CA BEST PRACTICES

CA IT Process Automation Manager

Using Variables and Dataset Fields

- OVERVIEW OF DATASETS
- JAVASCRIPT VARIABLES VERSUS DATASET FIELDS
- TIPS FOR USING AND REFERENCING VARIABLES AND DATASET FIELDS

THIS IS A DRAFT DOCUMENT – FEEDBACK IS WELCOME

Transforming IT Management

LEGAL NOTICE

This publication is based on current information and resource allocations as of its date of publication and is subject to change or withdrawal by CA at any time without notice. The information in this publication could include typographical errors or technical inaccuracies. CA may make modifications to any CA product, software program, method or procedure described in this publication at any time without notice.

Any reference in this publication to non-CA products and non-CA websites are provided for convenience only and shall not serve as CA's endorsement of such products or websites. Your use of such products, websites, and any information regarding such products or any materials provided with such products or at such websites shall be at your own risk.

Notwithstanding anything in this publication to the contrary, this publication shall not (i) constitute product documentation or specifications under any existing or future written license agreement or services agreement relating to any CA software product, or be subject to any warranty set forth in any such written agreement; (ii) serve to affect the rights and/or obligations of CA or its licensees under any existing or future written license agreement or services agreement relating to any CA software product; or (iii) serve to amend any product documentation or specifications for any CA software product. The development, release and timing of any features or functionality described in this publication remain at CA's sole discretion.

The information in this publication is based upon CA's experiences with the referenced software products in a variety of development and customer environments. Past performance of the software products in such development and customer environments is not indicative of the future performance of such software products in identical, similar or different environments. CA does not warrant that the software products will operate as specifically set forth in this publication. CA will support only the referenced products in accordance with (i) the documentation and specifications provided with the referenced product, and (ii) CA's then-current maintenance and support policy for the referenced product.

Certain information in this publication may outline CA's general product direction. All information in this publication is for your informational purposes only and may not be incorporated into any contract. CA assumes no responsibility for the accuracy or completeness of the information. To the extent permitted by applicable law, CA provides this document "AS IS" without warranty of any kind, including, without limitation, any implied warranties of merchantability, fitness for a particular purpose, or non-infringement. In no event will CA be liable for any loss or damage, direct or indirect, from the use of this document, including, without limitation, lost profits, lost investment, business interruption, goodwill or lost data, even if CA is expressly advised of the possibility of such damages.

COPYRIGHT LICENSE AND NOTICE:

This publication may contain sample application programming code and/or language which illustrate programming techniques on various operating systems. Notwithstanding anything to the contrary contained in this publication, such sample code does not constitute licensed products or software under any CA license or services agreement. You may copy, modify and use this sample code for the purposes of performing the installation methods and routines described in this document. These samples have not been tested. CA does not make, and you may not rely on, any promise, express or implied, of reliability, serviceability or function of the sample code.

TITLE AND PUBLICATION DATE:

CA IT PAM Best Practices: Using Variables and Dataset Fields

Draft – Last Update Date: April 4, 2011

ACKNOWLEDGEMENTS

Principal Authors and Technical Editors

George Curran
Terry Pisauro
Daniel Zilberman

CA PRODUCT REFERENCES

This document references the following CA products:

- CA IT Process Automation Manager (CA IT PAM) Feedback

This is a draft document – feedback is welcome! Please email us at impcdfeedback@ca.com to share your feedback on this publication. Please include the title of this publication in the subject of your email response. For technical assistance with a CA product, please contact CA Technical Support at http://ca.com/support. For assistance with support specific to Japanese operating systems, please contact CA at http://www.casupport.jp.

Contents

CA IT Process Automation Using Variables and Dataset Fields

Overview

CA IT Process Automation Manager (IT PAM) datasets, parameters and variables make it possible to limit the number of processes that must be defined and to control the behavior of those processes by modifying parameter values. To take full advantage of these features you must understand the context (or scope) in which parameters and variables are defined and how to properly reference these objects from various places within a process flow. The purpose of this document is not to list the options supported for a given circumstance but to describe an approach that will work generically in most - if not all - situations.

Before You Begin

The best way to learn how to apply the concepts described in this document is to experiment with them on your own. If you are new to CA IT PAM, you should refer to the *CA IT Process Automation Manager Quick Start Guide* for information on how to install and to familiarize yourself with the application. At a minimum, you should be familiar with JavaScript and the CA IT PAM "Calculation" operator as well as the pre-execution and post-execution options exposed by other CA IT PAM operators.

Datasets

In CA IT PAM "datasets" are collections of variables with their values - also referred to as "fields". There are five types of Datasets used in CA IT PAM:

- **System Dataset:** Contains predefined variables that are available in the context of the entire CA IT PAM domain. These fields access system parameters and are made available by the ***System*** keyword.
- **Process Dataset:** Defined in the context of a process. Fields in this dataset can be referred to in expressions using the ***Process*** keyword. For example:

  ```
  Process.variableName
  ```

- **State Policy Dataset:** Defines fields in the context of a State Policy object. It is referred to by expressions within the State Policy as the using the ***Process*** keyword.
- **Operator Dataset:** Defines fields in the context of an operator within a process. The expression ***Process[OpName]*** (where the keyword "OpName" resolves to the current operator name) can be used to refer to fields in this dataset in scripts and properties within that operator. References to fields in one operator's dataset from another operator in the same process must refer to the operator explicitly by name. So, for example, to reference "field_1" defined in the dataset for "Operator_A" in a script or property in "Operator_B" you must use the an expression like "Process["Operator_A"].field_1" or "Process.Operator_A.field1".

