This step must be completed in order to migrate new or customized content. New or customized data that is not located in this directory does not migrate properly.

8 Click **OK** to save your changes and close the Add Source Code Repository dialog box.

9 On the **Manage Source Code Repositories** dialog box, select the path that you added and click **OK**. The new path is displayed in the **Source code repository** field on the Properties window.

Note: This step only configures the server to look for the new or customized stored processes in this directory. Ensure that you have saved the new or customized stored process to the same directory.

10 Click **OK** to save your changes and close the Properties window.

3

The SAS Financial Management Java API

Overview

This chapter describes the following topics:

- using the SAS Financial Management Java API
- summary of the SAS Financial Management API classes
- model macros
- executing queries with the %FMQUERY macro

Using the SAS Financial Management Java API

About the SAS Financial Management Java API

The SAS Financial Management application programming interface (API) includes a set of Java classes and a set of SAS macros that are available for accessing SAS Financial Management data. You can use the SAS Financial Management Java API for multiple tasks, including the following:

- to execute a custom query against SAS Financial Management data
- to get a list of models or information about the properties, members, hierarchies, forms, or form sets that are associated with a specified model
- to reset or publish a form set
- to post adjustments for a model
- to reallocate the values of all forms in a form set or just for specific forms
- to reallocate the values of all tables in a form or just specific tables.

Note: The SAS Financial Management Java API does not support composite models, with one exception, the %FMQUERY macro.

You can use the SAS Financial Management Java API in a stored process or in code that is called by a SAS Data Integration Studio job. It can also be used in an interactive SAS session.

For information about creating and registering a stored process, see the *SAS Stored Processes: Developer's Guide*. For information about security for stored processes, see the *SAS Intelligence Platform: Security Administration Guide*.

Declaring the Javaobj

The SAS Financial Management Java API uses the *Javaobj interface*. The Javaobj interface is a mechanism that is similar to the Java Native Interface (JNI) for instantiating Java classes and accessing their methods and fields.

The DATA step that includes a Javaobj declaration must include the following options:

```
/picklist='finance/finance.txt' saveclassl='finbatch' getclassl='finbatch';
```

Where:

- picklist option—enables the Javaobj to access the necessary JAR files
- saveclass1 option—enables the Javaobj to access the necessary JAR files

Note: Do not call the logout method when you delete the object.

To declare a Javaobj object, use the following syntax:

```
dcl javaobj object-name (classname, constructor-arguments);
```

Where the parameters are as follows:

object-name
The handle to the Java object that is returned. You use this handle to access the object's methods.

classname
A string that contains the fully qualified name of the Java class that you are instantiating, such as “com/sas/solutions/finance/api/Form”.

constructor-arguments
Any arguments that are required by the constructor.

Authenticating the User

Authentication Using the METADATA_PASSID Function

In order to access SAS Financial Management data, the user must be authenticated on the middle tier. The recommended approach is to call the object's setEnvironment method and then call the METADATA_PASSID function in the DATA step.

For example:

```
data _null_ /picklist='finance/finance.txt' saveclassl="finbatch"
    getclassl="finbatch";
dcl javaobj j("com/sas/solutions/finance/api/AuditHistory");
j.ExceptionDescribe(1);
j.callVoidMethod("setEnvironment", "default");
call METADATA_PASSID("j", "");
```

This function creates a one-time user-password combination and authenticates the user on the middle tier.

In a stored process, the METADATA_PASSID function has access to the user ID and password. In an interactive SAS session, the user is asked for the user ID and password to be used for authentication on the middle tier. If the authentication fails, check the stored process log or the SAS log.

For an interactive SAS session, you also need the configuration and autoexec options that are specific to SAS Financial Management. One way to invoke the configuration and autoexec options is by starting an interactive SAS session from the ***SAS-config-dir*\Lev1\SASApp** folder.

Specifying the SAS Financial Management Environment

The environment argument is the name of an environment that is defined in the sas-environment XML file (for example, “default”, “dev”, or “prod”). The value of the environment variable should be the same value that a user would specify when logging on to the middle tier from Microsoft Excel.

If it was not added at installation time, add the JREOPTIONS option to the sasV9_usermods.cfg file located in each SAS Application Server context directory that you use. The env.factory.location argument should point to a network-accessible copy of the sas-environment XML file.

Calling the Methods of an Object

With a handle to the Java object, you can call its methods.

The following code calls the Form object's getState method, which returns a String value:

```
j.callStringMethod("getState", state);
```

In this example

- "j" represents the handle to the Form object
- The call statement matches the method's return type (for example, CALLSTRINGMETHOD, CALLDOUBLEMETHOD, CALLINTMETHOD, CALLBOOLEANMETHOD, or CALLVOIDMETHOD)
- The first parameter is always the method name, and the last parameter always contains the return value (if any). The remaining parameters are the parameters that the Java method requires.

  In the example above, the getState method has no parameters.

Note: The value returned by a Boolean method is `1` (**`true`**) or `0` (**`false`**).

Deleting the Javaobj

To avoid memory leaks, all instantiations of a Javaobj should be terminated by a call to the DELETE method, as in the following example:

```
dcl javaobj j("com/sas/solutions/finance/api/AuditHistory");
j.ExceptionDescribe(1);
...
j.delete();
```

Note: Do not call the object's logout method.

Retrieving Error Messages

Many methods return a Boolean value that indicates whether the action was successful. Because SAS does not have a true Boolean type, the return code is either `0` (failure) or `1` (success).

When the return code is `0`, the getErrorMessage method can be used to retrieve the pertinent error message, as in the following example:

```
if rc le 0 then do;
    j.callStringMethod("getErrorMessage", msg);
end;
```

Configuring a Log File

In addition to calling getErrorMessage, you can generate a more detailed log by creating a log4j.properties file.

For information about configuring a log file, see "Configure a Log File for the SAS Financial Management Reports" in the *SAS Financial Management: System Administrator's Guide*.

Handling Exceptions

You can use the EXCEPTIONCHECK method to determine whether an exception has been thrown. The EXCEPTIONCLEAR method clears any existing exceptions as in the following example:

```
/* clear any existing stored exception */
j.ExceptionClear();
j.callvoidmethod("getModelHierarchies", "Default_Model", "FMSData",
   "TstHierarchies");
/* check to see if an exception has been thrown */
rc = j.ExceptionCheck(exception);
if (exception) then
    put 'Exception occurred,Please check the log for more information';
```

Note: You cannot use the EXCEPTIONCHECK method to detect exceptions that are thrown when constructing an object.

A Simple Example

The following example of a stored process sets the default read and write members for the time hierarchy for a model:

```
*ProcessBody;
%stpbegin;

data _null_/picklist='finance/finance.txt' saveclassl="finbatch"
    getclassl="finbatch";
        length msg $ 1000;
        length ex 8;

        /* Instantiate the Javaobj */
        dcl javaobj oModel;
        oModel = _new_ javaobj("com/sas/solutions/finance/api/Model");
        oModel.ExceptionDescribe(1);

        oModel.callVoidMethod("setEnvironment", "default");

        /* Obtain a security context from the middle tier */
        call METADATA_PASSID("oModel", "");

        /* Set the model code. */
        oModel.callVoidMethod("setCode", "MyModel");

        /* Update the time dimension default read and write members */
        oModel.callVoidMethod("updateTimeDefaultMembers",
           "Q1 2012", "Jan 2012");

        /* Check for errors */
        rc = oModel.ExceptionCheck(ex);
        if (ex) then do;
            oModel.callStringMethod("getErrorMessage", msg);
            put msg;
            oModel.delete();
            abort;
        end;

        oModel.delete();
    run;

%stpend;
```

Note: The SAS Financial Management standard reports call the %RPTINIT macro, which extends and replaces the %STPBEGIN macro. The %RPTINIT macro has the following syntax:

```
%RPTINIT ( [STYLE=] [, DEVICE=] )
```

- STYLE specifies the value to use in the ODS STYLE option. It defaults to sasweb.
- DEVICE specifies the option for generating graphical output. It defaults to ACTXIMG for Windows and GIF for UNIX.

For more information about these options, see the *SAS Stored Processes: Developer's Guide*.

Summary of SAS Financial Management Java API Classes

The following table lists and briefly describes each class that is part of the SAS Financial Management Java API (com.sas.solutions.finance.api) package.

Table 3.1 Summary of Classes.

Class	Description
AdminQuery	Contains methods for running queries on the Base Facts data of SAS Financial Management.
AuditHistory	Contains methods for running queries on the AuditHistory data of SAS Financial Management.
BaseApi	Serves as the base class for the SAS Financial Management Java API. This class is extended by the BaseQuery, Form, Metadata, and Model classes.
BaseQuery	Contains methods for running queries. This class is extended by the AuditHistory, AdminQuery, and CycleQuery classes.
CellComments	Contains a method for extracting cell comments.
CycleQuery	Contains methods for extracting facts from a cycle.
Form	Contains methods for running queries on the properties of a planning form from SAS Financial Management.

Class	Description
FormSet	Contains methods for managing form sets, including the following: ■ resetting a form set ■ writing a batch script to publish a form set ■ reallocating the values of all forms in a form set or for specific forms in a form set ■ reallocating the values of all tables in a form or for specific tables in a form
Metadata	Contains methods for retrieving metadata about SAS Financial Management.
Model	Contains methods for completing the following: ■ retrieving information about SAS Financial Management models ■ running queries against a model ■ generating formula facts ■ posting adjustments ■ updating the model's default read and write members for the time dimension Many of the methods have corresponding macros, which should be used instead of calling the methods directly.

The AdminQuery Class

Overview

The com.sas.solutions.finance.api.AdminQuery class contains methods for running queries on the Base Facts data. It extends the com.sas.solutions.finance.api.BaseQuery class.

For an example of using the AdminQuery class, see the Facts stored process located in the following directories:

- Windows

  ```
  !sasroot\finance\sasstp\facts.sas
  ```

- UNIX

  ```
  !sasroot/sasstp/finance/facts.sas
  ```

The Facts stored process lists data records that are associated with a specified model. You can limit a Facts report several ways, including to a time period or an analysis member.

Method Summary

Table 3.2 *AdminQuery Class Method Summary*

Method	Description
`AdminQuery()`	Constructor. Throws: java.lang.Exception
boolean `executeQuery`()	Executes the query using the filters and any other parameters that have been previously specified. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
java.lang.String `getQueryColNames` (java.lang.String queryType)	Gets the list of column names for a specific query and model. This method can be executed before running a query. However, you must set the model to be used in the query before calling getQueryColNames. Parameters: ■ queryType—type of query to execute. For a list of possible values, see the definition of the setQueryType method. Returns: column names, separated by commas
java.lang.String `getQueryColNames` (java.lang.String queryType, java.lang.String modelCode)	Gets the list of column names for a specific query and model. This method can be executed before running a query. Parameters: ■ queryType—type of query to execute. For a list of possible values, see the definition of the setQueryType method. ■ modelCode—model code to be used in the query. Returns: column names, separated by commas Throws: java.lang.Exception
java.lang.String `getQueryColNamesWithSeparator` (java.lang.String queryType, java.lang.String modelCode, java.lang.String separator)	Gets the list of column names for a specific query and model. This method can be executed before running the query. Parameters: ■ queryType—type of query to execute. For a list of possible values, see the definition of the setQueryType method. ■ modelCode—model code to be used in the query. ■ separator—text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text Throws: java.lang.Exception

Method	Description
java.lang.String `getQuerySASNames` (java.lang.String queryType)	Gets a list of column names (in SAS valid name format) for a specific query and model. This method can be executed before running a query. However, you must first set the model to be used in the query. Parameters: ■ queryType—type of query to execute. For a list of possible values, see the definition of the setQueryType method. Returns: column names, separated by commas
java.lang.String `getQuerySASNames` (java.lang.String queryType, java.lang.String modelCode)	Gets a list of column names (in SAS valid name format) for a specific query and model. This method can be executed before running the query. It returns column names, separated by commas. Parameters: ■ queryType—type of query to execute. For a list of possible values, see the definition of the setQueryType method. ■ modelCode—model code to be used in the query. Returns: column names, separated by commas Throws: java.lang.Exception
java.lang.String `getQuerySASNamesWithSeparator` (java.lang.String queryType, java.lang.String modelCode, java.lang.String separator)	Gets the list of column names (in SAS valid name format) for a specific query and model. This method can be executed before running the query. Parameters: ■ queryType—type of query to execute. For a list of possible values, see the definition of the setQueryType method. ■ modelCode—model code to be used in the query. ■ separator—text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text Throws: java.lang.Exception
java.lang.String `getReportingCurrency()`	Gets the reporting currency member code. Returns: If the reporting currency has been set (via the setReportingCurrency method), then that member code is returned. Otherwise, the default reporting currency is returned. Throws: java.lang.Exception
boolean `setDimFilter` (java.lang.String code, java.lang.String value)	Sets a filter on a dimension; to filter on multiple values, call the method for each value. Parameters: ■ code—dimension code ■ value—member code to be used as the filter value Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception

Method	Description
boolean `setDimFilterID` (java.lang.String dimID, java.lang.String memID)	Sets a filter on a dimension; to filter on multiple values, call the method for each value. Parameters: ■ dimID—dimension ID ■ memID—member reference ID to be used in the filter Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setDimTypeFilter` (java.lang.String code, java.lang.String value)	Sets a filter on a dimension type; to filter on multiple values, call the method for each value. Parameters: ■ code—dimension type code. ■ value: the member code to be used in the filter. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setFactsParms` (java.lang.String otid, java.lang.String oid, java.lang.String ssid)	Deprecated. Use setParms instead.
boolean `setFormSetID` (java.lang.String name)	Sets the form set ID to be used in a query. Parameters: ■ name—name of the form set. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setModel` (java.lang.String name)	Sets the model to be used in a query. If the setLocale method was called, the specified locale is used to retrieve the correct model. Otherwise, the system default locale is used. Because model names can vary by locale, calling setLocale first is recommended. Parameters: ■ name—name of the model. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setModelCode` (java.lang.String code)	Sets the model to be used in a query. Parameters: ■ code—code of the model. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception

Method	Description
boolean `setModelID` (java.lang.String ID)	Sets the model to be used in a query. Parameters: ■ ID—model ID. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setParms` (java.lang.String otid, java.lang.String oid, java.lang.String convert)	Sets the parameters for the query. Parameters: ■ otid—object type ID, which must be a string containing one of these values: **`adjustmentsequence`**, **`attachment`**, **`cashinfusiontransaction`**, **`cellcomment`**, **`compositeresult`**, **`customanalytics`**, **`cycle`**, **`dataload`**, **`differentialwritedown`**, **`disposaltransaction`**, **`dividendtransaction`**, **`equityassignment`**, **`forcast`**, **`form`**, **`formset`**, **`formtemplate`**, **`holding`**, **`holdingmethodaccounts`**, **`lineitem`**, **`manualadjustment`**, **`measureexport`**, **`othercpolineitem`**, **`othercpotransaction`**, **`ownershipchangetransaction`**, **`period`**, **`phase`**, **`phaseset`**,**`pocconsolidationmethod`**, **`pocholdingfact`**, **`purchaseadjustment`**, **`purchasedifferential`**, **`purchasetransaction`**, **`result`**, **`rule`**, **`shortcut`**, **`standaloneparent`**, **`supplemental`**, **`task`**, or **`balsheet_reversal`**. ■ oid—object ID. Typically, this value is an empty string (**`""`**). ■ convert—currency conversion flag. A value of **`Y`** specifies that currency values should be converted from their functional currencies to a presentation currency. A value of **`N`** specifies that conversion should not take place. Returns: **`true`** if the parameter values are valid; otherwise, **`false`**
boolean `setQueryType` (java.lang.String queryType)	Sets the type of query to execute. Parameters: ■ queryType—type of query to execute, which must be a string containing one of these values: **`ELIMINATIONS`**, **`NONLEAF`**, **`DATAENTRY`**, **`TRIALBALANCE`**, **`INTERCOMPANY`**, **`NONINTERCOMPANY`**, **`RULESFACTS`**, **`RULE`**, **`MANUALADJUSTMENTS`**, **`FACTS`**, **`OWNERSHIP`**, **`ICACCOUNTS`**, **`FACTSR`**, **`DETAILS`**, **`OWNERSHIPTRANSACTIONS`**, or **`OWNERSHIPMETHODS`** Returns: **`true`** if the query type value is valid; otherwise, **`false`**

Method	**Description**
boolean `setReportingCurrency` (java.lang.String code)	Sets the currency to be used for reporting values. Parameters: ■ code—a currency code, such as **`EUR`**. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setRule` (java.lang.String name)	Sets the rule by name (required only by the RULE query). Parameters: ■ name—name of a rule. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setRuleID` (java.lang.String id)	Sets the rule by ID (required only by the RULE query). Parameters: ■ id—ID of a rule. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setVCubeID` (java.lang.String ID)	Sets the model using the ID of a virtual cube (vcube). Parameters: ■ ID—ID of a virtual cube. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception

The following methods are inherited from class com.sas.solutions.finance.api.BaseQuery:

- getColumnName
- getColumnSASName
- getColumnType
- getMaxRowsMessage
- getNumberOfColumns
- getNumberOfColumn
- getNumberOfColumns
- getNumericValue
- getQueryColNames
- getQueryColNamesWithSeparator
- getQueryRecordsNumber
- getQuerySASNames

- getQuerySASNamesWithSeparator
- getRecord
- getValue
- setMaxRows

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- trim

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The AuditHistory Class

Overview

The com.sas.solutions.finance.api.AuditHistory class contains methods for running queries on AuditHistory data from SAS Financial Management. This class extends the com.sas.solutions.finance.api.BaseQuery class.

For an example of using the AuditHistory class, see the Audit stored process that is located in the following directories:

- Windows

  ```
  !sasroot\finance\sasstp\audit.sas
  ```

- UNIX

  ```
  !sasroot/sasstp/finance/audit.sas
  ```

This stored process extracts audit and history data that is filtered by three optional parameters: a user, an action type, and a date range.

Note: By default, the Audit stored process returns a maximum of 10,000 records. You can modify the Audit stored process to return more records by adding the setMaxRows method to the stored process. When adding the setMaxRows method, ensure that you note memory considerations.

Method Summary

Table 3.3 *AuditHistory Class Method Summary*

Method	Description
`AuditHistory()`	Constructor. Throws: java.lang.Exception
boolean `executeQuery()`	Executes the query using the filters and any other parameter previously specified. The query generates records with the following columns (all are character data): USERNAME, ACTION_TYPE_ID, TIMESTAMP_TS, OBJECT_CLASS_ID, OBJECT_ID, TRANSACTION_ID, AUDIT_ID, PROPERTY_NM, OLD_VALUE, and NEW_VALUE. You can call the getValue method to retrieve these values. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
java.lang.String `getQueryColNames()`	Gets the list of column names that were returned by the AuditHistory query. This method can be called only after running the query. Returns: column names, separated by commas
java.lang.String `getQueryColNames` (java.lang.String separator)	Gets the list of column names that were returned by the AuditHistory query. This method can be called only after running the query. Parameters: ■ separator—text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
void `setDateFormat` (java.lang.String format)	Sets the desired format for passing the dates when calling setDateRange. Throws: java.lang.Exception
boolean `setDateRange` (java.lang.String from, java.lang.String to)	Sets a date range. Dates are expected to be in the format *mm*/*dd*/*yyyy* unless they are otherwise specified by a call to setDateFormat. Throws: java.lang.Exception

Method	Description
boolean `setFilter` (java.lang.String name, java.lang.String value)	Sets a filter on a column. To filter on multiple values, call the method for each value. Parameters: ■ name—column name. For a list of valid column names, see the description of the executeQuery method. ■ value—value for the filter. For example, if you wanted to see audit records for the sasdemo user, you would use a call like this: `j.callBooleanMethod("setFilter", "username", "sasdemo", rc);` Throws: java.lang.Exception

The following methods are inherited from class com.sas.solutions.finance.api.BaseQuery:

- getColumnName
- getColumnSASName
- getColumnType
- getMaxRowsMessage
- getNumberOfColumns
- getNumericValue
- getQueryColNames
- getQueryColNamesWithSeparator
- getQueryRecordsNumber
- getQuerySASNames
- getQuerySASNamesWithSeparator
- getRecord
- getValue
- setMaxRows

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- setLocale
- trim

The following methods are inherited from class java.lang.Object:

- equals

- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The BaseApi Class

Overview

The com.sas.solutions.finance.api.BaseApi class is extended by the BaseQuery, Form, Metadata, and Model classes.

Note: BaseApi methods should be called only by one of its subclasses.

Method Summary

Table 3.4 BaseApi Class Method Summary

Method	Description
`BaseApi()`	Constructor. Throws: java.lang.Exception
java.lang.String `getErrorMessage()`	Gets the localized error message from the last action. If the setLocale method was called, the specified locale is used. Otherwise, the system default locale is used. Returns: a localized message string

Method	Description

java.lang.String `getMessage` (java.lang.String message)	Gets the localized message that corresponds to a message code. If the setLocale method was called, the specified locale is used. Otherwise, the system default locale is used.
	
	Parameters:
	
	■ message—identifier for a localized message string.
	
	For a list of valid message codes, see the Resources_*language-code*.properties files in the sas.solutions.finance.api.nls.jar file.
	
	To locate the correct JAR file, open the **`!sasroot\picklist\finance\finance.txt`** file and find the following name: **`sas.solutions.finance.api`**. Make a note of the version that corresponds to this name. The JAR file is in the ***`SAS-install-dir`*** **`\SASVersionedJarRepository\`*****`version`*** directory.
	
	Returns: a localized message string
	
	Example:
	
	`j.callStringMethod("getMessage",`
	`   "Api.QueryReturnedNoFacts.txt", msg);`
	` call symput('msg', msg);`

boolean `setLocale` (java.lang.String l)	Sets the locale. (The default locale is the system default locale.) Parameters: ■ l—locale that is specified as *language-code_country-code*, such as **`en_US`** or **`es_SP`**. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. Returns: **`true`** if the action succeeded; otherwise, **`false`**
java.lang.String `trim` (java.lang.String s)	Returns the value passed in. Trailing blanks are removed.

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The BaseQuery Class

Overview

The com.sas.solutions.finance.api.BaseQuery class is extended by the AdminQuery, AuditHistory, and CycleQuery classes. This class contains methods for retrieving the results of a query.

Note: The methods of the BaseQuery class should be called only from one of its subclasses.

Method Summary

Table 3.5 *BaseQuery Class Method Summary*

Method	**Description**
`BaseQuery()`	Constructor. Throws: java.lang.Exception
java.lang.String `getColumnName` (double n)	Gets the name of the *n*th column.
java.lang.String `getColumnSASName` (double n)	Gets the SAS name of the *n*th column.
java.lang.String `getColumnType` (double n)	Gets the column type (numeric or character) of the *n*th column.
java.lang.String `getMaxRowsMessage()`	Gets the maximum number of rows that a query can return. If the query returns fewer than this maximum number of rows, the getMaxRowsMessage method returns an empty string. Otherwise, it returns a localized message with this string: **`Showing the first n rows`**, where *n* is the maximum number of rows that were requested.
int `getNumberOfColumns()`	Gets the number of columns returned by the query. This method can be executed only after a query has run.
double `getNumericValue` (double n, double m)	Gets the numeric value of the *n*th column of the *m*th record.
java.lang.String `getQueryColNames()`	Gets the list of column names that were returned by a query. This method can be executed only after the query has been run. Returns: column names, separated by commas

Method	Description
java.lang.String `getQueryColNamesWithSeparator` (java.lang.String separator)	Gets the list of column names that were returned by a query. This method can be executed only after the query has been run. Parameters: ■ separator—text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
int `getQueryRecordsNumber()`	Gets the number of records (facts) that were returned by the query.
java.lang.String `getQuerySASNames()`	Gets a list of column names (in SAS valid name format) that were returned by a query. This method can be executed only after the query has been run. Returns: column names, separated by commas
java.lang.String `getQuerySASNamesWithSeparator` (java.lang.String separator)	Gets the list of column names (in SAS valid name format) that were returned by a query. This method can be executed only after the query has been run. Parameters: ■ separator—text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
java.lang.String `getRecord` (double n)	Gets the *n*th record. Parameters: ■ **`n`**—index of a record in the query results. Returns: record values, separated by commas
java.lang.String `getRecord` (double n, java.lang.String separator)	Gets the *n*th record. Parameters: ■ n—the index of a record in the query results. ■ separator—text to be used as a separator, such as a comma. Returns: record values, separated by the separator text
java.lang.String `getValue` (double n, double m)	Gets the value of the *n*th column of the *m*th record.
boolean `setMaxRows` (java.lang.String s)	Sets the maximum number of records (or facts) a query can return. The default is 50,000,000 rows. Parameters: ■ s—maximum number of rows. A value of `0` specifies no limit. Returns: **`true`** if the action succeeded; otherwise, **`false`**

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- setLocale
- trim

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The CellComments Class

Overview

The com.sas.solutions.finance.api.CellComments class has a method for extracting cell comments from the SAS Financial Management data mart. Instead of calling this method directly, call the %FMCELLC macro.

The %FMCELLC Macro

The %FMCELLC macro extracts cell comments from the SAS Financial Management data mart for a specific cycle. The macro returns all comments, both public and private, that match the selection criteria such as cycle name.

Note: Only form administrators can execute a command using the %FMCELLC macro.

The macro writes those comments to a data set with the following variables:

Variable	Description
COMMENT_ID	The ID of the comment in the database.
USER_ID	The user ID of the user who made the comment.
DOCUMENT_TYPE	One of the following: `FINANCIAL_FORM_SET` or `FINANCIAL_REPORT`.

Variable	Description
DOCUMENT_NAME	For a form, this variable contains the form set name. For a report, this variable contains the report ID, in the form `FINANCIAL_REPORT`_*report-ID*.
CYCLE_TYPE	This column always contains `FINANCIAL`.
CYCLE_NAME	The name of the cycle.
PARENT_COMMENT_ID	If the comment is a reply to a previous comment, this variable contains the ID of the previous comment. Otherwise, the variable contains the ID of the current comment.
PRIVACY	One of the following: `PUBLIC` or `PRIVATE`. **Note:** Private comments are not filtered by user.
COMMENT_TEXT	The text of the comment.
CREATED_DATE_TIME	The date and time the comment was created.
VISIBILITY	One of the following: `FORM_SET` (visible within the form set or report in which the comment was made) or `CYCLE` (visible within the cycle).
MODEL_CODE	The code for the model that is associated with the comment.
*dimension*_CODE	Codes for dimensions that make up the crossing for the comment.

Note: The code uses a delimiter of "|" when creating the result rows. If any of the fields in the output (for example, the cycle name or the comment text) contain this character, an underscore is substituted in the output. For example, a cell comment of "a|b|c" would become "a_b_c".

