# RVL

## About



RVL stands for **Rapise Visual Language**. It is inspired by well known software testing methodologies *Keyword Driven Testing* and *Data Driven Testing*.

This section contains a review of current approaches and concepts to highlight the ideas behind RVL design. You don't need to read this section if you want to learn RVL. However you may need it if you want to understand how it compares to other approaches and why we believe it is not just yet another approach but the way forward to diminish struggling while building real live UI Automation.

### Keyword Driven Testing

Keywoard Driven Testing separates the documentation of test cases -including the data to use- from the prescription of the way the test cases are executed. As a result it separates the test creation process into two distinct stages: a design and development stage, and an execution stage.

| A | B | C | D |
|---|---|---|---|
| . | *First Name* | *Last Name* | *Age* |
| Enter Patient | John | Smith | 45 |
| Enter Patient | Sarah | Connor | 32 |

*Keyword Driven Testing*: Column *A* constains a *Keyword*, columns *B*, *C*, *D* provide parameters for a *Keyword*.

### Data Driven Testing

Data Driven Testing is the creation of test scripts to run together with their related data sets in a framework. The framework provides re-usable test logic to reduce maintenance and improve test coverage. Input and result (test criteria) data values can be stored in one or more central data sources or databases, the actual format and organization can be implementation specific.

| A | B | C |
|---|---|---|
| *First Name* | *Last Name* | *Age* |
| John | Smith | 45 |
| Sarah | Connor | 32 |

*Data Driven Testing*: We have test input and expected output in data sources.

### Gherkin / Cucumber

There are known approaches intended to make scripting more close to spoken languages.

This is a very wise approach improving test readability. The test case is described in Gherkin - business readable, domain specific language. It describes behavior without detailing how that behavior is implemented.

Essential part of this framework is implementation of Given-When-Then steps that should be done with one of the common programming languages. Here is the place where the need in scriping skills are still required.

### Why RVL?

Initially Rapise has everything to build *Data Driven* and *Keyword Driven* test frameworks. Even without RVL.

It is possible do define *scenarios* or *keywords*, connect to *Spreadsheet* or *Database* and build the test set.

Framework based approaches require one to split data from test logic and maintain them separately. So: * When *AUT* or *SUT* changes (new theme, new widget, new layout) then test logic is updated and data stays the same * When test scenarios are enriched or updated then test logic is kept intact and only data sheets are updated.

The reality of this approach leads to some challenges. These challenges are common for all test frameworks mentioned here.

1.  Design of test scripts require scripting and programming skills. That person is likely to be a programmer.

2.  Design of good test data requires knowledge in target domain. For example, if you application is for Blood Bank then one should have some medical skills. If it is some device control app, then you should have engeneering knowledge about physical limitations of the device.

So in ideal world there are two persons working as a team: UI Automation scripting expert and target domain specialist.

In reality we see that due to real life limitations it is common that all scripting and test data is done by one person. It is either a programmer who gets familiar with target AUT domain or analyst who has some scripting skills.

### Reasons for struggling

There are several reasons that make a learning curve longer and adoption harder.

#### Syntax Sugar

We found a reason why people get stuck while trying to implement a test case.

Most of programming languages including *JavaScript* were designed by people with mathematical background. So this statement appears clear and simple for a programmer:

```
Deposit('John', 'O\'Connor', 17.99);
```

Programmer easily reads this as:

```
Deposit $17.99 to John O'Connor
```

So what is the difference between these notations? We found that the first and most important difficulty lays in so called *syntactical sugar*. Symbols ` ' " ; , . ( ) [ ] { } & $ % # @` do have meaning for language notation however are not important for understainding the matter.

This is true even for programmers. When switching from similarly looking languages some differences easily cause frustration. For example, the same construct:

```
$a = "Number " + 1;
```

Means text concatenation in *JavaScript*, however the same is mathematical operation in *PHP*.

Comparison like:

```
if( value == "OK" )
```

Is good for *JavaScript* or *C#* world and leads `false` results in *Java*.

So even if we have programming skills it is still a problem to switch from one language to another and may produce potential issues.

#### Data Tables

With Keywword Driven and Data Driven approach we get a table that represents a sequence. Sequence of patients to proceed, sequence of user logins etc.

And sometimes we feel the lack of common debugging facilities: - run keyword for only one line, - start from specific row, - or stop before processing specific line.

So here we get to a point where the table should better be a part of the script rather than just external data source.

#### State of The Art

RVL reflects a common trend in programming languages where computational power and flexibility are sacrificed towards clarity and readability.

Some language is reduced to a reasonable subset in the sake of more concise and focused presentation. Just couple of examples.

Jade template engine simplifies writing HTML pages by clearing syntax sugar (`< > / %`) so HTML code:

```
<body> <p class="greeting">Hello, World!</p> </body>
```

Gets reduced to more textual view:

```
body p.greeting Hello, World!
```

Go language is promoted as *Go is expressive, concise, clean, and efficient.*. In fact its authors sacrificed many advanced features of common programming languages (classes, inheritance, templates) to get more clarity. This is extremely important because sophisticated features produce sophisticated problems that are hard to nail down. And if you deal with high-load distributed systems minor gain through use of unclear feature may lead to major unpredictable loss.

### RVL Concepts

RVL's goal is to minimize the struggling.

1. We assume that one should have minimal care about the syntax sugar and syntax rules. This means that we must avoid braces, quotes or any special symbols ' " ; , . ( ) [ ] { } & $ % # @ and make it possible to maintain the script without them.

2. We want script to be close to *Keyword Driven* and *Data Driven* testing concept. So test data and test results should be representable as data tables. This reduces the struggling of attaching the data feed to a test set.

3. We still want to have a solid language. We seek for a balance between clarity and power of language. So we want the script to be implemented on the same language. Both keyword, scenarios and data feeds should be done in a same way. This means one RVL skill is requried for everything.

4. In many cases grids or tables are used to represent test data. So we want the script itself to be a grid. So all parts of it includeing data tables are debuggable as a part of the solid script.

5. When we think about working with table data the most common format that comes to our mind is XLS, XLSX or CSV. These formats are supported by powerful tools that make it easier to prepare data for feeding into the test set. So RVL is itself an .xls spreadsheet so its logic is expressed right there.

