The system identifies macros using special parentheses. You need to enclose macro expressions into curly brackets and the percentage symbol: ***{% expression %}***

Kentico provides an object-oriented language named **K#**, which defines the syntax of macro expressions. The macro language is very similar to the C# programming language. This page provides an overview of the features available in K# and describes the differences from C#.

The K# syntax is ***case-insensitive*** – the system does not differentiate between upper and lower case letters in commands. However, letter case may have an effect in the values of constants and variables (for the purposes of comparisons and similar).

What is the # character at the end of macros?

When you save a macro expression, the system automatically adds the **#** character before the closing parentheses. The character indicates that the macro is signed.

See Working with macro signatures for more information.

In addition to the primary macro syntax, the system also supports several special macro types:

Special macro type	Description
Query string macros	To load the values of query string parameters from the URL, use macros in format: ***{? parameter ?}*** Query string macros support all K# syntax. The names of all available query string parameters automatically work as variables that store the value of the corresponding parameter. For example, on pages with URLs like */Home.aspx?nodeid=10*, *{? nodeid ?}* resolves into 10. Alternatively, you can get the values of query string parameters inside standard macros: *{% QueryString. parameter %}*
Localization strings	To add localization strings into text values, use macros in format: ***{$ ResourceStringKey $}*** The localization macro resolves into the string version that matches the currently selected language. See also: Localizing text fields Localization macros do NOT support K# syntax, or any expressions except for resource string keys. To localize strings inside standard macros: *{% GetResourceString("ResourceStringKey") %}*
Cookie macros (standardized)	To load values from the current user's browser cookies, use the following standard macro: *{% Cookies ["CookieName"] %}* For example: *The code of the currently selected language is: {% Cookies["CMSPreferredCulture"] %}*
Path expression macros (standardized)	To resolve page path expressions, use the following standard macro: *{% Path["path"] %}* For example, you can enter the following into an SQL where condition field: *NodeAliasPath LIKE {% Path["../%"] %}* (the macro's result matches the path of all pages under the current page's parent)

> The macro engine no longer supports custom macros in format *{# expression #}*. If you are upgrading from an older version of Kentico, and your data contains custom macros, the system automatically converts all occurrences to *{% ProcessCustomMacro("expression", "parameters") %}* to ensure backward compatibility.
>
> We strongly recommend implementing all custom macro functionality using the approaches described in Extending the macro engine.

Values and objects

When the system resolves a macro expression, the result is an object (value). Macros support the following types of objects:

- Standard scalar C# types (int, double, string, DateTime etc.).
- Kentico system types (objects representing pages, users etc.).

> **Note**: Kentico objects in macros provide properties according to the underlying database structure. This does not always match the properties available in the API of the corresponding *Info* classes.

- Collections based on the *IEnumerable* interface, which contain various types of objects. For example, *InfoObjectCollection* stores sets of Kentico objects.
- Context objects containing data related to the currently processed request (information about the current user, the currently viewed page, etc.). For example: *CMSContext*, *SiteContext*, *DocumentContext*, *ECommerceContext*.
- Macro namespaces that allow you to access other properties or methods (a namespace object itself does not return any data).

To work with fixed values (literals) in macro expressions:

- **numbers** - type the numbers directly. You can use integers or numbers with decimal points (double type).
- **text** - enclose text (string) values into quotes, for example: *"administrator"*
- **booleans** - use the *true* and *false* keywords.
- **date and time** - enclose date and time values into quotes. The date format depends on the culture context. For example: *"1/1/2014"*, *"9/3/2014 9:20:00 AM"*

Operators

K# uses the same basic operators as C#. See the Reference of C# operators to learn more.

The following table summarizes the differences in operator behavior that you need to keep in mind when writing macros.

Operator	Examples	Description
x.y	CurrentUser.UserName	Accesses the members (methods, properties) of macro objects or namespaces.
+	10+5 CurrentPageInfo. DocumentPageTitle + " | suffix"	Adds two operands together. Adding is supported for numbers, and the following types: • **String + String**: Returns the concatenation as a *String* • **DateTime + TimeSpan**: Adds the *TimeSpan* to the *DateTime* and returns the result as *DateTime* • **TimeSpan + TimeSpan**: Returns the sum as a *TimeSpan* String concatenation is the default option if none of the combinations above are detected – the operator returns the concatenation of the operands' *String* representations. For example, *{% "string" + 5 %}* returns **string5**.