Syntax

%FMCELLC (
  CYCLENAME=""
   [, **DOCUMENTNAME**=""
   [, **DOCUMENTTYPE**="FINANCIAL_FORM_SET"
   [, **STARTDATETIME**=""
   [, **ENDDATETIME**=""
   [, **LOCALSASLIBNAME**="Work"
   [, **RESULTDATASETNAME**="CellCommentDataSet"
   [, **ENVIRONMENT**="default"
)

CYCLENAME
  The name of the cycle for the query. Required.

DOCUMENTNAME
: If you want to restrict the query to comments that were added to a specific form set, specify this parameter with the form set name.

  If you specify this parameter, you must also specify the form set type (unless you use the default).

DOCUMENTTYPE
: Either **`FINANCIAL_FORM_SET`** or **`FINANCIAL_REPORT`**.

  If you omit the DOCUMENTNAME parameter, then the DOCUMENTTYPE parameter is ignored.

STARTDATETIME, ENDDATETIME
: Starting and ending dates and times for the query in the form *yyyy-mm-dd hh:mm:ss*.

LOCALSASLIBNAME
: The library in which to create the output data set. The default is the WORK library.

RESULTDATASETNAME
: The name of the result data set. The default name is “CellCommentDataSet”.

ENVIRONMENT
: An environment (such as “default”, “dev”, or “prod”) refers to an installation of SAS Financial Management. If you omit this parameter, “default” is used.

  The environment value is site-specific. For more information about the SAS Environment, see “Specifying the SAS Financial Management Environment” on page 18.

Note: These parameters are not case sensitive.

Examples

The following is a list of examples about how to use the %FMCELLC macro:

- Get all cell comments for the specified cycle. Comments are returned in WORK.CellCommentDataSet:

```
%FMCELLC(CYCLENAME="MyCycle");
```

- Get all cell comments for the specified cycle that were made since September 1, 2011. Display only comments that were added to reports:

```
%FMCELLC(CYCLENAME="MyCycle", STARTDATETIME="2011-09-01 00:00:00");
PROC PRINT DATA=CellCommentDataSet NOOBS;
    WHERE DOCUMENT_TYPE="FINANCIAL_REPORT";
run;
```

- Get all cell comments that were added to the specified form set. Display only public comments:

```
%FMCELLC(CYCLENAME="MyCycle", DOCUMENTNAME="MyFormset");
proc print data=CellCommentDataSet NOOBS;
    where PRIVACY="PUBLIC";
run;
```

For more information about cell comments, see the *SAS Financial Management Process Administrator’s Guide*.

The CycleQuery Class

Overview

The com.sas.solutions.finance.api.CycleQuery class contains methods for extracting facts from a cycle. It extends the com.sas.solutions.finance.api.BaseQuery class.

The CycleQuery class is similar to the AdminQuery class.

For an example of using the CycleQuery class, see the ETL Facts stored process that is located in the following directories:

- Windows

  `!sasroot\finance\sasstp\etlfacts.sas`

- UNIX

  `!sasroot/sasstp/finance/etlfacts.sas`

The ETL Facts stored process lists data records that have been loaded from SAS Data Integration Studio to a specified time period and analysis member within a specified cycle, and perhaps a specified organization.

Method Summary

Table 3.6 CycleQuery Class Method Summary

Method	Description
`CycleQuery()`	Constructor. Throws: java.lang.Exception
boolean `getETLFacts()`	Gets the ETL facts for the specified cycle and filters. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
java.lang.String `getQueryColNames` (java.lang.String cycleName, java.lang.String separator)	Gets the list of column names for the query. This method can be called before running the query. Parameters: ■ cycleName: the name of a cycle ■ separator: the text, such as a comma, to be used as a separator Returns: a list of column names, separated by the separator text. Throws: java.lang.Exception

Method	Description
java.lang.String `getQueryColNamesByID` (java.lang.String cycleID, java.lang.String separator)	Gets the list of column names for the query. This method can be called before running the query. Parameters: ■ cycleID: the ID of a cycle. ■ separator: the text, such as a comma, to be used as a separator. Returns: a list of column names, separated by the separator text Throws: java.lang.Exception
java.lang.String `getQuerySASNames` (java.lang.String cycleName, java.lang.String separator)	Gets a list of column names (in SAS valid name format) for a specific query and cycle. This method can be executed before running the query. Parameters: ■ cycleName: the cycle name. ■ separator: the text, such as a comma, to be used as a separator. Returns: a list of column names, separated by the separator text Throws: java.lang.Exception
java.lang.String `getQuerySASNamesByID` (java.lang.String cycleID, java.lang.String separator)	Gets a list of column names (in SAS valid name format) for a specific query and cycle. This method can be executed before running the query (after setting the cycle to be used in the query). Parameters: ■ cycleID: the cycle ID. ■ separator: the text, such as a comma, to be used as a separator. Returns: a list of column names, separated by the separator text Throws: java.lang.Exception
boolean `setCycleByID` (java.lang.String ID)	Sets the cycle for the query by ID. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setCycleByName` (java.lang.String name)	Sets the cycle for the query by name. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception

Method	Description
boolean `setDimTypeFilter` (java.lang.String code, java.lang.String value)	Sets a filter on a dimension type; to filter on multiple values, call the method for each value. Parameters: ■ code: the dimension type code. ■ value: the member code to be used in the filter. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean `setParms` (java.lang.String otid, java.lang.String oid)	Sets the parameters for the query. Parameters: ■ otid: the object type ID. Possible values are **`adjustmentsequence`**, **`attachment`**, **`cashinfusiontransaction`**, **`cellcomment`**, **`compositeresult`**, **`customanalytics`**, **`cycle`**, **`dataload`**, **`differentialwritedown`**, **`disposaltransaction`**, **`dividendtransaction`**, **`equityassignment`**, **`form`**, **`forecast`**, **`formset`**, **`formtemplate`**, **`holding`**, **`holdingmethodaccounts`**, **`lineitem`**, **`manualadjustment`**, **`measureexport`**, **`othercpolineitem`**, **`othercpotransaction`**, **`ownershipchangetransaction`**, **`period`**, **`phase`**, **`phasestep`**, **`pocconsolidationmethod`**, **`pocholdingfact`**, **`process`**, **`purchaseadjustment`**, **`purchasedifferential`**, **`purchasetransaction`**, **`shortcut`**, **`result`**, **`rule`**, **`standaloneparent`**, **`supplemental`**, **`task`**, or **`balsheet_reversal`**. ■ oid: the object ID. Returns: **`true`** if the parameter values are valid; otherwise, **`false`** Throws: java.lang.Exception

The following methods are inherited from class com.sas.solutions.finance.api.BaseQuery:

- getColumnName
- getColumnSASName
- getColumnType
- getMaxRowsMessage
- getNumberOfColumns
- getNumericValue
- getQueryColNames
- getQueryColNamesWithSeparator
- getQueryRecordsNumber

- getQuerySASNames
- getQuerySASNamesWithSeparator
- getRecord
- getValue
- setMaxRows

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- setLocale
- trim

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The Form Class

Overview

The com.sas.solutions.finance.api.Form class contains methods for running queries on the properties of a planning form from SAS Financial Management. It extends the com.sas.solutions.finance.api.BaseApi class.

For an example of using the Form class, see "Data Validation Example" on page 80.

Method Summary

Table 3.7 Form Class Method Summary

Method	Description
`Form` (java.lang.String sFormId, java.lang.String entityKey)	Constructor. This constructor can be used only in a stored process that is used in a workflow. Both the form ID and the security key (entityKey) are available as environment variables that are set by the workflow. Throws: java.lang.Exception
java.lang.String `getAuthors` (java.lang.String delimiter)	Returns the user IDs of all authors of a specified form, separated by the *delimiter* text if more than one author was found. Parameters: ▪ delimiter: the text (such as a space or semi-colon) that is used to separate author names in the return string. Throws: java.lang.Exception
java.lang.String `getDescription()`	Returns the form description.
java.lang.String `getDueDate()`	Returns the due date of the form.
java.lang.String `getFormSetDescription()`	Returns the description of the form set to which the form belongs.
int `getFormSetId()`	Returns the ID of the form set to which the form belongs.
java.lang.String `getFormSetName()`	Returns the name of the form set to which the form belongs.
java.lang.String `getId()`	Returns the form ID as a string.
java.lang.String `getInfo()`	Returns a formatted string with key information about the form.
java.lang.String `getName()`	Returns the form name.
java.lang.String `getPlanningAdministrators` (java.lang.String delimiter)	Returns a list of users with the role of Finance Process Administrator. Parameters: ▪ delimiter: the text that is used to separate names in the return string. Throws: java.lang.Exception
java.lang.String `getReviewers` (java.lang.String delimiter)	Returns all reviewers of a specified form. The reviewers are separated by the *delimiter* text if more than one reviewer was found. Parameters: ▪ delimiter: the text that is used to separate names in the return string. Throws: java.lang.Exception

Method	Description
java.lang.String `getState()`	Returns the form state.
java.lang.String `getTargetDimensionCode()`	Returns the code of the target dimension of the form set to which the form belongs.
java.lang.String `getTargetDimensionDescription()`	Returns the description of the target dimension of the form set to which the form belongs.
java.lang.String `getTargetDimensionName()`	Returns the name of the target dimension of the form set to which the form belongs.
java.lang.String `getTargetMemberCode()`	Returns the target member code of the form.
java.lang.String `getTargetMemberDescription()`	Returns the description of the target member of the form.
int `getTargetMemberId()`	Returns the target member ID of the form.
java.lang.String `getTargetMemberName()`	Returns the name of the target member of the form.
boolean `isLocked()`	Returns **`true`** if the form is locked by some process.

Note: The constructor that took a user ID and password as parameters is no longer supported. Use one of the other constructors instead.

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- setLocale
- trim

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The FormSet Class

Overview

You can use the methods of the com.sas.solutions.finance.api.FormSet class to manage form sets from SAS code. For example, you can reset a form set or you can write a batch script to publish a form set. You can also use the FormSet class to reallocate the values for some or all tables in some or all forms.

Example — Resetting Two Form Sets and Extending Their Deadlines

The following example resets two form sets and extends their deadlines:

Example Code 3.1 *Resetting Form Sets and Extending Deadlines*

```
dcl javaobj formset('com/sas/solutions/finance/api/FormSet');
formset.ExceptionDescribe(1);
call METADATA_PASSID('formset','');
formset.callVoidMethod('setCycle', 'MyCycleName');
formset.callVoidMethod('setFormSet','MyFormSetName');
formset.callVoidMethod('enableUserNotification');
formset.callVoidMethod('enableCommentRetirement');
formset.callVoidMethod('setComment','My reset comment');
formset.callVoidMethod('reset');
formset.callVoidMethod('setDeadline','Apr 10, 2010 9:00 AM');
formset.callVoidMethod('setComment','My publish comment');
formset.callVoidMethod('publish');
formset.callVoidMethod('setFormSet','MyOtherFormSetName');
formset.callVoidMethod('reset');
formset.callVoidMethod('moveDeadlineByCalendarMonth',1);
formset.callVoidMethod('moveDeadlineByDaysAndHours',0,1);
formset.callVoidMethod('publish');
formset.delete();
```

Note: When a date string is specified as a parameter, use a format such as the following: “Apr 10, 2010 3:00 PM” or “10 avr 2010 15:00:00”. Returned date strings have the same format. The system default locale is used.

Examples — Reallocating Table Values

Usage Notes

When using the Reallocate Table feature, note the following:

- The Reallocate Table option is available when the “Allocate from Parent members other than Time using predefined weights” is enabled. If the Allocation Weights are not “Same as Target value,” you might need to reallocate the values at the top member(s) in each dimension when new weight values are loaded. The Reallocate Table API methods enable the reallocation of values for some or all data entry tables in some or all forms.

The form set must be published and the form(s) must be writable. The API skips any forms that are not writable (for example, in a draft, locked, or submitted state). If a form is not writable, it is skipped by the reallocation process.

- When using the Reallocation Table methods, errors are recorded in SASSDM in a table that is named sas_table_reallc_report. The maximum number of errors recorded is set by the method. If you specify a maximum error threshold of “1”, then the first error is recorded in the table and the job stops. If you specify a maximum error threshold of “10”, then the first ten errors are recorded in the table and the job stops.

  A valid value for the maximum error threshold is a number between 1 and 1000, or 0. When you specify 0, the first 1000 errors are recorded and the job continues to run, but additional errors are not recorded. A value must be specified for the maximum error threshold or the job does not run.

  The sas_table_reallc_report table in SASSDM contains the following columns: report_id, form_set_id, form_id, internal_table_nm, external_table_nm, crossing, error_code_id, and transaction_id. The description of an error code is available by using the GetErrorDescriptions method. A user can use the GetErrorDescriptions method to create custom reports about the results of the table reallocation job. When a table reallocation job is re-executed, the records in the table are overwritten.

- If all tables in a form are reallocated, the default order is alphanumeric, based on the external_table_nm. If you want to reallocate the tables in a different order, you can specify the order of the tables in the array of tables.

- By default, tables are reallocated sequentially within the same job and in alphanumerical order within each form. For example: 1. Form 10 — Table A, 2. Form 10, Table B, 3. Form 20 — Table A, 4. Form Table B.

  If the tables are independent of each other, you can create a separate job for each form, or for each table in a form, so that they are processed concurrently. This might decrease processing time. For example: 1. Form 10 — Table A, 2. Form 10 — Table B, 1. Form 20 — Table A, 2. Form 20 — Table B, or 1. Form 10, Table A, 1. Form 10, Table B, 1. Form 20 — Table A, 1. Form 20, Table B.

- When Table Reallocation is initiated by using the API, a history record is generated for the form set, unless the API fails at the job level. The following conditions are examples of what causes a failure at the job level:
  - when the cycle or form set do not exist
  - when the value that is specified for the error tolerance is not valid

Examples — Reallocating Table Values

The following example reallocates the values in all of the tables in all forms in a form set, with the maximum number of errors that can occur set to 1000:

Example Code 3.2 *Reallocating the Values for All Forms in a Form Set*

```
dcl javaobj formset('com/sas/solutions/finance/api/FormSet');
formset.ExceptionDescribe(1);
call METADATA_PASSID('formset','');
formset.callVoidMethod('setCycle', 'MyCycleName');
formset.callVoidMethod('setFormSet','MyFormSetName');
formset.callVoidMethod('reallocateAllForms','1000');
```

The following example reallocates the values in tables in two forms in a form set, with the maximum number of errors that can occur set to 500:

Example Code 3.3 *Reallocating Values at the Table Level in Two Forms in a Form Set*

```
dcl javaobj formset('com/sas/solutions/finance/api/FormSet');
formset.ExceptionDescribe(1);
call METADATA_PASSID('formset','');
formset.callVoidMethod('setCycle', 'MyCycleName');
formset.callVoidMethod('setFormSet','MyFormSetName');
array memberCodes[2] $50
("1","2");
formset.callVoidMethod('reallocateSelectedForms',memberCodes,'500');
;
```

The following example reallocates the values for specific tables in a form, with the maximum number of errors that can occur set to 200:

Example Code 3.4 *Reallocating Values in Select Tables in a Form*

```
dcl javaobj formset('com/sas/solutions/finance/api/FormSet');
formset.ExceptionDescribe(1);
call METADATA_PASSID('formset','');
formset.callVoidMethod('setCycle', 'MyCycleName');
formset.callVoidMethod('setFormSet','MyFormSetName');
array tableCodes[2] $50
("Table1","Table2");
formset.callVoidMethod('reallocateSelectedTables',tableCodes,'200');
```

The following example reallocates the values for specific tables in specific forms in a form set with no maximum number of errors set:

Example Code 3.5 *Reallocating the Values for Select Forms and Tables in a Form Set*

```
dcl javaobj formset('com/sas/solutions/finance/api/FormSet');
formset.ExceptionDescribe(1);
call METADATA_PASSID('formset','');
formset.callVoidMethod('setCycle', 'MyCycleName');
formset.callVoidMethod('setFormSet','MyFormSetName');
array memberCodes[2] $50
("1","2");
array tableCodes[2] $50
("Table1","Table2");
formset.callVoidMethod('reallocateSelectedFormsAndTables',memberCodes,tableCodes,'0');
```

Note: When *0* is specified for the maximum number of errors, the first 1000 errors are recorded in the error table and the job continues without recording additional errors.

Method Summary

The following methods can be called for the FormSet class.

Table 3.8 FormSet Class Method Summary

Method	Description
`FormSet` ()	Constructor. Throws: java.lang.Exception
void `disableCommentRetirement` ()	Disables comment retirement for the publish action. With comment retirement disabled, when you publish the form set, comments that were associated with the form set are retained.
void `disableUserNotification` ()	Disables user notification for the publish and reset actions.
void `enableCommentRetirement` ()	Enables comment retirement for the publish action. This is the default behavior. When you publish the form set, comments that were associated with the form set are retired.
void `enableUserNotification` ()	Enables user notification for the publish and reset actions.
void `reallocateAllForms` (tolerance)	Reallocates the values of all tables in all of the forms in a form set. Parameters: ■ tolerance: the maximum number of errors that can occur during a job. When this value is reached, the job is stopped. Errors are based on the crossings the allocation is occurring from, not the crossings to which the value is being allocated. Therefore, the potential total number of errors in a job equals the number of crossings being allocated from. The error tolerance is a required parameter. A valid value is a number between 1 to 1000 or 0. If you do not specify a value for the error tolerance parameter, the job does not execute. You can run reallocation on only published (writable) form sets. Reallocation is run only on unlocked forms. Reallocation at the form set uses the weights specified at the form level.
void `reallocateSelectedForms` (memberCodes, tolerance)	Reallocates the values of the tables in select forms in a form set. Parameters: ■ memberCodes: codes that identify the forms. ■ tolerance: the maximum number of errors that can occur during a job. When this value is reached, the job is stopped. Errors are based on the crossings the allocation is occurring from, not the crossings to which the value is being allocated. Therefore, the potential total number of errors in a job equals the number of crossings being allocated from. The error tolerance is a required parameter. A valid value is a number between 1 to 1000 or 0. If you do not specify a value for the error tolerance parameter, the job does not execute.

Method	Description
void `reallocateSelectedTables` (tableCodes, tolerance)	Reallocates the values of select tables in a form. Parameters: ■ tableCodes: names of the tables in the form as defined and visible in the table properties. ■ tolerance: the maximum number of errors that can occur without stopping the process during a job. When this value is reached, the job is stopped. Errors are based on the crossings that the allocation is occurring from, not the crossings to which the value is being allocated. Therefore, the potential total number of errors in a job equals the number of crossings that are being allocated from. The error tolerance is a required parameter. A valid value is a number between 1 to 1000 or 0. If you do not specify a value for the error tolerance parameter, the job does not execute. **Note:** If running table reallocation on multiple tables, you can specify the names of the tables in the order in which you want them to be reallocated.
void `reallocateSelectedFormsandTables`(memberCodes, tableCodes, tolerance)	Reallocates the values of select forms and selected tables in a form set. Parameters: ■ memberCodes: codes that identify the forms. ■ tableCodes: names of the tables in the form set as defined and visible in the table properties. ■ tolerance: the maximum number of errors that can occur without stopping the process. Errors are based on the crossings that the allocation is occurring from, not the crossings to which the value is being allocated. Therefore, the potential total number of errors in a job equals the number of crossings that are being allocated from. The error tolerance is a required parameter. A valid value is a number between 1 to 1000 or 0. If you do not specify a value for the error tolerance parameter, the job does not execute. **Note:** If running table reallocation on multiple tables, you can specify the names of the tables in the order in which you want them to be reallocated.
callStringMethod (‘getErrorDescriptions’)	Returns a delimited string of error codes and descriptions specific to the Table Reallocation process. The delimiter is specified by the caller of the API.
java.lang.String `getComment` ()	Returns the comment to be used for publish and reset actions.
java.lang.String `getCycle` ()	Returns the cycle name.
java.lang.String `getDeadline` ()	Returns the due date for the form set. Throws: FinanceClientException
java.lang.String `getFormSet` ()	Returns the form set name.
java.lang.String `getTargetHierarchyAsOfDate` ()	Returns the as-of date for the target hierarchy. Throws: FinanceClientException

Method	Description
boolean `isCommentRetirementEnabled` ()	Returns the current comment retirement behavior for publication actions: **`true`** if comments are retired at publish time; otherwise **`false`**.
boolean `isLocked` ()	Returns the lock status of the form set: **`true`** if the form set is locked; otherwise **`false`**. Throws: FinanceClientException
boolean `isUserNotificationEnabled` ()	Returns the current user notification behavior for publish and reset: **`true`** if user notification is enabled; otherwise **`false`**.
void `lock` ()	Locks the form set. Throws: FinanceClientException
void `moveDeadlineByCalendarMonth` (double months)	Moves the current deadline by the specified number of calendar months. Parameters: ■ months: the number of months (positive or negative) by which to move the form set deadline. Throws: FinanceClientException
void `moveDeadlineByDaysAndHours` (double days, double hours)	Moves the current deadline by the specified number of days and hours. Parameters: ■ days: the number of days (positive or negative) by which to move the form set deadline. ■ hours: the number of hours (positive or negative) by which to move the form set deadline. Throws: FinanceClientException
void `publish` ()	Publishes the form set. The form set must be in Draft status, it cannot be locked, and it must have a valid template. Throws: FinanceClientException
void `publish` (double max_seconds_to_wait)	Initiates the publish action and waits for its completion. The form set must be in Draft status, it cannot be locked, and it must have a valid template. Parameters: ■ max_seconds_to_wait: the maximum time (in seconds) to wait before issuing a time-out exception. The publish() method (without a parameter) initiates the publish action and returns. The publish(*max_seconds_to_wait*) method initiates the publish operation and then checks the status of the publish activity for at most *max_seconds_to_wait* seconds. When the publish completes, the method returns. If the publish activity has not completed within the specified time, the method throws a time-out exception. However, it does not cancel the publish action. Throws: FinanceClientException

Method	Description
void `reset` ()	Resets the form set. This operation is not possible if the form set is locked. If comment text has been defined, the comment is applied to the operation. Throws: FinanceClientException
void `restoreDefaults` ()	Restores the following default attributes of the FormSet object: ■ User notification is enabled. ■ Comments are deleted when the form set is published. ■ The comment text is set to an empty string. ■ The locale is set to the system default locale. Calling this method does not affect the form set name, cycle name, and cycle type that are associated with the FormSet object.
void `setComment` (java.lang.String comment)	Defines the comment text to be used for publish and reset actions. By default, this text is an empty string.
void `setCurrentUntilPublished` ()	Sets the due date of the target hierarchy to "current until published". Throws: FinanceClientException
void `setCycle` (java.lang.String name)	Associates a cycle name with the FormSet object. You must call setCycle before calling any of the following methods: publish, reset, setDeadline, getDeadline, moveDeadlineByCalendarMonth, moveDeadlineByDaysAndHours, setTargetHierarchyAsOfDate, getTargetHierarchyAsOfDate, setCurrentUntilPublished, lock, unlock, and isLocked. Parameters: ■ name: the name of the cycle for this form set.
void `setDeadline` (java.lang.String dateString)	Sets the due date and time for the form set. Parameters: ■ dateString: a string that contains the date and time of the deadline. Throws: FinanceClientException
void `setFormSet` (java.lang.String name)	Associates a form set with the FormSet object. Parameters: ■ name: the name of the form set.
void `setTargetHierarchyAsOfDate` (java.lang.String dateString)	Sets the as-of date for the target hierarchy. This method is valid only when the form set is in Draft state. Parameters: ■ dateString: the date and time to use for the target hierarchy's as-of date. Throws: FinanceClientException

Method	Description
void `unlock` (double unlockForms)	Unlocks the form set. Optionally, also unlocks forms of a form set. If the form set is already unlocked, you can still call this method to unlock its forms. Parameters: ■ unlockForms: If the value is `0`, any forms that are explicitly locked remain locked, even though the form set itself is unlocked. If the value is `1`, any forms that are explicitly locked are unlocked. Throws: FinanceClientException

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- setLocale
- trim

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The Metadata Class

Overview

The com.sas.solutions.finance.api.Metadata class contains methods for looking up SAS Financial Management metadata. It extends the com.sas.solutions.finance.api.BaseApi class.

For an example of using the Metadata class, see Chapter 5, “Creating a Custom Cell Action,” on page 85.

Method Summary

Table 3.9 Metadata Class Method Summary

Method	Description
`Metadata ()`	Constructor. Throws: java.lang.Exception
java.lang.String `getDimensionCode` (java.lang.String dimID)	Gets the dimension code. Parameters: ■ dimID: the dimension ID. Returns: the dimension code that corresponds to the dimension ID
java.lang.String `getMemberCode` (java.lang.String dimID, java.lang.String memID)	Gets the member code. Parameters: ■ dimID: the dimension ID. ■ memID: the member ID. Returns: the member code that corresponds to the specified dimension ID and member ID

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- setLocale
- trim

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The Model Class

Overview

The com.sas.solutions.finance.api.Model class contains methods for retrieving information about SAS Financial Management models and for running queries against a model. It extends the com.sas.solutions.finance.api.BaseModel class (which in turn extends com.sas.solutions.finance.api.BaseApi).

Most methods of the Model class now have corresponding macros (as listed the Model Class Method Summary table). Those methods are still supported for backward compatibility. However, we recommend using the macros instead. A few methods have no macro equivalents.

The Model class does not support composite models, with one exception: the %FMQUERY macro. For more information about the %FMQUERY macro, see“Executing Queries with the %FMQUERY Macro” on page 64.

Note: The Model class is intended to be used by administrators and power users. Security is applied for the user who is running the query. Keep that in mind if you make the query results available to other users.

Method Summary

Table 3.10 *Model Class Method Summary*

Method	Description
`Model`	Constructor. Throws: java.lang.Exception
`Model` (java.lang.String storedProcessEntityKey)	Constructor. This constructor can be used only in a stored process that is part of a workflow. Parameters: ■ storedProcessEntityKey: the security key that is passed from the workflow. Throws: java.lang.Exception
`executeQuery`	Deprecated. Use the %FMQUERY macro instead. For information about the %FMQUERY macro, see “Executing Queries with the %FMQUERY Macro” on page 64.

