ISA Dialog Manager

METHODENREFERENZ

A.06.03.b

In diesem Handbuch sind alle vordefinierten Methoden der ISA Dialog Manager Objekte erläutert. Es enthält die Methodendefinitionen mit ihren Parametern und Rückgabewerten. Außerdem ist beschrieben, welche Methoden überschrieben (redefiniert) werden können.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Deutschland

Darstellungskonventionen

DM wird in diesem Handbuch synonym zu "Dialog Manager" verwendet.

Die Bezeichnung UNIX schließt generell alle unterstützten UNIX-Derivate ein - außer in den explizit angegebenen Fällen.

Dort wo für geläufige englische Fachbegriffe keine gängigen deutschen Übersetzungen existieren, wird zur Vermeidung von Unklarheiten der englische Begriff verwendet.

< >	muss durch einen entsprechenden Wert ersetzt werden
color	Schlüsselwort ("keyword")
.bgc	Attribut
{ }	optional (0 oder einmal)
[ ]	optional (0 oder n-mal)
<A> | <B>	entweder <A> oder <B>

Beschreibungsmodus

Alle Schlüsselwörter sind fett und unterstrichen, z.B.

variable **integer** **function**

Indizierung von Attributen

Syntax für indizierte Attribute:

[I]

[I,J] bzw. [row,column]

Identifikatoren

Identifikatoren müssen mit einem Großbuchstaben oder einem "Unterstrich" ('_') beginnen. Die weiteren Zeichen können Groß-, Kleinbuchstaben, Zahlen oder Unterstriche sein.

Der Bindestrich ('-') ist für die Benennung von Identifikatoren als Zeichen ***nicht*** zugelassen!

Die maximale Länge eines Identifikators beträgt *31* Zeichen.

Beschreibung der zugelassenen Identifikatoren in Backus-Naur-Form

<Identifikator> ::= <erstes Zeichen>{<Zeichen>}

<erstes Zeichen> ::= _ | <Großbuchstabe>

<Zeichen> ::= _ | <Kleinbuchstabe> | <Großbuchstabe> | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | … 9 | 0

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

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

Inhalt

1 Einleitung

Methoden dienen zum Aufruf objektspezifischer Funktionalität. Mit ihrer Hilfe kann ein Objekt beauftragt werden, eine bestimmte Operation durchzuführen, welche durch die jeweilige Methode festgelegt ist. Dabei können der Methode Parameter mitgegeben werden und die Methode kann einen Rückgabewert liefern.

Wie bei den Attributen ist im System eine Reihe von Methoden bereits definiert. Daneben können in sogenannten benutzerdefinierten Methoden auch eigene Methode zusätzlich definiert werden.

Im Folgenden werden die bereits im System definierten Methoden beschrieben.

2 Methoden in alphabetischer Reihenfolge

2.1 :action()

Die Methode **:action()** gibt es in zwei verschiedenen Ausprägungen:

» **Am mapping-Objekt definiert sie, wie Daten an den einzelnen Knoten transformiert werden**

» **Am transformer-Objekt wird sie von dessen :apply()-Methode für jeden besuchten Knoten aufgerufen**

2.1.1 :action() (mapping)

Diese Methode definiert, wie Daten während einer Transformation an einzelnen Knoten transformiert werden sollen, das heißt aus dem *Src*-Parameter in den *Dest*-Parameter übertragen werden sollen.

Jedes ***mapping***-Objekt besitzt eine **:action()**-Methode, die als Default-Implementierung einfach nur *true* zurückliefert. Diese Methode kann vom IDM-Programmierer überdefiniert werden (ähnlich wie **:init()**). Somit kann hier festgelegt werden, in welcher Weise die Daten vom *Src*-Parameter zum *Dest*-Parameter übertragen werden.

Die Methode **:action()** wird während einer Transformation vom Vater des ***mapping***-Objektes (***transformer***) aufgerufen (genauer von der **:action()**-Methode dieses ***transformers***), falls eine Entsprechung zwischen einem Knoten in einem XML-Baum bzw. einer IDM-Objekthierarchie und dem dazugehörigen ***mapping***-Objekt gefunden wird (siehe hierzu Dokumentation des transformer-Objekts).

Definition

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

Parameter

anyvalue *Src* input

Hier wird der aktuelle Knoten übergeben, an dem die Transformation gerade steht. Dabei ist zwischen zwei Fällen zu unterscheiden.

» Ein XML-Baum wird transformiert:
In *Src* wird ein ***Doccursor***-Objekt übergeben, das auf den aktuellen Knoten im XML-Baum verweist.

» IDM-Objekthierarchie wird transformiert:
In *Src* wird ein IDM-Objekt übergeben, das den aktuellen Knoten repräsentiert.

anyvalue *Dest* input output

Hier wird stets der Wert durchgeschleust, der beim Aufruf von **:apply()** als *Dest* übergeben oder in einem der früheren Aufrufe von **:action()**-Methoden verändert wurde.

Rückgabewert

Wenn die Methode *true* zurückliefert, so wird angenommen, das der aktuelle Knoten komplett abgearbeitet worden ist und kein weiteres ***mapping***-Objekt auf die Entsprechung mit dem Knoten geprüft wird. Es werden also keine weiteren **:action()**-Methoden aufgerufen auch wenn weitere passende ***mapping***-Objekte existieren.

Wird *false* zurückgeliefert, so wird der aktuelle Knoten mit den weiteren ***mapping***-Objekten verglichen und gegebenenfalls deren **:action()**-Methoden aufgerufen.

Objekte mit dieser Methode

mapping

2.1.2 :action() (transformer)

Diese Methode wird von der **:apply()**-Methode des ***transformers*** iterativ für jeden besuchten Knoten eines XML-Baumes bzw. einer IDM-Objekthierarchie aufgerufen. Die Methode muss dann alle ***mapping***-Kinder des Transformers durchgehen und testen, ob Muster in deren *.name*-Attributen auf den aktuellen Knoten (*Src*-Parameter) passen. Die Reihenfolge in der die ***mappings*** getestet werden, entspricht der Definitionsreihenfolge im *.mapping[]*-Vektor, wobei die geerbten Mappings als letzte drankommen.

Achtung

Die geerbten ***mappings*** sind nicht im *.mapping[]*-Vektor des Vaters enthalten. Auf sie kann deswegen nur über die Vererbungshierarchie zugegriffen werden. Das ist anders als bei normaler IDM-Vererbung.

Wird eine Übereinstimmung gefunden, so wird die **:action()**-Methode des dazugehörigen ***mapping***-Objekts aufgerufen, wobei *Src*- und *Dest*-Parameter einfach übergeben werden. Der Rückgabewert dieser **:action()**-Methode bestimmt, ob verbleibende ***mapping***-Objekte noch untersucht werden und gegebenenfalls deren **:action()**-Methoden auch für diesen Knoten aufgerufen werden (Rückgabewert ist *false*), oder ob der Knoten als abgearbeitet angesehen wird und die **:action()**-Methode des ***transformers*** beendet wird (Rückgabewert ist *true*).

Diese Methode kann überdefiniert werden (ähnlich zu **:init()**).

Definition

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

Parameter

anyvalue *Src* input

Hier wird der zu untersuchende Knoten übergeben.

- Enthält das root Attribut des ***transformers*** einen String, so wird hier ein ***doccursor***-Objekt erwartet, das auf den XML-Knoten verweist. Dieser Knoten wird dann mit den ***mappings*** verglichen.
- Enthält das root Attribut des ***transformers*** ein IDM-Objekt, so wird hier das IDM-Objekt erwartet, das den aktuellen Knoten repräsentiert. Dieses Objekt wird dann mit den ***mappings*** verglichen.

anyvalue *Dest* input output

Hier wird stets der Wert durchgeschleust, der beim Aufruf von **:apply()** als *Dest* übergeben oder in einem der früheren Aufrufe von **:action()**-Methoden verändert wurde.

Rückgabewert

Die Methode gibt *true* zurück, falls die Verarbeitung ohne Fehler durchlief, sonst *false*.

Objekte mit dieser Methode

transformer

2.2 :add()

Fügt einen Kindknoten als letzten Knoten an den aktuellen DOM-Knoten an. Der XML-Cursor wird auf das neue Element gesetzt.

Definition

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

Zweite Form (nur MICROSOFT WINDOWS)

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

Parameter

enum *Nodetype* input

Gibt die Art des DOM-Knotens an, der erzeugt wird.

Wertebereich

nodetype_attribute
: Der Knoten ist ein Attribut eines Elements.

nodetype_cdata_section
: Der Knoten ist ein Abschnitt mit ungeparsten Zeichendaten (CDATA).

nodetype_comment
: Der Knoten ist ein Kommentar.

nodetype_document
: Der Knoten ist ein vollständiges Dokument.

nodetype_document_fragment
: Der Knoten ist ein Ausschnitt aus einem Dokument.

nodetype_document_type
: Der Knoten ist eine Dokumenttyp-Deklaration.

nodetype_element
: Der Knoten ist ein Element.

nodetype_entity
: Der Knoten ist eine Deklaration einer Entität.

nodetype_entity_reference
Der Knoten ist eine Referenz auf eine deklarierte Entität.
nodetype_notation
Der Knoten ist eine in der DTD deklarierte Notation.
nodetype_processing_instruction
Der Knoten ist eine Verarbeitungsanweisung.
nodetype_text
Der Knoten ist der Textinhalt eines Elements oder ein Attributwert.

string ***Name*** *:= null input*
Optionaler Parameter, gibt den Namen bzw. Tag des neuen DOM-Knotens an. Wenn der Parameter *Nodetype* einen der folgenden Werte besitzt, wird der Parameter *Name* ignoriert und ein fest vorgegebener Name verwendet.

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

string ***Value*** *:= null input*
Optionaler Parameter, gibt den Wert des neuen DOM-Knotens an. Dies ist nur zulässig, wenn der Parameter *Nodetype* einen der folgenden Werte besitzt:

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

pointer ***IXMLDOMNode*** **input**
Gibt das MICROSOFT WINDOWS COM-Objekt an, das als Kindknoten eingefügt werden soll. Es kann auch ein MICROSOFT WINDOWS **IXMLDOMNodeList** COM-Objekt angegeben werden, wobei dann alle XML-Knoten, die in dieser Liste enthalten sind, eingefügt werden.

boolean ***Select*** *:= true input*
Optionaler Parameter, gibt an, ob der XML-Cursor auf den neu angelegten DOM-Knoten zeigen soll. Wenn der Parameter nicht angegeben ist, wird *true* genommen, also der XML-Cursor auf den neuen DOM-Knoten gesetzt.

Rückgabewert

Gibt an, ob das Anlegen erfolgreich war.

Im Fehlerfall bleibt der XML-Cursor auf dem aktuellen DOM-Knoten.

Objekte mit dieser Methode

doccursor

2.3 :apply()

Die Methode **:apply()** gibt es in zwei verschiedenen Ausprägungen:

» **Beim Datenmodell liest sie die Werte aller View-Attribute aus und weist sie den Model-Komponenten zu**

» **Am transformer-Objekt stößt sie die Transformation von Daten an**

2.3.1 :apply() (Datenmodell)

Beim Aufruf dieser Methode an einer View-Komponente werden alle Datenwerte von View-Attributen, die mit Model-Komponenten gekoppelt sind, ausgelesen (durch einen **:retrieve()**-Aufruf) und den korrespondierenden Model-Komponenten zugewiesen.

Durch Angabe der optionalen Parameter kann auch die Einschränkung auf ein spezielles View-Attribut bzw. auf einen speziellen indizierten Einzelwert erfolgen. Ohne Parameter erfolgt auch ein **:apply()**-Aufruf an allen Kindern und Kindeskindern.

Normalerweise sollte ein Aufruf dieser Methode nicht notwendig werden, da die Synchronisation zwischen View und Model über das Attribut *.dataoptions[]* gesteuert wird.

Die Kopplung zur Model-Komponente muss über die Attribute *.datamodel* und *.dataset* erfolgt sein.

Es erfolgt keine Fehlermeldung, wenn das Objekt keine gekoppelten Attribute besitzt oder das optional angegebene Attribut nicht existiert oder nicht gekoppelt ist.

Definition

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

Parameter

*attribute **Attribute** input*

Dieser optionale Parameter bezeichnet das View-Attribut welches ausgelesen und der zugehörigen Model-Komponente zugewiesen werden soll.

*anyvalue **Index** input*

Dieser optionale Parameter bezeichnet den Indexwert der für die Werteholung vom View-Objekt zu verwenden ist.

Beispiel

Im folgenden Beispieldialog erfolgt die Zuweisung des Inhaltsstrings ans Datenmodell über die Nutzung der **:apply()**-Methode am ***Edittext*** oder am ***Fenster***. Der ***Statictext*** am unteren Fensterrand zeigt den aktuellen Wert des Datenmodells „VarString“ an.

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

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

  child edittext Et
  {
    .xauto 0;
    .xright 80;
    .datamodel VarString;
    .dataset .value;
    .content "Samstag";
    .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)

Mit dieser Methode wird an einem ***transformer***-Objekt die Transformation von Daten angestoßen. Für Näheres siehe Kapitel „Das transformer-Objekt“ des Handbuchs „XML-Schnittstelle“.

Diese Methode kann überdefiniert werden (ähnlich zu **:init()**).

Definition

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

Parameter

anyvalue *Src* input

Hier wird der Wurzelknoten übergeben, von dem aus die Transformation gestartet werden soll. Dabei ist in der Default-Implementierung zwischen zwei Fällen zu unterscheiden.

» Ist *Src* ein Document-Objekt, so wird als Startknoten die Wurzel des XML-Baumes gewählt, die das ***document*** repräsentiert. Das ***document*** muss geladen sein.
Ist *Src* ein ***doccursor***-Objekt, so wird als Wurzel der Knoten im XML-Baum genommen, auf das das ***doccursor***-Objekt verweist.
Die Transformationsrichtung ist XML nach IDM.

» Ist *Src* ein IDM-Objekt (außer Document oder Doccursor), so wird als Wurzel das IDM-Objekt selbst gewählt.
Die Transformationsrichtung ist nach IDM beliebig.

*anyvalue **Dest** input output*

In diesem optionalen Parameter kann der IDM-Programmierer einen beliebigen Wert übergeben, der lediglich weiter an die **:action()**-Methoden von ***mapping***-Objekten durchgeschleust wird. D.h. innerhalb der Methode wird der Aufruf *:action(Next_Node, Dest)* iterativ für jeden besuchten Knoten ausgeführt.

Rückgabewert

Es wird *true* zurückgeliefert, falls die Transformation erfolgreich war, sonst ist der Wert *false*.

Objekte mit dieser Methode

transformer

2.4 :call()

Mit Hilfe der Methode **:call()** lassen sich beliebige eingebaute als auch benutzerdefinierte Methoden aufrufen. Dieses ist z.B. dann notwendig, wenn die Methode, die aufgerufen werden soll, berechnet wird oder in einer Variablen gespeichert ist. Der Aufruf von **:call()** ruft indirekt die angegebene Methode auf. Der Rückgabewert entspricht dem der indirekt aufgerufenen Methode. Beim Aufruf über **:call()** können jedoch nur maximal *15* weitere Parameter an die indirekt aufgerufene Methode übergeben werden. Hierauf ist beim Design von Methoden zu achten. Die Methode **:call()** kann auch eingebaute Methoden wie etwa **:insert()** oder **:delete()** aufrufen.

Definition

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

Parameter

method *Method* input

In diesem Parameter wird die aufzurufende Methode angegeben.

*anyvalue **Par1** input*

…

*anyvalue **Par15** input*

In diesen optionalen Parametern werden die Parameter der aufzurufenden Methode angegeben. Die Anzahl der angegebenen Parameter muss dabei der Anzahl der Parameter der aufzurufenden Methode entsprechen.

Rückgabewert

Rückgabewert der aufgerufenen Methode.

Objekte mit dieser Methode

Alle Objekte, die benutzerdefinierte Methoden haben können bzw. eingebaute Methoden haben.

