Overview

The Form Data Model class is part of a collection of classes that combine to create the Data Form and Data Table classes. These classes work together to enable you to

- view and edit SAS data files
- customize the functionality of these classes for your own application development needs.

Before you customize the functionality of these classes, you should understand the structure of the classes that underlie the Data Form and Data Table classes and how they communicate.

The Form Data Model class defines the communication protocol between the Form Editor class and its associated data model class.

Any model that has its data viewed using the Form Editor class must provide the methods defined by the Form Data Model class in order to give the form editor access to the data. You accomplish this by defining your model to be a subclass of the Form Data Model class and by defining the methods that are meaningful to your model. Note that none of the Form Data Model methods are required. For example, attaching the form data model itself to a form editor produces an empty form because the Form Data Model class does not own any data.

Model classes that need to be able to communicate with the table editor must also be subclasses of the Table Data Model class, which is a subclass of the Form Data Model class. For more information, see the Table Data Model class.

Model classes that only need to be used with the form editor can be direct subclasses of the Form Data Model class.

Parent:

sashelp.fsp.object.class

Class:

sashelp.fsp.Form_m.class
Using the Form Data Model Class

Drag and Drop and the Form Data Model

The drag and drop functionality is inherited from the Widget class and its functionality is extended by the Form Data Model class. During a drag and drop action involving the attached viewer, a sequence of methods is called on the viewer. For more general information about the drag and drop operations, see SAS/AF online help. The following methods are forwarded to the model for processing:

- _completeDrag
- _drop
- _getDragData
- _validateDropData.

The following drag-and-drop methods are used by the attached viewer to query the current drag and drop representations and actions the viewer should set on itself.

- _getDragInfo
- _getDropInfo

DATAFORM Catalog Entries

The Form Data Model class contains the functionality for reading from and writing to DATAFORM catalog entries. DATAFORM catalog entries are used to save the customizations of a model or a viewer connected to a model. Because DATAFORM entries are not tied directly to a particular object, multiple viewers can use the same DATAFORM entry. DATAFORM entries can also share customization information between a data form and a data table and between a form editor and a table editor.

You can make a data form in a frame and specify an associated DATAFORM catalog entry. You can then make some customizations, such as removing a column from the display. When you end from the frame or run the frame, this information is saved with the DATAFORM catalog entry. You can then make a data table in a separate frame and specify the same DATAFORM catalog entry, and the column you had removed is not displayed.

Methods

Methods specific to the Form Data Model class are described here. Inherited methods are described in the Object class.

Because the Form Data Model class is an abstract class, most of the Form Data Model methods are abstract methods; that is, the methods are defined but disabled at the Form Data Model level. These methods should be overridden by the subclasses of Form Data Model in order to make a useful model and provide model-specific functionality.
Dictionary

_attach

Connects the model to a viewer

Syntax

CALL SEND (model-id, '_attach', viewer-id);

Argument Type Description
viewer-id N specifies the object identifier of the viewer that is calling the model's _attach method

Details

The _attach model method is used to notify the model that the viewer is displaying the model. For example, when the viewer receives an _attach method, it calls the _attach model method.

The Form Data Model _attach method connects the model to the viewer so that the model can call viewer methods when necessary.

__completeDrag

Completes a viewer's drag operation

Syntax

CALL SEND (model-id, '_completeDrag', rep, op, data, code);

Argument Type Description
rep C specifies the representation selected for the drop
op C specifies the operation selected for the drop
data N specifies an SCL List identifier that contains the data defined by the representation
code C returns the completion code of the _validateDropData method:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ACCEPT'</td>
<td></td>
<td>successful completion</td>
</tr>
<tr>
<td>'CANCEL'</td>
<td></td>
<td>canceled completion</td>
</tr>
</tbody>
</table>

**Details**

The `_completeDrag` method is called automatically when a drag operation has been completed on the attached viewer and the viewer does not recognize the rep passed to it. The values passed for rep and code determine how the drag operation is completed. For example, a subclass of the Form Data Model is the Data Set Data Model, which understands a drag representation of `_dndDATASET`. This representation means to create a new table and pass the name of table as the drag data. If a rep of `_dndDATASET` was passed to an instance of Data Set Data Model and code was 'ACCEPT', no action would take place. If code was 'CANCEL', the instance of Data Set Data Model would need to delete the table that it created.

As another example, suppose a user is attempting to move rows from one data table to another. The user would use `_completeDrag` to delete the rows being moved if code was 'ACCEPT'.

For general information about drag and drop operations, see SAS/AF online help.

---

**_detach**

Disconnects the model from a viewer

**Syntax**

CALL SEND (model-id, '_detach');

**Details**

The `_detach` model method is used to notify the model that the viewer is no longer displaying it. For example, the `_detach` model method is called by the viewer when the viewer receives a `_detach` method. The `_detach` model method disconnects the model from the viewer to which it is attached.