Method	Description
int `generateFormulaFacts` (java.lang.String cycleName, java.lang.String formSetName)	Computes and stores all driver formula output values for crossings in the selected form set. This method corresponds to the **Run driver formulas** option of SAS Financial Management Studio. The **Run driver formulas** option is used to ensure that the stored output values of all driver formulas are current. Before calling this method for an imported form set, save the form set template (in Microsoft Excel). Otherwise, the method returns an error. (This is also true when you select **Run driver formulas** in SAS Financial Management Studio.) Parameters: ■ cycleName: the name of the cycle to use. ■ formSetName: the name of the form set to use. Returns an integer containing the status code: ■ `0`: SUCCESS ■ `1`: OBJECT NOT FOUND ■ `2`: FORM SET IS LOCKED ■ `3`: GENERIC ERROR
`getAllModels`	Deprecated. Use the %GETALLMODELS macro instead. For information about the %GETALLMODELS macro see “Model Macros” on page 57.
double `getCellValue` (java.lang.String resultCode, java.lang.String[] dimensionCodes, java.lang.String[] memberCodes)	Gets the value of a crossing. Parameters: ■ resultCode: the code of the results model. ■ dimensionCodes: the list of dimension codes that define the crossing. ■ memberCodes: a matching list of member codes that define the crossing. If you omit a dimension, the default member for the model’s hierarchy in that dimension (at the model’s hierarchy as-of date) is used instead. For the Time dimension, this value is not necessarily the same as the default read member for the time hierarchy that can be set in the model. To avoid unexpected results, we recommend that you always include the Time dimension in your specification. Returns: the value of the specified crossing Throws: java.lang.Exception For an example, see Chapter 5, “Creating a Custom Cell Action,” on page 85.
`getModelHierarchies`	Deprecated. Use the %GETMODELHIERARCHIES macro instead. For information about the %GETMODELHIERARCHIES macro, see “Model Macros” on page 57.
`getModelMemberProperties` (	Deprecated. Use the %GETMODELPROPERTIES macro instead. For information about the %GETMODELPROPERTIES macro, see “Model Macros” on page 57.

Method	**Description**
`getModelMembers`	Deprecated. Use the %GETMODELMEMBERS macro instead. For information about the %GETMODELMEMBERS macro, see “Model Macros” on page 57.
void `postAdjustments` (java.lang.String modelCode)	Post adjustments for all time periods and analysis members of the specified model. Calling this method has the same effect as selecting **Post Adjustments** for a model in SAS Financial Management Studio. The postAdjustments method is useful for situations such as the following: ■ You need to post adjustments for a model with many adjustment rules. ■ You need to post adjustments for multiple models at a time. ■ You need to post adjustments for many time periods and analyses at a time. Parameters ■ modelCode: the model to be affected by the adjustments. Example: `j.callVoidMethod('postAdjustments', 'MyModel');` To confirm the completion of batch posting, check the posting status in SAS Financial Management Studio. In addition, if the posting is successful, the postAdjustments method logs an information message that begins “Posting adjustments completed successfully for model *model-name* ....” Throws: FinanceClientException
void `postAdjustments` (java.lang.String modelCode, java.lang.String startTimeCode, java.lang.String endTimeCode, java.lang.String[] analysisCodes)	Post adjustments for the specified model, time spans, and analysis members. Parameters: ■ modelCode: the code for the model to be affected. ■ startTimeCode: the dimension member code for the start time period. ■ endTimeCode: the dimension member code for the end time period. ■ analysisCodes: array of dimension member codes for the analysis members to be affected. Example: `array j[2] $9 ("MyActual", "MyBudget");` `j.callVoidMethod('postAdjustments','MyModel', 'Jan 2009','Feb 2009',j);` Throws: FinanceClientException
void `setCode` (java.lang.String code)	Select the model to be used by the updateTimeDefaultMembers method. Parameters: ■ code: the code for the model.

Method	Description

void `updateTimeDefaultMembers` (java.lang.String timeDefaultReadMemberCode, java.lang.String timeDefaultWriteMemberCode)	Update the default read and write members for the time hierarchy for the selected model.
	
	Before calling this method, you must first call the model's setCode method.
	
	Parameters:
	■ timeDefaultReadMemberCode: the member code for the default read member. This member must exist within the model's start and end time periods.
	■ timeDefaultWriteMemberCode: the member code for the default write member. This member must be the same as the default read member or a descendant of the default read member.
	
	Example:
	`j.callVoidMethod("setCode", "Default_Model");`
	`j.callVoidMethod("updateTimeDefaultMembers",`
	`    "Q1 2012",  "Jan 2012");`

Note: The constructor that specifies user ID and password is no longer available. Call the METADATA_PASSID function for authentication instead.

The following methods are inherited from class com.sas.solutions.finance.api.BaseApi:

- authenticate
- buildExceptionMessageString
- getErrorMessage
- getMessage
- setEnvironment
- setLocale
- trim

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

Model Macros

Overview

The Model macros are intended for use only by administrators and power users. Security is applied for the user who is running the query. Keep that in mind if you make the results available to other users.

Table 3.11 Summary of Model Macros

Macro	Description
%GETALLMODELS	Gets the available models of the specified model type.
%GETFORMS	Gets the forms in the specified form set.
%GETFORMSETS	Gets the form sets that use the specified model.
%GETMODELHIERARCHIES	Gets the hierarchies that are associated with the specified model.
%GETMODELMEMBERS	Gets the members for the specified model.
%GETMODELPROPERTIES	Gets the member properties for the specified model.

To use these macros, first invoke the %MODEL macro. (You do not need to declare a Model object or authenticate the user; those functions are handled in the macro code.)

The following is an example of the %MODEL macro:

```
%MODEL
%GETALLMODELS ('WORK', 'ModelList')
```

Note: In the Model macros, the TRUSTEDUSERNAME and TRUSTEDPASSWORD parameters are no longer supported.

The %GETALLMODELS Macro

Overview

The %GETALLMODELS macro retrieves the available models of the specified model type and creates a result set with the following columns: MODEL_CD, MODEL_NAME, and MODEL_DESCRIPTION.

Syntax

%GETALLMODELS (
sasLibName
, *outputDataSetName*
[, **ENVIRONMENT**="default"]
[, **LOCALE**="default"]
)

sasLibName
The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

outputDataSetName
The name of the result set.

ENVIRONMENT
An environment (such as “default”, “dev”, or “prod”) refers to an installation of SAS Financial Management. If you omit this parameter, “default” is used.

The environment value is site-specific. For more information, see “Specifying the SAS Financial Management Environment” on page 18.

LOCALE
A locale that is specified as *language-code_country-code*, such as `en_US` or `es_SP`. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. If you omit this parameter, the system default locale is used.

Example

The following example creates a result set, ModelsOut, that contains information about the models that are available to this user. The example uses the default values for the optional parameters.

```
%MODEL
%GETALLMODELS('Work', 'ModelsOut')
```

The %GETFORMS Macro

Overview

The %GETFORMS macro returns a data set with information about the forms in the specified form set. The data set contains the following columns: FORM_ID, FORM_NAME, FORM_DESCRIPTION, FORM_AUTHOR, FORM_REVIEWER, FORM_DUE_DATE, and FORM_STATE. The FORM_STATE column contains the status of the form, with a value such as DRAFT, READY, or EDITED.

Syntax

%GETFORMS (
  sasLibName
   , *outputDataSetName*
   , *modelCode*
   , *formSetName*
   [, **ENVIRONMENT**=""]
   [, **LOCALE**="default"]
)

sasLibName
: The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

outputDataSetName
: The name of the result set.

modelCode
: The code for a model that is associated with the form set. The model code is used only to retrieve the correct cycle. If the form set uses more than one model, select any one of the models.

formSetName
: The name of the form set.

ENVIRONMENT
: An environment (such as “default”, “dev”, or “prod”) refers to an installation of SAS Financial Management. If you omit this parameter, then “default” is used.

  The environment value is site-specific. For more information, see “Specifying the SAS Financial Management Environment” on page 18.

LOCALE
: A locale that is specified as *language-code_country-code*, such as `en_US` or `es_SP`. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. If you omit this parameter, the system default locale is used.

Example

The following example creates a result set, FormsOut, that contains information about the forms that are part of the MyFormSet form set.

```
%MODEL
%GETFORMS('Work', 'FormsOut', 'MyModel', 'MyFormSet')
```

The %GETFORMSETS Macro

Overview

The %GETFORMSETS macro returns a data set with information about the form sets that use the specified model. The data set contains the following columns: FORMSET_ID, FORMSET_NAME, FORMSET_DESCRIPTION, and FORMSET_STATUS.

The FORMSET_STATUS column can have one of the following values:

- `0`: Processing
- `1`: Draft
- `2`: Published
- `4`: Complete

Syntax

%GETFORMSETS (
    sasLibName
    , *outputDataSetName*
    , *modelCode*
    [, **ENVIRONMENT**=""]
    [, **LOCALE**="default"]
)

sasLibName
The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

outputDataSetName
The name of the result set.

modelCode
The code for the associated model.

ENVIRONMENT
An environment (such as “default”, “dev”, or “prod”) refers to an installation of SAS Financial Management. If you omit this parameter, then “default” is used.

The environment value is site-specific. For more information, see “Specifying the SAS Financial Management Environment” on page 18.

LOCALE
A locale that is specified as *language-code_country-code*, such as `en_US` or `es_SP`. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. If you omit this parameter, the system default locale is used.

Example

The following example creates a result set, FormsetsOut, that contains information about the form sets that use MyModel.

```
%MODEL
%GETFORMSETS('Work','FormsetsOut', 'MyModel')
```

The %GETMODELHIERARCHIES Macro

Overview

The %GETMODELHIERARCHIES macro creates a result set with information about the hierarchies that are associated with the specified model. The result set contains these columns: DIMENSION_TYPE_CD, DIMENSION_CD, DIMENSION_NAME, DIMENSION_DESCRIPTION, HIERARCHY_CD, HIERARCHY_NAME, and HIERARCHY_DESCRIPTION.

Syntax

%GETMODELHIERARCHIES (

sasLibName

, *outputDataSetName*

, *modelCode*

[, **ENVIRONMENT**=""]

[, **LOCALE**="default"]

)

sasLibName
The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

outputDataSetName
The name of the result set.

modelCode
The code for the associated model.

ENVIRONMENT
An environment (such as “default”, “dev”, or “prod”) refers to an installation of SAS Financial Management. If you omit this parameter, then “default” is used.

The environment value is site-specific. For more information, see “Specifying the SAS Financial Management Environment” on page 18.

LOCALE
A locale that is specified as *language-code_country-code*, such as **en_US** or **es_SP**. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. If you omit this parameter, the system default locale is used.

Example

The following example creates a result set, HierarchiesOut, that contains information about the hierarchies that are part of MyModel.

```
%MODEL
%GETMODELHIERARCHIES ('Work', 'HierarchiesOut', 'MyModel')
```

The %GETMODELMEMBERS Macro

Overview

The %GETMODELMEMBERS macro retrieves a model's members and creates a result set with these columns: DIMENSION_TYPE_CD, HIERARCHY_CD, MEMBER_CD, MEMBER_NAME, MEMBER_DESCRIPTION, HIERARCHY_LEVEL, HIERARCHY_ORDER, PARENT_CD, and IS_LEAF. A value of `1` for IS_LEAF signifies a leaf member. Otherwise, the value is `0`.

Syntax

%GETMODELMEMBERS (
sasLibName
, *outputDataSetName*
, *modelCode*
[, **ENVIRONMENT**=""]
[, **LOCALE**="default"]
)

sasLibName
The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

outputDataSetName
The name of the result set.

modelCode
The code for the associated model.

ENVIRONMENT
An environment (such as “default”, “dev”, or “prod”) refers to an installation of SAS Financial Management. If you omit this parameter, then “default” is used.

The environment value is site-specific. For more information, see “Specifying the SAS Financial Management Environment” on page 18.

LOCALE
A locale that is specified as *language-code_country-code*, such as **`en_US`** or **`es_SP`**. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. If you omit this parameter, the system default locale is used.

Example

The following example creates a result set with information about the members that are part of MyModel:

```
%MODEL
%GETMODELMEMBERS ('Work', 'ModelMembersOut', 'MyModel')
```

The %GETMODELPROPERTIES Macro

Overview

The %GETMODELPROPERTIES macro retrieves the member properties for the specified model and creates a result set with the following columns: DIMENSION_TYPE_CD, HIERARCHY_CD, MEMBER_CD, PROPERTY_CD, PROPERTY_NAME, and PROPERTY_VALUE. You can limit the results by specifying dimension type codes, property codes, and member codes as parameters.

Note: The version of this macro that is in fmmodel.sas has been deprecated. Use this version instead.

Syntax

%GETMODELPROPERTIES (
    sasLibName
     , *outputDataSetName*
     , *modelCode*
     [, **ENVIRONMENT**=""]
     [, **DIMTYPECODES**=""]
     [, **PROPERTYCODES**=""]
     [, **MEMBERCODES**=""]
     [, **LOCALE**="default"]
     [, **DELIM**=';']
)

sasLibName
The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

outputDataSetName
The name of the result set.

modelCode
The code for the associated model.

ENVIRONMENT
An environment (such as “default”, “dev”, or “prod”) refers to an installation of SAS Financial Management. If you omit this parameter, then “default” is used.

The environment value is site-specific. For more information, see “Specifying the SAS Financial Management Environment” on page 18.

LOCALE
A locale that is specified as *language-code_country-code*, such as `en_US` or `es_SP`. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. If you omit this parameter, the system default locale is used.

DIMTYPECODES
A delimited list of unquoted dimension type codes to use in the query. By default, all dimension type codes are used.

PROPERTYCODES
A delimited list of unquoted property codes to use in the query. By default, all properties are used.

MEMBERCODES
A delimited list of unquoted member codes to use in the query. By default, all members are used.

DELIM
The delimiter to be used to parse the input (dimension type codes, property codes, and member codes). By default, the macro expects a semicolon.

Example

This example retrieves only the AccountType and AccountBehavior properties for the ACCOUNT dimension type:

```
%MODEL
%GETMODELPROPERTIES('Work', 'PropertiesOut', 'MyModel',
DIMTYPECODES='ACCOUNT', PROPERTYCODES='AccountType;AccountBehavior')
```

Executing Queries with the %FMQUERY Macro

Overview

The %FMQUERY macro executes a query against a model. The macro returns the same result as a query in Excel, including calculated members. Use this macro instead of the executeQuery method of the Model class.

Note: This macro applies to composite models as well as non-composite models.

The %FMQUERY macro is intended to be used only by administrators and power users. Security is applied to a query for the user who is running the query. (Keep that in mind if you make the query results available to other users.)

Note: The RUNASUSERID, TRUSTEDUSERNAME, and TRUSTEDPASSWORD parameters are no longer supported.

Query Types

The %FMQUERY macro supports two types of queries:

- **MDX queries**—queries that use MDX syntax, which is similar to SQL syntax.
- **Non-MDX queries**—queries that are based on a model code and a data set that contains query parameters.

Syntax

%FMQUERY (
    localSasLibName
      , *resultDataSetName*
      [, **MDXSTRING**=""]
      [, **MODELCODE**=""]
      [, **SASLIBNAME**=""]
      [, **QUERYDATASETNAME**=""]
      [, **FILTEROPTS**=0]
      [, **MEMBEROPTS**=0]
      [, **ENVIRONMENT**="default"]
)

localSasLibName
: The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

resultDataSetName
: The name of the result set to be produced by the query. It contains the following columns:

DIMENSION_TYPE_CD: the member code for each dimension type. The calling routine must handle illegal characters in the member codes.

VALUE: the corresponding value. `NaN` is represented as a period (.).

Records are filtered according to the *filterOptions*.

If the query fails and the result data set already exists, the data set is not deleted.

MDXSTRING
: The query to be executed. This parameter is required if you are performing an MDX-style query.

Note: For non-MDX-style queries, use the QUERYDATASETNAME parameter instead.

MODELCODE
: The identifier for the results model to be used in the query.

Note: This parameter is not used for MDX-style queries.

SASLIBNAME
: The libref for the SAS library that holds the query data set. This library must be registered in the metadata repository.

QUERYDATASETNAME
: The name of the SAS table that contains the query. This table must exist before you call %FMQUERY. For details, see "The Query Data Set" on page 66.

This parameter is not used for MDX-style queries. Instead, use the MDXSTRING parameter.

FILTEROPTS
: A value that specifies filters to be applied to the result set. Valid options are:

`0`: include all crossings (default)

`1`: exclude missing values

`2`: exclude zero values

`3`: exclude missing and zero values

MEMBEROPTS
: Additional member attributes, including hierarchical ordering, to be printed beside the member codes. The parameter can have any combination of these values:

`0`: include only member code (_CD) columns.

`1`: include member name (_NAME) columns.

`2`: include member description (_DESC) columns.

`4`: include member hierarchy sort (_SORT) columns. The sort values are represented as hierarchical child numbering of the member starting from the root of the hierarchy (such as `1.4.2.5`).

Regardless of other options, the member code column is always printed. The options can be used in any combination. For example, a value of `5` includes the member code (always), the member name, and the member hierarchy sort columns.

ENVIRONMENT

An environment (such as `default`, `dev`, or `prod`) refers to an installation of SAS Financial Management.

The environment value is site-specific. For more information, see "Specifying the SAS Financial Management Environment" on page 18.

If you omit a dimension type in your query, the default member for the model's hierarchy in that dimension (at the model's hierarchy as-of date) is used instead. For the Time dimension, this value is not necessarily the same as the default read member for the time hierarchy that can be set in the model. To avoid unexpected results, we recommend that you always include the Time dimension in your specification. This applies to both MDX and non-MDX queries.

The Query Data Set

For non-MDX queries, one parameter of the %FMQUERY macro is QUERYDATASETNAME, the name of a table that contains the query. This table must exist before you call the macro, and it must reside in the same library as the result set that is produced by the query.

Note: This parameter is not used for MDX queries.

The table has the following columns:

Table 3.12 Contents of the Query Data Set

Column	Description	Data Type
DIMENSION_TYPE_CD	Dimension type code	character
MEMBER_CD	Member code. The dimension type code and member code pair define the root of the subtree to be queried.	character
INCLUDE_MEMBER	`0`: exclude the member `1`: include the member	numeric
INCLUDE_LEAVES	`0`: exclude leaves `1`: include first-level leaves `2`: include all levels of leaves `3`: include first-level leaves and virtual children `4`: include all levels of leaves and virtual children	numeric
INCLUDE_ROLLUPS	`0`: exclude roll-ups `1`: include first-level roll-ups `2`: include all levels of roll-ups	numeric

%FMQUERY Example (Non-MDX)

The following example executes a query against a fictitious model that is named TESTING18_MODEL. The query data set name is QUERYPARAMETERS. In this example, the results are written to the NONMDXRESULTDATASETNAME data set in the WORK library.

Example Code 3.6 *Non-MDX Query*

```
LIBNAME stagefm BASE "C:\SAS\Config\Lev1\SASApp\Data\FinancialManagement\StageFM";

data stagefm.queryParameters;
   length DIMENSION_TYPE_CD MEMBER_CD $32;
   DIMENSION_TYPE_CD = "ACCOUNT"; MEMBER_CD = "A8420"; INCLUDE_MEMBER=1;
   INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
   DIMENSION_TYPE_CD = "TIME"; MEMBER_CD = "DEC1997"; INCLUDE_MEMBER=1;
   INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
   DIMENSION_TYPE_CD = "CURRENCY"; MEMBER_CD = "USD"; INCLUDE_MEMBER=1;
   INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
run;
%fmquery(modelCode="testing18_model",localSasLibName="Work", sasLibName="stagefm",
   queryDataSetName="queryParameters", resultDataSetName="NONMDXResultDataSetName",
      environment="default")
```

%FMQUERY Example with MDX String

The following is an example of calling %FMQUERY using an MDX string:

Example Code 3.7 *Query Using an MDX String*

```
%fmquery("Work", "MDXResultDataSetName",
   mdxString="SELECT {ACCOUNT.A8420} on 0 FROM testing18_model WHERE (TIME.DEC1997, CURRENCY.USD)",
  environment="default")
```

Note: The mdxString cannot include a line break.

For one approach to creating an MDX string, see "Copying an MDX String" on page 67.

For MDX reference information, see "MDX Reference for SAS Financial Management" on page 68.

Note: Currently, MDX queries in SAS Financial Management do not support the equivalent of the INCLUDE_MEMBER,INCLUDE_LEAVES, or INCLUDE_ROLLUPS options (that are available in non-MDX queries). In an MDX query, you must specify each member separately. To include leaves for one or more dimensions, specify those leaf members in the MDX string.

Copying an MDX String

To create an MDX string, one simple approach is to save the string that is created when you insert a read-only table in Microsoft Excel. Follow these steps:

1 In Microsoft Excel, log on to the middle tier.

2 Insert a read-only table.

3 Open the table properties.

4 Select the **Dimensions** tab.

5 Click **Query Diagnostics**.

6 Click **Copy ODCS MDX String to Clipboard**.

The MDX string for the read-only table is available on the Windows clipboard.

MDX Reference for SAS Financial Management

Overview

SAS Financial Management uses ODCS to support simple MDX queries that extend the capabilities that are available with the standard query parameters.

Previously, complex queries required exploding the cube or running multiple, smaller queries. By stacking multiple dimensions on an axis, MDX allows clients to express the specific query that they need.

Only a subset of MDX functionality is currently supported in ODCS:

- basic queries: `SELECT ... FROM ... WHERE ...`
- basic member functions

More sophisticated features are not currently supported. For example, these features are not currently supported:

- creating or manipulating metadata
- defining calculated members
- more advanced functions, such as filter, aggregate, and non-empty
- anything that is defined on a WITH clause

Members

A member is represented as *`DimensionTypeCode.MemberCode`*. For example:

- `CURRENCY.USD`
- `TIME.Jan2001`
- `INTORG.Legal`

Note: Standard MDX and OLAP do not have the concept of dimension types. Instead, they use dimension codes to define members. ODCS uses dimension types, because they make it easier to reuse queries between virtual cubes (vcubes). In this MDX reference, references to dimensions and dimension types are interchangeable.

All codes in ODCS are case sensitive. If a dimension type code or member code includes a non-alphanumeric character, the code must be wrapped in square brackets, as in these examples:

- `INTORG.[R&D]`
- `ANALYSIS.[My Analysis]`
- `PRODUCT.[Hershey's Kisses]`
- `[`*`CUSTOM TYPE`*`].[My Member]`

A member function can be appended to a member using the following syntax: *`DimensionTypeCode.MemberCode.Function`*

An example is the VC function, a SAS Financial Management function that returns the virtual child of the member:

- `INTORG.Legal.VC`
- `PRODUCT.[Hershey's Kisses].VC`

(In MDX, the virtual child is known as a DataMember.)

Tuples

A *tuple* is a combination of members from one or more dimensions, with only one member from each dimension. You can think of a tuple as a multidimensional member. The simplest example of a tuple has one member, such as `INTORG.Legal`.

When there are multiple members on a tuple, the members are separated by commas and the entire tuple is wrapped in parentheses, as in these examples:

- `(INTORG.Legal, TIME.Jan2001)`
- `(INTORG.Legal, TIME.Jan2001, PRODUCT.[Hershey's Kisses])`
- `(INTORG.Legal, TIME.Jan2001, PRODUCT.[Hershey's Kisses], CURRENCY.USD, ANALYSIS.Actuals)`

It is important to remember that tuples can have only one member from each dimension. The following tuples are invalid because they have multiple members from the same dimension:

- `(INTORG.Legal, TIME.Jan2001, TIME.Feb2001)`

  Invalid: two members from the TIME dimension.
- `(INTORG.Legal, TIME.Jan2001, INTORG.[R&D])`

  Invalid: two members from the INTORG dimension.

Tuple Sets: { }s

A *tuple set* is an ordered collection of tuples. A tuple set can have one tuple, multiple tuples, or even zero tuples. Within a set, tuples can be repeated.

Note: This definition differs from the mathematical definition of a set or the Set data structures in Java.

The tuples in a set can have one or more members. A set is wrapped in braces, and the tuples are separated by commas. Here are some examples:

- `{ INTORG.Legal, INTORG.[R&D] }`

  Set with two tuples, each containing one member.
- `{ (INTORG.Legal, TIME.Jan2001) }`

  Set with one tuple (wrapped in parentheses), containing two members.
- `{ (INTORG.Legal, TIME.Jan2001), (INTORG.[R&D], TIME.Feb2001) }`

  Set with two tuples, each tuple containing two members.
- `{ (INTORG.Legal, TIME.Jan2001, ANALYSIS.Actuals), (INTORG.[R&D], TIME.Feb2001, ANALYSIS.Budget) }`

  Set with two tuples, each tuple containing three members.
- `{ (INTORG.Legal, TIME.Jan2001), (INTORG.[R&D], TIME.Feb2001), (INTORG.[R&D], TIME.Feb2001) }`

Set with three tuples, each tuple containing two members. One tuple is repeated.

All tuples in a set must have the same dimensions, and the dimensions must be in the same order. This is called the dimensionality of the tuple. Notice that all of the examples above meet this requirement. The last example has three tuples. Each tuple has two members. All three tuples contain the same dimensions and specify the INTORG dimension first and the TIME dimension second. Thus, they have the same dimensionality.

The following sets are invalid because they do not have the same dimensionality:

- `{ (INTORG.Legal, TIME.Jan2001), (INTORG.[R&D], ANALYSIS.Budget) }`

  Invalid: TIME and ANALYSIS are different dimensions.

- `{ (INTORG.Legal, TIME.Jan2001), (TIME.Feb2001, INTORG.[R&D]) }`

  Invalid: tuple dimensions are not in the same order.

- `{ (INTORG.Legal), (ANALYSIS.Actuals) }`

  Invalid: INTORG and ANALYSIS are different dimensions.

- `{ INTORG.Legal, ANALYSIS.Actuals }`

  Invalid: INTORG and ANALYSIS are different dimensions.

  This example might look like a single tuple with two members. However, it is actually a tuple set with two tuples, each containing one member (using the convention of omitting parentheses for a tuple with a single member). Because the members are from different dimensions, the tuple set is invalid.

Basic Query Syntax

The MDX query syntax enables you to define the view of the data that you want returned. Syntactically, it is similar to an SQL query. The basic syntax of a SELECT clause is as follows:

This simple query retrieves data with TIME members on the columns and INTORG members on the rows:

-
  ```
  SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001,
  TIME.Q12001} ON COLUMNS, {INTORG.Legal, INTORG.[R&D]} ON
  ROWS FROM [My VCube]
  ```

Note: The example queries in this chapter contain line breaks only so that they fit on the page. In the %FMQUERY macro, MDX query strings cannot contain a line break. In addition, keywords are shown in uppercase. However, MDX queries are not case sensitive.

The results would resemble the following:

	TIME.Jan2001	TIME.Feb2001	TIME.Mar2001	TIME.Q12001
INTORG.Legal	2	6	10	18
INTORG.[R&D]	10	40	20	70

The SELECT clause defines one or more axes. Each axis is assigned a position on the table (columns or rows). The example above defines two axes: TIME on columns and INTORG on rows. Notice the braces in the row axis definition, denoting a tuple set. Each tuple in the set contains only one member. However, like any tuple set, it can contain multiple members. This feature enables you to stack multiple dimensions on an axis, and mixing and matching members between dimensions.

The following example crosses the INTORG members with different ANALYSIS members on the rows:

- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { (INTORG.Legal, ANALYSIS.Actuals), (INTORG.[R&D], ANALYSIS.Budget) } ON ROWS FROM [My VCube]`

The results would resemble the following:

	TIME.Jan2001	TIME.Feb2001	TIME.Mar2001	TIME.Q12001
INTORG.Legal ANALYSIS.Actuals	2	6	10	18
INTORG.[R&D] ANALYSIS.Budget	20	60	15	95

WHERE Clause: Defining a Slicer

The previous examples use only two or three dimensions in the queries. For any dimensions in the cube that were not specified (such as CURRENCY, PRODUCT, or ACCOUNT), the default member for the dimension is implicitly used in the query.