Beispiel

```
dialog Example

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

```
}

on dialog start
{
  Rec:call(:SomeMethod);     // ruft die Methode SomeMethod auf
  Rec:call(:Display, this);  // ruft die Methode Display
                             // mit dem Dialog als Objekt auf
}
```

Siehe auch

Methode **:recall()**

2.5 :calldata()

Beim Aufruf dieser Methode werden alle Datenmodelle, die am aufgerufenen Objekt und seinen Kindern über das *.datamodel*-Attribut definiert sind, aufgesammelt und eine „Aktion“, für jedes unterschiedliche Datenmodell jeweils nur einmal, in Form einer Methode mit Argumenten aufgerufen.

Mit dieser Methode kann also ohne die definierten Model-Objekte im Detail zu kennen an einem View-Objekt eine „Aktion“ an den beteiligten Datenmodellen ausgelöst werden.

Ein Rückgabewert oder eine Rückkopplung, falls z.B. die Methode an den Datenmodellen nicht vorhanden ist, ist nicht vorgesehen.

Definition

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

Parameter

method *Method* input

Dieser Parameter bezeichnet die Methode, die an den beteiligten Datenmodellen aufzurufen ist.

*anyvalue **Arg1** input*

…

*anyvalue **Arg15** input*

Diese optionalen Parameter beinhalten die Argumente, die für den Methodenaufruf benutzt werden.

Beispiel

```
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()

Mit Hilfe der Methode **:childcount()** kann beim ***Treeview*** die Anzahl der zu einem Element des Baumes gehörenden Unterelemente erfragt werden. Dabei kann über die Parameter gesteuert werden, wie viele Einrückungsstufen bei der Berechnung beachtet werden sollen.

Definition

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

Parameter

integer *Position* input

In diesem Parameter wird der Index des Eintrags angegeben, zu dem im Baum die Anzahl der dazugehörenden Elemente berechnet werden soll.

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

In diesem optionalen Parameter kann man die maximale Einrücktiefe angegeben, bis zu der die Elemente beachtet werden sollen. Wird dieser Parameter nicht angegeben, so werden alle Elemente beachtet.

Rückgabewert

Anzahl der gefundenen Elemente.

Objekte mit dieser Methode

Treeview

Beispiel

```
window Wn
{
  .title "Beispiel fuer die Methode :childcount()";
  child treeview Tv
  {
    .content[1] "Firma A";
    .content[2] "Mueller";
    .content[3] "Meier";
    .content[4] "Firma B";
    .content[5] "Schulz";
    .content[6] "Schmidt";
    .content[7] "Fischer";
    .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  "Kein Wert";
  }

  child pushbutton Pb
  {
    .xleft  226;
    .width  247;
    .ytop   16;
    .height 28;
    .text   "Zaehle die Mitarbeiter einer Firma";
    on select
    {
      St2.text := itoa(Tv:childcount(Tv.activeitem, 1));
    }
}
```

2.7 :childindex()

Mit Hilfe dieser Methode kann man den absoluten Index eines Kindes in einem ***Treeview*** berechnen lassen. Dazu übergibt man dieser Methode den relativen Index des Kindes bezüglich seines Vaters.

Definition

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

Parameter

integer *Parent* input
In diesem Parameter wird der Index des Vaters angegeben.

integer *Child* input
In diesem Parameter wird der Index des Kindes bezüglich dem angegebenen Vater angegeben.

Rückgabewert

0
Bei einem Fehler.

> 0
Absoluter Index des Kindes.

Objekte mit dieser Methode

Treeview

Beispiel

```
window Wn
{
  .title "Beispiel fuer die Methode :childindex()";
  child treeview Tv
  {
    .content[1] "Firma A";
    .content[2] "Mueller";
    .content[3] "Meier";
    .content[4] "Firma B";
    .content[5] "Schulz";
    .content[6] "Schmidt";
    .content[7] "Fischer";
    .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 "Bestimme absolute Nummer";
    on select
    {
      variable integer I := Tv:childindex(Tv.activeitem, atoi(EtIn.content));

      if fail(atoi(EtIn.content)) then
        St.text := "Haben sie einen Mitarbeiter eingegeben?";
      endif
      if (I = 0) then
        St.text := "Haben sie eine Firma selektiert?";
      else
        St.text := itoa(I);
      endif
    }
  }

  child edittext EtIn
  {
    .content   "Wievielter Mitarbeiter der selektierten Firma?";
    .multiline true;
  }

  child statictext St
  {
    .text "Ergebnis der Anfrage";
  }
}
```

2.8 :clean()

Diese Methode kann an Objekten eines Dialogs oder Moduls vor deren endgültiger Zerstörung noch Aktionen zum Säubern der Objekte (freigeben von zuvor allokierten Ressourcen) vornehmen. Die Methode ist an allen Objekten, Modellen und Defaults vordefiniert, kann jedoch überschrieben werden, um benutzerdefinierte Aktionen an den Objekten vorzunehmen.

Die **:clean()**-Methode kann nicht explizit aufgerufen werden. Die Aufrufe erfolgen ausschließlich implizit:

1. Wenn ein Dialog oder Modul entladen werden soll, wird die **:clean()**-Methode für alle Objekte, Modelle und Defaults des Dialogs bzw. Moduls nach der *finish*-Regel aufgerufen.
2. Wenn ein Objekt oder Modell zur Laufzeit zerstört wird (Aufruf der **:destroy()**-Methode, eingebauten Funktion **destroy()** oder Schnittstellenfunktion **DM_Destroy()** bzw. **DMcob_Destroy ()**), wird die **:clean()**-Methode unmittelbar vor der Zerstörung für dieses Objekt aufgerufen.

Definition

```
void :clean
(
)
```

Parameter

Keine.

Redefinition

Zur Redefinition der Methode genügt es, einfach an einem Objekt folgendes zu schreiben:

```
window W {
  :clean()
  {
    ...
    // Code um etwas freizugeben oder Daten in konsistenten Zustand zu
bringen.
    ...
  }
}
```

Möchte man einen Teil der Aufgaben an die Superklasse delegieren, so kann mit *this:super()* die **:clean()**-Methode des in der Vererbungshierarchie weiter oben liegenden Modells oder Defaults aufgerufen werden.

Hinweis

Der Aufruf von *this:super()* hat bei der **:clean()**-Methode eine weitere wichtige Funktion. Erfolgt der Aufruf dieser Methode auf der obersten Ebene (typischerweise dem Default), auch wenn es dort formal keine Superklasse mehr gibt, so werden die **:clean()**-Methoden an den Kindern des Objektes

ebenfalls ausgeführt. Demzufolge können die Aufrufe der **:clean()**-Methoden für die Kinder unterdrückt werden, wenn irgendwo in der Vererbungshierarchie kein Aufruf von *this:super()* erfolgt.

Beispiel

```
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 >>>";
    // Kein Aufruf von this:super().
    print "== MRec2 <<<";
  }
}

model MRec1 MRec3
{
```

```
  // Hier tritt die vordefinierte :clean()-Methode in Aktion.
  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();
  // Kein :clean() der Kinder R1 und R2.

  O := MRec3:create(CLEAN);
  print "on start: Destroy";
  O:destroy();
  // :clean()-Methoden der Kinder R1, R2 und R3 werden aufgerufen.

  exit();
}
```

Der abgeänderter Ausschnitt aus der Trace-Datei soll verdeutlichen, was bei der Ausführung dieses Codes passiert.

```
"on start: Destroy"
    "== MRec2 >>>"
    "== MRec2 <<<"
// Kein :clean() der Kinder R1 und R2.

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

```
          "== DEFAULT <<<"
      "== R3 <<<"
        "== DEFAULT <<<"
    "== MRec1 <<<"
// :clean()-Methoden der Kinder R1, R2 und R3 werden aufgerufen.
```

2.9 :clear()

Die Methode **:clear()** gibt es in zwei verschiedenen Ausprägungen:

» **Löschen des Inhalts von Einträgen in Listenobjekten**

» **Löschen des Inhalts von Elementen in Feldern (indizierten benutzerdefinierten Attributen)**

2.9.1 :clear() (Listenobjekte)

Mit Hilfe dieser Methode können die Werte in Zeilen und Spalten des Inhalts eines Objektes gelöscht werden, dessen Inhalt aus mehreren, mit *integer* oder *index* indizierten, Einträgen besteht. Zusammen mit dem Inhalt werden auch die Werte der dazugehörenden Attribute gelöscht.

Im Unterschied zur **:delete()**-Methode werden mit **:clear()** nur die Werte in den Zeilen bzw. Spalten gelöscht. Die Einträge selbst bleiben als „leere" Einträge erhalten, die – sofern vorhanden – den Standardwert aufweisen. **:clear()** verringert also die Anzahl der Zeilen bzw. Spalten nicht.

Definition

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

Parameter

integer *Start* input

Dieser Parameter definiert die Position, ab der die Inhalte der Zeilen bzw. Spalten gelöscht werden sollen.

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

Mit diesem optionalen Parameter wird die Anzahl der Zeilen bzw. Spalten festgelegt, deren Inhalte gelöscht werden sollen. Wird der Parameter nicht angegeben, so wird *1* angenommen.

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

Dieser optionale Parameter steuert beim ***tablefield***, ob die Inhalte von Zeilen oder von Spalten gelöscht werden. Dies ist abhängig vom Wert des ***tablefield***-Attributs *.direction*.
Der Parameter darf nur beim ***tablefield*** angegeben werden. Bei anderen Objekten als dem ***tablefield*** führt die Angabe des Parameters zu einem Fehler.

Wertebereich

false

Löschen entgegen der vom Attribut *.direction* angegebenen Richtung

true

Löschen in der vom Attribut *.direction* angegebenen Richtung

Damit werden beim ***tablefield*** folgende Inhalte gelöscht:

Parameter *Direction*	*.direction 1* (vertikal)	*.direction 2* (horizontal)
false	Zeile(n)	Spalte(n)
true	Spalte(n)	Zeile(n)

Rückgabewert

Die Methode gibt *true* zurück, wenn die Inhalte der Zeilen bzw. Spalten gelöscht werden konnten.

Wenn beim Löschen der Inhalte ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

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

Beispiel

```
window Wn
{
  .title "Beispiel fuer die Methode :clear()";

  child listbox Lb
  {
    .content[1] "Klettenheimer";
    .content[2] "Strassenmeier";
    .content[3] "Redweik";
    .content[4] "Tilly";
    .content[5] "Mueller";
    .content[6] "Meyer";
    .content[7] "Schulz";
  }

  child pushbutton Pb_clear
  {
    .text "Loesche Eintrag";

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

```
  }
}
```

2.9.2 :clear() (Felder)

Mit Hilfe dieser Methode können die Inhalte von Elementen in Feldern (indizierte, nicht assoziative, benutzerdefinierte Attribute) gelöscht werden.

Im Unterschied zur **:delete()**-Methode werden mit **:clear()** nur die Inhalte der Elemente gelöscht. Die Elemente selbst bleiben als „leere“ Elemente erhalten, die – sofern vorhanden – den Standardwert aufweisen. **:clear()** verringert also die Anzahl der Elemente nicht.

Besonderheit

Die **:clear()**-Methode kann auch dafür benutzt werden, **alle** Elemente eines **assoziativen** Felds zu löschen. In diesem Fall dürfen weder der *Start*- noch der *Count*-Parameter angegeben werden. Es ist also nicht möglich, mit **:clear()** einzelne Elemente aus einem assoziativen Feld zu löschen. Außerdem werden in diesem Fall nicht nur die Inhalte sondern die Elemente selbst gelöscht. Die Anzahl der Elemente ist danach also *0*.

Definition

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

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

*integer **Start** input*

Dieser Parameter definiert die Position, ab der die Inhalte der Elemente gelöscht werden sollen. Der Wertebereich für *Start* ist *0 … .count[Attr]*, d.h. es kann auch der Standardwert gelöscht werden.
Wenn weder *Start* noch *Count* angegeben sind, werden die Inhalte aller Elemente gelöscht.

*integer **Count** input*

Dieser optionale Parameter definiert die Anzahl der Elemente, deren Inhalte gelöscht werden sollen.
Wenn weder *Start* noch *Count* angegeben sind, werden die Inhalte aller Elemente gelöscht.
Wenn *Start* angegeben wird, ist der Standardwert von *Count 1*.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Inhalte der Elemente gelöscht werden konnten.

Wenn beim Löschen der Inhalte ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

Alle Objekte, die benutzerdefinierte Attribute haben können.

Siehe auch

Kapitel „Methoden für Arrays benutzerdefinierter Attribute“ im Handbuch „Benutzerdefinierte Attribute und Methoden“

2.10 :collect()

Beim Aufruf dieser Methode an einer Model-Komponente werden alle Datenwerte von den View-Komponenten erfragt, die gekoppelte Attribute aufweisen. Dazu erfolgt ein **:retrieve()**-Aufruf für die korrespondierenden View-Attribute um die Datenwerte zu ermitteln und sie dem Datenmodell zuzuweisen.

Normalerweise sollte ein Aufruf dieser Methode nicht notwendig werden, da die Synchronisation zwischen View und Model über das Attribut *.dataoptions[]* gesteuert wird.

Die Kopplung zur Model-Komponente muss über die Attribute *.datamodel* und *.dataset* erfolgt sein.

Es erfolgt keine Fehlermeldung wenn das Objekt keine Referenzierung durch eine View-Komponente oder keine gekoppelten Attribute besitzt.

Definition

```
void :collect
(
)
```

Beispiel

```
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()

Mit Hilfe dieser Methode können zur Laufzeit neue Objekte generiert werden. Die Methode liefert als Ergebnis das neu generierte Objekt zurück.

Definition

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

Parameter

object *Parent* input

Dieser Parameter bezeichnet den Vater des Objekts.

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

Dieser optionale Parameter bezeichnet den Dialog, zu dem das Objekt gehören soll.

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

Mit diesem optionalen Parameter können Sie angeben, ob Sie einen Default, ein Modell oder eine Instanz erzeugen wollen.

Wertebereich

1
: erzeugt einen Default

2
: erzeugt ein Modell

3
: erzeugt eine Instanz

Wenn dieser Parameter nicht angegeben ist, wird automatisch eine Instanz generiert.

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

In diesem optionalen Parameter wird angegeben, ob das Objekt unsichtbar oder wie in der Vorlage definiert generiert werden soll.

Wertebereich

true
: Das Objekt wird immer unsichtbar angelegt.

false
: Die Sichtbarkeit wird vom Modell oder Default übernommen.

Rückgabewert

objectId
: Id des neu generierten Objekts.

null
: Null-ID, da das Objekt nicht generiert werden konnte.

Beispiel

```
dialog Beispiel

model window MWnDetail
{
  !! Detailanzeige von Datensaetzen
  .title "Detailanzeige der Daten";
  rule void GetData()
  {
    !! Holen der Daten
    !! Fenster sichtbar machen
    this.visible ::= true;
  }
  child pushbutton PbExit
  {
    .text "&Abbrechen";
    on select
    {
      !! zerstoert das Instanzen-Fenster
      this.window:destroy();
    }
  }
}

window WnMain
{
  .title "Daten\374bersicht";
  child tablefield TfData
  {
    !! enthaelt eine Datenuebersicht
    on dbselect
    {
      variable object NewObj := null;

      !! sichtbares Erzeugen
      !! NewObj ::= MWnDetail:create(this.dialog);
      !! unsichtbares Erzeugen
      NewObj ::= MWnDetail:create(this.dialog, true);
      NewObj:GetData();
    }
```

```
  }
}
```

Siehe auch

Eingebaute Funktion create() im Handbuch „Regelsprache“

C-Funktion DM_CreateObject im Handbuch „C-Schnittstelle - Funktionen“

COBOL-Funktion DMcob_CreateObject im Handbuch „COBOL-Schnittstelle“

2.12 :delete()

Die Methode **:delete()** gibt es in drei verschiedenen Ausprägungen:

» **Löschen von Einträgen aus Listenobjekten**
» **Löschen von Elementen aus Feldern (indizierten benutzerdefinierten Attributen)**
» **Löschen des DOM-Knotens auf den ein XML-Cursor zeigt**

2.12.1 :delete() (Listenobjekte)

Mit Hilfe dieser Methode können Zeilen und Spalten aus dem Inhalt eines Objektes gelöscht werden, wenn der Inhalt dieses Objekts aus mehreren, mit *integer* oder *index* indizierten, Einträgen besteht. Zusammen mit dem Inhalt werden auch die dazugehörenden Attribute gelöscht.

Im Unterschied zur **:clear()**-Methode werden mit **:delete()** Zeilen bzw. Spalten vollständig gelöscht, d.h. die Anzahl der Zeilen bzw. Spalten verringert sich durch **:delete()**.

Definition

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

Parameter

integer *Position* input
Dieser Parameter definiert die Position, ab der die Zeilen bzw. Spalten gelöscht werden sollen.

*integer **Count** := 1 input*
Mit diesem optionalen Parameter wird die Anzahl der zu löschenden Zeilen bzw. Spalten festgelegt. Wird der Parameter nicht angegeben, so wird *1* angenommen.

*boolean **Direction** := false input*
Dieser optionale Parameter steuert beim ***tablefield***, ob Zeilen oder Spalten gelöscht werden. Dies ist abhängig vom Wert des ***tablefield***-Attributs *.direction*.
Der Parameter darf nur beim ***tablefield*** angegeben werden. Bei anderen Objekten als dem ***tablefield*** führt die Angabe des Parameters zu einem Fehler.

Wertebereich

false
Löschen entgegen der vom Attribut *.direction* angegebenen Richtung
true
Löschen in der vom Attribut *.direction* angegebenen Richtung

Damit werden beim ***tablefield*** folgende Inhalte gelöscht:

Parameter *Direction*	*.direction 1* (vertikal)	*.direction 2* (horizontal)
false	Zeile(n)	Spalte(n)
true	Spalte(n)	Zeile(n)

Rückgabewert

Die Methode gibt *true* zurück, wenn die Zeilen bzw. Spalten gelöscht werden konnten.

Wenn beim Löschen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

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

Beispiel

```
dialog D

window Wn
{
  .title "Beispiel fuer die Methode :delete()";

  child treeview Tv
  {
    .content[1] "Firma A";
    .content[2] "Mueller";
    .content[3] "Mayer";
    .content[4] "Firma B";
    .content[5] "Schulz";
    .content[6] "Schmidt";
    .content[7] "Fischer";
    .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 "Entferne einen Mitarbeiter";

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

2.12.2 :delete() (Felder)

Mit Hilfe dieser Methode können Elemente aus Feldern (indizierten benutzerdefinierten Attributen) gelöscht werden.