6. Even with Spreadsheet there is a question what may be entered into the particular cell. With RVL we have an editor where you start from left to right and each cell has limited number of options. So if you don't know language it will guide you.

## Columns

RVL script is a spreadsheet containing set of 7 columns in fixed order:

| | Flow | Type | Object | Action | ParamName | ParamType | ParamValue | H |
|---|---|---|---|---|---|---|---|---|
| 1 | Flow | Type | Object | Action | Param Name | Param Type | Param Value | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 ▶ | | Action ▼ | Global | DoLaunch | cmdLine | string | calc | |
| 5 | | Param | | | wrkDir | string | . | |
| 6 | | Param | | | attachIfExists | boolean | true | |
| 7 | | Param | | | attachToWindow | string | Calculator | |
| 8 | # | My scenario goes here | | | | | | |
| 9 | | Action | _1 | DoLClick | x | number | 18 | |
| 10 | | Param | | | y | number | 15 | |
| 11 | | Action | Add | DoLClick | x | number | 21 | |
| 12 | | Param | | | y | number | 19 | |
| 13 | | Action | _2 | DoLClick | x | number | 14 | |
| 14 | | Param | | | y | number | 13 | |
| 15 | | Action | Equals | DoLClick | x | number | 12 | |
| 16 | | Param | | | y | number | 23 | |
| 17 | | | | | | | | |

*Column View*

- 1st *Flow* -- Control flow. This column dedicated to specifying structural information such blocks, Branches (If-Else), loops.

  Also it contains information about single row and multi row comments. Possible values are limited by the list:

- `\#` or `//` - single row comment

- `/*` - begin of multi row comment (comment is valid up to line starting with `*/`)

- `*/` - end of multi row comment started earlier from `/*`

- `If` - conditional branch. Row type must be `Condition`. The row may be followed with one or more `ElseIf` statements, zero or one `Else` statement and then should end with `End`.

- 2nd *Type* - Type of operation specified in this row. One of:

- `Action` - row defines an action. Action is a call for operation for one of the objects. Object is defined in the next column. See Actions.

- `Param` - signals that this row contains action parameter or condition parameter defined in last 3 columns (`ParamName`, `ParamType` and `ParamValue`).

- `Output` - this type of row must go after last Param for an action and defines a variable that should accept output value retured from the call to the Action.

- `Variable` - this row defines or assigns value to a local or global variable. See Variables.

- `Assert` - first row for the Assertion. See Assertions.

- `Condition`

- 3rd *Object* - Id of the object to be used for action. Rapise provides set of predefined global objects and objects recorded/learned from the AUT.

- 4th *Action* - One of the actions. `DoAction`, `DoClick`, `GetText` etc.

- 5th *ParamName* - see Params for more information on last 3 columns

- 6th *ParamType*

- 7th *ParamValue*

In addition to these columns there may be any number of other columns used for storing supplementary data, comments, calculations, thoughts etc. Additional columns may be utilized for script itself (i.e. contain expected values or reference data).

## Comments

### Single Row Comments

RVL has two types of single line comments depending on the purpose.

Sometimes comment is used to exclude line of code from execution.

| | Flow | Type | Object | Action | ParamName | ParamType | ParamValue | H |
|---|---|---|---|---|---|---|---|---|
| 2 | | | | | | | | |
| 3 | // | Action | ⊙ Global | DoLaunch | cmdLine | string | calc.exe | |

There is a special type of single row comments intended to put long text comments into the document.

Single row comment is displayed as long text providing that: 1. Flow is set to `#` or `//` 2. Text is completely typed into the `Type` cell. 3. Other cells after `Type` are empty.

In such case the text is displayed through the whole line:

| 10 | | | | | | | |
|---|---|---|---|---|---|---|---|
| 11 | # | My scenario goes here. We are going to perform arithmetical operation with Calculator. | | | | | |
| 12 ▸ | | Action | ▾ ▣_1 | DoLClick | x | number | 18 |
| 13 | | Param | | | y | number | 15 |

### Multiple Row Comments

Used to disable several rows of script:

| 28 | | | | | | | |
|---|---|---|---|---|---|---|---|
| 29 | /* | | | | | | |
| 30 | | Assert | | | message | string | TBD |
| 31 | | Action | ⊙ Global | GetCurrentDir | | | |
| 32 | | Condition | | output IsTrue | | | |
| 33 | */ | | | | | | |

## Conditions

Conditions used in `If` and `Assert` statements.

### Types of Conditions

Condition accepts one or two [Params](#).

1. There might be just one *Param*. Such condition is called *unary*, for example `param1 is true` or `output1 is true`.

2. There might be second *Param*. Such condition is called *binary*, for example `param1 == param2`.

3. Condition parameter may be either *Param* or *Action* output.

4. *Param* is some fixed *value*, *variable* or *expression*.

Binary condition with two *Param*s named `param1` and `param2`:

| ... | Type | ... | Action | ParamName | ... |
|---|---|---|---|---|---|
| | Param | | | param1 | |
| | Condition | | *param1 == param2* | | |
| | Param | | | param2 | |

Binary condition with *Action* and *Param* named `output1` and `param2`:

| ... | Type | Object | Action | ParamName | ... |
|---|---|---|---|---|---|
| | Action | MyButton | GetText | | |
| | Condition | | *outpu1 == param2* | | |
| | Param | | | param2 | |

Binary condition with two *Action*s named `output1` and `output2`:

| ... | Type | Object | Action | ParamName | ... |
|---|---|---|---|---|---|
| | Action | MyButton1 | GetText | | |
| | Condition | | *outpu1 != output2* | | |
| | Action | MyButton2 | GetText | | |

Unary condition with *Param* param1:

| ... | Type | ... | Action | ParamName | ... |
|---|---|---|---|---|---|
| | Param | | | param1 | |
| | Condition | | *param1 IsFalse* | | |

Unary condition with *Action* output1:

| ... | Type | Object | Action | ParamName | ... |
|---|---|---|---|---|---|
| | Action | MyButton | GetEnabled | | |
| | Condition | | *outpu1 IsTrue* | | |