*Note:* Calling `_detach` does not terminate the object.

---

**_drop**

Completes a viewer’s drop operation

**Syntax**

CALL SEND (model-id, '_drop', rep, oper, data, row, column);
Form Data Model Class

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep</td>
<td>C</td>
<td>specifies the representation selected for the drop</td>
</tr>
<tr>
<td>oper</td>
<td>C</td>
<td>specifies the operation selected for the drop</td>
</tr>
<tr>
<td>data</td>
<td>N</td>
<td>specifies an SCL list identifier that contains the data defined by the representation</td>
</tr>
<tr>
<td>row</td>
<td>N</td>
<td>specifies the row coordinates list where the drop occurred</td>
</tr>
<tr>
<td>column</td>
<td>N</td>
<td>specifies the column coordinates list where the drop occurred</td>
</tr>
</tbody>
</table>

Details

The _drop method method is called automatically after a drag object has been dropped on the attached viewer and the viewer does not recognize the rep passed to it. The _drop method is passed the data returned from the call to _getDragData. The _drop method is the last method called in the drop sequence.

If you are using a viewer supplied with SAS/AF software, such as the data table or data form, the location of the drop is automatically translated for you by the attached viewer into coordinates that make sense for that viewer. For example, when the attached viewer is a data table, a table editor, or a subclass of the table editor, the location is translated into cell coordinates. If the attached viewer is a data form, a form editor, or a subclass of the form editor, the location of the drop is the pixel offset measuring from the upper left corner of the the form editor.

Note: If you are using a user-defined viewer, you must translate the pixel offset into units that make sense for your viewer.

For general information about drag and drop operations, see SAS/AF online help.

_execCmd

Processes a model command

Syntax

CALL SEND (model-id, '_execCmd');

Details

The _execCmd method is called for commands that are unrecognized by the viewer. Subclasses of the Form Data Model should perform the necessary operations for commands that you define for that class.

Note that the command is not passed as an argument. Rather, it processes commands that are sent to the object.

For more information on command processing, see the Widget class.
_getActions

Returns a list of items in the viewer’s pop-up

Syntax

CALL SEND (model-id, '_getActions', actions, map, row-addr, col-addr);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actions</td>
<td>N</td>
<td>specifies the identifier of an SCL list that will contain the pop-up actions to be displayed. This list is initially filled with the viewer’s actions. To add model items, insert the model actions into this list. The model can add and delete items in this list if corresponding changes are made in the map. The items in the actions list are not named. The value for each item is the text that will be displayed in the pop-up. Turn on the INACTIVE list item attribute for items that should be in the pop-up but should be grayed.</td>
</tr>
<tr>
<td>map</td>
<td>N</td>
<td>specifies the identifier of an SCL list that will contain the current map of pop-up actions to objects and methods. This list is initially filled with the viewer’s map. To add model items, insert the model map into this list. The model can add and delete items in this list if corresponding changes are made in the actions. For each item in actions, map should contain two items: 1) the object identifier to which the action method should be sent 2) the name of the method to be sent to the object identifier. The items in the map list are not named. The model’s identifier should precede the associated method name. Note that methods that require arguments cannot be called from the pop-up.</td>
</tr>
<tr>
<td>row-addr</td>
<td>N</td>
<td>specifies the row coordinates list of the cell that contains the pop-up event.</td>
</tr>
<tr>
<td>col-addr</td>
<td>N</td>
<td>specifies the column coordinates list of the cell that contains the pop-up event.</td>
</tr>
</tbody>
</table>

Details

The pop-up displayed by the viewer is a combination of items that the viewer puts on the pop-up and items obtained from the model on a call to the _getActions method. When a pop-up occurs, the _getActions method is called by a viewer connected to a model in order to retrieve the items that the model wants placed on the pop-up list.

Note: For every item added to the actions list, you must add two items to the map list. See the description of map in the previous table for more information.
Example

In the following example, the _getActions method is implemented for a model subclass of Form Data Model, such as Data Set Data model, to add two new items, Add Row and Copy Row to the pop-up menu. Copy Row is grayed. In this model subclass, the _addRow and _copyRow methods are supported.

GETACT: method actions map row col 8;

If you don’t want the default model actions, delete the CALL SUPER.

call super(_self_, 'get_actions',
           actions, map, row, col);

Insert Add Row into the actions list.

actions = insertc(actions, 'Add Row', -1);

Insert Copy Row into the actions list and mark it INACTIVE.

actions = insertc(actions, 'Copy Row', -1);
           setlattr(actions, 'INACTIVE',-1);

Insert Add Row into the map list. When the Add Row action item is selected, the _addRow method is sent to _SELF_.

map = insertn(map, _SELF_, -1);
           map = insertc(map, '_addRow', -1);

Insert Copy Row into the map list.

map = insertn(map, _SELF_, -1);
           map = insertc(map, '_copyRow', -1);

endmethod;