Im Unterschied zur **:clear()**-Methode werden die Elemente mit **:delete()** vollständig gelöscht, d.h. die Anzahl der Elemente verringert sich.

Definition

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

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

anyvalue *Position* input

In diesem Parameter wird der Index übergeben, ab dem die Elemente gelöscht werden sollen. Bei nicht-assoziativen Feldern hat *Position* den Datentyp *integer*, bei assoziativen Feldern den Datentyp, mit dem das assoziative Feld indiziert ist.
Der Wertebereich für *Position* bei nicht-assoziatven Feldern ist *0 … .count[Attr]*, d.h. es kann auch der Standardwert gelöscht werden. Wird der Standardwert gelöscht, dann wird der Wert des ersten Elements nach den gelöschten Elementen zum Standardwert.

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

Dieser optionale Parameter definiert die Anzahl der zu löschenden Elemente. Wird der Parameter nicht angegeben, so wird *1* angenommen.
Count kann bei assoziativen Feldern nicht verwendet werden, es kann immer nur *1* Element gelöscht werden.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Elemente gelöscht werden konnten.

Wenn beim Löschen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

Alle Objekte, die benutzerdefinierte Attribute haben können.

Siehe auch

Kapitel „Methoden für Arrays benutzerdefinierter Attribute“ im Handbuch „Benutzerdefinierte Attribute und Methoden“

Kapitel „Arbeiten mit assoziativen Arrays“ im Handbuch „Benutzerdefinierte Attribute und Methoden“

2.12.3 :delete() (doccursor)

Löscht den DOM-Knoten mit allen Kindknoten. Der XML-Cursor wird auf den Vaterknoten positioniert. XML-Cursor, die in den Unterbaum des gelöschten DOM-Knotens zeigen, werden ungültig. Bei einem ungültigen XML-Cursor besitzt das Attribut *.mapped* den Wert *false*.

Werden die Werte des Attributs *.path* woanders gespeichert (zum Beispiel im Attribut *.userdata*), dann ist zu beachten, dass diese gespeicherten Werte nicht angepasst werden, wenn die Struktur des DOM-Baums verändert wird. Ein anschließender Aufruf der Methode **:select** mit einem dieser gespeicherten Werte, kann demzufolge den XML-Cursor auf einen falschen DOM-Knoten zeigen lassen.

Definition

```
boolean :delete
(
)
```

Parameter

Keine.

Rückgabewert

true
: DOM-Knoten konnte gelöscht werden.

false
: Beim Löschen ist ein Fehler aufgetreten.

Objekte mit dieser Methode

doccursor

2.13 :destroy()

Mit Hilfe dieser Methode können beliebige Objekte oder Modelle eines Dialogs gelöscht werden. Dabei werden alle Kinder des Objektes mit gelöscht.

Definition

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

Parameter

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

Mit Hilfe dieses optionalen Parameters (Standardwert *false*) wird gesteuert, wie dieses Objekt gelöscht werden soll. Wenn hier *true* angegeben wird, werden das Objekt gelöscht und alle Regelteile, die dieses Objekt benutzen abgeändert, sodass die entsprechenden Anweisungen entfernt werden.
Wenn es sich bei dem zu löschenden Objekt um ein Modell handelt, wird und der Parameter ist *false*, dann wird das Modell nur gelöscht, wenn es von keinem anderen Objekt benutzt wird (Standard). Wird in diesem Fall *true* angegeben, dann wird das Modell gelöscht und die Objekte, die dieses Modell benutzen, beziehen sich wieder auf das nächsthöhere Modell bzw. den Default.

Rückgabewert

true
: Objekt konnte gelöscht werden.

false
: Objekt konnte nicht gelöscht werden.

Hinweis

:destroy() ruft die **:clean()**-Methode des Objekts auf.

Beispiel

```
dialog Beispiel

model window MWnDetail
{
  !! Detailanzeige von Datensaetzen
  .title "Detailanzeige der Daten";
  rule void GetData()
  {
    !! Holen der Daten
    !! Fenster sichtbar machen
    this.visible ::= true;
```

```
  }
  child pushbutton PbExit
  {
    .text "&Abbrechen";
    on select
    {
      !! zerstoert das Instanzen-Fenster
      this.window:destroy();
    }
  }
}

window WnMain
{
  .title "Daten\374bersicht";
  child tablefield TfData
  {
    !! enthaelt eine Datenuebersicht
    on dbselect
    {
      variable object NewObj := null;

      !! sichtbares Erzeugen
      !! NewObj ::= MWnDetail:create(this.dialog);
      !! unsichtbares Erzeugen
      NewObj ::= MWnDetail:create(this.dialog, true);
      NewObj:GetData();
    }
  }
}
```

Siehe auch

Eingebaute Funktion destroy() im Handbuch „Regelsprache“

C-Funktion DM_Destroy im Handbuch „C-Schnittstelle - Funktionen“

COBOL-Funktion DMcob_Destroy im Handbuch „COBOL-Schnittstelle“

2.14 :exchange()

Die Methode **:exchange()** gibt es in zwei verschiedenen Ausprägungen:

» **Vertauschen von Einträgen in Listenobjekten**

» **Vertauschen von Elementen in Feldern (indizierten benutzerdefinierten Attributen)**

2.14.1 :exchange() (Listenobjekte)

Mit Hilfe dieser Methode können zwei Zeilen bzw. Spalten des Inhalts eines Objektes miteinander vertauscht werden, wenn der Inhalt dieses Objekts aus mehreren, mit *integer* oder *index* indizierten, Einträgen besteht. Zusammen mit dem Inhalt werden auch die dazugehörenden Attribute getauscht.

Definition

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

Parameter

integer *Position1* input
integer *Position2* input

Diese Parameter definieren die beiden Zeilen bzw. spalten, die miteinander vertauscht werden sollen.

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

Dieser optionale Parameter steuert beim ***tablefield***, ob Zeilen oder Spalten vertauscht werden. Dies ist abhängig vom Wert des ***tablefield***-Attributs *.direction*.
Der Parameter darf nur beim ***tablefield*** angegeben werden. Bei anderen Objekten als dem ***tablefield*** führt die Angabe des Parameters zu einem Fehler.

Wertebereich

false
Vertauschen entgegen der vom Attribut *.direction* angegebenen Richtung

true
Vertauschen in der vom Attribut *.direction* angegebenen Richtung

Damit werden beim ***tablefield*** folgende Inhalte vertauscht:

Parameter *Direction*	*.direction 1* (vertikal)	*.direction 2* (horizontal)
false	Zeile(n)	Spalte(n)
true	Spalte(n)	Zeile(n)

Rückgabewert

Die Methode gibt *true* zurück, wenn die Zeilen bzw. Spalten vertauscht werden konnten.

Wenn beim Vertauschen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

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

Beispiel

```
dialog D

window Wn
{
  .title "Beispiel fuer die Methode :exchange()";

  child treeview Tv
  {
    .content[1] "Firma A";
    .content[2] "Mueller";
    .content[3] "Mayer";
    .content[4] "Firma B";
    .content[5] "Schulz";
    .content[6] "Schmidt";
    .content[7] "Fischer";
    .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 "Selektieren Sie einen Mitarbeiter";
  }

  child pushbutton Pb
  {
```

```
    .text "Vertausche zwei Mitarbeiter";

    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 := "Tausch ok";
        else
          St.text := "Oberer Partner ist eine Firma";
        endif
      else
        St.text := "Es gibt keinen oberen Partner";
      endif
    }
  }
}
```

2.14.2 :exchange() (Felder)

Mit Hilfe dieser Methode können zwei Elemente eines Feldes (indizierten, nicht-assoziativen, benutzerdefinierten Attributs) miteinander vertauscht werden.

Definition

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

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

integer *Position1* input
integer *Position2* input

In diesen Parametern werden die Indizes der beiden Elemente übergeben, die miteinander vertauscht werden sollen.
Der Wertebereich für beide Parameter ist *0 … .count[Attr]*, d.h. es kann auch der Standardwert mit dem Wert eines anderen Elements vertauscht werden.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Elemente vertauscht werden konnten.

Wenn beim Vertauschen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

Alle Objekte, die benutzerdefinierte Attribute haben können.

Siehe auch

Kapitel „Methoden für Arrays benutzerdefinierter Attribute“ im Handbuch „Benutzerdefinierte Attribute und Methoden“

2.15 :find()

Diese Methode sucht in vordefinierten und benutzerdefinierten, indizierten Attributen (eindimensionalen, zweidimensionalen und assoziativen Feldern) nach einem bestimmten Wert und gibt den ersten Index zurück an der dieser Suchwert zu finden ist.

Die Suche schließt dabei niemals das Standardelement (also z.B. Index *[0]* oder *[0,0]*) ein.

Durch Angabe eines Start- und Endindex kann die Suche beschränkt werden. Die gültigen Werte sind:

- Kein Wert angegeben (*void*).
- *integer*-Wert im Bereich von *1 … Feldgröße* (muss aber > *0* sein) für (assoziative) Felder.
- *index*-Wert im Bereich von *[1,1]* bis *[rowcount,colcount]* wobei auch hier *rowcount* und *colcount* > *0* sein müssen.

Alle anderen Werte erzeugen ein `fail`.

Dies impliziert, dass für die Suche in einem kompletten Feld ohne Start- und Endindex gearbeitet werden sollte, da ein Start- oder Endindex von *[0]* oder *[0,0]* ein `fail` verursacht.

Bei der Suche nach Strings kann auch „case-insensitive“ oder nach dem ersten Vorkommen als Präfix gesucht werden.

Die Suche wird nur im Bereich *1 … n* bzw. *[1,1] … [n,m]* durchgeführt, *0*-Indizes sind also nicht erlaubt.

Da die Indizes von assoziativen Feldern keiner Ordnung unterworfen sind, darf als Index nur ein *integer*-Wert im Bereich *1 … itemcount[<Attribut>]* angegeben werden, der die Position des Start- bzw. End-Elements angibt.

Bei zweidimensionalen Feldern (*.content[]* am ***tablefield***) bewirkt die Angabe eines Indexbereichs *[Sy,Sx]* bis *[Ey,Ex]* die Einschränkung des Suchbereichs auf den Bereich *[min(Sy,Ey),min(Sx,Ex)] … [max(Sy,Ey),max(Sx,Ex)]*. Somit ist die Einschränkung auf Zeilen und Spalten problemlos möglich.

Die Suche erfolgt in Richtung der angegebenen Indizes bzw. unter Berücksichtigung von *.direction* (nur für zweidimensionale Felder). Die Angabe von *10, 1* anstatt *1, 10* als Start- und Endindizes bewirkt also die Umkehrung der Suchrichtung. Ein vertikal ausgerichtetes zweidimensionales Feld wird in der Reihenfolge Zeilen/Spalten durchsucht, bei horizontaler Ausrichtung ist dies genau umgekehrt. Es wird der Index des ersten, dem Wert entsprechenden Feldes zurückgegeben.

Definition

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

```
  { , enum      CompareMode   := match_exact input }
)
```

Parameter

attribute *Attribute* input

In diesem Parameter wird das vordefinierte oder benutzerdefinierte Attribut angegeben, in dem nach dem Wert gesucht werden soll.
Wenn dieser Parameter nicht angegeben ist, dann wird bei ***poptext*** und ***spinbox*** das Attribut *.text [integer]* durchsucht, bei allen anderen Objekten das Attribut *.content[]*.

anyvalue *Value* input

Nach dem in diesem Parameter angegebenen Wert wird in dem indizierten Attribut gesucht. Grundsätzlich werden nur Werte verglichen, deren Typen „gleich“ bzw. überführbar sind (ein ***text*** wird hierzu in einen *string* überführt).
Wenn dieser Wert ein String oder eine ***text***-Ressource ist, werden auch die Parameter *CaseSensitive* und *CompareMode* für die Suche herangezogen.

*anyvalue **StartIdx** input*

In diesem optionalen Parameter wird der Startindex angegeben, ab dem in dem indizierten Attribut nach einem Wert gesucht werden soll. Wird hier kein Wert angegeben, wird *1* bei eindimensionalen bzw. *[1,1]* bei zweidimensionalen Attributen angenommen.

*anyvalue **EndIdx** input*

In diesem optionalen Parameter wird der Endindex angegeben, bis zu dem in dem indizierten Attribut nach einem Wert gesucht werden soll. Die Angabe ist nur zulässig, wenn auch ein Startindex angegeben wurde.

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

Dieser optionale Parameter findet nur bei *string*-Werten Beachtung und definiert, ob die Suche unter Beachtung der Groß-/Kleinschreibung erfolgen soll oder nicht. Der Standardwert ist *false*.

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

Dieser optionale Parameter findet nur bei *string*-Werten Beachtung und definiert wie nach einem String gesucht werden soll.

Wertebereich

match_begin
: Findet den ersten String, der mit dem Suchstring beginnt.

match_exact
: Findet den ersten String, der mit dem gesuchten String exakt übereinstimmt.

match_first
: Findet den String, dessen Anfang die größte Übereinstimmung mit dem Suchstring aufweist. Gibt es einen exakt mit dem Suchstring übereinstimmenden String, wird dessen Index zurückgegeben.

match_substr
: Findet den ersten String, der den gesuchten String enthält.

Rückgabewert

0
: Element wurde in dem übergebenen eindimensionalen, nicht assoziativen Attribut nicht gefunden.

[0,0]
: Element wurde in dem übergebenen zweidimensionalen Attribut nicht gefunden.

nothing
: Element wurde in dem übergebenen assoziativen Feld nicht gefunden.

sonst
: Index des gesuchten Elements in dem übergebenen Attribut. Der zurückgegebene Wert hat den Index-Datentyp des übergebenen Attributs.

Fehlerverhalten

Der Aufruf der Methode schlägt fehl, wenn das Attribut nicht bekannt ist oder der durch *StartIdx* und *EndIdx* definierte Bereich nicht innerhalb des gültigen Indexbereichs liegt.

Objekte mit dieser Methode

- listbox (Standard-Attribut *.content[integer]*)
- poptext (Standard-Attribut *.text[integer]*)
- spinbox (Standard-Attribut *.text[integer]*)
- tablefield (Standard-Attribut *.content[index]*)
- treeview (Standard-Attribut *.content[integer]*)
- Alle Objekte, die benutzerdefinierte Attribute haben können.

