ARIS

REPORT SCRIPTING - BEST PRACTICES

ARIS 10.0

April 2017

TABLE OF CONTENTS

1 Introduction

This document does not represent a comprehensive policy to the development of reports in the ARIS environment, but it describes a series of "best practices" and "mistakes" in the report development, which base on experience. Adherence to the guidelines described in this document does not necessarily secure the required and necessary running time and memory features, because this depends on the application of the reports to be developed.

Completed reports should always be tested on their memory performance and quality.

2 Java Script Basics

The following sections deal with basic guidelines and "mistakes" in the JavaScript environment.

2.1 Global variables

2.1.1 Avoid global variables

Global variables should not be used for referencing the data to be processed, because they are held in memory until the end of the report. The use of global variables should be restricted to constant data types that serve to control the parameters and report process.

Example:

```
var g_data_objDef;
var g_data_models;

...

function find() {

    var activeDatabase = ArisData.getActiveDatabase();
    g_data_objDef = activeDatabase.Find(Constants.SEARCH_OBJDEF);
    g_data_models = activeDatabase.Find(Constants.SEARCH_MODEL);

    var result_objDef = checkObjDefs(g_data_objDef);
    var result_modelDef = checkModelDefs(g_data_models);
    ...

}

...
```

```
var g_searchType = Constants.SEARCH_OBJDEF;

...

function find() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    var result = checkSearchResult(data);
    ...

}

...
```

2.1.2 Definition of variables

All variables should always be explicitly defined. This means that all variables are defined using "var". Variables which have been defined without the "var" are always implicitly global, and so they are to avoid as described above.

Example:

```
function find() {

    var activeDatabase = ArisData.getActiveDatabase();
    data_objDef = activeDatabase.Find(Constants.SEARCH_OBJDEF);
    data_models = activeDatabase.Find(Constants.SEARCH_MODEL);

    ...

}
...
```

In the above example the variables "data_objDef" and "data_models" are implicitly defined globally and therefore live up to the end of the report.

```
function find() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data_objDef = activeDatabase.Find(Constants.SEARCH_OBJDEF);
    var data_models = activeDatabase.Find(Constants.SEARCH_MODEL);

    ...

}
...
```

2.1.3 Variables in ‘main’ function

Reports often contain an entry method that controls the main program flow and is referred to as the "main" function. In general, the report begins and ends with call and end of this function and the entire call stack is clamped by this "main" function. Variables defined within the "main" function are living from the beginning until the end of report execution. Therefore variables should be defined only in the "main" function if they are necessary for the whole context.

Example:

```
main();

function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data_objDef = activeDatabase.Find(Constants.SEARCH_OBJDEF);

    var containsMainSystem = containsMainSystem(data_objDef);

    if(containsMainSystem) {
      // only result will be used
    ...
   }

}
...
```

If, as in the example above, only the processing result of a larger amount of data is required for further processing, it should be examined whether it is absolutely necessary to keep the entire data set in the "main" function.

Possible solutions here could be the outsourcing of the search in the "containsMainSystem" function or the release of the search result. Details are described in the Clean-up and Using functions.

```
function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data_objDef = activeDatabase.Find(Constants.SEARCH_OBJDEF);

    var containsMainSystem = containsMainSystem(data_objDef);
    data_objDef = null;

    if(containsMainSystem) {
      // only result will be used
    ...
   }

}
...
```

```
main();

function main() {

    var containsMainSystem = containsMainSystem();

    if(containsMainSystem) {
      // only result will be used
    ...
   }

}
...

function containsMainSystem() {

   var activeDatabase = ArisData.getActiveDatabase();
   var data_objDef = activeDatabase.Find(Constants.SEARCH_OBJDEF);

   ...

}
```

2.2 Clean up

In the development of reports, which operate on larger data sets it is important to release information that is no longer used. In particular this is relevant for global variables.
The following sections describe the main procedures.

2.2.1 Release referenced data

Referenced data that are no longer used and are not of interest for the further processing should always be released. Because JavaScript does not allow an explicit release of referenced data and these are only discarded when they are no longer referenced by any variable, all still valid reference variables of the released data structure are always set to "null".

Example:

```
var g_searchType = Constants.SEARCH_OBJDEF;

...

function find() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    var result = checkSearchResult(data);

    ...

    for(var i = 0; i < result.length; i++) {
```

```
      ...
    }

}
...
```

```
var g_searchType = Constants.SEARCH_OBJDEF;

...

function find() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    var result = checkSearchResult(data);

    data = null;
    ...

    for(var i = 0; i < result.length; i++) {

      ...
    }

}

...
```

In the example above the data returned by the search processing are no longer needed after execution of function "`checkSearchResults()`" and should therefore be released by setting the reference variable to "null".