For the Data Set Data Model subclass, you would not have to add Add Row or Copy Row to the actions list; they are already defined. Add Row and Copy Row are used in this example to illustrate how you would add your own user-defined actions and methods.
**_getColumnInfo**

Returns the attributes of a column

**Syntax**

CALL SEND (model-id, '_getColumnInfo', data-vector-id);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-vector-id</td>
<td>N</td>
<td>specifies the object identifier of an instance of the Row/Column Data Vector class</td>
</tr>
</tbody>
</table>

**Details**

The _getColumnInfo method requests information about one column. The data vector filled in by the _getColumnInfo method is an instance of the Row/Column Data Vector class. The data vector instance is created by the viewer. The _getColumnInfo method fills in the data vector’s information for the current column. The data vector’s information can be manipulated with calls to methods of the Row/Column Data Vector class that set data vector values. These methods are described in the Row/Column Data Vector class.

**_getData**

Returns the data and data attributes for a given row

**Syntax**

CALL SEND (model-id, '_getData', data-vector-id, numcols);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-vector-id</td>
<td>N</td>
<td>specifies the object identifier of an instance of the GET_DATA Data Vector class</td>
</tr>
<tr>
<td>numcols</td>
<td>N</td>
<td>specifies the number of columns of data the viewer is requesting</td>
</tr>
</tbody>
</table>

**Details**

The viewer calls the _getData method to obtain the data to display for a particular row. The _getData method can request information about one or more columns. The data vector filled in by the _getData method is an instance of the GET_DATA Data Vector class. The viewer creates the data vector instance.
You can fill the data vector's information with calls to methods of the GET_DATA Data Vector class that set data vector values. These methods are described in the GET_DATA Data Vector class.

---

**getDataformProperties**

Returns the information about the model to be saved in the DATAFORM entry

**Syntax**

CALL SEND (model-id, '_getDataformProperties', list);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>N</td>
<td>specifies the identifier of an SCL list that will contain the model-specific properties to be saved to the DATAFORM catalog entry</td>
</tr>
</tbody>
</table>

**Details**

The _getDataformProperties method can be called from the Form Data Model's _writeEntry method. The _getDataformProperties method is responsible for returning all model information that should be saved with the Form Data Model's DATAFORM catalog entry. This returned list is saved in the DATAFORM catalog entry on the _writeEntry call so that other users of that DATAFORM entry can use this information.

The _getDataformProperties method fills in list by calling the _getProperties method on itself. After filling this list, Form Data Model marks as NOWRITE the items from that list that it does not want saved in the DATAFORM catalog entry.

Therefore, by default the _getDataformProperties method for subclasses of the Form Data Model class will save any information on the subclass's instance variable list that are not marked as NOWRITE by the Form Data Model method call.

If there are items on the subclass instance variable list that should not be saved with the DATAFORM catalog entry, the subclass should mark those items as NOWRITE. To do so, override the _getDataformProperties method in the subclass of Form Data Model, and then perform a CALL SUPER to update the list. Then mark the items that you want as NOWRITE.

---

**getDragData**

Exports the data from the model

**Syntax**

CALL SEND (model-id, '_getDragData', rep, oper, data, srow, scol, erow, ecol);
### Argument Type Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep</td>
<td>C</td>
<td>specifies the representation selected for the drop</td>
</tr>
<tr>
<td>oper</td>
<td>C</td>
<td>specifies the operation selected for the drop</td>
</tr>
<tr>
<td>data</td>
<td>N</td>
<td>returns the identifier of an SCL list that contains the data defined by the representation</td>
</tr>
<tr>
<td>srow</td>
<td>N</td>
<td>specifies the row coordinates list of the starting row of the drag</td>
</tr>
<tr>
<td>scol</td>
<td>N</td>
<td>specifies the column coordinates list of the starting column of the drag</td>
</tr>
<tr>
<td>erow</td>
<td>N</td>
<td>specifies the row coordinates list of the ending row of the drag</td>
</tr>
<tr>
<td>ecol</td>
<td>N</td>
<td>specifies the column coordinates list of the ending column of the drag</td>
</tr>
</tbody>
</table>

### Details

The _getDragData method is called automatically after a drag object from the attached viewer has been dropped and the viewer does not recognize the rep passed to it. The _getDragData method runs before the _validateDropData method runs. This method provides the model with a way to export data in a format that depends on the passed representation to the drop site object. The data list passed to _getDragData is a global list to facilitate passing data between tasks; any lists added to data should be global lists. The data list will automatically be deleted for you.

The srow, scol, erow, and ecol parameters are passed as coordinates of the location of where the drag started. If you are using a viewer supplied with SAS/AF software, such as the data table or data form, these coordinates are automatically translated for you by the attached viewer into coordinates that make sense for the attached viewer. For example, when the attached viewer is a data table, a table editor, or a subclass of the table editor, the coordinates are translated into cell coordinates of the selected area that was dragged. If the attached viewer is a data form, a form editor, or a subclass of the form editor, srow and erow have the same value and scol and ecol have the same value. The value of these pairs of parameters (srow and scol, and erow and ecol) indicates the pixel offset measuring from the top left corner of the form editor to where the drag started.