Beispiel

```
window Wn
{
  .title "Beispiel fuer die Methode :find()";

  child listbox Lb
  {
    .content[1] "Klettenheimer";
    .content[2] "Strassenmeier";
    .content[3] "Redweik";
    .content[4] "Tilly";
    .content[5] "Mueller";
    .content[6] "Meyer";
    .content[7] "Schulz";

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

```
      !! Beachtet Gross-/Kleinschreibung und sucht nach erstem passenden
String.
      Index := Lb:find(What, true, match_begin);
      if (Index = 0) then
        Et_find.content := "Keinen Eintrag gefunden";
      else
        Lb.activeitem := Index;
        Et_find.content := "Naechster passender String";
      endif
    }
  }

  child pushbutton Pb_find
  {
    .text "Suche Eintrag";

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

  child edittext Et_find
  {
     .content "Welchen Eintrag";
  }
}
```

Siehe auch

Eingebaute Funktion find()

2.16 :findtext()

Die Methode **:findtext()** sucht im Text (*.content*) eines Edittextes nach dem angegebenen String.

Definition

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

Parameter

string *Text* input

In diesem Parameter wird der String übergeben, nach dem der Inhalt (*.content*) des Edittextes durchsucht werden soll.
Ein leerer Text wird sofort gefunden. Der Rückgabewert von **:findtext()** ist dann *Start*, wenn ein Bereich angegeben wurde, und die Cursorposition bzw. die Endposition der Selektion, wenn kein Bereich angegeben wurde.
Der zu suchende Text ist beim RTF-Edittext (*.options[opt_rtf] = true*) als reiner Text ohne Formatierungsanweisungen („Plain Text“) zu übergeben.

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

Mit diesen optionalen Parametern wird der Bereich definiert, in dem nach dem angegebenen String gesucht werden soll. Wird kein Bereich angegeben, beginnt die Suche an der Cursorposition bzw. hinter der Selektion und geht bis zum Ende des Textes.
Der Wertebereich von *Start* und *End* geht von *0* bis zur Anzahl der Zeichen im angezeigten Text, wobei jedes Zeichen zählt, das in eine Selektion eingeschlossen werden kann. Zeilenumbruch-Zeichen zählen daher mit.
Für *End* kann *-1* angegeben werden, um den Inhalt des Edittextes bis zu seinem Ende zu durchsuchen.
Beim RTF-Edittext (*.options[opt_rtf] = true*) beziehen sich *Start* und *End* – analog zu *.startsel* und *.endsel* – auf Positionen im formatierten Text. Aus ihnen kann **nicht** auf Positionen im Content-String des RTF-Edittextes geschlossen werden, da dieser zusätzlich Formatierungsanweisungen enthält. Sind die Parameter *Start* oder *End* größer als die Textlänge, dann wird beim RTF-Edittext die Textlänge verwendet.

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

Dieser optionale Parameter legt fest, ob die Groß- und Kleinschreibung bei der Suche berücksichtigt werden soll.

true
Groß- und Kleinschreibung der gefundenen Texte muss mit dem angegebenen String übereinstimmen.

false
Groß- und Kleinschreibung der gefundenen Texte kann vom angegebenen String abweichen.

Rückgabewert

-1
Angegebener String wurde nicht gefunden.

>= 0
Anfangsposition der ersten Fundstelle des angegebenen Strings im Text.

Objekte mit dieser Methode

edittext

2.17 :get()

Mit dieser Methode können an Objekten deren Attribute abgefragt werden. Diese Methode ist an allen Objekten, Modellen und Defaults vordefiniert.

Darüber hinaus ist es möglich, diese Methode zu überschreiben und somit die Art und Weise der Attributabfrage um weitere Aktionen (z.B. Berechnung der Attributwerte erst wenn diese tatsächlich benötigt werden) zu erweitern.

Die dabei zulässigen Attribute für den jeweiligen Objekttyp entnehmen Sie bitte der „Objektreferenz“.

Definition

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

Parameter

attribute *Attribute* input

Dieser Parameter enthält das Attribut, dessen Wert abgefragt wird.

anyvalue ***Index*** *input*

In diesem optionalen Parameter wird der Index des gesuchten Attributs angegeben, falls dieses ein indiziertes Attribut ist. Sein Datentyp muss dem Index-Datentyp des Attributs entsprechen. Also muss hier ein *integer*-Wert übergeben werden, wenn das Attribut eindimensional ist oder ein *index*-Wert, wenn das Attribut zweidimensional ist.

Rückgabewert

Wert des gesuchten Attributes mithilfe von `return`.

Die Weitergabe von `fail` in überschriebenen, vordefinierten Methoden ist ebenfalls möglich. Dazu gibt es folgende Schlüsselwörter in der Regelsprache:

Syntax

```
pass <Ausdruck> ;
```

Wertet einen Ausdruck analog zu einer `return`-Anweisung aus, um das Resultat daraus an den Aufrufer zurückzugeben. Kommt es dabei zu einem Fehler, so wird dieser Fehlerzustand an den Aufrufer weitergereicht. Eine normale `return`-Anweisung würde hier abbrechen und bei der nächsten Anweisung weitermachen.

Damit entspricht es quasi der Vereinfachung von

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

Syntax

```
throw ;
```

Mit diesem Schlüsselwort wird ein Fehler an den Aufrufer weiter signalisiert und die aktuelle Methode beendet.

Beispiel

```
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();
}
```

Impliziter Aufruf

Die Methode **:get()** wird auch implizit aufgerufen. Immer wenn ein Attribut (vordefiniert oder benutzerdefiniert) aus der Regelsprache oder mit einer Schnittstellenfunktion (z.B. **DM_GetValue()**) abgefragt wird, wird die Methode **:get()** aufgerufen, die dann den Wert des Attributs zurückliefern muss.

Redefinition

Die Redefinition der Methode **:get()** wird wie folgt durchgeführt:

```
window W {
  :get(<keine Parameter!>)
  {
    ...
    Aktionen, um den Wert des gesuchten Attributs zu berechnen.
    ...
  }
}
```

Innerhalb der Methodendefinition können die Parameter *Attribute* und *Index* abgefragt und gesetzt werden, um dann z.B. mit *this:super()* die Methode **:get()** der Superklasse mit den entsprechend veränderten Parametern aufzurufen, was allerdings in der Regel nicht sinnvoll ist.

Unterdrückung von rekursiven Aufrufen

Wird in der Methode **:get()** irgendein Attribut des selben Objektes, für das die Methode **:get()** bereits aufgerufen worden ist, abgefragt, so müsste eigentlich wieder die Methode **:get()** für das selbe Objekt implizit aufgerufen werden. Da dabei die Gefahr von Endlosrekursionen sehr groß ist (in vielen Fällen wäre das sogar unvermeidlich), wird ein solcher rekursiver Aufruf unterdrückt.

Beispiel

```
window {
  :get()
  {
    case (Attribute)
      in .title:
        return "<" + this.title + ">";  // kein impliziter Aufruf von :get()
      otherwise:
        return this:super();
    endcase
  }
}
```

In diesem Beispiel werden in der Methode **:get()** Abfragen des Fenstertitels überwacht und der Titel eingeklammert in „<>“ zurückgeliefert. Dabei wird auf *this.title* wiederum lesend zugegriffen. Ein impliziter Aufruf der Methode **:get()** wird jedoch unterdrückt. Die Methode **:get()** für ein anderes Objekt wird sehr wohl aufgerufen, falls die Attribute des anderen Objektes abgefragt werden.

Aufruf der Methode :get() an der übergeordneten Klasse

Das obige Beispiel zeigt auch den Umgang mit *this:super()*. Alle Attribute, für die sich die überschriebene Methode **:get()** nicht zuständig fühlt, sollten an die Superklasse delegiert werden, so dass irgendwann die vordefinierte Methode **:get()** den Attributwert zurückliefert.

Beispiel

Folgendes Beispiel soll die Aufrufhierarchie der **:get()**-Methode verdeutlichen.

```
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";
  // impliziter Aufruf der :get()-Methode
  print "*** Ergebnis Str: " + RecInst.Str;
    // Soll "MRec1(X).MRec2(X).RecInst(X)"
  // expliziter Aufruf der :get()-Methode
  print "*** Ergebnis Str: " + RecInst:get(.Str);
    // Soll "MRec1(X).MRec2(X).RecInst(X)"

  exit();
```

```
}
```

Siehe auch

Eingebaute Funktion getvalue() im Handbuch „Regelsprache“

C-Funktion DM_GetValue im Handbuch „C-Schnittstelle - Funktionen“

2.18 :getformat()

Mit der Methode **:getformat()** kann die Formatierung eines Textbereichs im RTF-Edittext erfragt werden. **:getformat()** liefert einen Wert zurück, der die Ausprägung des angegebenen Formatierungstyps (z. B. Schriftart, Schriftfarbe, Textausrichtung) angibt.

Definition

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

Parameter

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

Mit diesen optionalen Parametern wird der Bereich definiert, dessen Formatierung ermittelt werden soll. Fehlen diese Angaben, wird der Bereich von *.startsel* bis *.endsel* (Selektion bzw. Cursorposition) genommen.
Der Wertebereich von *Start* und *End* geht von *0* bis zur Anzahl der Zeichen im angezeigten Text, wobei jedes Zeichen zählt, das in eine Selektion eingeschlossen werden kann. Zeilenumbruch-Zeichen zählen daher mit.
Für *End* kann -1 angegeben werden, um die Formatierung des restlichen Textes ab der Startposition zu erhalten.
Start und *End* beziehen sich – analog zu *.startsel* und *.endsel* – auf Positionen im formatierten Text. Aus ihnen kann **nicht** auf Positionen im Content-String des RTF-Edittextes geschlossen werden, da dieser zusätzlich Formatierungsanweisungen enthält. Sind die Parameter *Start* oder *End* größer als die Textlänge, dann wird beim RTF-Edittext die Textlänge verwendet.
Wenn beim Aufruf keine Zeichen markiert sind (*.startsel* = *.endsel*) oder *Start* = *End* ist, dann wird das Format **links** von der angegebenen Position geliefert.

enum *Type* input

Dieser Parameter gibt den Typ der Formatierung an, deren Wert abgefragt werden soll. Die *enum*-Werte für die Formatierungstypen sind in der nachfolgenden Tabelle aufgelistet.

Rückgabewert

anyvalue

Ein Wert für die Ausprägung des angegebenen Formatierungstyps.
Wenn der Bereich, für den die Formatierung erfragt werden soll, mehrere Formatierungen des angegebenen Formatierungstyps enthält, wird *nothing* zurückgegeben.

Formatierungsarten, *enum*-Werte und Datentypen der Formatierungstypen

Formatierungstyp	Formatierungsart	*enum*-Wert	Datentyp
Schriftart, Font	Zeichenformat	*text_font*	*string*
Schriftgröße	Zeichenformat	*text_size*	*integer*
Vordergrundfarbe, Textfarbe	Zeichenformat	*text_fgc*	*integer*
Hintergrundfarbe	Zeichenformat	*text_bgc*	*integer*
fett	Zeichenformat	*text_bold*	*boolean*
kursiv	Zeichenformat	*text_italic*	*boolean*
unterstrichen	Zeichenformat	*text_underline*	*boolean*
Einrückung links	Absatzformat	*text_indent_left*	*integer*
Einrückung rechts	Absatzformat	*text_indent_right*	*integer*
Einrückung der Folgezeilen in einem Absatz	Absatzformat	*text_indent_offset*	*integer*
Textausrichtung	Absatzformat	*text_align*	*enum* [*align_left*, *align_right*, *align_center*, *align_justify*]

Maßeinheit für Größen und Positionen

Größen und Positionen werden – wie bei der Windows-Programmierung üblich – in „twips“ angegeben. Ein twip entspricht *1/20 point* (Punkt), das ist *1/1440 Zoll* (inch) bzw. *1/567 cm*.

Erläuterungen zu einzelnen Formatierungen

» Der Wert für *text_indent_right* ist eine absolute Angabe der Einrückung vom rechten Rand.

» Dagegen ist *text_indent_left* eine relative Angabe, die mehrfach angewendet werden kann. Die Einrückungen vom linken Rand werden akkumuliert, können insgesamt aber nie über den Rand hinausgehen. Der Wert kann negativ sein und bedeutet dann eine Verschiebung eines Absatzes nach links, während positive Werte eine Verschiebung nach rechts bedeuten.

- Der Wert von *text_indent_offset* gibt an, wie weit alle Zeilen eines Absatzes außer der ersten gegenüber der ersten Zeile nach rechts (positiver Wert) oder nach links (negativer Wert) verschoben sind.
- Für *text_font* liefert **:getformat()** einen String mit dem Fontnamen zurück.
- Für Vordergrundfarbe (*text_fgc*) und Hintergrundfarbe (*text_bgc*) werden *integer*-Werte zurückgegeben, die als RGB-Werte zu interpretieren sind.

Zuverlässigkeit der Rückgabewerte

:getformat() kann nur zuverlässig funktionieren, wenn der RTF-Edittext sichtbar (*.visible* = *true*) ist, da das zugrundeliegende Windows-Objekt einige Berechnungen für die Formatierung nur im sichtbaren Zustand ausführt. Beispielsweise kann die Fontinformation im unsichtbaren Zustand nicht abgefragt werden.

Objekte mit dieser Methode

edittext mit *.options[opt_rtf]* = *true*

2.19 :gettext()

Die Methode **:gettext()** liefert Text aus dem Inhalt eines Edittextes zurück.

Definition

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

Parameter

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

Mit diesen optionalen Parametern wird der Bereich definiert, dessen Text zurückgegeben werden soll. Wenn kein Bereich angegeben ist, wird der Text von *.startsel* bis *.endsel* zurückgegeben, also der Inhalt der Selektion.
Der Wertebereich von *Start* und *End* geht von *0* bis zur Anzahl der Zeichen im angezeigten Text, wobei jedes Zeichen zählt, das in eine Selektion eingeschlossen werden kann. Zeilenumbruch-Zeichen zählen daher mit.
Für *End* kann -1 angegeben werden, um den restlichen Text ab der Startposition zu erhalten.
Beim RTF-Edittext (*.options[opt_rtf] = true*) beziehen sich *Start* und *End* – analog zu *.startsel* und *.endsel* – auf Positionen im formatierten Text. Aus ihnen kann **nicht** auf Positionen im Content-String des RTF-Edittextes geschlossen werden, da dieser zusätzlich Formatierungsanweisungen enthält. Sind die Parameter *Start* oder *End* größer als die Textlänge, dann wird beim RTF-Edittext die Textlänge verwendet.

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

Mit diesem optionalen Parameter kann beim RTF-Edittext festgelegt werden, ob der Inhalt des Textbereichs als unformatierter, reiner Text („Plain Text“) oder als RTF inklusive der Formatierungsanweisungen zurückgegeben wird.

Wertebereich

content_plain
String enthält reinen, unformatierten Text.
content_rtf
String enthält RTF-Text.
Nur verwendbar bei *.options[opt_rtf] = true*.

Rückgabewert

String mit dem Inhalt des angegebenen Textbereichs.

Objekte mit dieser Methode

edittext

2.20 :has()

Mit Hilfe dieser Methode kann man dynamisch zur Laufzeit abfragen, ob ein bestimmtes Objekt eine bestimmte benutzerdefinierte Methode hat oder ein bestimmtes benutzerdefiniertes Attribut besitzt.

Die Methode ist **nicht** für vordefinierte Methoden oder Attribute zulässig.

Definition

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

Parameter

anyvalue *MethodOrAttr* input

In diesem Parameter wird das Attribut oder die Methode angegeben, die überprüft werden soll.

*anyvalue **Index** input*

Mit Hilfe dieses optionalen Parameters kann bei benutzerdefinierten Attributen abgefragt werden, ob das Attribut vorhanden ist und ein Zugriff mit dem angegebenen Index zulässig wäre.

Rückgabewert

true
: Attribut bzw. Methode ist bei dem Objekt vorhanden.

false
: Attribut bzw. Methode ist bei dem Objekt **nicht** vorhanden.

Objekte mit dieser Methode

» Alle Objekte, die benutzerdefinierte Attribute haben können.

» Alle Objekte, die benutzerdefinierte Methoden haben können.

Beispiel

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

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

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

2.21 :index()

Der Dialog Manager bietet die Möglichkeit, ein assoziatives Array in seiner Gesamtheit zu durchlaufen. Mit der Methode **:index()** kann der Index aus der inneren Ordnung berechnet werden.

Achtung

Die innere Ordnung der assoziativen Arrays dient nur zur internen Verwaltung. Diese innere Ordnung kann sich ändern. Sollten Sie eine Ordnung auf den Daten benötigen, was dann im allgemeinen auf einen Missbrauch der assoziativen Arrays schließen lässt, so müssen sie diese selbst berechnen.

Definition

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

Parameter

attribute *Attribute* input

In diesem Parameter wird der Name des assoziativen Feldes angegeben, dessen Inhalt abgefragt werden soll.

integer *InternalIndex* input

In diesem Parameter wird der aktuell zu erfragende Index aus dem Feld angegeben. Bitte beachten Sie, dass dieser Index sich nach Lösch- oder Zuweisungsaktionen in dem Feld ändern kann.

Rückgabewert

0
: Element nicht im Feld enthalten

sonst
: Index des Elementes im assoziativen Feld

Objekte mit dieser Methode

Alle Objekte, die benutzerdefinierte Attribute haben können.

Beispiel