2.2.2 Partial release of referenced data

Often the data to be processed consists of a list or a set of data objects that are processed within one iteration. Especially when the processing of individual elements of the list or set is expensive and takes a longer run time, it makes sense to release those elements that are already processed.

Example:

```
var g_searchType = Constants.SEARCH_OBJDEF;

...

function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    for(var i = 0; i < data.length; i++) {

      if(data[i].TypeNum() != Constants.OT_ACTION) {
```

```
            continue;
      }
      ...
    }

}
...
```

```
var g_searchType = Constants.SEARCH_OBJDEF;

...

function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    for(var i = 0; i < data.length; i++) {

      if(data[i].TypeNum() != Constants.OT_ACTION) {
            data[i] = null;
            continue;
      }
      ...
    }

}
...

}

...
```

```
var g_searchType = Constants.SEARCH_OBJDEF;

...

function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    for(var i = 0; i < data.length; i++) {

       process(data[i]);
       data[i] = null;
    }

}
...
```

2.2.3 Using functions

The use of a single and normally large "`main`" function, with the result that all variables are defined in this method, has a global character. If it is forgotten to release the referenced data structures, they remain there from the beginning to the end of the report.
Separating the report into smaller functional blocks ensures that data objects live only within their own context of use and that not explicitly released data objects are released with the termination of the function (if they were defined with "`var`").

Example:

```
...

function main() {

    var activeDatabase = ArisData.getActiveDatabase();

    // process actions
    var actionsDefs = activeDatabase.Find(
            Constants.SEARCH_OBJDEF,
            Constants.OT_ACTION);
    ...

    // process systems
    var actionsDefs = activeDatabase.Find(
            Constants.SEARCH_OBJDEF,
            Constants.OT_APPL_SYS_TYPE);
    ...

    // process process support units
    var actionsDefs = activeDatabase.Find(
            Constants.SEARCH_OBJDEF,
            Constants.OT_PROCESS_SUPPORT_UNIT);
    ...

}
...
```

```
...

function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    processActions(activeDatabase);
    processSystems(activeDatabase);
    processProcSupportUnits(activeDatabase);

}

function processActions(activeDatabase) {

    // process actions
    var actionsDefs = activeDatabase.Find(
            Constants.SEARCH_OBJDEF,
            Constants.OT_ACTION);
```



```
    ...


}

fnction processSystems(activeDatabase) {

   // process systems
   var actionsDefs = activeDatabase.Find(
            Constants.SEARCH_OBJDEF,
            Constants.OT_APPL_SYS_TYPE);
    ...

}

function processProcSupportUnits(activeDatabase) {
  // process process support units
  var actionsDefs = activeDatabase.Find(
            Constants.SEARCH_OBJDEF,
            Constants.OT_PROCESS_SUPPORT_UNIT);
  ...

}

...
```

2.2.4 Large function stacks

Basically deep function nesting and recursions should be avoided and only be used with caution. They have the consequence that the functions referenced in the parent data structures can be released only after returning to the function block. Depending on the call depth this can result in accumulation of larger amounts of data. (Furthermore, all function calls in the report environment are mapped internally by Java Reflection on their respective implementation. A large number of function calls can therefore lead to performance loss.)