If you want to cross your table with members that are not default members, in MDX you can use a WHERE clause to define members that apply to the entire table. This clause is known as a slicer. The example below defines a slicer for three dimensions that are not shown on the table:

- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, {INTORG.Legal, INTORG.[R&D]} ON ROWS FROM [My VCube] WHERE (CURRENCY.USD, ANALYSIS.Budget, FREQUENCY.PA)`

The results would resemble the following:

Slicer: CURRENCY.USD, ANALYSIS.Budget, FREQUENCY.PA				
	TIME.Jan2001	TIME.Feb2001	TIME.Mar2001	TIME.Q12001
INTORG.Legal	4	8	12	24
INTORG.[R&D]	20	60	15	95

Notice that the slicer in the WHERE clause is enclosed in parentheses: it is really just a tuple. Like any tuple, it can contain one or more members, and the members must be from different dimensions. In addition, the slicer in the tuple cannot contain a member from a dimension that is used in one of the axes. The following example is invalid because it uses the TIME dimension on both the rows and the slicer:

- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, {INTORG.Legal, INTORG.[R&D]} ON ROWS FROM [My VCube] WHERE (TIME.Apr2001, ANALYSIS.Budget)`

SELECT Clause: Defining Axes

So far, all the query examples have used only two axes: columns and rows. However, an MDX query can have anywhere from 0–64 axes. Beyond COLUMNS and ROWS, the axis keywords are PAGES, CHAPTERS, and SECTIONS.

The following is a list of examples of queries that use a different number of axes:

- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS, {INTORG.Legal} ON ROWS, {ANALYSIS.Actuals, ANALYSIS.Budget} ON PAGES FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS, {INTORG.Legal} ON ROWS, {ANALYSIS.Actuals, ANALYSIS.Budget} ON PAGES, {FREQUENCY.PTD} ON CHAPTERS FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS, {INTORG.Legal} ON ROWS, {ANALYSIS.Actuals, ANALYSIS.Budget} ON PAGES, {FREQUENCY.PTD} ON CHAPTERS, {PRODUCT.Widgets, PRODUCT.Gadgets} ON SECTIONS FROM [My VCube] WHERE (CURRENCY.USD)`

Instead of using the axis keywords such as COLUMNS or PAGES, you can refer to axes by numbers, beginning with 0. When using number, 0=COLUMNS, 1=ROWS, 2=PAGES, 3=CHAPTERS, and 4=SECTION). Beyond sections, you must use numbers. The following queries are the same as the examples above, except that they use axis numbers instead of keywords:

- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0 FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0, {INTORG.Legal} ON 1, {ANALYSIS.Actuals, ANALYSIS.Budget} ON 2 FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0, {INTORG.Legal} ON 1, {ANALYSIS.Actuals, ANALYSIS.Budget} ON 2, {FREQUENCY.PTD} ON 3 FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0, {INTORG.Legal} ON 1, {ANALYSIS.Actuals, ANALYSIS.Budget} ON 2, {FREQUENCY.PTD} ON 3, {PRODUCT.Widgets, PRODUCT.Gadgets} ON 4 FROM [My VCube] WHERE (CURRENCY.USD)`

Note: You cannot skip axis definitions. For example, you cannot specify `0` and `2` and omit `1`.

Specifying Excluded Members

In an ODCS query, you can specify excluded members (members on an axis that should be ignored while running a query). Because there is no equivalent concept in MDX, ODCS supports an MDX extension for using this functionality in SAS Financial Management. At the end of a query, you can add an EXCLUDE clause to specify the members to be excluded from the query. For each dimension from which you want to exclude members, the EXCLUDE clause contains a tuple set separated by commas, as in these examples:

- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { INTORG.All } ON ROWS FROM [My VCube] EXCLUDE { INTORG.Legal }`
- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { INTORG.All } ON ROWS FROM [My VCube] WHERE ( CURRENCY.USD ) EXCLUDE { INTORG.Legal, INTORG.[R&D] }`
- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { INTORG.All } ON ROWS, { PRODUCT.All } ON PAGES FROM [My VCube] WHERE ( CURRENCY.USD ) EXCLUDE { INTORG.Legal, INTORG.[R&D] }, { PRODUCT.Widgets }`

Note: Each set corresponds to a dimension in the query, and each tuple in the set contains only one member.

Supported Member Functions

ODCS supports the following functions:

`.VC`
Uses the virtual child of the member. For example:

- `INTORG.Legal.VC`
- `INTORG.[R&D].VC`

`.DataMember`
MDX term for the ODCS term “virtual child.” This function is interchangeable with the .VC function. For example: `INTORG.Legal.DataMember`

`.Ignore`
Placeholder member that is never calculated. This function is used by Excel to overlay client-side calculations after the MDX table is returned. Only the dimension type code must be valid; the member code is ignored by the server. Here is an example: `PRODUCT.MyClientSideCalc.Ignore`

ODCS versus Standard OLAP

The ODCS architecture differs from standard OLAP in a few ways. These following differences affect MDX usage and syntax support:

- ODCS supports only a single, numeric measure. Therefore, there is never a need to use the MEASURES keyword in a query.
- Levels are not supported explicitly in ODCS, except for certain dimensions such as TIME. Currently, there is no support for referencing Levels in the query syntax.

- In ODCS, members in the same dimension must have a unique code. Because a cube has only one dimension for each dimension type, a member code is always unique in a given dimension type at query time.

  This requirement provides the shortcut when defining member definitions of `DimensionTypeCode.MemberCode`, such as `TIME.Jan05`. If ODCS supported non-unique member codes in a dimension, you would need to follow the MDX standard and specify the ancestors of the member, such as `TIME.2005.Q1.Jan`.

4

Customizing a Workflow

Overview

This chapter describes the following topics:

- customizing a workflow
- workflow types
- adding custom code to a workflow
- data validation example

About Customizing a Workflow

In SAS Financial Management, a *workflow* defines the review and approval process used in budgeting, forecasting, and other planning activities. Each workflow consists of a collection of states (such as READY, EDITED, and COMPLETE) and actions (such as EDIT and SUBMIT). At run time, the actions advance the workflow from one state to the next. Each action triggers a corresponding policy file. The policy file is code that is associated with the actions.

You can customize a workflow by writing a stored process that executes before or after the workflow is advanced. This chapter explains how to add your custom code to a workflow. It also contains a short example of a workflow stored process.

Workflow Types

Overview

SAS Financial Management supports two types of workflows: top-down and bottom-up.

Note: For more information about SAS Financial Management workflows, see the *SAS Financial Management: User's Guide* or the online Help for the SAS Financial Management Add-In for Microsoft Excel.

Top-Down Workflow

A *top-down* workflow enables users at any roll-up point to make bulk updates and adjustments down and across multiple entities and dimensions.

A data-entry project that has a top-down workflow begins when a top-down form set is published from SAS Financial Management Studio. The workflow ends when a Finance Process Administrator applies the COMPLETE action to the form set in SAS Financial Management Studio.

Bottom-Up Workflow

In a *bottom-up* workflow, forms begin at the lower levels of the hierarchy and are aggregated and reviewed by the organization as they move up an approval hierarchy. A bottom-up workflow can be connected to a separate reviewer workflow that supports additional reviewers in the budget approval process.

A bottom-up workflow begins when a bottom-up form set is published from SAS Financial Management Studio. The workflow ends when a Finance Process Administrator applies the COMPLETE action to the form set in SAS Financial Management Studio.

Adding Your Custom Code to a Workflow

The Pre and Post Classes

Two Java classes (Pre and Post) form the bridges between the SAS Financial Management workflow system and the SAS stored processes in which the customized code is deployed.

Whenever a policy file is triggered, the Pre.invoke method is called before the policy file is executed. The Post.invoke method is called after the policy file is executed. These methods call a stored process if one is linked to this part of the workflow.

If the stored process fails (due to exception or error in the customized codes), the workflow does not advance to the next state.

- If the Pre operation fails, the policy file is not executed.
- If the Post operation fails, the workflow is rolled back to its previous state.

However, if the stored process itself makes any changes, such as updating the database, those changes remain.

Steps in Customizing a Workflow

Do not modify the Pre and Post classes directly. To customize the workflow, complete the following steps:

1. Write a SAS stored process to perform the necessary business logic.

   When writing the stored process, note the following:

   - The stored process must set the FM_SP_RESULT environment variable. If the operation fails, the program should set FM_SP_RESULT to `INVALID` and set the FM_SP_MESSAGE environment variable to an appropriate text message. Otherwise, the stored process should set FM_SP_RESULT to `VALID`.
   - If the value of FM_SP_RESULT is `INVALID`, an exception is thrown and the workflow is not advanced to the next state. You can make the value of the FM_SP_MESSAGE environment variable available in the stored process log.

   For information about writing a stored process, see the *SAS Stored Processes: Developer's Guide*. For an example stored process for workflow customization, see "Data Validation Example" on page 80.

2. On the data tier, save the stored process in a directory such as ***SAS-config-dir*\Lev1\SASApp\SASEnvironment\FinancialManagement \SASCode\UserDefined**. (Create the `UserDefined` directory if it does not already exist.)

3. Log on to SAS Management Console as an administrator and register the stored process in the `/Products/SAS Financial Management/ Customized workflow` folder. (You might need to create this folder.)

4. Create a resource file that links a workflow action to the stored process. If the resource file already exists, update the file with information about the new stored process. For information about resource files, see "The Resource File" on page 77.

The Resource File

Update the Resource File

The resource file is an XML file that provides the location of a stored process and associates it with a specific form set and an action. The following is a template for a resource file:

```
<SASWorkflowCustomizations>
  <Application name="SAS Financial Management">
    <Object type="FormSet" name="form_set_ID">
```

```
      <Action type="action_type">
        <Execute type="execute_type"
          storedProcessFullPath="path_to_stp"/>
      </Action>
    </Object>
  </Application>
</SASWorkflowCustomizations>
```

Replace the italicized strings with the appropriate values:

- *execute_type* specifies when the stored process is called, relative to execution of the policy file. It must have a value of **`pre`** or **`post`**.
- *action_type* is an action such as SUBMIT or REJECT.

  For a list of available action types, see Table 4.1 on page 78. Notice that some actions are available only in a top-down workflow or only in a bottom-up workflow.
- *path_to_stp* is the path to the stored process metadata definition, such as **`/Products/SAS Financial Management/Customized workflow/MyCustomWorkflow`**.

  You can link the same stored process to more than one form set or action: just create a separate <Object> entry for each form set, action type, and execute type combination.
- *form_set_ID* is the ID of the form set to which the action applies. To look up a form set ID in the SASSDM database, you can use the following SQL query:

  ```
  "select form_set_id from public.sas_form_set where form_set_nm='form-set-name'"
  ```

Here is an example:

Example Code 4.1 *Example Resource File*

```
<SASWorkflowCustomizations>
  <Application name="SAS Financial Management">
    <Object type="FormSet" name="2">
      <Action type="SUBMIT">
        <Execute type="pre" storedProcessFullPath= _
         "/Products/SAS Financial Management/Customized workflow/MyCustomWorkflow"/>
      </Action>
    </Object>
  </Application>
</SASWorkflowCustomizations>
```

Note: Line breaks ("_") added for readability.

Table 4.1 *Available Workflow Actions*

Action Type	Top-down Workflow	Bottom-up Workflow	Description
SUBMIT		√	Submits a form for approval.
EDIT			Opens a form for editing.
REVIEW		√	Opens a form in Read-Only mode so that it can be reviewed.

Action Type	Top-down Workflow	Bottom-up Workflow	Description
REJECT		√	Changes the form's state to REJECTED and notifies the user who submitted the form.
APPROVE		√	Approves a form and copies that form's data to its parent form.
RECALL	√	√	Recalls a form so that it can be further edited and then pushed again or resubmitted.
PUSH	√		Makes a form available to the users who are responsible for the top member's children. The amounts that have been allocated to the children of that member are copied to the forms for those child members. As a result, the users who are responsible for the child members to edit their forms, allocate the pushed amounts to the next level of child members. Then, the users push their forms.
PUSHTOALL	√		Makes a form available to the users who are responsible for all the top member's descendants. The amounts that have been allocated to the descendants of that member are copied to the forms for those descendant members. As a result, the users who are responsible for the descendant members to edit their forms. However, their editing is limited to redistributing amounts within their target member. No other user can push amounts to the next level of child members because PUSHTOALL cascades all the way down the target hierarchy in a single step.
COMPLETE	√	√	Ends the workflow. This action can be performed only by a Finance Process Administrator.
UNCOMPLETE	√	√	Reactivates a form for further work. This action can be performed only by a Finance Process Administrator.

When working with a resource file, note the following:

- You can associate as many actions with a form set as necessary.
- You can associate only one stored process with each action.
- You can associate the same stored process with multiple actions in multiple form sets, if applicable.
- Name the file `WorkflowCustomizations.xml` and save it on the middle tier, where the web application server resides. A good location is the following directory: ***`SAS-config-dir`***`\Lev1\CustomAppData\FMCustomizedWorkflow`

Set the JVM Options

To make the resource file available, add the following option to the JVM options for SASServer3 (the managed server to which SAS Financial Management is deployed).

```
-Dsas.workflow.customizations="file:///path-to-resource-file"
```

For example:

- on Windows:

  ```
  -Dsas.workflow.customizations=file:///C:/SAS/Config/Lev1/CustomAppData/
  FMCustomizedWorkflow/WorkflowCustomizations.xml
  ```

- on UNIX:

  ```
  -Dsas.workflow.customizations=file:////install/cfgsas94/SASConfig/Lev1/CustomAppData/
  FMCustomizedWorkflow/WorkFlowCustomizations.xml
  ```

Note: Line breaks are added for readability only. For information about configuring your web application server, go to http://support.sas.com/resources/thirdpartysupport/v93/index.html.

The option applies when you restart the managed server.

Note: You do not need to restart the managed server when you make updates to the resource file.

Data Validation Example

About the Data Validation Example

This section contains an example of cell-based data validation that uses a stored process, an execute type of `pre`, and a SUBMIT action. At run time, when a user submits a form in the specified form set, the stored process is automatically triggered. It validates a cell value in the form. If the value is greater than `0`, the SUBMIT succeeds. Otherwise, the SUBMIT fails.

The example in this section makes the following assumptions:

- A form set with ID `123` has been created.
- A form template with a result model (called `tst_model`) has been saved. It includes the dimensions shown in the Dimension column of Table 4.2 on page 81.
- The form cell whose value is to be validated is defined by the crossing that is exemplified by the codes in the Member Code column of the following table.

The dimensionCodes and memberCodes arrays in the example contain the values from the first and second columns, respectively, of the following table. You do not need to include all the values in the table in the two arrays, but the values of the two arrays must match. During the query, any missing dimension code-member code pairs are filled with default values from the dimensions that are defined for the results model and the default read member that is defined in the hierarchy for each dimension.

Table 4.2 *Example Dimensions and Member Codes*

Dimension	Member Code
ACCOUNT_FM	6232
ANALYSIS_FM	BUDGET
Cost Center	Total
CURRENCY	EUR
fm_INTORG_CODE	WW_SA
TIME_FM	012002
fm_INTORG_CODE_TRADER	EXT
SOURCE	BaseForm
PRODUCT_FM	Jackets

The actual query is carried out in the following code:

```
model.callDoubleMethod("getCellValue", "tst_model", dimensionCodes,
    memberCodes, value);
```

Depending on the return value, the program sets the FM_SP_RESULT and FM_SP_MESSAGE environment variables. If the return value is less than or equal to 0, the program sets FM_SP_RESULT to **INVALID** and sets FM_SP_MESSAGE to a text message. Otherwise, the program sets FM_SP_RESULT to **VALID**.

This example uses methods from the SAS Financial Management Java API. For information about the SAS Financial Management Java API, see Chapter 3, “The SAS Financial Management Java API,” on page 15.

Code for the Example

This SAS program retrieves the data from the cell and validates the data.

Note: If you are declaring a Javaobj, the picklist option is required in the DATA step so that the Javaobj can find the necessary JAR files.

Example Code 4.2 *Stored Process for Workflow Customization*

```
data _null_ /picklist='finance/finance.txt' saveclassl='finbatch' getclassl='finbatch';
    put 'This is a data entry validation test';

    /* Read and echo environment variables passed in from the middle tier */
    /* form ID */

    length formId $20;
    formId = symgetc("fm_sp_form_id");
    put formId=;

    /* security key */
```

```
length secKey $200;
secKey = symgetc("fm_sp_seckey");
put secKey=;

/* action on the form */
length action $20;
action = symgetc("fm_sp_action");
put action=;

/* user ID */
length userId $20;
userId = symgetc("fm_sp_user_id");
put userId=;

/* user name */
length userName $60;
userName = symgetc("fm_sp_user_name");
put userName=;

dcl javaobj oMetadata("com/sas/solutions/finance/api/Metadata");
    oMetadata.ExceptionDescribe(1);

oMetadata.callVoidMethod("setEnvironment", "default");
call METADATA_PASSID("oMetadata", "");

/* Instantiate the Form class */
dcl javaobj form("com/sas/solutions/finance/api/Form",formId, trim(secKey));
form.ExceptionDescribe(1);

/* Call methods of the Form class and echo the results */
/* Get the target member code */
length targetMemberCode $50;
form.callStringMethod("getTargetMemberCode", targetMemberCode);
put targetMemberCode=;

/* Get the target dimension code */
length targetDimensionCode $50;
form.callStringMethod("getTargetDimensionCode", targetDimensionCode);
put targetDimensionCode=;

length cFormInfo $20000;
form.callStringMethod("getInfo",cFormInfo);
put cFormInfo=;

length authors $ 200;
form.callStringMethod("getAuthors", " ", authors);
put authors=;

length admins $30000;
form.callStringMethod("getPlanningAdministrators", " ", admins);
put admins=;

/* Instantiate the Model class */
dcl javaobj model("com/sas/solutions/finance/api/Model", trim(secKey));
/* Set up two arrays, dimensionCodes and memberCodes */
array dimensionCodes[9] $50
```

```
(
"",
"ANALYSIS",
"ACCOUNT",
"CURRENCY",
"TIME",
"TRADER",
"SOURCE",
"PRODUCT",
 "CUSTOMER"
);
/* Set target dimension code */
dimensionCodes[1] = targetDimensionCode;
array memberCodes[9] $30
(
"",
"BUDGET",
"R28000",
"USD",
"YR2MTH01",
"EXT",
"BaseForm",
"P01000",
  "C00001"
);
/* Set target member code */
memberCodes[1] = targetMemberCode;
/* Call getCellValue method */
length value 8;
model.callDoubleMethod("getCellValue", "Financial", dimensionCodes,
    memberCodes, value);
put value=;

/* Test for value <= 0 and set environment variables accordingly */
if value <= 0 then do;
    call symput("fm_sp_result", "INVALID");
    call symput("fm_sp_message", "Account R28000 of YR2MTH01 should be greater than 0.");
end;
else do;
    call symput("fm_sp_result", "VALID");
end;

form.delete();
model.delete();

run;
```

Registering the Stored Process

Register the stored process in SAS Management Console. For the example code, neither **Stream** nor **Package** is selected for the results, because the only output is to the log file. In other cases, you might want the stored process to generate streaming or package output.

For more information about registering a stored process, see the *SAS Stored Processes: Developer's Guide* and the online Help for SAS Management Console.

Updating the Resource File

The resource file (**`SAS-config-dir\Lev1\CustomAppData\Workflow\WorkflowCustomizations.xml`**) might include the following entry:

```
<SASWorkflowCustomizations>
  <Application name="SAS Financial Management">
    <Object type="FormSet" name="123">
      <Action type="SUBMIT">
        <Execute type="pre" storedProcessFullPath=
          "/Products/SAS Financial Management/Customized workflow/validation"/>
      </Action>
    </Object>
  </Application>
</SASWorkflowCustomizations>
```

In entry above, the execute type is set to “pre”, which means that the stored process is executed before the workflow policy file.

5

Creating a Custom Cell Action

Overview

This chapter explains how to create a custom cell action for use with a SAS Financial Management table in Microsoft Excel.

This chapter describes the following topics:

- custom cell actions
- writing a stored process
- registering a stored process
- updating the resource file
- selecting an action

About Custom Cell Actions

By default, when a user selects a data cell and clicks the right mouse button, the **Contributing Data** action is available. The Contributing Data action enables the user to view the data records that make up the selected cell.

You can add your own custom actions that invoke a stored process that displays its output in a browser window. For example, you might create a custom action that displays the transactions that make up the selected cell. Or, you might create a custom action to reconcile adjustments in consensus forecasting.

To create a custom action, complete the following tasks:

1 Write a stored process to run when the action is invoked.

   See "Write the Stored Process" on page 86.

2 In SAS Management Console, define the stored process metadata.

   See "Register the Stored Process" on page 90.

3 Define the custom action in a resource file.

   For the first custom action, you must create this file and set a JVM option that points to the resource file.

   See "Update the Resource File" on page 92.

4 The new action is available from a read-only table in Microsoft Excel. When a user right-clicks a cell and selects **Tools**, the new action appears as a selection.

   See "Select the Action" on page 94.

Write the Stored Process

About the Stored Process

Typically, you use the SAS Financial Management Java API to write a stored process. For information about the classes and methods that make up the SAS Financial Management Java API, as well as information about declaring a Javaobj object and authenticating the user, see "Authenticating the User" on page 18.

After writing the stored process, save the code on the data tier, in a location such as the ***SAS-config-dir*****\Lev1\SASApp\SASEnvironment\FinancialManagement\SASCode\UserDefined** directory.

Note: If the **UserDefined** directory does not already exist, create it.

Parameters That You Can Expect

At run time, when a user selects a custom action, a URL is built to call the associated stored process. The URL includes the following parameters, which are available to the stored process:

Parameter Name	Value
`_model`	The model ID for this table
`_modelCode`	The model code for this table
`__`*`dimension-ID`*	The member ID for this dimension, for the selected crossing

The parameter names are available in the _APSLIST. For example:

```
_APSLIST=__19, __8,_archive_path,_model,_metaperson, _metauser, ...
```

Dimension IDs and member IDs are represented by parameters beginning with two underscores (__). The parameter name following the underscores is the dimension ID, and the parameter value is the member ID for the selected crossing. The simple example below begins by scanning the list for variables beginning with two underscores (such as `__19`) and extracting the dimension IDs and member IDs.

Note: The example code does not filter crossings based on SAS Financial Management Security. In addition, the example code does not restrict the stored process from returning source data to a user that might not have access to the source data.

Example Code 5.1 *Example Stored Process for Custom Cell Action*

```
/*+------------------------------------------------------------------
| Copyright (c) 2011 by SAS Institute Inc., Cary, NC, USA.
| All rights reserved.
| Name: viewtrans.sas
| Purpose: show StageFM transactions that make up a cell value
+------------------------------------------------------------------+*/
*ProcessBody;
ods path(prepend) sashelp.sasweb2(read);
%rptinit(style=sasweb2);


options mprint;

/*assign library (modify path as necessary) */
libname StageFM 'D:\SAS\SASBI\Lev1\SASApp\Data\FinancialManagement\StageFM';

%let modelCode=&_modelCode;
%global account intorg analysis time currency;

* extract crossing values from the parameter list;
%model
%getModelHierarchies('Work','HierOut',"&modelCode.", environment='default')

data DimType;
    set work.HierOut;
    If strip(dimension_type_cd) IN ("ACCOUNT", "INTORG", "ANALYSIS", "TIME", "CURRENCY")
        then call symputx(dimension_type_cd, dimension_cd);
run;

data _null_ /picklist='finance/finance.txt' saveclassl='finbatch' getclassl='finbatch';
    length parameter $32;
    length value $1000;
    length dimID dim member $200;
    dcl javaobj oMetadata("com/sas/solutions/finance/api/Metadata");
    oMetadata.ExceptionDescribe(1);
    oMetadata.callVoidMethod("setEnvironment", "default");
    call METADATA_PASSID("oMetadata", "");
    * get the list of filters ;
    do until(parameter = '');
        i+1;
        %put &_APSLIST;

        parameter = scan("&_APSLIST", i, ",");
        put parameter=;
        if parameter ne '' then do;
            value = symget(parameter);
            put value=;

            if substr(parameter,1,2)='__' then do;
                dimID=substr(parameter,3);
                *if dimID ne 'FREQ' then do;
                    put dimid=;
                    oMetadata.callStringMethod("getDimensionCode", trim(dimID),dim);
                    oMetadata.callStringMethod("getMemberCode", trim(dimID),trim(value), member);
                    /* set dimension values, such as ACCOUNT=10020 */
                    call symputx(dim, member);
                    put dim=;
                    put member=;
                *end;
            end;
        end;
    end;
    oMetadata.delete();
    run;

/*
data StageFM.queryParameters;
    length DIMENSION_TYPE_CD MEMBER_CD $32;
    DIMENSION_TYPE_CD = "ACCOUNT"; MEMBER_CD = symget("&ACCOUNT."); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=4; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "TIME"; MEMBER_CD = SYMGET("&TIME."); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=2; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "CURRENCY"; MEMBER_CD = symget("&CURRENCY."); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "ANALYSIS"; MEMBER_CD = symget("&ANALYSIS."); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "INTORG"; MEMBER_CD = symget("&INTORG."); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=4; INCLUDE_ROLLUPS=0; output;
run;


*/

%put &INTORG;
%put &ACCOUNT;
data StageFM.queryParameters;
    length DIMENSION_TYPE_CD MEMBER_CD $32;
    DIMENSION_TYPE_CD = "ACCOUNT"; MEMBER_CD = symget('ACCOUNT'); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=4; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "TIME"; MEMBER_CD = SYMGET('TIME'); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=2; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "CURRENCY"; MEMBER_CD = symget('CURRENCY'); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "ANALYSIS"; MEMBER_CD = symget('ANALYSIS'); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
    DIMENSION_TYPE_CD = "INTORG"; MEMBER_CD = symget("&INTORG."); INCLUDE_MEMBER=1;
    INCLUDE_LEAVES=4; INCLUDE_ROLLUPS=0; output;
RUN;

%fmquery(modelCode="&modelCode.",localSasLibName='Work', sasLibName='StageFM',
    queryDataSetName='queryParameters', resultDataSetName='NONMDXResultDataSetName',
    environment='default')

/*Store the records in NONMDXResultDataSetName into macro variables*/
data _null_;
    set NONMDXResultDataSetName nobs=end;
    account_code=tranwrd(account_code,".vc","");
    intorg_code=tranwrd(intorg_code,".vc","");
    string=strip(account_code)||"+"||strip(time_code)||"+"||strip(currency_code)||
        "+"||strip(analysis_code)||"+"||strip(intorg_code);
    call symputx("dim"||strip(put(_n_,8.)), string);
    call symputx("end", end);
run;

%global sqlobs;

%macro subset_transaction_table;
proc sql;
    select a.gl_account_id, a.initiating_internal_org_id, a.analysis_id,
        a.affected_time_period_id, a.currency_cd, a.transaction_amt
    from StageFM.gl_transaction_sum a
    where
        %do i=1 %to &end.;
            (gl_account_id = "%scan(&&dim&i.,1, "+")" and
                affected_time_period_id= "%scan(&&dim&i.,2, "+")" and
                currency_cd = "%scan(&&dim&i.,3, "+")" and
                analysis_id="%scan(&&dim&i.,4, "+")" and
                initiating_internal_org_id = "%scan(&&dim&i.,5, "+")")
                %if &i. ne &end %then %do;
                    or
                %end;
                %else %do;
                    ;
                %end;
        %end;
    ;
quit;
%mend;

%subset_transaction_table

data _null_;
    if (symget("SQLOBS") = 0) then do;
        file print;
        put "NOTE: No rows were found";
        value=symget("&ACCOUNT.");
        put "ACCOUNT= " value;
        value=SYMGET("&INTORG.");
        put "ORG= " value;
        value=symget("&ANALYSIS.");
        put "ANALYSIS= " value;
        value=SYMGET("&TIME.");
        put "TIME= " value;
        value=symget("&CURRENCY.");
        put "CURRENCY= " value;
        end;
run;
title;
footnote;
proc printto;
quit;

ods _all_ close;
ods listing;
%stpend;
```

Note: Line breaks inserted for readability.