```
window WMain
{
  integer Channel [ string ];
    .Channel [ "RTL" ] := 65;
    .Channel [ "ARD" ] := 14;
    .Channel [ "ZDF" ] := 11;
}
```

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

In der Schleife wird das Feld von *1* bis zur Anzahl der Einträge durchlaufen, gemäß der inneren Ordnung. Im Schleifeninneren wird der Index berechnet (*WMain:index(.Channel, I)*) und mit diesem dann das assoziative Array „.Channel“ indiziert.

Siehe auch

Kapitel „Arbeiten mit assoziativen Arrays“ im Handbuch „Benutzerdefinierte Attribute und Methoden“

2.22 :init()

Die eingebaute Methode **:init()** ist eine vordefinierte, überschreibbare Methode. Das heißt diese Methode wird intern immer nach dem Anlegen eines Objektes ausgeführt, nur kann der Benutzer diese Methode redefinieren und eine eigene Initialisierung angeben. Dem Benutzer steht frei, wie die Reihenfolge der Initialisierung sein soll; ob zuerst das Vaterobjekt und dann die Kindobjekte oder gar erst teilweise den Vater, dann die Kindobjekte und danach die restliche Initialisierung des Vaters.

Die **:init()**-Methode kann nicht direkt aufgerufen werden.Sie hat vordefinierte Parameter auf die innerhalb der Methode zugegriffen werden kann. Die Methode hat keinen Rückgabewert (also *void*).

Definition

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

Parameter

Die vordefinierten Parameter sind:

object *Model* input
: Entspricht `this.model`.

object *Parent* input
: Entspricht `this.parent`.

object *Module* input
: Entspricht `this.module`.

integer *Type* input
: Entspricht `this.scope`.

boolean *Invisible* input
: Abhängig vom **create()**-Aufruf.

Redefinition

Die Methode kann direkt am Objekt überschrieben werden.

```
:init()
{
  // Regelcode wie gehabt
}
```

Bei der Redefinition werden weder die vordefinierten Parameter angegeben noch können zusätzliche Parameter angegeben werden.

Aufruf

Die Methode kann nicht direkt aufgerufen werden. Sie kann nur durch das Laden eines Moduls/Dialogs oder durch das dynamische Erzeugen von neuen Objekten mit den Funktionen **create()** oder **DM_CreateObject()** ausgelöst werden. Der direkte Aufruf wird ignoriert bzw. führt zu einem Fehler.

Ereignisverarbeitung nach dem Laden

Die **:init()**-Methode wird vor der *start*-Regel des Dialogs ausgeführt. Eventuelle Ereignisse können nicht berücksichtigt werden, sie werden nicht verarbeitet! Ebenso werden *changed*-Ereignisse nicht verarbeitet, es werden also keine *on changed* Regeln ausgelöst!

this:super()

Benutzung der Modellmethoden

Oftmals werden am Modell auch Initialisierungsmethoden definiert, diese sollen ebenfalls ausgeführt werden. Mit dem Aufruf von `this:super()` (ohne Parameter) werden die Modellmethoden ausgelöst.

Steuerung der Kinder

Existiert keine Modellmethode mehr und wird `this:super()` aufgerufen, dann werden die **:init()**-Methoden der Kinder aufgerufen. Existieren keine Kinder wird ohne Fehlermeldung zurückgekehrt. Besitzen die Kinder keine **:init()**-Methoden, dann wird an den Kindeskindern die **:init()**-Methoden aufgerufen. Damit kann der Benutzer steuern, wann und ob die Kinder initialisiert werden. Die Kinder des Dialoges werden **nicht** durch `this:super()` gesteuert.

Wichtiger Hinweis

Die Nutzung von **fail()** und *pass this:super()* sollte in der **:init()**-Methode unbedingt unterbleiben. In beiden Fällen werden auftretende Fehler ansonsten nicht mehr in den Trace- bzw. Logfile ausgegeben (Ausgabe von *„***ERROR IN EVAL“* unterbleibt). Dies erschwert bzw. unterbindet eine Fehlererkennung beträchtlich.

Beispiel 1

```
dialog D1

model window MW
{
  :init()
  {
    static variable integer I := 1;
    !! Instanzen reichen uns zur Berechnung
    if Type = 3 then
      this.title := "Fenster Nr. " + I;
      I := I + 1;
```

```
    endif
  }
}
```

Beispiel 2

```
dialog D2

model record MRecWin
{
  !! einige wichtige Daten
  string Wichtig[100];
}

model window MW
{
  !! speichere hier den korrespondierenden Record
  object MyRec;
  :init()
  {
    this.MyRec := create(MRecWin, D);
  }
}
```

Anmerkung

Das obige Beispiel dient nur zur Illustration, das Beispiel lässt sich wesentlich einfacher formulieren:

```
dialog D2_Besser

 model window MW
{
  record MyRec
  {
    !! einige wichtige Daten
    string Wichtig[100];
  }
}
```

Beispiel 3

```
dialog D3

model window MW
{
  child groupbox G
  {
```

```
    :init()
    {
      !! mache etwas mit G
    }
  }
  :init()
  {
    !! mache etwas bevor :init() von den Kindern aufgerufen wird
    !! ...
    this:super();
    !! mache etwas nachdem :init() der Kinder ausgefuehrt wurde
    !! ...
  }
}
```

Beispiel 4

```
dialog D4

model window MW1
{
  child groupbox G
  {
    :init()
    {
      !! mache etwas mit G
    }
  }
  :init()
  {
    if Type = 3 then
      !! G wird nur initialisiert, wenn das this-Objekt kein Modell ist
      this:super ();
    endif
  }
}

model MW1 MW2
{
  :init()
  {
    !! mache etwas mit dem Objekt, hier wird die Methode von MW1 ausgefuehrt
    this:super ();
    !! mache noch etwas mit dem Objekt
  }
}
```

Beispiel 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);
}
```

Ausgabe

```
W
MG
D5
3
false
```

2.23 :insert()

Die Methode **:insert()** gibt es in zwei verschiedenen Ausprägungen:

» **Einfügen von Einträgen in Listenobjekten**

» **Einfügen von Elementen in Feldern (indizierten benutzerdefinierten Attributen)**

2.23.1 :insert() (Listenobjekte)

Mit Hilfe dieser Methode können Zeilen und Spalten in den Inhalt eines Objektes eingefügt werden, wenn der Inhalt dieses Objekts aus mehreren, mit *integer* oder *index* indizierten, Einträgen besteht. Das Einfügen ist auch möglich, wenn noch keine Einträge vorhanden sind. Beim Einfügen werden auch die dazugehörenden Attribute entsprechend erweitert. Die neu eingefügten Einträge sind „leer“ bzw. weisen – sofern vorhanden – den Standardwert auf.

Definition

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

Parameter

integer *Position* input

Dieser Parameter definiert, an welcher Stelle die neuen Zeilen bzw. Spalten eingefügt werden. Wenn sein Wert *0* ist, werden die Zeilen bzw. Spalten am Ende angehägt.

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

Mit diesem optionalen Parameter wird die Anzahl der einzufügenden Zeilen bzw. Spalten festgelegt. Wird der Parameter nicht angegeben, so wird *1* angenommen.

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

Dieser optionale Parameter steuert beim ***tablefield***, ob Zeilen oder Spalten eingefügt werden. Dies ist abhängig vom Wert des ***tablefield***-Attributs *.direction*.
Der Parameter darf nur beim ***tablefield*** angegeben werden. Bei anderen Objekten als dem ***tablefield*** führt die Angabe des Parameters zu einem Fehler.

Wertebereich

false
: Einfügen entgegen der vom Attribut *.direction* angegebenen Richtung

true
: Einfügen in der vom Attribut *.direction* angegebenen Richtung

Damit werden beim ***tablefield*** folgende Inhalte eingefügt:

Parameter *Direction*	*.direction 1* (vertikal)	*.direction 2* (horizontal)
false	Zeile(n)	Spalte(n)
true	Spalte(n)	Zeile(n)

Rückgabewert

Die Methode gibt *true* zurück, wenn die Zeilen bzw. Spalten eingefügt werden konnten.

Wenn beim Einfügen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

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

Beispiel

```
dialog D

window Wn
{
  .title "Beispiel fuer die Methode :insert()";

  child treeview Tv
  {
    .content[1] "Firma A";
    .content[2] "Mueller";
    .content[3] "Mayer";
    .content[4] "Firma B";
    .content[5] "Schulz";
    .content[6] "Schmidt";
    .content[7] "Fischer";
    .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 "Fuege neuen Mitarbeiter ein";

    on select
    {
      if fail(atoi(Et_level.content)) then
        Et_level.content := "Sie haben keine Stufe angegeben";
      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 := "Auf welcher Stufe?";
      endif
    }
  }
  child edittext Et_name
  {
    .content "Hier Name eingeben";
  }
  child edittext Et_level
  {
    .content "Auf welcher Stufe?";
  }
}
```

2.23.2 :insert() (Felder)

Mit Hilfe dieser Methode können neue Elemente in Felder (indizierte, nicht-assoziative, benutzerdefinierte Attribute) eingefügt werden. Die neu eingefügten Elemente sind „leer“ bzw. weisen – sofern vorhanden – den Standardwert auf.

Definition

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

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

integer *Position* input

In diesem Parameter wird der Index übergeben, an dem die neuen Elemente eingefügt werden sollen. Wenn sein Wert *0* ist, werden die Elemente am Ende angehängt.

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

Dieser optionale Parameter definiert die Anzahl der einzufügenden Elemente. Wird der Parameter nicht angegeben, so wird *1* angenommen.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Elemente eingefügt werden konnten.

Wenn beim Einfügen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

Alle Objekte, die benutzerdefinierte Attribute haben können.

Siehe auch

Kapitel „Methoden für Arrays benutzerdefinierter Attribute“ im Handbuch „Benutzerdefinierte Attribute und Methoden“

2.24 :instance_of()

Mit Hilfe dieser Methode kann festgestellt werden, ob ein Objekt von einem bestimmten Modell abgeleitet wurde.

Definition

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

Parameter

object *Model* input
: Der Modellidentifikator des Modells nach dem innerhalb der *.model*-Kette gesucht werden soll.

Rückgabewert

Der Aufruf liefert *true* zurück sobald in der *.model*-Kette der Instanz das angegebene Modell auftaucht, ansonsten wird *false* zurückgegeben.

Siehe auch

Attribut model

2.25 :load()

Lädt ein XML-Dokument von der angegebenen Datei oder URL. Der gespeicherte DOM-Baum wird gelöscht und ein neuer DOM-Baum aufgebaut. Alle bestehenden XML-Cursor werden ungültig.

Bei einem ungültigen XML-Cursor besitzt das Attribut *.mapped* den Wert *false*.

Definition

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

Parameter

string *URL* input

Der Dateipfad des Dokuments, das geladen werden soll. An Stelle eines Pfades kann auch eine URL angegeben sein.

Rückgabewert

Gibt an, ob die Datei geladen werden konnte.

Im Fehlerfall bleibt der DOM-Baum des XML-Dokuments erhalten oder wird gelöscht, eine teilweise Transformation gibt es nicht.

Objekte mit dieser Methode

document

2.26 :match()

Testet, ob der DOM-Knoten dem angegebenen Muster entspricht.

Definition

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

Parameter

string *Pattern* input

Das Muster, gegen das der DOM-Knoten verglichen wird.
Der Aufbau des Musters ist in Kapitel „Muster für die Methoden :match() und :select()“ des Handbuchs „XML-Schnittstelle“ beschrieben.

Rückgabewert

Gibt an, ob der DOM-Knoten dem Muster entspricht.

Es ist zu beachten, dass ein XML-Cursor, dessen Attribut *.mapped* den Wert *false* besitzt, automatisch auf die Wurzel des DOM-Dokuments positioniert wird.

Objekte mit dieser Methode

doccursor

2.27 :move()

Die Methode **:move()** gibt es in zwei verschiedenen Ausprägungen:

» **Verschieben von Einträgen in Listenobjekten**

» **Verschieben von Elementen in Feldern (indizierten benutzerdefinierten Attributen)**

2.27.1 :move() (Listenobjekte)

Mit Hilfe dieser Methode können Zeilen und Spalten innerhalb des Inhalts eines Objektes an eine andere Stelle verschoben werden, wenn der Inhalt dieses Objekts aus mehreren, mit *integer* oder *index* indizierten, Einträgen besteht. Zusammen mit dem Inhalt werden auch die dazugehörenden Attribute entsprechend verschoben.

Definition

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

Parameter

integer *Position* input

Dieser Parameter definiert die erste zu verschiebende Zeile bzw. Spalte.

integer *Target* input

Dieser Parameter gibt an, wohin die Zeilen bzw. Spalten verschoben werden sollen.

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

Mit diesem optionalen Parameter wird die Anzahl der zu verschiebenden Zeilen bzw. Spalten festgelegt. Wird der Parameter nicht angegeben, so wird *1* angenommen.

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

Dieser optionale Parameter steuert beim ***tablefield***, ob Zeilen oder Spalten verschoben werden. Dies ist abhängig vom Wert des ***tablefield***-Attributs *.direction*.
Der Parameter darf nur beim ***tablefield*** angegeben werden. Bei anderen Objekten als dem ***tablefield*** führt die Angabe des Parameters zu einem Fehler.

Wertebereich

false

Verschieben entgegen der vom Attribut *.direction* angegebenen Richtung

true

Verschieben in der vom Attribut *.direction* angegebenen Richtung

Damit werden beim ***tablefield*** folgende Inhalte verschoben:

Parameter *Direction*	*.direction 1* (vertikal)	*.direction 2* (horizontal)
false	Zeile(n)	Spalte(n)
true	Spalte(n)	Zeile(n)

Rückgabewert

Keiner.