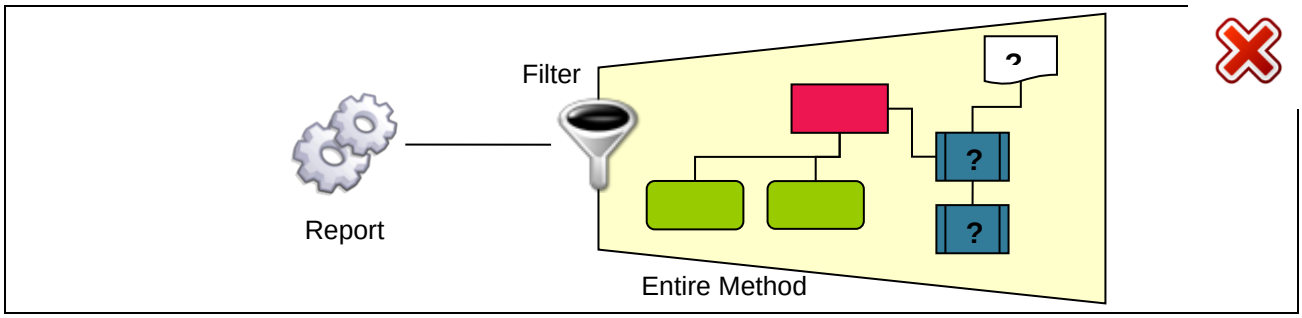
3 Working On reports

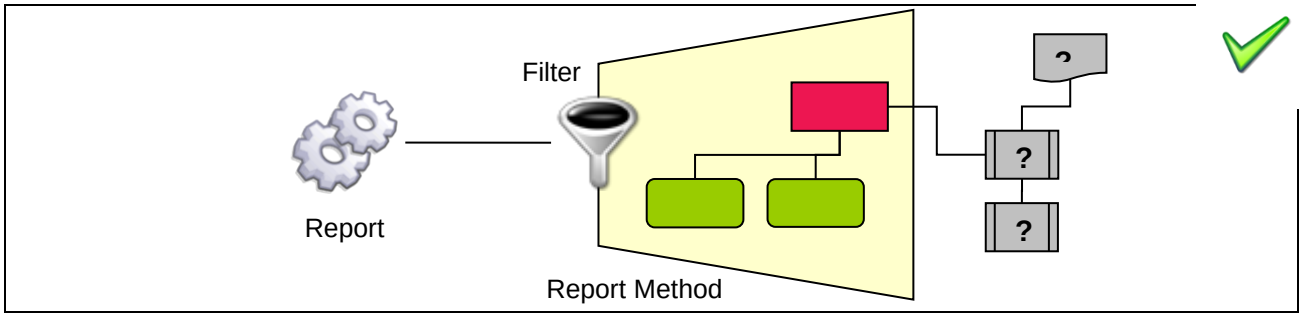
The following sections deal with simple and basic tools and guidelines for report development.

3.1 Using an appropriate method or evaluation filter

A decisive factor in the amount of processed data objects and thus hold by the report in memory, is the used method filter. For the execution of the report therefore always a method filter should be used, which corresponds to the viewing area of the report. If such a filter not exists in the environment of the executed report, a dedicated filter for use by the report should be created.

Example:





3.2 Working with the Object Model

3.2.1 Reading one attribute means reading all attributes

When reading object attributes it have to be noted, that reading a single attribute leads to procuring all defined attributes of the corresponding object definition and keeping them in memory.

Example:

```
var g_searchType = Constants.SEARCH_OBJDEF;

...

function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    for(var i = 0; i < data.length; i++) {
      var nameAttr =
data[i].Attribute(Constants.AT_NAME,Constants.LCID_GERMAN);
      ...

    }

}
...
```

In the example above all attributes of the object are read into memory by calling `data[i].Attribute( Constants.AT_NAME, Constants.LCID_GERMAN)`, and released from memory not until discarding of the object from memory.

3.2.2 Reading one connection means reading all connections

When reading object connections it have to be noted, that always all the defined object connections are read. Furthermore, it should be noted that reading of connections using filtering methods, meant here are functions that filter the selection like `CxnListFilter(int nCxnKind, int typeNum)` will mean that the connected objects are also read and stored in memory.

Example:

```
var g_searchType = Constants.SEARCH_OBJDEF;

...


function main() {

    var activeDatabase = ArisData.getActiveDatabase();
    var data = activeDatabase.Find(g_searchType);

    for(var i = 0; i < data.length; i++) {

      // read the connection definition only
      var cxnList = data[i].CxnList();
      ...

      // read the connection definition and the assigned object definitions
      var filtredCxnList = data[i].CxnListFilter(
            Constants.EDGES_INOUT, Constants.CT_USE_1);
      ...


    }

}
...
```

3.2.3 Context of the report execution

Report should be developed for an appropriate execution context. It should be noted that data selected for the execution context is held in memory for the whole runtime.

3.3 Working with external resources

The following sections describe what is to consider in dealing with external resources.

3.3.1 Reading from external resources

When reading external resources the release of the referenced data must be ensured. In particular, pay attention to the release of resources that have been procured by means of the dialog or context object. It should be noted as a rule that the memory required for the read resource is twice as large as the needed disk space from the resource itself.

