ISA Dialog Manager

Method Reference

A.06.03.b

In this manual all predefined methods of the ISA Dialog Manager objects are explained. It comprises the method definitions with their parameters and return values. It is also described, which methods may be overwritten (redefined).

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Germany

Notation Conventions

DM will be used as a synonym for Dialog Manager.

The notion of UNIX in general comprises all supported UNIX derivates, otherwise it will be explicitly stated.

< >	to be substituted by the corresponding value
color	keyword
.bgc	attribute
{ }	optional (0 or once)
[ ]	optional (0 or n-times)
<A> | <B>	either <A> or <B>

Description Mode

All keywords are bold and underlined, e.g.

variable **integer** **function**

Indexing of Attributes

Syntax for indexed attributes:

[I]

[I,J] meaning [row, column]

Identifiers

Identifiers have to begin with an uppercase letter or an underline ('_'). The following characters may be uppercase or lowercase letters, digits, or underlines.

Hyphens ('-') are ***not*** permitted as characters for specifying identifiers.

The maximal length of an identifier is *31* characters.

Description of the permitted identifiers in the Backus-Naur form (BNF)

<identifier>	::=	<first character>{<character>}
<first character>	::=	_ | <uppercase>
<character>	::=	_ | <lowercase> | <uppercase> | <digit>

<digit> ::= 1 | 2 | 3 | … 9 | 0

<lowercase> ::= a | b | c | … x | y | z

<uppercase> ::= A | B | C | … X | Y | Z

Table of Contents

1 Introduction

Methods serve the calling of object-specific functionality. You can instruct an object to carry out a certain operation which is defined by the relevant method. A method can have parameters and return values.

A certain number of methods is already defined in the system. In addition to these so-called user-defined methods you can define your own methods.

In the following the methods defined in the system will be described in more detail.

2 Methods in Alphabetical Order

2.1 :action()

There are two different forms of the **:action()** method:

- **With mapping objects it defines, how data is transformed at the individual nodes**
- **With transformer objects it is invoked from their :apply() methods for each visited node**

2.1.1 :action() (mapping)

During a transformation, this method determines how data is transformed at particular nodes, that is how the data from the *Src* parameter is passed to the *Dest* parameter.

Each ***mapping*** object possesses an **:action()** method, which in its default implementation simply returns *true*. The method can be redefined (similar to **:init()**). This enables to define, in which way the data from the *Src* parameter is processed before it is transferred to the *Dest* parameter.

The **:action()** method is invoked from the parent ***transformer*** of the ***mapping*** object (to be more precisely from the **:action()** method of this ***transformer***), when a node of an XML tree or an IDM object hierarchy matches the corresponding ***mapping*** object. Please refer to chapter “The transformer Object” of manual “XML Interface” for further details.

Definition

```
boolean :action
(
  anyvalue Src  input,
  anyvalue Dest input output
)
```

Parameters

anyvalue *Src* input

In this parameter the current node, which just is to be transformed, is passed. There are two different cases:

- Transformation of an XML tree:
  In *Src* a ***doccursor*** that points to the current node is passed.
- Transformation of an IDM object hierarchy:
  In *Src* an IDM object that represents the current node is passed.

anyvalue *Dest* input output

In this parameter the value is transferred, which has been passed as *Dest* with the invocation of **:apply()** or has been changed in one of the previous calls of **:action()** methods.

Return value

When the method returns *true*, it is assumed that the node has been processed completely and no more ***mapping*** objects are tested whether they match that node. Thus no further **:action()** methods will be called even though other matching ***mapping*** objects may exist.

When the method returns *false*, the current node is compared to the other ***mapping*** objects whose **:action()** methods will be invoked if applicable.

Objects with this method

mapping

2.1.2 :action() (transformer)

This method is invoked from the **:apply()** method of the ***transformer*** for each visited node of an XML tree or an IDM object hierarchy. The method then has to check all ***mapping*** children of the transformer whether the patterns in their *.name* attributes match the current node (*Src* parameter). The sequence for checking the ***mapping*** objects is determined by the sequence they are defined in the *.mapping[]* vector, with inherited ***mapping*** objects being checked last.

Attention

Inherited ***mapping*** objects are not contained in the *.mapping[]* vector of the parent. Therefore, they can only be accessed through the inheritance hierarchy. This is different from usual IDM inheritance.

When a match is found, the **:action()** method of the corresponding ***mapping*** object is called with the *Src* and *Dest* parameters simply passed on. The return value of this **:action()** method determines if the remaining ***mapping*** objects are checked and their **:action()** methods are also called for this node if applicable (return value *false*), or whether the node is considered as completed and the **:action()** method of the ***transformer*** is finished (return value *true*).

The method can be redefined (similar to **:init()**).

Definition

```
boolean :action
(
  anyvalue Src  input,
  anyvalue Dest input output
)
```

Parameters

anyvalue *Src* input
: In this parameter, the node to be examined is passed.

- When the root attribute of the ***transformer*** contains a string, a ***doccursor*** that points to the XML node is expected. This node will then be compared to the ***mapping*** objects.
- When the root attribute of the ***transformer*** contains an IDM object, the IDM object that represents the current node is expected. This object will then be compared to the ***mapping*** objects.

anyvalue *Dest* input output

In this parameter the value is transferred, which has been passed as *Dest* with the invocation of **:apply()** or has been changed in one of the previous calls of **:action()** methods.

Return value

The method returns *true* if processing was completed without errors, *false* otherwise.

Objects with this method

transformer

2.2 :add()

This method appends a child node to the current DOM node. The XML Cursor then points to the newly added node.

Definition

```
boolean :add
(
      enum    Nodetype input
  { , string  Name    := null input }
  { , string  Value   := null input }
  { , boolean Select := true input }
)
```

Second Form (MICROSOFT WINDOWS Only)

```
boolean :add
(
      pointer IXMLDOMNode input
  { , boolean Select := true input }
)
```

Parameters

enum *Nodetype* input

Type of the appended DOM node.

Value range

nodetype_attribute
: The node is an attribute of an element.

nodetype_cdata_section
: The node is a section with unparsed character data (CDATA).

nodetype_comment
: The node is a comment.

nodetype_document
: The node is a complete document.

nodetype_document_fragment
: The node is a section of a document.

nodetype_document_type
: The node is a document type declaration.

nodetype_element
: The node is an element.

nodetype_entity
: The node is an entity declaration.

nodetype_entity_reference
The node is a reference to a declared entity.
nodetype_notation
The node is a notation declared in the DTD.
nodetype_processing_instruction
The node is a processing instruction.
nodetype_text
The node is the text content of an element or an attribute value.

string ***Name*** *:= null input*

Optional parameter that specifies the name or tag for the new DOM node. When the *Nodetype* parameter has one of the values from the table below, *Name* is ignored and a predefined name is used.

Nodetype	Predefined Name
nodetype_cdata_section	*#cdata-section*
nodetype_comment	*#comment*
nodetype_document	*#document*
nodetype_document_fragment	*#document-fragment*
nodetype_text	*#text*

string ***Value*** *:= null input*

Optional parameter that specifies the value of the new DOM node. Only applicable if the *Nodetype* parameter has one of these values:

- *nodetype_attribute*
- *nodetype_cdata_section*
- *nodetype_comment*
- *nodetype_processing_instruction*
- *nodetype_text*

pointer *IXMLDOMNode* input

A MICROSOFT WINDOWS COM object that is appended as child node. May also be aMICROSOFT WINDOWS **IXMLDOMNodeList** COM object, of which all XML nodes contained in the list are appended.

boolean ***Select*** *:= true input*

Optional parameter that defines, whether the XML Cursor shall point to the newly appended DOM node. When this parameter is missing, *true* is used as default so that the XML Cursor is set to point to the new DOM node.

Return value

The method returns *true*, when the child node could be appended, *false* otherwise.

In case of an error the XML Cursor remains pointing to the same DOM node as before the method call.

Objects with this method

doccursor

2.3 :apply()

There are two different forms of the **:apply()** method:

» **With the Datamodel, it retrieves the values of all View attributes and assigns them to the Model components**

» **On the transformer object, it triggers the transformation of data**

2.3.1 :apply() (Datamodel)

When calling this method on a View component, all data values of View attributes that are linked to Model components are read (using a **:retrieve()** call) and assigned to the corresponding Model components.

By specifying the optional parameters, the restriction to a special view attribute or to a special indexed single value is possible. Without parameters **:apply()** will also be invoked on all children and their children.

Normally calling this method should not be necessary because synchronization between View and Model is controlled through the *.dataoptions[]* attribute.

Linkage to the Model component must have been established with the attributes *.datamodel* and *.dataset*.

There will be no error message if the object has no linked attributes or the optionally specified attribute does not exist or is not linked.

Definition

```
void :apply
(
  { attribute Attribute input { , anyvalue  Index input } } |
  { anyvalue  Index     input }
)
```

Parameters

*attribute **Attribute** input*

This optional parameter specifies the View attribute to be retrieved and assigned to the corresponding Model component.

*anyvalue **Index** input*

This optional parameter specifies the index value to be used for getting the value from the View object.

Example

In the following example dialog, the content string is assigned to the Data Model using the **:apply()** method on the ***edittext*** or on the ***window***. The ***statictext*** at the bottom of the window shows the current value of the Data Model “VarString”.

```
dialog D
accelerator AcF5 "F5";
variable string VarString := "Friday";

window Wi
{
  .title ":apply demo";
  .width 200;

  child edittext Et
  {
    .xauto 0;
    .xright 80;
    .datamodel VarString;
    .dataset .value;
    .content "Saturday";
    .toolhelp "Press F5 to apply";

    on key AcF5
    {
      this:apply(.content);
    }
  }

  child pushbutton Pb
  {
    .xauto -1;
    .width 80;
    .text "Apply";

    on select
    {
      this.window:apply();
    }
  }

  child statictext St
  {
    .xauto 0;
    .yauto -1;
    .datamodel VarString;
    .dataget .value;
  }

  on close { exit(); }
}
```

2.3.2 :apply() (transformer)

With this method the transformation of data is triggered at a ***transformer*** object. Please refer to chapter “The transformer Object” of manual “XML Interface” for further details.

The method can be redefined (similar to **:init()**).

Definition

```
boolean :apply
(
      anyvalue Src  input
  { , anyvalue Dest input output }
)
```

Parameters

anyvalue *Src* input

In this parameter the root node is transferred from which the transformation starts. The default implementation differentiates between two cases:

- If *Src* is a ***document*** object, the root of the XML tree, which represents the ***document***, is taken as start node. The ***document*** must be loaded.
  If *Src* is a ***doccursor*** object, the node in the XML tree to which the ***doccursor*** points is taken as root node.
  The transformation is carried out from XML to IDM.
- If *Src* is an IDM object (except for document and doccursor), this object itself is taken as root. The transformation is carried out to user-defined IDM data.

*anyvalue **Dest** input output*

This optional parameter is simply passed on to the **:action()** methods of the ***mapping*** objects. This means that within the method, *:action(Next_Node, Dest)* is invoked iteratively for each visited node.

Return value

The method returns *true* if the transformation was successful, *false* otherwise.

Objects with this method

transformer

2.4 :call()

The method **:call()** is used to invoke any built-in method or user-defined method. This is necessary if the method to be invoked is calculated or saved in a variable. Calling **:call()** indirectly results in the call of the given method. The return value corresponds to the indirectly called method. When invoked via **:call()**, however, only a maximum of 15 parameters can be passed to the indirectly called method. Please note this when you design methods. The method **:call()** can also call built-in methods such as **:insert()** or **:delete()**.

Definition

```
anyvalue :call
(
     method   Method input
  { ,anyvalue Arg1   input }
  ...
  { ,anyvalue Arg15  input }
)
```

Parameters

method *Method* input

This parameter contains the method to be called.

*anyvalue **Par1** input*

…

*anyvalue **Par15** input*

In these optional parameters the parameters of the method to be called are specified. The number of specified parameters must correspond to the number of parameters of the method to be called.

Return value

Return value of method being invoked.

Objects with this method

All objects which can have user-defined methods or have built-in methods.

Example

```
dialog Example

record Rec
{
  string String;
  rule void SomeMethod() { }
  rule boolean Display(object Obj input) { }
}
```

```
on dialog start
{
  Rec:call(:SomeMethod);     // calls the method SomeMethod
  Rec:call(:Display, this);  // calls the method Display
                             // with the dialog being the object
}
```

See also

Method **:recall()**

2.5 :calldata()

When this method is called, all Data Models defined at the called object and its children with the *.datamodel* attribute are gathered and an “action” in form of a method with arguments is invoked only once for each Data Model.

With this method, at a View object an “action” can be triggered on the involved Data Models without having detailed knowledge about the defined Model objects.

A return value or feedback, e.g. if the method does not exist on the Data Models, is not provided.

Definition

```
void :calldata
(
      method   Method input
  { , anyvalue Arg1   input }
  ...
  { , anyvalue Arg15  input }
)
```

Parameters

method *Method* input
This parameter defines the method to be invoked on the involved Data Models.

*anyvalue **Arg1** input*
…
*anyvalue **Arg15** input*
These optional parameters contain the arguments used for the method invocation.

Example

```
dialog D

record RecPart
{
  string Name[5] := "";
  .Name[1] := "car";
  .Name[2] := "wheel";
  .Name[3] := "seat";
  .Name[4] := "front pane";
  .Name[5] := "break pedal";

  rule void Add()
  {
    variable integer Idx := this.count[.Name] + 1;
    this.count[.Name] := Idx;
    this.Name[Idx] := "part#" + Idx;
```

```
  }
}

record RecLevel
{
  integer Level[5] := 0;

  rule void Add()
  {
    this.count[.Level] := this.count[.Level] + 1;
  }

  rule void Indent(integer Idx)
  {
    if Idx>0 andthen Idx<=this.count[.Level] then
      this.Level[Idx] := (this.Level[Idx]+1) % 5;
    endif
  }
}

window Wi
{
  .height 200;
  .title ":calldata demo";
  .datamodel RecPart;

  treeview Tv
  {
    .xauto 0;
    .yauto 0; .ybottom 30;
    .datamodel[.level] RecLevel;
    .dataget[.content] .Name;
    .dataget[.level] .Level;
    .open[0] true;
  }

  pushbutton PbIndent
  {
    .yauto -1;
    .text "Indent >";

    on select
    {
      this.window:calldata(:Indent, Tv.activeitem);
    }
  }
```

```
  pushbutton PbAdd
  {
    .xauto -1;  .yauto -1;
    .text "Add";

    on select
    {
      this.window:calldata(:Add);
    }
  }

  on close { exit(); }
}
```

2.6 :childcount()

This method is used for the ***treeview*** object to query the number of subelements belonging to a tree element. Via parameters you can control how many indention levels are to be considered for the calculation.

