Scripting in Intellicus

Version: 18.1

intellicus

Copyright © **2018** Intellicus Technologies

This document and its content is copyrighted material of Intellicus Technologies.

The content may not be copied or derived from, through any means, in parts or in whole, without a prior written permission from Intellicus Technologies. All other product names are believed to be registered trademarks of the respective companies.

Dated: October 2018

Acknowledgements

Intellicus acknowledges using of third-party libraries to extend support to the functionalities that they provide.

For details, visit: http://www.intellicus.com/acknowledgements.htm

Contents

1 Scripting 4

Scripts in SQL 5

Scripts in Parameter Definition 11

Report Scripting 12

Using your Custom Functions in Script 14

Parsing Data from Complex Fields 18

2 Script Editor 25

Context-Sensitive Help 25

Accessing Fields 26

Accessing Layout Objects 26

Compiling Scripts 26

Find and Replace 27

3 Examples 28

Conditional Formatting 28

Conditional Suppressing Of Rows 29

Conditional Calculation 30

1 Scripting

Scripting in Intellicus enables you to control behavior of various report controls at run time. For example, you can use script to decide at runtime:

- A field should be displayed on report or not
- A report section should be rendered or not
- A parameter on Input Parameter Form should be displayed or not
- Validate the parameter value entered by the user
- Change the report SQL dynamically (for example, append filter clause, or specify table name at run time)

Broadly scripting can be classified in three categories:

- **Scripts for SQL:** This allows dynamic query generation through support of executable script blocks within the query.
- **Scripts for Parameter:** This allows validation of parameter values by writing script that can assert true or false for every parameter value on events like on submission of Input Parameter Form. It also allows changing UI elements of Input Parameter Form dynamically.
- **Scripts for Report:** This allows changing report layout dynamically at various events (like onReportStart(), onDataFetch() etc.) to control/change the way report should render.

You can use the following objects and their properties to control the behavior of the report at runtime. These report objects are accessible in a specific hierarchy as explained below:

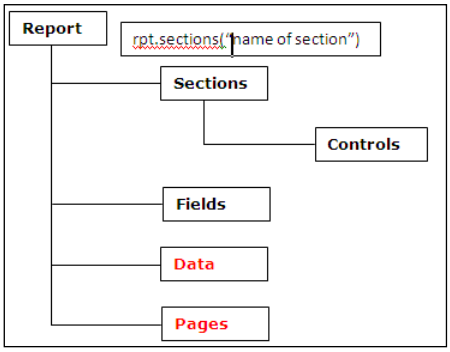

Figure 1: Report Execution Script

The objects or events in the code editor are dependent on each other, for one object there is a specified set of events and vice-versa.

Scripts in SQL

Intellicus supports writing an executable script block in queries. You can insert the script code anywhere in the SQL. It should be within start and end tags (<@% and %@>). This executable block returns a string, which replaces the script block in the SQL query.

The block is enclosed in tags:

- Script block start tag: <@%
- Script block end tag: %@>

A script block may have a valid java script code, including all java script programming constructs E.g. logical conditions (if...else clause) on which the block can take decisions.

You can access various objects from within the script block based on run time context.

Objects accessible from SQL script

Parameters

To refer any parameter within this executable block user has to use the object model i.e. he/she has to write

params("parameterName").getXXX()

Where, parameterName is the name of the parameter and getXXX are various getter methods for different attributes of the input parameter:

- **getValue():** Returns value of the parameter. For example, params("prmCity").getValue(). Return type is string.
- **getDataType():** Returns data type of the parameter. It can be: NUMBER (for numeric parameters), CHAR (for character parameters), DATE (for date type parameters) and BOOL (for Boolean parameters). It will return NULL if parameter datatype is unknown.

If you do not append any attribute (for example, params("prmEmpl"), it is assumed as parameter itself. You can compare this to null to check existence of the parameter.

Note: Only getValue() is supported for SYSTEM type parameters or the parameters whose definition is not available.