Register the Stored Process

Define the stored process with package output to the personal repository, as follows:

1 Log on to SAS Management Console as an administrator.

2 On the **Folders** tab, right-click a shared folder and select **New Stored Process**. One possible location is the `/Products/SAS Financial Management/Custom Cell Actions` folder.

   Note: Create the `Custom Cell Actions` folder if it does not already exist.

3 On the Execution page of the wizard, complete the following:

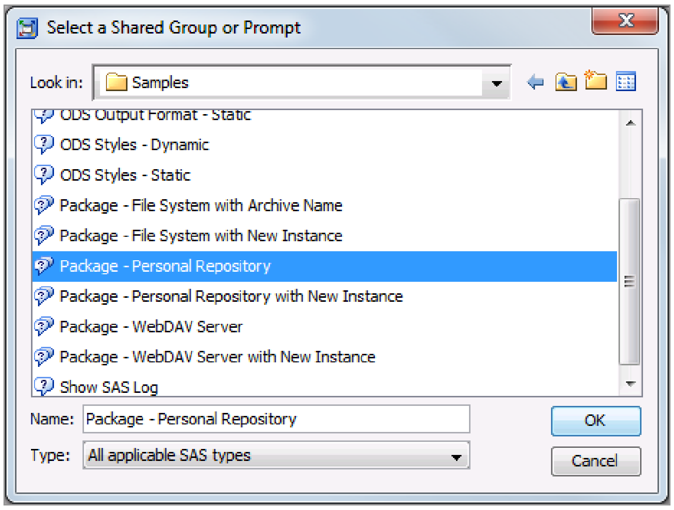
   - select the stored process server
   - define the name and source for the stored process
   - select the **Package** check box

4 On the Parameters page, define any input parameters that are required by the stored process.

5 For the results options, select output to the personal repository as follows:

   a Click **Add Shared**.

   b In the Select a Shared Group or Prompt dialog box, navigate to `SAS Folders\Products\Intelligence Platform\Samples`. Select **Package - Personal Repository**.



Select a Shared Group or Prompt

Look in: Samples

- ODS Output Format - Static
- ODS Styles - Dynamic
- ODS Styles - Static
- Package - File System with Archive Name
- Package - File System with New Instance
- Package - Personal Repository
- Package - Personal Repository with New Instance
- Package - WebDAV Server
- Package - WebDAV Server with New Instance
- Show SAS Log

Name: Package - Personal Repository

Type: All applicable SAS types

CAUTION! Do not select **Package - Personal Repository with New Instance**.

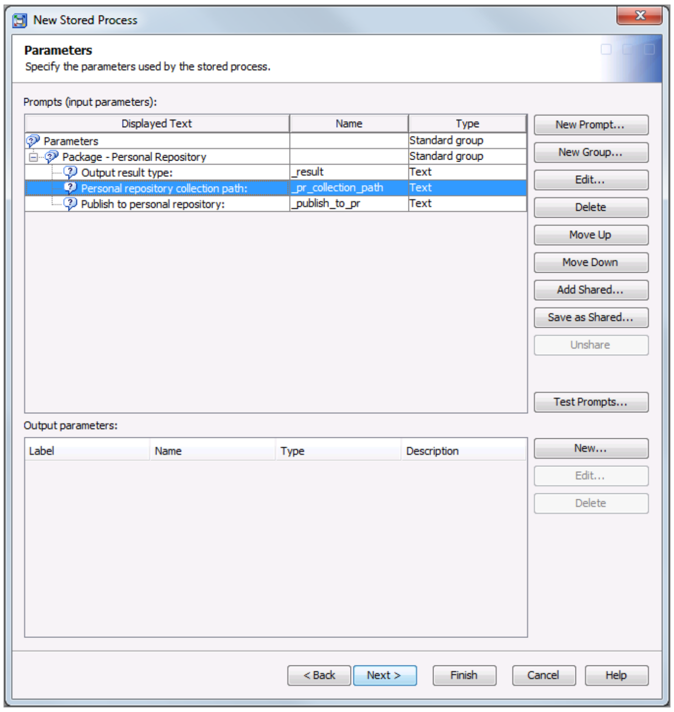
c Click **OK** to save your changes and close the dialog box.

d In the Add Package dialog box, click **OK**.

6 In the parameter list, select **Package - Personal Repository** and click **Unshare** to unshare the prompts so that you can modify them.

A warning message is displayed that asks whether you want to continue. Click **Yes**.

7 Expand the options for the **Package - Personal Repository** prompt.



New Stored Process

Parameters

Specify the parameters used by the stored process.

Prompts (input parameters):

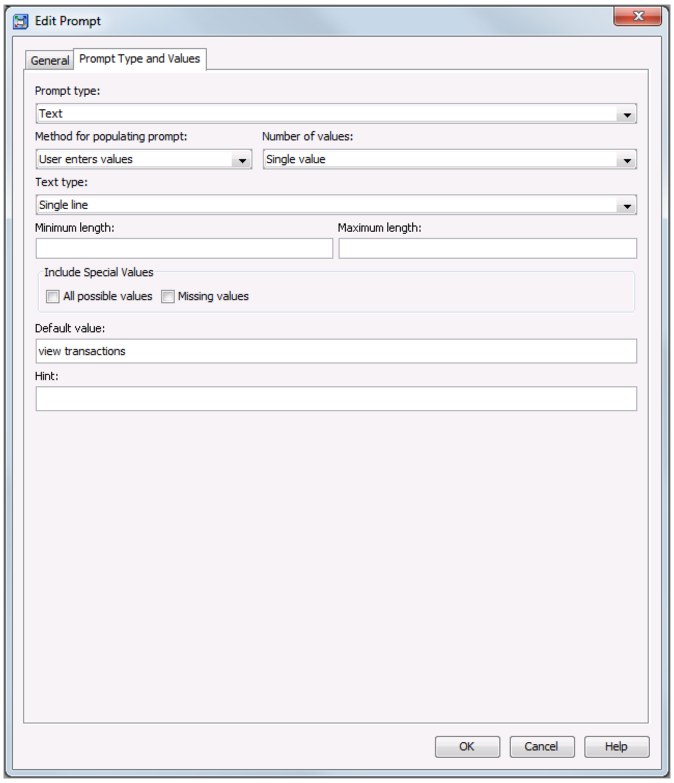
Displayed Text	Name	Type
Parameters		Standard group
Package - Personal Repository		Standard group
Output result type:	_result	Text
Personal repository collection path:	_pr_collection_path	Text
Publish to personal repository:	_publish_to_pr	Text

Output parameters:

Label	Name	Type	Description

8 Select the **Personal Repository collection path** prompt and click **Edit**.

9 On the **Prompt Type and Value** tab, enter a name (such as `viewtransactions`) in the **Default value** text box.



Prompt type: Text

Method for populating prompt: User enters values

Number of values: Single value

Text type: Single line

Minimum length:

Maximum length:

Include Special Values

[ ] All possible values [ ] Missing values

Default value: view transactions

Hint:

10 Keep the defaults for the other results prompts, and click **OK** to save the stored process definition.

11 Make sure that users have ReadMetadata and WriteMetadata access to the stored process.

For more information about registering a stored process, see the *SAS Stored Processes: Developer's Guide* and the online Help for SAS Management Console.

Update the Resource File

Define the Custom Action

Custom actions are defined in a resource file that is stored on the middle tier, where the web application server resides. A good location is a directory such as ***SAS-config-dir*\Lev1\CustomAppData\FMCustomActions**.

Note: Create the **FMCustomActions** directory if it does not already exist.

The resource file is an XML file that contains the following contents:

```
<?xml version="1.0"?>
<customActions>
  <action name="action-name" onCell="true|false" onRollups="true|false"
      onLabels="true|false" onReadTable="true|false" onWriteTable="true|false">
    <description>description of this stored process</description>
    <url>URL to fallback page</url>
    <path>path to stored process metadata definition</path>
  </action>
```

```
</customActions>
```

Where:

- *action-name* is the name of the stored process, as defined in the metadata repository. In Microsoft Excel, it appears as the custom action.
- *fallback page* is the page to be displayed if the custom action fails for some reason. SAS Financial Management expects this file (with a name of main.html) to be available from the URL that you define in the resource file.

  In the example below, the fallback page would be http://www.mycompany.com/CustomActions/Error/main.html. Each custom action can have its own page (with its own URL), or you can specify the same URL for multiple actions.
- *path* is the path to the stored process definition in the metadata repository. Do not include a slash (/) before **`Products`**, and do not include the name of the stored process.

  For example:

  ```
  <?xml version="1.0"?>
  <customActions>
    <action name="View transactions" onCell="true" onRollups="true"
        onLabels="false" onReadTable="true" onWriteTable="false">
      <description>View transactions</description>
      <url>www.mycompany.com/CustomActions/Error</url>
      <path>Products/SAS Financial Management/Custom Cell Actions</path>
    </action>
  </customActions>
  ```

Set the JVM Option

If you have not already done so, tell SAS Financial Management where to find the resource file. Add the following option to the JVM options for the managed server to which SAS Financial Management is deployed (by default, SASServer3):

```
-Dsas.customActions.customizations=file:///path-to-resource-file
```

For example:

- on Windows

  ```
  -Dsas.customActions.customizations=file:///C:/SAS/Config/Lev1/CustomAppData/
  FMCustomActions/CustomActions.xml
  ```
- on UNIX

  ```
  -Dsas.customActions.customizations=file:////install/cfgsas94/SASConfig/Lev1/
  CustomAppData/FMCustomActions/CustomActions.xml
  ```

Note: Line breaks are added for readability only. For information about configuring your web application server, see the *SAS Intelligence Platform: Web Application Administration Guide, Second Edition* located at http://support.sas.com/documentation/onlinedoc/intellplatform/index.html.

The JVM option applies when you restart the managed servers for SAS Financial Management and ODCS (typically, SASServer3, SASServer4, and SASServer5).

Note: If you update the resource file, you must also restart the managed servers.

Select the Action

In Microsoft Excel, right-click a cell in a read-only table and select **Tools** to see the new action that you created.

6

Customizing the Form Status Dashboard

Overview

This chapter describes the following topics:

- the form status dashboard
- indicators in the form status dashboard
- customizing the form status

The Form Status Dashboard

Overview

Users can view a graphical display of form status in a SAS BI Dashboard portlet in the SAS Information Delivery Portal or in the SAS BI Dashboard Viewer.

The display is filtered to contain information only about forms that are available to the user. It contains links that take the user to the Forms workspace.

The Dashboard Portlet

To display the Form Status dashboard in the SAS Information Delivery Portal, add a SAS BI Dashboard portlet to a portal page.

- To modify the display, click the **Edit Content** button in the portlet's toolbar.
- To view the indicators in the SAS BI Dashboard viewer instead of in a portlet, click in the **BI Dashboard** toolbar.

Click an indicator to open the Forms workspace of SAS Financial Management, with that filter applied. (This might trigger an additional logon.) For example, if you click the **Overdue** section of the **Approaching deadline** indicator, the display is filtered to show only forms that are overdue.

SAS BI Dashboard Viewer

The SAS BI Dashboard Viewer is also available in the SAS BI Dashboard application. To open the application, enter the following web address in your web browser:

```
http://server:port/SASBIDashboard
```

Where:

- *server* is the host name of the web application server.
- *port* is the port number of the managed server to which SAS BI Dashboard is deployed (typically, SASServer1).

Requirements for the Form Status Dashboard

To view the Form Status dashboard, users must belong to either the BI Dashboard Users group or the BI Dashboard Administrators group. In addition, users must have at least one of the following capabilities:

- Form Administration
- Submit Financial Forms
- Approve Financial Forms

Users must also have ReadMetadata permission for the **`/Products/SAS Financial Management/5.5 Standard Reports`** folder and ReadMetadata permission for the **`/Products/SAS Financial Management/Dashboards/Form status`** folder.

Note: Without ReadMetadata permission on a folder, users cannot navigate to items beneath that folder. For more information, see “Best Practices for Managing SAS Folders” in the *SAS Intelligence Platform: System Administration Guide*.

Form Status Dashboard Indicators

Overview

By default, the Form Status dashboard contains the following indicators:

- **Status – all forms**—forms that are not yet started, in progress, or completed
- **Approaching deadline – all forms**—forms that are overdue or approaching deadline

Two additional indicators are available that display the same information by form set.

Status Indicator

This indicator displays a count of forms that are not yet started, in progress, or completed. The status groups map to status in the Forms workspace as follows:

Bottom-up forms:

- **Not yet started**—forms with a status of READY, HOLDING, or any variant of UNEDITED.

  UNEDITED includes the following variants:

  - UNEDITED_READY_TO_SUBMIT. This status can apply to a form that has no child forms, or to a parent form with child forms that have been submitted.
  - UNEDITED_AWAIT_CHILDREN. This status can apply to a parent form with child forms that have not yet been submitted. Therefore, the parent form is not ready to be submitted.
- **Completed** includes forms with a status of APPROVED.
- **In progress** includes forms with a status of EDITED, SUBMITTED, CHECKED OUT, REJECTED, or PARTIALLY APPROVED.

Top-down forms:

- **Not yet started** includes forms with a status of HOLDING or UNEDITED.

  Note: Forms with a status of HOLDING are available only to form administrators. For other users, these forms are not included in the dashboard or the Forms workspace.
- **Completed** includes forms with a status of PUSHED or COMPLETED.
- **In progress** includes forms with a status of EDITED.

Approaching Deadline Indicator

This indicator displays a count of forms that are approaching deadline or past due. The status groups map to status in the Forms workspace as follows:

- **Approaching deadline** includes forms without **Completed** status that are within a specified number of days of the form's deadline. The range is customizable. The default is 5 days.
- **Overdue** includes forms without **Completed** status whose overdue flag has been set by the server.
- **Other** includes forms that are not overdue and not approaching deadline.

Customizing the Form Status Dashboard

Overview

The form status dashboard is based on the following components:

- dashboard content, located in the`/Products/SAS Financial Management/Dashboards/Form status` folder of the SAS content server. It includes the **Form process status** dashboard (Form process status.dcx) and the following indicators:
  - **Status – all forms** (Form status indicator - all forms.idx)
  - **Status – by form set** (Form status indicator - by form set.idx)
  - **Approaching deadline – all forms** (Forms approaching deadline indicator - all forms.idx
  - **Approaching deadline – by form set** (Forms approaching deadline indicator - by form set.idx)

  This folder also contains four indicator data files (IMX files) that provide data to the four indicators.
- a stored process, located in `/Products/SAS Financial Management/5.5 Standard Reports/Form Status`. Refreshing the dashboard publishes a SAS package to an archive that contains SAS data sets. Each data set provides data to the corresponding IMX files. The archive is specific to the user who is logged on to the portal.
- a Java API class, located in com/sas/solutions/finance/api/FormStatusModel, with methods that are called by the stored process.

You can customize the dashboard and the underlying code in these ways:

- You can modify the dashboard design.
- You can modify the underlying stored process (formstat.sas).

Modifying the Dashboard Design

If you belong to the BI Dashboard Administrators group (or have the BI Dashboard: Administration role), you can modify the dashboard design. For example, you can change the dashboard layout, add or remove indicators, or change the graph style for an indicator.

Note: Because the dashboard data is generated by a stored process, some dashboard customizations are not appropriate. In particular, do not add links to an indicator. The Form Status stored process creates links to the Forms workspace for the four indicators that are described above.

For more information about modifying the dashboard design, see the *SAS BI Dashboard: User's Guide* at http://support.sas.com/documentation/onlinedoc/bidashboard/index.html

Customizing the Stored Process

The stored process that generates the indicator data is defined in the SAS content folders (**`/Products/SAS Financial Management/5.5 Standard Reports/Form Status`**). Refreshing the dashboard publishes a SAS package to an archive that contains SAS data sets. Each data set provides data to the corresponding IMX files.

The stored process code, formstat.sas, is located in the following directories:

- on Windows

  `!SASHome\SASFoundation\9.4\finance\sasstp`

- on UNIX

  `!SASHome\SASFoundation\9.4\sasstp\finance`

By default, the filter selects forms with a deadline within the next 5 days. To change this range, complete the following steps:

1. Open the formstat.sas file for editing.

2. Locate the following line:

   ```
   j.callStaticIntMethod("getFilterApproachingDeadlineParamValue",
       filterApproachDlineParamValue);
   ```

3. Comment out the line and add code that sets a new parameter value.

   The following example sets the parameter value to 12. A value of 12 configures the filter to select forms that have a deadline within the next 12 days.

   ```
   /*
   j.callStaticIntMethod("getFilterApproachingDeadlineParamValue",
   filterApproachDlineParamValue);
   */
   filterApproachDlineParamValue = 12;
   ```

4. Save the file.

Note: You can customize only the forms selected by the deadline value in the formstat.sas file. Note that the underlying code, as well as this stored process, might change in a subsequent release of SAS Financial Management. If you do make changes, we recommend that you use a copy of formstat.sas so that your changes are not lost in the event of an upgrade.

7

The SAS Financial Management Add-In API for Microsoft Excel

Overview

This chapter describes the following topics:

- about the SAS Financial Management Add-In API for Microsoft Excel
- setup for using the API
- general usage information
- summary of classes
- including CDA functions

About the SAS Financial Management Add-In API for Microsoft Excel

With the SAS Financial Management Add-In for Microsoft Excel and the SAS Financial Management Add-In API for Microsoft Excel, you can use Visual Basic for Applications (VBA) to write macros that interact with SAS Financial Management objects. For example, you might perform some of the following tasks:

- Launch a SAS Financial Management report in batch mode, automatically log on to the SAS Financial Management server, and print the report with updated numbers.
- Retrieve SAS Financial Management data and metadata.
- Use the FMMember selection dialog box in a cell data access (CDA) report.
- Execute code that is based on events from the SAS Financial Management objects.
- Apply custom formatting to SAS Financial Management tables.

Setup for Using the API

The API requires a reference to the SASSESExcelAddin.tlb type library. In Microsoft Excel, follow these steps to add the reference:

1. Click the **Developer** tab.
2. Click **Visual Basic**.
3. From the **Tools** menu of the Visual Basic Editor, select **References**.
4. From the list of available references, select **SAS Financial Management Add-in**.

If **SAS Financial Management Add-in** is not in the list, click **Browse** to select the TLB file and add it to the list. The file is located in the directory where you installed the add-in.

Note: If you had an earlier version of the SAS Financial Management Add-In for Microsoft Excel, deselect the check box for **SAS Financial Management Add-in** on the References page, click **OK**, and exit Excel. Then re-open Excel and add the new TLB file as described above.

5 Click **OK**.

In the Microsoft Excel options, verify that your default file save format is one that supports macros:

1 In Microsoft Excel 2010, open **File** ▶ **Options**. From the options menu on the left, click **Save**.

2 From the **Save files in this format** drop-down list, select one of the following:

- **Excel Binary Workbook (*.xlsb)**
- **Excel Macro-Enabled Workbook (*.xlsm)**

3 Click **OK**.

General Usage Information

Declaring the FMAddIn Object

In the `Declarations` section of the Workbook module, declare the FMAddIn object and other SAS Financial Management objects in code that resembles the following:

```
Public addin As FMAddIn
Public table As FMTable
Public cube As FMCube
Public user As FMUser
```

To use the events framework, the declarations for FMAddin and FMTable should resemble the following code:

```
Public WithEvents addin As FMAddIn
Public WithEvents table As FMTable
```

For more information about the events framework, see “Handling Events” on page 105.

Working with Objects

The FMAddin Object

To get a reference to the FMAddIn object, use code that resembles the following:

```
Dim conn As Connect
...
```

```
Set conn = Application.COMAddIns.Item("SASSESExcelAddIn.Connect").Object
Set addin = conn.FMAddIn
```

For the remainder of this chapter, the code examples assume that you already have a reference (called **`addin`**) to the FMAddIn object. (Your code should contain only one instance of the FMAddIn object.)

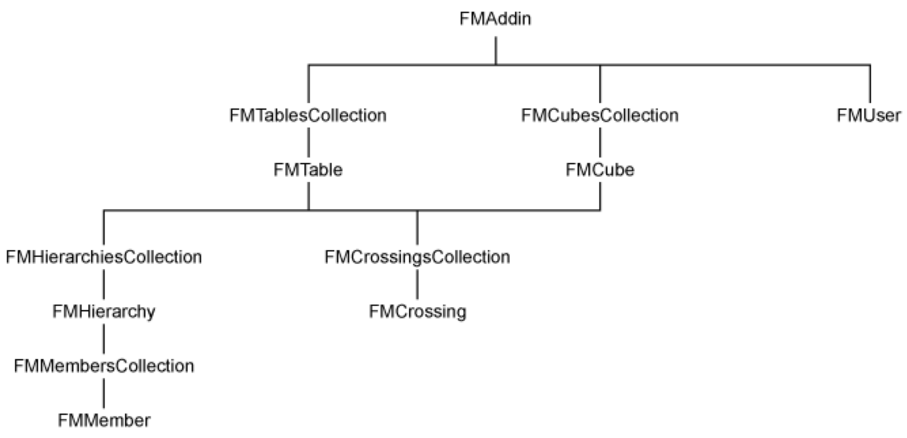
From the FMAddIn object, you can get a reference to the FMTablesCollection object or to an FMCubesCollection object. The tables collection represents all tables in the workbook. Each FMTable object in the collection represents a data entry or read-only table in the current workbook. The cubes collection represents all virtual cubes (results models) on the server. Each FMCube object represents a virtual cube.

Note: We recommend using `Option Explicit` in your code. This option requires all variables to be explicitly declared.

The following diagram shows classes in the SAS Financial Management Add-In API for Microsoft Excel. It indicates which classes contain references to other classes.

Note: The diagram is not intended to imply any inheritance from one class to another.

Figure 7.1 Classes in the SAS Financial Management Add-In API for Microsoft Excel



Objects in a Collection

For information about instantiating a collection, see "The FMCollections Class" on page 112.

To get a reference to an object in a collection, you can specify an index into the collection. For example, **`addin.Tables(0)`** references the first table in an FMTablesCollection object.

You can also name an object in the collection. To get a reference to an object in the FMCubesCollection, FMHierarchiesCollection, or FMMembersCollection, you specify the code for the cube, hierarchy, or member.

Example:

```
Dim cube As FMCube
Set cube = addin.cubes("Default_Model")
```

Another approach is to iterate through the collection. The following code iterates over a collection of server hierarchies in a cube:

```
For Each hierarchy In cube.ServerHierarchies
    ...
 Next hierarchy
```

Table Objects

To get a reference to a table, use the table name (tables do not have codes).

Example:

```
Dim table As FMTable
Set table = addin.Tables("NewTable0")
```

In Excel, the location of a table is defined as a named range. When you add the first table, it is automatically named **`NewTable0`**. The next table is named **`NewTable1`**, and so on.

Note: A user might change the name of a table (in the table properties), but the new name is only for display purposes and cannot be used in the code. For more information, see the getTableName method of the FMTable class.

Case Sensitivity

In function or subroutine arguments, member codes are not case sensitive. All other codes, such as dimension codes and model codes, are case sensitive.

Handling Events

About Events

An event is an action that happens in Excel (for example, logging in or refreshing a table). Event handlers are called when the user performs the specified action.

For an event to be captured:

- The object that the event is associated with must be declared using the **`WithEvents`** clause.
- There must be an existing reference to the object.

  Example:

  ```
  Set table = addin.tables("NewTable0")
  ```

Write an Event Handler

To write an event-handling procedure, use code similar to the following example, which is invoked when the user refreshes the worksheet:

```
Public Sub addin_AfterRefresh()
    MsgBox "Refresh event trapped in VBA"
End Sub
```

The name of the procedure is *object-name* + _ + *event-name*.

Note that an action can trigger multiple events. For example, if a user selects **View** ▶ **Refresh**, the table refresh event is triggered, followed by the worksheet's refresh event. If a table object's Refresh method is called, or if the user performs an action that affects a single table, then only that table's refresh event is triggered.

Suppose that you want to resize the columns for a table each time the table is refreshed. To ensure that the table columns are always resized correctly, you must add the resizing code to both the **`table0_AfterRefresh`** event handler and the **`addin_AfterRefresh`** event handler.

Example:

Example Code 7.1 *Event Handler*

```
Public WithEvents table0 As FMTable
Public WithEvents addin As FMAddin
...
' Event handler for table0
Private Sub table0_AfterRefresh()
    ' Temporarily disable screen updating
    Application.ScreenUpdating = False

    ' Resize columns to have a uniform width
    startColumn = table0.Position(fmArea_Column, fmType_startColumn)
    endColumn = table0.Position(fmArea_Column, fmType_endColumn)
    For col = startColumn To endColumn
        Columns(col).ColumnWidth = 20
    Next col

    ' Re-enable screen updating
    Application.ScreenUpdating = True
End Sub

' addin object's AfterRefresh event handler
Private Sub addin_AfterRefresh()
    Application.ScreenUpdating = False

   ' Check to be sure this table is in the active worksheet
   If Range(table0.Name).Worksheet.Name = ActiveSheet.Name Then
        ' Resize columns
        startColumn = table0.Position(fmArea_Column, fmType_startColumn)
        endColumn = table0.Position(fmArea_Column, fmType_endColumn)
        For col = startColumn To endColumn
            Columns(col).ColumnWidth = 20
        Next col
    End If

    Application.ScreenUpdating = True
End Sub
```

To handle a table refresh that occurs when the user selects **View** ▸ **RefreshAll**, you would write similar code for the **`addin_AfterRefreshAll`** event handler.