## All Conditions

### Unary conditions with *Param*

| Caption | Description |
|---|---|
| param1 IsTrue | Check if param1 is true |
| param1 IsFalse | Check if param1 is false |
| param1 IsNull | Check if param1 is null |
| param1 IsNotNull | Check if param1 is NOT null |
| param1 IsSet | Check if param1 is NOT null, false, 0, empty string or undefined |
| param1 IsNotSet | Check if param1 is null, 0, false, empty string or undefined |

### Unary conditions with *Action*

| Caption | Description |
|---|---|
| output1 IsTrue | Check if output1 is true |
| output1 IsFalse | Check if output1 is false |
| output1 IsNull | Check if output1 is null |
| output1 IsNotNull | Check if output1 is NOT null |
| output1 IsSet | Check if output1 is NOT null, false, 0, empty string or undefined |
| output1 IsNotSet | Check if output1 is null, 0, false, empty string or undefined |

### Binary conditions with *Param*s

| Caption | Description |
|---|---|
| param1 == param2 | Check if param1 equals to param2 |
| param1 != param2 | Check if param1 NOT equal to param2 |
| param1 > param2 | Check if param1 is more than param2 |
| param1 >= param2 | Check if param1 is more or equal to param2 |
| param1 <= param2 | Check if param1 is less or equal to param2 |
| param1 < param2 | Check if param1 is less than param2 |
| param1 contains param2 | Check if param1 contains param2 as substring |
| CmpImage param1, param2 | Compare 1st image and image represented by param2 |

### Binary conditions with *Action* and *Param*

| Caption | Description |
|---|---|
| output1 == param2 | Check if output1 equals to param2 |
| output1 != param2 | Check if output1 NOT equal to param2 |
| output1 > param2 | Check if output1 is more than param2 |
| output1 >= param2 | Check if output1 is more or equal to param2 |
| output1 <= param2 | Check if output1 is less or equal to param2 |
| output1 < param2 | Check if output1 is less than param2 |
| output1 contains param2 | Check if output1 contains param2 as substring |
| CmpImage output1, param2 | Compare 1st image and image represented by param2 |

### Binary conditions with *Action*s

| Caption | Description |
|---|---|
| output1 == output2 | Check if output1 equals to output2 |
| | |

| | |
|---|---|
| output1 != output2 | Check if output1 NOT equal to output2 |
| output1 > output2 | Check if output1 is more than output2 |
| output1 >= output2 | Check if output1 is more or equal to output2 |
| output1 <= output2 | Check if output1 is less or equal to output2 |
| output1 < output2 | Check if output1 is less than output2 |
| output1 contains output2 | Check if output1 contains output2 as substring |
| CmpImage output1, output2 | Compare 1st image and image represented by output2 |

### *And*, *Or* **Conditions**

It is possible to make more complex conditions by using *And* and *Or* keyword in the *Flow* column.

| Flow | Type | ... | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| If | Param | | | param1 | *variable* | Result1 |
| | Condition | | *param1 IsFalse* | | | |
| **And** | Param | | | param1 | *variable* | Result2 |
| | Condition | | *param1 IsTrue* | | | |
| ... | ... | | ... | ... | ... | ... |

This pice forms a condition checking that Result1 is false AND Result2 is true at the same time.

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| If | Action | MyButton | GetEnabled | | | |
| Condition | | *output1 IsFalse* | | | | |
| **Or** | Param | | | param1 | *variable* | Result1 |
| Condition | | *param1 IsTrue* | | | | |
| ... | ... | | ... | ... | ... | ... |

This pice forms a condition checking that *MyButton* is Enabled OR Result2 is true at the same time.

### **Examples**

Condition is never used alone. You may find examples of conditions in chapters devoted to [Assertions](#) and [If-Then-Else](#).

## Actions

In RVL Action always refers to an operation performed with object.

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| | Action | MyButton | DoClick | x | number | 5 |
| | Param | | | y | number | 7 |

If row type is Action then there must be *Object* and *Action* cells defined.

**Note**: In this example we call an operation that would look in JavaScript as follows:

```
SeS('MyButton').DoClick(5,7);
```

Object is an ID of learned or Global object. Available objects may be found in the Object Tree:
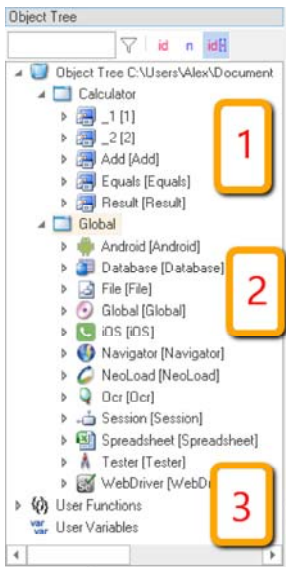
*Object tree* contains list of available objects, including: 1. *Local objects* (1) learned recorded or learned from the application under test. 2. *Global object*. Always available set of obj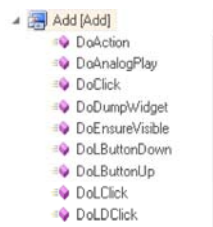ects containing most common utility functions and operations. 3. *Functions*. Represent global JavaScript functions. Each time you define a global function in .user.js file it becomes available for calling from RVL with special object ID Functions.



Each Object has its own set of actions. You may also see them in the object tree:



An *Action* may have any number of parameters. See Params for more info.

### Editing Action

An Action may have both mandatory and optional params. When action is selected from the dropdown its params are displayed:



By default RVL editor pre-fills only mandatory params for you when you select an action from the dropdown. In this example DoLaunch has one mandatory parameter cmdLine so here is what you get when you select it:



However the situation is differs if you hold the **Shift** key while choosing an Action from the dropdown:

You may see that all parameters are applied in this case.

- **Note:** if you you already have have the same action and select it with **Shift** key again, no optional params are applied. You need to clean the *Action* cell and re-select it with **Shift** if you want to achieve the desired effect.

### Examples

Action without parameters

| | | |
|---|---|---|
| Action | A Home | DoClick |

Action with single parameter. In RVL each parameter takes one line with *Action*=Param. However for the 1st param there is an exception. It may occupy the same line as Action itself:

| | | | |
|---|---|---|---|
| Action | abl Username_ | DoSetText | txt |

Action with many parameters:

| 39 | Action | Global | DoLaunch | cmdLine | string | |
|---|---|---|---|---|---|---|
| 40 | Param | | | wrkDir | string | . |
| 41 | Param | | | attachIfExists | boolean | false |
| 42 | Param | | | attachToWindow | string | null |

## Variables

In RVL, variables are useful for storing intermediate results as well as accessing and passing global values to external *JavaScript* functions.