Note: If you are using a user-defined viewer, you must translate the pixel offset into units that make sense for your viewer.

For general information about drag and drop operations, see SAS/AF online help.

### _getDragInfo

Queries the model for its drag information

### Syntax

CALL SEND (model-id, '_getDragInfo', info-list);
## _getDragInfo

Queries the model for its drag information

### Syntax

```sql
CALL SEND (model-id, '_getDragInfo', info-list);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info-list</td>
<td>N</td>
<td>specifies the identifier of an SCL list that will contain the drag information. The drag information list should contain a sublist named REPS, where the items in the REPS sublist represent the supported drag representations. The drag information list should also contain a sublist named OPS, where the items on the OPS sublist represent the valid operations (COPY, LINK, MOVE).</td>
</tr>
</tbody>
</table>

## Details

The _getDragInfo method is called when the attached viewer needs to reset its drag representation based on a user’s action. For example, when the table editor is the attached viewer, each time the user selects (or highlights) a new block of cells, the table editor calls the _getDragInfo method to see if the drag representation changed based on the selected area.

For general information about drag and drop operations, see SAS/AF online help.

## _getDropInfo

Queries the model for its drop information

### Syntax

```sql
CALL SEND (model-id, '_getDropInfo', info-list);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info-list</td>
<td>N</td>
<td>specifies the identifier of an SCL list that will contain the drop information. The drop information list should contain a sublist named REPS, where the items in the REPS sublist represent the supported drop representations. The drop information list should also contain a sublist named OPS, where the items on the OPS sublist represent the valid operations (COPY, LINK, MOVE). By default, the table editor always adds MOVE as a valid operation.</td>
</tr>
</tbody>
</table>

## Details

The _getDropInfo method is called when an instance of this class is attached to a viewer. For general information about drag and drop operations, see SAS/AF online help.

## _getEntry

Returns the name of the DATAFORM catalog entry associated with the model
Syntax
CALL SEND (model-id, '_getEntry', entry);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>C</td>
<td>returns the name of the DATAFORM catalog entry associated with the model</td>
</tr>
</tbody>
</table>

Details
The DATAFORM catalog entry name can be a two-level or four-level name. The _getEntry method returns a blank if no DATAFORM catalog entry name is specified.

_getMaxcol

Returns the number of columns (excluding hidden columns)

Syntax
CALL SEND (model-id, '_getMaxcol', numcols);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numcols</td>
<td>N</td>
<td>returns the number of nonhidden columns in the model:</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>the number of columns is unknown</td>
</tr>
<tr>
<td>≥0</td>
<td></td>
<td>the number of columns</td>
</tr>
</tbody>
</table>

Details
The _getMaxcol method returns the number of nonhidden columns in the model. Columns allocated by the _setNumberOfColumns method may be hidden and therefore are not returned to the viewer on a _getMaxcol call.

_getMaxrow

Returns the number of rows

Syntax
CALL SEND (model-id, '_getMaxrow', numrows);
**_getMsgLevel_**

Returns the current message level

**Syntax**

CALL SEND (object-id, ‘_getMsgLevel’, error_flag, warning_flag, note_flag);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_flag</td>
<td>C</td>
<td>returns whether error messages from this class are to be displayed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘Y’ error messages are displayed (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘N’ error messages are not displayed</td>
</tr>
<tr>
<td>warning_flag</td>
<td>C</td>
<td>returns whether warning messages from this class are to be displayed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘Y’ warning messages are displayed (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘N’ warning messages are not displayed</td>
</tr>
<tr>
<td>note_flag</td>
<td>C</td>
<td>returns whether notes from this class are to be displayed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Y' notes are displayed (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'N' notes are not displayed</td>
</tr>
</tbody>
</table>
### _getNumberOfColumns

**Returns the number of columns**

**Syntax**

```
CALL SEND (model-id, '_getNumberOfColumns', numcols);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numcols</td>
<td>N</td>
<td>returns the number of columns in the model</td>
</tr>
</tbody>
</table>

**Details**

The `_getNumberOfColumns` method returns the number of columns set with the `_setNumberOfColumns` method.

### _getProperties

**Returns the current state**

**Syntax**

```
CALL SEND (model-id, '_getProperties', list);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>N</td>
<td>specifies the identifier of an SCL list that will contain the model-specific properties</td>
</tr>
</tbody>
</table>

**Details**

The `_getProperties` method is used to save the model’s state. The list returned can be sent to the `_setProperties` method or to the `_new` method.
When the viewer needs to save the most up-to-date information about the model on its MODEL_INFO list, you can call the `_getProperties` model method from the viewer's `_getProperties` method to provide that information. The `_getProperties` method is responsible for returning all model information. Note: Some of the items in the list are marked NOWRITE and are not written out when you execute a SAVELIST.

