Skin Logic Expressions

Overview

Skin functions and properties provide great flexibility to the skin designer, but sometimes simple property replacement is not enough. One such case is when you need to combine a property with a localized string. Skin expressions provide a way to calculate more complex operations based on skin properties, localizable strings, and other data.

A skin expression is a construct that allows calculations to be performed on property values and the result substituted in the relevant skin control tag or attribute. It can be used (almost) anywhere a property can, such as in label text, a texture filename, and so on. It can even be used within <define> tags to avoid repeating the same expression over and over. It cannot however be used in visibility conditions specified via the <visible> tag.

An expression is any of the following:

- A string literal enclosed in single quotes; example: **'This is a string literal'**
- An integer literal; example: **123**
- A floating point literal; examples: **12.34** and **12.** (note the dot at the end)
- A property; example: **#itemcount**
- A function call; example: **string.format(123, #itemcount)**

Function calls have the following general syntax:

```
_functionname_([_expression_[,_expression_[...]]])
```

Because each argument to a function call is also an expression, it is possible to use nested function calls. Example:

```
string.format('page {0}/{1}', div(#itemindex, 10), div(#itemcount, 10))
```

To distingush expressions from plain text, an expression has to be enclosed in a #(...) construct. Example:

```
<control>
  <description>Number of Files Label</description>
  ...
  <label>#selectedindex/#(string.formatcount(#itemcount, 'no items|{0} item|{0} items'))</label>
  ...
  <visible>string.equals(#selectedindex)+string.equals(#itemcount)</visible>
</control>
```

In the above example, only the **#(string.formatcount(...))** function call and its enclosed arguments are expressions; **#selectedindex** is not an expression. Notice however that the **#(...)** construct has to be used only once, and that is where the expression is to be evaluated. That means that expressions in <define> tags should *not* be enclosed in **#(...)**.

Whitespace is ignored within an expression, but in order to make parsing and cacheing of expressions faster, expressions are compared including whitespace. If you use an expression multiple times or on multiple panels, make sure you use the same whitespace. The following two expressions return the same result, but if both are used in a skin, two expressions will be cached:

```
#(string.format(123,#itemcount))
#( string.format( 123, #itemcount ) )
```

To ensure the best performance of your skin, decide on a whitespace usage convention and stick to it.

Functions

MediaPortal comes with a set of core skin functions that can be used in skin xml. Plugin Developers can provide additional custom functions through their plugins. All functions (core and custom) are treated equally. The only difference is that core functions are guaranteed to be present in every MediaPortal installation.

The following categories of core functions are supported:

- String
- Conversion
- Math
- Date/Time
- Conditionals / Flow Control
- Boolean
- System

String

String functions let you manipulate strings in several ways. They either take strings as arguments or return a string as a result (often both). Since skin properties are always strings, and the purpose of skin expressions is usually to construct some text to display or a filename to use as a texture, these functions are the most commonly used ones.

L(*id* **)**

[Since 1.2]

This function returns the localized string whose identification number is *id*. When a localized string is required by itself, this function need not be used (see first example). However, this function allows you to combine a localized string with constant text and skin variables in order to construct a composite string (see third example). Examples:

```
<label>135</label>
<label>#(L(135))</label>
<label>#selectedindex/#itemcount #(L(507))</label>
```

The first and second examples produce identical output. Note that the first example does not use expression notation **#(...)**, whereas the second example *does* use expression notation. The third example displays a composite string in the style **8/53 items** (depending on the values of the variables). The string with id 507 must contain the localized string whose value is **items**. Note though that you can achieve equivalent results using **string.format()**.

string.contains(*string a, string b* **)**

[Since 1.1]

This function returns true if *string a* contains *string b*. Example:

```
<label>#(switch(
  eq(string.contains(#Play.Current.File,'.dvr-ms'),'true'),
    #Play.Current.File,
  eq(string.contains(#Play.Current.File,'.wtv'   ),'true'),
    #Play.Current.File,
  eq(1,1),
    #Play.Current.Title))</label>
```

For dvr-ms and wtv files, the built-in variable #Play.Current.Title is not set. So this example displays the value of **#Play.Current.File** for dvr-ms and wtv files, and displays the value of **#Play.Current.Title** for all other types of file.