If you wanted to resize the columns of all tables to have a uniform width, then you would write an **`addin_AfterTableRefresh`** event handler, which would be called for each table that was refreshed.

For more information about specific events, see the event summaries for the FMAddin class and the FMTable class.

Activating the Log

If enabled, a log file records information about queries generated by the SAS Financial Management Add-In for Microsoft Excel. You can also write to the log using the traceWrite function of the FMAddin class.

By default, this log is disabled.

- For information about activating the log, see the SAS Usage Note at http://support.sas.com/kb/46/178.html.
- To prevent the log file from becoming too long, we recommend that you specify a DebugLevel no higher than `2` in the configuration file.

Summary of Classes

The following table summarizes the classes that make up the SAS Financial Management Add-In API for Microsoft Excel.

Table 7.1 Summary of Classes

Class	Description
FMAddIn	The top-level class for manipulating the add-in.
FMCollections	Base class for other collections such as FMCrossingsCollection and FMTablesCollection. Its properties and methods are inherited by these subclasses.
FMCrossing	Provides access to the properties of a crossing in a table or cube.
FMCrossingsCollection	Represents a collection of crossings.
FMCube	Represents a virtual cube (results model).
FMCubesCollection	Represents a collection of cubes.
FMHierarchy	Represents a hierarchy.
FMHierarchiesCollection	Represents a collection of hierarchies.
FMMember	Represents a member of a hierarchy.
FMMembersCollection	Represents a collection of members.
FMTable	Represents a table.
FMTablesCollection	Represents a collection of tables.

Class	Description
FMUser	Represents the user who is currently logged on.

The FMAddIn Class

The FMAddIn class is the top-level class in the API. From the FMAddIn object, you can get a reference to the tables in the workbook, the cubes that are on the server, and the current user.

Table 7.2 FMAddIn Property Summary

Property	Description
Property `Cubes` As FMCubesCollection	A collection of cubes that are on the server (and that you have access to). Read-only.
Property `isLoggedIn` As Boolean	If **`True`**, the user is logged on. Read-only.
Property `MessageBoxEnabled` As Boolean	If **`False`**, pop-up messages are disabled from the SAS Financial Management Add-In. Typically, you would set this property to **`False`** when you are running in batch mode. The default is **`True`**. Read/write.
Property `MessageBoxResponseOK` As Boolean	The default response to any suppressed message boxes. This property applies only if MessageBox Enabled is set to **`False`**. A value of **`True`** sets the default response to **`Yes`** or **`OK`**. A value of **`False`** sets the default response to **`No`** or **`Cancel`**. The default is **`True`**. Read/write.
Property `Port` As Long	The port number of the middle-tier server on which SAS Financial Management is running. Read-only.
Property `ReadOnly` As Boolean	This property applies if the user is viewing a data-entry form. If **True**, the form cannot be edited.
Property `Secure` As Boolean	**`True`** if the middle-tier server is using the Secure Socket Layer (SSL) protocol. Otherwise, **`False`**. Read-only.
Property `Server` As String	The name of the middle-tier server on which SAS Financial Management is running. Read-only.

Property	Description
Property `Tables` As FMTablesCollection	A collection of tables. Read-only.
Property `Url` As String	The URL to the middle-tier server on which SAS Financial Management is running. Read-only.
Property `User` As FMUser	A FMUser object that represents the user who is currently logged on. Read-only.
Property `Version` As String	The name and version number of this software. Read-only.
Property `VersionDate` As String	The date of this version of the software. Read-only.
Property `VersionID` As String	The version number of this software. Read-only.

Table 7.3 *FMAddIn Class Method Summary*

Function `enumString` (fmEnum As fmEnums, enumValue As Long) As String	Returns the String equivalent of an enumerated constant (for example, the value returned from a Write operation, the name of a role, or an area of the table).
	Parameters:
	■ *fmEnum*: the type of enumerated constant. This parameter can be one of the following: **`fmBudgetMode`**, **`fmDisplayMode`**, **`fmRole`**, **`fmType`**, **`fmArea`**, **`fmSelection`**, **`fmCreditsDebitsDisplay`**, or **`fmWriteBackReturn`**.
	■ *enumValue*: the value to be converted into a string.
	Returns: a string that corresponds to *enumValue* for the specified type of constant.
	Many methods take enumerated constants as parameters or return them as return values. The Write method returns an enumerated constant (a numeric value). You can declare the variable that you are using for the return value as an enumerated constant and then access its string representation. The following code fragment displays a message box for a Write operation that failed, with the reason for the failure:
	`Dim rc As fmWriteBackReturn`
	`...`
	`Set crossing = addin.Tables(0).crossing(4, 3)`
	`rc = crossing.Write(111)`
	`MsgBox "return from write: " & _`
	`  addin.enumString(fmEnums_fmWriteBackReturn, rc)`

Function `findTable` (sheetName As String, row As Long, column As Long) As FMTable	Finds the table object that corresponds to the specified sheet and position. Parameters: ■ *sheetname*—name of a sheet in the workbook. ■ *row*, *column*—position of a table element in the worksheet. Returns: an FMTable object.
Function `GetNewCubesCollection` () As FMCubesCollection	Returns an (empty) FMCubesCollection object.
Function `GetNewTablesCollection` () As FMTablesCollection	Returns an (empty) FMTablesCollection object.
Function `getTableName` (username As String) As String	Returns the internal name of the table that corresponds to a name in the table properties. By default, the first table a user inserts is named **`NewTable0`**, the second table is **`NewTable1`**, and so on. The user might rename the table in the table properties. However, the new name is only a display name. The code requires the original name, which is available via the getTableName function. Parameters: ■ *username*—table name in the table properties. Returns: the original table name.
Function `Login` (environment As String, username As String, password As String) As Boolean	Logs the user on to the middle tier. If the user is already logged on, this function returns **`True`** even if the parameter values are incorrect. Parameters: ■ *environment*, *username*, and *password*—environment, user name, and password for logging on to the middle tier. These parameters are the same values that you would use to log on from the **SAS Financial Management** menu in Excel. The environment value is site-specific. Environments are defined in the sas-environment XML file. For more information, see “Specifying the SAS Financial Management Environment” on page 18. We recommend generating an encoded or encrypted password that you can copy and paste into your code, rather than using a plain-text password. For more information, see the *SAS Intelligence Platform: Security Administration Guide*. Returns: **`True`** if the user is already logged on or if the login succeeds; otherwise, **`False`**.
Function `LoginSharedCredentials`() As Boolean	Logs the user on to the middle tier using the credential from another file that is already open and logged on. The other file must be in the same Microsoft Excel instance. Returns: **`True`** if the user is already logged on or if the login succeeds; otherwise, **`False`**.
Function `Logoff`() As Boolean	Logs the user off the middle tier. Returns: **`True`** if the action succeeded; otherwise, **`False`**.
Function `Refresh`() As Boolean	Refreshes the selected worksheet. This action is similar to the **Refresh** action from the toolbar menu. Returns: **`True`** if the action succeeded; otherwise, **`False`**.

Function `RefreshAll()` As Boolean	Refreshes all open worksheets in the selected file. This action is similar to the **Refresh All** action from the toolbar menu. Returns: **`True`** if the action succeeded; otherwise, **`False`**.
Function `traceWrite` (traceString As String) As Boolean	Writes the contents of *traceString* to the log. This method is helpful in debugging your code. Parameters: ■ *traceString*—string to write. By default, the log is disabled. For information about enabling the log, see "Activating the Log" on page 107.

Table 7.4 *FMAddin Event Summary*

Event	**Description**
Event `AfterLogOff()`	Triggered after the user logs off from the middle tier. A logoff event occurs when there is a call to the Logoff method of the FMAddin object or when the user selects **Log Off** from the toolbar.
Event `AfterLogon()`	Triggered after the user has logged on. This event occurs when there is a call to the Login method of the FMAddin object, when the user selects **Log On** from the toolbar, or when the user opens an Excel report from the portal.
Event `AfterRefresh()`	Triggered after a refresh action—for example, if there is a call to **`addin.Refresh()`** or if the user selects **Refresh** from the toolbar.
Event `AfterRefreshAll()`	Triggered if there is a call to **`addin.RefreshAll()`** or if the user selects **Refresh All** from the toolbar.

Event `AfterTableRefresh` (table As FMTable)	Triggered after a table has been refreshed. This event might occur if there is a call to the **`Refresh`** method of a table object, if the user selects **Refresh** or **Refresh All** from the toolbar, or if the user performs some other manual action, such as a pivot, that triggers a refresh.
	
	If the user refreshes a worksheet, then the table refresh event is triggered, followed by the addin refresh event.
	
	Parameters:
	
	■ *table*—FMTable object that represents the table that was refreshed. If you refresh a worksheet that contains multiple tables, the AfterTableRefresh event is triggered multiple times, once for each table.
	
	The following code configures the event handler to display a message that includes the name of the table that was refreshed:
	
	`Private Sub addin_AfterTableRefresh(ByVal table As FMTable)`
	`   txt = "Addin afterRefresh: " + table.Code`
	`   MsgBox txt`
	`End Sub`

Event	Description

Event `BeforeLogOff`()	Triggered when logoff has been requested but before the user logs off. This event handler returns a Boolean. If the return value is **`True`**, the logoff continues. If the return value is **`False`**, the logoff is canceled.
	
	Example:
	
	`Private Function addin_BeforeLogOff() As Boolean`
	`   response = MsgBox("Do you really want to log off?", _`
	`      vbOKCancel, "SAS Financial Management")`
	`   If response = vbOK Then`
	`      addin_BeforeLogOff = True`
	`   Else`
	`   ' Cancel the logoff process`
	`      addin_BeforeLogOff = False`
	`   End If`
	`End Function`

Event `BeforeTableRefresh` (table As FMTable)	Triggered before a table is refreshed. You might use this event handler to disable screen updating while you are modifying the screen. In the AfterTableRefresh event handler, you could re-enable screen updating. Parameters: ■ *table*—FMTable object that represents the table that was refreshed. If you refresh a worksheet that contains multiple tables, the BeforeTableRefresh event is triggered multiple times, once for each table.

The FMCollections Class

The FMCollections class is the base class for the following collections: FMCrossingsCollection, FMCubesCollection, FMHierarchiesCollection, FMMembersCollection, and FMTablesCollection. Its properties and methods for manipulating a collection are inherited by these subclasses. Use one of the subclasses rather than invoking this class directly.

Note: Do not use `New` to instantiate one of these collections. Instead, use one of the following methods:

- FMAddin.GetNewCubesCollection
- FMAddin.GetNewTablesCollection
- FMCube.GetNewHierarchiesCollection
- FMCube.GetNewCrossingsCollection
- FMHierarchy.GetNewMembersCollection

Table 7.5 *FMCollections Class Property Summary*

Property	Description
Property `Count` As Long	The number of items in the collection. Read-only.

Table 7.6 *FMCollections Class Method Summary*

Class	Description

Sub `Add` (item)	Adds a single item to the collection.
	Parameters:
	■ *item*—item to add.
	This example creates an FMMembersCollection object and adds two members of the **`ACCOUNT.AccountType`** hierarchy to the collection:
	`Dim hierarchy As FMHierarchy`
	`Dim excmems As FMMembersCollection`
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT.AccountType")`
	`Set excmems = hierarchy.GetNewMembersCollection()`
	`Call excmems.Add(hierarchy.Members("StatisticalBalance"))`
	`Call excmems.Add(hierarchy.Members("Equity"))`

Sub `AddAll`(item)	Adds a collection of items to the collection.
	Parameters:
	■ *item*—collection of items to add (for example, an FMMembersCollection object that represents a collection of members).
	The following example creates a FMMembersCollection object and adds all the members of the **`ACCOUNT.AccountType`** hierarchy to the collection:
	`Dim hierarchy As FMHierarchy`
	`Dim mems As FMMembersCollection`
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT.AccountType")`
	`Set mems = hierarchy.GetNewMembersCollection()`
	`Call mems.AddAll(hierarchy.Members)`

Sub `Clear`()	Clears the collection.
Function `Contains` (item) As Boolean	Returns: **`True`** if the collection contains the specified item. Parameters: ■ *item*—single item (for example, an FMMember object if you are searching an FMMembersCollection instance).
Function `IndexOf` (item) As Long	Returns: the zero-based index (position) of the specified item in the collection, or **`-1`** if the item is not found. Parameters: ■ *item*—an object of the collection type (for example, an FMMember object).

Class	Description
Sub `Insert` (index As Long, item)	Inserts *item* at the *index* position in the collection. Parameters: ■ *index*—(zero-based) index into the collection. ■ *item*—object of the collection type.
Sub `InsertAll` (index As Long, item)	Inserts a collection at the *index* position in the collection. Parameters: ■ *index*—(zero-based) index into the collection. ■ *item*—object that represents a collection (for example, an FMMembersCollection object).
Sub `Remove` (item)	Removes an object from a collection (if the object is found). Parameters: ■ *item*—item in a collection.
Sub `RemoveAt` (index As Long)	Removes the item at the *index* position in the collection. Parameters: ■ *index*—(zero-based) index into the collection.
Function `ToString` () As String	Returns: a string that represents the concatenated codes of all the elements in the collection.

The FMCrossing Class

The FMCrossing class provides access to the properties of a crossing. For a table, a crossing is determined by its position in the worksheet (row and column). In a cube, a crossing is determined by a two-dimensional String array of dimension codes and member codes. For examples, see the FMCube class ("The FMCube Class" on page 116) or the FMTable class ("The FMTable Class" on page 131).

Table 7.7 FMCrossing Class Property Summary

Property	Description
Property `Code` As String	An identifier for this crossing. For table crossings, the code is a string that contains information about the row and column for the crossing. For crossings in a cube, the code is a concatenated string of model code, dimension codes, and member codes, such as the following: `DefaultModel_ACCOUNT_NETINCOME_TIME_JAN2003_ANALYSIS_BUDGET...` Read-only.
Property `Column` As Long	The column position of this crossing in the worksheet. Applies only to tables. Read-only.

Property	Description
Property `ColumnRelative` As Long	The column position of this crossing, relative to the table (*crossingColumn* - *firstColumn* + 1). Applies only to tables. Read-only.
Property `DimensionMembers` As String()	A two-dimensional array of strings that contain the dimensions and members that apply to this crossing, in the form (*dimension*, *member*). Read-only.
Property `Length` As Long	The number of dimensions in this crossing. Read-only.
Property `NewValue` As Double	For a cube or a table, the NewValue is the value that is written to the server for this crossing when the BatchWrite method is called. Read/write.
Property `Row` As Long	The row position of this crossing in the worksheet. Applies only to tables. Read-only.
Property `RowRelative` As Long	The row position of this crossing, relative to the table (*crossingRow* - *firstRow* + 1). Applies only to tables. Read-only.
Property `ScaledValue` As Double	For tables, this property contains the value of the crossing divided by the current scale of the table. This value is similar to the value that is shown in the table. For cubes, this property contains the value of the crossing. (It is identical to the Value property.) Read-only.
Property `Value` As Double	The value of this crossing, before any table scaling is applied. Read-only.
Property `Writeable` As Boolean	If **`True`**, the crossing is writable. For crossings that are derived from a cube, this property does not honor member security. Read-only.

Table 7.8 *FMCrossing Class Method Summary*

Method	Description
Function `GetMember` (item As String) As FMMember	Returns the member of this crossing for the specified dimension code. Parameters: ■ *item*—dimension code.

Method	Description
Function `GetMemberCode` (item As String) As String	Returns the member code in this crossing for the specified dimension code. Parameters: ■ *item*—dimension code.
Function `Write` (value As Double) As fmWriteBackReturn	Writes *value* to this crossing. For crossings that are derived from a cube, this method does not honor member security. This method ignores the **Intelligent Writeback** option, even if it is enabled in the form's Global properties. Returns: the status of the Write operation. (See Table 7.3 on page 109.)

The FMCrossingsCollection Class

The FMCrossingsCollection class represents a collection of crossings. This class is a subclass of FMCollections.

The following properties are inherited from FMCollections: Count.

The following methods are inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

To create an FMCrossingsCollection object, use the FMCube.GetNewCrossingsCollection method.

The FMCube Class

Overview

An FMCube object represents a virtual cube (results model). With the FMCube class, you can access metadata from the SAS Financial Management data mart. An instance of the FMCube class can be the entry point for metadata about results models, hierarchies, and members. With FMCube methods, you can also read/write facts.

The FMCube class works independently of read-only tables, data entry tables, and CDA expressions. As a result, your code is able to interact with metadata and with facts.

To perform a query for a cube, complete the following steps:

1 Get a reference to the cube.

2 Get a reference to the cube's crossings collection, which is empty to begin with.

3 Set the crossings one or more sets of (*dimension code*, *member code*) values, and add them to the cube's crossings collection. These are the parameters for the query. You must specify dimension and member codes (not names) for each server hierarchy. Dimension codes are case sensitive.

4 Call the cube's ExecuteQuery method to get the values for each crossing in the collection.

The following example creates a set of query parameters, performs a query, and displays the results.

Example Code 7.2 *Query on a Cube*

```
Public Sub TestQuery()
    Dim addin As FMAddIn
    Dim conn As Connect
    Dim cube As FMCube
    Dim Crossings As FMCrossingsCollection
    Dim Crossing As FMCrossing
    Dim crs As FMCrossing
    Dim member As FMMember

         If addin Is Nothing Then
        Set conn = Application.COMAddIns.Item("SASSESExcelAddIn.Connect").Object
        Set addin = conn.FMAddIn
    End If

    ' Get reference to cube
    Set cube = addin.Cubes("My_Model")
    cube.ClearQuery

    ' Specify dimension, member pairs
    Dim dm() As String
    ReDim dm(cube.ServerHierarchies.Count - 1, 1)

    dm(0, 0) = "My_Account"
    dm(0, 1) = "Rent"
    dm(1, 0) = "My_Analysis"
    dm(1, 1) = "Budget"
    dm(2, 0) = "My_Currency"
    dm(2, 1) = "USD"
    dm(3, 0) = "My_IntOrg"
    dm(3, 1) = "Chicago"
    dm(4, 0) = "My_Time"
    dm(4, 1) = "Jan 2013"
    dm(5, 0) = "My_IntOrg_TRADER"
    dm(5, 1) = "All"
    dm(6, 0) = "SOURCE"
    dm(6, 1) = "Total"
    dm(7, 0) = "My_Product"
    dm(7, 1) = "R1000"
    dm(8, 0) = "FREQUENCY"
    dm(8, 1) = "PA"
    dm(9, 0) = "My_Customer"
    dm(9, 1) = "Acme"

    Set Crossings = cube.Crossings
    For Each member In cube.Hierarchies("My_Product").GetMembers("R1000", True, False)
        dm(7, 1) = member.Code
        Set Crossing = cube.Crossing(dm)
        If Crossing Is Nothing Then
            MsgBox "Null Crossing"
            Exit Sub
        End If
        Crossing.newValue = newValue
        Call Crossings.Add(Crossing)
    Next member

    ' Execute query to fetch values
    cube.ExecuteQuery

    Cells(10, 4) = "Value"
    Cells(10, 5) = "Code"

    i = 11
    For Each crs In cube.Crossings

        Cells(i, 4) = Str(crs.Value)
        Cells(i, 5) = crs.Code
        i = i + 1

    Next crs

End Sub
```

After you perform a query, the values that the query returns are available locally. Before performing any additional queries, you would call the cube's ClearQuery method and then define the parameters for the new query.

Writing Values to a Cube

To write values to a cube, you can call the cube's Write method with the crossing and new value as arguments. Alternatively, you can set the NewValue property for each crossing that you want to affect and then call the cube's BatchWriteNew method.

When writing values to a cube, note the following:

- These methods can be used only in a data-entry form or form template.
- These methods cannot be used to write to a crossing that contains a parent member in any of its dimensions.
- The Write, BatchWrite, and BatchWriteNew methods ignore the **Intelligent Writeback** option, even if it is enabled in the form’s Global properties.
- These methods do not honor member security, properties that are associated with tables, cell protection, or visibility rules.
- These methods do not honor driver formulas. As an alternative, you can include the driver formula calculations in your code or execute the **Run driver formula** function on the form set. See the online Help for SAS Financial Management Studio, or the description of the generateFormulaFacts method in “The Model Class” on page 53.

The following example performs a batch write to several crossings. To view the results of the write, you can use the cube's executeQuery method. For an example of the executeQuery method, see Example Code 7.2 on page 117.

Example Code 7.3 *Batch Write Example*

```
Public Sub cubeJustWrite3()
Dim cube As FMCube
Dim Crossings As FMCrossingsCollection
Dim Crossing As FMCrossing
Dim crs As FMCrossing
Dim member As FMMember


    ' If necessary, set addin object (see previous example)
    If addin Is Nothing Then
        Call Setup
    End If

    Set cube = addin.Cubes("My_Model")
    cube.ClearQuery

    ' New value
    Dim newValue As Double
    newValue = 56

    ' Specify dimension, member pairs
    Dim dm() As String
    ReDim dm(cube.ServerHierarchies.Count - 1, 1)

    dm(0, 0) = "My_Account"
    dm(0, 1) = "Rent"
    dm(1, 0) = "My_Analysis"
    dm(1, 1) = "Budget"
    dm(2, 0) = "My_Currency"
    dm(2, 1) = "USD"
    dm(3, 0) = "My_IntOrg"
    dm(3, 1) = "Chicago"
    dm(4, 0) = "My_Time"
    dm(4, 1) = "Jan 2013"
    dm(5, 0) = "My_IntOrg_TRADER"
    dm(5, 1) = "EXT"
    dm(6, 0) = "SOURCE"
    dm(6, 1) = "BaseForm"
    dm(7, 0) = "My_Product"
    dm(7, 1) = "R1001"
    dm(8, 0) = "FREQUENCY"
    dm(8, 1) = "PA"
    dm(9, 0) = "My_Customer"
    dm(9, 1) = "Acme"

    Set Crossings = cube.Crossings

    For Each member In cube.Hierarchies("My_Product").GetMembers("Cakes", True, False)
        dm(7, 1) = member.Code
        Set Crossing = cube.Crossing(dm)
        If Crossing Is Nothing Then
```

```
                MsgBox "Null Crossing"
                Exit Sub
            End If
            Crossing.newValue = newValue
            Call Crossings.Add(Crossing)
        Next member

        ' Write new values
        rcs = cube.BatchWriteNew(True)

        For i = 0 To UBound(rcs)
            Cells(i + 33, 2) = addin.enumString(fmEnums_fmWriteBackReturn, rcs(i))
        Next i

        Application.EnableEvents = True

    End Sub
```

Property Summary

Table 7.9 FMCube Class Property Summary

Property	Description
Property `Code` As String	The code for this cube (for example, **`Default_Model`**). Read-only.
Property `Crossings` As FMCrossingsCollection	The collection of crossings in this cube. Read-only.
Property `CurrencyHierarchy` As FMHierarchy	An FMHierarchy object that contains the currency hierarchy for this cube. Read-only.
Property `Description` As String	The description of this cube. Read-only.
Property `Hierarchies` As FMHierarchiesCollection	A collection of hierarchies (both server hierarchies and client attribute hierarchies) in this cube. Server hierarchies are hierarchies that are defined on the server and that are being used in the specified results model, either directly (via an FMCube object) or via a virtual cube that is attached to an FMTable object. They are based on the required dimensions (such as **`ACCOUNT`**, **`ANALYSIS`**, and **`TIME`**) and any custom defined dimensions (such as **`PRODUCT`** or **`COSTCENTER`**). Client attribute hierarchies are virtual hierarchies that exist only on the client side. They are based on dimension attributes (both system properties and custom properties). Read-only.
Property `Id` As Long	A unique numeric identifier from the SAS Financial Management data mart. Read-only.

Property	Description
Property `Index` As Long	The position of this cube within the cubes collection. Read-only.
Property `Name` As String	The name of this cube. Read-only.
Property `ServerHierarchies` As FMHierarchiesCollection	The collection of server hierarchies that are associated with the cube. For more information about server hierarchies, see the description of the Hierarchies property. Read-only.

Method Summary

Table 7.10 *FMCube Class Method Summary*

Method	Description
Function `BatchWrite` (reQuery As Boolean) As fmWriteBackReturn	Writes the accumulated transactions to the server. **Note:** This method has been deprecated. Use BatchWriteNew instead.
Function `BatchWriteNew` (reQuery As Boolean) As fmWriteBackReturn()	Writes the accumulated transactions to the server. This method does not honor member security. See the note above for further restrictions. Parameters: ■ *reQuery*—requery flag. If **`True`**, the function performs a requery and repaint after the Write operation. If **`False`**, the function performs a repaint only. Returns: an array containing the status of each Write operation. To check the status, use code such as the following: `Range("B8") = addin.enumString(fmEnums_fmWriteBackReturn, rcs(0))`
Sub `ClearQuery` ()	Clears the query definitions on the cube.
Function `Crossing` (item) As FMCrossing	Returns the crossing that is represented by *item*. Parameters: ■ *item*—array of *dimension code*/*member code* value pairs. If any dimension or member is invalid, a null value is returned. Your code should check for this before proceeding.
Sub `ExecuteQuery` ()	Queries the server for all crossings that are defined for the cube.
Function `GetNewCrossingsCollection` () As FMCrossingsCollection	Returns an (empty) FMCrossingsCollection object.

Method	Description
Function `GetNewHierarchiesCollection`() As FMHierarchiesCollection	Returns an (empty) FMHierarchiesCollection object.
Function `isWriteable`() As Boolean	Returns **`True`** if the cube is writable. This method does not honor member security.
Function `Write` (crossing As FMCrossing, newValue As Double) As fmWriteBackReturn	Writes *newValue* to the specified *crossing*. This method does not honor member security. See the note above for further restrictions. Parameters: ■ *crossing*—FMCrossing object. ■ *newValue*—value to write. Returns: the status of the Write operation. (See Table 7.3 on page 109.)

The FMCubesCollection Class

The FMCubesCollection class represents a collection of cubes that are available on the server (and that the user has permission to access). This class is a subclass of FMCollections.

The following properties are inherited from FMCollections: Count.

The following methods are inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

To create an FMCubesCollection object, use the FMAddin.GetNewCubesCollection method.

The FMHierarchiesCollection Class

The FMHierarchiesCollection class represents a collection of hierarchies. This class is a subclass of FMCollections.

The following properties are inherited from FMCollections: Count.

The following methods are inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

To create an FMHierarchiesCollection object, use the FMCube.GetNewHierarchiesCollection method.

The FMHierarchy Class

The FMHierarchy class has properties and methods for accessing members of a hierarchy. For more information about types of hierarchies, see the Hierarchies property of the FMCube class in "The FMCube Class" on page 116 or the FMTable class in "The FMTable Class" on page 131.

Note: Some properties (such as position, role, and available members) apply only to hierarchies that are associated with a table, rather than a cube.