Definition

```
integer :childcount
(
      integer Position input
  { , integer Levels   := 65535 input }
)
```

Parameters

integer *Position* input

In this parameter the index of the item is specified which the number of subelements in a tree is to be calculated for.

*integer **Levels** := 65535 input*

In this optional parameter the maximum indention level to be considered for the items can be specified. If this parameter is not specified all items will be considered.

Return value

Number of found elements

Objects with this method

Treeview

Example

```
window Wn
{
  .title "Example for method :childcount()";
  child treeview Tv
  {
    .content[1] "Company A";
    .content[2] "Miller";
    .content[3] "Meyer";
    .content[4] "Company B";
    .content[5] "Davis";
    .content[6] "Smith";
    .content[7] "Fisher";
    .style[style_lines]   true;
    .style[style_buttons] true;
    .style[style_root]    true;
```

```
    .level[2] 2;
    .level[3] 2;
    .level[5] 2;
    .level[6] 2;
    .level[7] 2;
  }

  child pushbutton Pb_Exit
  {
    .text "Exit";
    on select
    {
      exit();
    }
  }

  child statictext St2
  {
    .xleft 355;
    .ytop  71;
    .text  "no value";
  }

  child pushbutton Pb
  {
    .xleft  226;
    .width  247;
    .ytop   16;
    .height 28;
    .text   "Count staff working in a company";
    on select
    {
      St2.text := itoa(Tv:childcount(Tv.activeitem, 1));
    }
}
```

2.7 :childindex()

This method is used to calculate the absolute index of a child in a ***treeview***. For this purpose the relative index of the child in relation to its parent is passed on to this method.

Definition

```
integer :childindex
(
  integer Parent input,
  integer Child  input
)
```

Parameters

integer *Parent* input

In this parameter the index of a parent is specified.

integer *Child* input

In this parameter the index in relation to the relevant parent is specified.

Return value

0

if error occurred

> 0

absolute index of child

Objects with this method

Treeview

Example

```
window Wn
{
  .title "Example for method :childindex()";
  child treeview Tv
  {
    .content[1] "Company A";
    .content[2] "Miller";
    .content[3] "Meyer";
    .content[4] "Company B";
    .content[5] "Davis";
    .content[6] "Smith";
    .content[7] "Fisher";
    .style[style_lines]   true;
    .style[style_buttons] true;
    .style[style_root]    true;
```

```
    .level[2] 2;
    .level[3] 2;
    .level[5] 2;
    .level[6] 2;
    .level[7] 2;
  }

  child pushbutton PbIn
  {
    .text "Define absolute number";
    on select
    {
      variable integer I := Tv:childindex(Tv.activeitem, atoi(EtIn.content));

      if fail(atoi(EtIn.content)) then
        St.text := "Have you indicated a person?";
      endif
      if (I = 0) then
        St.text := "Have you selected a company?";
      else
        St.text := itoa(I);
      endif
    }
  }

  child edittext EtIn
  {
    .content   "Which person of the selected company is this?";
    .multiline true;
  }

  child statictext St
  {
    .text "Result of query";
  }
}
```

2.8 :clean()

This method can be used to clean up the objects of a dialog or module (release of previously allocated resources) before they are finally destroyed. The method is predefined on all objects, Models and Defaults, but may be overwritten to carry out custom actions on the objects.

The **:clean()** method cannot be invoked explicitly. It is only called implicitly:

1. If a dialog or module is to be unloaded, the **:clean()** method is called for all objects, Models and Defaults of the dialog or module after the *finish* rule.
2. If an object or model is destroyed at runtime (calling the **:destroy()** method, built-in function **destroy()** or the interface function **DM_Destroy()** or **DMcob_Destroy()**), the **:clean()** method is invoked for that object immediately before it is destroyed.

Definition

```
void :clean
(
)
```

Parameters

None.

Redefinition

To redefine the method, it is sufficient to simply write the following at an object:

```
window W {
  :clean()
  {
    ...
    // Code to release things or bring data into consistent state.
    ...
  }
}
```

In order to delegate some of the tasks to the superclass, *this:super()* can be used to call the **:clean()** method of the Model or Default further up in the inheritance hierarchy.

Note

Calling *this:super()* has another important function for the **:clean()** method. If this method is called at the top level (typically the Default), even if there is no more formal superclass, the **:clean()** methods on the children of the object will also be invoked. Hence, invocations of the **:clean()** methods on the children can be suppressed if somewhere in the inheritance hierarchy there is no call of *this:super()*.

Example

```
dialog CLEAN

default record {
  :clean() {
    print "== DEFAULT >>>";
    this:super();
    print "== DEFAULT <<<";
  }
}

model record MRec1 {
  :clean() {
    print "== MRec1 >>>";
    this:super();
    print "== MRec1 <<<";
  }

  child record R1 {
    :clean() {
      print "== R1 >>>";
      this:super();
      print "== R1 <<<";
    }
  }
  child record R2 {
    :clean() {
      print "== R2 >>>";
      this:super();
      print "== R2 <<<";
    }
  }
}

model MRec1 MRec2 {
  :clean() {
    print "== MRec2 >>>";
    // No call of this:super().
    print "== MRec2 <<<";
  }
}

model MRec1 MRec3
{
  // Here the predefined :clean() method comes into action.
  child record R3 {
    :clean() {
```

```
      print "== R3 >>>";
      this:super();
      print "== R3 <<<";
    }
  }
}

on dialog start
{
  variable object O;

  O := MRec2:create(CLEAN);
  print "on start: Destroy";
  O:destroy();
  // No :clean() of children R1 and R2.

  O := MRec3:create(CLEAN);
  print "on start: Destroy";
  O:destroy();
  // :clean() methods of children R1, R2 and R3 are invoked.

  exit();
}
```

The modified section from the trace file should illustrate what happens when this code is executed.

```
"on start: Destroy"
    "== MRec2 >>>"
    "== MRec2 <<<"
// No :clean() of children R1 and R2.

"on start: Destroy"
    "== MRec1 >>>"
        "== DEFAULT >>>"
      "== R1 >>>"
          "== DEFAULT >>>"
          "== DEFAULT <<<"
      "== R1 <<<"
      "== R2 >>>"
          "== DEFAULT >>>"
          "== DEFAULT <<<"
      "== R2 <<<"
      "== R3 >>>"
          "== DEFAULT >>>"
          "== DEFAULT <<<"
      "== R3 <<<"
        "== DEFAULT <<<"
    "== MRec1 <<<"
```

```
// :clean() methods of children R1, R2 and R3 are invoked.
```

2.9 :clear()

There are two different forms of the **:clear()** method:

» **Deleting the content of entries in list objects**

» **Deleting the content of items in fields (indexed user-defined attributes)**

2.9.1 :clear() (List Objects)

This method deletes the values from rows and columns in the contents of an object whose contents consist of multiple entries indexed with *integer* or *index*. Along with the contents, the values of the associated attributes are deleted as well.

In contrast to the **:delete()** method, **:clear()** only deletes the values in the rows or columns. The entries themselves remain as "empty" entries, which will have the default value if available. Thus **:clear()** will not reduce the number of rows or columns.

Definition

```
boolean :clear
(
      integer Start input
  { , integer Count := 1 input }
  { , boolean Direction := false input }
)
```

Parameters

integer *Start* input

This parameter defines the position from which the contents of the rows or columns should be deleted.

*integer **Count** := 1 input*

This optional parameter defines the number of rows or columns whose contents are to be deleted. If the parameter is not specified, *1* is taken as default.

*boolean **Direction** := false input)*

This optional parameter controls whether the contents of rows or of columns are deleted in the ***tablefield***. This depends on the value of the ***tablefield*** attribute *.direction*.
The parameter may only be passed for the ***tablefield***. For other objects than the ***tablefield***, passing the parameter will result in an error.

Value range

false

Deletes against the direction given in the *.direction* attribute

true

Deletes in the same direction as given in the *.direction* attribute

Thus these contents are deleted from the ***tablefield***:

Parameter *Direction*	*.direction 1* (vertical)	*.direction 2* (horizontal)
false	row(s)	column(s)
true	column(s)	row(s)

Return value

The method returns *true* if the contents of the rows or columns could be deleted.

If an error has occurred when deleting the contents, the method will return a `fail`.

Objects with this method

- listbox (attribute *.content[integer]*)
- poptext (attribute *.text[integer]*)
- spinbox (attribute *.text[integer]*), nur bei *.style = string*
- tablefield (attribute *.content[index]*)
- treeview (attribute *.content[integer]*)

Example

```
window Wn
{
  .title "Example for method :clear()";

  child listbox Lb
  {
    .content[1] "McCusker";
    .content[2] "McNeill";
    .content[3] "Pinter";
    .content[4] "Tilly";
    .content[5] "Miller";
    .content[6] "Meyer";
    .content[7] "Davis";
  }

  child pushbutton Pb_clear
  {
    .text "Clear entry";

    on select
    {
      Lb:clear(Lb.activeitem);
    }
```

```
  }
}
```

2.9.2 :clear() (Arrays)

This method deletes the contents of elements in fields (indexed, non-associative, user-defined attributes).

In contrast to the **:delete()** method, **:clear()** only deletes the contents of the items. The items themselves remain as “empty” items, which will have the default value if available. Thus **:clear()** will not reduce the number of items.

Particularity

The **:clear()** method can also be used to delete **all** items of an **associative** array. In this case, neither the *Start* nor the *Count* parameter may be specified. Therefore, **:clear()** cannot be used to delete single items from an associative array. Additionally, not only the contents but the elements themselves are deleted in this case. Thus, the number of elements afterwards will be *0*.

Definition

```
boolean :clear
(
      attribute Attr input
  { , integer Start input }
  { , integer Count input }
)
```

Parameters

attribute *Attr* input
: This parameter defines the user-defined attribute on which the method should be applied.

*integer **Start** input*
: This parameter defines the position from which the contents of the items should be deleted.
The value range for *Start* is *0 … .count[Attr]*, i.e. the default value can also be deleted.
If neither *Start* nor *Count* is specified, the contents of all elements are deleted.

*integer **Count** input*
: This optional parameter defines the number of items whose contents are to be deleted.
If neither *Start* nor *Count* is specified, the contents of all elements are deleted.
If *Start* is specified, the default value of *Count* is *1*.

Return value

The method returns *true* if the contents of the items could be deleted.

If an error has occurred when deleting the contents, the method will return a `fail`.

Objects with this method

All objects that may have user-defined attributes.

See also

Chapter “Methods for Arrays of User-defined Attributes” in manual “User-defined Attributes and Methods”

2.10 :collect()

Calling this method on a Model component retrieves all data values from the View components that have linked attributes. For this, **:retrieve()** is invoked for the respective View attributes to determine their data values and assign them to the Data Model.

Normally calling this method should not be necessary because synchronization between View and Model is controlled through the *.dataoptions[]* attribute.

Linkage to the Model component must have been established with the attributes *.datamodel* and *.dataset*.

There is no error message if the object has no reference by a View component or no linked attributes.

Definition

```
void :collect
(
)
```

Example

```
dialog D
record Rec
{
  .dataoptions[dopt_propagate_on_start] false;
  string  FirstName := "";
  string  LastName  := "";
  boolean Female    := false;
}

timer Ti
{
  .starttime "+00:00:02";
  .incrtime "+00:00:01";
  .active true;

  on select
  {
    Rec:collect();  /* read data explictly (from EtFirst/LastName&CbFemale)
*/
  }
}

window Wi
{
  .title ":collect demo";
  .width 200;  .height 200;
  .xleft 250;
```

```
  .datamodel Rec;
  .dataoptions[dopt_represent_on_map] false;

  statictext
  {
    .text "First Name"; .width 100;
  }

  edittext EtFirstName
  {
    .content "Ina";
    .xauto 0;
    .xleft 100;
    .dataset .FirstName;
  }

  statictext
 {
    .text "Last Name";  .width 100;  .ytop 30;
  }

  edittext EtLastName
  {
    .ytop 30;
    .xauto 0;  .xleft 100;
    .content "Bausch";
    .dataset .LastName;
  }

  checkbox CbFemale
  {
    .ytop 60;
    .text "Female";
    .active true;
  }

  statusbar Sb
  {
    statictext StFirstName
    {
      .width 100;
      .dataget .FirstName;
    }

    statictext StLastName
    {
      .dataget .LastName;
```

```
      .text "Update in 2 secs";
    }
  }

  on close { exit(); }
}
```

2.11 :create()

With this method new objects can be created at runtime. The method returns the newly created object.

Definition

```
object :create
(
      object  Parent input
  { , object  Dialog    := null  input }
  { , integer Type      := 3     input }
  { , boolean Invisible := false input }
)
```

Parameters

object *Parent* input

This parameter defines the parent of the new object.

*object **Dialog** := null input*

This optional parameter defines to which dialog the new object shall belong.

*integer **Type** := 3 input*

This optional parameter defines whether a Default, a Model or an instance is created.

Value range

1
: creates a Default

2
: creates a Model

3
: creates an instance

If this parameter is missing, an instance is created.

*boolean **Invisible** := false input*

This optional parameter controls, whether the new object is created invisible or with the same visibility as its Model or Default.

Value range

true
: The object is always created invisible.

false
: Visibility is adopted from the Model or Default.

Return value

objectId
: The newly created object.

null
: If the object could not be created.

Example

```
dialog Example

model window MWnDetail
{
  !! detail view of data sets
  .title "Detail view of data";
  rule void GetData()
  {
    !! fetching the data
    !! make window visible
    this.visible ::= true;
  }
  child pushbutton PbExit
  {
    .text "&Exit";
    on select
    {
      !! destroys the window instance
      this.window:destroy();
    }
  }
}

window WnMain
{
  .title "Data overview";
  child tablefield TfData
  {
    !! contains a data overview
    on dbselect
    {
      variable object NewObj := null;

      !! create visible
      !! NewObj ::= MWnDetail:create(this.dialog);
      !! create invisible
      NewObj ::= MWnDetail:create(this.dialog, true);
      NewObj:GetData();
    }
```

```
  }
}
```

See also

Built-in function create() in manual “Rule Language”

C function DM_CreateObject in manual “C Interface - Functions”

COBOL function DMcob_CreateObject in manual “COBOL Interface”

2.12 :delete()

There are three different forms of the **:delete()** method:

- **Deleting entries from list objects**
- **Deleting items from fields (indexed user-defined attributes)**
- **Deleting the DOM node an XML Cursor points to**

2.12.1 :delete() (List Objects)

This method deletes rows and columns from the contents of an object whose contents consist of multiple entries indexed with *integer* or *index*. Along with the contents, the associated attributes are deleted as well.

Unlike the **:clear()** method, with **:delete()** rows or columns are deleted completely, i.e. the number of rows or columns will be reduced by **:delete()**.