Wenn beim Verschieben ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

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

Beispiel

```
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() (Felder)

Mit Hilfe dieser Methode können Elemente eines Feldes (indizierten, nicht-assoziativen, benutzerdefinierten Attributs) an eine andere Stelle im Feld verschoben werden.

Definition

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

Parameter

attribute *Attr* input
: Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

integer *Position* input
: In diesem Parameter wird der Index des ersten Elements angegeben, das verschoben werden soll.

integer *Target* input
: Dieser Parameter legt fest, wohin die Elemente verschoben werden sollen.

integer *Count* input
: Dieser Parameter definiert, wie viele Elemente verschoben werden.

Rückgabewert

Keiner.

Wenn beim Verschieben ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

Objekte mit dieser Methode

Alle Objekte, die benutzerdefinierte Attribute haben können.

Siehe auch

Kapitel „Methoden für Arrays benutzerdefinierter Attribute“ im Handbuch „Benutzerdefinierte Attribute und Methoden“

2.28 :openpopup()

Mit Hilfe dieser Methode kann an einem Objekt das zugehörige Popup-Menü geöffnet werden.

Definition

```
boolean :openpopup
(
)
```

Parameter

Keine.

Rückgabewert

true
: Menü konnte geöffnet werden.

false
: Menü konnte **nicht** geöffnet werden.

Objekte mit dieser Methode

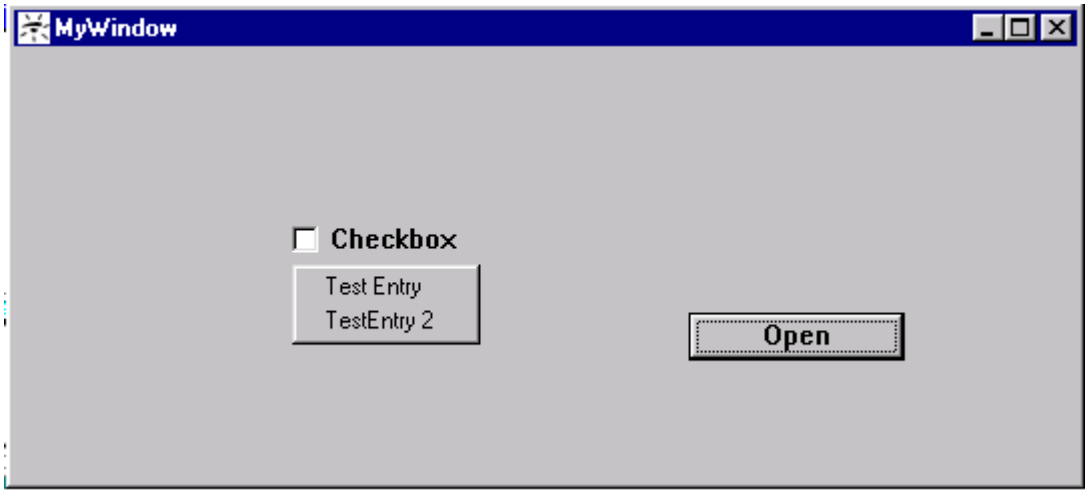
- canvas
- checkbox
- control
- edittext
- groupbox
- image
- layoutbox
- listbox
- notebook
- notepage
- poptext
- progressbar
- pushbutton
- radiobutton
- rectangle
- scrollbar
- spinbox
- splitbox
- statictext

- statusbar
- tablefield
- toolbar
- treeview

Die Methode existiert nicht für die Objekte ***Fenster*** und ***Menübox***.

Beispiel

```
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";
      }
    }
  }
}
```



Siehe auch

Attribut .menu

2.29 :parent()

Mit Hilfe dieser Methode kann innerhalb eines Treeviews von einem Element der zugehörige Vater erfragt werden.

Definition

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

Parameter

integer *Index* input

In diesem Parameter wird der Index des Elements angegeben, von dem der Vater berechnet werden soll.

Rückgabewert

0

Kein gültiges Element angegeben

andere Werte

Index des Vaters im Baum

Objekte mit dieser Methode

treeview

Beispiel

```
window Wn
{
  .title "Beispiel zur Methode :parent()";
  child treeview Tv
  {
    .content[1] "Firma A";
    .content[2] "Mueller";
    .content[3] "Meyer";
    .content[4] "FirmaCompany B";
    .content[5] "Schulz";
    .content[6] "Schmidt";
    .content[7] "Fischer";
    .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 "Kein Wert";
  }
  child pushbutton PbPar
  {
    .text "Bestimme die Firma eines Mitarbeiters";
    on select
    {
      if (Tv:parent(Tv.activeitem) = 0) then
        St.text := "Bitte Mitarbeiter selektieren";
      else
        St.text := Tv.content[Tv:parent(Tv.activeitem)];
      endif
    }
  }
}
```

2.30 :parent_of()

Mit Hilfe dieser Methode kann festgestellt werden, ob ein Objekt Vater, Großvater, Urgroßvater… eines bestimmten anderen (Kind-)Objekts ist. Von diesem (Kind-)Objekt aus gesehen kann also das „fragende“ Objekt mittels ein- oder mehrfachen rekursiven Aufrufs von *.parent* erreicht werden.

Definition

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

Parameter

object *Child* input

Der Identifikator des Kindobjekts nach dem gesucht werden soll.

Rückgabewert

Der Aufruf liefert *true* zurück wenn das im Parameter *Child* enthaltene Objekt ein Kind, Kindeskind… ist, ansonsten wird *false* zurückgegeben.

Siehe auch

Attribut parent

2.31 :propagate()

Beim Aufruf dieser Methode an einer Model-Komponente werden alle gekoppelten View-Komponenten dazu aufgefordert, ihre View-Attribute zu aktualisieren, die mit diesem Model-Objekt gekoppelt sind.

Normalerweise sollte ein Aufruf dieser Methode nicht notwendig werden, da die Synchronisation zwischen View und Model über das Attribut *.dataoptions[]* gesteuert wird.

Die Kopplung muss über die Attribute *.datamodel* und *.dataget* an der View-Komponente erfolgt sein.

Es erfolgt keine Fehlermeldung wenn das Model-Objekt keine gekoppelten Attribute besitzt.

Definition

```
void :propagate
(
)
```

Beispiel

```
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()

Diese Methode kann beliebige, vordefinierte und benutzerdefinierte Methoden aufrufen. Dies ist z.B. dann notwendig, wenn die Methode, die aufgerufen werden soll, selbst berechnet wird oder in einer Variablen gespeichert ist.

Der Aufruf der Methode **:recall()** ruft indirekt die angegebene Methode auf und liefert als Rückgabewert den Rückgabewert der aufgerufenen Methode zurück. Die Anzahl der Parameter die bei der Verwendung von **:recall()** mit übergeben werden können, ist jedoch auf maximal *15* beschränkt. Darauf ist beim Entwurf von eigenen Methoden zu achten.

Definition

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

Parameter

method *Method* input
In diesem Parameter wird die aufzurufende Methode übergeben.

*anyvalue **Par1** input*
…
*anyvalue **Par15** input*
In diesen Parametern werden die Parameter der aufzurufenden Methode übergeben. Die angegebenen Parameter müssen in Anzahl und Datentypen den Parametern der aufzurufenden Methode entsprechen.

Rückgabewert

Rückgabewert der aufzurufenden Methode.

Erzwingen des rekursiven Aufrufs anderer Methoden

Die Methode **:recall()** kann dazu benutzt werden, einen rekursiven Aufruf anderer Methoden zu erzwingen, z.B. von **:get()** und **:set()**. Dies unterscheidet sie von der Methode **:call()**.

Beispiel

```
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
        // Die Konsistenzpruefung fuer .Progress soll noch einmal
        // durchgefuehrt werden. Dazu wird :set() rekursiv aufgerufen.
        this:recall(:set, .Progress, this.Max, false);
      endif
    endcase
  }
}
on dialog start
{
  Rec.Progress := 90;  // Soll .Progress = 90.
  Rec.Max := 50;       // .Progress muss mit angepasst werden.
                       // Soll .Progress = 50, .Max = 50.
}
```

Siehe auch

Methode **:call()**

2.33 :reparent()

Mit Hilfe dieser Methode können Elemente eines Baumes umgehängt werden. Dabei gibt es zwei Ausprägungen von Bäumen im IDM:

» **Elemente eines treeviews**

» **Elemente eines XML-Dokuments**

Die Informationen zu dieser **Methode an einem treeview** finden Sie im nachfolgenden Kapitel, die zu **XML-Dokumenten** im übernächsten.

2.33.1 :reparent() (treeview)

Mit Hilfe dieser Methode kann im ***treeview*** ein beliebiger Teilbaum umgehängt werden.

Definition

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

Parameter

integer *Start* input
In diesem Parameter wird der Index des Teilbaumes angegeben, der umgehängt werden soll.

integer *Count* input
Mit Hilfe dieses Parameters wird angegeben, wie viele Elemente im Baum umgehängt werden sollen.

integer *Parent* input
In diesem Parameter wird der Index des neuen Vaters des Teilbaumes angegeben.

integer *Target* input
In diesem Parameter wird der absolute Index angegeben, hinter dem der verschobene Teilbaum eingefügt werden soll.

Rückgabewert

true
Der Teilbaum wurde erfolgreich umgehängt.
false
Das Umhängen des Teilbaumes hat nicht funktioniert.

Objekte mit dieser Methode

treeview

Beispiel

```
dialog D

window Wn

{
  .title "Beispiel fuer die Methode :reparent()";

  child treeview Tv
  {
    .content[1] "Firma A";
    .content[2] "Mueller";
    .content[3] "Mayer";
    .content[4] "Firma B";
    .content[5] "Schulz";
    .content[6] "Schmidt";
    .content[7] "Fischer";
    .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 "Firma angliedern";
```

```
    on select
    {
      variable integer Index;

      Tv.Index := Tv.activeitem;
      St.text := "Waehlen sie eine neue Firma";
      Tv.Select := true;
    }
  }

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

2.33.2 :reparent() (doccursor)

Diese Methode hängt den DOM-Knoten mit allen Kindknoten um.

Werden die Werte des *.path*-Attributs woanders gemerkt (zum Beispiel im *.userdata*-Attribut), dann ist zu beachten, dass diese gemerkten Werte nicht angepasst werden, wenn die Struktur des DOM-Baums verändert wird. Ein anschließender Aufruf der Methode **:select()** mit einem dieser gemerkten Werte, kann demzufolge den XML-Cursor auf einen falschen DOM-Knoten zeigen lassen.

Definition

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

boolean :reparent
(
  integer Index input
)
```

Parameter

object *Parent* input

Gibt den neuen Vaterknoten an. Muss vom Typ ***doccursor*** sein.
Ist kein Index angegeben, dann wird der DOM-Knoten als letzter eingefügt.

integer *Index* input

Gibt die Position an, an der der DOM-Knoten eingefügt werden soll. Ist *-1* als Index angegeben, dann wird der DOM-Knoten als letzter Knoten angefügt.

Rückgabewert

Gibt an, ob die Aktion erfolgreich war.

Objekte mit dieser Methode

doccursor

2.34 :replacetext()

Die Methode **:replacetext()** ersetzt Text in einem Edittext durch einen anderen Text.

Definition

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

Parameter

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

Mit diesen optionalen Parametern wird der Bereich definiert, der durch den angegebenen String ersetzt werden soll. Wenn kein Bereich angegeben ist, wird die Selektion, also der Bereich von *.startsel* bis *.endsel*, durch den angegebenen String ersetzt. *.startsel* und *.endsel* werden ans Ende dieses Textes gesetzt, das heißt der Cursor befindet sich nach der Ersetzung hinter dem eingefügten Text. Wird ein Bereich angegeben, behalten *.startsel* und *.endsel* ihre Werte, sofern das durch die Ersetzung nicht unmöglich wird.
Der Wertebereich von *Start* und *End* geht von *0* bis zur Anzahl der Zeichen im angezeigten Text, wobei jedes Zeichen zählt, das in eine Selektion eingeschlossen werden kann. Zeilen- und Absatzumbrüche zählen daher auch als Zeichen.
Für *End* kann -1 angegeben werden, um den restlichen Text ab der Startposition zu ersetzen.
Beim RTF-Edittext (*.options[opt_rtf] = true*) beziehen sich *Start* und *End* – analog zu *.startsel* und *.endsel* – auf Positionen im formatierten Text. Aus ihnen kann **nicht** auf Positionen im Content-String des RTF-Edittextes geschlossen werden, da dieser zusätzlich Formatierungsanweisungen enthält. Sind die Parameter *Start* oder *End* größer als die Textlänge, dann wird beim RTF-Edittext die Textlänge verwendet.

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

Mit diesem optionalen Parameter wird beim RTF-Edittext angegeben, ob der einzufügende String reinen, unformatierten Text („Plain Text“) oder RTF-Text enthält.

Wertebereich

content_plain
String enthält reinen, unformatierten Text.
content_rtf
String enthält RTF-Text.
Nur verwendbar bei *.options[opt_rtf] = true*.

string *Text* input

In diesem Parameter wird der String übergeben, der Text im RTF-Edittext ersetzen soll.

Rückgabewert

true
: Textbereich im Edittext wurde durch den angegebenen String ersetzt.

false
: Fehler: Es hat keine Ersetzung stattgefunden.

Objekte mit dieser Methode

edittext

2.35 :represent()

Diese Methode hat zweierlei Funktion.

1. Beim Aufruf ohne Parameter werden an der View-Komponente alle gekoppelten Attribute aktualisiert. Dies wird ebenso für alle Kinder und Kindeskinder ausgeführt.
   Durchgeführt wird dies, indem für jedes View-Attribut der Datenwert von der korrespondierenden Model-Komponente erfragt wird und durch Aufruf der internen und überdefinierbaren **:represent()**-Methode (mit Parametern *Value*, *Attribute*, *Index*) die Repräsentation durch das Objekt erfolgt.
   Die Kopplung zur Model-Komponente muss über die Attribute *.datamodel* und *.dataset* erfolgt sein.

   Wichtig
   Ein direkter Aufruf mit Parametern wird vom IDM nicht unterstützt und ist für zukünftige Erweiterungen vorbehalten.
2. Die überdefinierbare **:represent()**-Methode erlaubt einen Eingriff des Dialogprogrammierers vor der eigentlichen Zuweisung des Datenwertes an das Präsentationsobjekt. Dadurch können notwendige Konvertierungen oder Transformationen zwischengeschaltet werden. Die überdefinierte Methode sollte mit `pass this:super();` abgeschlossen werden um das normale Verhalten des IDM zu gewährleisten bzw. um die automatische Typkonvertierung auszunutzen.

   Hinweis
   Die überdefinierbare Methode **:set()** wird bei der Zuweisung von Datenwerten an das Präsentationsobjekt nicht aufgerufen.

Definition

```
void :represent
(
)
```

Überdefinierbare Methode

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

Parameter

anyvalue *Value* input
Dieser Parameter beinhaltet den Datenwert der präsentiert werden soll.

attribute *Attribute* input
Dieser Parameter bezeichnet das View-Attribut in dem der Datenwert repräsentiert werden soll.

anyvaluet *Index* input

Dieser Parameter bezeichnet den Indexwert für die Präsentation des Datenwertes im View-Attribut.

Beispiel

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

record Rec
{
  .dataoptions[dopt_propagate_on_start] false;
  string Name := "Hugo Walter";
  string ZIP  := "O7123";
}

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 "Country";  .width 100;  .ytop 60; }

poptext PtCountry
{
  .ytop 60;
  .xauto 0;  .xleft 100;
  .dataget[.activeitem] .ZIP;
  .dataset[.activeitem] .ZIP;
  .text[1]  "???";  .text[2]  "NRW";
  .text[3]  "BER";  .text[4]  "HH";
  .text[5]  "HRO";  .text[6]  "HAN";
  .text[7]  "NS";   .text[8]  "SH";
  .text[9]  "BW";   .text[10] "S";
  .text[11] "BAY";

  :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()

Diese überdefinierbare Methode wird bei der Synchronisation zwischen View- und Model-Komponten am Präsentationsobjekt aufgerufen um den Datenwert für ein View-Attribut zu ermitteln. Dies geschieht auch implizit bei Aufruf der Methoden **:apply()** oder **:collect()**.

Wichtig

Ein direkter Aufruf wird von IDM nicht unterstützt und ist für zukünftige Erweiterungen vorbehalten.

Das Standardverhalten der Methode ist, dass das entsprechende Attribut als Einzelwert oder aggregierter Gesamtwert zurückgeliefert wird. Dabei erfolgt für die Ermittlung der Datenwerte des Präsentationsobjekts kein Aufruf der überdefinierbaren Methode **:get()**. Es werden also immer die konkreten Attributwerte im Standardverhalten erfragt. Indizierte Attribute liefern dabei beim Zugriff auf den Gesamtwert den Datenwert als Vektor zurück (siehe auch die Funktionen **setvector()** und **getvector()**).

Durch Überdefinition dieser Methode kann das Standardverhalten umgangen bzw. ergänzt werden, um so eine Anpassung der Werte zu erlauben.

Die Kopplung zur Model-Komponente muss über die Attribute *.datamodel* und *.dataset* erfolgt sein.

Definition

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

Parameter

attribute *Attribute* input
Dieser Parameter bezeichnet das View-Attribut dessen Datenwert erfragt werden soll.

anyvalue *Index* input
Dieser Parameter bezeichnet den Indexwert, der für den Zugriff auf den Datenwert des View-Attributs am Präsentationsobjekt benutzt werden soll.

Rückgabewert

Datenwert entsprechend der Attribut- und Indexangabe.