Example:

```
...

function main() {

    var xlsTemplate=Context.getFile(
            "xxx.XLT",Constants.LOCATION_SCRIPT);

    var oWorkbook = Context.createExcelWorkbook(
            Context.getSelectedFile(),xlsTemplate);


    var sheet = oWorkbook.getSheetAt(0);

    // process sheet
    ...
}
...
```

```
...

function main() {

    var xlsTemplate=Context.getFile(
            "xxx.XLT",Constants.LOCATION_SCRIPT);

    var oWorkbook = Context.createExcelWorkbook(
            Context.getSelectedFile(),xlsTemplate);

    // read data is no longer used
    xlsTemplate = null;

    var sheet = oWorkbook.getSheetAt(0);

    // process sheet
    ...
}
...
```

```
...

function main() {
```



```
      var sel = Dialogs.getFilePath (
            "",
            "*.xls!!Chart Files (*.xlc)|*.xlc|Worksheet Files (*.xls)|
            *.xls|Data Files (*.xlc;*.xls)|*.xlc; *.xls|All Files (*.*)|*.*||",
            "",
            "Select an excel file",
            0 );

      var oWorkbook = Context.createExcelWorkbook("test.xls",
sel[0].getData());

    var sheet = oWorkbook.getSheetAt(0);

    // process sheet
    ...
}
...
```

```
...

function main() {

      var sel = Dialogs.getFilePath (
            "",
            "*.xls!!Chart Files (*.xlc)|*.xlc|Worksheet Files (*.xls)|
            *.xls|Data Files (*.xlc;*.xls)|*.xlc; *.xls|All Files (*.*)|*.*||",
            "",
            "Select an excel file",
            0 );

      var oWorkbook = Context.createExcelWorkbook("test.xls",
sel[0].getData());

    var sheet = oWorkbook.getSheetAt(0);

    // process sheet
    ...
}
...
```

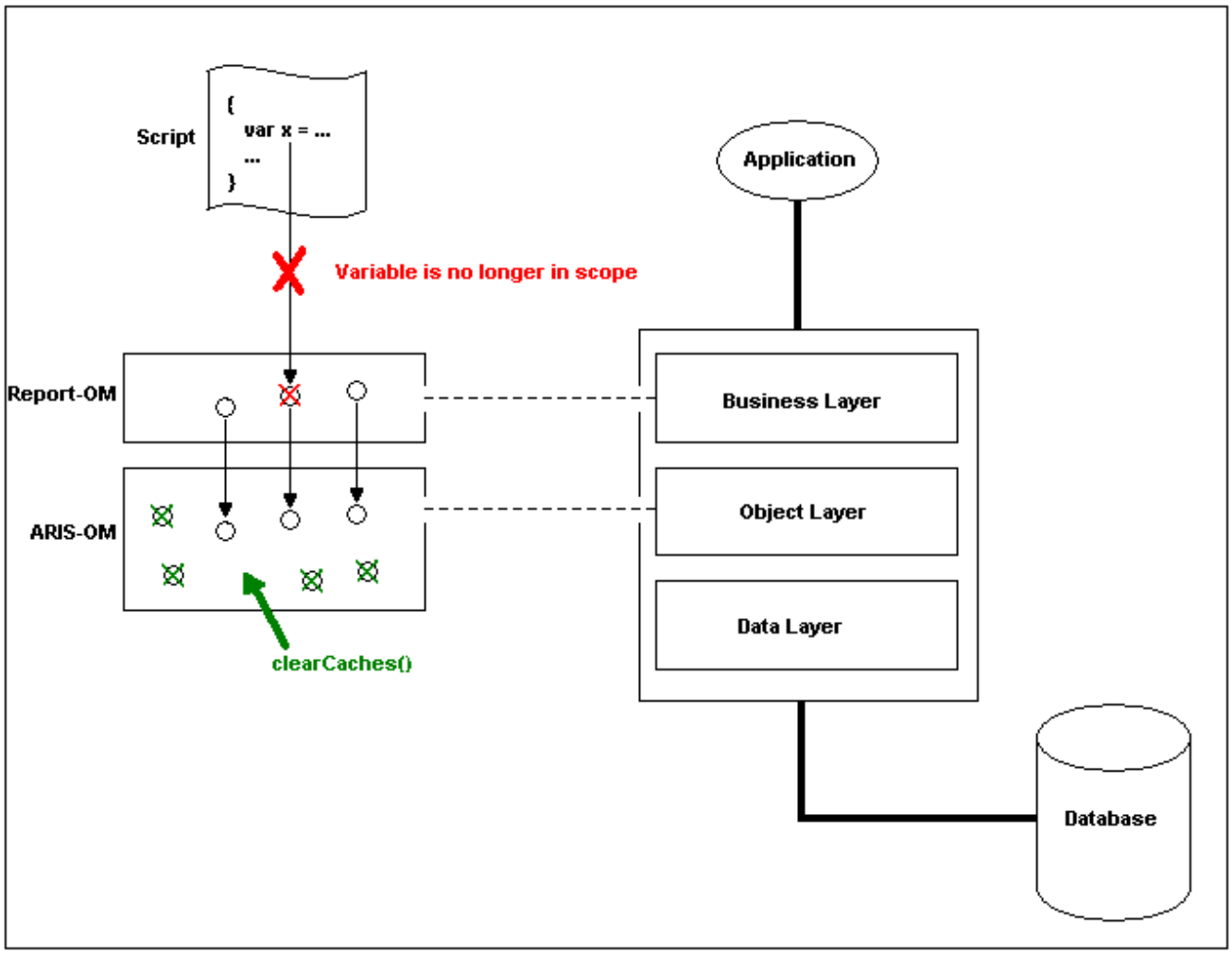
```
...

function main() {

    var sel = Dialogs.getFilePath (
            "",
            "*.xls!!Chart Files (*.xlc)|*.xlc|Worksheet Files (*.xls)|
            *.xls|Data Files (*.xlc;*.xls)|*.xlc; *.xls|All Files
(*.*)|*.*||",
            "",
            "Select an excel file",
            0 );

    var oWorkbook = Context.createExcelWorkbook("test.xls", sel[0].getData());

    // read data is no longer used
```

```
    sel = null;

    var sheet = oWorkbook.getSheetAt(0);

    // process sheet
    ...
}
...
```