Definition

```
boolean :delete
(
      integer Position input
  { , integer Count := 1 input }
  { , boolean Direction := false input }
)
```

Parameters

integer *Position* input

This parameter defines the position from which the rows or columns should be deleted.

*integer **Count** := 1 input*

This optional parameter defines the number of rows or columns that are deleted. If the parameter is not specified, *1* is taken as default.

*boolean **Direction** := false input*

This optional parameter controls whether rows or columns are deleted in the ***tablefield***. This depends on the value of the ***tablefield*** attribute *.direction*.
The parameter may only be passed for the ***tablefield***. For other objects than the ***tablefield***, passing the parameter will result in an error.

Value range

false
: Deletes against the direction given in the *.direction* attribute

true
: Deletes in the same direction as given in the *.direction* attribute

Thus these contents are deleted from the ***tablefield***:

Parameter *Direction*	*.direction 1* (vertical)	*.direction 2* (horizontal)
false	row(s)	column(s)
true	column(s)	row(s)

Return value

The method returns *true* if the rows or columns could be deleted.

If an error has occurred when deleting, the method will return a `fail`.

Objects with this method

- listbox (attribute *.content[integer]*)
- poptext (attribute *.text[integer]*)
- spinbox (attribute *.text[integer]*)
- tablefield (attribute *.content[index]*)
- treeview (attribute *.content[integer]*)

Example

```
dialog D

window Wn
{
  .title "Example for the method :delete()";

  child treeview Tv
  {
    .content[1] "Company A";
    .content[2] "Miller";
    .content[3] "Mayer";
    .content[4] "Company B";
    .content[5] "Jones";
    .content[6] "Smith";
    .content[7] "Fisher";
    .style[style_lines]   true;
    .style[style_buttons] true;
    .style[style_root]    true;
    .level[2] 2;
    .level[3] 2;
    .level[5] 2;
    .level[6] 2;
    .level[7] 2;
  }

  child pushbutton Pb
```

```
  {
    .text "Remove an employee";

    on select
    {
      Tv:delete(Tv.activeitem);
    }
  }
}
```

2.12.2 :delete() (Arrays)

This method deletes items from arrays (indexed user-defined attributes).

Unlike the **:clear()** method, **:delete()** completely deletes the elements, i.e. the number of elements is reduced.

Definition

```
boolean :delete
(
      attribute Attr input,
      anyvalue Position input
  { , integer Count := 1 input }
)
```

Parameters

attribute *Attr* input

This parameter defines the user-defined attribute on which the method should be applied.

anyvalue *Position* input

In this parameter the index of the first item to be deleted is passed. For non-associative arrays *Position* has to be of the data type *integer*, for associative arrays it has to be of the data type the associative array is indexed with.

The value range for *Position* with non-associative fields is *0 … .count[Attr]*, i.e. the default value can also be deleted. If the default value is deleted, the value of the first item after the deleted items becomes the default value.

integer ***Count*** *:= 1 input*

This optional parameter defines the number of items that are deleted. If the parameter is not specified, *1* is taken as default.

Count cannot be used with associative arrays, only *1* item can be deleted at once.

Return value

The method returns *true* if the rows or columns could be deleted.

If an error has occurred when deleting, the method will return a `fail`.

Objects with this method

All objects that may have user-defined attributes.

See also

Chapter “Methods for Arrays of User-defined Attributes” in manual “User-defined Attributes and Methods”

Chapter “Working with Associative Arrays” in manual “User-defined Attributes and Methods”

2.12.3 :delete() (doccursor)

This method deletes the DOM node that the ***XML Cursor*** points to with all of its child nodes from an ***XML Document***. The XML Cursor is then positioned to the parent node. XML Cursors, which point to the deleted DOM nodes in the sub-tree, become invalid. The attribute *.mapped* has the value *false* for invalid XML Cursors.

If the value of the *.path* attribute is stored elsewhere (e.g. in the *.userdata* attribute), it should be noted that the stored value is not adjusted when the structure of the DOM tree changes. When the **:select** method is invoked with the stored value afterward, the XML Cursor may be pointing to an incorrect DOM node.

Definition

```
boolean :delete
(
)
```

Parameters

None.

Return value

true
: The DOM node could be deleted.

false
: An error occurred during deletion.

Objects with this method

doccursor

2.13 :destroy()

With this method any objects or models of a dialog can be deleted. All children of the object are also deleted.

Definition

```
boolean :destroy
(
  { boolean DoIt := false input }
)
```

Parameters

*boolean **DoIt** := false input*

This optional parameter (default value *false*) controls how this object should be deleted. If *true* is specified here, the object is deleted and all rule parts that use this object are modified so that the respective statements are removed.
When the object to be deleted is a Model and the parameter is *false*, the Model is only deleted if it is not used by any other object (default). If *true* is specified in this case, then the Model is deleted and the objects that use this Model will refer to the next higher Model or the Default again.

Return value

true
Object could be deleted.

false
Object could not be deleted.

Note

:destroy() invokes the **:clean()** method of the object.

Example

```
dialog Example

model window MWnDetail
{
  !! detail view of data sets
  .title "Detail view of data";
  rule void GetData()
  {
    !! fetching the data
    !! make window visible
    this.visible ::= true;
  }
  child pushbutton PbExit
  {
```

```
    .text "&Exit";
    on select
    {
      !! destroys the window instance
      this.window:destroy();
    }
  }
}

window WnMain
{
  .title "Data overview";
  child tablefield TfData
  {
    !! contains a data overview
    on dbselect
    {
      variable object NewObj := null;

      !! create visible
      !! NewObj ::= MWnDetail:create(this.dialog);
      !! create invisible
      NewObj ::= MWnDetail:create(this.dialog, true);
      NewObj:GetData();
    }
  }
}
```

See also

Built-in function destroy() in manual “Rule Language”

C function DM_Destroy in manual “C Interface - Functions”

COBOL function DMcob_Destroy in manual “COBOL Interface”

2.14 :exchange()

There are two different forms of the **:exchange()** method:

» **Exchanging entries in list objects**

» **Exchanging items in fields (indexed user-defined attributes)**

2.14.1 :exchange() (List Objects)

This method exchanges two rows or columns in the contents of an object whose contents consist of multiple entries indexed with *integer* or *index*. Along with the contents, the associated attributes are exchanged as well.

Definition

```
boolean :exchange
(
      integer Position1 input,
      integer Position2 input
  { , boolean Direction := false input }
)
```

Parameters

integer *Position1* input
integer *Position2* input

These parameters define the two rows or columns that should be exchanged with each other.

*boolean **Direction** := false input*

This optional parameter controls whether rows or columns are exchanged in the ***tablefield***. This depends on the value of the ***tablefield*** attribute *.direction*.
The parameter may only be passed for the ***tablefield***. For other objects than the ***tablefield***, passing the parameter will result in an error.

Value range

false
Exchanges against the direction given in the *.direction* attribute

true
Exchanges in the same direction as given in the *.direction* attribute

Thus these contents are exchanged in the ***tablefield***:

Parameter *Direction*	*.direction 1* (vertical)	*.direction 2* (horizontal)
false	row(s)	column(s)
true	column(s)	row(s)

Return value

The method returns *true* if the rows or columns could be exchanged.

If an error has occurred when exchanging, the method will return a `fail`.

Objects with this method

- listbox (attribute *.content[integer]*)
- poptext (attribute *.text[integer]*)
- spinbox (attribute *.text[integer]*)
- tablefield (attribute *.content[index]*)
- treeview (attribute *.content[integer]*)

Example

```
dialog D

window Wn
{
  .title "Example for the method :exchange()";

  child treeview Tv
  {
    .content[1] "Company A";
    .content[2] "Miller";
    .content[3] "Mayer";
    .content[4] "Company B";
    .content[5] "Jones";
    .content[6] "Smith";
    .content[7] "Fisher";
    .style[style_lines]   true;
    .style[style_buttons] true;
    .style[style_root]    true;
    .level[2] 2;
    .level[3] 2;
    .level[5] 2;
    .level[6] 2;
    .level[7] 2;
  }

  child statictext St
  {
    .text "Select an employee";
  }

  child pushbutton Pb
  {
```

```
    .text "Exchange two employees";

    on select
    {
      if (Tv.activeitem > 1) then
        if (Tv.level[Tv.activeitem] = Tv.level[(Tv.activeitem - 1)]) then
          Tv:exchange(Tv.activeitem, (Tv.activeitem - 1));
          St.text := "Exchange ok";
        else
          St.text := "Upper partner is a company";
        endif
      else
        St.text := "There is no upper partner";
      endif
    }
  }
}
```

2.14.2 :exchange() (Arrays)

This method exchanges two items of an array (indexed, non-associative, user-defined attribute).

Definition

```
boolean :exchange
(
  attribute Attr input,
  integer Position1 input,
  integer Position2 input
)
```

Parameters

attribute *Attr* input

This parameter defines the user-defined attribute on which the method should be applied.

integer *Position1* input
integer *Position2* input

In these parameters the indices of the two items to be exchanged with each other are passed. The value range for both parameters is *0 … .count[Attr]*, i.e. the default value can also be exchanged with the value of another item.

Return value

The method returns *true* if the items could be exchanged.

If an error has occurred when exchanging, the method will return a `fail`.

Objects with this method

All objects that may have user-defined attributes.

See also

Chapter “Methods for Arrays of User-defined Attributes” in manual “User-defined Attributes and Methods”

2.15 :find()

This method searches in predefined and user-defined indexed attributes (one-dimensional, two-dimensional and associative arrays) for a specific value and returns the first index where this search value is found.

The search never includes the default element (e.g. index *[0]* or *[0,0]*).

By specifying a start and end index, the search can be restricted. The valid values are:

- No value specified (*void*).
- *integer* value in the range *1 … field size* (but must be > *0*) for (associative) arrays.
- *index* value in the range from *[1,1]* to *[rowcount,colcount]* where *rowcount* and *colcount* must also be > *0*.

All other values will cause a `fail`.

This implies that searching a complete array should be carried out without a start and end index, because a start or end index of *[0]* or *[0,0]* would cause a `fail`.

When searching for strings it is also possible to search case-insensitive or for the first occurrence as prefix.

The search is only performed in the range *1 … n* or *[1,1] … [n,m]*, i.e. indexes of *0* are not allowed.

Since the indices of associative arrays are not subject to any order, only an *integer* value in the range *1 … itemcount[<attribute>]* can be specified as index, which indicates the position of the start or end element.

For two-dimensional arrays (*.content[]* at the ***tablefield***), specifying an index range *[Sy,Sx]* to *[Ey,Ex]* restricts the search scope to the range *[min(Sy,Ey),min(Sx,Ex)] … [max(Sy,Ey),max(Sx,Ex)]*. Thus the restriction to rows and columns is easily possible.

The search is performed in the direction of the specified indices and taking *.direction* into account (only for two-dimensional arrays). Specifying *10, 1* instead of *1, 10* as start and end indexes therefore reverses the search direction. A vertically aligned two-dimensional array is searched in the order rows/columns, with horizontal alignment this is exactly the opposite. The index of the first field matching the value is returned.

Definition

```
anyvalue :find
(
  {   attribute Attribute input, }
      anyvalue  Value     input
  { , anyvalue  StartIdx  input
  { , anyvalue  EndIdx    input } }
  { , boolean   CaseSensitive := false       input }
  { , enum      CompareMode   := match_exact input }
)
```

Parameters

attribute *Attribute* input

This parameter specifies the predefined or user-defined attribute where the value is searched for. If this parameter is not specified then for ***poptext*** and ***spinbox*** the *.text[integer]* attribute is searched and for all other objects the *.content[]* attribute.

anyvalue *Value* input

The value specified in this parameter is searched for in the indexed attribute. Only values are compared whose types are “equal” or can be converted (a ***text*** is converted to a *string* for this purpose).

If this value is a string or a ***text*** resource, the parameters *CaseSensitive* and *CompareMode* are also applied for the search.

anyvalue ***StartIdx*** *input*

This optional parameter specifies the start index from which to search for the value in the indexed attribute. If no value is specified here, *1* is assumed for one-dimensional and *[1,1]* for two-dimensional attributes.

anyvalue ***EndIdx*** *input*

This optional parameter specifies the end index up to which the indexed attribute shall be searched for the value. This argument is only permitted if a start index has also been specified.

boolean ***CaseSensitive*** *:= false input*

This optional parameter only applies to *string* values and defines whether the search should be case-sensitive or not. The default value is *false*.

enum ***CompareMode*** *:= match_exact input*

This optional parameter only applies to *string* values and defines how to search for strings.

Value range

match_begin
: Finds the first string that starts with the search string.

match_exact
: Finds the first string that exactly matches the string that is searched for.

match_first
: Finds the string whose beginning has the largest match with the search string. If there is a string that matches the search string exactly, its index will be returned.

match_substr
: Finds the first string that contains the string that is searched for.

Return value

0
: Item was not found in the passed one-dimensional, non-associative attribute.

[0,0]
: Item was not found in the passed two-dimensional attribute.

nothing
: Item was not found in the passed associative array.

otherwise
: Index of the searched item in the passed attribute. The value returned has the index data type of the passed attribute.

Fault behavior

Invocation of the method fails if the attribute is not known or the range defined by *StartIdx* and *EndIdx* is not within the valid index range.

Objects with this method

- listbox (default attribute *.content[integer]*)
- poptext (default attribute *.text[integer]*)
- spinbox (default attribute *.text[integer]*)
- tablefield (default attribute *.content[index]*)
- treeview (default attribute *.content[integer]*)
- All objects that may have user-defined attributes.

Example

```
window Wn
{
  .title "Example for method :find()";

  child listbox Lb
  {
    .content[1] "McCusker";
    .content[2] "McNeill";
    .content[3] "Pinter";
    .content[4] "Tilly";
    .content[5] "Miller";
    .content[6] "Meyer";
    .content[7] "Davis";

    rule void Showfind (string What input)
    {
      variable integer Index;

      !! Observes upper/lower case and searches for the first matching
string.
      Index := Lb:find(What, true, match_begin);
      if (Index = 0) then
        Et_find.content := "No item found";
      else
        Lb.activeitem := Index;
        Et_find.content := "Next matching string";
      endif
```

```
    }
  }

  child pushbutton Pb_find
  {
    .text "Search item";

    on select
    {
      Lb:Showfind(Et_find.content);
    }
  }

  child edittext Et_find
  {
     .content "Which item";
  }
}
```

See also

Built-in function find()

2.16 :findtext()

The **:findtext()** method searches for a given string in the text (*.content*) of an edittext.