string.starts(*string a, string b* **)**

[Since 1.1]

This function returns true if *string a* starts with *string b*.

string.equals(*string a, string b* **)**

[Since 1.1]

This function returns true if *string a* and *string b* are identical. Some notes:

- The compare is case sensitive.
- Quotes (") are evaluated as well, so **string.equals(a,"b")** is completly different from **string.equals(a,b)**.
- You can use operators like **+**, **|**, and **&**, but be careful with spaces:
    - **string.equals(a,b) | string.equals(c,d)** <<< not working
    - **string.equals(a,b)|string.equals(c,d)** <<< OK

string.format(*format string id*, *arg1*, *arg2*, ... **)**

[Since 1.2]

This function formats parameters according to *format string* or localized string referenced by *format string id.* The format string contains format items in the syntax:

{*index*[,*length*][:*formatString*]}

Elements in square brackets are optional. The following describes each element.

index

The zero-based position in the parameter list of the object to be formatted. If there is no parameter in the index position, an error is returned.

,length

The minimum number of characters in the string representation of the parameter. If positive, the parameter is right-aligned; if negative, it is left-aligned.

:*formatSpecifier*

A standard or custom format string that is supported by the object to be formatted. Possible values for *formatSpecifier* are the same as the values supported by the object's ToString(format) method. If *formatSpecifier* is not specified a default format is used.

For more details on the syntax of format string, see the .Net Framework function StringString.Format(). Especially useful are Standard and Custom number /date formats. Examples:

```
#(string.format('{0}/{1} items', #selectedindex, #itemcount))
#(string.format(100, #selectedindex, #itemcount))
```

Both will return a string like this: "3/85 items" (assuming that the string with id=100 contains the localizable string "{0}/{1} items").

Some number format examples (assuming #starrating = 4.56 and #votes = 1234567):

```
#(string.format('{0:f1} ({1:n}', cflt(#starrating), cint(#votes)))
```

Returns **4.6 (1,234,567)** on en-US regional settings but **4,6 (1.234.567)** on el-GR. The standard number formats use the regional settings to determine how to format the numbers.

```
#(string.format('{0:0.0} ({1:#,0})', cflt(#starrating), cint(#votes)))
```

Returns **4.6 (1,234,567)** on any regional settings. Regional settings still determine group and decimal separators but not whether they will be used or not. Also note that the 0.0 format will output 4.0 if #starrating = 4.

Some date format examples (assuming #date = 17/3/2010):

```
#(string.format('{0:D}', cdate(#date)))
```

Returns **Wednesday, March 17, 2010** on en-US regional settings but **?, 17 ? 2010** on el-GR. The standard date formats use the regional settings to determine how to format the date.

```
#(string.format('{0:dd/MM/yy}', cdate(#date)))
```

Returns **17/03/10** on any regional settings.

```
#(string.format('{0:MMMM d, yyyy}', cdate(#date)))
```

Returns **March 17, 2010** on en-US regional settings and **? 17, 2010** in el-GR. Note that the format does not change based on regional settings but the month names get translated.

In general use standard number/date formats to show the values formatted according to the user's regional settings. Use custom number/date formats to show the values formatted the way you want regardless of the user's regional settings.

Also never forget to convert properties to the proper type before formatting. Trying to format an unconverted (string) property as date, will simply output the property string unmodified.

string.formatcount(*value*, *multi format* **)**

string.formatcount(*value*, *multi format id* **)**

[Since 1.2]

This function formats *value* according to one of three formats depending on whether *value* is 0, 1 or greater than 1. The argument *multi format* (or the localizable string specified by *multi format id*) should contain the the formats separated by the | character. Example:

```
#(string.formatcount(#itemcount, 'no items|{0} item|{0} items'))
```

If #itemcount is 0, this function returns **no items**.

If #itemcount is 1, this function returns **1 item**.