- **Named Dataset:** A distinct object that is edited and saved to a folder in an Automation Library and contains a number of fields with values that can be accessed from different ITPAM processes. A sample syntax for accessing fields of a Named Dataset is:

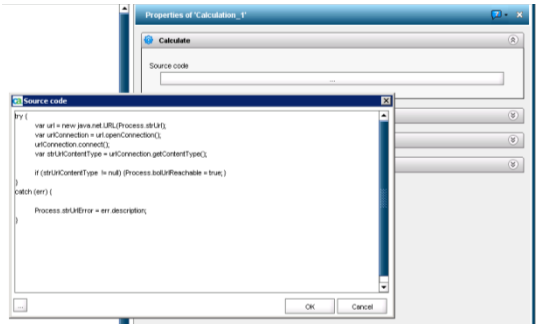
  `Datasets["<path to DS object>"].variableName.`

JavaScript Variables versus Dataset Fields

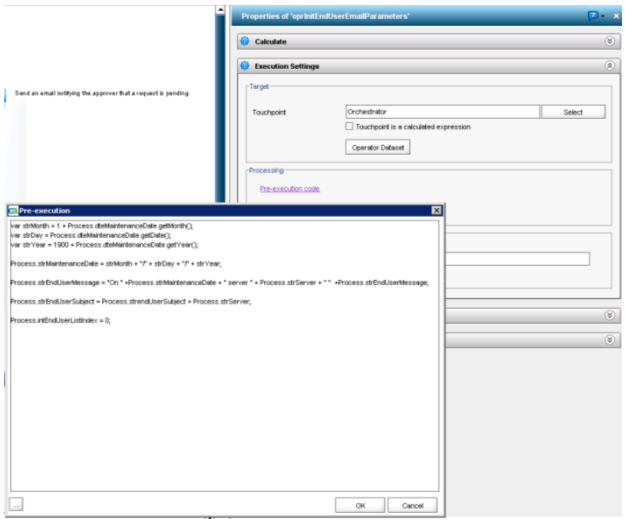
Although both JavaScript variables and Dataset fields are used similarly in CA IT PAM, there are several key differences that you should be aware of to help understand when to use a Javascript variable and when it is better to use an ITPAM dataset field/variable.

JavaScript Variables

JavaScript objects are defined, set and read within snippets of code inserted in the CA IT PAM "Calculation" operator as well as in pre-execution and post-execution scripts you may use with other CA IT PAM operators. Since these variables are not defined in any of the dataset types mentioned in the previous section, the most important fact to keep in mind is that they can never be accessed (read or set) outside the script container in which they are defined –like the "Source Code" container of the "Calculation" operator…



…or the pre-execution or post-execution code containers…



To put it another way, JavaScript variables defined within these containers are only visible to the code running in these containers, subject, of course, to the normal scoping principles applicable to most programming languages including JavaScript. That means, for example, if you define a function within a code container and define a variable within that function the variable is only visible within that function.

Explaining all the rules and principles for defining and referencing JavaScript variables is, of course, beyond the scope of this document, however, there is sufficient reference material on the subject readily available on the internet. The critical thing to remember is the limit to the visibility of JavaScript variables within CA IT PAM. For example, in the pre-execution code container of an Operator we can have the following:

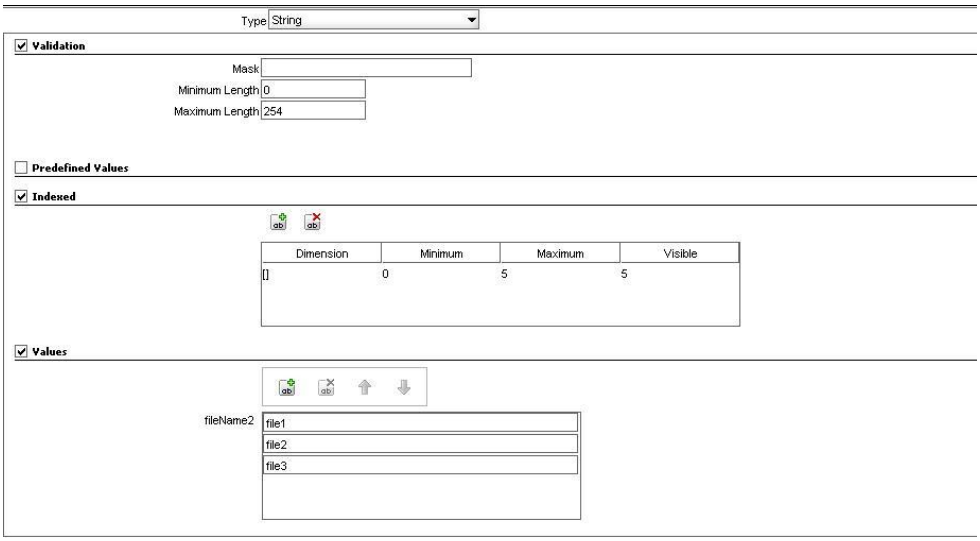
```
Var message = "Hello World";
for (int i=0; i <3: i++) {
  message +=" "+ (i+1);
}
```

At the end of execution of this code snippet the value of the "message" variable will be "Hello World 1 2 3". Once this code is executed, however, and the process moves to the next step that value can no longer be accessed.