Definition

```
integer :findtext
(
      string  Text  input
  { , integer Start := 1  input,
      integer End   := -1 input }
  { , boolean CaseSensitive := true input }
)
```

Parameters

string *Text* input

This parameter passes the string to be searched for in the content of the edittext.
An empty text is found immediately. The return value of **:findtext()** will then be *Start* if a range was specified and the cursor position or the end position of the selection if no range was specified.
For the RTF edittext (*.options[opt_rtf] = true*), the text to be searched for has to be passed as plain text without formatting instructions.

*integer **Start** := 1 input*
*integer **End** := 1 input*

These optional parameters define the range in which to search for the specified string. If no range is given, the search starts at the cursor position or after the selection and continues until the end of the text.
The value range of *Start* and *End* goes from *0* to the number of characters in the displayed text, with every character that can be included in a selection counting. Line break characters therefore count as well.
-1 can be specified for *End* to search the contents of the edit text to its end.
With the RTF edittext (*.options[opt_rtf] = true*), *Start* and *End* – similar to *.startsel* and *.endsel* – refer to positions in the formatted text. They **cannot** be used to infer positions in the content string of the RTF edittext, since this also contains formatting instructions. If the *Start* or *End* parameters are greater than the text length, then the text length is used for the RTF edittext.

*boolean **CaseSensitive** := true input*

This optional parameter determines whether the search will be case-sensitive.

true
Upper and lower case of the found texts must match the specified string.

false
Upper and lower case of the found texts may differ from the specified string.

Return value

-1
: The specified string was not found.

>= 0
: Starting position for the first occurrence of the specified string in the text.

Objects with this method

edittext

2.17 :get()

With this method, the attributes of objects can be queried. This method is predefined for all objects, Models and Defaults.

Furthermore, it is possible to overwrite this method and thus to extend the way the attributes are queried by further actions (e.g. calculating attribute values only when they are actually needed).

You can find the attributes available for the relevant object type in the “Object Reference”.

Definition

```
anyvalue :get
(
      attribute Attribute input
  { , anyvalue  Index     input }
)
```

Parameters

attribute *Attribute* input

This parameter contains the attribute whose value is queried.

anyvalue ***Index*** *input*

In this optional parameter the index of the queried attribute is specified, if it is an indexed attribute. Its data type must correspond to the index data type of the attribute. Therefore, an *integer* value must be passed here if the attribute is one-dimensional or an *index* value if the attribute is two-dimensional.

Return value

Value of the queried attribute using `return`.

Passing `fail` in overwritten, predefined methods is also possible. To do so, there are the following keywords in the Rule Language:

Syntax

```
pass <expression> ;
```

Evaluates an expression similar to a `return` statement in order to return the result to the caller. If an error occurs, this error status is passed on to the caller. A normal `return` statement would terminate here and continue with the next statement.

Thus it virtually corresponds to the simplification of

```
variable anyvalue V;
if fail(V:=<expression>) then
  throw;
else
```

```
  return V;
endif
```

Syntax

```
throw ;
```

This keyword is used to signal an error to the caller and terminate the current method.

Example

```
dialog D
model edittext MEt {
  :get() {
    case Attribute
      in .text:
        if this.content="" then
          throw;
        else
          return this.content;
        endif
    endcase
    pass this:super();
  }
}
on dialog start {
  variable string S := "???";
  print fail(MEt.text);           // => true
  MEt.content := "Hello";
  print fail(S := MEt.text);      // => false
  print S;                        // => Hello
  print fail(MEt:get(.docking));  // => true
  exit();
}
```

Implicit Invocation

The **:get()** method is also invoked implicitly. Whenever an attribute (predefined or user-defined) is queried from the Rule Language or by an interface function (e.g. **DM_GetValue()**), the method **:get()** is called, which must then return the value of the attribute.

Redefinition

Redefinition of the **:get()** method is done like this:

```
window W {
  :get(<no parameters!>)
  {
```

```
    ...
    Actions to compute the value of the queried attribute.
    ...
  }
}
```

Within the method definition, the parameters *Attribute* and *Index* can be queried and set, in order to then call e.g. with *this:super()* the method **:get()** of the superclass with the accordingly changed parameters, which, however, is usually not useful.

Suppression of Recursive Calls

If within the method **:get()** any attribute of the same object for which the **:get()** method has already been called is queried, the method **:get()** should actually be invoked implicitly again for the same object. Since the danger of an infinite recursion is very high in this case (in many cases it would even be inevitable), such a recursive call is suppressed.

Example

```
window {
  :get()
  {
    case (Attribute)
      in .title:
        return "<" + this.title + ">";  // no implicit call of :get()
      otherwise:
        return this:super();
    endcase
  }
}
```

In this example, the **:get()** method monitors window title queries and returns the title enclosed in "<>" brackets. In the process, there is another read access to *this.title*. But an implicit call of the **:get()** method is suppressed. However, the **:get()** method of another object will still be invoked if the attributes of the other object are queried.

Call of the :get() Method on the Superclass

The above example also shows how to use *this:super()*. All attributes for which the overwritten method **:get()** does not feel responsible should be delegated to the superclass, so that the predefined method **:get()** will return the attribute value at some point.

Example

The following example illustrates the call hierarchy of the **:get()** method.

```
dialog GET
```

```
model record MRec1 {
  string Str;

  :get()
  {
    if Attribute = .Str then
      return "MRec1(" + this.Str + ").";
    else
      return this:super();
    endif
  }
}

model MRec1 MRec2 {
  :get()
  {
    if Attribute = .Str then
      return this:super() + "MRec2(" + this.Str + ").";
    else
      return this:super();
    endif
  }
}

MRec2 RecInst {
  :get()
  {
    if Attribute = .Str then
      return this:super() + "RecInst(" + this.Str + ")";
    else
      return this:super();
    endif
  }
}

on dialog start {
  RecInst.Str := "X";
  // implicit invocation of the :get() method
  print "*** Result Str: " + RecInst.Str;
    // should be "MRec1(X).MRec2(X).RecInst(X)"
  // explicit invocation of the :get() method
  print "*** Result Str: " + RecInst:get(.Str);
    // should be "MRec1(X).MRec2(X).RecInst(X)"

  exit();
}
```

See also

Built-in function getvalue() in manual “Rule Language”

C function DM_GetValue in manual “C Interface - Functions”

2.18 :getformat()

The **:getformat()** method can be used to query the formatting of a text range in an RTF edittext. **:getformat()** returns a value that reflects the specification of the provided formatting type (e.g. font, font color, text alignment).

Definition

```
anyvalue :getformat
(
  { integer Start := 1  input,
    integer End   := -1 input, }
    enum    Type  input
)
```

Parameters

*integer **Start** := 1 input*
*integer **End** := -1 input*

These optional parameters define the range whose formatting shall be determined. If these parameters are missing, the range from *.startsel* to *.endsel* (selection or cursor position) is used.
The value range of *Start* and *End* goes from *0* to the number of characters in the displayed text, with every character that can be included in a selection counting. Line break characters therefore count as well.
-1 can be specified for *End* to get the formatting of the remaining text from the start position.
Start and *End* – similar to *.startsel* and *.endsel* – refer to positions in the formatted text. They **cannot** be used to infer positions in the content string of the RTF edittext, since this also contains formatting instructions. If the *Start* or *End* parameters are greater than the text length, then the text length is used for the RTF edittext.
If no characters are selected on invocation (*.startsel* = *.endsel*) or *Start* = *End*, the format **left** of the specified position will be returned.

enum *Type* input

This parameter specifies the type of formatting whose value shall be queried. The *enum* values for the formatting types are listed in the table below.

Return value

anyvalue

A value for the specification of the given formatting type.
If the range for which the formatting is queried contains multiple formattings of the specified type, *nothing* is returned.

Formatting Kinds, *enum* Values and Data Types of the Formatting Types

Formatting Type	Formatting Kind	*enum* Value	Data Type
typeface, font	character format	*text_font*	*string*
font size	character format	*text_size*	*integer*
foreground color, text color	character format	*text_fgc*	*integer*
background color	character format	*text_bgc*	*integer*
bold	character format	*text_bold*	*boolean*
italic	character format	*text_italic*	*boolean*
underlined	character format	*text_underline*	*boolean*
left indentation	paragraph format	*text_indent_left*	*integer*
right indentation	paragraph format	*text_indent_right*	*integer*
indentation of continuation lines in a paragraph	paragraph format	*text_indent_offset*	*integer*
text alignment	paragraph format	*text_align*	*enum* [*align_left*, *align_right*, *align_center*, *align_justify*]

Unit of Measure for Sizes and Positions

Sizes and positions are specified in “twips” – as usual in Windows programming. A twip is equivalent to *1/20 point*, that is *1/1440 inch* or *1/567 cm*.

Explanations on Particular Formattings

- The value for *text_indent_right* is an absolute specification of the indentation from the right margin.
- In contrast, *text_indent_left* is a relative specification that can be used several times. The indents from the left margin are accumulated, but can never extend beyond the margin. The value can be negative and then means that a paragraph is shifted to the left, while positive values mean a shift to the right.
- The value of *text_indent_offset* determines how far all lines of a paragraph except the first are shifted to the right (positive value) or to the left (negative value).
- For *text_font* **:getformat()** returns a string with the font name.
- For foreground color (*text_fgc*) and background color (*text_bgc*) *integer* values are returned which should be interpreted as RGB values.

Reliability of Return Values

:getformat() can only work properly if the RTF edittext is visible (*.visible = true*), because the underlying Windows object performs some formatting calculations only in visible state. For example, the font information cannot be queried in the invisible state.

Objects with this method

edittext with *.options[opt_rtf] = true*

2.19 :gettext()

The **:gettext()** method returns text from the contents of an edittext.

Definition

```
string :gettext
(
  { integer Start input,
    integer End   := -1 input, }
  { enum    Type  := content_plain input }
)
```

Parameters

*integer **Start** input*
*integer **End** := -1 input*

These optional parameters define the range whose text shall be returned. If no range is specified, the text from *.startsel* to *.endsel* is returned, that is, the contents of the selection.
The value range of *Start* and *End* goes from *0* to the number of characters in the displayed text, with every character that can be included in a selection counting. Line break characters therefore count as well.
-1 can be specified for *End* to get the remaining text from the start position.
With the RTF edittext (*.options[opt_rtf] = true*), *Start* and *End* – similar to *.startsel* and *.endsel* – refer to positions in the formatted text. They **cannot** be used to infer positions in the content string of the RTF edittext, since this also contains formatting instructions. If the *Start* or *End* parameters are greater than the text length, then the text length is used for the RTF edittext.

*enum **Type** := content_plain input*

This optional parameter can be used with the RTF edittext to define whether the content of the text range will be returned as unformatted, plain text or as RTF including the formatting instructions.

Value range

content_plain
String contains plain, unformatted text.
content_rtf
String contains RTF text.
Only usable when *.options[opt_rtf] = true*.

Return value

String with the content of the specified text range.

Objects with this method

edittext

2.20 :has()

With this method it can be queried dynamically at runtime whether the object has a certain user-defined attribute or user-defined method.

The method **cannot** be used to query the existence of predefined attributes and methods.

Definition

```
boolean :has
(
      anyvalue MethodOrAttr input
  { , anyvalue Index        input }
)
```

Parameters

anyvalue *MethodOrAttr* input
The user-defined attribute or user-defined method whose existence shall be checked.

anyvalue ***Index*** *input*
For user-defined attributes, this optional parameter is used to query whether the attribute is available and if it can be accessed with the indicated index.

Return value

true
The attribute or method is available at the object.

false
The attribute or method is **not** available at the object.

Objects With This Method

» All objects that may have user-defined attributes.

» All objects that may have user-defined methods.

Example

```
dialog Example
record Rec
{
  string String;
  rule void Method {}
}

on dialog start
{
  print Rec:has(:Method);  // returns true
  print Rec:has(.String);  // returns true
```

```
  print Rec:has(:Print);   // returns false
  print Rec:has(.Data);    // returns false
}
```

2.21 :index()

Dialog Manager offers the possibility to go through an entire associative array. The method **:index()** is used to calculate the index from the internal structure.

Attention

The internal structure of associative arrays is used for the internal administration only. This internal structure can vary. In case you need structured data, which usually is not necessary, you will have to calculate the structure yourself.

Definition

```
anyvalue :index
(
  attribute Attribute     input,
  integer   InternalIndex input
)
```

Parameters

attribute *Attribute* input

In this parameter the name of the associative array, whose contents are to be queried, is specified.

integer *InternalIndex* input

In this parameter the current index to be queried is specified. Please note that this index may change after executing delete or allocation actions in the array.

Return value

0
: element is not contained in the array

others
: index of element in the associative array

Objects with this method

All objects that may have user-defined attributes.

Example

```
window WMain
{
  integer Channel [ string ];
    .Channel [ "MTV" ] := 65;
    .Channel [ "CNN" ] := 14;
    .Channel [ "NBC" ] := 11;
}
```

```
rule void PrintStation()
{
  variable integer I;
  for I := 1 to WMain.itemcount[ .Channel ] do
    print WMain.Channel[ WMain:index(.Channel, I) ];
  endfor
}
```

In the loop the array accesses *1* up to the number of items, according to the internal structure. Within the loop the index is calculated (*WMain:index(.Channel, I)*). This index is used to index the associative array ".Channel".

See also

Chapter "Working with Associative Arrays" in manual "User-defined Attributes and Methods"

2.22 :init()

The **:init()** method is a predefined, overwritable method that is always invoked internally after the creation of an object. The method can be overwritten to define an own initialization. The **:init()** method enables to influence the sequence of initialization, e.g. initializing the parent object before its children or performing a partial initialization of the parent, then initialize its children and finally perform the remaining initialization of the parent.

The **:init()** method cannot be invoked directly. It provides preset parameters that can be accessed within the method. The method does not have a return value (i.e. *void*).

Definition

```
void :init
(
  object  Model     input,
  object  Parent    input,
  object  Module    input,
  integer Type      input,
  boolean Invisible input
)
```

Parameters

The predefined parameters are preset as follows:

object *Model* input
Is set to `this.model`.

object *Parent* input
Is set to `this.parent`.

object *Module* input
Is set to `this.module`.

integer *Type* input
Is set to `this.scope`.

boolean *Invisible* input
Is set according to the respective parameter of the **create()** call.

Redefinition

The method can be redefined directly at the object.

```
:init()
{
  // Rule Language code
}
```

Please note that neither the predefined parameters are indicated nor additional parameters may be defined when redefining the method.

Invocation

The method cannot be called directly. It can be triggered only either by loading a module or dialog or by creating new objects dynamically with the functions **create()** or **DM_CreateObject()**. Direct invocations are ignored or rather lead to errors.

Event Processing After Loading

The **:init()** method is executed before the *start* rule of the dialog. Possible events are ignored and therefore not processed. This means that *changed* events are not processed either and no *on changed* rules are triggered.

this:super()

Using Model Methods

Frequently, initialization methods are defined at the model and those shall be executed as well. They can be invoked by calling `this:super()` within the **:init()** method of an instance. Please note that `this:super()` is always called without parameters.

