Function Reference

JACAMAR

version 5

KATLA GmbH
Immermannstraße 28
D - 39108 Magdeburg

Document Reference No.: Ka_000167
Version 1.4
Issue Date: 2019-01-09

TABLE OF CONTENTS

1 Introduction.....5
2 Functions in JACAMAR.....6
2.1 Element Functions.....6
2.1.1 pk().....6
2.1.2 this().....6
2.2 List Functions.....7
2.2.1 empty().....7
2.2.2 has().....7
2.2.3 size().....7
2.2.4 first().....8
Syntax: first(type).....8
2.2.5 last().....8
2.2.6 Range Operation.....9
2.3 Sorting Functions.....10
2.3.1 ascending().....10
2.3.2 descending().....10
2.3.3 sort().....10
2.4 Grouping & Pivot Functions.....11
2.4.1 groupedBy().....11
2.4.2 members().....12
2.4.3 single().....12
2.4.4 getMultiple().....12
2.4.5 getSiblings().....12
2.4.6 allElements().....13
2.5 Cascade Functions.....13
2.5.1 lineElement().....13
2.5.2 lineAbove().....14
2.5.3 lineLevelBelow().....14
2.5.4 level().....14
2.5.5 getParentChain().....15
2.5.6 CheckParents().....15
2.5.7 hasChildren().....15
2.5.8 markSuccess().....16
2.5.9 showChildren().....16
2.6 Multi-Occurrence Functions.....18
2.6.1 cPath().....18
2.7 Parametric Functions.....19
2.7.1 filter().....19
2.7.2 condition().....19
2.7.3 input().....20
2.8 Colour Functions.....20
2.8.1 color().....20
2.8.2 textColor().....21
2.9 Conditional Functions.....22
2.9.1 if().....22
2.9.2 map().....22
2.10 Boolean Operations and Functions.....23
2.10.1 & (AND) operator.....23
2.10.2 | (OR) operator.....23
2.10.3 Equations.....24
2.10.4 not().....24

2.11 Numerical Operations and Functions....25
2.11.1 + (add) and - (subtract) operators....25
2.11.2 / (divide by) and * (multiply by) operators....25
2.11.3 Average()....26
2.11.4 min() and max()....26
2.11.5 Sum()....26
2.11.6 square (), cube() and sqrt()....27
2.11.7 mod()....27
2.11.8 divisible()....28
2.11.9 random()....28
2.11.10 Round()....28
2.11.11 StandardDeviation()....28
2.11.12 variance()....29
2.12 Date functions....29
2.12.1 Date()....29
2.12.2 day()....29
2.12.3 Month()....30
2.12.4 Week()....30
2.12.5 Year()....30
2.12.6 Today()....30
2.13 Text functions....32
2.13.1 Substring()....32
2.13.2 Pattern()....32
2.14 Conversion-functions....33
2.14.1 toBoolean()....33
2.14.2 toNumber()....33
2.14.3 toString()....33
2.14.4 toDate()....33
2.14.5 toMonth()....33
2.14.6 toYear()....34
3 Appendix....35
3.1 Regular Expression Syntax....35

1 Introduction

There are many functions and operations that are available for the evaluation process in delivering specific results in views and reports.

Some functions provide a list of elements; others provide numeric, **Boolean** or date related values.

This section is designed to explain and illustrate the purpose and special behaviour of the functions so as to optimise the information that is displayed to users.

These functions will be an essential Item of many working views and tables.

Knowledge of the repository Meta Model design and of structuring **navigation rules** in lines and columns of a view is essential to understand and implement the functions and operations effectively.

A navigation rule is the path description of how to 'navigate" around the model elements, like a spider's web, from one base element to other related elements and collect property values from those elements.

Navigation rules can be **absolute** paths and directly from a Type as configured in line rules e.g. `Wine[filter statement]` or `Wine.item` where the lines displayed in the view come directly from a **Base Type**.

Navigation paths can also be **relative** paths that use base elements then via relations navigate to related elements. Relative paths are configured in column rules, **variables** and when importing related data with Import Manager e.g. on a Item Type Line: column definition: `size(order)` where the number of orders relative to the Item is returned.

Most functions can be used in line and column rules in a view and the results of functions are based on evaluating the **input elements** i.e. the elements of the Type that flow into the function: e.g. `Order.ascending(parameter)` evaluates the Order elements in the function. Input elements are generally accepted as the **navigation segment** before the function.

When functions are used inside a filter statement [] each evaluation, or operation, must return a Boolean (true or false) result:
e.g. Statement: This element has a related element. Evaluation is true or false and only results that are true will be returned and therefore the line rule would be: `Type[has(relation)].`

Navigation rule segments are separated by a "`.`" dot and the final segment in a navigation path could be a function or resulting property values.

Functions can also be used with **Vector** behaviour in relative paths for when you have multiple input elements and parameters. This returns multiple output results for each input element based on the parameters.

If there are elements with **to-many relations** and multiple input elements are evaluated, a result is returned for each input element e.g. `Wine` Type Line: column definition:
`wine.average(selling price)` which calculates the average of the wine's own quantities and returns a list for each wine that exists in the database.

2 Functions in JACAMAR

2.1 Element Functions

2.1.1 pk()

Appearance: Column-rules

Syntax: **`pk()`**

JACAMAR internally records each element with a unique ID based on the number sequence the elements in each Type are created.

The **`pk()`** function can be used to show the ID number for each element in a table or structure, or a list of ID numbers where there are multiple related elements.

The **`pk()`** function returns the ID of the resulting element(s) and generally doesn't require a parameter. The line rules in a view determine the Type elements displayed, therefore this function knows the incoming Type element it is referencing.

This function is used in column definitions and provides results based on the internal unique elementID number that is stored in the raw tables of each Type e.g. Item[0] is the first element in the Meta Model Type 'Item" and the pk() is therefore 0.

If on a Item line in a view you wanted to display the Item or Order IDs the column syntax could be: **`pk(Item)`** or **`pk(order)`** and would show the corresponding IDs for the related elements the same as **`Item.pk()`** or **`order.pk()`** would produce.

2.1.2 this()

Appearance: Column-rules

Syntax: **`this()`**

This function returns the current input elements of the path evaluation chain.

This function refers to intermediate elements in the path (which may be the line base type if the path has only one segment).

2.2 List Functions

These functions are used in line and column rules to filter out and only display a list of elements as defined by the function and its parameters. List functions are applicable whenever multiple input elements were returned by a Base Type or previous calculation step (line rules as well as relations in a column rule or other context).

Some of these functions however are generally only used in column rules as they depend on the line rules having been defined first before they can be applied. Details of where these functions can be used are specified in the descriptions, where necessary.

