JS Calculator

When installing the *JavaScript* syntax, the expression code area must contain JavaScript. The next example shows how to calculate the sum of two *Calculator* fields:

```
// Calculation of the sum of fields
let result;
result = COL1 + COL2;

return result;
```

It is possible to use references to other *Calculator* fields or variables created in the expressions list, including the new ones.

> **Note:** It is possible to use references to the expressions calculated earlier in the expression, namely, the expressions that are higher in the list of expressions. Consequently, incorrect list item position can cause a mistake.

In this example, it is required to use references to the fields with `COL1` and `COL2` names to calculate the `result` variable.

To show the result in the calculated calculator field, it is required to use the `return` command that returns the `result` variable value which contains the calculated sum of the *Calculator* fields.

> **Important:** The script must be finished by `return` command that shows the result in the calculated calculator field. Otherwise, the script will be executed but the result will not be shown in the calculated field.

Compact code variant for calculation of the calcualtor fields sum:

```
// Calculation of the sum of fields
return COL1 + COL2;
```

Example of calculation of the number power:

```
// Function of calculation of the number power
function pow(x, n) {
  if (n != 1) {
    return x * pow(x, n - 1);
  } else {
    return x;
  }
}
// Show the function result in the calculation field
return pow(COL1, COL2);
```

`pow(x, n)` function is defined in this script. It takes number and power to which it is risen as arguments. To show the script result in the calculated calculator field, it is required to use `return pow(COL1, COL2)` command where `pow(COL1, COL2)` enables to call the declared function with transfer to `x` and `n` parameters of the values from the calculator fields (or variables) `COL1` and `COL2`, correspondingly.

It is possible to use built-in functions *of Calculator* in the JavaScript code:

```
function my_concat(x, n) {
 let s;
 // s = x.concat(n);         - the JavaScript method is used
 s = Concat(x,n);        //   - the calculator function is used

return s;
}

return my_concat(COL1, COL2);
```

Names of the built-in functions of *Calculator* and JavaScript do not intersect each other. The *Calculator* functions always start from the capital letter. However, it is possible to redefine them in the JS code. It is not required to import functions *of the Calculator* distinctly (as it is done in the component *of JavaScript*).

> **Note:** As opposed to the component *of JavaScript* in *Calculator* the Promise object is not supported.

Import of External Modules

External modules are supported as in the JavaScript component but it is possible to use only CommonJS module systems in *Calculator*. ES6 module system (ECMAScript 6) is not supported. Examples and documents related to the usage of the CommonJS modules are described in the following article: JavaScript: Import of External Modules.

Features of External Modules Usage

To import the CommonJS module, it is required to use `require` function. For example:

```
const foo = require("foo/foo.js");
```

The module objects are initialised upon the first call. In the case of subsequent calls, cache is used. The module objects state of the previous call is preserved in it. As the script is executed in *Calculator* for each of the calculated table cells, the included `require` module is repeatedly called, and cache is used. Thus, the changed state of the module objects can be transferred from call to call. It is required to take into account and clean out the cache available via `require.cache` object when required:

```
let path = require.resolve("foo/foo.js");
delete require.cache[path];  // The cache of the "foo/foo.js" module is
cleaned out.
```

> **Important**: The cache usage when transferring values between `require` calls can cause unpredictable results of calculations due to the usage of calculation multithreading.

Quick Access Toolbar

The toolbar includes the buttons that enable to insert templates or preset layout into the expression code area upon their pressing.

Logical operations:

- **&&** — Logical "AND".
- **||** — Logical "OR".
- **!** — Logical "NOT".

Comparisons operator:

- **=** — Equals.
- **!=** — Not equal.
- **<** — Less than.

- **>** — Greater than.
- **<=** — Less than or equal to.
- **>=** — Greater than or equal to.

Templates:

- 9.0: `0.0` will be inserted to enter the real number.
- “”: `""` will be inserted to enter the string.
- 31: to enter the date, the object creation construction containing the current date will be inserted. For example: `new Date(2020, 1, 5)`;
- 🕓: to enter the date/time, the object creation construction containing the current date/time will be inserted. For example: `new Date(2020, 1, 5, 13, 12, 50, 100)`.

Logical values:

- **false** — False.
- **true** — True.