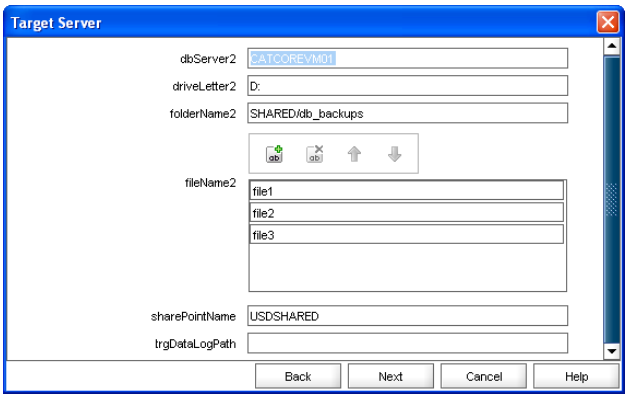
Dataset Fields

Dataset fields are objects that are used similarly to JavaScript variables but, unlike JavaScript variables, dataset fields are defined in CA IT PAM dataset collections. Their values can be set and stored prior to execution, read and/or modified during execution and are persisted (continue to exist) after execution completes for troubleshooting purposes. Unlike ordinary JavaScript variables, dataset field values can be modified and read outside of the code container in which they are defined by qualifying references with the dataset context in which they are defined.

Dataset set field types are limited to a subset of types supported by JavaScript; *Boolean*, *date*, *string*, *integer*, *long*, *double*, *password*, *object reference*, and *ValueMap (special type that can include other variables of simple types, see below)*. All data types can be configured to contain a single value or an array of values (called an *indexed field*). An indexed field can define an array of one or more dimensions and number of visible and maximum elements can be specified.



The values of these variables can also be “tested” during design time using the “Test Dialog” button to the left of the Help button.



Tips for Using Variables and Fields

While there may be many ways in CA IT PAM to accomplish a given task and achieve the same results following a few simple guidelines may help you avoid unnecessary troubleshooting and updates.

Adopt and Apply a Consistent Naming Convention

Employing a consistent naming convention that includes some indication of the type of variable or field being used can save you many hours of debugging and troubleshooting. It will also make it easier for colleagues and others to understand and adapt process definitions you may share with them. There are number of “naming standards” posted on the Internet. Your organization may already use one of these or one of their own. Whatever the case may be, the few extra keystrokes required when defining the process can save you hours later. For example:

- **dataSetPath** variable may contain a String value of path to external Dataset object,
- **numberOfResources** variable may contain an Integer value of a number of Resource objects to be used
- **actionListValueMap** variable may contain a Value Map- a set of “embedded” variables to be used in a Form’s actionList drop-down field

Use JavaScript Variables to Store Temporary or Intermediate Values in Scripts

Since the expressions used in pre-execution, post-execution or “Calculation” operator scripts can often be quite complex it may be necessary to temporarily store results in local variables. For example, the following JavaScript snippet could be used to set a string type process dataset field, “strDate”, to a string representation of the value of the date type process dataset field “dteDate”:

```
Process.strDate = (1 + Process.dteDate.getMonth()) + "/" +
    Process.dteDate.getDate()+ "/" + (1900 + Process.dteDate.getYear());
```

Although the end result is correct, it may be difficult to read and understand for anyone attempting to troubleshoot your process. The following example, using JavaScript variables to temporarily store intermediate values might be considered more intuitive and easier to understand:

```
var strMonth = 1 + Process.dteDate.getMonth();
var strDay = Process.dteDate.getDate();
var strYear = 1900 + Process.dteDate.getYear();

Process.strDate = strMonth + "/" + strDay + "/" + strYear;
```

Using temporary JavaScript variables will not have a measureable impact on performance or resource consumption but it may save many hours when the time comes for troubleshooting or modifying your process definitions.

In some cases, inside code snippets we may need to use JavaScript variables of types that are not supported by CA ITPAM. An example of this is the following code which checks the validity of a URL value stored in the Process.strUrl dataset variable and assigns a value to the Boolean dataset variable:

```
try {
                var url = new java.net.URL(Process.strUrl);  //unsupported ITPAM type
                var urlConnection = url.openConnection();  //unsupported ITPAM type
                urlConnection.connect();
                var strUrlContentType = urlConnection.getContentType();

                if (strUrlContentType  != null) {Process.bolUrlReachable = true;}
}
catch (err) {

                Process.strUrlError = err.description;
}
```

Always Qualify Dataset Fields

Depending on the current context, certain dataset fields may not require qualification. Understanding when a field must be qualified and when it may be accessible without qualification can be confusing and can lead to unexpected results or failures. The small additional effort required to clarify this will save you time troubleshooting and make your process's definition easier for others to understand and adapt.

CA IT PAM provides some useful keywords to make it easy to qualify the context (source dataset for a field) without the pitfalls of "hard coding". These include:

- **System:** Resolves to the "System" dataset.
- **Process:** Resolves to the dataset for the current instance of the process.
- **Process[OpName]:** Resolves to the dataset for the current operator for the current instance of the process.

For example:

```
var strMonth = 1 + dteDate.getMonth();
var strDay = dteDate.getDate();
var strYear = 1900 + dteDate.getYear();

strDate = strMonth + "/" + strDay + "/" + strYear;
```

Since CA IT PAM assumes the default context to be the current process instance, it is typically not necessary to qualify the process dataset fields. However, in a long or complex code snippet it might be difficult to determine which are local JavaScript variables and which are CA IT PAM process dataset field objects (such as strDate in the example above). Indeed, without looking back in the document it may not be clear in the simple example above, which is which.

Although it may not be necessary in certain cases, it is never wrong to qualify variables:

```
var strMonth = 1 + Process.dteDate.getMonth();
var strDay = Process.dteDate.getDate();
var strYear = 1900 + Process.dteDate.getYear();

Process.strDate = strMonth + "/" + strDay + "/" + strYear;
```

In the example above, it should be clear which are simple local JavaScript variables and which are persistent process dataset fields.