3.4 Using more effective methods

The following section describes methods that should be foreseen to bring the greatest improvements here. In most of the examples below it is shown how it was implemented before and how it was updated afterwards.

3.4.1 Database.clearCaches()



If a variable is no longer in scope the referenced data in the "Report OM" which belongs to are released. This, however, does not mean that the referenced data in the "ARIS-OM" are also released.

To release them, too, the database method "`clearCaches()`" must be called explicitly. But this is only necessary if the report holds large amounts of data such as a group with all their models, objects,...

Attention:
In combination with the use of the database method "`Save()`" and the parameter "`SAVE_ONDEMAND`" (compare: ArisData.Save()) it have to be ensured that all desired objects have been stored in the database before they are removed from the cache by using "`clearCaches()`".

Example:

```
...
var oGroups = ArisData.getSelectedGroups();
for (var i = 0; i < oGroups.length; i++) {
    ArisData.getActiveDatabase().clearCaches();

    var oModels = oGroups.ModelList();
    ...

}
...
```

3.4.2 ArisData.Unique(Object[] aObjects)

This method ensures that the array of ARIS objects (only for these and not for Strings, …) does not contain any duplicate items. So do not implement this functionality by your own.

Example:

```
...
function getUniqueList(p_objList) {
    p_objList = p_objList.sort();
    if (p_objList.length > 1) {
        var i = p_objList.length-1;
        for (i = p_objList.length-1; i > 0; i--) {

            if (p_objList[i].IsEqual(p_objList[i-1])) {
                p_objList.splice(i,1);
            }
        }
    }
    return p_objList;
}
...
```

```
...

function getUniqueList(p_objList) {
    return ArisData.Unique(p_objList);
}
...
```

In general, it should always be considered whether to call the method is really necessary and whether the list can contain any duplicate at all.
The call of this method makes sense for example in the following cases:

- If you have a list of models, starting with occurrences
- If you have a list of (object) definitions, starting with (object) occurrences
- If you have a list of (object) definitions, starting with models

But it makes no sense for example in the following cases because these lists can implicitly contain no duplicates:

- If you have a list of (sub)groups, starting with a group
- If you have a list of (object) occurrences, starting with models

3.4.3 ArisData.Save()

This method manages the storage property on changing accesses on ARIS elements.

- SAVE_AUTO: (Default setting) All changes are optimally stored in the database. In contrast to SAVE_IMMEDIATELY, objects are stored in larger blocks just early enough before the next read operation occurs. Resets "SAVE_ONDEMAND" to "SAVE_AUTO"
- SAVE_IMMEDIATELY: All changes will be stored immediately into the database. Resets "SAVE_ONDEMAND" to "SAVE_IMMEDIATELY".
- SAVE_ONDEMAND: All subsequent changes to ARIS elements will be stored to the database when Save(Constants.SAVE_NOW) is invoked for the next time.
- SAVE_NOW: Stores all changes after Save(Constants.SAVE_ONDEMAND). The storage mode SAVE_ONDEMAND remains

It is only necessary to use the adapted storage behavior (SAVE_ONDEMAND) if large amounts of data should be stored. **In this case you should also try to separate reading and writing sections in your report as far as possible.**
If you decide to use the adapted storage behavior (SAVE_ONDEMAND) in your report we suggest using it for the whole report and the concrete storing (SAVE_NOW) at all relevant times.
Additionally you should call ‘SAVE_NOW’ at the end of the report (and maybe also at further times when you are not really sure of report behavior) to ensure correct saving.
If components (like Merge) are used in a report the data is (internally) automatically stored before executing to ensure that all data is available for this component.
In general save operations of models are expensive. Therefore models should be stored as one block – if possible it even makes sense to store multiple models (e.g . up to 50 models) in one block

