Rapise®

# Rapise® Visual Language (RVL) User Guide

Version 5.1

Inflectra Corporation

Date: May 21st, 2017

inflectra®

# 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

Keyward 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.

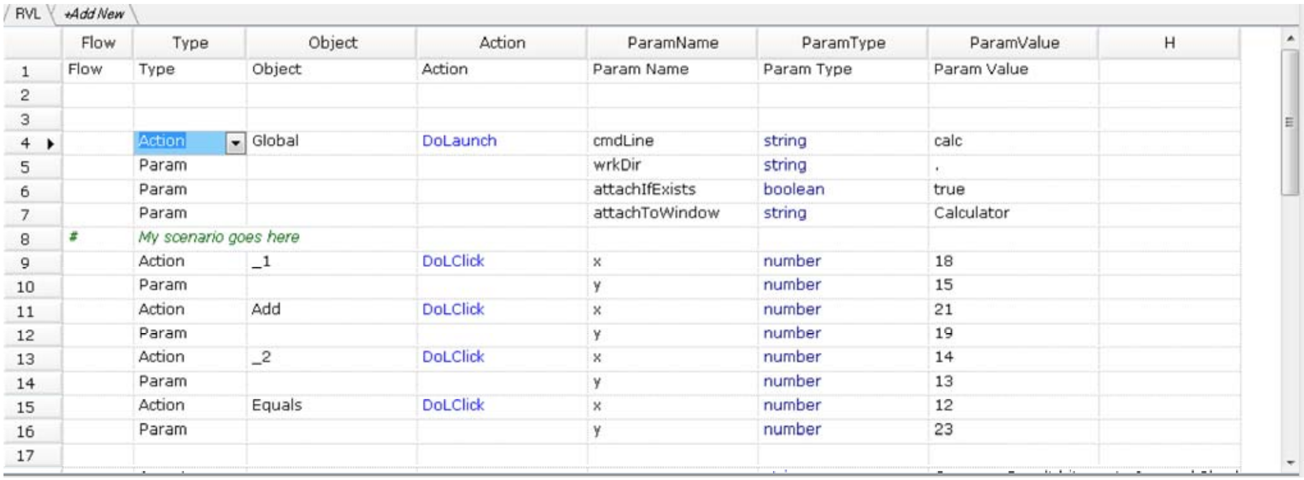
4. In many cases grids or tables are used to represent test data. So we want the script itself to be a grid. So all parts of it includeing data tables are debuggable as a part of the solid script.

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*

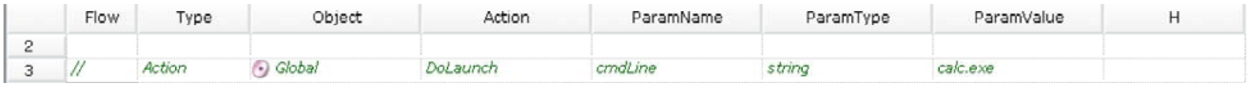
In addition to these columns there may be any number of other columns used for storing supplementary data, comments, calculations, thoughts etc. Additional columns may be utilized for script itself (i.e. contain expected values or reference data).

## 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.

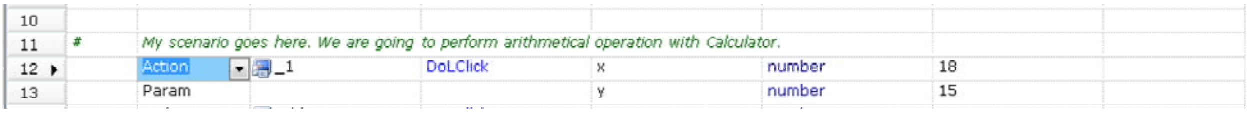


| | Flow | Type | Object | Action | ParamName | ParamType | ParamValue | H |
|---|---|---|---|---|---|---|---|---|
| 2 | | | | | | | | |
| 3 | // | Action | Global | DoLaunch | cmdLine | string | calc.exe | |

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.

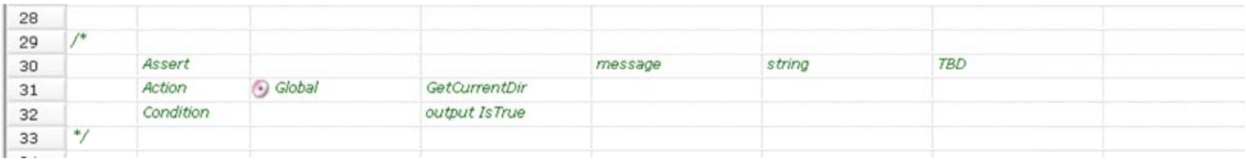
In such case the text is displayed through the whole line:



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 *Params* 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 *Actions* 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 *Params***

| *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 *Actions***

| *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.