Variables may be used in Params to Conditions and in Actions.

### Declaring and Assigning

This line declares a variable without any values. Its value may be assigned later:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| | **Variable** | | | MyVar1 | | |

This line declares and assigns value *5* to a variable MyVar2:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| | **Variable** | | | MyVar2 | *number* | 5 |

If the variable is declared earlier, then assignment just changes its value. If the variable is not yet declared, then assignment is actually a declaration with assignment.

### Using

Any Params value may accept a *variable*:

| ... | Type | ... | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|
| ... | Param | | text | *variable* | MyVar1 |

Any Params value may accept an *expression* using variables:

| ... | Type | ... | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|
| ... | Param | | text | *expression* | MyVar2 + 4 |

Any Action may write its return value to a variable using the *Output* statement:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| | Action | Global | DoTrim | str | string | text to trim |
| | Output | | | | variable | MyVar1 |

The Output value may then be used as a param value in actions, conditions, assertions and expressions.

### Local Variables

By default declared variables are assumed to be local. Local variables may be used only within the current RVL script and not visible from other RVL scripts or *JavaScript* code.

### Global Variables

You may have a *JavaScript* variable defined in the user *Functions* file (`*.user.js`), i.e.:

```
// Piece from MyTest1.user.js var globalVar = "Value";
```

Then in the RVL you may declare `globalVar` as global and access it (read or assign values). Declaring a variable as global is simple:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
|      | Variable | | Global | globalVar | | |

Global variables are useful for exchanging and/or sharing data between different RVL scripts or between *RVL* and *JavaScript*.

**Variable Actions**

One may use an expression to change the value of a variable. Here are several common variable operations that may be used to modify variable values:

1. *Increment* is an operation where numeric value is increased by `1` or any other specified value. The variable must have a numeric value. Otherwise the result is `NaN`.

   If no param to *Increment* is specified then `1` is assumed:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
|      | Variable | | Increment | numVar | | |

Otherwise it is any *value*:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
|      | Variable | | Increment | numVar | number | value |

2. *Decrement* is the same as increment but the value is subtracted from the variable.

3. *Append* adds the value as text to the specified variable. This operation is useful for constructing text messages:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
|      | Variable | | Append | textVar | string | Final value: |
|      | Variable | | Append | textVar | variable | numVar |

In this example if `textVar` was empty and `numVar` had value `5` then the final value of `textVar` is the following text: `Final value: 5`

**Examples**

Variables may be declared as *Local* or *Global*. Declaration may or may not contain initial value

| Declare global variables. If it is assigned earlier then keep its value | | | |
|------|------|------|------|
| Variable | | Global | g_bookName | |
| Declare global variable and assign its value | | | |
| Variable | | Global | g_genre | string |
| Declare local variable witout value | | | |
| Variable | | Local | OsVersion | |
| Declare local variables and assign initial values | | | |
| Variable | | Local | StringVar | string |
| Variable | | Local | NumVar | number |
| Variable | | Local | BoolVar | boolean |

Variables may accept output from the *Action*:

| Declare local variable witout value | | | |
|---|---|---|---|
| Variable | | Local | OsVersion |
| Action | ⊙ Global | GetOsVersion | |
| Output | | | variable |

Variables may be used as input to the *Action*:

| Use variable as a parameter | | | | |
|---|---|---|---|---|
| Action | ⚠ Tester | Message | message | variable |

## Assertions

*Assert* is an essential operation for testing and validation. RVL provides special structure for it to make it more readable.

Assertion has 2 parts: 1st row is Assert containing assertion message and then goes Condition:

| ... | Type | ... | Action | ParamName | ... |
|---|---|---|---|---|---|
| | Assert | | | message | string |
| | Param | | | param1 | |
| | Condition | | condition statement | | |
| | Param | | | param2 | |

Assertion first line is always the same except the *Param Value*.

In RVL Action always refers to an operation performed with object.

| ... | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| | Assert | | | message | string | Assertion text to be displayed in the report |
| | Param | | | param1 | string | Text1 |
| | Condition | | param1!=param2 | | | |
| | Param | | | param2 | string | Text2 |

**Examples**

Compare object property *InnerText* with expected value:

| Verify that: InnerText=Sister Carrie | | | |
|---|---|---|---|
| Assert | | | message |
| Action | ☐ Sister_Carrie | GetInnerText | |
| Condition | | output1 == param2 | |
| Param | | | param2 |

Check if object exists on the screen:

| Check that object 'Sister_Carrie' exists | | | |
|---|---|---|---|
| Assert | | | message |
| Action | ⊙ Global | DoWaitFor | objectId |
| Condition | | output1 IsSet | |

Check if variable `Age` has value '74':

| *Check that variable Age contains value '74'* | | |
|---|---|---|
| Assert | | message |
| Param | | param1 |
| Condition | param1 == param2 | |
| Param | | param2 |

## If-Else

If using for branching statements in RVL.

Basic branch statement has 2 parts: 1st row is If flow with [Condition](#):

**If**

| Flow | Type | ... | Action | ParamName | ... |
|---|---|---|---|---|---|
| If | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | **some** | **actions** | **go** | **here** | |
| End | | | | | |

Actions after If condition and up to End statement are executed when condition is truth.

### If-Else

If-Else statement is similar to If with one extension. It contains an alternative Else section that is executed when If condition is false:

| Flow | Type | ... | Action | ParamName | ... |
|---|---|---|---|---|---|
| If | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | **some** | **actions** | **go** | **here** | |
| Else | | | | | |
| | **other** | **actions** | **go** | **here** | |
| End | | | | | |

### If-ElseIf

ElseIf is a way to establish a chain of conditions. Each condition is evaluated with previous is false.