Examples

Example 1. Check if the parameter prmEmpNo exists and its value is blank. If so, append a condition to where clause in the SQL.

```
Select * from emp where 1=1
<@%
if(params("prmEmpNo") != null &&
params("prmEmpNo").getValue() != "")
```

```
{
              return " AND empno in (<%prmEmpNo%>)";
}
else
{
              return "";
}
%@>
```

Example 2. Check if prmSelectTable exists. If it does, return value of the parameter as table name. If it does not, return table name as emp.

```
Select * from
<@%
if(params("prmSelectTable") == null)
{
return " emp ";
}
else
{
return params("prmSelectTable").getValue();
}
%@>
```

Example 3. Check if parameter p_EmpNo exists. If it does not exist, call procedure with any arguments. If it exists, then it call the procedure that takes one argument and it passes this parameter as the argument of procedure.

```
<@%
if(params("p_EmpNo") != null)
{
    return "EMP_DETAILSWITHARG <%p_EmpNo%>"
}
return "EMP_DETAILS";
%@>
```

Report Layout

To refer report layout from SQL script block, you have to use the object model as:

rpt.layout.getXXX()

You can add script block to an SQL inside a standard report or to an SQL inside a Query Object. In both cases, when SQL is verified in SQL Editor for compile errors and result set, the report layout object may not be available to the script.

When the report is executed – Preview, Run, or Run Ad hoc report using Query Object, then report layout object is available to the script.

So, you essentially check whether your SQL is running for result set or for report run, before accessing report layout object.

```
var arl = rpt.layout.getArl();
if (arl != null)
{ var fields = arl.getFields();
}
OR
var irl = rpt.layout.getirl();
if (irl != null)
{ var fields = irl.getFields();
}
```

Below is a long example of accessing various parts of report layout to dynamically construct an optimized SQL for an Ad hoc report.

Example 4. Check if SQL is running for an Ad hoc report. If yes check the fields used in various constructs of Ad hoc report layout – Select, Filters, Sort, Group, Chart, and cross tab. Create select clause of SQL with only the fields used in any of the ad hoc report construct.

```
SELECT
<@%
// check if report run
var arl = rpt.layout.getArl();
if (arl != null)
{
var fields = arl.getFields();
var myArray = new Array();
var str = "";
var i = 0;

for (i = 0; i < fields.getCount(); i++)
{
var field = fields.get(i);

if ("TRUE" == field.getDisplayEnabled())
{
myArray[field.getName()] = field.getName();

}

}
// check Group section
var grpCount = rpt.layout.getGroupsCount();
for (var grpIndex = 0; grpIndex < grpCount; grpIndex++)
{
var grp = rpt.layout.getGroup(grpIndex);
myArray[grp.getFieldName()] = grp.getFieldName();
```

```
}
// check Filter section
var fltrCount = rpt.layout.getFiltersCount();

for (var fltrIndex = 0; fltrIndex < fltrCount ; fltrIndex++)
{
var fltr= rpt.layout.getFilter(fltrIndex);
myArray[fltr.getFieldName()] = fltr.getFieldName();

}
// check sort section
var sortCount = rpt.layout.getSortParamsCount();

for (var sortIndex = 0; sortIndex < sortCount ; sortIndex++)
{
var sortParam = rpt.layout.getSortParam(sortIndex);
myArray[sortParam.getFieldName()] = sortParam.getFieldName();

}
// check Highlighting section

var adhcConditions = arl.getAConditions();

if(adhcConditions != null)
{

var adhcConCount= adhcConditions.getCount();
for (var adhcConIndex= 0; adhcConIndex< adhcConCount; adhcConIndex++)
{
var adhcCondition = adhcConditions.get(adhcConIndex);
var clause = adhcCondition.getAClause(0);
myArray[clause.getFieldName() ] = clause.getFieldName() ;
}
}
// check chart section

var chart = arl.getChart();

if(chart != null)

{

var XCount = chart.getChartXAxisCount();
for (var index = 0; index < XCount; index++)
{
var XAxis = chart.getChartXAxis(index);
myArray[XAxis.getFieldName()] = XAxis.getFieldName();
}

var seriesEnum = chart.getSeriesEnum()

var seriesCount = seriesEnum.size();
```

```
for (var index = 0; index < seriesCount; index++)
{
var series = seriesEnum.get(index);
myArray[series.getFieldName()] = series.getFieldName();

}
}

// check Matrix section
var matrix = arl.getMatrix();

if(matrix != null)
{

var summaries = matrix.getMatrixSummaries();

var summarySize = summaries.size();



for (var index = 0; index < summarySize; index ++)
{
var summary = summaries.get(index);
myArray[summary.getFieldName()] = summary.getFieldName();

}

var xAxes = matrix.getMatrixXAxes();

var xAxesSize = xAxes.size();



for (var index = 0; index < xAxesSize; index ++)
{
var xAxis = xAxes.get(index);
myArray[xAxis.getFieldName()] = xAxis.getFieldName();

}

var yAxes = matrix.getMatrixYAxes();

var yAxesSize = yAxes.size();



for (var index = 0; index < yAxesSize; index ++)
{
var yAxis = yAxes.get(index);
myArray[yAxis.getFieldName()] = yAxis.getFieldName();

}
}
```

```
var FormulaField= new Array("Ccy1amount1","Ccy1amount2");
for (var cnt= 0; cnt< FormulaField.length; cnt++)
{
myArray[FormulaField[cnt]] = FormulaField[cnt];
}



for(var key in myArray)
{
str += myArray[key]+",";
}
return str.substring(0, str.length - 1);
}
else
{
return " * ";
}
%@>
FROM Call_Log
```