Operator Dataset Fields versus Process Dataset Fields

Fields that are referenced only within the context of a single operator and…

- …are referenced in different aspects of that operator (i.e., in both pre-execution and post-execution scripts) AND
- …need to be persisted for debugging or troubleshooting

…should be defined to the operator dataset. While it may not be necessary, under certain circumstances, to qualify the field's name with the "**Process[Opname]"** prefix it is never wrong to do that. It will certainly help you avoid issues and make it easier for others to understand your process definitions.

Fields that must be referenced or updated beyond the context of an individual operator in a process should be maintained in the process dataset. Again, while it may not be necessary to use the "Process" prefix when referencing process dataset fields it is never wrong to do so and the benefits already mentioned apply.

Avoid referencing the operator dataset fields of one operator in the properties or scripts in another operator. While it is possible to explicitly reference another operator's dataset fields it increases the risk of "breaking" the process flow if modified. To illustrate, suppose two operators, "Operator_A" and "Operator_B" both need to use the string representation of a date from our previous example.Assume the following code snippet was executed by "Operator_A":

```
var strMonth = 1 + Process.dteDate.getMonth();
var strDay = Process.dteDate.getDate();
var strYear = 1900 + Process.dteDate.getYear();

Process[OpName].strDate = strMonth + "/" + strDay + "/" + strYear;
```

Notice this time the field "strDate" has been defined and set in the current operator dataset, "Operator_A" (remember, "**Process[OpName]** resolves to the current operator).Now, for whatever reason, the date string value, "strDate" is needed in a property or script for "Operator_B". You cannot use the "**Process[OpName]"** prefix as that would resolve to the current operator ("Operator_B"). You could explicitly reference the dataset and field defined in "Operator_A" using "**Process["Operator_A"].strDate**" or "**Processs.Operator_A.strDate**". Although these would work, both processes could be broken by simply changing the name of "Operator_A" at some point. Understandably, that may be a low risk but it is completely avoidable by following the simple principle of defining a field that is used by multiple operators in a process in the process dataset.

Therefore, a safer solution would be to use a previous example in the "**Operator_A**":

```
var strMonth = 1 + Process.dteDate.getMonth();
var strDay = Process.dteDate.getDate();
var strYear = 1900 + Process.dteDate.getYear();

Process.strDate = strMonth + "/" + strDay + "/" + strYear;
```

and then reference Process data set variable from "**Operator_B**":

```
var userMessage = "Today's date is: "+Process.strDate
```

*This will always work as it is using the value of the process dataset variable **Process.strDate** inside the script of* "**Operator_B**".

These types of considerations are especially useful when working with "Run Process" operators that execute other CA ITPAM processes and may return large sets of variables that need to be used later in a process execution. It is a good practice to copy these values to process dataset variables.

Using CA IT PAM ValueMap variables

CA IT PAM has a special data type called **ValueMap** (see detailed definition in the "CA IT Process Automation Manager Reference Guide", pages 177-188, 350) that can contain other fields. ValueMap type essentially is a JavaScript hash table structures which are in turn permutations of associative arrays (i.e. Name => Value pairs).

The Javascript language implements very loose and somewhat limited support for associative arrays. Any JavaScript array can use other objects as keys, making it a hash, but there is no formal constructor for initializing them. A short example of a hash structure in JavaScript would be as follows:

```
var myArray = new Array();

myArray['one'] = 1;

myArray['two'] = 2;

myArray['three'] = 3;

// show the values stored
```

```
for (var key in myArray) {
        alert('key is: ' + key + ', value is: ' + myArray[key]);
}
```

If executed in JavaScript, this should display messages like:

key is: one, value is: 1

key is: two, value is: 2

key is: three, value is: 3

In JavaScript, every variable is in fact an object. That essentially means that no matter what the variable, it can be used as though it were an instance of an object. This means it has a constructor, methods and properties. A property is just a variable that is owned by the object and thus local to that object. A property is accessed using the syntax:

myArray.one

or

myArray['one']

Initialization of ValueMap Fields

In CA IT PAM , ValueMap fields are defined and used very similarly to general JavaScript hash objects

1. All ValueMap variables – non-indexed and indexed – have to be initialized before assigning values to them.

For the non-indexed variables we can use standard ITPAM function **newValueMap():**

```
Process.VMvariable = newValueMap();    //for Process variable

Form.VMvariable = newValueMap();      //for Form variable in Form initialization code
```

Indexed VM variables need to be initialized with an array of VM objects:

```
var vm = newValueMap ();
vm.Select = false;
vm.c1="a";
vm.c2="b";
vm.c3="c";

Process.VMVarINdexed = [vm]; //Create a new array of vm object;
Process.theSRReview.length = 10; //set the length of the array
```

Similarly, for a Form field that as an array of ValueMap objects, we can do:

```
Form.VMVarIndexed = [vm];
Form.VMVarIndexed.length=10;
```

2. After proper initialization, we can use either **VMvar.fieldName** or **VMvar[‘fieldName’]** syntax for accessing values of these variables:

Form.VMVarname.fieldName = ‘something’;

Form. VMVarname[‘fieldName’] = ‘something’ ; //same as above

In a Process dataset:

Process.VMVarname.fieldName = ‘something’;

Process. VMVarname[‘fieldName’] = ‘something’; //same as above

For indexed VM variables, we can assign their values iterating through and array like in this example:

```
for  (i=0; i<Process.theSRReview.length; i++) {

        Form.VMVarIndexed[i].fieldName = 1;  //use  .fieldName notation

        Form.VMVarIndexed[i][‘fieldName’] = ‘something’;  // use [‘fieldName’] notation

        Process.VMVarIndexed[i] [‘fieldName’] = ‘something else’;

}
```

The example above would achieve a result that all elements of ***Form.VMVarIndexed*** and ***Process.VMVarIndexed*** indexed variables will have values of field ‘filename’ set to values on the right hand site of “=” operator.