If #itemcount is greater than 1, this function returns something like **53 items**.

string.tolower(*string* **)**

This function returns *string* converted to lower case. Examples:

```
<label>#(string.tolower('ThIs iS A sTrInG'))</label>
<label>#(string.tolower(#TV.RecordedTV.Genre))</label>
```

The first example displays the text **this is a string**.

string.toupper(*string* **)**

This function returns *string* converted to upper case. Examples:

```
<label>#(string.toupper('ThIs iS A sTrInG'))</label>
<label>#(string.toupper(#music.codec))</label>
```

The first example displays the text **THIS IS A STRING**.

string.titlecase(*string* **)**

This function returns *string* converted to title case. Title case converts the first letter of each word to upper case, and the remaining letters of each word to lower case. Examples:

```
<label>#(string.titlecase('ThIs iS A sTrInG'))</label>
<label>#(string.titlecase(#new_file_name))</label>
```

The first example displays the text **This Is A String**.

string.ltrim(*string* [, *charsToTrim*] **)**

[Since 1.2]

This function trims whitespace, or the characters specified by *charsToTrim*, from the left end of the string. If *charsToTrim* is present, it must be a single character string with the characters to trim specified consecutively. Do not use commas or blanks to separate the characters to trim, as the commas or blanks will themselves be treated as characters to trim. The following example trims whitespace (blank characters):

```
<label>#(string.ltrim(#date))</label>
```

The following example trims the upper case letters ABC. These letters can occur in any order, and can occur more than once. Trimming stops when a character which is not one of *charsToTrim* is found in the string. Note that the lower case equivalents are *not* trimmed:

```
<label>#(string.ltrim(#skin.prefix,'ABC'))</label>
```

The following example trims the upper and lower case letters ABC, plus any blanks, commas, or dots that may be present:

```
<label>#(string.ltrim(#date,'ABC ,.abc'))</label>
```

string.rtrim(*string* [, *charsToTrim*] **)**

[Since 1.2]

This function trims whitespace, or the characters specified by *charsToTrim*, from the right end of the string. Refer to the description of the function **string.ltrim()** for details of the parameters. Examples:

```
<label>#(string.rtrim(#time))</label>
<label>#(string.rtrim(#time,'AMPamp'))</label>
```

string.trim(*string* [, *charsToTrim*] **)**

[Since 1.2]

This function trims whitespace, or the characters specified by *charsToTrim*, from both ends of the string. Refer to the description of the function **string.ltrim()** for details of the parameters. Examples:

```
<label>#(string.trim(#time))</label>
<label>#(string.trim(#time,'1234567890'))</label>
```

string.replace(*string, fromSubstring, toSubstring* **)**

[Since 1.22]

This function returns *string* with all occurrences of the substring *fromSubstring* changed to the substring *toSubstring*. Examples:

```
<label>#(string.replace(#TV.RecordedTV.Time,' - ','-'))</label>
<label>#(string.replace(#mpei.updates,'.::.','\r'))</label>
```

string.length(*string* **)**

[Since 1.19]

This function returns the length of the string.

string.valuecontains(*string a, string b* **)**

[Since 1.18]

This function returns true if string or value of property 'a' contains string or value of property 'b'.

string.valuestarts(*string a, string b* **)**

[Since 1.18]

This function returns true if string or value of property 'a' starts with string or value of property 'b'.

string.valueequals(*string a, string b* **)**

[Since 1.18]

This function returns true if string or value of property "a" and "b" are identical.

Conversion

Skin Properties in MediaPortal are always strings. But some functions either require some other type (integer, date etc.) as parameter, or work differently based on the type of the parameters passed. In most cases strings are implicitly converted to the required type. But if the conversion is ambiguous, or results in loss of precision, it cannot be applied implicitly and you have to convert explicitly to the desired type. At other times the implicit conversion will choose a type to convert to that is not appropriate for your needs. In these cases you may choose to convert explicitly to the type you desire. To convert explicitly any value to some other type, use one of the following functions.

cint(*value* **)**

[Since 1.2]

Convert *value* to integer. If *value* is not a number, returns an error.

cflt(*value* **)**

[Since 1.2]

Convert *value* to float. If *value* is not a number, returns an error.

cdate(*value* **)**

[Since 1.2]

Convert *value* to date. If *value* is not a valid date, returns an error.

cdate(*value, format* **)**

[Since 1.19]

Convert *value* to date with *format* of value. If *value* is not a valid date, returns an error.

ctimespan(*value* **)**

[Since 1.19]

Convert *value* to ctimespan. If *value* is not a valid ctimespan, returns an error.

ctimespan(*value, format* **)**

[Since 1.19]

Convert *value* to ctimespan with *format* of value. If *value* is not a valid ctimespan, returns an error.