Ensure that you don't delete and create the same occurrence or item in one storage block (not a must)

Attention:

- Unsaved items will not be found for example by Database.Find() and similar commands.
- In combination with the use of the database method "`clearCaches()`" (compare: Database.clearCaches()) and the adapted storage behavior (SAVE_ONDEMAND) it have to be ensured that all desired objects have been stored in the database before they are removed from the cache.
- If you work on databases which you have opened via "`ArisData.openDatabase()`" you have to change the storage behavior of this database. You can do this by calling "`<database>.getArisData().Save()`"
- "`ArisData.Save()`" only manages the behavior of the login database.

Example:

```
...
ArisData.Save(Constants.SAVE_ONDEMAND);
...
/* Block of changes which should be stored */
...
ArisData.Save(Constants.SAVE_NOW);
...
ArisData.Save(Constants.SAVE_IMMEDIATELY);
...
```

3.4.4 Group.ModelList(boolean bRecursive, int[] modelTypeNums)

This method returns a list of all models contained in this group and (optionally) all subgroups. Additionally the requested model types can be defined.

Example:

```
...
var oGroup = ArisData.getSelectedGroups()[0];
var oAllGroups = getAllGroups([oGroup]);
var oModels = getModels(oAllGroups, Constants.MT_EEPC);
...

function getAllGroups(p_oParentGroups) {
    var oAllGroups = new Array();

    for(var i = 0; i < p_oParentGroups.length; i++) {
        oAllGroups = oAllGroups.concat(p_oParentGroups[i]);
        var oChildGroups = p_oParentGroups[i].Childs();
        if(oChildGroups.length > 0) {
            oAllGroups = oAllGroups.concat(getAllGroups(oChildGroups));
        }
```

```
    }
    return oAllGroups;
}

function getModels(p_oGroups, p_nModelType) {
    var oModels = new Array();

    for (var i = 0; i < p_oGroups.length; i++) {
        oModels = oModels.concat(p_oGroups[i].ModelListFilter(p_nModelType));
    }
    return oModels;
}
...
```

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList(true, Constants.MT_EEPC);
...
```

3.4.4.1 Group.ModelList(…, ISearchItem p_searchSpec)

This method offers a fast way to determine models contained in this group and (optionally) all its child groups where the models need to match the given criteria.

Example:

```
...

var currentDB = ArisData.getActiveDatabase();
var currentLocale = Context.getSelectedLanguage();
var searchItem = currentDB.createSearchItem(Constants.AT_NAME, currentLocale,
                                            "A*", Constants.SEARCH_CMP_EQUAL,
                                            false, true) //non-case-sensitive
+
                                                         //search using
wildcards
var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList(true,
                               [Constants.MT_EEPC, Constants.MT_ORG_CHRT],
                               searchItem);
...
```

3.4.5 Group.ObjDefList(boolean bRecursive, int[] objectTypes)

This method returns a list of all object definitions contained in this group and (optionally) all subgroups. Additionally the requested object types can be defined.
Compare: Group.ModelList(boolean bRecursive, int[] modelTypeNums)

3.4.5.1 Group.ObjDefList(…, ISearchItem p_searchSpec)

This method offers a fast way to determine object definitions contained in this group and (optionally) all its child groups where the object definitions need to match the given criteria.
Compare: Group.ModelList(…, ISearchItem p_searchSpec)

3.4.6 Model.ObjOccListBySymbol(int[] p_iSymbolNum)

This method returns the object occurrences of the model having one of the specified symbol types.

Example:

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjOccs = oModels[i].ObjOccList();

    for (var j = 0; j < oObjOccs.length; j++) {
        if (oObjOccs[j].SymbolNum() == Constants.ST_PRCS_IF ||
            oObjOccs[j].SymbolNum() == Constants.ST_FUNC) {
            ...
        }
    }
}
...
```

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjOccs = oModels[i].ObjOccListFilter(-1, Constants. PRCS_IF);
    oObjOccs = oObjOccs.concat(oModels[i].ObjOccListFilter(-1,
                                                          Constants.ST_
FUNC));
    ...
}
...
```

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjOccs = oModels[i].ObjOccListBySymbol([Constants.ST_PRCS_IF,
                                                  Constants.ST_FUNC]);
    ...
}
...
```

3.4.7 Model.ObjDefListByTypes(int[] objTypeNum)

This method returns the object definitions of the model having one of the given object types.