The `_getProperties` method fills in list with a copy of the instance variables of the model. Therefore, by default the `_getProperties` method for subclasses of the Form Data Model class saves any information on the subclass's instance variable list.

If there are items for the subclass that are not on the instance variable list but should be saved with the DATAFORM catalog entry, the subclass can still add these items to the list. To do so, the subclass of Form Data Model should override the `_getProperties` method by performing a CALL SUPER to the Form Data Model's `_getProperties` method.

The model's `_setProperties` method is then responsible for taking the items it needs from the list.

---

**_getRowInfo**

*Returns the attributes of a row*

---

**Syntax**

CALL SEND (model-id, '_getRowInfo', data-vector-id);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-vector-id</td>
<td>N</td>
<td>specifies the object identifier of an instance of the Row/Column Data Vector class</td>
</tr>
</tbody>
</table>

**Details**

The `_getRowInfo` method requests information about one row.

The data vector filled in by the `_getRowInfo` method is an instance of the Row/Column Data Vector class. The viewer creates the data vector instance.

The `_getRowInfo` method should fill in the data vector's information for the current row. You can manipulate the data vector's information with calls to methods of the Row/Column Data Vector class, which sets data vector values. These methods are described in the Row/Column Data Vector class.

---

**_getSource**

*Returns the name of the source entry associated with the model*

---

**Syntax**

CALL SEND (model-id, '_getSource', source);
Argument | Type | Description
---|---|---
source | C | returns the name of the source entry associated with the model

**Details**
For more information, see the Data Set Data Model class.

__getUpdateStatus__

Returns whether updates are allowed

**Syntax**
CALL SEND (model-id, '_getUpdateStatus', status);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>N</td>
<td>returns the update status of the model:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 update is allowed (the default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 update is not allowed</td>
</tr>
</tbody>
</table>

**Details**
If _getUpdateStatus returns a value of 1, then the following methods are not called:

- _getRowInfo
- _getRowDimInfo
- _getData
- _getColumnInfo
- _getColumnDimInfo

For example, if you attempt an update while in the _setDataset method, the Data Set Data Model subclass returns a value of 1.

__hscroll__

Returns whether a horizontal scroll should be allowed by the viewer

**Syntax**
CALL SEND (model-id, '_hscroll', amount, return-code);
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>N</td>
<td>specifies the number of columns to be scrolled:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values &gt;0 indicate scrolling to the right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values &lt;0 indicate scrolling to the left</td>
</tr>
<tr>
<td>return-code</td>
<td>N</td>
<td>returns whether scrolling should be allowed by the viewer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 scrolling is allowed (the default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 scrolling should not be allowed</td>
</tr>
</tbody>
</table>

**Details**

The viewer calls the _hscroll method when a horizontal scroll is about to occur to make sure that scrolling should be allowed by the viewer.

---

**_init**

Initializes the model

---

**Syntax**

CALL SEND (model-id, '_init', model-list);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| model-list   | N    | specifies the identifier of an SCL list that will contain the information the model should use when initializing. If this list is specified and it contains a named item called "NORUNSCL" set to 1, the model will not execute SCL code.

**Details**

The _init method is called whenever you make an instance of the Form Data Model class, either through a call to _new or to the instance function. The _init method does initialization for the Form Data Model class. It also calls the _init method of the parent class.

If your model subclass of the Form Data Model class needs to do initialization, it should override the _init method. When you override the _init method for your model subclass, call the _init method of the parent class (by using CALL SUPER) before adding your own processing so that the Form Data Model _init method can run first.

The model-list list is optional. If the NORUNSCL item is on this list and is set to 1, the model's SCL labels will not run even if there is an SCL entry specified for this model. For example, you can use this functionality when you want a viewer to instantiate a model at build time without executing any SCL code.
_preterm

Verifies that the model can terminate properly

Syntax

CALL SEND (model-id, '_preterm', return-code);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return-code</td>
<td>N</td>
<td>returns a code indicating to the caller whether normal termination should be allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not 0</td>
</tr>
</tbody>
</table>

Details

The viewer calls the model's _preterm method from the viewer's _preterm method. The viewer's _preterm method is called immediately before the frame SCL's term label runs.

If return-code is set to 0 by the model _preterm method, the viewer assumes that normal termination is allowed. If return-code is set to a nonzero value by the model _preterm method, the viewer assumes that normal termination is not allowed and sets status to resume. This means that the frame does not terminate but continues execution. For example, the model should not be configured to allow termination if there are data in error.

_readEntryIntoList

Returns the model's information

Syntax

CALL SEND (model-id, '_readEntryIntoList', list<update>);
Details

The _readEntryIntoList method saves the model’s state. The model can later be updated from the information returned by the _readEntryIntoList method by calling the _writeEntryFromList method.