Math

Some times the values returned in properties are not exactly how you want them. You many need to do some calculations on these values. Although operators are not currently supported, you can still do basic math using function syntax.

neg(*arg* **)**

[Since 1.2]

Return the negative of *arg*.

add(*arg1*, *arg2*, ... **)**

[Since 1.2]

Return the sum of all arguments (i.e. *arg1*+*arg2*+....*argN*).

sub(*arg1*, *arg2* **)**

[Since 1.2]

Return the difference *arg1* - *arg2*.

mul(*arg1*, *arg2*, ... **)**

[Since 1.2]

Return the product of all arguments (i.e. *arg1 * arg2 *... * argN*).

div(*arg1*, *arg2* **)**

[Since 1.2]

Return the quotient *arg1 / arg2*. If *arg2* is zero, a "division by zero" error will be returned.

math.round(*number* [, *digits*] **)**

[Since 1.2]

Round *number* to the closest number having *digits* decimal digits. If digits is not supplied, 0 is assumed (i.e. round to closest integer).
Note that you can also use negative values for digits to round to multiples of 10, 100 etc.

Examples:

math.round(16.32, 1) returns **16.3**

math.round(16.36, 1) returns **16.4**

math.round(16.32) returns **16**

math.round(16.32, -1) returns **20**

math.ceil(*number* [, *digits*] **)**

[Since 1.2]

Return the smallest number having *digits* decimal digits that is greater than or equal to *number*. Similar to **math.round()** but instead of rounding, truncates upwards.

Examples:

math.ceil(16.32, 1) returns **16.4**

math.ceil(16.36, 1) returns **16.4**

math.floor(*number* [, *digits*] **)**

[Since 1.2]

Return the largest number having *digits* decimal digits that is less than or equal to *number*. Similar to **math.round()** but instead of rounding, truncates downwards.

Examples:

math.floor(16.32, 1) returns **16.3**

math.floor(16.36, 1) returns **16.3**

Date/Time

Working with dates is always tricky. You can't treat them as strings, and you can't treat them as numbers. You have to use specialized functions that work on dates. Core functions have been included to allow adding and subtracting dates and times as well as extracting date/time parts. There are two basic types used:

- **date** which is actually date/time. It can hold any date and/or time
- **timespan** which can hold the difference between two dates/times. It can later be added/subtracted from any date/time, or specific parts of it extracted.

date.add(*interval*, *number*, *date* **)**

[Since 1.2]

Add the *number* of *intervals* to *date* and return the resulting date. The parameter *number* can be positive or negative, and in some cases even decimal. The valid values for *interval* are:

- **d** or **dd** or **y** or **dy** or **w** or **dw**: Days
- **wk** or **ww**: Weeks
- **m** or **mm**: Months
- **q** or **qq**: Quarters
- **yy** or **yyyy**: Years
- **h** or **hh**: Hours
- **n** or **nn**: Minutes
- **s** or **ss**: Seconds
- **ms**: Milliseconds

date.add(*date*, *timespan* **)**

[Since 1.2]

Add *timespan* to *date* and return the resulting date. Timespans represent the difference between two dates.

date.sub(*date1*, *date2* **)**

[Since 1.2]

Subtract two dates, returning the timespan from *date2* to *date1*.

date.sub(*date*, *timespan* **)**

[Since 1.2]

Subtract *timespan* from *date* and return the resulting date. Timespans represent the difference between two dates.

date.extract(*interval*, *date* **)**

date.extract(*interval*, *timespan* **)**

[Since 1.2]

Extract a date part from a date or timespan. Return the number of occurrences of *interval* in *date* or *timespan*. The valid values for *interval* are:

- **d** or **dd**: Day of month (dates) or Days (timespans)
- **y** or **dy**: Day of year (only use with dates)
- **w** or **dw**: Day of week (only use with dates)
- **wk** or **ww**: Week of year (dates) or weeks (timespans)
- **m** or **mm**: Month (only use with dates)
- **q** or **qq**: Quarter (only use with dates)
- **yy** or **yyyy**: Year (only use with dates)
- **h** or **hh**: Hours
- **n** or **nn**: Minutes
- **s** or **ss**: Seconds
- **ms**: Milliseconds