Insert script block in SQL

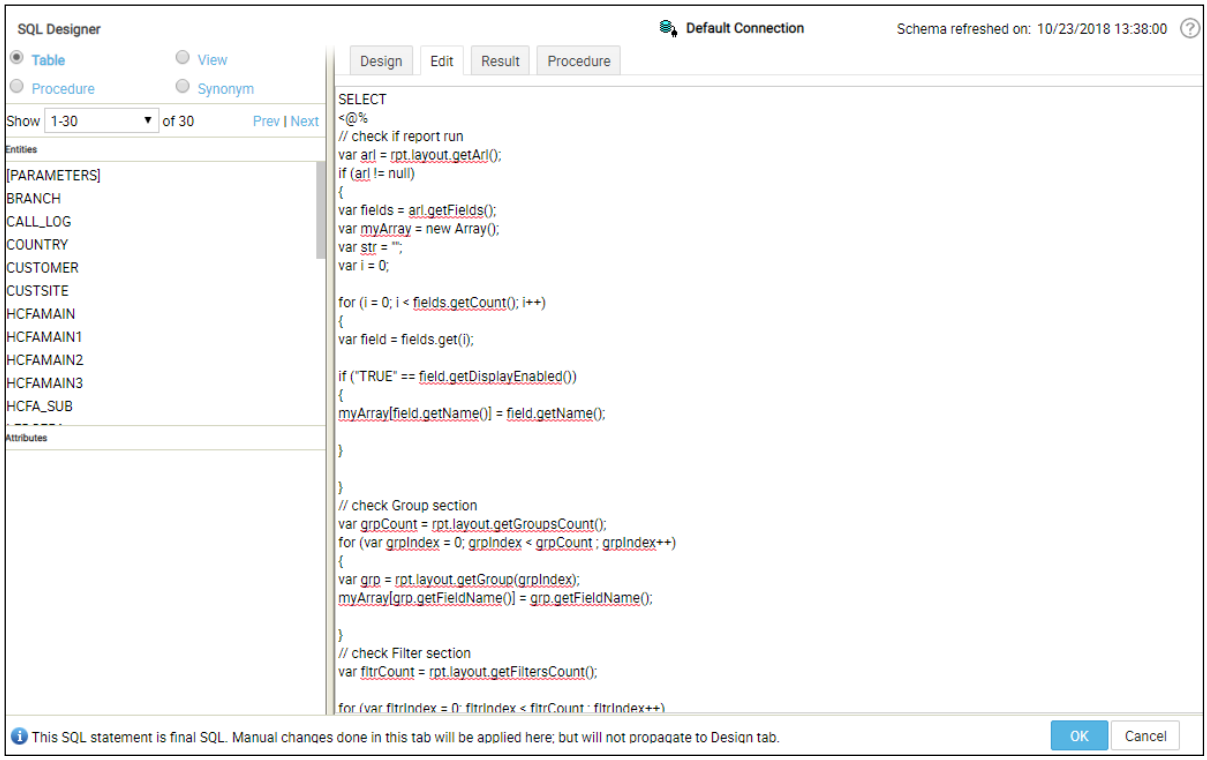

Figure 2: Script Editor with the query

Scripts in Parameter Definition

Intellicus provides scripting support for parameter value validation for all parameter types (combo, textbox, radio, checkbox, etc.) Script will return true (parameter value is valid) or false (parameter value is invalid). If validation fails, it can also set an error message. You can add script under the Advanced tab of Parameter Object Editor or select Scripting under Tools menu in Studio application.

Scripting is available at:

- Form level (can be defined on Parameter Form layout dialog on Studio)
- Parameter level (defined on Parameter detail dialog box)

Input Parameter Form Level

When a report has user parameters, **Input Parameter Form** is presented to the user to enter run time parameter values. Use Form level scripting to validate parameter values entered on IPF.

You can open **Script Editor** dialog from **Parameter Form Layout** dialog to write the script.

At IPF level (Form level), OnSubmit() event is supported. It means script is executed when user clicks OK / Run button on IPF. A user can write any valid java script code within this function. This function MUST return a boolean value.

If all the parameter values are valid, it should return true, if one or more parameter values are invalid, it should return false. If script returns false then an error message will be displayed to the user and he/she would not be allowed to submit the IPF and execute the report. You can also set the error message that should be displayed to the user.

Script can access any parameter of the report. This includes parameter objects (even if not imported) and global business parameters. This will be Read-only access (parameter objects and global business parameters).

Note: In case of JavaScript error, Report Server will respond with ERROR.

Parameter Level

You can add a validation script for a parameter being designed on **Parameter Detail** dialog. You can open **Script Editor** dialog from here to write the parameter validation script.

At parameter level, OnChange() event is supported. It means, validation script will be executed when:

User types in a value for the parameter (for input type TEXT), or

Selects/Unselects value from the parameter combo/list/tree.

Checks/ Un-checks a check box.

Validation script written at parameter level can access other report parameters. It can also access parameter objects (even if not imported) and global business parameters. This will be Read-only access.

If the parameter value is valid, script should return true. If it is invalid, script should return false. You can also set an error message that should be displayed when parameter validation fails. Report will not be generated if parameter validation fails.