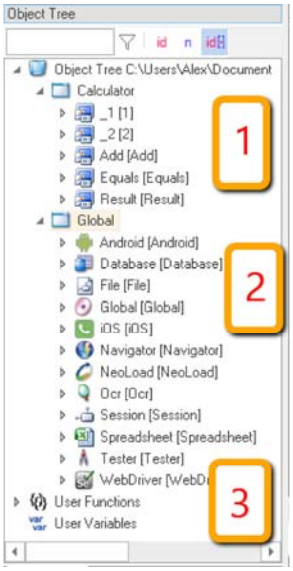
| *Flow* | *Type* | *Object* | *Action* | *ParamName* | *ParamType* | *ParamValue* |
|---|---|---|---|---|---|---|
| | Action | MyButton | DoClick | x | number | 5 |
| | Param | | | y | number | 7 |

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

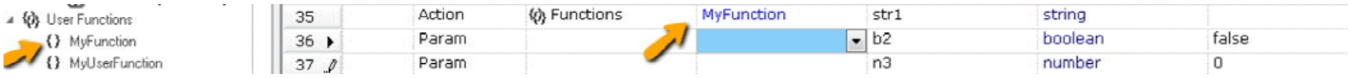
**Note**: In this example we call an operation that would look in JavaScript as follows:

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

`Object` is an ID of learned or Global object. Available objects may be found in the `Object Tree`:

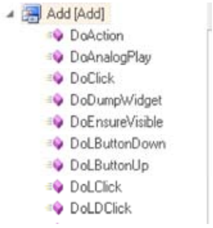


*Object tree* contains list of available objects, including: 1. *Local objects* (1) learned recorded or learned from the application under test. 2. *Global object*. Always available set of objects containing most common utility functions and operations. 3. *Functions*. Represent global JavaScript functions.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



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 35 | | Action | Functions | MyFunction | str1 | string | |
| 36 | | Param | | | b2 | boolean | false |
| 37 | | Param | | | n3 | number | 0 |

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

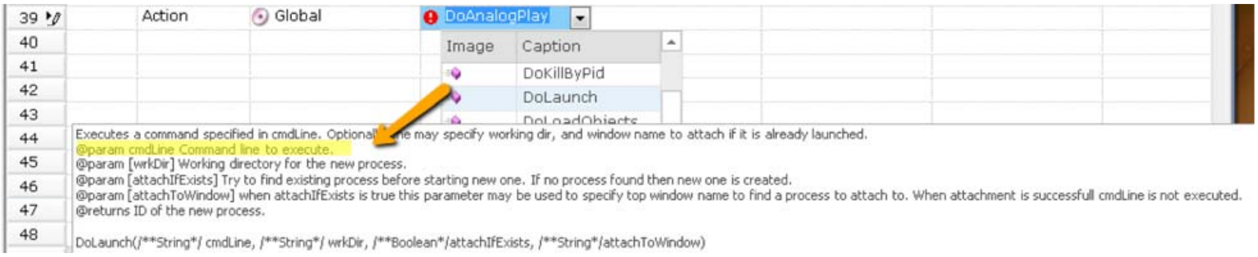


```
Add [Add]
    DoAction
    DoAnalogPlay
    DoClick
    DoDumpWidget
    DoEnsureVisible
    DoLButtonDown
    DoLButtonUp
    DoLClick
    DoLDClick
```

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:

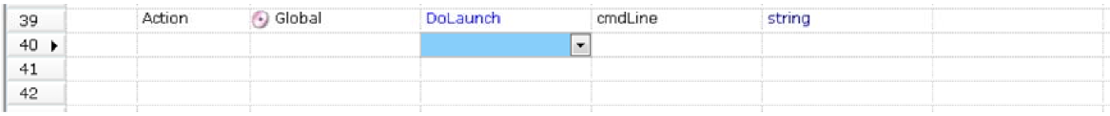


| | | | |
|---|---|---|---|
| 39 | Action | Global | DoAnalogPlay |

Executes a command specified in cmdLine. Optionally one may specify working dir, and window name to attach if it is already launched.
@param cmdLine Command line to execute.
@param [wrkDir] Working directory for the new process.
@param [attachIfExists] Try to find existing process before starting new one. If no process found then new one is created.
@param [attachToWindow] when attachIfExists is true this parameter may be used to specify top window name to find a process to attach to. When attachment is successfull cmdLine is not executed.
@returns ID of the new process.