Controlling Initialization of the Children

If no more model method exists, calls of `this:super()` invoke the **:init()** methods of the children. When there are no children, the call returns without error. When the children have no **:init()** methods, the **:init()** methods of the children's children are invoked. In this way it can be controlled whether and when the children are initialized. Please note, however, that the children of the dialog are **not** controlled by `this:super()`.

Important Note

Using **fail()** and *pass this:super()* in the **:init()** method should be absolutely avoided. In either case, occurring errors are no longer output to the trace or log files (output of *"***ERROR IN EVAL"* is ceased). This hinders error detection considerably up to preventing it completely.

Example 1

```
dialog D1

model window MW
{
  :init()
  {
    static variable integer I := 1;
    !! take only instances into account
    if Type = 3 then
      this.title := "Window No. " + I;
      I := I + 1;
```

```
    endif
  }
}
```

Example 2

```
dialog D2

model record MRecWin
{
  !! some important data
  string Important[100];
}

model window MW
{
  !! store the corresponding record here
  object MyRec;
  :init()
  {
    this.MyRec := create(MRecWin, D);
  }
}
```

Remark

The previous example serves just for illustration. It may be expressed much easier:

```
dialog D2_Improved

 model window MW
{
  record MyRec
  {
    !! some important data
    string Important[100];
  }
}
```

Example 3

```
dialog D3

model window MW
{
  child groupbox G
  {
```

```
    :init()
    {
      !! do something with G
    }
  }
  :init()
  {
    !! do something before :init() of the children is invoked
    !! ...
    this:super();
    !! do something after :init() of the children has been executed
    !! ...
  }
}
```

Example 4

```
dialog D4

model window MW1
{
  child groupbox G
  {
    :init()
    {
      !! do something with G
    }
  }
  :init()
  {
    if Type = 3 then
      !! G is initialized only if the this object is no Model
      this:super ();
    endif
  }
}

model MW1 MW2
{
  :init()
  {
    !! do something with the object, here the method of MW1 is invoked
    this:super ();
    !! further operations on the object
  }
}
```

Example 5

```
dialog D5

model groupbox MG
{
  :init()
  {
    print Parent;
    print Model;
    print Module;
    print Type;
    print Invisible;
  }
}
window W
{
}
on dialog start
{
  MG:create(W);
}
```

Output

```
W
MG
D5
3
false
```

2.23 :insert()

There are two different forms of the **:insert()** method:

» **Inserting entries in list objects**

» **Inserting items in fields (indexed user-defined attributes)**

2.23.1 :insert() (List Objects)

This method inserts rows and columns into the contents of an object whose contents consist of multiple entries indexed with *integer* or *index*. Inserting is also possible if there are no entries yet. When inserting, the associated attributes are also extended accordingly. The newly inserted entries are “empty” or have the default value if available.

Definition

```
boolean :insert
(
      integer Position input
  { , integer Count := 1 input }
  { , boolean Direction := false input }
)
```

Parameters

integer *Position* input

This parameter defines the position where the new rows or columns are inserted. If its value is *0*, the rows or columns are appended to the end.

*integer **Count** := 1 input*

This optional parameter defines the number of rows or columns that are inserted. If the parameter is not specified, *1* is taken as default.

*boolean **Direction** := false input*

This optional parameter controls whether rows or columns are inserted in the ***tablefield***. This depends on the value of the ***tablefield*** attribute *.direction*.
The parameter may only be passed for the ***tablefield***. For other objects than the ***tablefield***, passing the parameter will result in an error.

Value range

false
: Inserts against the direction given in the *.direction* attribute

true
: Inserts in the same direction as given in the *.direction* attribute

Thus these contents are inserted in the ***tablefield***:

Parameter *Direction*	*.direction 1* (vertical)	*.direction 2* (horizontal)
false	row(s)	column(s)
true	column(s)	row(s)

Return value

The method returns *true* if the rows or columns could be inserted.

If an error has occurred when inserting, the method will return a `fail`.

Objects with this method

- listbox (attribute *.content[integer]*)
- poptext (attribute *.text[integer]*)
- spinbox (attribute *.text[integer]*)
- tablefield (attribute *.content[index]*)
- treeview (attribute *.content[integer]*)

Example

```
dialog D

window Wn
{
  .title "Example for the method :insert()";

  child treeview Tv
  {
    .content[1] "Company A";
    .content[2] "Miller";
    .content[3] "Mayer";
    .content[4] "Company B";
    .content[5] "Jones";
    .content[6] "Smith";
    .content[7] "Fisher";
    .style[style_lines]   true;
    .style[style_buttons] true;
    .style[style_root]    true;
    .level[2] 2;
    .level[3] 2;
    .level[5] 2;
    .level[6] 2;
    .level[7] 2;
  }

  child pushbutton Pb
```

```
  {
    .text "Add new employee";

    on select
    {
      if fail(atoi(Et_level.content)) then
        Et_level.content := "You have not entered a level";
      else
        Tv:insert((Tv.activeitem + 1));
        Tv.level[(Tv.activeitem + 1)] := atoi(Et_level.content);
        Tv.content[(Tv.activeitem + 1)] := Et_name.content;
        Et_level.content := "On which level?";
      endif
    }
  }
  child edittext Et_name
  {
    .content "Enter name here";
  }
  child edittext Et_level
  {
    .content "On which level?";
  }
}
```

2.23.2 :insert() (Arrays)

This method inserts new items into arrays (indexed, non-associative, user-defined attributes). The newly inserted items are “empty” or have the default value if available.

Definition

```
boolean :insert
(
      attribute Attr input,
      integer Position input
  { , integer Count := 1 input }
)
```

Parameters

attribute *Attr* input

This parameter defines the user-defined attribute on which the method should be applied.

integer *Position* input

In this parameter the index where the new elements should be inserted is passed. If its value is 0, the elements are appended to the end.

integer ***Count*** *:= 1 input*

This optional parameter defines the number of items that are inserted. If the parameter is not specified, *1* is taken as default.

Return value

The method returns *true* if the items could be inserted.

If an error has occurred when inserting, the method will return a `fail`.

Objects with this method

All objects that may have user-defined attributes.

See also

Chapter “Methods for Arrays of User-defined Attributes” in manual “User-defined Attributes and Methods”

2.24 :instance_of()

This method examines if the object is directly or indirectly derived from a particular Model.

Definition

```
boolean :instance_of
(
  object Model input
)
```

Parameters

object *Model* input

The Model that shall be searched in the *.model* chain of the object.

Return value

The method returns *true* if the Model given in the *Model* parameter is contained in the *.model* chain of the object, *false* otherwise.

See also

Attribute model

2.25 :load()

This method loads an XML Document from a URL or file. The stored DOM tree is deleted and a new tree is built up. All existing XML Cursors become invalid.

The attribute *.mapped* has the value *false* for invalid XML Cursors.

Definition

```
boolean :load
(
  string URL input
)
```

Parameters

string *URL* input
The location of the XML Document to be loaded. May be a URL or a file system path.

Return value

The method returns *true* if the XML Document could be loaded, *false* otherwise.

In case of an error, the DOM tree either remains unchanged or is deleted completely; there will never be a partial transformation.

Objects with this method

document

2.26 :match()

This method checks whether the current DOM node, to which the ***XML Cursor*** points, matches a certain pattern.

Definition

```
boolean :match
(
  string Pattern input
)
```

Parameters

string *Pattern* input

The pattern to compare the DOM node against.
The pattern syntax is described in chapter “Pattern for the Methods :match() and :select()” of manual “XML Interface”.

Return value

The method returns *true* if the DOM node matches the given pattern, *false* otherwise.

Please note that an XML Cursor, whose attribute *.mapped* possesses the value *false*, will automatically be positioned to the root of the DOM Document.

Objects with this method

doccursor

2.27 :move()

There are two different forms of the **:move()** method:

» **Moving entries in list objects**

» **Moving items in fields (indexed user-defined attributes)**

2.27.1 :move() (List Objects)

This method moves rows and columns to another position within the contents of an object whose contents consist of multiple entries indexed with *integer* or *index*. Along with the contents, the associated attributes are moved accordingly.

Definition

```
void :move
(
      integer Position input,
      integer Target input
  { , integer Count := 1 input }
  { , boolean Direction := false input }
)
```

Parameters

integer *Position* input

This parameter defines the first row or column to be moved.

integer *Target* input

This parameter specifies where the rows or columns should be moved to.

*integer **Count** :=1 input*

This optional parameter defines the number of rows or columns that are moved. If the parameter is not specified, *1* is taken as default.

*boolean **Direction** := false input*

This optional parameter controls whether rows or columns are moved in the ***tablefield***. This depends on the value of the ***tablefield*** attribute *.direction*.
The parameter may only be passed for the ***tablefield***. For other objects than the ***tablefield***, passing the parameter will result in an error.

Value range

false
Moves against the direction given in the *.direction* attribute

true
Moves in the same direction as given in the *.direction* attribute

Thus these contents are moved in the ***tablefield***:

Parameter *Direction*	*.direction 1* (vertical)	*.direction 2* (horizontal)
false	row(s)	column(s)
true	column(s)	row(s)

Return value

None.

If an error has occurred when moving, the method will return a `fail`.

Objects with this method

- listbox (attribute *.content[integer]*)
- poptext (attribute *.text[integer]*)
- spinbox (attribute *.text[integer]*)
- tablefield (attribute *.content[index]*)
- treeview (attribute *.content[integer]*)

Example

```
dialog MethodMove

model pushbutton MPb {
  .xleft 216;
  .width 72;
  .sensitive false;
  .datamodel Lb;
  .dataget[.sensitive] .activeitem;
  integer Direction ::= 0;
  boolean Max ::= false;

  on select {
    this.datamodel:MoveItem(this.Direction, this.Max);
  }
}

model MPb MPbUp {
  .Direction ::= -1;
  :represent() {
    case Attribute
      in .sensitive:
        Value := (Value > 1);
        pass this:super();
      otherwise:
        // Handle other attributes
    endcase
```

```
  }
}

model MPb MPbDown {
  .Direction ::= 1;
  :represent() {
    case Attribute
      in .sensitive:
        Value := (Value > 0 and Value < this.datamodel
[.sensitive].itemcount);
        pass this:super();
      otherwise:
        // Handle other attributes
    endcase
  }
}

window Wn {
  .width 320;
  .height 240;
  .title "IDM Example: Method :move()";

  on close { exit(); }

  statictext St {
    .xleft 12;
    .ytop 12;
    .sensitive false;
    .text "Select an item and move it with the buttons";
  }

  listbox Lb {
    .xleft 12;
    .width 180;
    .yauto 0;
    .ytop 36;
    .ybottom 12;
    .selstyle single;
    .activeitem 0;
    .content[1] "Rome";
    .content[2] "Stockholm";
    .content[3] "Madrid";
    .content[4] "London";
    .content[5] "Oslo";
    .content[6] "Berlin";
    .content[7] "Paris";
    .content[8] "Lisbon";
```

```
    .content[9] "Helsinki";

    on select {
      this:propagate();
    }

    rule void MoveItem(integer Direction, boolean Max) {
      if Max then
        if Direction = -1 then
          this:move(this.activeitem, 1, 1);
        endif
        if Direction = 1 then
          this:move(this.activeitem, this.itemcount, 1);
        endif
      else
        this:move(this.activeitem, this.activeitem + Direction, 1);
      endif
      this:propagate();
    }
  }

  MPbUp PbFirst {
    .ytop 36;
    .text "First";
    .Max ::= true;
  }
  MPbUp PbUp {
    .ytop 72;
    .text "Up";
  }
  MPbDown PbDown {
    .ytop 108;
    .text "Down";
  }
  MPbDown PbLast {
    .ytop 144;
    .text "Last";
    .Max ::= true;
  }
}
```

2.27.2 :move() (Arrays)

This method moves items of an array (indexed, non-associative, user-defined attribute) to another position in the array.

Definition

```
void :move
(
  attribute Attr input,
  integer Position input,
  integer Target input,
  integer Count input
)
```

Parameters

attribute *Attr* input
This parameter defines the user-defined attribute on which the method should be applied.

integer *Position* input
This parameter specifies the index of the first item to be moved.

integer *Target* input
This parameter determines where the items should be moved to.

integer *Count* input
This parameter defines how many items are moved.

Return value

None.

If an error has occurred when moving, the method will return a `fail`.

Objects with this method

All objects that may have user-defined attributes.

See also

Chapter “Methods for Arrays of User-defined Attributes” in manual “User-defined Attributes and Methods”

2.28 :openpopup()

This method opens the pop-up menu of the object, provided a menu has been defined.

Definition

```
boolean :openpopup
(
)
```

Parameters

None.

Return value

true
: The pop-up menu could be opened.

false
: The pop-up menu could **not** be opened. Perhaps no pop-up menu has been defined at the object.

Objects with this method

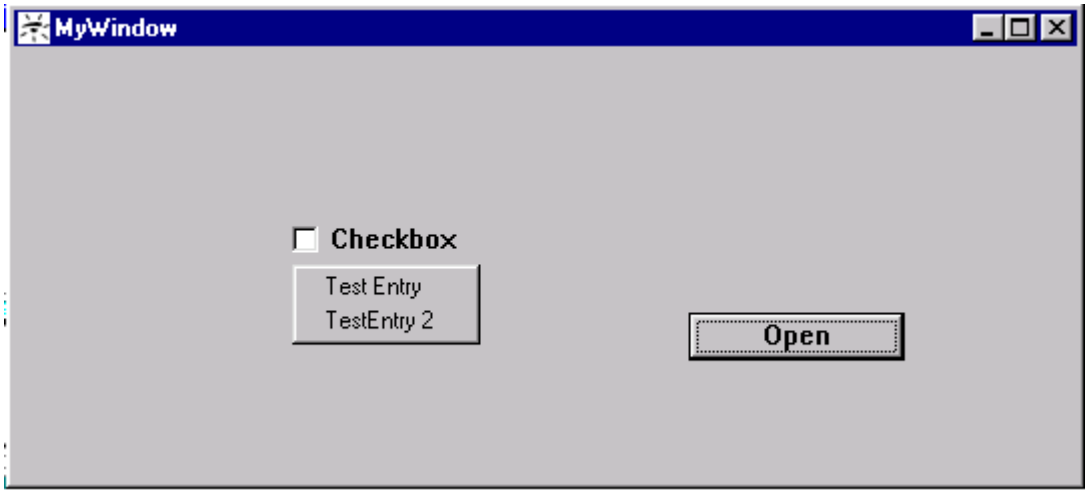
- canvas
- checkbox
- control
- edittext
- groupbox
- image
- layoutbox
- listbox
- notebook
- notepage
- poptext
- progressbar
- pushbutton
- radiobutton
- rectangle
- scrollbar
- spinbox
- splitbox

- statictext
- statusbar
- tablefield
- toolbar
- treeview

The objects ***menubox*** and ***window*** do **not** have this method.