One method to note in the FMHierarchy class is the ShowMemberSelectionDialog method. This method displays a dialog box from which users can select a member of a specified hierarchy.

The following example presents a dialog box from which the user can select a member of the TIME hierarchy:

Example Code 7.4 *Selecting a Hierarchy Member*

```
Dim cube As FMCube
Dim hier As FMHierarchy
Dim selmem As FMMember
Dim premem As FMMember
Dim exclude As FMMembersCollection
...
' Get an instance of the results model /cube + hierarchy
Set cube = addin.Cubes("Default_Model")
Set hier = cube.Hierarchies("TIME")

' Set a preselected member
Set premem = hier.Members("YR2005")
' ... or use the default
' Set selmem = Nothing

' Prepare a list of members to exclude from the dialog
Set exclude = hier.GetNewMembersCollection()

' Add YR1997 and all descendants of YR1997 to the list
Call exclude.Add(hier.Members("YR1997"))
For Each member In hier.GetMembers("YR1997", True, False)
    Call exclude.Add(member)
Next member

' Display the dialog with preselected member and exclusion list
Set selmem = hier.ShowMemberSelectionDialog(premem, exclude, fmDisplayMode_CodeAndDescription)
' ...or use default member and no exclusion list
' Set selmem = hier.ShowMemberSelectionDialog(Nothing, Nothing, fmDisplayMode_CodeAndDescription)
```

The ShowMemberSelectionDialog method displays the Select Member dialog box. In this case, `YR2005` would be pre-selected, and `YR1997` and its descendants would be excluded.

The user selects a member of the hierarchy and clicks **OK**. The return value is the selected member.

Table 7.11 FMHierarchy Class Property Summary

Property	Description

Property `Asof` As Double	The as-of date for this hierarchy.
	To view this date as an Excel date, store it in a Date field.
	Example:
	`Dim hier as FMHierarchy`
	`Dim dt as Date`
	`...`
	`dt = hier.Asof`
	`MsgBox Str(dt)`
	Read-only.

Property `AvailableMembers` As FMMembersCollection	A list of the hierarchy members that are available after the member selection rules have been applied. Read-only.
Property `Code` As String	The dimension code that applies to this hierarchy. Read-only.
Property `Description` As String	The description of this hierarchy. Read-only.
Property `DimensionCode` As String	The dimension code that applies to this hierarchy. Read-only.
Property `DimensionDescription` As String	The dimension description that applies to this hierarchy. Read-only.
Property `DimensionId` As Long	The dimension ID that applies to this hierarchy. Read-only.
Property `DimensionName` As String	The dimension name that applies to this hierarchy. Read-only.
Property `DimensionTypeCode` As String	The dimension type code that applies to this hierarchy. Read-only.
Property `DimensionTypeDescription` As String	The dimension type description that applies to this hierarchy. Read-only.
Property `DimensionTypeID` As Long	A unique numeric identifier from the SAS Financial Management data mart. Read-only.
Property `DimensionTypeName` As String	The dimension type name that applies to this hierarchy. Read-only.

Property	Description

Property `DisplayedMembers` As FMMembersCollection	A collection of the hierarchy members that are currently being displayed.
	The following example retrieves the members of the ACCOUNT hierarchy that are currently displayed in the specified table and displays the results in a message box:
	`Dim txt as String`
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT")`
	`txt = "All displayed members in " & hierarchy.Description _`
	`   & Chr$(10)`
	`For Each member In hierarchy.DisplayedMembers`
	`   txt = txt & " " & member.Code`
	`Next member`
	`MsgBox txt`
	Read-only.

Property `DisplayMode` As fmDisplayMode	The labeling method for this hierarchy, which specifies how displayed members are identified. The value can be one of the following: ■ **`fmDisplayMode_Code`** ■ **`fmDisplayMode_Name`** ■ **`fmDisplayMode_Description`** ■ **`fmDisplayMode_CodeAndName`** ■ **`fmDisplayMode_CodeAndDescription`** Read/write.
Property `HierarchyCode` As String	The code for this hierarchy.
Property `HierarchyIndex` As Long	For a table, this value represents the index of this hierarchy in the set of dimensions that make up the query for the table. For cubes, this value is always **`-1`**. Read-only.
Property `ID` As Long	A unique numeric identifier from the SAS Financial Management data mart. Read-only.
Property `LeafMembers` As FMMembersCollection	A collection of the leaf members of this hierarchy. Read-only.
Property `Members` As FMMembersCollection	A collection of the members of this hierarchy. Read-only.
Property `Name` As String	The name of this hierarchy. Read-only.
Property `Position` As Long	The position of this hierarchy within its section. The section is determined by the Role property. (See below.) If the hierarchy is in the **Available** list, its position is **`-1`**. Read/write.

Property	Description
Property `ReadableMembers` As FMMembersCollection	A collection of hierarchy members that are readable by the current user. Read-only.
Property `ReadDefaultMember` As FMMember	The default Read member for this hierarchy. Read/write.

Property `Role` As fmRole	The role of this hierarchy.
	The role determines the section in which the hierarchy appears. It can have one of the following values:
	■ **`fmRole_Row`**: row
	■ **`fmRole_Column`**: column
	■ **`fmRole_Slicer`**: slicer
	■ **`fmRole_Available`**: available for use in a row, column, or slicer
	The following example performs a pivot of a table by changing the Role and Position properties of a Hierarchy object. When the code has been executed, the ACCOUNT hierarchy is the first column heading in the table.
	`Set table = addin.Tables("NewTable0")`
	`Set hierarchy = table.Hierarchies("ACCOUNT")`
	`hierarchy.role = fmRole_Column hierarchy.Position = 0`
	`' Refresh with a requery`
	`table.Refresh (True)`
	Read/write.

Property `ShowAllMember` As Boolean	This property applies to member property hierarchies (such as Product.Color). If **`True`**, the **All** member (representing all members of the hierarchy) is displayed in a dialog box when the SelectSlicerMember or ShowMemberSelectionDialog method is called. Selecting this member in the dialog box returns a member code of **`_All`**.
	If the hierarchy is derived from a table, then enabling ShowAllMember also affects the hierarchy display in a slicer in the table.
	The default is **`True`**.
	In the following example, if hierProp represents a member property hierarchy, the code disables the display of **`All`** in the member selection dialog box for that hierarchy:
	`boolValue = hierProp.ShowAllMember`
	`  hierProp.ShowAllMember = False`
	`  Set selMember = hierProp.ShowMemberSelectionDialog _`
	`     (Null, Nothing, fmDisplayMode_Code)`
	`  MsgBox "selected " & selMember.Code`
	Read/write.

Property `TargetMember` As FMMember	The hierarchy member that a form is assigned to. Read-only.

Property	Description
Property `VCFilter` As Boolean	If **`True`**, the table is filtered so that virtual children are not included. If **`False`**, virtual children are included. Read/write.
Property `WriteDefaultMember` As FMMember	The default Write member for this hierarchy. Read/write.

Table 7.12 FMHierarchy Class Method Summary

Method	Description

Function `ChangeSlicer` (item) As Boolean	Changes the hierarchy member that is used as the slicer. For example, if you are using a member of the TIME hierarchy as a slicer, you might change from one year to another. (The hierarchy must already be functioning as a slicer. In other words, if the hierarchy is being used in a column or row or is simply available for use, the ChangeSlicer method does not work.)
	
	Parameters:
	
	■ *item*—member of the hierarchy that to become the new slicer. It can be an index into the hierarchy or a member code.
	
	Returns: **`True`** if the change succeeded; otherwise, **`False`**.
	
	In this example, a member of the ACCOUNT hierarchy is being used as a slicer. The code changes the member to **`A1000`**:
	
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT")`
	`rc = hierarchy.ChangeSlicer("A1000")`

Function `GetMembers` (item, recurse As Boolean, reverse As Boolean) As FMMembersCollection	Returns a collection of members of this hierarchy, beginning with the first child of the member that is selected by *item*.
	
	Parameters:
	
	■ *item*—hierarchy member on which to begin processing. This parameter can be an index into the hierarchy or a member code.
	
	■ *recurse*—recursion flag. If **`True`**, the function performs a recursive search and returns all descendants. If **`False`**, it returns only the member's children.
	
	■ *reverse*—reverse order flag. If **`True`**, the function returns the results in reverse order. If **`False`**, the children are returned in the same order in which they appear in the hierarchy.
	
	The following example returns all descendants of the first member of the ACCOUNT hierarchy for a table:
	
	`Dim txt As String`
	`Dim member As FMMember`
	`Set hierarchy = _`
	`   addin.Tables("NewTable0").Hierarchies("ACCOUNT")`
	`txt = "All descendants of " _`
	`   & hierarchy.Members(0).Description & " in " _`
	`   & hierarchy.Description & " hierarchy" & Chr$(10)`
	`For Each member In hierarchy.GetMembers(0, True, False)`
	`   txt = txt & " " & member.Code`
	`Next member`
	`MsgBox txt`

Method	Description
Function `GetNewMembersCollection`() As FMMembersCollection	Returns an (empty) FMMembersCollection object.
Function `IsFlatDimensionType` () As Boolean	Returns: **`True`** if this hierarchy belongs to a flat dimension type; otherwise, **`False`**. Read-only. There are three dimension types that must have flat hierarchies: ANALYSIS, CURRENCY, and FREQUENCY. In addition, a client attribute hierarchy is a flat dimension type. For more information about client attribute hierarchies, see the description of the FMTable.Hierarchies property at "The FMTable Class" on page 131.
Function `IsNonVirtualChildDimensionType` () As Boolean	Returns: **`True`** if the hierarchy does not include virtual children; otherwise, **`False`**. Read-only.
Function `IsServer` () As Boolean	Returns: **`True`** if this hierarchy is defined on the server.
Function `SelectSlicerMember` (DisplayMode As fmDisplayMode) As FMMember	Displays the **Select Member** dialog box for a slicer. Parameters: ▪ *displayMode*—labeling method. The way that displayed members are identified. The value can be one of the following: **`fmDisplayMode_Code`**, **`fmDisplayMode_Name`**, **`fmDisplayMode_Description`**, **`fmDisplayMode_CodeAndName`**, or **`fmDisplayMode_CodeAndDescription`**. Returns: the selected member of the hierarchy.
Function `ShowMemberSelectionDialog` (item, exclude As FMMembersCollection, displayMode As fmDisplayMode) As FMMember	Displays the Show Members dialog box, from which users can select a member of the specified hierarchy. Parameters: ▪ *item*—the member that you want to be highlighted in the dialog box. ▪ *exclude*—collection of members to be excluded from the dialog box. To display all members, create a collection but do not assign it any values. ▪ *displayMode*—labeling method. The way that displayed members are identified. The value can be one of the following: **`fmDisplayMode_Code`**, **`fmDisplayMode_Name`**, **`fmDisplayMode_Description`**, **`fmDisplayMode_CodeAndName`**, or **`fmDisplayMode_CodeAndDescription`**. Returns: the selected member of the hierarchy, which can be used in several ways. For example, the selected member could be used as input to code that modifies a CDA expression.

The FMMember Class

The FMMember class represents a member of a hierarchy, which can be displayed or not. For members of displayed hierarchies, the selection rules can be modified.

Note: Some properties and methods (such as the SelectionRule property and the Expand method) apply only to members of hierarchies that are associated with tables.

Table 7.13 FMMember Class Property Summary

Property	Description
Property `Asof` As Double	The as-of date for this member. To view this date as an Excel date, store it in a Date field. Read-only.
Property `Code` As String	The code for this hierarchy member. Read-only.
Property `Column` As Long	The column position of the top left cell of this member's position in the table. Read-only.
Property `Description` As String	The description of a member of a hierarchy. Read-only.
Property `ID` As Long	A unique numeric identifier from the SAS Financial Management data mart. Read-only.
Property `Level` As Long	The level of this member in the current hierarchy. The top member of the hierarchy has a level of `0`. Read-only.
Property `Name` As String	The name of a member of a hierarchy. Read-only.
Property `Row` As Long	The row position of the top left cell of this member's position in the table. Read-only.

Property	Description
Property `SelectionRule` As fmSelection	Gets or sets the selection rule for a displayed hierarchy member. The value can be one of the following enumerated constants: **`fmSelection_Member`**: selects the designated member. **`fmSelection_Descendants`**: selects the entire subhierarchy subordinate to the designated member but not including the designated member itself. **`fmSelection_MemberAndChildren`**: selects the designated member and all members that are immediately subordinate to it. **`fmSelection_MemberAndDescendants`**: selects the entire subhierarchy from the designated member down. **`fmSelection_MemberAndLeaf`**: selects the designated member and all members that are subordinate to it but that have no members under them. For example, in a Time hierarchy that included years, quarters, and months, this value would select year and months, but not quarters. **`fmSelection_Children`**: selects all members that are immediately subordinate to the designated member. **`fmSelection_Leaf`**: selects all members that are subordinate to the designated member but have no members subordinate to them. **`fmSelection_NoMember`**: excludes the designated member from the subset. All members that are subordinate to the designated member are also excluded, unless you apply additional rules to one or more of these subordinate members. **`fmSelection_NoRule`**: removes any rules from the designated member. Read/write.

	The following code modifies selection rules for the **`ACCOUNT`**, **`ANALYSIS`**, and **`ORG`** hierarchies in a table:
	`Set addin = conn.FMAddIn`
	`Set table = addin.Tables("NewTable0")`
	
	`Set hier = table.Hierarchies("ACCOUNT")`
	`hier.Members("A8000").SelectionRule = fmSelection_NoMember`
	`hier.Members("A7400").SelectionRule = fmSelection_Member`
	`hier.Members("A6300").SelectionRule = fmSelection_NoMember`
	`hier.Members("A7800").SelectionRule = _`
	`   fmSelection_MemberAndDescendants`
	`hier.Members("A7900").SelectionRule = fmSelection_NoMember`
	`hier.Members("A8100").SelectionRule = fmSelection_Member`
	
	`Set hier = table.Hierarchies("ANALYSIS")`
	`hier.Members("ACTUAL").SelectionRule = fmSelection_Member`
	`hier.Members("BUDGET").SelectionRule = fmSelection_NoMember`
	
	`Set hier = table.Hierarchies("ORG")`
	`hier.Members("PLD").SelectionRule = fmSelection_Member`
	
	`' Refresh the table to see the results`
	`table.Refresh (True)`

Table 7.14 FMMember Class Method Summary

Method	Description
Sub `Collapse`()	Collapses the member to hide all its descendants.
Sub `Expand` ()	Expands the member to display all its children.
Sub `ExpandAll` ()	Expands the member to display all its descendants.
Function `getParent`() As FMMember	Returns: the parent of this member. If this member is a top-level member, it returns this member.
Function `IsClientAttributeFilter`() As Boolean	Returns: **`True`** if this member is a member attribute filter.
Function `IsClientCalculatedMember`() As Boolean	Returns: **`True`** if this member is a client calculated member.
Function `IsLeaf`() As Boolean	Returns: **`True`** if this member is a leaf member.
Function `IsReadable`() As Boolean	Returns: **`True`** if this member is readable by the current user.
Function `IsServer`() As Boolean	Returns: **`True`** if this member is defined on the server.
Function `IsVirtual`() As Boolean	Returns: **`True`** if this member is a virtual child.
Function `IsWriteable`() As Boolean	Returns: **`True`** if this member is writable by the current user. This method does not honor member security for a member that is derived from a cube.

The FMMembersCollection Class

The FMMembersCollection class represents members of a hierarchy. This class is a subclass of FMCollections.

The following properties are inherited from FMCollections: Count.

The following methods are inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

The FMTable Class

An FMTable object represents a read-only table or a data entry table. Some elements, such as layout and scale, can be manipulated directly on the table object. Other operations, such as filtering virtual children and showing or hiding members, must be manipulated on the FMHierarchy or FMMember objects that belong to the table.

If you change table properties, call the table's Refresh method with a requery to display the changes. (Alternatively, you could call the addin's Refresh or RefreshAll function.)

Note: The FMTable class does not apply to CDA tables. Use the FMCube class instead.

Table 7.15 FMTable Class Property Summary

Property	Description
Property `Code` As String	The code for this table. Read-only.
Property `Credit` As fmCreditsDebitsDisplay	The manner in which credit values are displayed in this table. Possible values are as follows: **`fmCreditsDebitsDisplay_Default`**: uses the default for the result model **`fmCreditsDebitsDisplay_Negative`**: displays credits as negative numbers **`fmCreditsDebitsDisplay_Positive`**: displays credits as positive numbers Read/write.
Property `Crossings` As FMCrossingsCollection	A collection of crossings in this table. Read-only.
Property `Debit` As fmCreditsDebitsDisplay	The manner in which debit values are displayed in this table. Possible values are as follows: **`fmCreditsDebitsDisplay_Default`**: uses the default for the result model **`fmCreditsDebitsDisplay_Negative`**: displays debits as negative numbers **`fmCreditsDebitsDisplay_Positive`**: displays debits as positive numbers Read/write.
Property `DisplayDebitCreditOnLabel` As Boolean	This setting applies to account member labels. If **`True`**, each row and column heading contains the word **`(debit)`** or **`(credit)`**, whichever is applicable. Read/write.
Property `FilterInvalid` As Boolean	If **`True`**, rows or columns that contain only invalid values are not displayed. Read/write.
Property `FilterInvalidOnColumns` As Boolean	If **`True`**, columns that contain only invalid values are not displayed. Read/write.

Property	Description
Property `FilterInvalidOnRows` As Boolean	If **`True`**, rows that contain only invalid values are not displayed. Read/write.
Property `FilterZeros` As Boolean	If **`True`**, rows or columns that contain only zero values are not displayed. Read/write.
Property `FilterZerosOnColumns` As Boolean	If **`True`**, columns that contain only zero values are not displayed. Read/write.
Property `FilterZerosOnRows` As Boolean	If **`True`**, rows that contain only zero values are not displayed. Read/write.
Property `FreezeCells` As Boolean	If **`True`**, users cannot alter the table layout by operations such as changing the role of dimensions, expanding or collapsing hierarchies, adding or removing filters, and adding or removing calculated members. Read/write.
Property `Hierarchies` As FMHierarchiesCollection	A collection of hierarchies (both server hierarchies and client attribute hierarchies) in this table. Server hierarchies are hierarchies that are defined on the server and that are being used in the specified results model, either directly (via an FMCube object) or via a virtual cube that is attached to an FMTable object. They are based on the required dimensions (such as **`ACCOUNT`**, **`ANALYSIS`**, and **`TIME`**) and any custom defined dimensions (such as **`PRODUCT`** or **`COSTCENTER`**). Client attribute hierarchies are virtual hierarchies that exist only on the client side. They are based on dimension attributes—both system properties and custom properties. The collection of hierarchies includes both hierarchies that are displayed in the table and hierarchies with a role of **`fmRole_Available`**, meaning that they are part of the table but are not currently displayed. Instead, their default Read/write members are used in the table crossings. Read-only. **Note:** Custom properties hierarchies are treated like any other hierarchy.
Property `Index` As Long	The position of this table within the tables collection. Read-only.
Property `Model` As String	The code for the model that is used in this table. Read/write.
Property `Name` As String	The name of this table. Read-only.
Property `ReadOnly` As Boolean	If **`True`**, this table is read-only. Read-only.

Property	Description
Property `RefreshOnOtherTableUpdate` As Boolean	If **`True`**, this table is refreshed when other tables in the same worksheet change. For example, you might set this property to **`True`** for a read-only table so that it is refreshed when a user enters a value in a data entry table in the same worksheet. Read/write.
Property `ScaleValue` As Double	The value by which displayed values are scaled. The actual computed values are divided by this number before they are displayed. Read/write.
Property `ServerHierarchies` As FMHierarchiesCollection	The collection of server hierarchies that are associated with the table. For more information about server hierarchies, see the description of the Hierarchies property. Read-only.

Table 7.16 FMTable Class Method Summary

Method	Description

Sub `AddHold` (FMCrossingsCollection crossings)	Adds a hold for each crossing in the crossings collection. This method is used for the hold and reconciliation feature, which is used to protect cells from indirect changes. This method applies only to bottom-up form sets.
	
	For more information about the hold and reconciliation feature, see “Working with Holds and Reconciliation” in the *SAS Financial Management: Process Administrator's Guide*.
	
	Parameters:
	
	■ *crossings*—collection of crossings for which to hold the value.
	
	Example:
	
	`...`
	`Dim xing1 As FMCrossing`
	`Dim xing2 As FMCrossing`
	`Dim xing3 As FMCrossing`
	`Dim xing4 As FMCrossing`
	`Dim xColl As FMCrossingsCollection`
	
	`Set xing1 = table.Crossing(17, 4)`
	`Set xing2 = table.Crossing(17, 5)`
	`Set xing3 = table.Crossing(17, 3)`
	`Set xing4 = table.Crossing(16, 3)`
	`Set xColl = table.GetNewCrossingsCollection`
	`xColl.Add xing1`
	`xColl.Add xing2`
	`xColl.Add xing3`
	`xColl.Add xing4`
	`table.AddHold xColl`
	
	Note: Holding must be enabled in the table properties. Otherwise, the method is not successful and a message is written to the log.

Method	Description

Sub `BatchWrite` ()	Writes the accumulated transactions to the server.
	
	Writeback is the process of writing back facts to the server. A writeback occurs when the user enters a value in a data entry table and presses Enter. Normally, this operation requires one trip to the server for each value that the user enters. The BatchWrite method enables you to perform multiple updates with a single writeback. The process is as follows:
	
	1 Call the TransactionBegin method for a data entry table, to begin accumulating values.
	2 Write values to the table cells, either manually or programmatically. At this point, only the client-side representation is updated.
	3 Call the BatchWrite method to perform the writeback of the accumulated values.
	
	Example:
	
	`Table.TransactionBegin`
	`For Each c In target`
	`   If c.Value <> newValue Then`
	`      table.Crossing(c.row, c.column).newValue = newValue`
	`   End If`
	`Next c`
	`Table.BatchWrite`
	
	Note: This method ignores the **Intelligent Writeback** option, even if it is enabled in the form's Global properties.

Sub `Collapse` (member As FMMember)	Collapses the selected member to hide all its descendants. Parameters: ■ *member*—hierarchy member to collapse.
Function `Crossing` (row As Long, column As Long) As FMCrossing	Returns the crossing at (*row*, *column*). Parameters: ■ *row* and *column*—Excel worksheet row and column values, after converting the column letter to a number (**`A=1`**, **`B=2`**, and so on).
Sub `Expand` (member As FMMember)	Expands the selected member to display all its children. Parameters: ■ *member*—hierarchy member to expand.
Sub `ExpandAll` (member As FMMember)	Expands the selected member to display all its descendants. Parameters: ■ *member*—hierarchy member to expand.
Sub `FilterMemberCombinations` (mems As FMMembersCollection)	Creates and applies a multi-member table filter based on the selected members. For more information, see the description of the **Filter Member Combination** option in the online Help for the SAS Financial Management Add-In for Microsoft Excel. Parameters: ■ *mems*—collection of members.

Method	Description
Sub `FilterMembers` (row As Long, column As Long)	Creates and applies a single-member table filter based on the selected row or column heading. For more information, see the description of the **Filter Member Combination** option in the online Help for the SAS Financial Management Add-In for Microsoft Excel. Parameters: ■ *row*, *column*—worksheet coordinates of the row or column heading that is used as the filter.
Function `GetNewCrossingsCollection`() As FMCrossingsCollection	Returns an (empty) FMCrossingsCollection for the table.
Function `isDataArea` (sheetName As String, row As Long, column As Long) As Boolean	Returns **`True`** if the specified position is within the data area. For more information about the data area, see the description of the Position function. Parameters: ■ *sheetName*—name of the worksheet. ■ *row*, *column*—worksheet coordinates of the position.

Function `Pivot` (hierarchy As FMHierarchy, role As fmRole, position As Long) As Boolean	Changes the layout of the selected table by changing the role of the hierarchy that was passed in. You must refresh the table in order to see the effects of the pivot operation.
	
	Parameters:
	■ *hierarchy*—hierarchy whose role is to be changed.
	■ *role*—new role for this hierarchy. A role of **`fmRole_Slicer`**, **`fmRole_Row`**, or **`fmRole_Column`** places the hierarchy in the slicer, row, or column section of the table. A role of **`fmRole_Available`** removes the hierarchy from its previous role as a slicer, row, or column and places it in the list of available hierarchies.
	■ *position*—hierarchy's position within its section (slicer, row, or column). If the section contains more than one hierarchy, existing hierarchies are pushed up or down as necessary to accommodate the position that you specify for this hierarchy. A position of **`0`** represents the highest position for the specified role.
	
	Returns: **`True`** if the operation succeeded; **`False`** if the role is the same as the current role or if an error is encountered.
	
	The following example uses the Pivot method of the Table object to change the layout of a table.
	
	With the assumption that the column headings for a table are **`TIME`** and **`ANALYSIS`**, and the only row heading is **`ACCOUNT`**, the following code removes **`ANALYSIS`** from the column headings and adds it to the row headings. The new row headings would be **`ANALYSIS`** and **`ACCOUNT`**, in that order.
	
	`Dim table as FMTable`
	`Dim hierarchy as FMHierarchy`
	`set table = addin.tables("NewTable0")`
	`Set hierarchy = table.Hierarchies("ANALYSIS")`
	`rc = table.Pivot(hierarchy, fmRole_Row, 0)`
	`table.refresh(true)`

Method	Description

Function `Position` (area As fmArea, type As fmType) As Long	Returns the position of a table element. One common use for this method is to determine a range for applying custom formats to a table.
	
	Parameters:
	
	■ *area*—area of the table for which you want to know the position. This parameter can be one of the following:
	
	`fmArea_Table`—entire table
	
	`fmArea_Slicer`—table slicer area
	
	`fmArea_Row`—row heading area
	
	`fmArea_Column`—column heading area
	
	`fmArea_Data`—data area
	
	`fmArea_Drillpath`—drill-path area of the table
	
	The diagram in Figure 7.2 on page 140 shows the location of the areas in an example table.
	
	■ *type*—type of position to return, which can be one of the following:
	
	`fmType_startRow`—position of the starting row of the specified area
	
	`fmType_endRow`—position of the ending row of the specified area
	
	`fmType_startColumn`—position of the starting column of the specified area
	
	`fmType_endColumn`—position of the ending column of the specified area
	
	`fmType_width`—width of the specified area, in terms of number of columns
	
	`fmType_height`—height of the specified area, in terms of number of rows
	
	`fmType_rowOffset`—number of rows before the start of this table (regardless of the area)
	
	`fmType_columnOffset`—number of columns before the start of this table (regardless of the *area* parameter)
	
	Returns: a value for the specified area and type. Row and column values are relative to the worksheet, not the table.
	