2.2.1 empty()

Appearance: line-rules

Syntax: `empty(type)`

This function returns `true` if the list of input elements is empty.

Syntax1: `order.empty()` or `empty(order)` – returns true if the Item line element has no order, or false if it has.

Syntax2: `Wine[empty(item Order) = false]` – returns true or false for each Wine element that has an order

2.2.2 has()

Appearance: line-rules

Syntax: `has(type)`

This function returns true if the list of input elements is not empty and is the reverse of the empty() function.

Syntax1: `order.has()` or `has(order)` – returns true if the Item line element has an order, or false if not.

Vector syntax when the input list has multiple elements e.g. Wine line element

Syntax2: `Items.order.has()` or `Items.has(order)` – returns true or false for each Item element of the Wine.

Example: `Items[has(children)]` followed in the next line rule by `Items[empty(children)]`.

This would show each structure with elements that have children listed first followed by elements that have no children in the structure sequence.

2.2.3 size()

Appearance: Column-rules

Syntax: **`size(type)`**

This function counts and delivers the number of elements in a list and especially useful for the number of related elements of a to-many relation.

This function is generally used in column definitions or within variables because this function does not deliver but counts a list of elements.

On a Item line rule in a view where there is only one incoming element: **`size(order)`** and **`order.size()`** would return the same value: the amount of orders.

If however on an Wine line rule whereby there is a 'to-many" relation to **`Item Order`** the different syntax sequence offers varying results:

Syntax1: **`Wine.Item Order.size()`** This expression calculates the total of all orders for all the Items of the Wine and the result is one number.

Syntax2: **`Wine.size(Item order)`** This vector syntax expression calculates the number of orders for each Item and returns a list for the Wine for each Item.

The **`size()`** function references the Type or related Type of the input elements and therefore ending properties are not required when using this function: Item Order.size(invoice.qty) would ignore the property and still return the number of invoices.

Example: Test if – at least one – invoice exist: **`if(item Order.invoice.size() > 0, "X")`**

2.2.4 first()

Appearance: Column-rules

Syntax: **`first(type)`**

This function returns the first element of the input list of related elements.

Relations are created based on the queuing principle, 'first in, first out" therefore the **`first()`** function returns the oldest relation e.g. **`Item.first(order)`** would return the oldest order for a Item, the first in the list of related orders for the Item.

For both the **`first()`** and **`last()`** functions the listing is based on the sequence the relations were created on the elements. These functions are most commonly used in column definitions where multiple (to-many) relations exist and they are usually used with a resulting property to show actual values.

The first element can also be found using the range filter "`[0]`" e.g. **`Item.order[0]`**.

2.2.5 last()

Appearance: Column-rules

Syntax: **`last(type)`**

This function returns the last element of the input list of related elements.

Relations are created based on the queuing principle, 'last in, last out" therefore the **`last()`** function returns the most recent relation e.g. **`Item.last(order)`** would return the most recent order for a Item, the last in the list of related orders for the Item.

For both the **`first()`** and **`last()`** functions the listing is based on the sequence the relations were created on the elements. These functions are most commonly used in column definitions where multiple (to-many) relations exist and they are usually used with a resulting property to show actual values.

2.2.6 Range Operation

This operation returns a reduced list from the incoming element list, whereby the borders are defined by the first and optional second parameter.

The counting of a range operation starts at 0 therefore **`[0]`** would display the first element in the list.

Here are some possibilities for syntax:

[3]	Filters the 4th element in the list
[2..5]	Filters from the 3rd up to and including the 6th element
[2..]	Filters all elements starting with the 3rd
[..4]	Filters all elements up to and including the 5th, the same as [0..4]

2.3 Sorting Functions

2.3.1 ascending()

Appearance: line-rules

Syntax: **`ascending(statement1, statement2, .. statementN)`**

This function returns the elements in ascending order based on the parameter Type natural order e.g. numerical order: 0-9 or alphabetical order: a-z.

Parameters are optional and if not specified the element **`pk(ID)`** is the default natural order for Type elements e.g. **`invoice.ascending()`**.

Multiple parameters can be used in the case where there are equal results in a previous parameter e.g. **`invoice.ascending(number,paid)`** returns 1001,1002,1003,.. in the list ordered by paid (Y,N).

2.3.2 descending()

Appearance: line-rules

Syntax: **`descending(statement1, statement2, .. statementN)`**

This function is the reverse of ascending() and returns the elements in descending natural order e.g. alphabetical order: z-a or numerical order: 9-0.

Parameters are optional and if not specified the element **`pk(ID)`** is the default natural order for Type elements e.g. **`invoice.descending()`**.

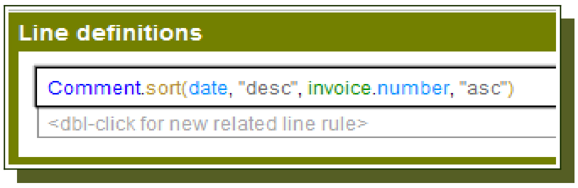
Multiple parameters can be used in the case where there are equal results in a previous parameter e.g. **`Account.descending(Surname,Firstname)`** returns *Müller, Thomas* before *Müller, Robert* in the list.

2.3.3 sort()

Appearance: line-rules

Syntax: **`sort()`**

This function can be used to combine the **`ascending()`** and **`descending()`** functionality when returning a list of elements in line rules for a view.



```
Comment.sort(date, "desc", invoice.number, "asc")
<dbl-click for new related line rule>
```

If no parameters are entered the **`sort()`** function operates by default exactly like the **`ascending()`** function, however by using the 'desc" and 'asc" parameters you can switch the sequence of ascending and descending sorting.

2.4 Grouping & Pivot Functions

2.4.1 groupedBy()

Appearance: line-rules

Syntax: **`groupedBy(relation)`**

This function takes the incoming Base Type elements and groups them based on the parameter (related Type). The parameter groupings are displayed in the view.

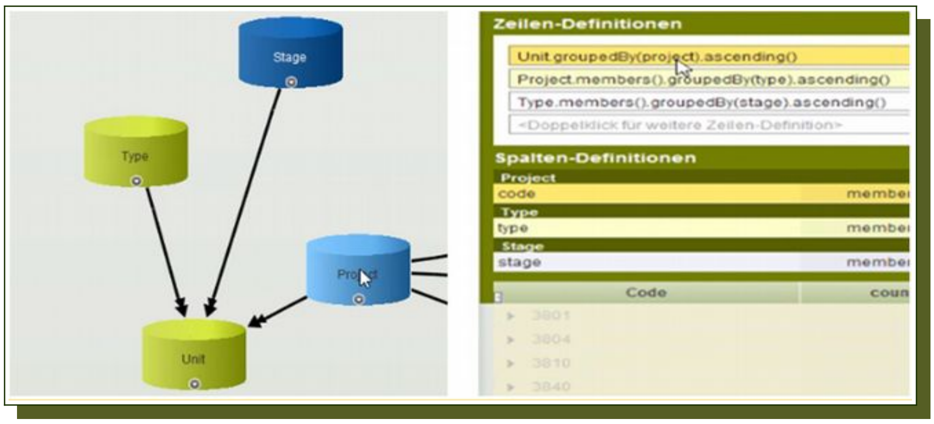
Each grouping has a list of elements from the incoming Type that is internally recognised in the background and referred to as its members list. Each members list is stored under a 'key" which is used to do the grouping.