Beispiel

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

record Rec
```

```
{
  .dataoptions[dopt_propagate_on_start] false;
  string Name := "Hugo Walter";
  string ZIP  := "O7123";
}

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 "Country";  .width 100;  .ytop 60; }
```

```
poptext PtCountry
{
  .ytop 60;
  .xauto 0;  .xleft 100;
  .dataget[.activeitem] .ZIP;
  .dataset[.activeitem] .ZIP;
  .text[1]  "???";  .text[2]  "NRW";
  .text[3]  "BER";  .text[4]  "HH";
  .text[5]  "HRO";  .text[6]  "HAN";
  .text[7]  "NS";   .text[8]  "SH";
  .text[9]  "BW";   .text[10] "S";
  .text[11] "BAY";

  :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()

Speichert das XML-Dokument in einer Datei oder unter einer URL ab.

Definition

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

Parameter

string *URL* input

Der Pfad der Datei, in die gespeichert werden soll. Anstelle eines Pfades kann auch eine URL angegeben werden.

Rückgabewert

Gibt an, ob das Speichern erfolgreich war.

Objekte mit dieser Methode

document

2.38 :select()

Bewegt den XML-Cursor in der angegebenen Richtung oder auf den ersten Knoten im DOM-Baum, der einem angegebenen Muster entspricht.

Definition

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

Parameter

anyvalue *DirectionOrPattern* input

Der Parameter darf nur von folgenden Typen sein:

enum *DirectionOrPattern* input

Gibt die Richtung an, in der der XML-Cursor innerhalb des DOM-Baums bewegt wird.

Wertebereich

select_document
: Der XML-Cursor wird auf das Dokument gesetzt. Der XML-Cursor muss noch nicht auf einen DOM-Knoten verweisen

select_first
: Der XML-Cursor wird auf den ersten DOM-Knoten gesetzt, der dem zuletzt gesuchten Muster entspricht.

select_first_child
: Der XML-Cursor wird auf den ersten direkten Kindknoten gesetzt.

select_first_sibling
: Der XML-Cursor wird auf den ersten DOM-Knoten mit demselben Vater gesetzt.

select_last
: Der XML-Cursor wird auf den letzten DOM-Knoten gesetzt, der dem zuletzt gesuchten Muster entspricht. Diese Operation sucht den betreffenden DOM-Knoten.

select_last_child
: Der XML-Cursor wird auf den letzten direkten Kindknoten gesetzt. Diese Operation sucht den betreffenden DOM-Knoten.

select_last_sibling
: Der XML-Cursor wird auf den letzten DOM-Knoten mit demselben Vater gesetzt. Diese Operation sucht den betreffenden DOM-Knoten.

select_next
: Der XML-Cursor wird auf den nächsten DOM-Knoten gesetzt, der dem zuletzt gesuchten Muster entspricht.

select_prev
: Der XML-Cursor wird auf den vorherigen DOM-Knoten gesetzt, der dem zuletzt gesuchten Muster entspricht.

select_next_sibling
Der XML-Cursor wird auf den nächsten DOM-Knoten mit demselben Vater gesetzt.
select_prev_sibling
Der XML-Cursor wird auf den vorherigen DOM-Knoten mit demselben Vater gesetzt.
select_root
Der XML-Cursor wird auf den Wurzelknoten des DOM-Baums gesetzt. Dies ist der Initialwert eines XML-Cursors. Der XML-Cursor muss noch nicht auf einen DOM-Knoten verweisen
select_up
Der XML-Cursor wird auf den Vaterknoten gesetzt.

string *DirectionOrPattern* input
Gibt ein Muster an. Der XML-Cursor wird auf den ersten Knoten im DOM-Baum gesetzt, der dem Muster entspricht. Das Muster wird gemerkt, sodass mit der Methode **:select** und *select_first*, *select_last*, *select_next* und *select_prev* der XML-Cursor auf weitere DOM-Knoten, die dem Muster entsprechen, gesetzt werden kann.
Der Aufbau des Musters ist in Kapitel „Muster für die Methoden :match() und :select()“ des Handbuchs „XML-Schnittstelle“ beschrieben.

Rückgabewert

Gibt an, ob der XML-Cursor umgesetzt werden konnte.

Im Fehlerfall steht der XML-Cursor weiterhin auf dem ursprünglichen DOM-Knoten.

Es ist zu beachten, dass ein XML-Cursor, dessen Attribut *.mapped* den Wert *false* besitzt, automatisch auf die Wurzel des DOM-Dokuments positioniert wird.

Objekte mit dieser Methode

doccursor

2.39 :select_next()

Diese Methode wird vom ***transformer***-Objekt aufgerufen, um während einer Transformation den nächsten zu besuchenden Knoten zu bestimmen. Mit dieser Methode kann also die Reihenfolge, in der die Knoten eines XML-Baums bzw einer IDM-Objekthierarchie während einer Transformation besucht werden, überdefiniert werden. Die Default-Implementierung dieser Methode definiert einen Tiefendurchlauf.

Die Methode kann überdefiniert werden (ähnlich zu **:init()**).

Definition

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

Parameter

object *Src* input

In diesem Parameter wird der aktuelle, gerade abgearbeitete Knoten übergeben. In der Default-Implementierung orientiert sich **:select_next** am Typ des *.root*-Attributs.

» *.root* ist ein String
In *Src* wird ein ***Doccursor***-Objekt erwartet, das auf den aktuellen Knoten im XML-Baum verweist. Dieser Cursor wird auf den nächsten (im Sinne der Pre-Order-Reihenfolge) Knoten im XML-Baum gesetzt und als Rückgabewert wieder zurückgeliefert.

» *.root* ist ein IDM-Objekt
In *Src* wird ein IDM-Objekt erwartet, das den aktuellen Knoten repräsentiert. Ausgehend von diesem wird der Nachfolger bestimmt.

Rückgabewert

Es wird der nächste Knoten zurückgeliefert oder *null*, falls die Transformation beendet werden muss.

Die Default-Implementierung liefert folgendes:

» *.root* ist ein String
Hier wird das ***Doccursor***-Objekt zurückgeliefert, das in *Src* übergeben und nun weitergestellt wurde.

» *.root* ist ein IDM-Objekt
Hier wird ein IDM-Objekt zurückgeliefert, das als nächstes untersucht werden soll.

Objekte mit dieser Methode

transformer

Siehe auch

Attribut root

2.40 :set()

Diese Methode kann Attribute an einem Objekt setzen. Sie ist an allen Objekten, Modellen und Defaults vordefiniert.

Darüber hinaus ist es möglich, diese Methode zu überschreiben und somit die Art und Weise wie Attribute gesetzt werden um weitere Aktionen (z.B. Konsistenzprüfungen) zu erweitern.

Die dabei zulässigen Attribute für den jeweiligen Objekttyp entnehmen Sie bitte der „Objektreferenz“.

Definition

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

Parameter

attribute *Attribute* input

In diesem Parameter wird der Methode mitgeteilt, welches Attribut gesetzt werden soll.

anyvalue *Value* input

In diesem Parameter wird der gewünschte, neue Wert übergeben, den das Attribut annehmen soll.

*anyvalue **Index** input*

Dieser optionale Parameter muss übergeben werden, wenn ein indiziertes Attribut gesetzt werden soll. Sein Datentyp muss dem Index-Datentyp des Attributs entsprechen. Soll also ein eindimensionales Attribut gesetzt werden, so wird hier ein *integer*-Wert erwartet. Wenn es sich hingegen um ein zweidimensionales Attribut handelt, so muss hier ein *index*-Wert übergeben werden.

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

Über diesen optionalen Parameter wird gesteuert, ob durch das erfolgreiche Setzen des Attributes ein *changed*-Ereignis verschickt werden soll oder nicht. Wenn kein Ereignis verschickt werden soll, muss hier *false* übergeben werden. Der Standardwert ist *true*, sodass ein Ereignis verschickt wird, auch wenn hier nichts angegeben wird.

Rückgabewert

true

Die Methode liefert *true* zurück, wenn das Setzen des Attributs erfolgreich war.

false

Wenn das Setzen des Attributs abgelehnt wurde (entweder vom IDM oder IDM-Programmierer), wird *false* zurückgeliefert.

Impliziter Aufruf

Die Methode **:set()** wird auch implizit aufgerufen. Bei Veränderung eines Attributs (vordefiniert oder benutzerdefiniert) aus der Regelsprache (durch Zuweisung mit `:=` bzw. `::=` oder die eingebaute Funktion **setvalue()**) oder mit einer Schnittstellenfunktion (z.B. **DM_SetValue()**), wird die Methode **:set()** aufgerufen, die dann das eigentliche Setzen des Attributs durchführt. Der Aufruf der Methode **:set()** erfolgt dabei „in der Mitte“ und führt die eigentliche Zuweisung des neuen Werts durch.

Redefinition

Zur Redefinition der **:set()**-Methode genügt es, einfach an einem Objekt folgendes zu schreiben:

```
window W {
  :set(<keine Parameter!>)
  {
    ...
    Aktionen, um ein Attribut zu setzen.
    ...
  }
}
```

Innerhalb der Methodendefinition können die Parameter *Attribute*, *Value*, *Index* und *SendEvent* abgefragt und gesetzt werden, um dann z.B. mit *this:super()* die Methode **:set()** der Superklasse mit den entsprechend veränderten Parametern aufzurufen. Die Änderung der Parameter wie *Attribute* und *Index* ist allerdings nicht sinnvoll, da solcher Code schwer zu verstehen wäre.

Unterdrückung von rekursiven Aufrufen

Wird in der Methode **:set()** irgendein Attribut des selben Objektes, für das die Methode **:set()** bereits aufgerufen worden ist, gesetzt, so müsste eigentlich wieder die Methode **:set()** für das selbe Objekt implizit aufgerufen werden. Da dabei die Gefahr von Endlosrekursionen sehr groß ist, wird ein solcher rekursiver Aufruf unterdrückt.

Beispiel

```
window {
  :set()
  {
    case (Attribute)
    in .title:
      this.title := "<" + Value + ">";
        // Kein impliziter Aufruf von :set().
        // Wegen ":=" wird ein "changed"-Ereignis verschickt.
        // Mit "::=" könnte der IDM-Programmierer auch dies unterdrücken.
      OtherObject.title := "<" + Value + ">";
        // :set() von OtherObject wird aufgerufen.
        // Der SendEvent-Parameter von :set() wird auf "true" gesetzt.
    otherwise:
      this:super();
```

```
    endcase
  }
}
```

In diesem Beispiel wird innerhalb der Methode **:set()** das Setzen des Fenstertitel überwacht und der Titel in „<>“ eingeklammert. Während der Zuweisung *this.title :=* … wird die Methode **:set()** nicht nochmals aufgerufen. Die Methode **:set()** für ein anderes Objekt wird sehr wohl aufgerufen.

Aufruf der Methode :set() an der übergeordneten Klasse

Das obige Beispiel zeigt auch den Umgang mit *this:super()*. Alle Attribute, für die sich die überschriebene Methode **:set()** nicht zuständig fühlt, sollten an die Superklasse delegiert werden, so dass irgendwann die vordefinierte Methode **:set()** das Setzen dieser Attribute durchführt.

Der Wert des Parameters *Value* wird auf jeden Fall berücksichtigt und dem Attribut zum Schluss zugewiesen. Das obige Beispiel könnte deswegen auch so geschrieben werden:

Beispiel

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

Erzwingen von rekursiven Aufrufen

Soll die Methode **:set()** für das selbe Objekt nochmals mit den veränderten Werten aufgerufen werden, so kann mit der Methode **:recall()** erzwungen werden, dass der Aufruf der als Parameter übergebenen **:set()**-Methode auch rekursiv stattfindet. Es ist jedoch Vorsicht geboten, da es auf diese Weise sehr leicht zu Endlosrekusionen kommen kann.

Beispiel

```
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  // Konsistenzprüfung.
        Value := this.MaxCount;      // Value korrigieren.
      endif;
      RetVal := this:super();
    in .MaxCount:
      if this.MaxCount > Value then
        this.MaxCount := Value;
        this:recall(:set, .Count, Value);
          // .Count auf Value setzen, wobei :set() nochmals aufgerufen wird.
        RetVal := true;
      else
        RetVal := false;             // Setzen ablehnen.
      endif
    otherwise:
      RetVal := this:super();
    endcase
    return RetVal;
  }
}

MRec Rec {
}

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

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

Siehe auch

Eingebaute Funktion setvalue() im Handbuch „Regelsprache“

C-Funktion DM_SetValue im Handbuch „C-Schnittstelle - Funktionen“

2.41 :setclip()

Werte, die bei einer **Drag&Drop**-Operation von der Quelle zum Ziel weitergereicht werden, ergeben sich aus den Typen, die an der ***source***-Ressource des Quell-Objektes angegeben sind und direkt aus dem Text-Inhalt des Objektes.

Durch **Überschreiben** der vordefinierten Methode **:setclip()** am Quell-Objekt können in der Regelsprache die zu übergebenden Werte beliebig gesetzt werden.

Aufruf der Methode

Die Methode **:setclip()** wird bei einer Drag&Drop-Operation beim ersten Drücken und Bewegen der Maustaste implizit und sofort für das Quell-Objekt aufgerufen.

Der explizite Aufruf von **:setclip()** in einer Regel ist nicht erlaubt.

Definition

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

Parameter

object *Clipboard* input

Im Parameter *Clipboard* werden die Datenwerte gesetzt, die vom Quell- zum Ziel-Objekt verschoben werden. Der Parameter verweist auf ein Objekt der Klasse ***clipboard***, in dessen über type-Enums indizierten Attribut *.value[enum]* die Datenwerte gelesen und gesetzt werden können. Erlaubt sind allerdings nur Typen, die in der vom Quell-Objekt verwendeten ***source***-Ressource ausgewiesen sind. Wertzuweisungen auf *.value[enum]* mit anderen Typen als Index erzeugen einen Fehler.

Aufruf der Methode :setclip() an der übergeordneten Klasse

In einer überschriebenen **:setclip()**-Methode kann durch den Aufruf von *this:super()* die **:setclip()**-Methode des Modells aufgerufen bzw., wenn keine weitere Modell-Methode vorhanden ist, auf das Standardverhalten zugegriffen werden.

Beispiel

```
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-Methode fuer die Objektklasse.
    Clipboard.value[type_file] := "FILE: " +  Clipboard.value[type_file];
  }
}

window Wi
{
  child MPb PbSource
  {
    .text "Source";
    :setclip()
    {
      // Ueberschreiben von :setclip() durch eine benutzerdefinierte Methode.
      // Ein Parameter mit dem Namen Clipboard enthaelt die ID des clipboard-
      // Objektes, dem man die Werte im .value[]-Attribut zuweist.
      this:super();  // :setclip()-Methode des Modells aufrufen.