The _readEntryIntoList method fills in a list with named items corresponding to the current model information. Note that the _readEntryIntoList method fills in the list from the model’s current information, which is not necessarily the information from the model’s DATAFORM entry, because customizations may have been performed on the model since _writeEntry was called.

_changesColumn

Changes the column width

Syntax
CALL SEND (model-id, '_resizeColumn', col-address, width, old-width);

Details

The viewer calls the _resizeColumn method to notify the model that the user has resized a column. This resize is considered a hard resize as opposed to a soft(ware) resize. For more information, see the Table Data Model class.

The column width should be stored by the model and returned as a fixed width whenever this column’s attributes are queried.

_changesColumnDim

Changes column dimension height
Syntax
CALL SEND (model-id, '_resizeColumnDim', col-address, height, old-height);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-address</td>
<td>N</td>
<td>specifies the identifier of an SCL list that contains the relative address of the column that was resized.</td>
</tr>
<tr>
<td>height</td>
<td>N</td>
<td>specifies the new height of the column dimension, in points</td>
</tr>
<tr>
<td>old-height</td>
<td>N</td>
<td>specifies the old height of the column dimension, in points</td>
</tr>
</tbody>
</table>

Details
The viewer calls the _resizeColumnDim method to notify the model that the user has resized a column dimension. This is considered a hard resize as opposed to a soft(ware) resize. For more information, see the Table Data Model class.

This height should be stored by the model and returned as a fixed height whenever this dimension’s attributes are queried.

_setColumnInfo
Sets the attributes of a column

Syntax
CALL SEND (model-id, '_setColumnInfo', data-vector-id);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-vector-id</td>
<td>N</td>
<td>specifies the object identifier of an instance of the Row/Column Data Vector class</td>
</tr>
</tbody>
</table>

Details
The _setColumnInfo method sets information about one column.

The data vector queried by the _setColumnInfo method is an instance of the Row/Column Data Vector class. The viewer creates and sets up the data vector instance.

You can manipulate the data vector’s information with calls to methods of the Row/Column Data Vector class, which get data vector values. These methods are described in the Row/Column Data Vector class.
**_setData**

Sets the data and data attributes for a given row

---

**Syntax**

CALL SEND (model-id, '_setData', data-vector-id, numcols);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-vector-id</td>
<td>N</td>
<td>specifies the object identifier of an instance of the GET_DATA Data Vector class</td>
</tr>
<tr>
<td>numcols</td>
<td>N</td>
<td>specifies the number of columns for which to set data</td>
</tr>
</tbody>
</table>

**Details**

The _setData method is called by the viewer to set the data for a particular row. The _setData method can set information about one or more columns.

The data vector queried by the _setData method is an instance of the GET_DATA Data Vector class. The data vector instance is created and set up by the viewer.

The data vector's information can be retrieved with calls to methods of the GET_DATA Data Vector class, which get data vector values. These methods are described in the GET_DATA Data Vector class.

---

**_setEntry**

Sets the name of the DATAFORM catalog entry associated with the model

---

**Syntax**

CALL SEND (model-id, '_setEntry', entry<, return-code<, replace>>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>C</td>
<td>specifies the name of the DATAFORM catalog entry associated with the model</td>
</tr>
<tr>
<td>return-code</td>
<td>N</td>
<td>contains the return code from _setEntry:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 entry does not yet exist and previously specified entry does exist. Result: A DATAFORM catalog entry by the new name is created on the next _writeEntry call.</td>
</tr>
</tbody>
</table>
Argument | Type | Description
--- | --- | ---
2 | entry could not be opened for writing; the entry exists but the user does not have WRITE access to it. Result: The DATAFORM catalog entry is successfully read, but calls to the _writeEntry method do not save any customizations to that DATAFORM entry.
1 | entry does not yet exist and previously specified entry does not exist. Result: A DATAFORM catalog entry by the new name is created on the next _writeEntry call.
0 | entry successfully opened
-1 | invalid library or entry name Result: The model is not changed at all by the _setEntry call.
-2 | discrepancy between the column information in the DATAFORM catalog entry and that in the model information. This situation can happen if the DATAFORM entry contains a column that is noncomputed and does not currently exist in the model. Result: The new noncomputed column from the entry is not added to the model.
-3 | out-of-memory condition
-4 | there is already a column in the model by the same name as a column in the entry and the two like-named columns are of different type. Result: The new column from the entry is not moved to the model; the existing model column remains unchanged.
-5 | both the conditions described above for values of -2 and -4 occurred (discrepancy between the columns, and there is already a column in the model by the same name as a column in entry and the existing columns are of different types)

replace C specifies whether to replace the existing columns with the columns from the new entry:

‘Y’ replace the existing columns with columns from the new entry (default)
‘N’ do not replace the existing columns; the columns from the new entry are merged with the existing columns.