Example

```
window Wn
{
  .title "MyWindow";
  child pushbutton POpen
  {
    .xleft 37;
    .ytop 5;
    .height 1;
    .text "Open";
    on select
    {
      Cb:openpopup();
    }
  }
  child checkbox Cb
  {
    .xleft 15;
    .width 24;
    .ytop 3;
    .height 1;
    .text "Checkbox";
    .state state_unchecked;
    child menubox Mb
    {
      .title "MyMenu";
      child menuitem
      {
        .text "Test Entry";
      }
      child menuitem
      {
        .text "TestEntry 2";
      }
    }
  }
}
```



See also

Attribute .menu

2.29 :parent()

This method is used to query the parent of an element within the treeview object.

Definition

```
integer :parent
(
  integer Index input
)
```

Parameters

integer *Index* input

In this parameter the index of the element whose parent is to be calculated is specified.

Return value

0

no valid element specified

others

index of parent in tree

Objects with this method

treeview

Example

```
window Wn
{
  .title "Example for method :parent()";
  child treeview Tv
  {
    .content[1] "Company A";
    .content[2] "Miller";
    .content[3] "Meyer";
    .content[4] "Company B";
    .content[5] "Davis";
    .content[6] "Smith";
    .content[7] "Fisher";
    .style[style_lines]   true;
    .style[style_buttons] true;
    .style[style_root]    true;
    .level[2] 2;
    .level[3] 2;
    .level[5] 2;
    .level[6] 2;
    .level[7] 2;
```

```
    integer Index;
  }
  child statictext St
  {
    .text "No value";
  }
  child pushbutton PbPar
  {
    .text "Define company of a person";
    on select
    {
      if (Tv:parent(Tv.activeitem) = 0) then
        St.text := "Please select person";
      else
        St.text := Tv.content[Tv:parent(Tv.activeitem)];
      endif
    }
  }
}
```

2.30 :parent_of()

This method examines if the object is a direct or indirect parent of a particular child object, i.e. if the object may be accessed through recursive queries of *.parent* starting from the child object.

Definition

```
boolean :parent_of
(
  object Child input
)
```

Parameters

object *Child* input

The child object that shall be searched among the children of the object.

Return value

The method returns *true* if the object given in the *Child* parameter is a direct or indirect child of the object, *false* otherwise.

See also

Attribute parent

2.31 :propagate()

When this method is called on a Model component, all linked View components are requested to update their View attributes which are linked to this Model object.

Normally calling this method should not be necessary because synchronization between View and Model is controlled through the *.dataoptions[]* attribute.

Linkage must have been established with the attributes *.datamodel* and *.dataset* on the View component.

There will be no error message if the Model object has no linked attributes.

Definition

```
void :propagate
(
)
```

Example

```
dialog D

timer Ti
{
  .starttime "+00:00:03";
  .incrtime "+00:00:01";
  .active true;
  string Time := "??:??";
  integer Min:=0;
  integer Sec:=0;
  integer Counter := 100;
  .dataoptions[dopt_propagate_on_start] false;
  .dataoptions[dopt_propagate_on_changed] false;

  on select
  {
    this.Counter := this.Counter-1;
    if this.Counter<0 then
      exit();
      return;
    endif
    if this.Sec=59 then
      this.Min := (this.Min + 1) % 60;
    endif
    this.Sec := (this.Sec + 1) % 60;
    this.Time := sprintf("%02d:%02d", this.Min, this.Sec);
    this:propagate();  /* propagate explicitly every second */
  }
```

```
}

window WiTime
{
  .width 200;
  .title ":propagate demo";
  .datamodel Ti;
  .dataoptions[dopt_represent_on_map] false;

  statictext StMin
  {
    .xauto 0;
    .dataget .Time;
  }

  statictext StCounter
  {
    .xauto 0;
    .ytop 50;
    .dataget .Counter;
    .text "Please wait...";

    :represent()
    {
      if Attribute=.text then
        Value := "Time to exit: "+Value;
      endif
      return this:super();
    }
  }

  on close { exit(); }
}

window WiMinSec
{
  .title ":propagate demo - min/sec";
  .width 200;  .height 200;
  .xleft 250;

  tablefield Tf
  {
    .xauto 0;  .yauto 0;
    .rowcount 2;  .colcount 2;
    .rowheader 1;
    .colwidth[0] 50;  .rowheight[0] 30;
    .content[1,1] "Min";
```

```
    .content[1,2] "Sec";
    .dataoptions[dopt_represent_on_map] false;
    .datamodel Ti;
    .dataget[.field] .Sec;
    .dataget[.content] .Min;
    .dataindex[.field] [1,2];
    .dataindex[.content] [2,1];
  }

  on close { exit(); }
}
```

2.32 :recall()

This method can call any predefined and user-defined methods. This is necessary, for example, if the method to be called is calculated itself or stored in a variable.

Calling the **:recall()** method indirectly invokes the specified method and returns the return value of the invoked method. The number of parameters that can be passed when using **:recall()** is limited to *15*. This is important when designing own methods.

Definition

```
anyvalue :recall
(
      method Method  input
  { , anyvalue Par1  input }
      ...
  { , anyvalue Par15 input }
)
```

Parameters

method *Method* input
The method to be invoked is passed in this parameter.

*anyvalue **Par1** input*
...
*anyvalue **Par15** input*
The parameters of the method to be caled are passed in these parameters. The specified parameters must correspond in number and data types to the parameters of the method to be called.

Return value

Return value of the method to be called.

Forcing the Recursive Call of Other Methods

The **:recall()** method can be used to force a recursive call of other methods, e.g. of **:get()** and **:set()**. This distinguishes it from the **:call()** method.

Example

```
dialog RECALL

record Rec
{
  integer Progress :=   0;
  integer Max := 100;

  :set() {
    case (Attribute)
```

```
    in .Progress:
      if Value < 0 then
        Value := 0;
      else
        if Value > this.Max then
          Value := this.Max;
        endif
      endif
      this:super();
    in .Max:
      this:super();
      if this.Max<this.Progress then
        // The consistency check for .Progress should be carried out
        // again. For this purpose :set() is called recursively.
        this:recall(:set, .Progress, this.Max, false);
      endif
    endcase
  }
}
on dialog start
{
  Rec.Progress := 90;  // Target .Progress = 90.
  Rec.Max := 50;       // .Progress must also be adjusted.
                       // Target .Progress = 50, .Max = 50.
}
```

See also

Method **:call()**

2.33 :reparent()

This method reallocates sub-nodes within tree structures. The sub-nodes are moved with their entire sub-trees. There are two different forms of the **:reparent()** method:

» **Moving sub-trees of a treeview**
» **Moving nodes of an XML Document**

2.33.1 :reparent() (treeview)

With the **:reparent()** method any sub-tree can be reallocated within a ***treeview***.

Definition

```
boolean :reparent
(
  integer Start  input,
  integer Count  input,
  integer Parent input,
  integer Target input
)
```

Parameters

integer *Start* input
In this parameter the index of the sub-tree to be moved is specified.

integer *Count* input
This parameter defines how many sub-trees are moved.

integer *Parent* input
In this parameter the index of the new parent node of the reallocated sub-tree is determined.

integer *Target* input
This parameter indicates the absolute index of the node after which the moved sub-tree is inserted.

Return value

true
The sub-tree has been moved successfully.
false
Moving the sub-tree failed.

Objects with this method

treeview

Example

```
dialog D

window Wn

{
  .title "Example for the :reparent() method";

  child treeview Tv
  {
    .content[1] "Company A";
    .content[2] "Miller";
    .content[3] "Mayer";
    .content[4] "Company B";
    .content[5] "Jones";
    .content[6] "Smith";
    .content[7] "Fisher";
    .style[style_lines] true;
    .style[style_buttons] true;
    .style[style_root] true;
    .level[2] 2;
    .level[3] 2;
    .level[5] 2;
    .level[6] 2;
    .level[7] 2;

    boolean Select := false;
    integer Index := 0;

    on select
    {
      if Tv.Select then
        Tv:reparent(Tv.Index, 1, Tv.activeitem, Tv.activeitem);
        Tv.Select := false;
        St.text := "";
      endif
    }
  }

  child pushbutton Pb

{
    .text "Link company";

    on select
    {
      variable integer Index;
```

```
      Tv.Index := Tv.activeitem;
      St.text := "Select new company";
      Tv.Select := true;
    }
  }

  child statictext St
  {
    .text "";
  }
}
```

2.33.2 :reparent() (doccursor)

The **:reparent()** method reallocates the DOM node that the ***XML Cursor*** points to with all of its child nodes within an ***XML Document***.

If the value of the *.path* attribute is stored elsewhere (e.g. in the *.userdata* attribute), it should be noted that the stored value is not adjusted when the structure of the DOM tree changes. When the **:select()** method is invoked with the stored value afterward, the XML Cursor may be pointing to an incorrect DOM node.

Definition

```
boolean :reparent
(
      object  Parent input
  { , integer Index  := -1 input }
)

boolean :reparent
(
  integer Index input
)
```

Parameters

object *Parent* input

This parameter indicates a ***doccursor*** that points to the new parent node. Indicates the new parent node.
When the *Index* parameter is missing, the reallocated DOM node is appended as last child of the new parent node.

integer *Index* input

This parameter defines the position where the moved DOM node is inserted. With the value *-1*, the reallocated DOM node is appended as last node.

Return value

The method returns *true* if the DOM node has been moved successfully, *false* otherwise.

Objects with this method

doccursor

2.34 :replacetext()

The **:replacetext()** method substitutes text in an edittext with another text.

Definition

```
boolean :replacetext
(
  { integer Start := 1  input,
    integer End   := -1 input, }
  { enum    Type  := content_plain input, }
    string  Text  input
)
```

Parameters

*integer **Start** := 1 input*
*integer **End** := -1 input*

These optional parameters define the range to be replaced by the specified string. If no range is specified, the selection, that is, the range from *.startsel* to *.endsel*, is replaced by the specified string. *.startsel* and *.endsel* are placed at the end of this text, i.e. after the replacement, the cursor is positioned behind the inserted text. If a range is specified, *.startsel* and *.endsel* retain their values, unless the replacement makes this impossible.
The value range of *Start* and *End* goes from *0* to the number of characters in the displayed text, with every character that can be included in a selection counting. Line and paragraph breaks therefore also count as characters.
-1 can be specified for *End* to replace the remaining text from the start position.
With the RTF edittext (*.options[opt_rtf] = true*), *Start* and *End* – similar to *.startsel* and *.endsel* – refer to positions in the formatted text. They **cannot** be used to infer positions in the content string of the RTF edittext, since this also contains formatting instructions. If the *Start* or *End* parameters are greater than the text length, then the text length is used for the RTF edittext.

*enum **Type** := content_plain input*

For the RTF edittext, this optional parameter specifies whether the string to be inserted contains plain, unformatted text or RTF text.

Value range

content_plain
String contains plain, unformatted text.
content_rtf
String contains RTF text.
Only usable when *.options[opt_rtf] = true*.

string *Text* input

This parameter passes the string that will replace text in the RTF edittext.

Return value

true
: Text range in the edittext has been replaced by the specified string.

false
: Error: There was no substitution.

Objects with this method

edittext

2.35 :represent()

This method has two functions.

1. When called without parameters, all linked attributes at the View component are updated. This happens for all children and their children as well.
   This is done by retrieving the data value for each View attribute from the corresponding Model component and invoking the internal and redefinable **:represent()** method (with parameters *Value*, *Attribute*, *Index*) to represent it through the object.
   Linkage to the Model component must have been established with the attributes *.datamodel* and *.dataset*.

   Important
   Direct invocation with parameters is not supported by the IDM.
2. The redefinable **:represent()** method enables the dialog programmer to intervene before the actual assignment of the data value to the presentation object. This allows to interpose required conversions or transformations. The redefined method should be completed with `pass this:super();` to ensure the normal behavior of the IDM respectively make use of the automatic type conversion.

   Note
   The redefinable method **:set()** is not invoked for assigning data values to the presentation object.

Definition

```
void :represent
(
)
```

Redefinable Method

```
void :represent
(
  anyvalue  Value     input,
  attribute Attribute input,
  anyvalue  Index     input
)
```

Parameters

anyvalue *Value* input
This parameter contains the data value to be presented.

attribute *Attribute* input
This parameter indicates the View attribute where the data value shall be represented.

anyvaluet *Index* input
This parameter defines the index value for presenting the data value in the View attribute.

Example

```
dialog D
default statictext { .sensitive false; }

record Rec
{
  .dataoptions[dopt_propagate_on_start] false;
  string Name := "Henry Walter";
  string ZIP  := "80246";
}

window Wi
{
  .title ":represent demo";
  .width 200;  .height 150;
  .datamodel Rec;
  .dataoptions[dopt_represent_on_map] false;

  statictext
  { .text "Name"; .width 100; }

  edittext EtName
  {
    .xauto 0;  .xleft 100;
    .dataget .Name;
    .dataset .Name;

    on deselect_enter
    {
      this:apply();
    }
  }

  statictext
  { .text "ZIP";  .width 100;  .ytop 30; }

  edittext EtZIP
  {
    .ytop 30;  .xauto 0;  .xleft 100;
    .dataget .ZIP;
    .dataset .ZIP;
    .format "XNNN";

    on deselect_enter
    {
      this:apply();
    }
```

```
}

statictext
{ .text "State";  .width 100;  .ytop 60; }

poptext PtState
{
  .ytop 60;
  .xauto 0;  .xleft 100;
  .dataget[.activeitem] .ZIP;
  .dataset[.activeitem] .ZIP;
  .text[1]  "???";  .text[2]  "PA";
  .text[3]  "FL";   .text[4]  "IN";
  .text[5]  "WY";   .text[6]  "KY";
  .text[7]  "NE";   .text[8]  "MA";
  .text[9]  "CO";   .text[10] "TX";
  .text[11] "AK";

  :represent()
  {
    /* convert the XNNN-string into an index */
    if Attribute=.activeitem then
      Value := 2 + atoi("0" + substring("" + Value, 2, 1));
    endif
    pass this:super();
  }

  :retrieve()
  {
    if Attribute = .activeitem then
      if this.activeitem > 1 then
        return sprintf("%s%d%02d",
                       substring(this.text[this.activeitem], 1, 1),
                       this.activeitem - 2,
                       random(100));
      else
        return "";
      endif
    endif
    pass this:super();
  }

  on select
  {
    this:apply();
  }
}
```

```
  pushbutton PbRepresent
  {
    .text "Represent";
    .yauto -1;
    .width 100;  .xleft 50;

    on select
    {
      this.window:represent();  /* explicit represent of all views */
    }
  }

  statusbar Sb
  {
    statictext StInfo
    {
      .dataget .Name;
      .dataget[.At] .ZIP;
      string Name := "";
      string ZIP := "";

      :represent()
      {
        /* merge .Text & .At into a single string */
        case Attribute
        in .text:
          this.Name := toupper(Value);
        in .At:
          this.ZIP := Value;
        in .text, .At:
          this.text := sprintf("%s @ %s", this.Name, this.ZIP);
          return;
        endcase
        pass this:super();
      }
    }
  }

  on close { exit(); }
}
```

2.36 :retrieve()

This redefinable method is invoked for synchronization between View and Model components in the presentation object to retrieve the data value for a View attribute. This also happens implicitly when the **:apply()** or **:collect()** methods are called.