Using Parameter Level script, you can also modify attributes of parameters and control/change the way IPF is shown. For example, if paramA is invalid, disable paramB. IPF will reload parameters that are affected by the script.

Using script, you can also change parameter UI control attribute:

ENABLE: READ / MODIFY at Parameter level and READ ONLY at Input Parameter Form level.

Note: If such a report is scheduled, then IPF is presented at the time of setting the schedule and so, script will be executed only at the time of scheduling (and not at every time when schedule executes).

In case of scheduled report execution, IPF is not displayed. Hence, script will be executed at the time of saving of schedule tasks. Script will not be executed at report run time.

Report Scripting

Intellicus Studio provides facility to customize the events using the Script Editor. Coding is done in JavaScript syntax, and is event based. The following table lists the events that are supported/provided at report level.

Report level events

Important: You need to make sure that the code pertaining to a particular event is written within the appropriate event only.

Event	Description
OnReportStart	This event is fired before report objects such as fields and sections are constructed. (Before the report starts to execute itself)
OnReportEnd	This event fires after execution of the report.
OnPageStart	This event is fired before displaying every page. This event does not ensure that the previous page's display has been completed.

Event	Description
OnPageEnd	This event is fired after the display of every rendering.
OnHyperlink (Button, Link)	This event is fired when the end user clicks on the hyperlink on the report output. The mouse button and the URL are passed in as parameters.
OnDataInitialize	This event is fired after the report is loaded, or SQL statement is fired or SQL fields are created. In this event, new report fields can be added and existing fields removed.
OnFetchData (eof)	This event is fired after each row of the report SQL statement is fetched from the database. In this event, the report field's data can be accessed for calculation and manipulation.
OnNoData	This event is fired when zero rows are fetched from the report SQL statement.
OnPrintProgress (PageNumber)	This event is fired when the printing progresses to next page. The printing process refers to sending the page data to the printer driver and not printing the page on the paper.
OnError (Number, Description, Scode, Source, HelpFile, HelpContext, CancelDisplay)	This event is fired when any error occurs while running the report. In this event, the error messages can be changed or can be suppressed.