Details
The DATAFORM catalog entry name can be a two-level or four-level name.
You can use the optional return-code argument to determine if the _setEntry call was successful.
If there is a column in the entry that is not in the model, it will be added to the model only if the new column is a computed column. Otherwise, the new column will not be added (see -2 above).
If there is a column in the entry that has the same name as a column in the model, the column from the entry will be added to the model if the new column and the model column are both of the same type. Otherwise, the new column will not be added (see -4 above).
**_setMsg**

Displays a message on the message line, the message window, or in the log.

**Syntax**

CALL SEND (object-id, '_setMsg', string);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>C</td>
<td>specifies the message to be displayed on the message line, the message window, or in the log</td>
</tr>
</tbody>
</table>

**Details**

The message is displayed on the message line of the attached viewer's frame unless one or more of the following conditions exist:

- the object is not currently attached to a viewer
- two or more messages have been issued since the last window refresh
- the message level has been set to ignore this type of message.

In the first two cases, the message will be displayed in the SAS log. If the message level has been set to prevent that type of message from being displayed (for example, if the message begins with ERROR:, NOTE:, or WARNING:, and that message type is turned off), then the message will not be displayed.

If you want to display custom error messages one at a time on the message line such as in the case where multiple columns are in error, you can have the last message written to the message line in most situations by setting the _msg system variable to your message text instead of making multiple calls to _setMsg.

**_setMsgLevel**

Specifies whether to issue error, warning, and note messages.

**Syntax**

CALL SEND (object-id, '_setMsgLevel', error_flag<, warning_flag<, note_flag>>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| error_flag   | C    | specifies whether error messages from this class are to be displayed:
| 'Y'          |      | displays error messages (default) |
| 'N'          |      | does not display error messages |
### _setNumberOfColumns

Sets the number of columns

#### Syntax

CALL SEND (model-id, '_setNumberOfColumns', numcols);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numcols</td>
<td>N</td>
<td>specifies the number of columns in the model</td>
</tr>
</tbody>
</table>

### _setProperties

Restores a previous state

#### Syntax

CALL SEND (model-id, '_setProperties', list);
The _setProperties method is called from the form data model’s _setEntry method and the viewer’s _attach method. Optionally, the _setProperties method also is called from the _writeEntryFromList method. The _setProperties method retrieves from the list all of the information it recognizes and updates the model appropriately.

Call the _setProperties list with a list returned from a previous call to _getProperties on a model. These methods can save and restore a model’s state.

The _setProperties method copies list into its instance variables list. Therefore, by default the _setProperties method for subclasses of the Form Data Model class copies the list into the subclass’s instance variable list. The _setProperties model method checks for items that may have changed and does the necessary updates. For example, if the list that is passed to _setProperties at the Form Data Model level contains an item called SOURCE, the _setSource method is called to ensure that the model is updated to use the new source entry.

Additional information besides instance variables in the list can occur if the subclass of the Form Data Model class overrides the _getProperties method to insert items that are not instance variables. In this case, the _setProperties method of the subclass should check for this additional information as well as check for the appropriate instance variables.

### _setRowInfo

Sets the attributes of a row

**Syntax**

CALL SEND (model-id, '_setRowInfo', data-vector-id);

**Argument** | **Type** | **Description**
---|---|---
data-vector-id | N | specifies the object identifier of an instance of the Row/Column Data Vector class

**Details**

The _setRowInfo method sets information about one row.

The data vector queried by the _setRowInfo method is an instance of the Row/Column Data Vector class. The viewer creates the data vector instance.

You can manipulate the data vector’s information with calls to methods of the Row/Column Data Vector class, which get data vector values. These methods are described in the Row/Column Data Vector class.
**_setSource**

Sets the source entry to associate with the model

---

**Syntax**

CALL SEND (model-id, '_setSource', source<, compile>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>C</td>
<td>specifies the source entry to associate with the model:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘*’ an asterisk specifies to use the source entry saved in the DATAFORM catalog entry</td>
</tr>
<tr>
<td>compile</td>
<td>C</td>
<td>specifies whether source should be compiled:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘Y’ source will be compiled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘N’ source will not be compiled</td>
</tr>
</tbody>
</table>

**Details**

You can set the source argument to ‘*’ (an asterisk) to indicate to use the source entry saved in the DATAFORM catalog entry. The source entry saved with the DATAFORM entry is the source entry that was specified the last time a data form object or data table object using that DATAFORM entry catalog was saved. Setting source to ‘*’ has the effect of setting the source entry to this source entry associated with the DATAFORM entry in the attributes screen for the data form object or data table object. If a previous source entry exists, its DFTERM label will run. Then the _setSource method will run the new source entry's DFINIT label. For more information, see the Data Set Data Model class.