Using the Meta Model pictured below as an example, and the line rule: **`Unit.groupedBy(project)`**:

All of the incoming *Unit* elements are evaluated with their *project* relation. For each *project* a new members list is created and the *Unit* elements stored under that *project* element key. If the next *Unit* element has a *project* that is already registered with a members list, that *Unit* element is added to that members list. This process results in a list of *projects*, each displayed only once, each with a stored members list based on the related *Unit* elements.

This function is used in conjunction with the members()function to display the members and as a combination they are designed to function as **pivot tables**.

The purpose of this function is to group a Type by its relations and to be able to refine each grouping using other relation groupings to display multi-level categorisation of the initial Base Type elements e.g. you have 1000 *Unit* elements and at first group them under a *project*, then the same members list is grouped by *stages* within each project and lastly regrouped under *types* within each stage to display a cascaded structure of categorised *Unit* elements (see view configuration image below).



Zeilen-Definitionen

```
Unit.groupedBy(project).ascending()
Project.members().groupedBy(type).ascending()
Type.members().groupedBy(stage).ascending()
<Doppelklick für weitere Zeilen-Definition>
```

Spalten-Definitionen

Project	
code	member
Type	
type	member
Stage	
stage	member

Code	coun
3801	
3804	
3810	
3840	

2.4.2 members()

Appearance: line/column-rules

Syntax: **`members()`**

This function returns a list of elements that was collected with the **`groupedBy()`** function.

No parameter is required as the members list is defined with the **`groupedBy()`** function.

2.4.3 single()

Appearance: Column-rules

Syntax: **`single(type)`**

This function returns a reduced list of elements where each input element exists only once.

Generally used in column definitions and for to-many relations, it is useful to condense a long list of values to only show each value once.

> Syntax1: **`Item Order.single(wine)`** or **`Item Order.wine.single()`** returns the Wine varieties for the incoming list of Orders listing each wine once.
> Syntax2: **`Item Order.single().wine`**.....

2.4.4 getMultiple()

Appearance: Column-rules

Syntax: **`getMultiple(prop)`**

This function returns all elements of an input list that occur multiple times and is generally used in line rules. If a parameter is entered it is used as the reference to be tested when matching against each input element.

This function can quickly identify identical values of a property for comparing those elements in a list e.g. **`Item.getMultiple(ItemNo)`** would return all Items that have a '**`ItemNo`**" that appears more than once.

This function lists the elements where the parameter has duplicates and filters out the elements where the parameter appears only once.

> Only 1 parameter is allowed however a combination of values can be added using '+" to form a **concatenated string** that is used to check multiples
> e.g. **`Item.getMultiple(ItemNo + qty)`** returns a list of Items based on the two values matching.

2.4.5 getSiblings()

Appearance: line-rules

Syntax: **`getSiblings(parm1, parm2)`**

This function returns a list of elements which have the same value as the input element based on the parameter entered. This function is designed for column rules however it can be used in line rule filter statements if required.

Multiple parameters can be entered for more specific matches to be found either separated by the "**`+`**", as a concatenated string, or with a comma as a parameter list.

This function is useful to find other elements that have matching values to a line element in a view e.g. on a Item line as a column definition you could request: **`getSiblings(ItemNo + order.)`** which would return a list of Item elements that have the same ItemNo and quantity values (including the line element itself).

2.4.6 allElements()

Appearance: line-rules

Syntax: **`allElements(Type)`**

This function returns all elements of the Type that is provided in the parameter.

The **`allElements()`** function ignores any line rules or navigation paths and lists all results for the base Type. This list, if filtered further, can provide results for unrelated elements for reference and comparisons.

Normally a line rule starts with one Type and related lines refer to elements that have relations to the line above elements.

When you have a root rule and cascaded rules you can only access related elements of the root rule e.g. **`Items.children`**, where you only get children of the Items.

This function enables foiling that principle if other elements, which don't belong to the initial navigation path need to be accessed and used in filters or for other means.

Further filtering or resulting property values can be specified after the **`allElements()`** function.

2.5 Cascade Functions

2.5.1 lineElement()

Appearance: column-rules

Syntax: **`lineElement()`**

This function returns the line base element and with reference to itself (element ID) this function is mainly used in column definitions.

Each line in a view is the **`lineElement()`** and so, in conjunction with other functions, this function is best used for elements that have been grouped and it is useful for making comparisons with property values.

When using the **`allElements()`** function to ignore rules the **`lineElement()`** function works as an immediate reference back to the rules and the lines in the view.

2.5.2 lineAbove()

Appearance: column-rules

Syntax: **`lineAbove()`**

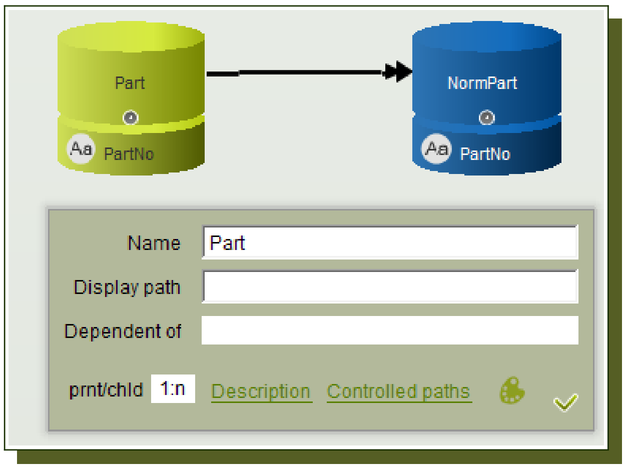
This function returns the element which is represented by the line one level above and it is generally used in column rules.

This function takes the element 'Base Type" from the line level above in the view and uses it as the reference for the path, therefore **`lineAbove()`** should only be used in cascaded views.

2.5.3 lineLevelBelow()

Appearance: Column-rules

Syntax: **`lineLevelBelow(type)`**



Name	Part
Display path	
Dependent of	
prnt/chld	1:n

This function returns all elements which are represented by the lines one level below and it is generally used in column rules and variables.