Section level events

Intellicus passes three events at section level for all sections. The sequence of events depends on the summary objects and their section dependencies. The event sequence is:

OnFormat

This event is fired after the data is loaded and bound to the fields, but before the section is laid out for printing. You can use this event to modify the layout of the section or any of the controls on it.

Note: This is the only event in which you can modify the height of the section.

OnBeforePrint

It is fired before the section is rendered to the 'Canvas' object. You can use this event to modify the values of the controls before they are printed. Any changes that are made here will not affect the height of the section.

Important: It is recommended, NOT to access any report fields in this event. If you need the value of a field in this event, you should use a hidden control to store the value temporarily in the format event.

OnAfterPrint

It is fired after the section is rendered to the 'Canvas' object. You can use this event to update any counters that you need to use after the report is completed.

Using your Custom Functions in Script

You can use external (custom) Java library classes and objects in scripts.

For example, there is a database that holds XML as data. User can create a formula field, which will parse this XML and return data values from it. To achieve this, user will access third-party xml parser classes.

Few applications where user may need to use custom Java objects can be:

- Script can use some external java libraries for XML Parsing.
- Script can pass some parameters to an external custom Java class which connects to a web service using the given parameters and returns the resultant data, e.g. a web service which returns the live price of a stock or current weather condition of a city.
- Script can use JDBC APIs to connect to a database and do tasks listed in above point.
- Script can use java or external APIs to read external file (.txt, .xls etc.) data.
- Script can log required information during report events.

Prerequisite

The third party (custom) class(s) to be accessed from Intellicus script needs to be included in Intellicus report server class path.

Note: Easiest way to do this is to make a jar containing custom class(s) and place it in the Report Engine's lib folder

<Install path>\Intellicus\ReportEngine\lib

To access a class in script

You can access a custom class in a script by using a keyword **Packages** (case sensitive) that will be followed by the class name or the entire package pattern (if the class is in a specific package) including class name.

For Example,

var classObj = new Packages.MyClass();

OR

var classObj = new Packages.mypackage.MyClass();

Instance of the class once generated, can be used to invoke any of the method (that contains logic, for e.g. XML parsing code) of the custom class from the script.

var resultValue = classObj.parseXMLData(xmlData);

For the package pattern starting with “java”, “com” or “org”, Packages keyword can be omitted.

var fileObj = new java.io.File(filepath);

var domObj = new org.apache.xerces.parsers.DOMParser();

Importing class in a script

You can import all required packages or classes once in the beginning and then user can use the class directly with the class name anywhere in the script. To achieve this, you can use any of the following techniques.

Using importPackage(packagename)

Use importPackage() to import all classes within a package.

@param packagename: Entire package path whose classes needs to be imported in a script.

You can import multiple packages by specifying all package paths separated by comma as argument of importPackage function.

Import statement should be written before first usage.

If the package starts with java, org or com, You can omit Packages key word if the package starts with **java**, **org** or **com**.

Examples:

Importing Single Package

```
importPackage(Packages.java.io);
var fileObj = new File(“d:\temp.txt”);
if(fileObj.exists())
{
//do something}
else
{
// do something else}
```

Importing Multiple Packages

```
importPackage(Packages.java.io, Packages.java.lang); var fileObj = new
File(“d:\temp.txt”);
```

Using importClass(classname)

If a user needs to import specific class/classes within a package, importClass() can be used.

@param classname: Path of the class (including entire package path) which needs to be imported in a script.

Multiple classes can also be imported by specifying all class path names separated by comma as an importClass function argument.