-	-10 10-5	Unary - returns the numeric negation of a number. Binary - subtracts the second operand from the first. In addition to numeric types, you can subtract objects of the following types: • **DateTime - TimeSpan**: Subtracts the *TimeSpan* from the *DateTime* and returns the result as *DateTime* • **DateTime - DateTime**: Subtracts the second *DateTime* from the first, and returns the difference as a *TimeSpan* • **TimeSpan - TimeSpan**: Returns the difference as a *TimeSpan*
== !=	50 == 5*10 CurrentUser.UserName != "administrator" CurrentDocument == Documents["/Services/WebDesign"]	Operators that check for equality or inequality of the operands. Return a boolean value. Equality checks support all available object types, with the following special rules: • Empty strings are equal to *null* • Simple data types are equal to their string representation • *Info* objects are equal to string constants that match the object's display name or code name • Two *Info* objects are equal when they have the same object type and ID • GUID values are equal to the string representation of the GUID (always case insensitive) • Enumeration values are equal to constants of the enum's underlying type (integer or string) **Tip**: Use macro parameters to specify case sensitivity and the culture context in equality checks.
< <= > >=	CurrentPageInfo.DocumentPublishFrom <= DateTime.Now	Comparison operators that return boolean values: • **<** (true if the left operand is lesser than the right) • **<=** (true if the left operand is lesser than or equal to the right) • **>** (true if the left operand is greater than the right) • **>=** (true if the left operand is greater than or equal to the right) You can compare objects of the following types: • numeric (Int, Double) • String (comparison based on lexicographical order) • DateTime • TimeSpan **Note**: When comparing strings, use macro parameters to specify case sensitivity and the culture context.
mod	5 mod 2	Performs the modulo operation – computes the remainder after division of two integer operands.
%	30%	In K#, the % character represents percentage values. You **cannot** use the % operator for the modulo operation. Adding the percentage sign converts the preceding number to a double equivalent (multiplies the number by 0.01). For example, *{% 30% %}* returns **0.3**.

??	CurrentDocument.Children.FirstItem ?? "No child pages"	Null-coalescing operator. Returns the left operand if the operand is not *null*, otherwise returns the right operand. **Note**: Empty strings are not considered as null values by the operator.

Macro methods

Methods allow you to perform tasks (execute code) inside macro expressions. You typically call methods with at least one argument.

The recommended K# syntax for method calls is **infix** notation for the first argument. Prefix notation is also supported. For example:

```
// Returns "WORD"
{% "word".ToUpper() %}

// Returns "The sky is red on red planets"
{% "The sky is blue on blue planets".Replace("blue", "red") %}

// Alternative method calls with prefix notation (not supported by the autocomplete
help)
{% ToUpper("word") %}
{% Replace("The sky is blue on blue planets", "blue", "red") %}
```

Kentico provides an extensive set of default methods that you can use in macro expressions. See: Reference - Macro methods

Developers can also extend the macro engine and register custom macro methods.

Compound expressions and declaring variables

K# allows you to write compound expressions, containing any number of simple macro expressions. You need to terminate each expressions (except for the last) using a semicolon. The overall result of a compound expression is the result of the last expression.

Variables allow you to store and manipulate values inside macro expressions. You do not need to explicitly declare the type of variables.

Example

```
// returns "12"
{% x = 5; x + 7 %}

// returns "10"
{% x = 5; y = 3; x += 2; x + y %}
```

The scope of variables spans from the point of declaration to the end of the area containing the macro (such as text fields, email templates or Text / XML transformations), including all separate macro expressions in the given area.