If-Else statement is similar to If with one extension. It contains an alternative Else section that is executed when If condition is false:

| Flow | Type | ... | Action | ParamName | ... |
|---|---|---|---|---|---|
| If | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | **some** | **actions** | **go** | **here** | |
| ElseIf | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | **other** | **actions** | **go** | **here** | |
| End | | | | | |

There may be many ElseIf` blocks:

| Flow | Type | ... | Action | ParamName | ... |
|---|---|---|---|---|---|
| If | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | **some** | **actions** | **go** | **here** | |
| ElseIf | Param | | | param1 | |
| | | | | | |

| Flow | Type | ... | Action | ParamName | ... |
|------|------|-----|--------|-----------|-----|
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | *other* | *actions* | *go* | *here* | |
| ElseIf | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | *other* | *actions* | *go* | *here* | |
| End | | | | | |

And there might also be an `Else` block in the end:

| Flow | Type | ... | Action | ParamName | ... |
|------|------|-----|--------|-----------|-----|
| If | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | *some* | *actions* | *go* | *here* | |
| ElseIf | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | *other* | *actions* | *go* | *here* | |
| ElseIf | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | *other* | *actions* | *go* | *here* | |
| Else | | | | | |
| | *other* | *actions* | *go* | *here* | |
| End | | | | | |

**Examples**

Check if `Log In` link available. If so, do login:

| If | Action | ⊙ Global | DoWaitFor |
|----|--------|----------|-----------|
| | Condition | | output1 IsSet |
| # | *If actions* | | |
| | Action | Log_In | DoClick |
| | Action | Username_ | DoSetText |
| | Action | Password_ | DoSetText |
| | Action | ctl00$MainContent$LoginUser$Logi | DoClick |
| End | | | |

Check if we use old version of OS and assign a variable `OldWindows` accordingly:

| | Variable | | Local | OldWindows |
|---|---|---|---|---|
| If | Action | ⊙ Global | GetOsType | |
| | Condition | | output1 contains param2 | |
| | Param | | | param2 |
| # | *If actions* | | | |
| | Variable | | | OldWindows |
| Else | | | | |
| # | *Else actions* | | | |
| | Variable | | | OldWindows |
| End | | | | |

## Parameters

The last 3 columns in the RVL table are used for passing parameters:

| ... | *ParamName* | *ParamType* | *ParamValue* |
|---|---|---|---|
| ... | text | string | John Smith |
| ... | x | number | 5 |
| ... | y | number | 7 |
| ... | forceEvent | boolean | true |

- 5th column - *ParamName* - name of the parameter. This column's intention is readability and it does not affect execution. However it names input parameters and makes it easier to understand each provided input option.

- 6th column - *ParamType* - value type. This may be a basic scalar type (`number`, `string`, `boolean`, `object`) as well as one of the following additionals 'special' types:

  - `expression` - any valid JavaScript expression that may involve global variables and functions and local variables.

  - `variable` - the parameter value is read from a variable.

  - `objectid` - ID of one of the learned Objects.

- 7th column - *ParamValue* - a value that is acceptable for the specified *ParamType*. For `boolean` it is `true` or `false`. For `number` is is any floating point number (i.e. `3.14`). For `string`, any text without quotes or escape signs.

### Param Rows

In RVL each parameter takes one row:

| ... | *Type* | ... | *ParamName* | *ParamType* | *ParamValue* |
|---|---|---|---|---|---|
| ... | Param | | text | string | John Smith |
| ... | Param | | x | number | 5 |
| ... | Param | | y | number | 7 |
| ... | Param | | forceEvent | boolean | true |

### Param Arrays

Some methods accept arrays of values as input values. For example `Tester.Message` may take its 1st `message` parameter as an array and prints them combined. Making an array is easy, several consequent parameters having the same name are combined into an array, i.e.:

| *Flow* | *Type* | *Object* | *Action* | *ParamName* | *ParamType* | *ParamValue* |
|---|---|---|---|---|---|---|
| | Action | Tester | Message | **message** | string | MyVar1 value: |
| | Param | | | **message** | variable | **MyVar1** |
| | Param | | | **message** | string | MyVar2 value: |
| | Param | | | **message** | variable | **MyVar2** |

Should report a message like:

```
MyVar1 value: 25 MyVar2 value: 33
```

**Mixed Rows**

In some cases it is convenient to mix parameter cells with an *Action* or *Condition*.

For example, the 1st parameter of an *Action* may share the `Action` row:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
|      | Action | MyButton | DoClick | **x** | **number** | **5** |
|      | Param |        |        | y | number | 7 |

And this is equivalent to putting it in the next row:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
|      | Action | MyButton | DoClick |         |           |            |
|      | Param |        |        | **x** | **number** | **5** |
|      | Param |        |        | y | number | 7 |

Or `param2` of the [condition](#) may be on the same row:

| ... | Type | Object | Action | ParamName | ParamType | ParamValue |
|-----|------|--------|--------|-----------|-----------|------------|
|     | Param |       |        | param1 | string | Text1 |
|     | Condition |   | param1!=param2 | **param2** | **string** | **Text2** |

Which is equivalent to:

| ... | Type | Object | Action | ParamName | ParamType | ParamValue |
|-----|------|--------|--------|-----------|-----------|------------|
|     | Param |       |        | param1 | string | Text1 |
|     | Condition |   | param1!=param2 |       |           |            |
|     | Param |       |        | **param2** | **string** | **Text2** |

This allows saving space while keeping same readability.

**Map Params**

If map is defined in the script it may be used directly as a parameter. *ParamType* should be set to Map Name and *ParamValue* is a column (or row) name:

| Flow | Type | Object | Action | ParamName |
|------|------|--------|--------|-----------|
| Map | Rows | Logins |        |           |
|      | **Login** | **Password** |        |           |
|      | John | pass1 |        |           |
|      | Sarah | pass2 |        |           |
| End  |      |        |        |           |
|      |      |        |        |           |
|      | Action | ⋀ Tester | Message | message |

# Maps

A *Map* is designed to be an easy way to define tables of data. Items in the map may be accessed by name (if defined) or by index.

The indexed dimensions in the map may also be iterated by the [Loop][Loops.md] function, thus making it useful feature for Data-Driven Testing.

| Flow | Type | Object | Action |
|------|------|--------|--------|
|      |      |        |        |
| Map  | Rows | Logins |        |
|      | **Login** | **Password** |        |
|      | John | pass1 |        |
|      | Sarah | pass2 |        |
| End  |      |        |        |

An RVL script has at least 7 columns. However the *Map* may take as many columns as needed.