It takes the element 'Base Type" from the line level below in the view and uses it as the reference for the path, therefore **`lineLevelBelow()`** should only be used in cascaded views.

A parameter must be entered and reference a suitable line Type in the view for paths to be complete and for results to be displayed.

Rules which display *part, part.children* and *part.normpart* in a view could display Item line elements that have a *child* element AND a *normpart* element in its lines below so the parameter must be one or the other.

Further relations, properties or filters segments should be continued after the **`lineAbove()`** function for more detailed results to be displayed.

2.5.4 level()

Appearance: Column-rules

Syntax: **`level()`**

This function is used independently of any other functions or elements and is generally used in column definitions of cascade structured views. With multi-level cascades and parent/children structures it can be useful to determine the depth (level) of each line in the view.

```
A cascade structure of

element1
        element2
                element3
        element4
```

with the column definition **`level()`** results in a number for each line representing the cascade level of the element e.g. element1 = 0, element2 & element4 = 1 and element3 = 2.

2.5.5 getParentChain()

Appearance: Column-rules

Syntax: **`getParentChain()`**

This function can only be used in column definitions and returns a list of the parents for each element in a structure.

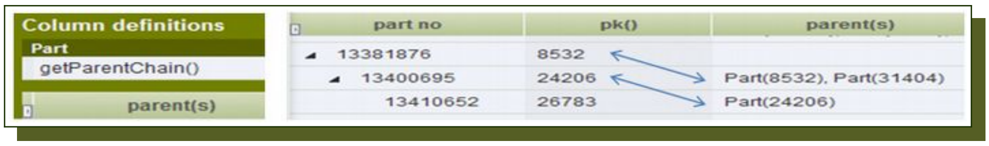
This function is only applicable to Types that have a *prnt/chld* structure as defined in the Meta Model and it can be used to quickly display the chain of parents in a structure for easy reference.

2.5.6 CheckParents()

Appearance: Column-rules

Syntax: **`chain.checkParents(evaluation, exitCondition) – where the exit condition is optional`**

This function is only applicable to Types that have a *prnt/chld* structure as defined in the Meta Model and it can be used to go upwards through a parent chain and check if each level fulfills the evaluation condition.



part no	pk()	parent(s)
13381876	8532	
13400695	24206	Part(8532), Part(31404)
13410652	26783	Part(24206)

This function, with a condition, is used to check all the parents up the line of a structure and returns true if one parent chain meets the evaluation criteria, if no complete parent chain is fulfilled then false is returned.

2.5.7 hasChildren()

Appearance: line-rules

Syntax: **`path[filterConditions & hasChildren()]`**

This function is a special function that is only applicable in cascaded views with multiple cascading line rules. It is a Boolean function that returns true if there is at least one result for a related line below, 'under" the currently evaluated element. This function therefore only displays the incoming line elements when the lines specifications for the related line(s) below have been met.

It is a rule based function with direct reference to the 'next" line rule(s) (line below) and filters out line elements that do not meet the next line rule conditions.

This function can be understood as: 'Show the current line only if there is at least one line in the next indented level below."

Having this function in the filter of a line rule (either on its own or together with other filter restrictions in an **`&`** (AND) operation) causes the evaluated intermediate line to not pass the filter if there are no visible lines in the next indented level below.

2.5.8 markSuccess()

Appearance: Column-rules

Syntax: **`path[filterConditions & markSuccess()]`**

This function is only applicable in cascaded views and the **`markSuccess()`** function always works together with the **`showChildren()`** function.

The purpose is to switch off further filter restrictions of lower level line evaluations if one intermediate line fulfills the criteria.

For example we have a cascade of Items which have again Items in the next level (so called parent/child relation, but this behaviour is not restricted to them). If a filter condition of an iterating rule now looks for a specific Project being related with one Item, the **`markSuccess() / showChildren()`** combination causes that all lower level Items are shown, even if they are not related to that Project.

The **`markSuccess()`** function is a Boolean function which always returns true. The main task of that function is to mark a created line element with a 'success" flag using a filter: … **`[someFilterCondition & markSuccess()]`**. Due to the nature of the **`&`** (AND) operation, the right side is only evaluated if the left side was successful, therefore only 'successful" elements get that success flag.

This function must follow the **`"&"`** (AND) evaluation and when used it internally remembers all elements that meet the line conditions (left of the '&" evaluation).

This function is only used in conjunction with the **`showChildren()`** function which is later used to show all of the children elements whose parents have been marked. That is to say that even if the children do not meet the children line specifications they are still shown below the marked parent element in the structure.

2.5.9 showChildren()

Appearance: line-rules

Syntax: **`Type.children[filterConditions & markSuccess() | showChildren()]`**

This function is only applicable in cascaded views. This function also follows the **`|`** (OR) evaluation and shows the children of the parent line element even if the children don't meet the requirement to the left of the OR condition.

This function is mainly used after the **`markSuccess()`** function and all parent elements that are marked will have their children visible in the structure.

The **`showChildren()`** function is also a Boolean function. It checks all line levels above to see if one of them has the 'success flag" set in a previous calculation. Using that in a filter within an OR operation … **`[someFilterCondition | showChildren()]`** makes the overall filter condition result true, even if the left side of the OR was false.

2.6 Multi-Occurrence Functions

2.6.1 cPath()