Using ValueMap Variables in Process dataset

One of many uses of Process variable of ValueMap type is local copy of external Dataset object into that variable. The reason is that ITPAM locks access to a Dataset object for other processes while it is being accessed by a process instance and we therefore need to minimize that access duration.

//read the Dataset into local VM variable

Process.theDataset = Datasets["ValueMap Dataset"];

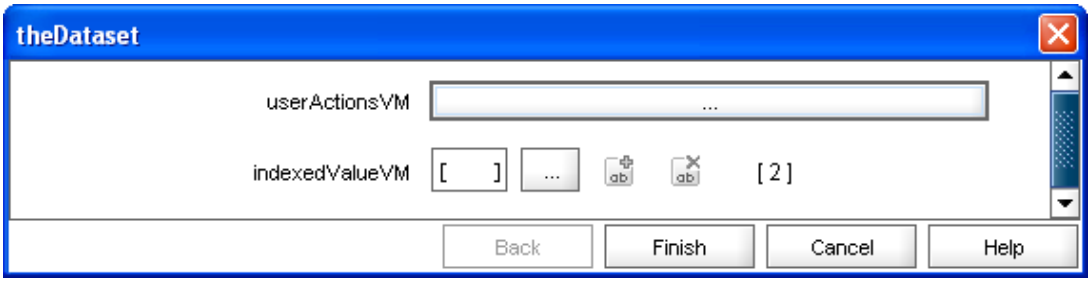
This essentially copies the content of that Dataset object into the “theDataset” variable. Then, to access individual fields of that Dataset we can use same syntax as for any VM variable:

Process.theDataset.username

Or

Process.theDataset[‘username’]

Will contain the value of “username” field that was stored in a Dataset object at the time it was copied to the Process:

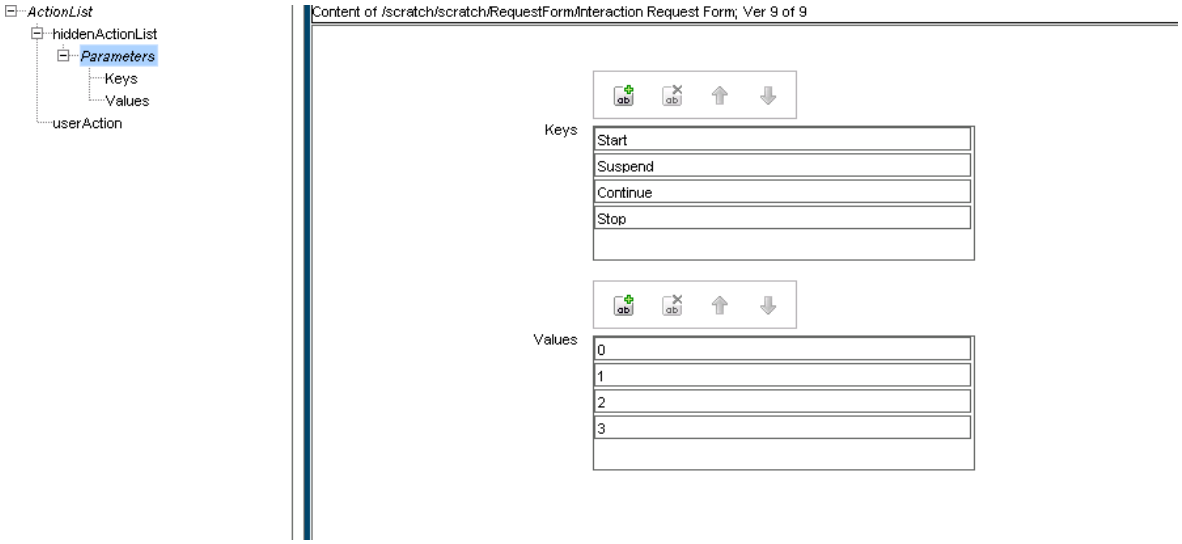


Note that in this example we did not have to define "theDataset" field in the Process dataset – just like for any other ITPAM variable type it was created and assigned appropriate type as a result of assignment.

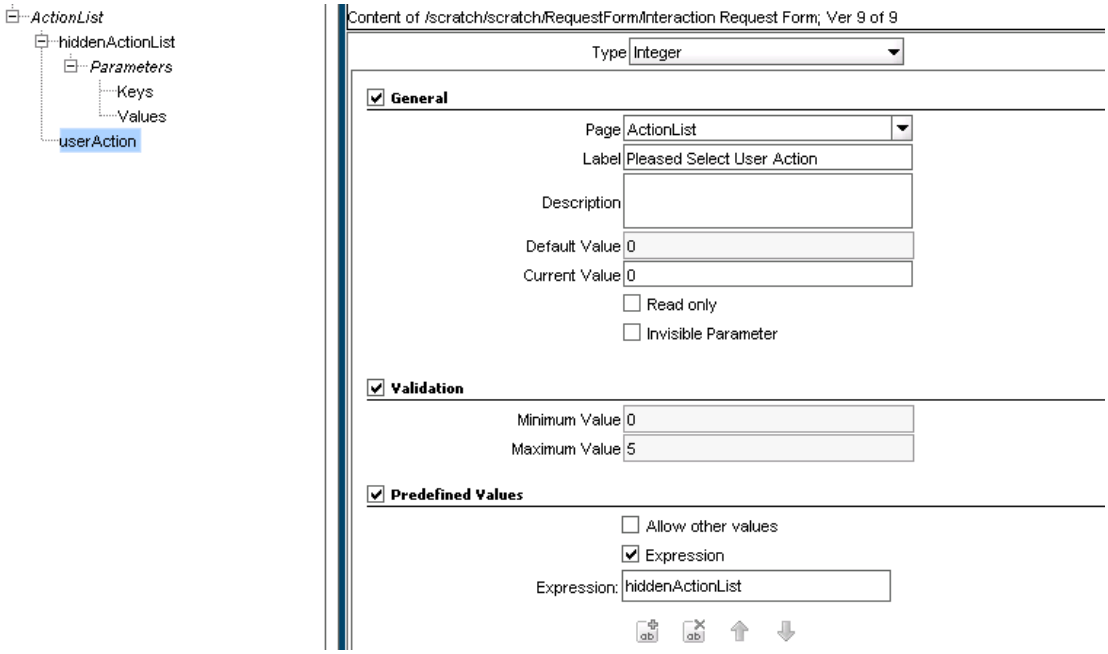
Using ValueMap Variables in User Interface Forms