**Map Definition**

Typical declaration of map looks like:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
| Map  | MapType | **MapName** |   |           |           |            |
| ...  | ...  | ...    |        |           |           |            |
| End  |      |        |        |           |           |            |

Where MapType is either inplace: *Table*, *Rows*, *Columns*, or external: *Range* or *Database*.

**In-place maps**

In-place map data is defined right in the RVL script. In-place map rows may be selected using *This* flow or skipped with a Comment. So in-place maps serve as a part of the executable script.

- *Table*

- *Rows*

- *Columns*

**External maps:**

- *Range*

- *Database*

External maps are defined in an external spreadsheet, file or a database.

**Using Maps**

Once map is defined it may be used as a regular Object.

| Map | Rows | Logins | | |
|-----|------|--------|---|---|
| | **Login** | **Password** | | |
| | John | pass1 | | |
| | Sarah | pass2 | | |
| End | | | | |
| | | | | |
| | Action | Logins | ⊘ ⌄ | |

- ⦿ DoMoveToColumn
- ⦿ DoMoveToFirstColumn
- ⦿ DoMoveToFirstRow
- ⦿ DoMoveToLastColumn
- ⦿ DoMoveToLastRow
- ⦿ DoMoveToRow
- ⦿ DoSequential
- ⦿ GetCell

**Reading in a Loop**

See Loops part for Map type of loops.

**Maps Types**

**Rows Map**

A Rows Map is the most useful for data feeds. Each of the set of values is a row in a table that look like:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
| Map | Rows | **MapName** | | | | |
| | Col1 | Col2 | Col3 | Col4 | | |
| | val11 | val12 | val13 | val14 | | |
| | ... | | | | | |
| | ... | | | | | |
| End | | | | | | |

This and comments are specific features of the Rows Map. For example, only the 2nd row of data will be executed in this case:

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
| Map | Rows | **MapName** | | | | |
| | Col1 | Col2 | Col3 | Col4 | | |
| | ... | | | | | |
| This | ... | | | | | |
| | ... | | | | | |
| End | | | | | | |

Rows are designed to be iterated in a Loop

In real example it looks like this:

| Map | Rows | MyMap1 |
|---|---|---|
|  | **Login** | **Password** |
|  | John | testpass |
|  | Sarah | testpass |
| This | Jim | testpass |
|  | Peter | testpass |
|  | John | testpass |
|  | Fred | testpass |
| End |  |  |

[Comments](#) may also be used to skip specific rows or row sets.

**Columns Map**

A `Columns` Map is a convenient way for representing data when you have many options combined in few sets.

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| Map | Columns | **MapName** |  |  |  |  |
|  | **Row1** | ... |  |  |  |  |
|  | **Row2** | ... |  |  |  |  |
|  | **Row3** | ... |  |  |  |  |
| End |  |  |  |  |  |  |

The same may be represented as `Rows` but would require many columns and sometimes it is harder to read. So columns is ideal for storing configuration structures:

| Map | Columns | ConfigData |
|---|---|---|
|  | **Url** | http://localhost:8080/ |
|  | **Login** | testuser |
|  | **Password** | testpass |
|  | **Age** | 44 |
| End |  |  |

When a `Columns` Map is used in the Loop, then the iteration is performed through the columns and addresses the rows by name within the loop. I.e. the 1st iteration chooses 1st column, 2nd goes to 2nd column and so on.

**Table Map**

A Table map has both columns and rows named.

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|---|---|---|---|---|---|---|
| Map | Table | **MapName** |  |  |  |  |
|  |  | **Col1** | **Col2** | **Col3** | **Col4** |  |
|  | **Row1** | ... |  |  |  |  |
|  | **Row2** | ... |  |  |  |  |
|  | **Row3** | ... |  |  |  |  |
| End |  |  |  |  |  |  |

| Map | Table | TableMap | | | |
|---|---|---|---|---|---|
| | | | **Staging** | **QA** | **Prod** |
| | **Url** | | http://staging.myho... | http://qa.myhost.co... | http://myhost.com |
| | **User** | | test | qatest | john |
| | **Password** | | pass | pass | QAasd*&983 |
| | **Age** | | 33 | 33 | 33 |
| End | | | | | |

When a `Table` Map is used in the Loop, then the iteration is performed through the columns and addresses the rows by name within the loop. I.e. 1st iteration chooses 1st column, 2nd goes to 2nd column and so on.

It is convenient to use a `Table` Map when you have several columns and many rows so it perfectly fits into the screen. For example you may have several alternative configuration sections and want to use them depending on the situation. In the example below we have several sites (Testing, QA, Prod) each having own Url, Login etc. So we want to quickly switch between sites when working with test.

| Map | Table | TableMap | | | |
|---|---|---|---|---|---|
| | | | **Staging** | **QA** | **Prod** |
| | **Url** | | http://staging.myho... | http://qa.myhost.co... | http://myhost.com/ |
| | **User** | | test | qatest | john |
| | **Password** | | pass | pass | QAasd*&983 |
| | **Age** | | 33 | 33 | 33 |
| End | | | | | |
| | | | | | |
| | Action | TableMap | | DoMoveToColumn | colInd |
| | | | | | |
| | Action | Navigator | | Navigate | url |

**Range Map**

`Range` map contains no in-place data, but defines a region in the external spreadsheet to read information from.

| Map | Range | MyMap1 | | fileName | string |
|---|---|---|---|---|---|
| | Param | | | sheetName | string |
| | Param | | | fromRow | number |
| | Param | | | fromCol | number |
| | Param | | | toRow | number |
| | Param | | | toCol | number |
| End | | | | | |

A `Range` map definition contains a number of required parameters:

- *fileName* Path to file containing data. It may point to .xls, .xlsx or .csv file. If when it is empty we assume that data is stored in the same .rvl.xls spreadsheet as the script.

- *sheetName* Excel Sheet name. May be empty for .csv spreadsheets.

- *fromRow* 0-based index of the first row containing data. Usually first row is assigned as a header containing column names.

- *fromCol* 0-based index of the first column containing data.

- *toRow* final row index. If set to -1 then final row is detected automatically (as last row containing some data in the 1st column)

- *toCol* final column index. If set to -1 then final column is detected automatically as last column containing data in the 1st row.