	The following example finds the worksheet positions of the start and end rows and columns in the data area of a table:
	
	`startRow = table.Position(fmArea_Data, fmType_startRow)`
	`endRow = table.Position(fmArea_Data, fmType_endRow)`
	`startColumn = table.Position(fmArea_Data, fmType_startColumn)`
	`endColumn = table.Position(fmArea_Data, fmType_endColumn)`

Method	Description

Function `Reallocate`(row as Long, column as Long) As fmWriteBackReturn	Reallocates the value at the cell at (*row*, *column*).
	
	It applies only to bottom-up form sets. The following conditions must be true:
	
	■ Either automatic allocation or writing to parent members of the Time dimension is enabled for the current data-entry table.
	■ The (*row*, *column*) parameters refer to a nonzero writable parent cell in the table. They are worksheet coordinates. Express the column value as a number (A=1, B=2, and so on).
	
	The method allocates the current value of the selected parent cell, according to the specified allocation weights. This option is useful if the parent cell value did not change, but you want the allocation to reflect other changes, such as a change in allocation weights, holds, or protection rules.
	
	For more information, see the descriptions of automatic allocation and the **Enable Hold Rules** table property in the *SAS Financial Management: Process Administrator's Guide*.
	
	Returns: the status of the operation. (See Table 7.3 on page 109.)
	
	Example:
	
	`Dim row As Long`
	`    Dim col As Long`
	`    row = 16`
	`    col = 3`
	`    rc = table.Reallocate(row, col)`
	`    MsgBox "return from reallocate: " & _`
	`      addin.enumString(fmEnums_fmWriteBackReturn, rc)`

Method	Description

Function `ReallocateTable()`	Reallocates the values at the table level in the same way that the reallocate table action works in the SAS Financial Management Microsoft Excel Add-In and in the SAS Financial Management web application.
	
	When using the ReallocateTable method, note the following:
	■ Stops at the first error.
	■ In a form template, honors visibility and cell protection rules that are defined in the model and form template, as well as any holds in the template.
	■ In a form, honors model- and template-level rules, as well as the items that are added in the form (for example, cell protection rules and holds).
	
	Example:
	
	`Dim conn As Connect' Connection object`
	`Dim addin As FMAddIn' FM Add-In object with events framework activated`
	`Public Sub APIReallocate()`
	`    Dim table As FMTable`
	`    Dim rc As fmWriteBackReturn`
	
	`    Set conn = Application.COMAddIns.Item("SASSESExcelAddIn.Connect").Object`
	
	`    Set addin = conn.FMAddIn`
	
	`    Set table = addin.Tables("NewTable0")`
	
	`    rc = table.ReallocateTable()`
	`    MsgBox rc`
	
	`End Sub`

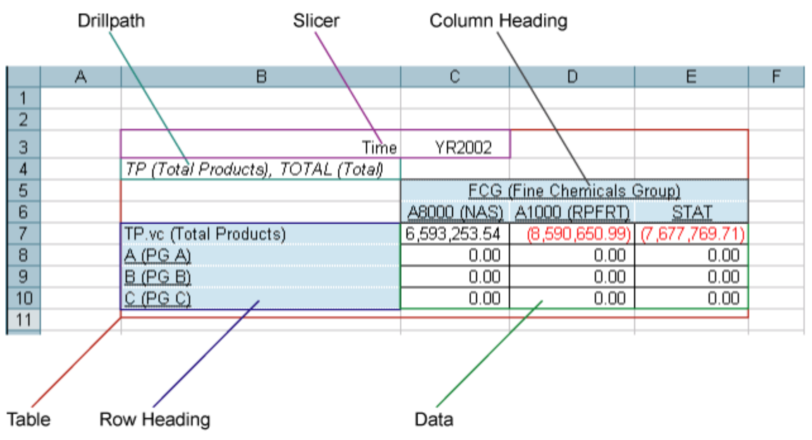
Sub `Refresh` (reQuery As Boolean)	Refreshes the table.
	
	Parameters:
	■ *reQuery*—requery flag. If **`True`**, the function performs a requery and repaint. If **`False`**, the function performs a repaint only.
	Consider carefully whether a requery is needed or whether a repaint is sufficient. The refresh operation requires more resources when a requery is included.
	
	The following example changes the model for the specified table and refreshes the table with a requery:
	
	`table.Model = "MyNewModel"`
	`table.Refresh(True)`
	
	The following example repaints the table without performing a requery:
	
	`table.Refresh(False)`

Sub `RemoveAllMemberCombinationFilters()`	Deletes all table filters.
Function `TargetHierarchy()` As FMHierarchy	Returns the target hierarchy for this table and model.

Method	Description
Sub `TransactionBegin`()	Begins accumulating transactions for later writeback using the BatchWrite method.
Sub `UnfilterMemberCombinations` (mems As FMMembersCollection)	Deletes the table filter that is specified by the combination of members. Parameters: ■ *mems*—collection of members.
Function `Write` (row As Long, column As Long, newValue As Double) As fmWriteBackReturn	Writes *newValue* to the crossing that is specified by *row* and *column*. Parameters: ■ *row* and *column*—row and column values (relative to the worksheet) that determine the crossing. For column values, convert letters to numbers (for example, cell **`A3`** is the crossing that is determined by a row value of `3` and a column value of `1`). ■ *newValue*—value to write. Returns: the status of the Write operation. For information about the Write operation, see Table 7.3 on page 109. **Note:** This method ignores the **Intelligent Writeback** option, even if it is enabled in the form's Global properties.
Function `Writeable` (row As Long, column As Long) As Boolean	Determines whether a specified crossing is writable. Read-only. Parameters: ■ *row* and *column*—worksheet row and column values that determine the crossing. For column values, convert letters to numbers. Returns: **`True`** if the crossing is writable; otherwise, **`False`**.

The following diagram illustrates the areas of a table. For more information about the areas of a table, see the Position method of the FMTable class.

Figure 7.2 *Areas of a Table*



	A	B	C	D	E	F
1						
2						
3		Time	YR2002			
4		TP (Total Products), TOTAL (Total)				
5			FCG (Fine Chemicals Group)			
6			A8000 (NAS)	A1000 (RPFRT)	STAT	
7		TP.vc (Total Products)	6,593,253.54	(8,590,650.99)	(7,677,769.71)	
8		A (PG A)	0.00	0.00	0.00	
9		B (PG B)	0.00	0.00	0.00	
10		C (PG C)	0.00	0.00	0.00	
11						

Table 7.17 *FMTable Class: Event Summary*

Event	Description

Event `AfterRefresh()`	Triggered after the table has been refreshed. This event might occur if the code calls the **`Refresh`** method of a table object, if the user selects **Refresh** or **RefreshAll** from the toolbar menu, or if the user performs some other manual action that triggers a refresh. If the user refreshes a worksheet, then the table refresh event is triggered, followed by the addin refresh event.
	
	The following is an example of an event handler for a table called **`table1`**:
	
	`Private Sub table1_AfterRefresh()`
	`  ' Assumes that screen updating has been disabled`
	`  ' in the BeforeRefresh event handler`
	`  ...`
	`  ' Perform some actions`
	`  ...`
	`  ' Re-enable screen updating`
	`  Application.ScreenUpdating = True`
	`End Sub`
	
	Notice that the name of the event handler includes the name of the object (in this case, **`table1`**). It applies only to this table, not to any other tables in the worksheet. If you wanted to affect a different table (for example, **`table2`**), you would write a second event handler, **`table2_afterRefresh()`**. To handle all table refresh events identically, you could use the **`addin_AfterTableRefresh`** event handler.

Event `BeforeRefresh()`	Triggered before the table is refreshed. One use for this event handler is to disable screen updating. You could re-enable screen updating in the AfterRefresh event handler. For more information, see the description of AfterRefresh.
	
	The following is an example of an event handler for a table called **`table1`**:
	
	`Private Sub table1_BeforeRefresh()`
	`  ' Disable screen updating`
	`  Application.ScreenUpdating = False`
	`  ' Perform some actions`
	`  ...`
	`End Sub`

The FMTablesCollection Class

The FMTablesCollection class represents a collection of tables. This class is a subclass of FMCollections.

The following properties are inherited from FMCollections: Count.

The following methods are inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

To create a new FMTablesCollection object, use the FMAddin.GetNewTablesCollection method.

The FMUser Class

The FMUser class contains information about the user who is currently logged on.

Table 7.18 FMUser Class Property Summary

Property	Description
Property `BudgetMode` As FMBudgetMode	Returns one of the following values: ■ **`fmBudgetMode_Create`**: if the user is editing a template ■ **`fmBudgetMode_Entry`**: if the user is editing a form Otherwise, the value is **`fmBudgetMode_None`**. Read-only.
Property `FormId` As Long	The form ID. Read-only.
Property `FormSetId` As Long	The form set ID. Read-only.
Property `FormSetName` As String	The name of the form set. Read-only.
Property `FormTemplateId` As Long	The form template ID. Read-only.
Property `LockName` As String	If the form is opened in data-entry mode, this property contains the name that is associated with the lock. The lock name can be viewed in the log. For information about the log, see "Activating the Log" on page 107.
Property `ReadOnly` As Boolean	This property applies only if a user has a form open. It has a value of **`True`** if the form is read-only. The user might have launched the form for viewing only, or the user might not have permission to edit the form. Read-only.
Property `UserContext` As String	Returns the user context (session ID). Read-only.
Property `UserId` As String	The user ID (for example, **`sasdemo`**). Read-only.
Property `UserName` As String	The user display name (for example, **`SAS Demo User`**). Read-only.

Property	**Description**
Property `WorkflowMethod` As String	The workflow method, which can be one of the following: **`TopDown`** or **`BottomUp`**. Read-only.

Including CDA Functions

To use CDA functions in your VBA code, create a string that is the CDA expression (using a function such as CDAGet or CDADesc).

The following example creates an expression to get the value of the ColorChoice custom property that is associated with the My_Product.TShirts dimension member. It inserts the CDA expression in the current worksheet.

```
Dim prop As String
Dim Target As Range
prop = "=CDAProperty(""My_Model"", ""My_Product"", ""TShirts"", ""ColorChoice"")"
Set Target = Cells(20, 3)
Target.Value = prop
```

8

Custom Analytics Stored Processes

Overview

This chapter describes the following topics:

- custom analytics stored processes
- writing the stored process
- registering the stored process
- migrating a customized stored process

About Custom Analytics Stored Processes

The Custom Analytics wizard enables users to select and run a custom stored process that is integrated with SAS Financial Management data. The stored process is available as part of the user interface for the SAS Financial Management Add-In for Microsoft Excel.

The stored processes that are available depend on the customizations at the site and the user's access permissions. An example might be a stored process that performs an analysis to determine the best allocation across a dimension.

This chapter explains how to create a custom stored process, and its metadata, to support this feature. This chapter refers to an example of a simple stored process.

Note: The following example is intended to illustrate only the concepts involved.

Writing the Stored Process

Example Stored Process

The following example of a stored process performs the following:

- extracts SAS Financial Management data based on account, analysis, and time members, as well as by variables and slicers
- adds 1 to fact values
- loads the data back into the database

```
/*--------------------------------------------------------------
*  Copyright (c) 2013 by SAS Institute Inc., Cary, NC, USA. All Rights Reserved.
*
*  PURPOSE:   Provide an example custom analytics stored process
*
*  Process includes:
*
*     - extracting SAS Financial Management data based on account,
*       analysis and time members; by variables and slicers
*
*     - adding 1 to fact values
*
*     - loading data back into SAS Financial Management
*
*
*-------------------------------------------------------------*/
 data _null_;
    put '<pre>';
 run;
 option validvarname=ANY;
 %global _FMErrorCode;
 %let _FMErrorCode=1; /* 0: error, 1:normal execution */
 %global today_dttm_val;
 %let today_dttm_val=%qsysfunc(datetime());
 %global cSDMLib;
 %let cSDMLib = _SDMLIB;
 %startsln(FMS);
 %bldLibNm(sesetl_sasLibRef=&cSDMLib
         , sesetl_OMRSASLibraryObjName=&Solsystem_folder./&SdmLibObj./&SdmLibObj.
         , sesetl_useType=BASE
         , sesetl_validateLib=Y
          ) ;
 %bldLibNm(sesetl_OMRSASLibraryObjName=&Solsystem_folder./&SdmLibObj./&SdmLibObj.
         , sesetl_useType=CON
          ) ;
 %global cSASData;
 %let cSASData = SASDATA;
 libname &cSASData "%sysfunc(pathname(SASDATA))";

 /********** Use these macro variables for debugging preferences *****************/
 %global cpmdbug;
```

```
%let cpmdbug = 0;        /* Set to 1 for debugging */
/* If debugging is enabled, a physical folder needs to be created
/* and a libname for this location needs to be assigned.                     */
/*                                                                               */
/* Example:                                                                      */
/*                                                                               */
/* 1) Create folder C:\CustomAnalytics                                           */
/* 2) Assign libname: libname debug 'C:\CustomAnalytics';                        */
/* 3) Set dbuglib macro to this libname:  %let dbuglib = debug;                  */
%global dbuglib;
%let dbuglib = ;
/*****************************************************************************************/
/* The cpm_xtrt_input macro extracts the data from the data mart.                         */
/*****************************************************************************************/
%cpmxtrt(input_dataset_name=work.inputds);
/*****************************************************************************************/
/* Add custom written code here, using the                                                */
/* dataset output by cpm_xtrt(&input_dataset_name).                                      */
/*****************************************************************************************/
/***** BEGINNING OF CUSTOM WRITTEN CODE *****/
%macro add_one;
  data _null_;
     /* FM_DBCSMULT is a global macro variable that is used in computing the length of a variable.
   FM_DBCSMULT is defined in startsln.sas and contains the number of bytes per character. */
     length tmp_dim_code $ %eval(32*&FM_DBCSMULT);
     length dimension_type_cd $ %eval(32*&FM_DBCSMULT);
     %do i = 1 %to &slicer_dim_codes_count;
        %let slicer_dtype_code_&i=;
        tmp_dim_code = "&&slicer_dim_codes&i";
        dimension_type_cd = put(tmp_dim_code, $fmt_dim_to_dtype.);
        call symput("slicer_dtype_code_&i", strip(dimension_type_cd));
     %end;
  run;
  data work.output_data;
     set work.inputds;
     fact_value = sum(value, 1);
     if account_code = "&input_account_1" and analysis_code = "&input_analysis_1"
        and time_code = "&input_time_1" then do;
            account_code = "&output_account_1";
            analysis_code = "&output_analysis_1";
            time_code = "&output_time_1";
     end;
     else if account_code = "&input_account_2" and analysis_code = "&input_analysis_2"
         and time_code = "&input_time_2" then do;
             account_code = "&output_account_2";
             analysis_code = "&output_analysis_2";
             time_code = "&output_time_2";
     end;
     else if account_code = "&input_account_3" and analysis_code = "&input_analysis_3"
         and time_code = "&input_time_3" then do;
             account_code = "&output_account_3";
             analysis_code = "&output_analysis_3";
             time_code = "&output_time_3";
     end;
     %do i = 1 %to &slicer_dim_codes_count;
        &&slicer_dtype_code_&i.._CODE = "&&slicer_write_member_code_&i";
     %end;
     drop value;
  run;
%mend add_one;
%add_one;
/***** END OF CUSTOM WRITTEN CODE *****/
/*************************************************************************************/
/* The cpm_load macro loads the output dataset (&output_data_set_name)              */
/* from the custom code into the SAS Financial Management Data Mart.                */
/*************************************************************************************/
%cpmload(output_dataset_name=work.output_data);
data _null_;
   put '</pre>';
run;
```

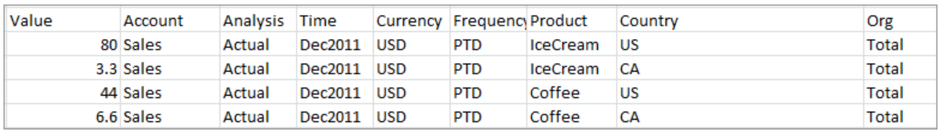
Customize the Example Code

To customize the example, complete the following steps:

1 In the call to %CPMXTRT, specify the data set to contain the results of the query:

```
%cpmxtrt(input_dataset_name=work.inputds);
```

This data set is the input for the custom code in the next step. It has the following format:



Value	Account	Analysis	Time	Currency	Frequenc	Product	Country	Org
80	Sales	Actual	Dec2011	USD	PTD	IceCream	US	Total
3.3	Sales	Actual	Dec2011	USD	PTD	IceCream	CA	Total
44	Sales	Actual	Dec2011	USD	PTD	Coffee	US	Total
6.6	Sales	Actual	Dec2011	USD	PTD	Coffee	CA	Total

Note: %CPMXTRT extracts only nonzero facts.

2 Modify the block of code between “BEGINNING OF CUSTOM CODE” and “END OF CUSTOM CODE”.

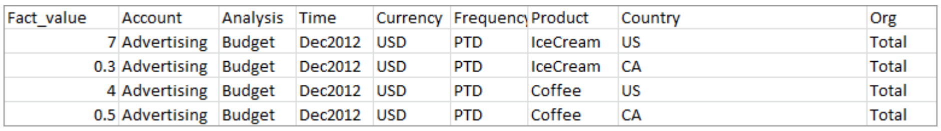
Read in the data from the input data set, process it, and output the results to the output data set:

```
data work.output_data;
      set work.inputds;
      /* process data here */
```

3 Call %CPMLOAD to prepare the output data from the previous step for writeback to the SAS Financial Management Data Mart. Specify the name of the data set to contain the crossings and values to be written back to the data mart:

```
%cpmload(output_dataset_name=work.output_data);
```

This data set should have the following format:



Fact_value	Account	Analysis	Time	Currency	Frequenc	Product	Country	Org
7	Advertising	Budget	Dec2012	USD	PTD	IceCream	US	Total
0.3	Advertising	Budget	Dec2012	USD	PTD	IceCream	CA	Total
4	Advertising	Budget	Dec2012	USD	PTD	Coffee	US	Total
0.5	Advertising	Budget	Dec2012	USD	PTD	Coffee	CA	Total

Parameters for the Stored Process

Overview

The stored process is defined by the following parameters:

Table 8.1 Stored Process Parameters

Parameter	Description
STORED_PROCESS_ DESCRIPTION_LABEL	Contains the text that is displayed when a user selects the **Custom Analytics** option in the SAS Financial Management Add-in for Microsoft Excel.
INPUT_ACCOUNT_1 INPUT_ACCOUNT_2 INPUT_ACCOUNT_3 OUTPUT_ACCOUNT_1 OUTPUT_ACCOUNT_2 OUTPUT_ACCOUNT_3	Members of the Account dimension
INPUT_ANALYSIS_1 INPUT_ANALYSIS_2 INPUT_ANALYSIS_3 OUTPUT_ANALYSIS_1 OUTPUT_ANALYSIS_2 OUTPUT_ANALYSIS_3	Members of the Analysis dimension
INPUT_TIME_1 INPUT_TIME_2 INPUT_TIME_3 OUTPUT_TIME_1 OUTPUT_TIME_2 OUTPUT_TIME_3	Members of the Time dimension
BY_VAR_DIM_CODE_1 BY_VAR_DIM_CODE_2	Dimension codes for By variables These parameters are not used in the example code, but they are available if you define them in the stored process metadata. **Note:** BY_VAR_DIM_CODE_1 is required for custom analytics.

When working with a stored process, note the following:

- The Account, Analysis, and Time dimensions are required. The members of the Account dimension define the variables under consideration. The same account members can be used for model input and output, or more than once for model input, if the crossings are qualified with time and analysis member combinations. Typically, a historical analysis-time combination would be used for the input, and a planning analysis-time combination would be used for the output.

- Parameter names must end with `_1`, `_2`, and so on. These suffixes are used to associate members from the Account dimension with members from the Time and Analysis dimensions.
- If your stored process supports it, you can define the Time prompts to allow multiple values.
- The actual number of input and output parameters that you define can vary. The example included in this section contains three of each.
- The By_VAR_DIM_CODE_1 variable is required. The By_VAR_DIM_CODE_2 variable is optional.

  A separate result set is generated for each selected By variable, which acts as a classification variable. The more members you select, the longer the stored process takes to execute.
- The _collection_url parameter must be included and correctly defined.

Other Dimensions

If a dimension is on the table but is not one of Account, Analysis, or Time, and is not used as a BY variable, the user is asked to make a slicer selection in the Custom Analytics wizard. If a dimension is off the table, its read default and write default members are supplied internally.

These dimension codes and member codes are available by means of the following variables.

Note: You do not define these variables as stored process parameters.

Variable	Description
`slicer_dim_codes_count`	The number of slicer dimensions.
`slicer_dim_codes`*`num`*	The set of variables that contain the slicer dimension codes. The variables are named `slicer_dim_codes1`, `slicer_dim_codes2`, and so on.
`slicer_write_member_code_`*`num`*	The set of variables that contain the slicer dimension write member codes. The variables are named `slicer_write_member_code_1`, `slicer_write_member_code_2`, and so on. If a dimension is on the table, the variable contains the member that was selected by the user in the Custom Analytics wizard. If a dimension is off the table, the variable contains the default write member for that dimension.

Variable	Description
`slicer_read_member_code_`*`num`*	The set of variables that contain the slicer dimension read member codes. The variables are named `slicer_read_member_code_1`, `slicer_read_member_code_2`, and so on. If a dimension is on the table, the read member is the same as the write member. If a dimension is off the table, it might have a different default read member.

Note: These dimensions are called slicer dimensions because only one member of each dimension is used in the stored process. The term does not refer to the dimension's position on (or off) the table.

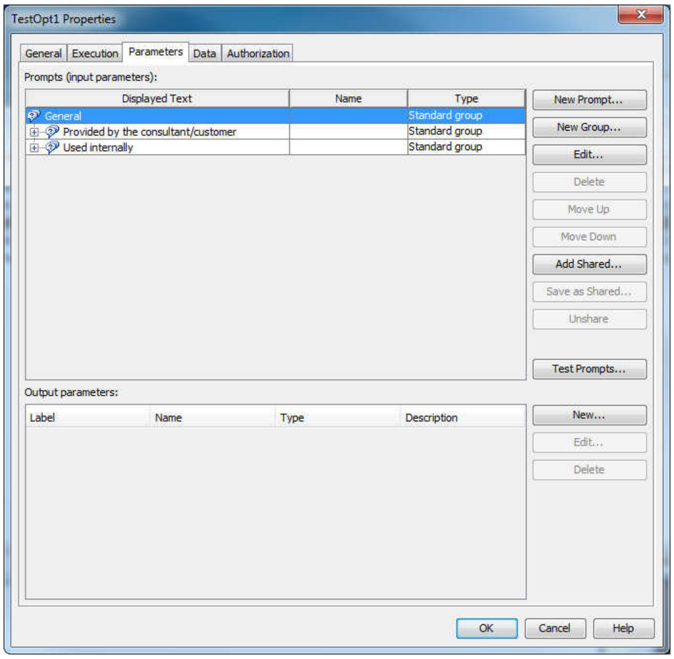
Registering the Stored Process

Note: You can complete the following configuration when adding a new stored process or by editing an existing stored process.

To register a stored process, complete the following steps:

1. In SAS Management Console, select the **Folders** tab.
2. In the **Folders** tree, select **Products** ▸ **SAS Financial Management** ▸ **Custom Analytics**.
3. Double-click on the name of stored process that you want to register. The Properties window for that stored process is displayed.
4. Select the Parameters table to display the stored process' input and output parameters.

Display 8.1 Stored Process Properties Window — Parameters View



Displayed Text	Name	Type
General		Standard group
Provided by the consultant/customer		Standard group
Used internally		Standard group

Label	Name	Type	Description

5 Add the parameters that are required by your stored process, such as STORED_PROCESS_DESCRIPTION_LABEL, INPUT and OUPUT parameters, and optional BY variable parameters.

Click **New Prompt** to the right of the **Prompt** pane to add input parameters.

Click **New** to the right of the **Output parameters** pane to add output parameters that are required by your stored process.

Note: When adding a new prompt, **Displayed Text** is the text that appears to the user in the Custom Analytics wizard. **Name** is the variable name that can be referenced in the stored process. You can also use other features of the prompting framework, such as assigning a default value.

6 Click **OK** to save your changes and close the stored process Properties window.

9 Configuring SSL

Overview

This chapter describes the following topics:

- installing and configuring SSL for SAS Financial Management
- importing the certificate into your browser
- importing the certificate to client machines

Installing and Configuring SSL for SAS Financial Management

Deploying SAS Financial Management 5.5 with one-way SSL is optional. If you deploy SAS Financial Management with one-way SSL, the SSL configuration is primarily automated during installation by the SAS Deployment Wizard. However, you might need to manually complete the following post configuration tasks after installation:

- import the server's certificate into your browser's trusted signer's area
- configure desktop clients for use with an SSL-enabled server by importing the certificate to the client machines, if necessary

Note: This chapter assumes that you have a basic understanding of one-way Secure Socket Layer (SSL) and that you know how to obtain and use trusted certificates.

For information about OpenSSL, see http://www.openssl.org.

Importing the Certificate into Your Browser

Import the server's certificate into your browser's trusted signer's area so that the browser knows that it can trust the certificate.

Note: A self-signed certificate must be in the Windows System trust store, not the private JRE trust store.

For detailed instructions about how to import the certificate into your browser, consult the Help for your browser.

Importing a Certificate to Client Machines

Note: The steps in this section are required only if a customer site uses a self-signed certificate that is not issued by a trusted root authority. If a customer site deploys a certificate issued by a trusted root authority, the steps in this section are not required.

We recommend that a customer site deploys a certificate issued by a trusted root authority on the server. Because the certificate is signed, there are no issues with any client, and you do not need to modify any client component to trust the connection.

However, if a self-signed certificate that is not issued by a trusted third-party certificate authority (CA) is used, you must update the client's Java Runtime Environment (JRE) to import the certificate. Importing the unsigned certificate enables the client to communicate over SSL to the test configuration on the middle-tier server.

To import a certificate to a client machine, complete the following steps:

1. On the client machine, locate the **`lib\security`** directory of the JRE that SAS Financial Management Studio uses.

   Note: Typically, SAS Financial Management Studio uses the JRE on the client tier that is provided by SAS and installed by the SAS Deployment Wizard. However, you can configure SAS Financial Management Studio to use a different JRE by adding a -vm option to the SAS Financial Management Studio .ini file. For more information about the JRE used by client applications, see "Installing Client Applications" in the *SAS Financial Management: Installation and Configuration Guide*.

2. Copy the certificate file from the middle-tier server to the **`lib\security`** directory on the client machine.

3. On the client machine, open a command prompt window and change the directory to the **`lib\security`** directory.

4. Execute the following command:

   ```
   ..\..\bin\keytool.exe -import -alias alias -file certificate-name
   -keystore cacerts
   ```

where *certificate-name*.cer is the name of the certificate file that you copied from the middle-tier server.

5 The keytool program prompts you for a password. The default password is **`changeit`**.

6 When the **Trust this certificate?** prompt is displayed, specify **`Y`**.

The keytool program displays a message confirming that the certificate was imported.

Index