ValueMap variables can contain different fields "inside" them and therefore can be used for various User Interface purposes where information needs to be stored and presented in a certain way.

For example, if a User Interaction Form contains a field of ValueMap type called "hiddenActionList" with the following parameters:

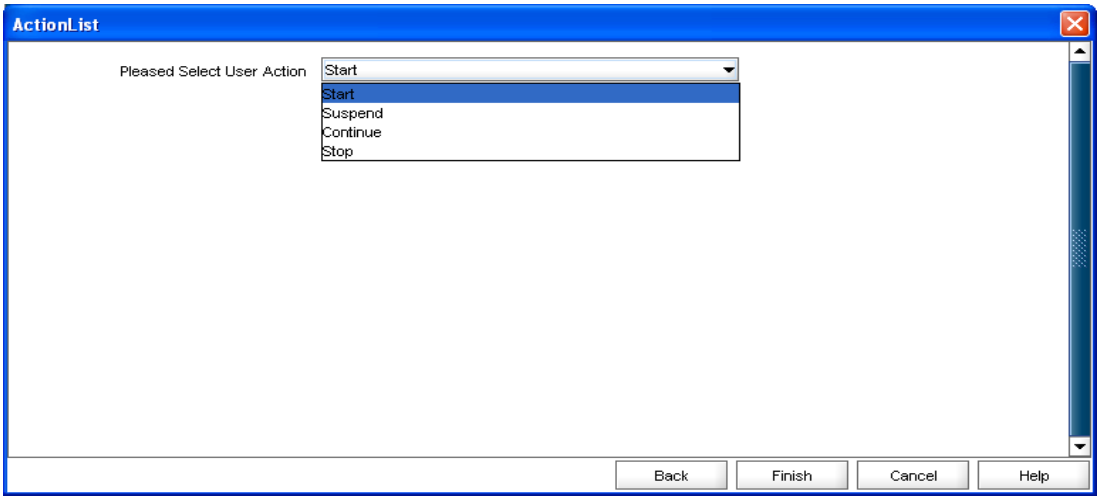


and another Field of Integer type – "userAction" - that references that field name via "Predefined Values - Expression" settings:



NOTE: as follows from the field name “hiddenActionList” – it should not be displayed on the UIF. That features will be available in ITPAM 3.0

This UIF form will contain a drop-down element with Keys displayed in the list and Values associated with these keys contained in the “nested” fields of the *hiddenActionList* variable



That is a “design time” setting of drop-down lists. By assigning values to the “hiddenActionList” in the “Form data initialization code” tab of the “User Interaction” operator:

Form.hiddenActionList = Process.theDataset.userActionsVM;

we can actually manipulate values displayed in the drop-down list. In the example above, we assign values stored in the ***userActionsVM*** field of ***theDataset*** process variable (which contains a copy of external Dataset object). As a result of assignment above, the UIF will contain the following drop-down list:

NOTE: there is a new type of UI component –“table” - that can be associated with fields of ValueMap type and can be used for visual display of all fields of such a variable. It will be available in ITPAM 3.0

CA IT PAM System Functions related to ValueMap type

ITPAM offers a number of system functions that work with variables of ValueMap data type. They can be found in the “ITPAM Reference manual” document or via in-context help.

The example below demonstrates two of them – ***hasField(<VM var name>, <field Name>)*** and ***deleteValueMapField(<VM Var name>, <field Name>)***. The first one returns

Process.theVMVar = newValueMap();

theVMvar['Var0']='something'; //use ['field'] notation

theVMVar['VMvariable']='something else'; //use VMOBject['field'] notation

Process.bHasVar0 = hasField(Process.theVMVar, "Var0"); //should be true

Process.bHasVMvariable = hasField(Process.theVMVar, "VMvariable"); //should be true

//now, delete Var0 field from theVMVar and test again:

var bSuccess = deleteValueMapField(Process.theVMVar, "Var0");

Process.bHasVar0Deleted = hasField(Process.theVMVar, "Var0"); //should be **false** ***now***

After the code above executes, the “flag” variables have the following values

As expected – after deletion of the ***"Var0"*** field from the ***Process.theVMVar*** ValueMap variable, result of the hasField(..) function call using that field name is *false*.