Important

Direct invocation is not supported by the IDM.

The default behavior of the method is to return the respective attribute as a single value or aggregated entire value. The redefinable method **:get()** is not invoked to retrieve the data values of the presentation object. This means that in the default behavior, always the actual attribute values are retrieved. Indexed attributes return the data value as a vector when the entire value is accessed (see also the **setvector()** and **getvector()** functions).

By redefining this method, its default behavior can be omitted or extended so that the values may be adapted.

Linkage to the Model component must have been established with the attributes *.datamodel* and *.dataset*.

Definition

```
anyvalue :retrieve
(
  attribute Attribute input,
  anyvalue  Index     input
)
```

Parameters

attribute *Attribute* input

This parameter defines the View attribute whose data value is to be queried.

anyvalue *Index* input

This parameter indicates the index value to be used when accessing the data value of the View attribute at the presentation object.

Return value

Data value appropriate to the attribute and index specification.

Example

```
dialog D
default statictext { .sensitive false; }

record Rec
```

```
{
  .dataoptions[dopt_propagate_on_start] false;
  string Name := "Henry Walter";
  string ZIP  := "80246";
}

window Wi
{
  .title ":represent demo";
  .width 200;  .height 150;
  .datamodel Rec;
  .dataoptions[dopt_represent_on_map] false;

  statictext
  { .text "Name"; .width 100; }

  edittext EtName
  {
    .xauto 0;  .xleft 100;
    .dataget .Name;
    .dataset .Name;

    on deselect_enter
    {
      this:apply();
    }
  }

  statictext
  { .text "ZIP";  .width 100;  .ytop 30; }

  edittext EtZIP
  {
    .ytop 30;  .xauto 0;  .xleft 100;
    .dataget .ZIP;
    .dataset .ZIP;
    .format "XNNN";

    on deselect_enter
    {
      this:apply();
    }
  }

  statictext
  { .text "State";  .width 100;  .ytop 60; }
```

```
poptext PtState
{
  .ytop 60;
  .xauto 0;  .xleft 100;
  .dataget[.activeitem] .ZIP;
  .dataset[.activeitem] .ZIP;
  .text[1]  "???";  .text[2]  "PA";
  .text[3]  "FL";   .text[4]  "IN";
  .text[5]  "WY";   .text[6]  "KY";
  .text[7]  "NE";   .text[8]  "MA";
  .text[9]  "CO";   .text[10] "TX";
  .text[11] "AK";

  :represent()
  {
    /* convert the XNNN-string into an index */
    if Attribute=.activeitem then
      Value := 2 + atoi("0" + substring("" + Value, 2, 1));
    endif
    pass this:super();
  }

  :retrieve()
  {
    if Attribute = .activeitem then
      if this.activeitem > 1 then
        return sprintf("%s%d%02d",
                       substring(this.text[this.activeitem], 1, 1),
                       this.activeitem - 2,
                       random(100));
      else
        return "";
      endif
    endif
    pass this:super();
  }

  on select
  {
    this:apply();
  }
}

pushbutton PbRepresent
{
  .text "Represent";
  .yauto -1;
```

```
    .width 100;  .xleft 50;

    on select
    {
      this.window:represent();  /* explicit represent of all views */
    }
  }

  statusbar Sb
  {
    statictext StInfo
    {
      .dataget .Name;
      .dataget[.At] .ZIP;
      string Name := "";
      string ZIP := "";

      :represent()
      {
        /* merge .Text & .At into a single string */
        case Attribute
        in .text:
          this.Name := toupper(Value);
        in .At:
          this.ZIP := Value;
        in .text, .At:
          this.text := sprintf("%s @ %s", this.Name, this.ZIP);
          return;
        endcase
        pass this:super();
      }
    }
  }

  on close { exit(); }
}
```

2.37 :save()

This method saves the XML Document as a file or under a URL.

Definition

```
boolean :save
(
  string URL input
)
```

Parameters

string *URL* input

The location where the XML Document is saved. May be a URL or a file system path.

Return value

The method returns *true* if the XML Document could be saved, *false* otherwise.

Objects with this method

document

2.38 :select()

This method moves the ***XML Cursor*** in a given direction or to the first node of the DOM tree that matches a given pattern.

Definition

```
boolean :select
(
  anyvalue DirectionOrPattern input
)
```

Parameters

anyvalue *DirectionOrPattern* input

This parameter may have one of these data types:

enum *DirectionOrPattern* input

When the parameter has the data type *enum*, it specifies in which direction the XML Cursor is moved within the DOM tree.

Value range

select_document
: The XML Cursor is set to the document. The XML Cursor does not necessarily reference a DOM node yet.

select_first
: The XML Cursor is set to the first DOM node, which matches the pattern that has been searched for last.

select_first_child
: The XML Cursor is set to the first direct child node.

select_first_sibling
: The XML Cursor is set to the first DOM node of the same parent.

select_last
: The XML Cursor is set to the last DOM node, which matches the pattern that has been searched for last. This operation searches for the respective DOM node.

select_last_child
: The XML Cursor is set to the last direct child node. This operation searches for the respective DOM node.

select_last_sibling
: The XML Cursor is set to the last DOM node of the same parent. This operation searches for the respective DOM node.

select_next
: The XML Cursor is set to the next DOM node, which matches the pattern that has been searched for last.

select_prev
: The XML Cursor is set to the previous DOM node, which matches the pattern that has been searched for last.

select_next_sibling
The XML Cursor is set to the next DOM node of the same parent.
select_prev_sibling
The XML Cursor is set to the previous DOM node of the same parent.
select_root
The XML Cursor is set to the root node of the DOM tree. This is the initial value of an XML Cursor. The XML Cursor does not necessarily reference a DOM node yet.
select_up
The XML Cursor is set to the parent node.

string *DirectionOrPattern* input

When the parameter has the data type *string*, it specifies a pattern. The XML Cursor is set to the first node of the DOM tree that matches the pattern. The pattern is stored, so that subsequent calls of **:select** with the arguments *select_first*, *select_last*, *select_next* and *select_prev* move the XML Cursor to further DOM nodes that match the pattern.
The pattern syntax is described in chapter “Pattern for the Methods :match() and :select()” of manual “XML Interface”.

Return value

The method returns *true*, when the XML Cursor could be moved, *false* otherwise.

In case of an error the XML Cursor remains pointing to the same DOM node as before the method call.

Please note that an XML Cursor, whose attribute *.mapped* possesses the value *false*, will automatically be positioned to the root of the DOM Document.

Objects with this method

doccursor

2.39 :select_next()

This method is invoked by the ***transformer*** object during a transformation to determine the next node to visit. Hence the method defines the sequence in which the nodes of an XML tree or an IDM object hierarchy are transformed. In the default implementation of **:select_next()**, the sequence is a depth-first traversal.

The method can be redefined (similar to **:init()**).

Definition

```
object :select_next
(
  object Src input
)
```

Parameters

object *Src* input

In this parameter the current, most recently processed node, is transferred. In the default implementation, *Src* depends on the type of the *.root* attribute.

- *.root* contains a string
  In *Src* a ***doccursor*** that points to the current node in the XML tree is expected. The ***doccursor*** is set to next node (according to pre-order sequence) in the XML tree and returned by the method.
- *.root* contains an IDM object
  In *Src* an IDM object representing the current node is expected. Based on this object, the successor is determined.

Return value

The method returns either the next node or *null* if there are no more nodes to visit.

The default implementation of **:select_next** returns the next node depending on the type of the *.root* attribute:

- *.root* contains a string
  The method returns the ***doccursor***, which was passed in the *Src* parameter, now pointing to the next DOM node according to pre-order sequence.
- *.root* contains an IDM object
  The method returns the IDM object to examine next.

Objects with this method

transformer

See also

Attribute root

2.40 :set()

This method can set attributes of an object. It is predefined for all objects, Models and Defaults.

Furthermore, it is possible to overwrite this method and thus to extend the way attributes are set by further actions (e.g. consistency checks).

You can find the attributes available for the relevant object type in the “Object Reference”.

Definition

```
boolean :set
(
      attribute Attribute input,
      anyvalue  Value     input
  { , anyvalue  Index     input }
  { , boolean   SendEvent := true input }
)
```

Parameters

attribute *Attribute* input

This parameter informs the method which attribute shall be set.

anyvalue *Value* input

In this parameter, the new value for the attribute is passed.

*anyvalue **Index** input*

This optional parameter needs to be specified if an indexed attribute shall be set. Its data type must match the index data type of the attribute. Thus, if a one-dimensional attribute should be set, an *integer* value is expected here. However, if the attribute is two-dimensional, an *index* value has to be passed here.

*boolean **SendEvent** := true input*

This optional parameter controls whether a *changed* event should be sent or not when the attribute is successfully set. If no event should be sent, *false* must be passed here. The default value is *true* so an event will be sent if nothing is specified here.

Return value

true

The method returns *true* if setting the attribute has been successful.

false

If setting the attribute was refused (either by the IDM or the IDM programmer), *false* is returned.

Implicit Invocation

The **:set()** method is also invoked implicitly. When an attribute (predefined or user-defined) is changed from the Rule Language (by assignment with `:=` respectively `::=` or the built-in function

setvalue()) or with an interface function (e.g. **DM_SetValue()**), the **:set()** method is called, which then actually sets the attribute. The method **:set()** is invoked “in the middle” and does the actual assignment of the new value.

Redefinition

To redefine the **:set()** method it is sufficient to simply write the following at an object:

```
window W {
  :set(<no parameters!>)
  {
    ...
    Actions to set an attribute.
    ...
  }
}
```

Within the method definition, the parameters *Attribute*, *Value*, *Index* and *SendEvent* can be queried and set, in order to then call e.g. with *this:super()* the method **:set()** of the superclass with the accordingly changed parameters. However, changing the parameters such as *Attribute* and *Index* is not useful because such code would be difficult to understand.

Suppression of Recursive Calls

If within the method **:set()** any attribute of the same object for which the **:set()** method has already been called is set, then the method **:set()** should actually be invoked implicitly again for the same object. Since the danger of an infinite recursion is very high in this case, such a recursive call is suppressed.

Example

```
window {
  :set()
  {
    case (Attribute)
    in .title:
      this.title := "<" + Value + ">";
        // No implicit call of :set().
        // Due to ":=" a "changed" event is sent.
        // With ":::=" the IDM programmer could also suppress this.
      OtherObject.title := "<" + Value + ">";
        // :set() of OtherObject is invoked.
        // The SendEvent parameter of :set() is set to "true".
    otherwise:
      this:super();
    endcase
  }
}
```

In this example, the **:set()** method monitors the setting of the window title and the title is enclosed in "<>" brackets. During the assignment *this.title := …* the method **:set()** is not called again. However, the method **:set()** for another object will still be invoked.

Call of the :set() Method on the Superclass

The above example also shows how to use *this:super()*. All attributes for which the overwritten method **:set()** does not feel responsible should be delegated to the superclass, so that the predefined method **:set()** will set these attributes at some point.

The value of the *Value* parameter is always taken into account and assigned to the attribute at the end. The example above could therefore also be written like this:

Example

```
window {
  :set()
  {
    if Attribute = .title then
      Value := "<" + Value + ">";
      OtherObject.title := "<" + Value + ">";
    endif
    this:super();
  }
}
```

Enforcing Recursive Calls

If the method **:set()** should be called again for the same object with the changed values, the **:recall()** method can be used to enforce that the **:set()** method passed as parameter is also invoked recursively. Caution should be taken, however, as this can very easily lead to infinite recursions.

Example

```
dialog SET

model record MRec {
  integer MaxCount:=100;
  integer Count:=0;
  :set()
  {
    variable boolean RetVal := true;

    case (Attribute)
    in .Count:
      if this.MaxCount < Value then  // Consistency check.
        Value := this.MaxCount;      // Correct Value.
      endif;
      RetVal := this:super();
```

```
    in .MaxCount:
      if this.MaxCount > Value then
        this.MaxCount := Value;
        this:recall(:set, .Count, Value);
          // Set .Count to Value, calling :set() again.
        RetVal := true;
      else
        RetVal := false;             // Decline setting.
      endif
    otherwise:
      RetVal := this:super();
    endcase
    return RetVal;
  }
}

MRec Rec {
}

on dialog start {
  print Rec.Count;      // set value = 0
  Rec.Count := 110;
  print Rec.Count;      // set value = 100

  print Rec.MaxCount;   // set value = 100
  Rec.MaxCount := 200;
  print Rec.MaxCount;   // set value = 100
  Rec.MaxCount := 10;
  print Rec.MaxCount;   // set value = 10
  print Rec.Count;      // set value = 10
  exit();
}
```

See also

Built-in function setvalue() in manual “Rule Language”

C function DM_SetValue in manual “C Interface - Functions”

2.41 :setclip()

Values that are transferred from the source to the target during a **Drag&Drop** operation are determined by the types specified at the ***source*** resource of the source object and directly by the text content of the object.

By **overwriting** the predefined method **:setclip()** on the source object, the values to be transferred can be set as required in the Rule Language.

Invocation of the Method

The method **:setclip()** is invoked implicitly and immediately for the source object during a Drag&Drop operation when the mouse button is pressed and moved for the first time.

The explicit call of **:setclip()** in a rule is not allowed.

Definition

```
void :setclip
(
  object Clipboard input
)
```

Parameters

object *Clipboard* input

In the *Clipboard* parameter, the data values that are moved from the source to the target object are set. The parameter refers to an object of the ***clipboard*** class in whose *.value[enum]* attribute, which is indexed with type enums, the data values can be read and set. However, only those types are permitted that are specified in the ***source*** resource used by the source object. Value assignments to *.value[enum]* with other types as index will cause an error.

Call of the :setclip() Method on the Superclass

In an overwritten **:setclip()** method, calling *this:super()* invokes the **:setclip()** method of the model or, if no further model method exists, accesses the standard behavior.