Also there is a hidden parameter:

- *hasColumnNames* boolean. By default it is `true` meaning that 1st rows is assumed to contain column names. Once it is `false` the columns will have no names and may only be accessed by 0-based index.

Data in the `Range` map is assumed to be similar to `Rows` map, but defined externally. Looping is done by rows. Typical external file containing data may look like that:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Item1 | Operation | Item2 | Result |
| 2 | 15 | + | 13 | 28 |
| 3 | 5 | * | 6 | 30 |
| 4 | 19 | - | 3 | 16 |
| 5 | 8 | / | 4 | 2 |

**Database Map**

A `Database` map contains no in-place data, but defines a connection to the database result set.

| Map | Database | MyMap1 | | connectionString | string |
|---|---|---|---|---|---|
| | Param | | | query | string |
| End | | | | | |

The `Database` map definition contains two parameters:

- *connectionString* ADO connection string.

- *query* usually it is an SQL query to execute.

*connectionString* parameter allows accessing wide variety of different database sources. You may learn ore here: https://docs.microsoft.com/en-us/sql/ado/reference/ado-api/connectionstring-property-ado.

Some samples of typical ADO connection string values:

**Microsoft Access**

```
Provider=MSDASQL; Driver={Microsoft Access Driver (*.mdb)}; DBQ=C:\path\filename.mdb;
```
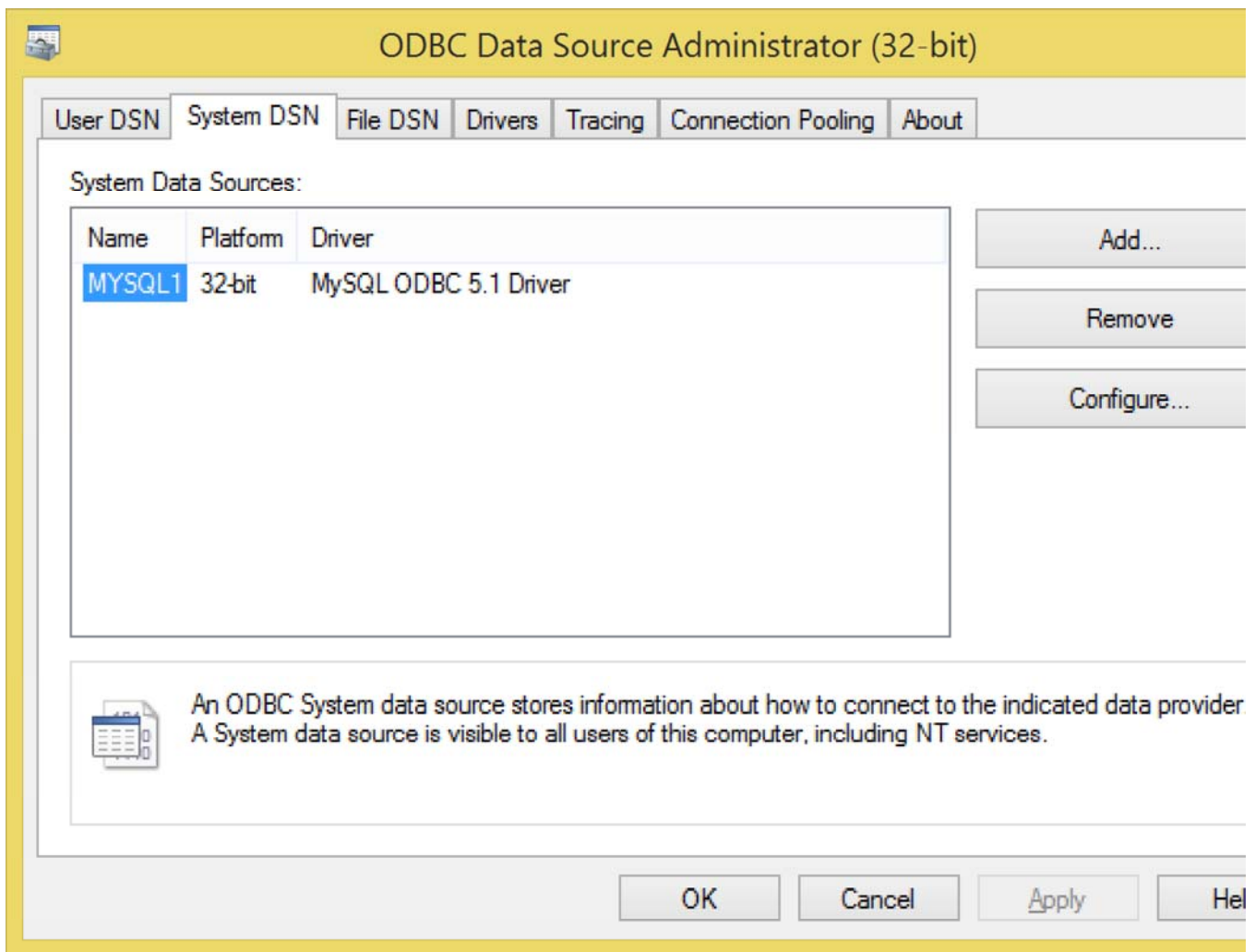
**Microsoft Excel**

```
Provider=MSDASQL; Driver={Microsoft Excel Driver (*.xls)}; DBQ=C:\path\filename.xls;
```

**Microsoft Text**

```
Provider=MSDASQL; Driver={Microsoft Text Driver (*.txt; *.csv)}; DBQ=C:\path\;
```

An example below refers to ODBC Data Source defined as follows:

## Loops

*Loops* serve several needs in RVL:

1. Iterate through Maps to make data-driven testing easier.

2. Allows you to repeat a set of actions for a given number of iterations.

3. Lets you repeat a loop body while some Condition is satisfied.

### Loop Map

A Map allows both reading script data from the table defined in the same script or from external data source such as spreadsheet, file or database. Once a Map is defined, the loop is the simplest way of traversing it.

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
| Loop | Map | *MapName* | | | | |
| | ... | | ... | ... | ... | ... |
| End | | | | | | |

Where *MapName* should be name of the map declared earlier in the same script.

The loop goes through either the map rows or through the map columns depending on the type of map:

- For Rows, Range or Database type of Map, the loop goes through rows. I.e. 1st iteration points to 1st Row, then 2nd iteration points to 2nd row etc.