Another helpful system function is ***getValueMapFields(<VM Var Name>)*** which returns a String array of field names contained in a ValueMap variable

The code below calls that function for a standalone ("VMVariable") and indexed ("SR2Review") form fields in the post-execution section of User Interaction operator

//get String arrays of ValueMap fields' names

Process.rgsFieldNames1 = getValueMapFields(Process[OpName].VMvariable);

Process.rgsFieldNames2 = getValueMapFields(Process[OpName].SR2Review[0]);

As a result – there are two String array variables created in Process dataset that contain field names of ValueMap fields of User Interaction form

Practical Tips on Using Complex ValueMap objects

Array and ValueMap Basics

Even the most complex data structures are composed of layers of base structures. Arrays and ValueMap objects are no more than collections of simpler strings, numbers, etc. The elements of an array must be identical in terms of data type and structure. Individual elements are referenced using an integer index (*var element = array[n]*). ValueMap objects can encapsulate fields of different data types and the values are referenced using the field name (*var value = vmap.field_name*). Since elements of array and fields in ValueMap objects can themselves be array and ValueMap objects, it is possible to build very sophisticated and complex data structures. Fortunately, no matter how sophisticated or complex the overall structure becomes, the structures can be easily navigated, level by level using only the well-known methods that apply to the object type(s) at each level.

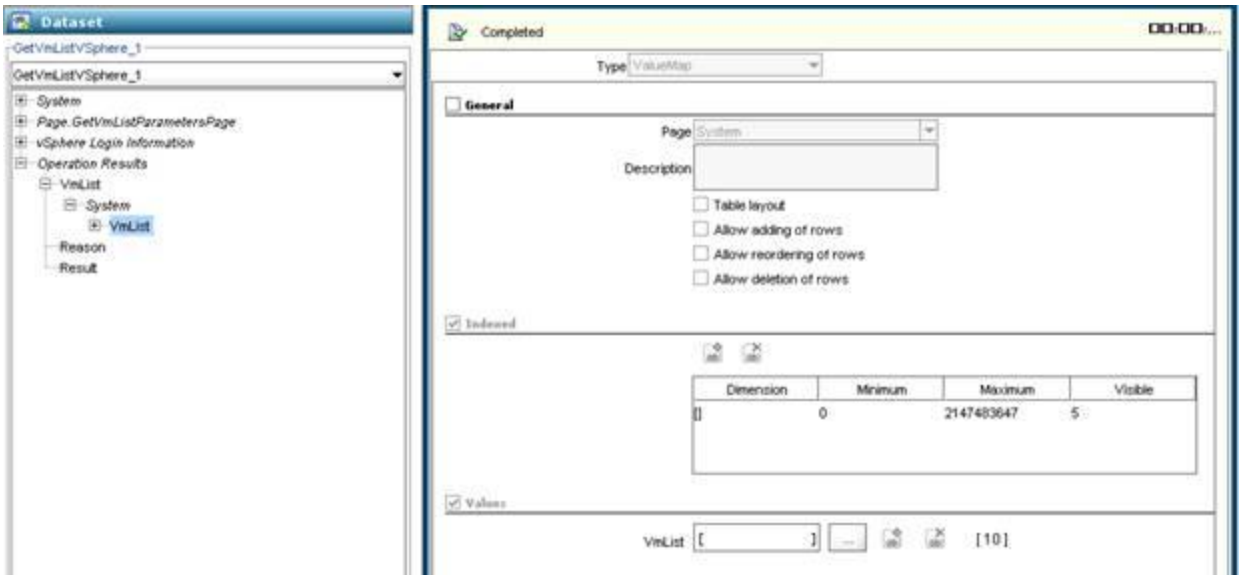
Apply the Basics

Work is in progress to revise and expand upon available documentation. In the meantime, to fully understand the behavior of an operator, determine what information it surfaces and how it is stores it is sometimes necessary to run a simple test process and examine the resulting dataset to locate the "interesting" object or objects. Raymond Ho's query related to iterating the content returned by the "Get VM List – Sphere" operator will be the basis of the example use case.

Based on the example, it appears that the ValueMap object "VmList" under "Operations Results" may contain the information needed.

In JavaScript, "Process.GetVmListVSphere_1.VmList" returns the ValueMap object.

Expand the "VmList" node to examine its structure.

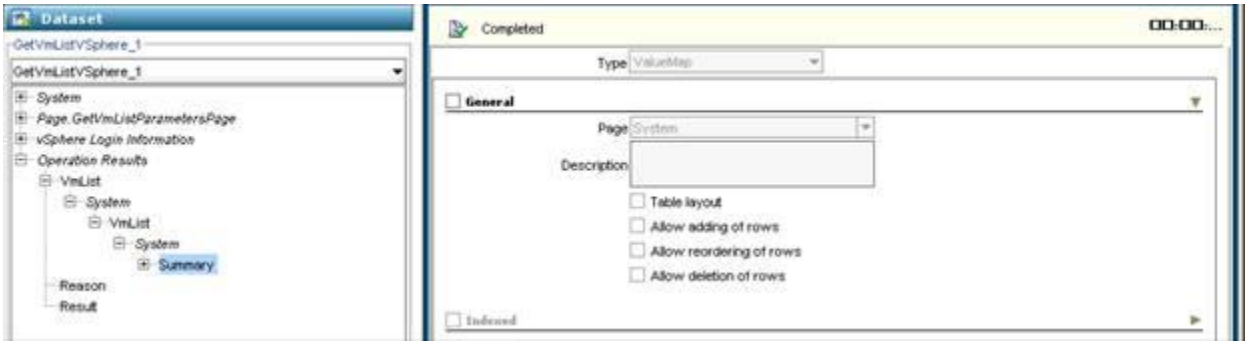


The top level ValueMap object, “VmList”, has one child object also name “VmList”. From the properties displayed in the right hand pane, we see the child “VmList” object is an array of ValueMap objects (the “Indexed” option is enabled) encapsulating 10 elements. Agree, it may be atypical and confusing, but to be clear, the top level “VmList” ValueMap object encapsulates an array of ValueMaps also named “VmList”.