Example:

```
<label>Recorded #(string.formatcount(
                  date.extract('d',date.sub(cdate(#date),cdate(#recordeddate))),
                  'today|yesterday|{0} days ago'))</label>
```

Depending on the values of **#date** and **#recordeddate**, this example shows one of the following:

Recorded today

Recorded yesterday

Recorded 3 days ago

Conditionals / Flow control

You often need to check for certain conditions in your skin and provide a different panel content based on that. You might want to change the background according to the time of day, or use appropriate greeting text. Conditionals do exactly that, and there are several flavours to choose from, depending on the complexity of the situation.

iif(*condition*, *true part*, *false part* **)**

[Since 1.2]

This function returns *true part* if the *condition* is true, and returns *false part* if the *condition* is false. The *condition* can be any expression returning a boolean result. Examples:

```
<label>#(iif(eq(#currentmoduleid,'603'),
         'Search TV Guide','Search Radio Guide'))</label>
<hyperlink>#(iif(eq(#currentmoduleid,'603'),604,8900))</hyperlink>
```

The first example displays the string **Search TV Guide** or **Search Radio Guide**, according to the module id of the panel that is currently on display. The second example selects which of two panels to display next, when the control containing that hyperlink is clicked.

choose(*index*, *arg0*, *arg1*, ... **)**

[Since 1.2]

This function returns the argument identified by *index*. The parameter *index* is an integer with a value that is zero or greater. A value of 0 returns *arg0*, a value of 1 returns *arg1*, and so on. Example:

```
<label>#(choose(date.extract('dw',#date),
                'Mon','Tue','Wed','Thu','Fri','Sat','Sun'))</label>
```

If the function **date.extract()** returns the value 3, this example displays the text **Thu**.

switch(*condition1*, *value1*, *condition2*, *value2*, ... **)**

[Since 1.2]

This function returns the first *value* for which the corresponding *condition* is true. Example:

```
<define>#hour:date.extract('h',#time)</define>
...
<texture>#(switch(
  and(gte(#hour,6),lt(hour,12)),
    'back-morning.png',
  and(gte(#hour,12),lt(#hour,15)),
    'back-noon.png',
  and(gte(#hour,15),lt(#hour,19)),
    'back-evening.png',
  or(gte(#hour,19),lt(#hour,6)),
    'back-night.png'))</texture>
```

If the texture specified is that of the background, this example rotates the background between morning, noon, evening and night, based on the time of day. Note that in the example above, blanks and indentation have been used to improve the readability of the code. Blanks that occur within a #(...) expression, but which are not quoted, are discarded when the control is processed.

Boolean

Having conditional functions would be of no use, without a way to build as complex conditions as necessary to suit your needs. Boolean funtions allow you to build such conditions. There are two types of boolean functions:

- Comparison functions allow you to check for specific values / ranges of values
- Boolean logic functions can be used to combine conditions to build more complex ones

eq(*value1*, *value2* **)**

[Since 1.2]

This function returns true if *value1* equals *value2*.

neq(*value1*, *value2* **)**

[Since 1.2]

This function returns true if *value1* is not equal to *value2*.

gt(*value1*, *value2* **)**

[Since 1.2]

This function returns true if *value1* is greater than *value2*.

gte(*value1*, *value2* **)**

[Since 1.2]

This function returns true if *value1* is greater than or equal to *value2*.

lt(*value1*, *value2* **)**

[Since 1.2]

This function returns true if *value1* is less than *value2*.

lte(*value1*, *value2* **)**

[Since 1.2]

This function returns true if *value1* is less than or equal to *value2*.

not(*condition* **)**

[Since 1.2]

This function returns true if *condition* is false.

and(*condition1*, *condition2*, ... **)**

[Since 1.2]

This function returns true if all *conditions* are true.

or(*condition1*, *condition2*, ... **)**

[Since 1.2]

This function returns true if one or more *condition* is true.

System

system.idletime(*time* **)**

[Since 1.19]

This function returns true if the MP idle time (in seconds) is more than *time* (in seconds).

Additional Information and References

- **Mantis Issue:** 3023

XML/Code Samples

Screenshots