- For Columns and Table types of Map, the iteration goes through the columns.

### Loop Variable

| Flow | Type | Object | Action | ParamName | ParamType | Param |
|------|------|--------|--------|-----------|-----------|-------|
| Loop | Variable | ind | | from | number | 1 |
| | Param | | | to | number | 10 |
| # | Loop body | | | | | |
| | | | | | | |
| End | | | | | | |

| Flow | Type | Object | Action | ParamName | ParamType | ParamValue |
|------|------|--------|--------|-----------|-----------|------------|
| Loop | **Variable** | *VarName* | | from | number | 1 |
| | Param | | | to | number | 10 |
| | ... | | ... | ... | ... | ... |
| End | | | | | | |

Where:

- *VarName* is an optional name of variable. It may be avoided if the goal is just to do specified number of iterations. If *VarName* is set, then the corresponding variable is assigned with the from value and incremented up to the to value throughout the loop. If *VarName* refers to an existing local or global variable then it is used, otherwise a local variable named *VarName* is created.

- *from* initial value of the loop variable

- *to* final value of the loop variable

- *step* optional, default is 1. Loop step to increment in each iteration.

### Loop Condition

| Loop | Param | | | | param1 | variable |
|------|-------|--|--|--|--------|----------|
| | Condition | | param1 < param2 | | | |
| | Param | | | | param2 | number |
| # | Loop body | | | | | |
| | | | | | | |
| End | | | | | | |

Loop repeats while condition is satisfied (i.e. while(someCondition)).

## RVL Object

RVL Object

Some common tasks related to script execution, such as calling scripts, executing separate sheets, returning, exiting and bailing out is served by RVL.

### Actions

### DoPlayScript

`DoPlayScript(/**String*/scriptPath, /**String*/sheetName)`

Play RVL script using specified

- scriptPath {/**string*/}: Path to script

- sheetName {/**string*/}: Excel sheet containing the script

### Exit

`Exit(/**String*/ message, /**Boolean*/isError)`

Break execution at the specified line

- message {/**string*/}: Exit message

- isError {/**boolean*/}: Specify 'false' if you want just exit without exit message

### Return

```
Return(/**String*/ message)
```

Return from specified line. This method should be called from within RVL

- `message` {/**string*/}: Return message

### DoPlaySheet

```
DoPlaySheet(/**String*/sheetName)
```

Run current script from specified sheet

- `sheetName` {/**string*/}: Sheet Name

## Properties

### CurrentScriptPath

** `GetCurrentScriptPath()` **

Return path to currently executed .rvl.xls file

### CurrentScriptSheet

** `GetCurrentScriptSheet()` **

Return sheet name of the currently executed .rvl.xls file

# Map Object

Map Object

Represents an RVL Map object and all its operations. The same operations are used by the RVL runtime implicitly to read the cell value or iterate through the Map.

## Actions

### DoMoveToRow

```
DoMoveToRow(/**Number*/ colInd)
```

Moves to a given row.

`rowInd` Row index (or name) to set active.

### DoSequential

```
DoSequential()
```

Advances to the next row in the range. The range is either set by SetRange or it is the default range that includes all rows on the sheet except first row which is considered to contain column names. When the end of the range is reached, DoSequential rewinds back to the first row in the range and returns 'false'.

Returns 'false' if being called when active row is the last row or the spreadsheet is not attached, 'true' - otherwise.

### DoMoveToColumn

```
DoMoveToColumn(/**Number|String*/colInd)
```

Moves to a given column.

`colInd` Column index (or name) to set active.

### DoMoveToFirstColumn

```
DoMoveToFirstColumn()
```

Moves to a first column in the map.

### DoMoveToFirstRow

```
DoMoveToFirstRow()
```

Moves to a first row in the map.

### DoMoveToLastColumn

```
DoMoveToLastColumn()
```

Moves to a last column in the map.

**DoMoveToLastRow**

`DoMoveToLastRow()`

Moves to a last row in the map.

## Properties

**Cell**

** `GetCell(/**Number|String*/ columnId, /**Number*/ rowId)` **

Gets a cell value by its coordinates. It returns the current cell value after DoSequental or DoRandom if the parameters are not set.

`[columnId]` Column index or name. If not set ActiveColumn is used.

`[rowId]` Row index. If not set ActiveRow is used.

**ColumnCell**

** `GetColumnCell(/**Number*/ rowId)` **

Gets cell value by its coordinates. Returns current cell value after DoSequental. If not set ActiveColumn is used.

`[rowId]` Row index. If not set ActiveRow is used.

**ColumnCount**

** `GetColumnCount()` **

Gets columns count.

Returns Number of columns in the spreadsheet.

**ColumnIndexByName**

** `GetColumnIndexByName(/**String*/name)` **

Gets column name.

`name` Column name.

Returns column index if found, or -1.

**ColumnName**

** `GetColumnName(/**Number*/ ind)` **

Gets column name.

`ind` Column index.

Returns Name of column in the spreadsheet.

**RowCount**

** `GetRowCount()` **

Gets rows count.

Returns Number of rows in the spreadsheet.

**RowIndexByName**

** `GetRowIndexByName(/**String*/name)` **

Gets row name.

`name` Row name.

Returns row index if found, or -1.

**CurrentRowIndex**

** `GetCurrentRowIndex()` **

Get zero based current row index.

**EOF**

** `GetEOF()` **

Is current position is beyond the map boundaries range.

**RowCell**

** `GetRowCell(/**Number|String*/ columnId)` **

Gets cell value for current row. Returns current cell value after DoSequental. ActiveRow is used.

`[columnId]` Column index or name. If not set ActiveColumn is used.

**RowName**

** `GetRowName(/**Number*/ ind)` **

Gets row name.

`ind` Row index.

Returns Name of row in the map.

**Value**

** `GetValue(/**Number|String*/ rowOrColumnNameOrId)` **

Gets cell value by its name or id. Returns current cell value after DoSequental. If it is Rows or Table then the parameter needs to be a column name or index, and ActiveRow is used. If it is Columns then the parameter needs to be a row name or index, and ActiveRow is used.

`[rowOrColumnNameOrId]` Row or Column index or Name.