In JavaScript, “Process.GetVmListVSphere_1.VmList.VmList” returns the array. The expression “Process.GetVmListVSphere_1.VmList.VmList.length” returns the number of elements in the array. The expression “Process.GetVmListVSphere_1.VmList.VmList[n]” returns the ValueMap object corresponding to element “n” in the array where “n” is an greater than or equal starting at 0 and less than the “length” of the array. To iterate all the ValueMap objects in the array:

```
for (var i = 0; I < Process.GetVmListVSphere_1.VmList.VmList.length; i++){
        var objValueMap = Process.GetVmListVSphere_1.VmList.VmList[i];
        <Do stuff with the ValueMap object>
        }
```

Expand the child “VmList” array of ValueMaps to view the structure of each value map it contains.

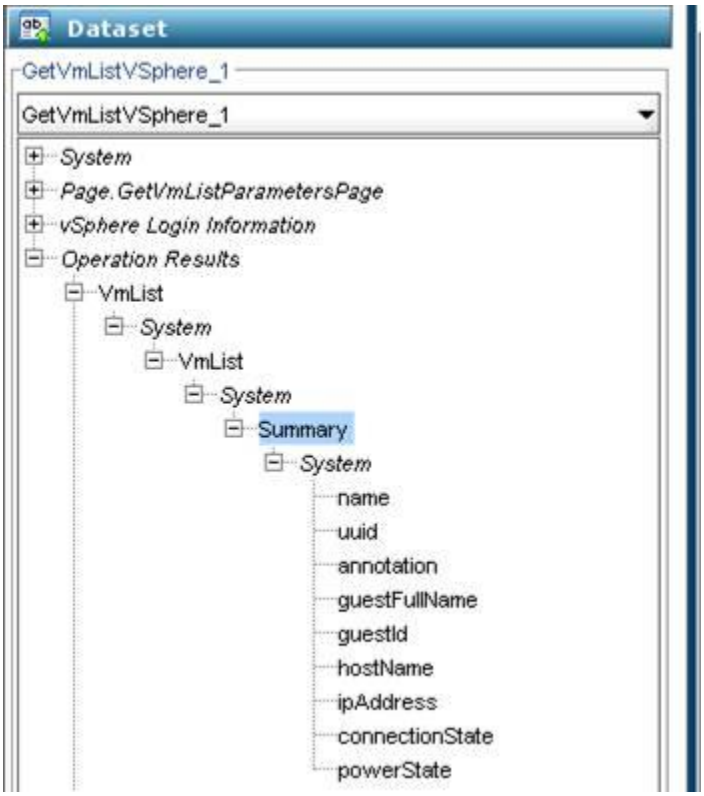


The ValueMap object in “VmList” array of ValueMap objects encapsulates a single ValueMap object named “Summary”. Once again, it may be atypical and confusing, but to be clear, each ValueMap in the array contains a ValueMap in this case.

In JavaScript, “Process.GetVmListVSphere_1.VmList.VmList[n].Summary” returns the ValueMap object that is the “n” element of the array. To iterate each “Summary” ValueMap object:

```
for (var i = 0; I < Process.GetVmListVSphere_1.VmList.VmList.length; i++){
        var objValueMap = Process.GetVmListVSphere_1.VmList.VmList[i].Summary;
        <Do stuff with the ValueMap object>
}
```

Finally, drill into “Summary”.



The “Summary” ValueMap object encapsulates the really interesting properties.

In JavaScript, “Process.GetVmListVSphere_1.VmList.VmList[n].Summary.name” returns String object that set to the value of the “name” field in the “Summary” ValueMap object that is the “n” element of the array. To iterate each “name”:

```
for (var i = 0; I < Process.GetVmListVSphere_1.VmList.VmList.length; i++){
        var strName = Process.GetVmListVSphere_1.VmList.VmList[i].Summary.name;
        <Do stuff with the name string>
}
```

Scripting Tips

As evidenced by the preceding example, referencing the “interesting” property values surfaced in complex data structures by operators can be cumbersome. The long complex expressions are difficult to read and easy mistyped when coding. Going back to the dataset object each time in JavaScript can also be expensive.

The code examples in the previous section deliberately reference the dataset object directly to reinforce the concepts of navigating complex data structures. Given state use case for the example (iterate the names returned), the following JavaScript would be more efficient and less prone to error.

```
var aryVmapsSummary = Process.GetVmListVSphere_1.VmList.VmList;
for (var i = 0; I < aryVmapsSummary.length; i++){
        varVmapSummary = aryVmapsSummary[i].Summary;
        var strName = varVmapSummary.name;
        <Do stuff with the name string>
}
```

Notice the first set a simple JavaScript variable to the “interesting” array of ValueMaps stored in the complex dataset structure. It is the only time in the script where is it is necessary to traverse the cumbersome path back through the dataset to access the value needed.

Summary

Many of the most common problems that a user may encounter when defining CA IT PAM processes can be avoided by applying some simple principles when referencing variables and dataset fields:

- Apply an easy to follow naming convention.
- Use JavaScript variables to store intermediate values to simplify expressions in code.
- Define dataset fields in the correct context (for example, operator, process, named Dataset object).
- Always qualify dataset fields with the correct context prefix.
- Use ValueMap variable types to store other fields inside these variables
- While navigating complex data structures the encapsulate ValueMaps in arrays and even other ValueMaps may seem a bit overwhelming, it can be done breaking the structure down level by level then applying the well-known principles for accessing child objects to progress to the next level.

Adhering to these few simple guidelines will make you more productive and your process definitions easier to understand.