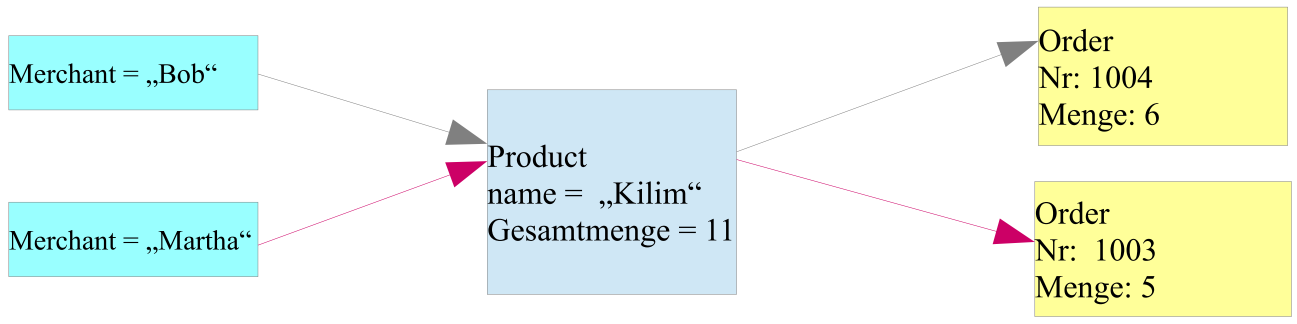
Appearance: Column-rules

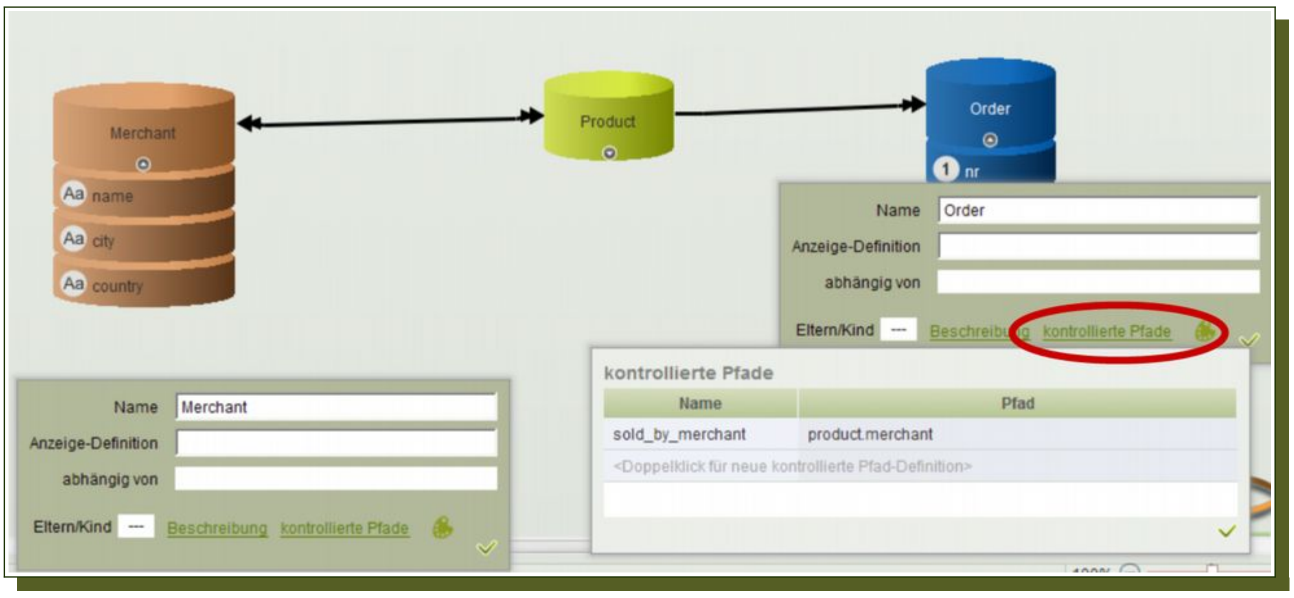
Syntax: **`cPath(cPathProperty)`**

This function is used as a multi-occurrence function and with the help of this function multi-occurring elements can be differentiated. This function can only be used when a controlled path has been set in the repository Meta Model.

This function returns false if a cPath element was found for a given evaluation although it doesn't match the current parent cascade.

Controlled paths are set to follow the individual path and sequence of elements back to the source and for instances where the same element can be used many times but their paths must be separated and remain fixed (see an example in *JACAMAR Admin Guide, pp. 32).*





Name	Order
Anzeige-Definition	
abhängig von	
Eltern/Kind	--- Beschreibung kontrollierte Pfade

kontrollierte Pfade

Name	Pfad
sold_by_merchant	product.merchant
<Doppelklick für neue kontrollierte Pfad-Definition>	

Name	Merchant
Anzeige-Definition	
abhängig von	
Eltern/Kind	--- Beschreibung kontrollierte Pfade

2.7 Parametric Functions

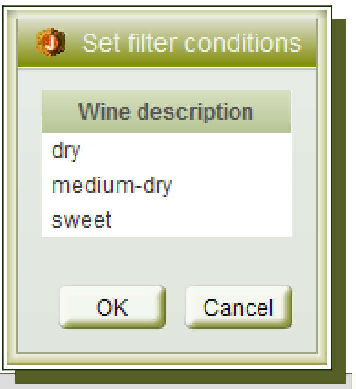
2.7.1 filter()

Appearance: line-rules

Syntax: **`filter(categoryName)`**

This function filters the input elements and returns a list of elements which fulfill the conditions. It is only used in conjunction with predefined Parametric filter categories and conditions.

Parametric views are based on view configurations that work with global filters (menu: *Operation → Parametric Filter*) and are used to filter and list similar data with several parameters.



The relating function is named: **`filter(Color)`** e.g.
`wine.filter(color)`
When opening a view, that contains such filter() functions a dialog appears from which to choose the concrete filter (see image above).

This function is a place-holder for special filters, which are defined by an administrator and have specific category names.

2.7.2 condition()

Appearance: line/column-rules

Syntax: **`condition(categoryName)`**

This function filters the input elements and returns 'true" for elements which fulfill the conditions. It is only used in conjunction with predefined Parametric filter categories and conditions.

Parametric views are based on view configurations that work with global filters and are used to filter similar data with several parameters.

This function is a place-holder for special filters, which are defined by an administrator and have specific category-names. When opening a view, that contains such **`condition()`** functions a dialog appears from which to choose the concrete filter. This function operates differently to the **`filter()`** function as it returns a Boolean (true/ false) evaluation and therefore allows the filter to be used inside filter statements [] or inside if() statements in line and column rules.

Line rule example where multiple filter conditions can be applied:
`Wine[condition(sweetness) & `*`other conditions`*`]`

Column rule example:
`if(condition(specialPrice, "X"))` – would return 'X" for a line in a structure when a special offer exists.

2.7.3 input()

Appearance: line-rules

Syntax: **`input("please enter a value")`**

This function filters and returns a list by asking the user for input of a value. The incoming list is defined in the navigation rules and selection and display criteria is determined in the **`input()`** function.

Syntax2: **`input("please enter a value")`** - for entering free form text as input values.

This 1st parameter is the headline of the input dialog. If no further parameters are entered the input values must be known and accurately entered to provide results.

Syntax2: **`input("please select", path.for.value.list)`** - for selecting from a value list.

The 2nd parameter defines the path and criteria to the elements for selection e.g. **`input("please select",Item).`** This parameter is optional, however it is beneficial to 'see" the elements from which you can make a selection.

Syntax3: **`input("please select", path.for.value.list, display.path)`** - for displaying the value list elements differently.

The 3rd parameter is relative to the 2nd parameter and is optional. It should only be used if the selection list display is to be more detailed or to be represented differently to the elements.

2.8 Colour Functions

2.8.1 color()

Appearance: line/column-rules

Syntax: **`color(colorName | RGB color)`**

This function is used to determine the background colour of cells.