Conditional statements

Use the **if** command to create conditions in format: ***if (<condition>) {<expressions>}*** . The condition statement returns the value of <*expressions*> if the condition is true, and a *null* value if false.

```
// returns "z is less than 3"
{% z = 1; if (z<3) {"z is less than 3"} %}
```

To create conditions with a result for the false branch, use the following syntax: ***if (<condition>) {<expressions 1>} else {<expressions 2>}***

```
// returns "z is greater than or equal to 3"
{% z = 5; if (z<3) {"z is less than 3"} else {"z is greater than or equal to 3"} %}
```

The **ternary operator** allows you to create compact conditions with a different result for each branch. Use the following syntax: ***<condition> ? <expressions 1> : <expressions 2>***

If the condition is true, the statement returns the value of *<expressions 1>*. In the case of a false condition, the return value is *<expressions 2>*.

```
// returns "The second parameter is greater"
{% x=1; y=2; x > y ? "The first parameter is greater" : "The second parameter is
greater" %}
```

Open conditions and loops

When defining conditions or loops, you can leave the body of the loop/condition open and close it later in another macro expression. This allows you to apply the command to text content or HTML code placed between the macro expressions. Open commands can be particularly useful in macro-based transformations or various types of HTML templates.

You can also nest additional macro expressions inside open loops or conditions.

Example

```
// Displays a message including the current date if the year is 2014 or more
{% date = CurrentDateTime; if (date.Year > 2013) { %}
The current date is: {% date %}. The registration period has ended.
{% } %}
```

Iteration (loops)

K# provides several types of loop commands:

- **while** (<condition>) {<executed expressions>}
- **for** (<init expression>; <condition>; <increment expression>) {<executed expressions>}
- **foreach** (<variable> in <enumerable object>) {<executed expressions>}

If a loop command is the last expression in a macro, the return value is a concatenated list containing the results of the sub-expressions executed by all iterations of the loop.

While loop example

```
// returns "10"
{% z = 1; while (z<10) {++z}; z %}

// returns "2 3 4 5 6 7 8 9 10"
{% z = 1; while (z<10) {++z} %}
```

For loop example

```
// returns "5"
{% z = 0; for (i = 0; i < 5; i++) { z += 1 }; z %}

// returns "1 2 3 4 5"
{% z = 0; for (i = 0; i < 5; i++) { z += 1 } %}
```

Foreach example

```
// returns "HELLO"
{% z = ""; foreach (x in "hello") {z += x.toupper()}; z %}

// returns "H HE HEL HELL HELLO"
{% z = ""; foreach (x in "hello") {z += x.toupper()}%}
```

Use the **break** command to terminate loops. For nested loops, the command closes the innermost loop.

```
// returns "1 2 3 4 5"
{% z = 0; while (z < 10) {if (z > 4) {break}; ++z} %}
```

The **continue** command skips to the next iteration of the current loop.

```
// returns "0 1 2 4 5"
{% for (i=0; i<=5 ; i++) {if (i == 3) {continue}; i} %}
```

Nested loops and advanced expressions

Do not use the automatic return value for loops that are nested inside another loop or in compound expressions. Instead, choose one of the following approaches:

- Use console output (the *print(...)* method) to directly output the results of individual loop iterations
  OR
- Concatenate the iteration results into a variable that you return at the end of the expression

Example - Nested loop with console output

```
// Returns an HTML list of all items from all orders made by the current customer
{%
        orders = ECommerceContext.CurrentCustomer.AllOrders;
        if (orders.Count > 0) {
                print("<ul>");
                foreach (order in orders) {
                        foreach (item in order.OrderItems)
                                { print("<li>" + item.OrderItemSKUName + "</li>") }
                };
                print("</ul>");
        }
%}
```

Example - Nested loop with concatenation into a variable

```
// Returns an HTML list of all items from all orders made by the current customer
{%
        orders = ECommerceContext.CurrentCustomer.AllOrders;
        if (orders.Count > 0) {
                result = "<ul>";
                foreach (order in orders) {
                        foreach (item in order.OrderItems)
                                { result += "<li>" + item.OrderItemSKUName + "</li>" }
                };
                return result + "</ul>";
        }
%}
```

Return command

Use the **return** command to terminate the processing of a macro, and set the attached expression as the macro's final result. You can use the return command inside loops, or anywhere in compound macro expressions.

Example

```
// returns green
{% "red"; "yellow"; return "green"; "blue" %}

// returns "ignore the loop"
{% z = ""; foreach (x in "hello") {return "ignore the loop"; z += x } %}
```

Console output

Console output allows you to build the results of macro expressions without declaring variables. Use the **print(<expressions>)** or **println(<expressions>)** syntax. Each console output expression adds to the macro's return value, and the system continues processing the macro.