Example:

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjDefs = oModels[i].ObjDefList();

    for (var j = 0; j < oObjDefs.length; j++) {
        if (oObjDefs[j].TypeNum() == Constants.OT_FUNC ||
            oObjDefs[j].TypeNum() == Constants.OT_EVT) {
            ...
        }
    }
}
...
```

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjDefs = oModels[i].ObjDefListFilter(Constants.OT_FUNC);
    oObjDefs = oObjDefs.concat(oModels[i].ObjDefListFilter(Constants.OT_EVT));
    ...
}
...
```

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjDefs = oModels[i].ObjDefListByTypes([Constants.OT_FUNC,
                                                 Constants.OT_EVT]);
    ...
}
...
```

3.4.8 ConnectableDef.AssignedModels(int[] modelTypes)

This method returns the assigned models having one of the given model types.

Example:

```
...
var oGroup = ArisData.getSelectedGroups()[0];
var oObjDefs = oGroup.ObjDefList();
for (var i = 0; i < oObjDefs.length; i++) {
    var oAssignedModels = oObjDefs[i].AssignedModels();

    for (var j = 0; j < oAssignedModels.length; j++) {
        if (oAssignedModels[j].TypeNum() == Constants.MT_EEPC ||
            oAssignedModels[j].TypeNum() == Constants.MT_EEPC_COLUMN ||
            oAssignedModels[j].TypeNum() == Constants.MT_EEPC_ROW) {
            ...
        }
    }
}
...
```

```
...
var oGroup = ArisData.getSelectedGroups()[0];
var oObjDefs = oGroup.ObjDefList();
for (var i = 0; i < oObjDefs.length; i++) {
    var oAssignedModels = oObjDefs[i].AssignedModels([Constants.MT_EEPC,
                                                      Constants.MT_EEPC_COLUMN,
                                                      Constants.MT_EEPC_ROW]);
    ...
}
...
```

3.4.9 Using userdefined symbols

When using user-defined symbols it must be considered that the type number may change each time the server is restarted and only the “TypeGUID” never changes. For this reason, the type number has always to be determined from the “TypeGUID” by using the filter method “`UserDefinedSymbolTypeNum()`”.

Example:

```
...
var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjOccs = oModels[i].ObjOccListFilter(-1, 65820);
    ...
}
...
```

```
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjOccs = oModels[i].ObjOccList();

    for (var j = 0; j < oObjOccs.length; j++) {
        if (oObjOccs[j].SymbolNum() == 65820) {
            ...
        }
    }
}
...
```

```
var cSYMBOL = ArisData.ActiveFilter().UserDefinedSymbolTypeNum("90fb6e70-
                1660-11db-688f-001485f6fd0c"); //65820
...