Example

```
dialog D
source Src
{
  0: .action action_copy, action_cut;
     .type   type_text, type_file, type_object;
}

model pushbutton MPb
{
  .source Src;
```

```
  :setclip()
  {
    this:super();  // Standard method for the object class.
    Clipboard.value[type_file] := "FILE: " +  Clipboard.value[type_file];
  }
}

window Wi
{
  child MPb PbSource
  {
    .text "Source";
    :setclip()
    {
      // Overwriting :setclip() by a custom method.
      // A parameter named clipboard contains the ID of the clipboard object
      // to which the values are assigned in the .value[] attribute.
      this:super();  // Invoke :setclip() method of the model.

      // Set a new value for the text type:
      Clipboard.value[type_text] := "MY TEXT DATA";
    }
  }
  on close  { exit(); }
}
```

2.42 :setformat()

The **:setformat()** method can be used to format text ranges in an RTF edittext.

Definition

```
boolean :setformat
(
  { integer  Start := 1  input,
    integer  End   := -1 input, }
    enum     Type  input,
    anyvalue Value input
)
```

Parameters

*integer **Start** := 1 input*
*integer **End** := -1 input*

These optional parameters define the range that shall be formatted. If these parameters are missing, the range from *.startsel* to *.endsel* (selection or cursor position) is formatted.
Which range will actually be formatted depends on the kind of formatting:

- Character formats affect exactly the specified range.
- Paragraph formats affect all paragraphs that are contained completely or partly in the specified range.

The formatting kind is listed in the table below.
The value range of *Start* and *End* goes from *0* to the number of characters in the displayed text, with every character that can be included in a selection counting. Line break characters therefore count as well.
-1 can be specified for *End* to format the remaining text from the start position.
Start and *End* – similar to *.startsel* and *.endsel* – refer to positions in the formatted text. They **cannot** be used to infer positions in the content string of the RTF edittext, since this also contains formatting instructions. If the *Start* or *End* parameters are greater than the text length, then the text length is used for the RTF edittext.

enum *Type* input

This parameter defines the type of formatting to be applied. The enum values for the formatting types are listed in the table below.

anyvalue *Value* input

This parameter contains a value that determines the characteristics of the formatting type defined in *Type*. The data types for the formatting types are listed in the table below.

Return value

true

The text range has been formatted.

false
Error: The formatting could not be applied (e.g. due to incorrect arguments).

Formatting Kinds, *enum* Values and Data Types of the Formatting Types

Formatting Type	Formatting Kind	*enum* Value	Data Type
typeface, font	character format	*text_font*	*string*, ***font***
font size	character format	*text_size*	*integer*
foreground color, text color	character format	*text_fgc*	*integer*, ***color***
background color	character format	*text_bgc*	*integer*, ***color***
bold	character format	*text_bold*	*boolean*
italic	character format	*text_italic*	*boolean*
underlined	character format	*text_underline*	*boolean*
left indentation	paragraph format	*text_indent_left*	*integer*
right indentation	paragraph format	*text_indent_right*	*integer*
indentation of continuation lines in a paragraph	paragraph format	*text_indent_offset*	*integer*
text alignment	paragraph format	*text_align*	*enum* [*align_left*, *align_right*, *align_center*, *align_justify*]

Depending on the font, it may happen that formatting instructions or formatting attributes set with **:setformat()** are ignored by the Windows object that the IDM uses for the RTF edittext. On Windows 7, this happens, for example, with *text_bold* if no font is set explicitly and therefore the “System” font is used implicitly.

Unit of Measure for Sizes and Positions

Sizes and positions are specified in “twips” – as usual in Windows programming. A twip is equivalent to *1/20 point*, that is *1/1440 inch* or *1/567 cm*.

Explanations on Particular Formattings

- The value for *text_indent_right* is an absolute specification of the indentation from the right margin.
- In contrast, *text_indent_left* is a relative specification that can be used several times. The indents from the left margin are accumulated, but can never extend beyond the margin. The value can be negative and then means that a paragraph is shifted to the left, while positive values mean a shift to the right.
- The value of *text_indent_offset* determines how far all lines of a paragraph except the first are shifted to the right (positive value) or to the left (negative value).
- With *text_font*
  - a string with a font name,
  - a string with size and font name in the form <*size*>.<*font name*>, similar to ***font*** resources on MICROSOFT WINDOWS, or
  - a ***font*** resource

  may be specified. **:setformat()** extracts information such as font name and size from a ***font*** resource and applies the appropriate formatting.
  If the font name is correct, formatting will be performed regardless of whether the font is installed on the system. However, if the font is not installed, you will not see any effect. But you would see it when the RTF text is displayed on a system where the font is installed.
  Specifications in the form <*size*>.<*font name*> are supported as of IDM version A.05.02.I. Size and font name are evaluated and set.
- Foreground color (*text_fgc*) and background color (*text_bgc*) can be passed as *integer* values, which are treated as RGB values, or as ***color*** resources.

Objects with this method

edittext with *.options[opt_rtf] = true*

2.43 :setinherit()

This method can reset attributes of objects to the value of the corresponding Models or Default. The method is predefined on all objects, Models and Defaults.

Furthermore, it is possible to override this method and thus influence the way in which attributes are reset.

Definition

```
boolean :setinherit
(
      attribute Attribute input
  { , anyvalue  Index     input }
  { , boolean   SendEvent := true input }
)
```

Parameters

attribute *Attribute* input

This parameter informs the method which attribute shall be reset.

anyvalue ***Index*** *input*

This optional parameter needs to be set if an indexed attribute shall be reset. Its data type must match the index data type of the attribute. If a one-dimensional attribute shall be reset, an *integer* value is expected here. However, if it is a two-dimensional attribute, an *index* value must be passed here.

boolean ***SendEvent*** *:= true input*

This optional parameter controls whether a *changed* event should be sent or not when the attribute is successfully reset. If no event should be sent, *false* must be passed here. The default value is *true*, so an event will be sent if nothing is specified here.

Return value

The method returns *true* if resetting the attribute was successful, *false* otherwise.

Implicit Invocation

The **:setinherit()** method is also called implicitly. Whenever an attribute (predefined or user-defined) is reset from the rule language or by an interface function (e.g. **DM_ResetValue()**), the method **:setinherit()** is invoked. This must then reset the value of the attribute.

Redefinition

To redefine the method, it is sufficient to simply write the following at an object:

```
window W {
  :setinherit(<no parameters!>)
  {
```

```
    ...
    Actions to reset the value of the attribute.
    ...
    pass this:super();
    or
    return <boolean value>;
  }
}
```

Within the method definition the parameters *Attribute*, *Index* and *SendEvent* can be queried and set, in order to then call e.g. with *this:super()* the **:setinherit()** method of the superclass with the accordingly changed parameters, which, however, is usually not useful.

Suppression of Recursive Calls

If within the method **:setinherit()** any attribute of the same object for which the **:setinherit()** method has already been called is reset, the method **:setinherit()** should actually be invoked implicitly again for the same object. Since the danger of an infinite recursion is very high in this case, such a recursive call is suppressed.

Example

```
window {
  :setinherit()
  {
    case (Attribute)
      in .width:
        setinherit(this, .height);  // no implicit call of :setinherit()
        pass this:super();
      otherwise:
        pass this:super();
    endcase
  }
}
```

In this example, an implicit invocation of the :setinherit() method is suppressed for this.height in the :setinherit() method. However, the value is reset.

Call of the :setinherit() Method on the Superclass

The above example also shows how to use *this:super()*. All attributes for which the overwritten method **:setinherit()** does not feel responsible should be delegated to the superclass, so that the predefined **:setinherit()** method will reset the attribute value at some point.

If the redefined **:setinherit()** method resets an attribute of the object itself (e.g. with *this.Attr :=* …), the inherited bit is not reset as well; i.e. the built-in function **inherited()** will return *false* in this case.

Example

```
dialog SETINHERIT

model record MRec {
  integer X:=0;

  :setinherit()
  {
    if Attribute = .X then
      this.X := 1;
      return true;
    else
      pass this:super();
    endif
  }
}

MRec Rec {
}

on dialog start {
  Rec.X := 2;
  Rec:setinherit(.X);
  print "X: " + Rec.X;                  // should be X = 2
  print "inherited?= " + inherited();  // should be false
  exit();
}
```

See also

Built-in function setinherit() in manual “Rule Language”

2.44 :super()

The problem with methods is that general methods are often defined at the default or at superordinate models. At subordinated objects these methods will then be specialized. At derived objects these methods are overwritten by a new definition. To avoid code doubling the methods of superordinate models can also be used, i.e. they can be called by hierarchically subordinate methods. This must be carried out by calling the method super.

Definition

```
anyvalue :super
(
  {   anyvalue Par1  input }
  { , anyvalue Par2  input }
      ...
  { , anyvalue Par16 input }
)
```

Parameters

All parameters necessary for the method to be called.

Objects with this method

All objects which can have methods.

Example

This example illustrates how certain attributes are reset after closing a window. The methods are directly at the object at which the attributes have been defined. By calling the method :super the order of the methods in the model hierarchy can be defined freely. this:super() always looks for the next method in the model hierarchy of the current object.

```
dialog Super

model window MWin
{
  integer Status := 0;
  rule void CleanAfterClose(integer S)
  {
    !! Reset attribute.
    this.Status := S;
  }
  !! Both rules react when window is getting invisible
  !! and will trigger centrally the"clear rule".
  on close before
  {
    this:CleanAfterClose();
  }
```

```
  on .visible changed before
  {
    if this.visible = false then
      this:CleanAfterClose(0);
    endif
  }
}
model MWin MMainWin
{
  integer MainStatus := 0;
  rule void CleanAfterClose()
  {
    !! Reset only those attributes which have been defined here.
    this.MainStatus := 0;
    !! Clear remaining attributes.
    this:super(0);
  }
}
MMainWin MainWin
{
}
```

2.45 :transform()

This method transforms the object using a given schema. When the transformation target is an ***XML Document***, the stored DOM tree is deleted and a new tree is built up. All existing XML Cursors become invalid. The attribute *.mapped* has the value *false* for invalid XML Cursors.

Alternatively, the transformation target may be a text or a file. If the transformation produces no legal XML format, its immediate result must be a text or a file, because the result cannot be assigned to an XML Document. This applies for transformations to HTML, for example.

Definition

```
boolean :transform
(
      object   Schema input
  { , anyvalue Target := null        input output }
  { , enum     Type   := type_object input }
)
```

Parameters

object *Schema* input

The XSLT document that defines the transformation rules. The parameter must be an ***XML Document***.

*anyvalue **Target** := null input output*

Optional parameter that specifies the target of the transformation. When the parameter is missing, the *this* object is taken as target. This parameter may have one of these data types:

*object **Target** input*

Target is an input parameter representing an ***XML Document***. The stored DOM tree is deleted and a new tree is built up in accordance with the transformation.

*string **Target** input output*

Depending on the *Type* parameter, *Target* is either an input parameter that specifies a file name or an output parameter that contains the string resulting from the transformation.

*enum **Type** := type_object input*

Optional parameter that defines the transformation type.

type_file

The *Target* parameter is a file.

type_object

The *Target* parameter is an ***XML Document***. This is the default value when the *Target* parameter has the data type *object*.

type_text

The *Target* parameter is an output parameter with data type *string*. This is the default value when the *Target* parameter has the data type *string*.

Return value

The method returns *true* after a successful transformation, *false* if the transformation failed.

In case of an error, the DOM tree of the XML Document passed in *Target* either remains unchanged or is deleted completely; there will never be a partial transformation.

Objects with this method

- doccursor
- document

2.46 :unuse()

This method removes or unloads a module that has been included with “use” from a ***dialog*** or ***module*** object. If not used elsewhere, it is unloaded completely (including the removal of all instances).

Definition

```
boolean :unuse
(
  anyvalue UsepathOrModul input
)
```

Parameters

anyvalue *UsepathOrModule* input

This parameter is used to specify the module as a Use Path (in the form of an identifier path) or directly as a module ID.

Return value

false

Module is imported not at all or not by “use”.

true

Access to the module per “use” has been removed.

Objects with this method

» dialog

» module

Example

```
dialog D
use Customer.Models;

window Wi
{
  MGbCustomer {}
  pushbutton PbUnuse
  {
    .yauto -1;
    .text "Unuse";

    on select
    {
      if this.module:unuse("Customer.Models") then
        this.sensitive := false;
      endif
```

```
    }
  }
}
```

Availability

Since IDM version A.06.02.g

See also

Method :use()

Chapters “The Alternative Import Mechanism” and “Language Specification and Use Path” in manual “Programming Techniques”

2.47 :use()

This method imports a new module into a ***dialog*** or ***module*** object from its Use Path. This allows to include the module in a similar way as with a static **use** statement, so that the module is loaded only once in the dialog.

If the *Check* parameter is set to *true* during the call, it is only tested whether an import with “use” already exists on this object.

Definition

```
object :use
(
      string  Usepath input
  { , boolean Export  := false input
  { , boolean Check   := false input } }
)
```

Parameters

string *Usepath* input

This parameter is used to specify the module as a Use Path (in the form of an identifier path).

*boolean **Export** := false input*

If this optional parameter is *true*, the “use” is additionally flagged as exported upon creation (default value *false*).
The parameter is only relevant when *Check* = *false*.

*boolean **Check** := false input*

If this optional parameter is *true*, it is only tested whether the module given by the Use Path has already been imported with “use” (default value *false*).

Return value

null

With *Check* = *false*:
Module could not be found or loaded.
With *Check* = *true*:
No “use” exists for this Use Path.

<Id>

Id of the imported module.

Objects with this method

» dialog

» module

Example

```
window Wi
{
  edittext EtUsepath
  {
    .xauto 0;
    .content "Plugins.View.CustomerModels";
  }

  pushbutton PbUse
  {
    .yauto -1;
    .text "Use";

    on select
    {
      if this.module:use(EtUsepath.content, false, true) = null then
        if this.module:use(EtUsepath.content) = null then
          print "Can't import module!";
        endif
      endif
    }
  }
}
```

Availability

Since IDM version A.06.02.g

See also

Method :unuse()

Chapters “The Alternative Import Mechanism” and “Language Specification and Use Path” in manual “Programming Techniques”

2.48 :validate()

This method checks if the XML Document conforms to the document type indicated in its DOM tree. When no document type is contained, an error is returned.

Definition

```
integer :validate
(
  { string ErrorMsg := null output }
)
```

Parameters

*string **ErrorMsg** := null output*

When this optional parameter is given, it is initialized with "". If an error occurs, the parameter is assigned with the error message from the system. However, it is more reliable to query the return value of the method.

Return value

0

Successful validation; the XML Document meets the document type.

DOM error code

The XML Document does **not** conform to the document type.

-1

Internal error. *ErrorMsg* is **not** set.

Objects with this method

document

Index

N

O

P

R

S

T

U

V

X