---

**_term**

Terminates the model

---

**Syntax**

CALL SEND (model-id, '_term');

**Details**

The _term method runs the source entry's DFTERM label. It also calls the super _term method. (For more information on DFTERM, see the Data Set Data Model class.)
If your model subclass of the Form Data Model class has special requirements for termination, your subclass should override the _term method. The subclass should also call the super _term method so that the Form Data Model class method can run.

_validateDropData

Allows the model to validate the drag operation’s data

Syntax

CALL SEND (model-id, '_validateDropData', rep, oper, data, row, column, return-code);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep</td>
<td>C</td>
<td>specifies the representation selected for the drop</td>
</tr>
<tr>
<td>oper</td>
<td>C</td>
<td>specifies the operation selected for the drop</td>
</tr>
<tr>
<td>data</td>
<td>N</td>
<td>specifies the identifier of an SCL list that contains the data defined by the representation</td>
</tr>
<tr>
<td>row</td>
<td>N</td>
<td>specifies an SCL list identifier that contains the row coordinates of the row on which the drop occurred</td>
</tr>
<tr>
<td>column</td>
<td>N</td>
<td>specifies an SCL list identifier that contains the column coordinates of the column on which the drop occurred</td>
</tr>
<tr>
<td>return-code</td>
<td>C</td>
<td>specifies the return code:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_blank the data were successfully validated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘C’ cancel the drag operation</td>
</tr>
</tbody>
</table>

Details

The _validateDropData method is called when a drag object has been dropped on the attached viewer and the viewer does not recognize the rep passed to it. This method is passed the data returned from the _getDragData method, which was called on the drag object. This method is called to give the model a chance to validate the data and possibly cancel the drag operation if the data is not valid.

The location of the drop has been translated by the attached viewer into coordinates that make sense for that viewer. For example, when the attached viewer is a table editor or a subclass of the table editor, the location is translated into cell coordinates.

No new FRAME entries or applications can be started from this method, and the execution of the SCL code for this method cannot be followed through the SCL Debugger.

Note: The _validateDropData method is followed by the _drop method, so it is advisable to test the representation and the action before responding to the drop operation.

For general information about drag and drop operations, see SAS/AF online help.
_vscroll

Returns whether a vertical scroll should be allowed by the viewer

Syntax

CALL SEND (model-id, '_vscroll', amount, return-code);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>N</td>
<td>specifies the number of rows to be scrolled:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values &gt;0 indicate scrolling down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values&lt;0 indicates scrolling up</td>
</tr>
<tr>
<td>return-code</td>
<td>N</td>
<td>returns whether scrolling is allowed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0    scrolling is allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1    scrolling should not be allowed</td>
</tr>
</tbody>
</table>

Details

The viewer calls the _vscroll method when a vertical scroll is about to occur to make sure that scrolling should be allowed by the viewer. For example, the Data Set Data Model subclass checks the current row to see if there is anything to prevent you from leaving the row, such as a column in error.

_writeEntry

Writes the model’s information to the DATAFORM entry

Syntax

CALL SEND (model-id, '_writeEntry', entry);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>C</td>
<td>specifies the DATAFORM entry to which the model's information is written</td>
</tr>
</tbody>
</table>

Details

The _writeEntry method is called from the viewer’s _prewrite method.
If no arguments are passed to the `_writeEntry` method, the model’s information will be written to the entry specified by a previous call to the `_setEntry` method. If you would like to save to a different DATAFORM entry, pass this entry name as entry. Call `_writeEntry` whenever you want to save the model’s customizations.

### `_writeEntryFromList`

Updates the model’s information from a list

#### Syntax

CALL SEND (model-id, '_writeEntryFromList', list<, update<, return-code>>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>N</td>
<td>specifies the identifier of an SCL list that contains named items corresponding to the model information to be updated</td>
</tr>
<tr>
<td>update</td>
<td>C</td>
<td>specifies whether to send the latest information to the viewer and model after reading the list: 'Y' update, 'N' do not update</td>
</tr>
<tr>
<td>return-code</td>
<td>N</td>
<td>contains the return code from <code>_writeEntryFromList</code>: 0 entry successfully updated, -2 discrepancy between the column information in the list and that in the current model information. This discrepancy can happen if the new list contains a column that does not exist in the current model information and the new column is not a computed column, -3 out-of-memory condition, -4 a column in the list is already in the current model information but in one place the column is numeric while in the other it is character, -5 the conditions corresponding to a return code of -2 as well as the conditions corresponding to a return code of -4 both occurred (discrepancy between the columns and like-named column in list and entry of different types).</td>
</tr>
</tbody>
</table>

#### Details

The `_writeEntryFromList` method is useful for updating the model’s state. You can call this method with a list that was obtained by a previous `_readEntryIntoList` method call. The `_writeEntryFromList` method updates the model’s information from the named items in list.

The `_writeEntryFromList` method does not actually save a DATAFORM entry. To save the information from the list to a DATAFORM entry, you should call `_writeEntry`. 
You can use the return-code argument to determine if the `_writeEntryFromList` call was successful.