var oGroup = ArisData.getSelectedGroups()[0];
var oModels = oGroup.ModelList();
for (var i = 0; i < oModels.length; i++) {
    var oObjOccs = oModels[i].ObjOccListBySymbol([cSYMBOL]);
    ...
}
...
```

3.4.10 Using userdefined attributes

When using userdefined attributes it must be considered that the type number may change each time the server is restarted and only the “TypeGUID” never changes. For this reason, the type number has always to be determined from the “TypeGUID” by using the filter method “`UserDefinedAttributeTypeNum()`”.
Compare: Using userdefined symbols

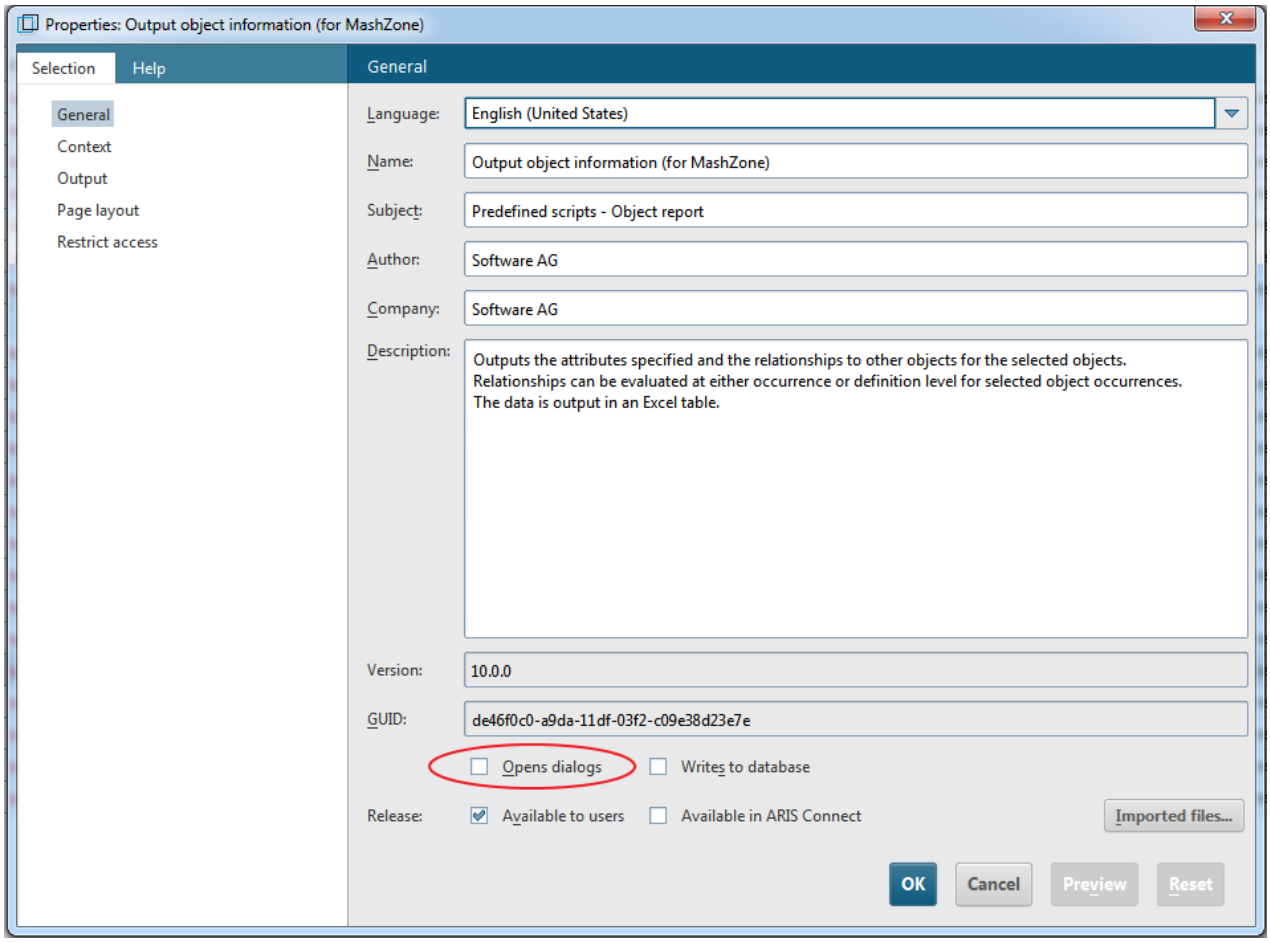
3.4.11 Using userdefined models

When using userdefined models it must be considered that the type number may change each time the server is restarted and only the “TypeGUID” never changes. For this reason, the type number has always to be determined from the “TypeGUID” by using the filter method “`UserDefinedModelTypeNum()`”.
Compare: Using userdefined symbols

3.4.12 Preparing report for use with Report Scheduler

Reports, which should be scheduled, may have no dialogs for user interaction.
For this reason, all method calls of the dialog object have to be removed or at least to be commented out.

Additionally, the report flag "Opens dialogs" have to be deactivated, so that the report is available in the selection for the scheduling:



3.4.13 ArisData.openDatabase()

You can log into the same DB as another user or with another filter or into further databases.
In this case don't forget to close them afterwards by using "`<database>.close()`".

3.4.14 Clean up after an error!

Don't forget to clean up after an error – especially after "`openDatabase`" commands.
And also handle output file content in case of an error: "`Context.setScriptError( Constants.ERR_CANCEL )`"

3.4.15 Always use try / catch for error handling

You can report detailed error logs to the user by using this code:

```
try {
    <your code which might throw exceptions>
}
catch(ex) {
    var line = ex.lineNumber
    var message = ex.message
    var filename = ex.fileName
    var exJava = ex.javaException
```



```
    if(exJava!=null) {
        var aStackTrace = exJava.getStackTrace()
        for(var iST=0; iST<aStackTrace.length; iST++) {
            message = message + "\n" + aStackTrace[iST].toString()
        }
    }
    Dialogs.MsgBox("Exception in file "+filename+", line "+line+":\n"+message
)
}
```

3.4.16 Use Context.setProperty("model-as-emf", true)

Use "`Context.setProperty("model-as-emf", true)`" once (before writing model graphics to the report). This makes model graphics in PDF output more brilliant.

The images in the result document can be zoomed without loss or pixel artifacts and you get smaller PDF result files.