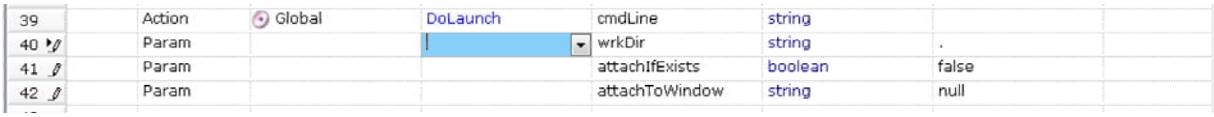
DoLaunch(/**String*/ cmdLine, /**String*/ wrkDir, /**Boolean*/attachIfExists, /**String*/attachToWindow)

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:



| | | | | | |
|---|---|---|---|---|---|
| 39 | Action | Global | DoLaunch | cmdLine | string |
| 40 | | | | | |
| 41 | | | | | |
| 42 | | | | | |

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



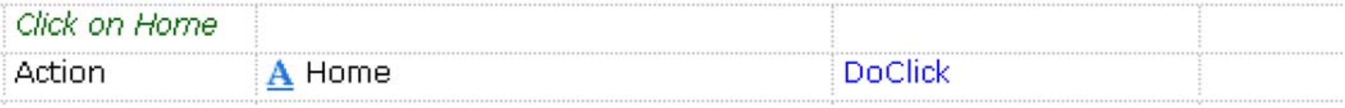
| | | | | | | |
|---|---|---|---|---|---|---|
| 39 | Action | Global | DoLaunch | cmdLine | string | |
| 40 | Param | | | wrkDir | string | . |
| 41 | Param | | | attachIfExists | boolean | false |
| 42 | Param | | | attachToWindow | string | null |

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.

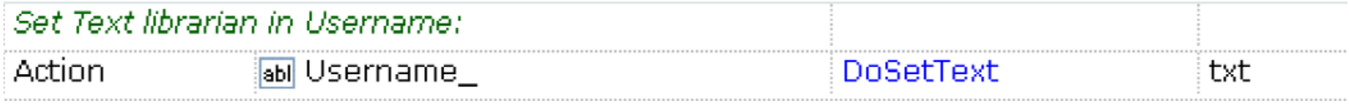
### Examples

Action without parameters



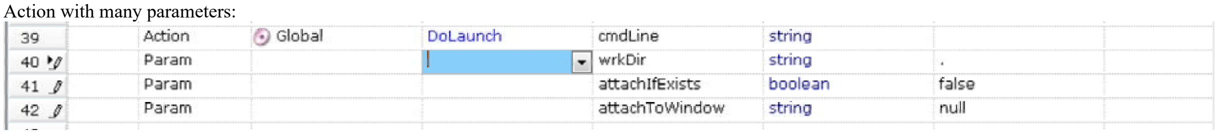
| | | | |
|---|---|---|---|
| Click on Home | | | |
| Action | Home | DoClick | |

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:



| | | | |
|---|---|---|---|
| Set Text librarian in Username: | | | |
| Action | 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

RVL variables 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 variable without any values. Its value may be assigned later:

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

This line and assigns value 5 to a variable `MyVar2`:

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

If variable is declared earlier then assignment just changes its value. If variable is not yet declared then assignment is actually 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 *Output* statement:

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

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

### Local Variables

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

### Global Variables

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

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

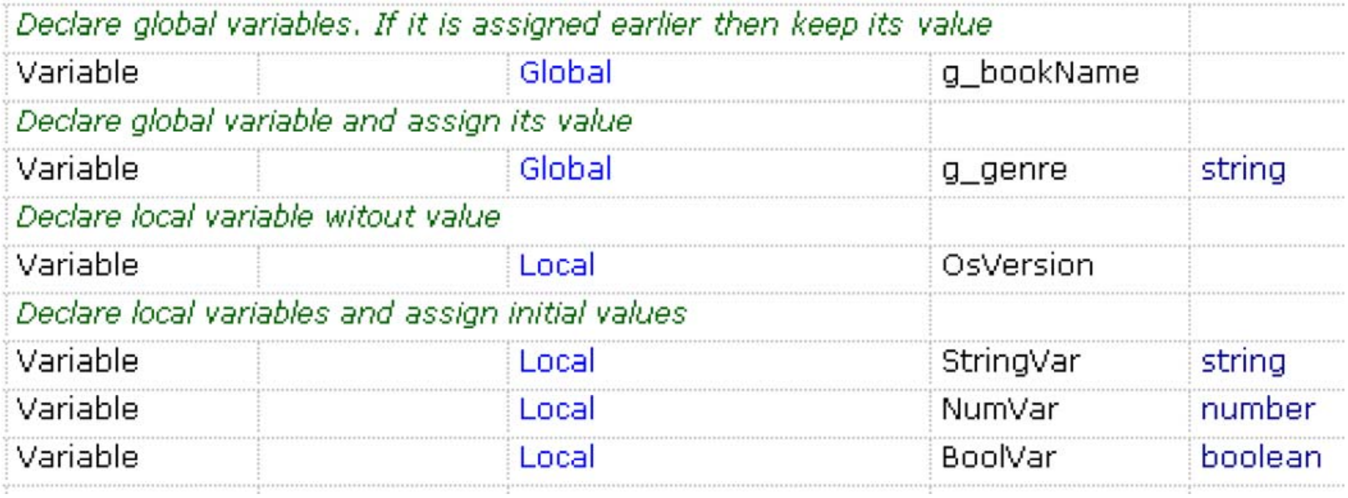
Then in the RVL you may declare `globalVar` as global and access it (read or assign values). Declaring variable as global is simple:

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

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

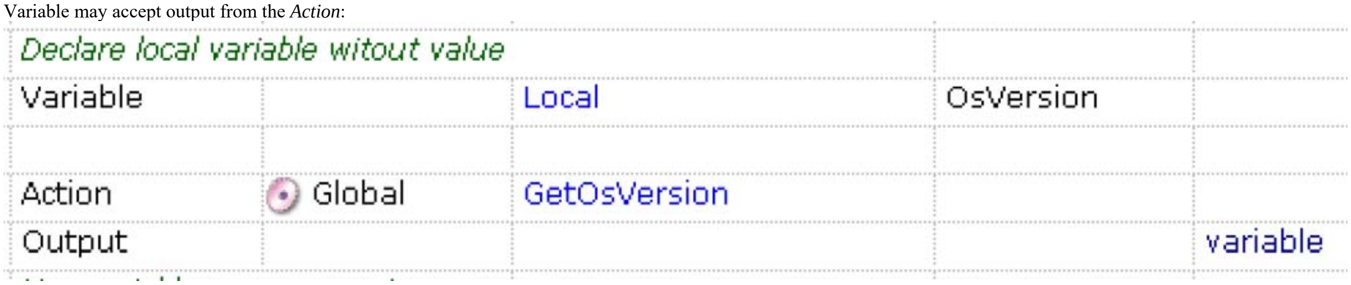
### Examples

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



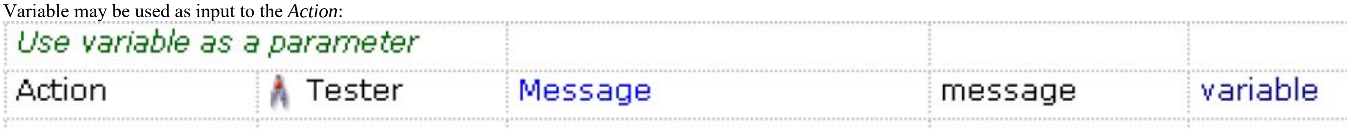
| | | | | |
|---|---|---|---|---|
| *Declare global variables. If it is assigned earlier then keep its value* | | | | |
| Variable | | Global | g_bookName | |
| *Declare global variable and assign its value* | | | | |
| Variable | | Global | g_genre | string |
| *Declare local variable witout value*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

Variable may accept output from the *Action*:



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

Variable 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 | |

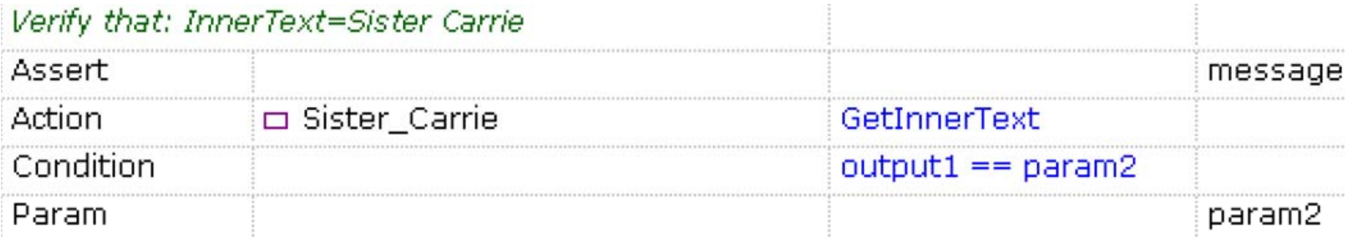
Assertion first line is always the same except the *Param Value*.

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 |

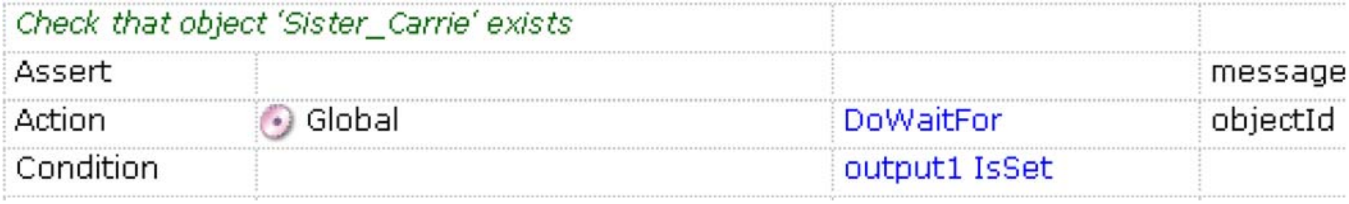
### Examples

Compare object property *InnerText* with expected value:



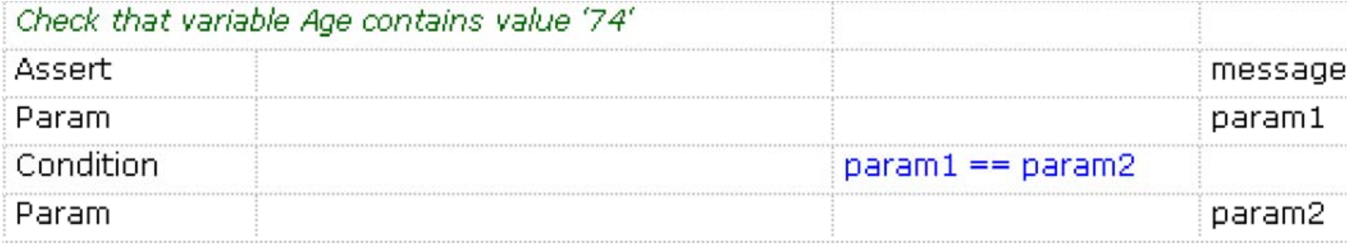
| Verify that: InnerText=Sister Carrie | | | |
|---|---|---|---|
| Assert | | | message |
| Action | □ Sister_Carrie | GetInnerText | |
| Condition | | output1 == param2 | |
| Param | | | param2 |

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:

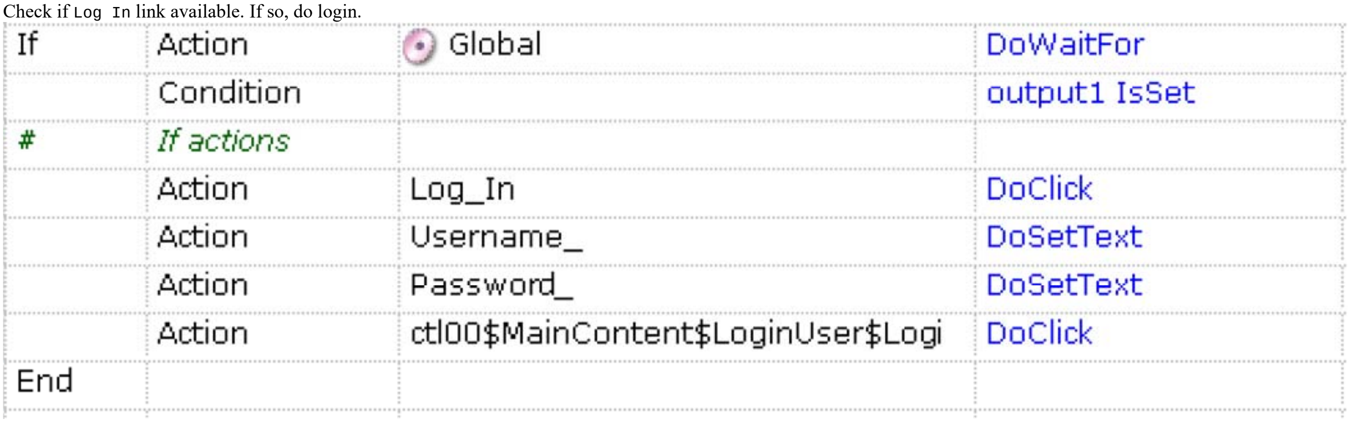
| ***Flow*** | ***Type*** | ***...*** | ***Action*** | ***ParamName*** | ***...*** |
|---|---|---|---|---|---|
| If | Param | | | param1 | |
| | Condition | | *condition statement* | | |
| | Param | | | param2 | |
| | ***some*** | ***actions*** | ***go*** | ***here*** | |
| ElseIf | Param | | | param1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

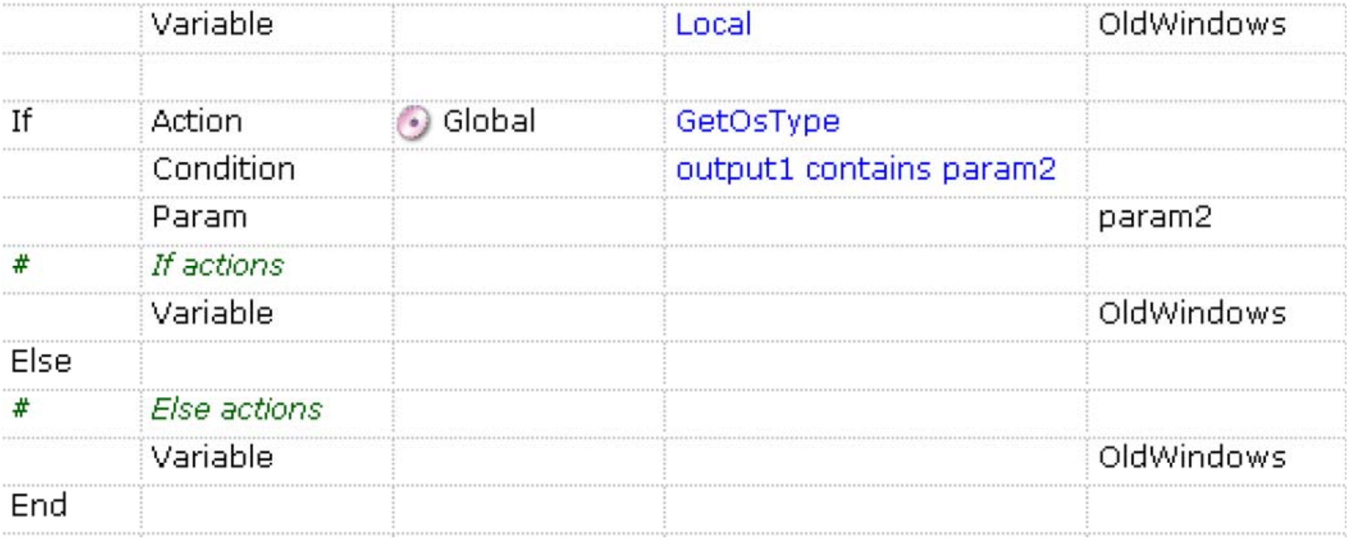
### Examples

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

Last 3 columns in RVL always using 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. May be basic type (`number`, `string`, `boolean`, `object`) as well as additional types:
- 
  - `expression` - any valid JavaScript expression that may involve global variables and functions and local variables.
- 
  - `variable` - parameter value is read from variable.
- 
  - `objectid` - ID of one of the learned Objects.
- 7th column - *ParamValue* - value acceptable for a specified *ParamType*. For `boolean` it is `true` or `false`. For `number` is is any floating point number (i.e. `3.14`). For `string` just 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 |

### Mixed Rows

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

For example 1st param of the *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 to 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 place:

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

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.