Console output has higher priority than the standard result of macro expressions, but lower than the **return** command.

Example

```
// returns "123"
{% i = 1; while (i < 4) {print(i++)}; "ignored" %}

// returns "result"
{% i = 1; while (i < 4) {print(i++)}; return "result" %}
```

Tip: To explicitly set the current console output as the return value of a macro, use the plain **return** command without an attached expression.

Indexing

K# supports indexing for collections and objects that serve as containers for other data:

- **DataRow**, **DataRowView**, **DataRowContainer** (returns the value at the specified index)
- **DataTableContainer** (returns the row at the specified index)
- **DataSetContainer** (returns the table at the specified index)
- **String** (returns the character at the specified index)
- **IEnumerable** collections, such as *InfoObjectCollection* (returns the object at the specified index in the collection)

Example

```
// returns "e"
{% "hello"[1] %}

// returns the value of the FirstName column from the DataRow
{% dataRow["FirstName"] %}
```

Comments

To add explanatory text inside macro expressions, use one-line, multi-line or inline comments.

Example

```
{%
// This is a one-line comment. Initiated by two forward slashes, spans across one full
line.

/*
This is a multi-line comment.
Opened by a forward slash-asterisk sequence and closed with the asterisk-forward slash
sequence.
Can span across any number of lines.
*/

x = 5; y = 3; /* This is an inline comment nested in the middle of an expression. */
x+= 2; x + y
%}
```

Lambda expressions

Lambda expressions are ad-hoc declarations of inline functions that you can use inside macro expressions.

Example

```
// returns "4"
{% lambdaSucc = (x => x + 1); lambdaSucc(3) %}

// returns "6"
{% lambdaMultiply = ((x, y) => x * y); lambdaMultiply(2,3) %}
```

The scope of lambda expressions spans from the point of declaration to the end of the area containing the macro (such as text fields, email templates or Text / XML transformations), including all separate macro expressions in the given area.

Macro parameters

Macro parameters allow you to modify how the system resolves individual expressions. To append parameters to macro expressions, use the following syntax:

```
{% ...|(<parameter>)<value> %}
```

You can add multiple parameters to a single expression. Use backslashes to escape the | separator in parameter values if necessary.

> **Note**: The | separator must be placed directly between the macro expression and the parameter ***without any whitespace characters*** .

The following macro parameters are available:

Parameter	Example	Description
default	`{% CurrentUser.UserDateOfBirth|(default)N|A %}`	Sets a default value that the macro returns if the result of the expression is an empty value.
encode	`{% ArticleSummary|(encode)true %}`	Enables HTML encoding for the result of the macro (converts reserved HTML characters to equivalent character entities).
recursive	`{% ArticleText|(encode)true|(recursive)true %}`	If true, the system resolves macro expressions contained in the macro's result recursively. Note: If a macro returns text containing localization expressions in the *{$ResourceStringKey$}* format, these are automatically resolved recursively.
culture	`{% DocumentPublishFrom|(culture)en-us %}`	Sets the culture (language) used when formatting numbers and dates in the macro result.

casesensitive	`{% Contains("term", ArticleText)|(casesensitive)true %}`	Determines whether string comparisons and other operations inside the macro expression are case sensitive. False by default. You can enable case sensitivity globally for all expressions by adding the following key to the *appSettings* section of your web.config file: `<add key="CMSMacrosCaseSensitiveComparison" value="true">`
timeout	`{% GetDocumentUrl()|(timeout)2000 %}`	Sets the maximum time allowed to resolve the expression (in milliseconds). If unspecified, the default timeout is 1000 ms. If the timeout is reached, the system aborts the resolving process. The macro returns a null value and an entry is logged in the event log.
handlesqlinjection	`{% QueryString.Param|(handlesqlinjection)true %}`	If true, the system replaces single quote characters (') in the macro result with two single quotes ('').
debug	`{% Documents["/News"].Children.WithAllData|(debug)true %}`	Enables detailed macro debugging (only for the given expression).