      // Setzen eines neuen Wertes fuer den Text-Typ:
      Clipboard.value[type_text] := "MY TEXT DATA";
    }
  }
  on close  { exit(); }
}
```

2.42 :setformat()

Mit der Methode **:setformat()** können Textbereiche in einem RTF-Edittext formatiert werden.

Definition

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

Parameter

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

Mit diesen optionalen Parametern wird der Bereich definiert, der formatiert werden soll. Fehlen diese Angaben, wird der Bereich von *.startsel* bis *.endsel* (Selektion bzw. Cursorposition) formatiert.
Welcher Bereich tatsächlich formatiert wird, hängt von der Art der Formatierung ab:

- Zeichenformatierungen wirken sich genau auf den angegebenen Bereich aus.
- Absatzformatierungen wirken sich auf alle Absätze aus, die ganz oder teilweise im angegebenen Bereich enthalten sind.

Die Formatierungsart ist in der nachfolgenden Tabelle mit angegeben.
Der Wertebereich von *Start* und *End* geht von *0* bis zur Anzahl der Zeichen im angezeigten Text, wobei jedes Zeichen zählt, das in eine Selektion eingeschlossen werden kann. Zeilenumbruch-Zeichen zählen daher mit.
Für *End* kann -1 angegeben werden, um den restlichen Text ab der Startposition zu formatieren.
Start und *End* beziehen sich – analog zu *.startsel* und *.endsel* – auf Positionen im formatierten Text. Aus ihnen kann **nicht** auf Positionen im Content-String des RTF-Edittextes geschlossen werden, da dieser zusätzlich Formatierungsanweisungen enthält. Sind die Parameter *Start* oder *End* größer als die Textlänge, dann wird beim RTF-Edittext die Textlänge verwendet.

enum *Type* input

Dieser Parameter definiert den Typ der Formatierung, die angewendet werden soll. Die *enum*-Werte für die Formatierungstyen sind in der nachfolgenden Tabelle aufgelistet.

anyvalue *Value* input

In diesem Parameter wird ein Wert übergeben, der die Ausprägung des in *Type* definierten Formatierungstyps festlegt. Die Datentypen für die Formatierungstypen sind in der nachfolgenden Tabelle mit angegeben.

Rückgabewert

true
: Der Textbereich wurde formatiert.

false
: Fehler: Die Formatierung konnte nicht angewendet werden (z.B. wegen fehlerhafter Argumente).

Formatierungsarten, *enum*-Werte und Datentypen der Formatierungstypen

Formatierungstyp	Formatierungsart	*enum*-Wert	Datentyp
Schriftart, Font	Zeichenformat	*text_font*	*string*, ***font***
Schriftgröße	Zeichenformat	*text_size*	*integer*
Vordergrundfarbe, Textfarbe	Zeichenformat	*text_fgc*	*integer*, ***color***
Hintergrundfarbe	Zeichenformat	*text_bgc*	*integer*, ***color***
fett	Zeichenformat	*text_bold*	*boolean*
kursiv	Zeichenformat	*text_italic*	*boolean*
unterstrichen	Zeichenformat	*text_underline*	*boolean*
Einrückung links	Absatzformat	*text_indent_left*	*integer*
Einrückung rechts	Absatzformat	*text_indent_right*	*integer*
Einrückung der Folgezeilen in einem Absatz	Absatzformat	*text_indent_offset*	*integer*
Textausrichtung	Absatzformat	*text_align*	*enum* [*align_left*, *align_right*, *align_center*, *align_justify*]

Je nach Schriftart kann es vorkommen, dass Formatierungsanweisungen bzw. mit **:setformat()** gesetzte Formatierungsattribute vom Windows-Objekt, das der IDM für den RTF-Edittext verwendet, nicht beachtet werden. Unter Windows 7 ist dies zum Beispiel bei *text_bold* so, wenn explizit keine Schriftart gesetzt und daher implizit die Schriftart „System“ verwendet wird.

Maßeinheit für Größen und Positionen

Größen und Positionen werden – wie bei der Windows-Programmierung üblich – in „twips“ angegeben. Ein twip entspricht *1/20 point* (Punkt), das ist *1/1440 Zoll* (inch) bzw. *1/567 cm*.

Erläuterungen zu einzelnen Formatierungen

- Der Wert für *text_indent_right* ist eine absolute Angabe der Einrückung vom rechten Rand.
- Dagegen ist *text_indent_left* eine relative Angabe, die mehrfach angewendet werden kann. Die Einrückungen vom linken Rand werden akkumuliert, können insgesamt aber nie über den Rand hinausgehen. Der Wert kann negativ sein und bedeutet dann eine Verschiebung eines Absatzes nach links, während positive Werte eine Verschiebung nach rechts bedeuten.
- Der Wert von *text_indent_offset* gibt an, wie weit alle Zeilen eines Absatzes außer der ersten gegenüber der ersten Zeile nach rechts (positiver Wert) oder nach links (negativer Wert) verschoben sind.
- Bei *text_font* kann
  - ein String mit einem Fontnamen,
  - ein String mit Größe und Fontnamen in der Form <*size*>.<*font name*>, analog zu ***font***-Ressourcen unter MICROSOFT WINDOWS, oder
  - eine ***font***-Ressource

  angegeben werden. **:setformat()** extrahiert aus einer ***font***-Ressource Informationen wie Fontname und Größe und wendet die entsprechenden Formatierungen an.
  Wenn der Fontname korrekt ist, wird die Formatierung durchgeführt, unabhängig davon, ob der Font auf dem System installiert ist. Wenn der Font nicht installiert ist, wird man allerdings keine Auswirkung sehen. Man würde sie aber sehen, sobald der RTF-Text auf einem System angezeigt wird, auf dem der Font vorhanden ist.
  Angaben in der Form <*size*>.<*font name*> werden ab IDM-Version A.05.02.I unterstützt. Dabei werden Größe und Fontname ausgewertet und gesetzt.
- Vordergrundfarbe (*text_fgc*) und Hintergrundfarbe (*text_bgc*) können als *integer*-Wert, der als RGB-Wert interpretiert wird, oder als ***color***-Ressource übergeben werden.

Objekte mit dieser Methode

edittext mit *.options[opt_rtf] = true*

2.43 :setinherit()

Diese Methode kann Attribute von Objekten auf den Wert der entsprechenden Modelle oder des entsprechenden Defaults zurücksetzen. Die Methode ist an allen Objekten, Modellen und Defaults vordefiniert.

Darüber hinaus ist es möglich diese Methode zu überschreiben und somit die Art und Weise des Zurücksetzens von Attributen zu beeinflussen.

Definition

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

Parameter

attribute *Attribute* input

In diesem Parameter wird der Methode mitgeteilt, welches Attribut zurückgesetzt werden soll.

*anyvalue **Index** input*

Dieser optionale Parameter muss gesetzt sein, wenn ein indiziertes Attribut zurückgesetzt werden soll. Sein Datentyp muss dem Index-Datentyp des Attributs entsprechen. Soll ein eindimensionales Attribut zurückgesetzt werden, so wird hier ein *integer*-Wert erwartet. Wenn es sich hingegen um ein zweidimensionales Attribut handelt, so muss hier ein *index*-Wert übergeben werden.

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

Über diesen optionalen Parameter wird gesteuert, ob durch das erfolgreiche Zurücksetzen des Attributs ein *changed*-Ereignis verschickt werden soll oder nicht. Wenn kein Ereignis verschickt werden soll, muss hier *false* übergeben werden. Der Standardwert ist *true*, so dass ein Ereignis verschickt wird, wenn hier nichts angegeben wird.

Rückgabewert

Die Methode gibt *true* zurück, falls das Zurücksetzen des Attributs erfolgreich war, ansonsten *false*.

Impliziter Aufruf

Die Methode **:setinherit()** wird auch implizit aufgerufen. Immer wenn ein Attribut (vordefiniert oder benutzerdefiniert) aus der Regelsprache oder mit einer Schnittstellenfunktion (z.B. **DM_ResetValue ()**) zurückgesetzt wird, wird die Methode **:setinherit()** aufgerufen. Diese muss dann den Wert des Attributes zurücksetzen.

Redefinition

Zur Redefinition der Methode genügt es, einfach an einem Objekt folgendes zu schreiben:

```
window W {
  :setinherit(<keine Parameter!>)
  {
    ...
    Aktionen, um den Wert des Attributs zurueckzusetzen.
    ...
    pass this:super();
    oder
    return <boolean-Wert>;
  }
}
```

Innerhalb der Methodendefinition können die Parameter *Attribute*, *Index* und *SendEvent* abgefragt und gesetzt werden, um dann z.B. mit *this:super()* die **:setinherit()**-Methode der Superklasse mit den entsprechend veränderten Parametern aufzurufen, was allerdings in der Regel nicht sinnvoll ist.

Unterdrückung von rekursiven Aufrufen

Wird innerhalb der Methode **:setinherit()** irgendein Attribut des selben Objektes zurückgesetzt, für das die Methode **:setinherit()** bereits aufgerufen worden ist, so müsste eigentlich wieder die Methode **:setinherit()** für das selbe Objekt implizit aufgerufen werden. Da dabei die Gefahr von Endlosrekursionen sehr groß ist, wird ein solcher rekursiver Aufruf unterdrückt.

Beispiel

```
window {
  :setinherit()
  {
    case (Attribute)
      in .width:
        setinherit(this, .height);  // kein impliziter Aufruf von :setinherit
()
        pass this:super();
      otherwise:
        pass this:super();
    endcase
  }
}
```

In diesem Beispiel wird in der Methode **:setinherit()** ein impliziter Aufruf der Methode **:setinherit()** für *this.height* unterdrückt. Der Wert wird jedoch sehr wohl zurückgesetzt.

Aufruf der Methode :setinherit() an der übergeordneten Klasse

Das obige Beispiel zeigt auch den Umgang mit *this:super()*. Alle Attribute, für die sich die überschriebene Methode **:setinherit()** nicht zuständig fühlt, sollten an die Superklasse delegiert werden, so dass irgendwann die vordefinierte Methode **:setinherit()** den Attributwert zurücksetzt.

Wenn die überdefinierte **:setinherit()**-Methode ein Attribut des Objektes selbst zurücksetzt (etwa mit *this.Attr :*= …), dann wird das Inherited-Bit nicht ebenfalls zurückgesetzt; d.h. die eingebaute Funktion **inherited()** wird in diesem Fall *false* zurückgeben.

Beispiel

```
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;                 // Soll X = 2
  print "inherited?= " + inherited();  // Soll false
  exit();
}
```

Siehe auch

Eingebaute Funktion setinherit() im Handbuch „Regelsprache“

2.44 :super()

Bei Arbeiten mit Methoden kommt es häufig vor, dass am Default oder an relativ weit oben stehenden Modellen allgemeine Methoden definiert sind. An davon abgeleiteten Objekten werden diese Methoden durch eine Neudefinition überschrieben. Um eine Codeverdoppelung zu vermeiden, können die Methoden an oben stehenden Modellen auch mitbenutzt d.h. von hierarchisch tieferliegenden Methoden aufgerufen werden. Dieses muss explizit durch den Aufruf der Methode super erfolgen.

Definition

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

Parameter

Alle Parameter, die die aufzurufende Methode benötigt.

Objekte mit dieser Methode

Alle Objekte, die Methoden haben können.

Beispiel

Dieses Beispiel zeigt, wie nach dem Schließen eines Fensters, bestimmte Attribute wieder zurückgesetzt werden. Die Methoden stehen direkt an dem Objekt, wo die Attribute definiert wurden. Durch den Aufruf der Methode :super ist die Reihenfolge des Ablaufs der Methoden in der Modellhierarchie frei definierbar. this:super() sucht immer die nächste Methode in der Modellhierarchie des aktuellen Objektes.

```
dialog Super

model window MWin
{
  integer Status := 0;
  rule void CleanAfterClose(integer S)
  {
    !! Das Attribut zurücksetzen.
    this.Status := S;
  }
  !! Beide Regeln reagieren darauf, wenn das Fenster unsichtbar wird
  !! und stoßen dann zentral die "Aufräum-Regel" an.
  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()
  {
    !! Nur die Attribute zurücksetzen, die hier definiert wurden.
    this.MainStatus := 0;
    !! Den Rest aufräumen.
    this:super(0);
  }
}
MMainWin MainWin
{
}
```

2.45 :transform()

Transformiert das Objekt mit dem angegebenem Schema. Wenn das Ziel ein XML-Dokument ist, wird der gespeicherte DOM-Baum gelöscht und ein neuer DOM-Baum aufgebaut. Alle bestehenden XML-Cursor werden ungültig.Bei einem ungültigen XML-Cursor besitzt das Attribut *.mapped* den Wert *false*.

Alternativ kann das Ziel der Transformation auch ein Text oder eine Datei sein. Ist das Resultat der Wandlung kein legales XML-Format, dann muss direkt in einen Text oder eine Datei gewandelt werden, da das Resultat nicht einem XML-Dokument zugewiesen werden kann. Dies ist zum Beispiel der Fall, wenn zu HTML gewandelt wird.

Definition

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

Parameter

object *Schema* input

Ist das XSL-Transformation-Dokument, mit dem transformiert wird. Der Parameter muss ein XML-Dokument sein.

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

Optionaler Parameter, ist das Ziel der Transformation. Ist der Parameter weggelassen, dann wird das *this*-Objekt als Ziel genommen. Der Parameter kann von folgenden Typen sein:

*object **Target** input*

Der Parameter ist Eingabeparameter und muss ein XML-Dokument darstellen. Der gespeicherte DOM-Baum wird gelöscht und ein neuer DOM-Baum gemäß der Transformation aufgebaut.

*string **Target** input output*

Abhängig vom *Type*-Parameter ist der Parameter ein Eingabeparameter und gibt einen Dateinamen an oder es ist ein Ausgabeparameter und erhält den String, der durch die Transformation entstanden ist.

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

Optionaler Parameter, gibt die Art der Wandlung an. Mögliche Werte sind:

type_file

Gibt an, dass der *Target*-Parameter eine Datei bezeichnet.

type_object

Gibt an, dass der *Target*-Parameter ein XML-Dokument ist. Dieser Wert wird standardmäßig angenommen, wenn der *Target*-Parameter vom Typ *object* ist.

type_text

Gibt an, dass der *Target*-Parameter ein Ausgabeparameter vom Typ *string* ist. Dieser Wert wird standardmäßig angenommen, wenn der *Target*-Parameter vom Typ *string* ist.

Rückgabewert

Gibt an, ob die Transformation erfolgreich war.

Im Fehlerfall bleibt der DOM-Baum des *Target*-XML-Dokuments erhalten oder wird gelöscht, eine teilweise Transformation gibt es nicht.

Objekte mit dieser Methode

- doccursor
- document

2.46 :unuse()

Mit dieser Methode wird an einem ***Dialog***- oder ***Module***-Objekt ein über „use“ eingebundenes Modul wieder entfernt bzw. entladen. Falls es nicht anderweitig verwendet wird, so wird es komplett entladen (mitsamt der Entfernung aller Instanzen).

Definition

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

Parameter

anyvalue *UsepathOrModule* input

Mit Hilfe dieses Parameters wird als Use-Pfad (in Form eines Bezeichner-Pfades) oder direkt als Module-ID das Modul angeben.

Rückgabewert

false

Modul ist gar nicht bzw. nicht durch „use“ importiert.

true

Der Zugriff auf das Modul per „use“ wurde entfernt.

Objekte mit dieser Methode

- dialog
- module

Beispiel

```
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
    }
  }
}
```

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Methode :use()

Kapitel „Der alternative Import-Mechanismus“ und „Sprachbeschreibung und Use-Pfad“ im Handbuch „Programmiertechniken“

2.47 :use()

Mit dieser Methode wird an einem ***Dialog***- oder ***Module***-Objekt ein neues Modul über seinen Use-Pfad importiert. Man kann damit das Modul analog zu einer statischen **use**-Anweisung einbinden, sodass das Modul im Dialog nur einmal geladen wird.

Ist der *Check*-Parameter beim Aufruf auf *true* gesetzt, so wird nur geprüft ob schon ein Import mit „use“ an diesem Objekt vorhanden ist.

Definition

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

Parameter

string *Usepath* input

Mit Hilfe dieses Parameters wird als Use-Pfad (in Form eines Bezeichner-Pfades) das Modul angeben.

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

Wenn dieser optionale Parameter *true* ist, wird der „use“ beim Anlegen zusätzlich als exportiert gekennzeichnet (Standardwert *false*).
Der Parameter ist nur von Bedeutung wenn *Check* = *false* ist.

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

Ist dieser optionale Parameter *true*, so wird nur geprüft, ob das per Use-Pfad angegebene Modul schon mit „use“ importiert wurde (Standardwert *false*).

Rückgabewert

null

Bei *Check* = *false*:
Modul konnte nicht gefunden oder geladen werden.
Bei *Check* = *true*:
Kein „use“ für diesen Use-Pfad vorhanden.

<Id>

Id des importierten Moduls.

Objekte mit dieser Methode

- dialog
- module

Beispiel

```
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
    }
  }
}
```

Verfügbarkeit

Ab IDM-Version A.06.02.g

Siehe auch

Methode :unuse()

Kapitel „Der alternative Import-Mechanismus“ und „Sprachbeschreibung und Use-Pfad“ im Handbuch „Programmiertechniken“

2.48 :validate()

Überprüft das XML-Dokument, ob es dem im DOM-Baum angegebenen Dokumenttyp entspricht. Ist kein Dokumenttyp enthalten, wird ein Fehler gemeldet.

Definition

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

Parameter

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

Optionaler Ausgabeparameter, wenn er angegeben ist, wird der Parameter mit "" initialisiert. Im Fehlerfall wird dem Parameter die Systemfehlermeldung zugewiesen. Es ist jedoch sicherer den Rückgabewert abzufragen.

Rückgabewert

0

Erfolgreiche Validierung; das XML-Dokument entspricht dem Dokumententyp.

DOM-Fehlercode

Das XML-Dokument entspricht **nicht** dem Dokumententyp.

-1

Interner Fehler. *ErrorMsg* ist **nicht** initialisert.

Objekte mit dieser Methode

document

Index

S

T

U

V

Z