Parameters must be 'names" of standard colours or three numbers representing the RGB scale coordinates e.g. **`Item.color("red")`** or **`Item.color(255,0,0).`**

This function is useful in conjunction with the if() function when certain conditions are met e.g.
`if(total>"80",total.color("red"),total.color("black"))`
which would display the results when the total is greater than 80 with a red background and otherwise if not it shows the total values with a black background.

2.8.2 textColor()

Appearance: line/column-rules

Syntax: **`textColor(colorName | RGB color)`**

This function is used to determine the text colour of cells, either in line rules or column rules in a view.

Parameters must be 'names" of standard colours or three numbers representing the RGB scale coordinates e.g. **`Item.textColor("red")`** or **`Item.textColor(255,0,0)`**.

This function is useful in conjunction with the if() function when certain conditions are met e.g. **`if(total>"80",total.textColor("red"),total.textColor("green"))`**
which would display the results when the total is greater than 80 in red font colour and otherwise if not it shows the total values in green font colour.

2.9 Conditional Functions

These functions return results based on a true or false system of checking whether criteria has been met and they will be an essential Item of many working views and tables.

2.9.1 if()

Appearance: Column-rules

This function tests a condition and returns one result if the condition is true, and another if the condition is false. The if() function should be used when you want to manage the evaluation of different expressions based on a condition.

Syntax1: **`if(condition, trueExpression, falseExpression)`**

The *condition* is what you want to test and *trueExpression* is the rule that is evaluated if the result of the condition is true. The *falseExpression* is optional and it represents the rule that is evaluated if the result of the condition is false e.g. **`if(size(order)>1,size(order),"one")`**.

This example demonstrates that the two expressions can have two very different purposes, here the number of orders is displayed if greater than 1 and if not the text value 'one" is displayed.

Using the if() function within another if() function is also possible and this is known as 'nesting".

Syntax2:
```
if(condition1,"truevalue1",if(condition2,"truevalue2",
if(condition3,"truevalue3","falsevalue")))
```

The if() function is similar to the map() function, but returns one consolidated result by OR-ing the condition results whereas map() returns multiple values for multiple input elements.

2.9.2 map()

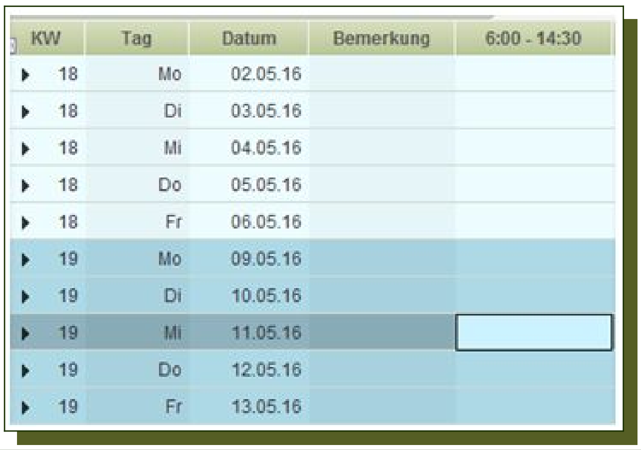
Appearance: Column-rules

Syntax: **`map(condition,"truevalue","falsevalue")`**

This function is similar to the if() function, but returns a list of results for multiple condition results. It maps the true or false results and triggers the evaluation of expressions accordingly.

This function is useful for testing to-many related input elements for to-many output elements to be able to treat each element of a to-many relation separately based on the condition.

Aim: set the background-color of weekdays alternating per week (lightblue, *nothing*):



KW	Tag	Datum	Bemerkung	6:00 - 14:30
18	Mo	02.05.16		
18	Di	03.05.16		
18	Mi	04.05.16		
18	Do	05.05.16		
18	Fr	06.05.16		
19	Mo	09.05.16		
19	Di	10.05.16		
19	Mi	11.05.16		
19	Do	12.05.16		
19	Fr	13.05.16		

The *condition* is what you want to test and *trueExpression* is the rule that is evaluated if the result of the condition is true. → **mod(*parm*,2)**(see page 22) in the example below returns 0 for even and 1 for odd parameters

line rule condition: **dayofWeek.map(mod(week(Datum), 2), color("lightblue"), this())**

all elements (*dayofWeek*) of odd week numbers are displayed with lighblue background color.

2.10 Boolean Operations and Functions

The Boolean operations are commonly used to expand the usefulness of other functions that perform logical tests. For example, the if() function performs a logical test and then returns one value if the test evaluates to true and another value if the test evaluates to false. By using the '&" operator as Item of the evaluation of the if() function, you can test many different conditions instead of just one.

The operators are mainly used within filter statements **[]** to combine evaluations for the given elements.

2.10.1 & (AND) operator

This function must be used with Boolean parameters and it returns true if all its evaluations are true.

If during the evaluation one value is false or not a Boolean, the false result is returned and the remaining parameters will not be evaluated.

This tests a number of user-defined conditions and returns *true* if ALL of the conditions evaluate to true, or *false* otherwise.

2.10.2 | (OR) operator

This function must be used with Boolean parameters and it returns true if any of its evaluations are true.

If during the evaluation one value is true, the true result is returned and the remaining parameters will not be evaluated.

This tests a number of user-defined conditions and returns *true* if ANY of the conditions evaluate to true, or *false* otherwise.

2.10.3 Equations

This is the syntax of equation operators:

```
      property = property
```
or
```
      property = „hardcoded text"
```

besides the equal sign here is the full list of operators that can be used:

=	equals
!=	not equal to
>..	starts with ... (for text)
!>..	does not start with ... (for text)
..<	ends with ... (for text)
!..<	does not end with ... (for text)
>..<	contains ... (for text)
!>..<	does not contain ... (for text)
>=	greater than or equal to
>	greater than
<=	lower than or equal to
<	lower than

2.10.4 not()

This function returns the reverse logical value that is the opposite of the input or parameter Boolean value i.e. returns *false* if the supplied evaluation is true and returns *true* if the supplied evaluation is false.

The **`not()`** function is used for values or expressions that can be evaluated to true or false e.g. when you want to make sure a value is not equal to one particular value.

If there are multiple input elements each input element must be false for the not() function to return true.

Syntax1: **`size(Item order)<1`** – returns true when the amount of orders is 0, however **`not(size(order)<1)`** - would return false

Syntax2: **`Wine[not(special Price ="X"]`** – returns all Wine Items that do not meet the filter criteria.

2.11 Numerical Operations and Functions

The following can be used to show numeric values or calculated content in column rules e.g. Line Type Item: Column rule: **`size(order)`** returns the number of orders for the Item.

These functions may also be used as Item of filter statements in line rules to display elements that meet line specified conditions e.g. **`Item[size(order)=1]`** to only display Item elements with 1 order.

Numeric functions generally return integer values although decimal format will be displayed when at least one of the evaluated values is decimal. Formatting can be manually changed in columns for other numeric display preferences.

These functions assume the input values are numeric (integer or decimal) however for convenience they also automatically convert Boolean evaluations and strings represented by numeric values so that the result of the function can be evaluated.

The numeric functions will provide an error if the parameters are invalid and the value 0.00 will be assumed for non-numeric values in the lists (unless otherwise stated in the specific function description).

2.11.1 + (add) and - (subtract) operators

These operators add or subtract the given values and require a left and right side of operation.

Syntax1: **`property1 + property2`** – returns a numeric total

If all values are numeric integer values, the result is also an integer; otherwise it returns a decimal value.

If one of the given values is non-numeric the + *(add)* operation is treated as a concatenation of strings.

Syntax2: **`string1 + property1`** – returns a combined string of values e.g. '**`string1property1`**"

2.11.2 / (divide by) and * (multiply by) operators

These operators divide or multiple the given values and require a left and right side of operation.

Syntax1: **`property1 * property2`** – returns a numeric total

If all values are numeric integer values, the result is also an integer; otherwise it returns a decimal value.

Non-numeric values are treated as 0 and return a decimal value.

2.11.3 Average()

The average function is used to calculate the average value from the sum of a list of supplied values (integer and decimal).

The syntax of the function is:
`average(number1,number2,...)` where the number evaluations are a set of one or more numeric values for which you want to calculate the average.

Syntax1: **`order.qty.average()`** or **`average(order.quantity)`** This expression calculates the average of all input numbers (here the quantities of each Order). The result is one number.

`order.average(quantity)` would also return the same result provided only one input element is being evaluated.

Syntax2: **`average(qty1,qty2,qty3)`** This expression calculates the average of all numbers that result from the evaluation of the parameter rules. The result is one number.

Syntax3: **`Items.average(order.qty1,order.qty2)`** or **`Items.average(order item.bestellt)`** This is the so-called Vector syntax. That means for each input element (here all Items) a number is created based on the relative parameter rules.

In Syntax3 the first expression calculates the average of the Item's own quantities and returns a list for each Item that exists in the database. The second expression calculates the average of all related ppo.bestellt values and returns a list for each Item that exists in the database.

2.11.4 min() and max()

These functions return the smallest (minimum) or the largest (maximum) value of a list of numeric values (integer and decimal).

These functions can also be used with dates however due to the automatic conversion to numeric values, the **`toDate()`** function is required to display the results as a date.

The syntax variations for these functions are consistent with **`average()`** including the Vector syntax e.g. **`Items.min(order.qty1,order.qty2)`** would return the smallest quantity value of the parameters for each Item.

2.11.5 Sum()

This function returns the sum of a supplied list of numbers. All numbers will be handled as numeric values, all words and other elements will be set to zero (0), however text representations of numbers and dates will be treated as numeric values. Boolean returns of True (1) and False (0) are also treated as numeric values.

It has to be remembered that as long as integer data types will be summed up, the result will be an integer too. If there are decimals, the result will be delivered in decimal form even if the final total is integer.

The syntax variations for these functions are consistent with average() including the Vector syntax.

2.11.6 square (), cube() and sqrt()

These functions calculate the square (a^2), cube (a^3) and square root ($\sqrt{} a$) of each supplied value (integer and decimal).

The syntax variations for these functions are consistent with average() including the Vector syntax, however each parameter is treated and evaluated individually e.g. **`cube(1,2)`** would return 1, 8 - not the cube of 3 (27) !

The numerical values can be supplied directly to the function, as values returned from other functions, or as references to integer and decimal properties.

The sqrt() function calculates the positive square root of a supplied number and it will return an evaluation error if a supplied number is negative.

2.11.7 mod()

This function, known as the 'modulo operation", returns the remainder of a division between two supplied numbers.

The format of the function can be: **`mod(number,divisor)`** or **`number.mod(divisor)`** i.e. the number to be divided followed by the divisor that the evaluation is divided by (last parameter).

An example evaluation is **`mod(8,6)`** where the result would be 2 i.e. 8 divided by 6 is 1 remainder 2.

2.11.8 divisible()

This function returns true if a number is exactly divisible by another number (last parameter is always the divisor).

Syntax1: **`property.divisible(divisor)`**

Syntax2: **`divisible(property,divisor)`**

Syntax3: **`divisible(Type.property,divisor)`** – would return a boolean for each Type.property

2.11.9 random()

This function returns a randomly calculated integer. If a parameter is provided this defines the maximum number of the random selection. A new random number is returned when the table is recalculated or when the view is reopened.

Syntax: **`random(), random(100) or random(order.qty)`**
The parameter must be an integer or it will return an error.

2.11.10 Round()

This function is used to round all input values to the nearest integer (whole number).

Syntax: **`round(Item.price)`** and **`Item.price.round()`** results in decimal values being rounded up or down to the nearest whole number.

Alternatively a parameter can be added to round and define the decimal places.

Syntax: **`round(material.weight,3)`**,**`material.round(weight,3)`** or **`material.weight.round(3)`** results in all material weights being rounded up or down to three decimal places.

This function is useful for property results that have many digits after the decimal point and you only want to display a certain number.

2.11.11 StandardDeviation()

This function returns the standard deviation of a supplied set of values. Standard deviation is a measure of how spread out the numbers are and this function is useful to identify values that are within one ‘standard deviation” of the average, therefore being able to identify what is normal, what is extra large or what is extra small in a set of values.

The **`standardDeviation()`** function calculates the average of the numbers, then the square root of the variance, see variance() function.

2.11.12 variance()

This function returns the variance of a supplied set of values. Variance is defined as the average of the squared difference from the average.

> The variance() function calculates the average of the numbers, then squares the +/- difference from the average for each value and as a total is then divided by the number of values to create the variance.

2.12 Date functions

When using date elements in numeric evaluations or converting numbers into date elements it needs to be remembered that a date in a numeric context is treated as the number of days since 01/01/1970.

This could be useful for highlighting date values when timelines for tasks or projects have been reached or expired
e.g. **`if(dateElement<today()-30,dateElement.textColor("red"),dateElement)`** which would show the *dateElement* in red font when it is 30 days old.

In combination with the sum() function you can use the date functions for comparisons and to calculate time periods and distinguish which date in a timeline comes first.

2.12.1 Date()

This function when supplied with three integers returns a date representation and can be used for keeping a consistent format for numbers represented by dates.

> Syntax: **`date(year,month,date)`** – where the year, month, and day evaluations are integers representing the year, month and the day.

Typically, when using the date() function, the supplied year will be between 0001 and 9999, the month will be between 1 and 12 and the day will be between 1 and 31. However, these values can extend outside these ranges, in which case, they behave as follows:

If the supplied year evaluation is negative or is 0, this value is added onto 0001.
For example: **`date(-10,2,5)`** returns: 5/2/0011.

If the supplied month evaluation is negative or is greater than 12, the date extends back or forward, into the previous or following year. For example: **`date(2016,-1,5)`** returns: 5/11/2015.

If the supplied day evaluation is negative or is greater than 31, the date extends back or forward, into the previous or following month. For example: **`date(2015,12,33)`** returns: 2/1/2016.

2.12.2 day()

This function returns an integer representing the day of the month (from 1 - 31) from a supplied date or number.

It is generally used for property results of a date format and is similar to changing the column format to DD, however for numbers or further calculations the day() function must be used.

Syntax1: **`day()`** – on its own and without a parameter this returns the day of the current month.

Syntax2: **`day(Date)`** or **`Date.day()`** – returns the day of the month of the 'Date" property value.

Syntax2: **`day(number)`** or **`number.day()`** – the number is treated as days since 01 01 1970

For hard-coded numbers you require the **`date()`** or **`toDate()`** functions to convert the string into a date for the **`day()`** function to provide results
e.g. **`day(toDate("20.5.2016"))`** or **`day(date(2016,5,20))`**.

For multiple input elements or a parameter list of numbers this returns one result (one integer) for each element.

2.12.3 Month()

This function returns an integer representing the month of the year (from 1 - 12) from a supplied date or number.

It is generally used for property results of a date format and is similar to changing the column format to MM, however for numbers or further calculations the month() function must be used.

The syntax and application of the month() function is the same as day() with the result being the month of the year representation.

2.12.4 Week()

This function returns an integer representing the week of the year (from 1 - 53) from a supplied date or number.

The syntax and application of the week() function is the same as day() with the result being the week of the year representation.

2.12.5 Year()

This function returns an integer representing the year from a supplied date or number.

It is generally used for property results of a date format and is similar to changing the column format to **`YYYY`**, however for numbers or further calculations the year() function must be used.

The syntax and application of the year() function is the same as day() with the result being the year representation.

2.12.6 Today()

This function returns the current day as a Date element.

No parameters are required and the syntax is simply: **`today()`**.

This function is useful for making comparisons or differences between date values and todays date or to set the current date for a property during the data-merger process.

2.13 Text functions

2.13.1 Substring()

This function returns the substring starting with the first given position and ends with the last given position.

If the last parameter is missing it will take all remaining characters, and for multiple input elements it will create a String for each of them.

Assuming our input element value is the word 'JACAMAR" the following results would be provided:

Syntax1: **`Type.property.substring(3), Type.substring(property,3)`** or **`substring(type.property,3)`** – starting after the 3rd character would return 'AMAR"

Syntax2: **`Type.property.substring(3,5), Type.substring(property,3,5)`** or **`substring(type.property,3,5)`** – starting after the 3rd character up to and including the 5th would return 'AM"

This function could be useful for importing values from other source documents where only a section or segment of the original values are stored in the repository.

2.13.2 Pattern()

This function extracts text based on a pattern.

Syntax: **`Type.pattern(name, "(.{6}) (.)", 2, " ", 1)`**

Assuming the Type has an element with a name 'string1" the above syntax would return a String element with text '1 string".

This function could be useful for importing long strings from other source documents where text is separated by symbols e.g. **|** to find patterns which are needed as values that are stored in the repository.

2.14 Conversion-functions

2.14.1 toBoolean()

This function converts any element to a Boolean value. All numeric values, except zero are treated as the logical value true, and the value zero is treated as the logical value false.

2.14.2 toNumber()

This function converts an element value into an integer or decimal value.

This function changes the format of the values to a different format for the purpose of display or further functions e.g. we have a string property which has some string values and some numeric values and we want to calculate the numeric values. We can't change the property data format to integer because we would lose the string values! So we could '**`sum(toNumber(property))`**' and where the string can be read as a number it will be included in the sum.

All numeric functions are designed to automatically convert string and Boolean values to numerical representations but adding and subtracting calculations may need the toNumber() function to distinguish between numbers and concatanted strings.

2.14.3 toString()

This function returns the text value element of a property evaluation (integer and Boolean properties).

Fixed value list – takes index for integer and Boolean and returns string or text value, in all other cases it returns the element as a String element.

Integers and decimals are treated as numeric therefore when used with "**+**" to form a text display they need to be converted to a string value.

Syntax:
`order.qty1+order.qty2` = numeric total, however
`toString(order.qty1)+toString(order.qty2)` returns a concatenated string i.e. qty1qty2.

2.14.4 toDate()

This function converts a number (integer) to a Date element where the number represents the number of days since **`01/01/1970`**.

The conversion of the numeric value (integer) is represented as a date in JACAMAR's date-time code.

2.14.5 toMonth()

This function converts a date element and returns an integer representing the months since 01/01/1970 of the date element.

Syntax: **`toMonth(dateElement)`**

This function can be useful with the **`groupedBy()`** function for date elements over a period of multiple years.

2.14.6 toYear()

This function converts a date element and returns an integer representing the years since 01/01/1970 of the date element.

Syntax: **`toYear(dateElement)`**

This function can be useful with the **`groupedBy()`** function for date elements over a period of multiple years.

3 Appendix

3.1 Regular Expression Syntax

Here is the table listing down all the *regular expression* metacharacter syntax available in Jacamar

Sub expression	*Matches*
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character except newline. Using **m** option allows it to match the newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
\A	Beginning of the entire string.
\z	End of the entire string.
\Z	End of the entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of the preceding expression.
re+	Matches 1 or more of the previous thing.
re?	Matches 0 or 1 occurrence of the preceding expression.
re{ n}	Matches exactly n number of occurrences of the preceding expression.
re{ n,}	Matches n or more occurrences of the preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of the preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers the matched text.
(?: re)	Groups regular expressions without remembering the matched text.
(?> re)	Matches the independent pattern without backtracking.
\w	Matches the word characters.
\W	Matches the nonword characters.
\s	Matches the whitespace. Equivalent to [\t\n\r\f].
\S	Matches the nonwhitespace.
\d	Matches the digits. Equivalent to [0-9].
\D	Matches the nondigits.
\A	Matches the beginning of the string.
\Z	Matches the end of the string. If a newline exists, it matches just before newline.
\z	Matches the end of the string.
\G	Matches the point where the last match finished.
\n	Back-reference to capture group number "n".
\b	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
\B	Matches the nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\Q	Escape (quote) all characters up to \E.
\E	Ends quoting begun with \Q.