Examples:

Importing Single Class

```
importClass(Packages.java.io.File);
var fileObj = new File(“d:\temp.txt”);
if(fileObj.exists())
  {
            java.lang.System.out.println(“File exists...!”);
  }
else
  {
            java.lang.System.out.println(“File doesn’t exist...!”);
  }
```

Importing Multiple Classes

```
importClass(Packages.java.io.File,Packages.java.lang.System);
var fileObj = new File(“d:\temp.txt”);
if(fileObj.exists())
{
System.out.println(“File exists...!”);
}
else
{
System.out.println(“File doesn’t exist...!”);
}
```

Using JavaImporter(path)

If a user needs to import entire package or a specific class, JavaImporter() can be used.

@param path: Path of the package/class (including entire package path) which needs to be imported in a script.

Multiple packages/classes can also be imported by specifying full path names for all packages/classes separated by comma as JavaImporter function argument.

Example,

Importing Packages

```
var importer = JavaImporter(Packages.java.io,Packages.java.lang);
var fileObj;
with(importer)
{
fileObj = new File(“d:\temp.txt”);
}
if(fileObj.exists())
{
System.out.println(“File exists...!”);
}
else
{
System.out.println(“File doesn’t exist...!”);
}
```

Importing Classes

```
var importer = JavaImporter(Packages.java.io.File,Packages.java.lang.System);
var fileObj;with(importer)
{            fileObj = new File(“d:\temp.txt”)
;}
if(fileObj.exists())
{
System.out.println(“File exists...!”);
}
```

```
else
{
System.out.println("File doesn't exist...!");
}
```

Importing Packages & Classes together

```
var importer = JavaImporter(Packages.java.io.File,Packages.java.lang);
with(importer)
{
var fileObj = new File("d:\temp.txt");
if(fileObj.exists())
{
System.out.println("File exists...!");
}
else
{
System.out.println("File doesn't exist...!");
}
}
```

Parsing Data from Complex Fields

Intellicus supports parsing complex data stored in data field.

The complex data could be passed as:

- XML
- CSV
- Regular expression
- Fixed length

You can specify various functions to extract values from complex data fields under the **Formula** tab.

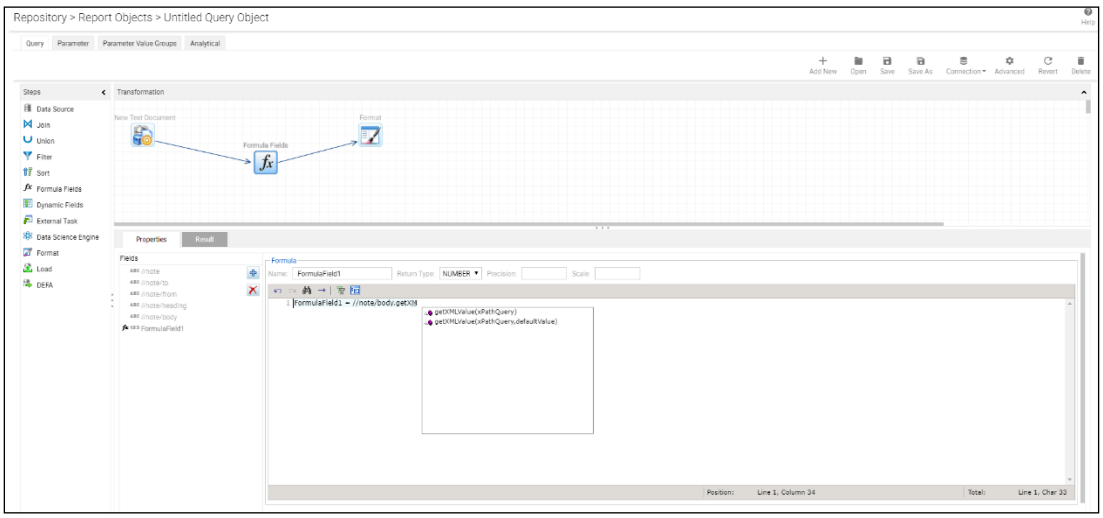

Figure 3: Formula Tab

Certain functions that are supported are given below:

Fields with XML data

Method	<COLUMN_NAME>.**getXMLValue**(String xPathQuery)
Return	String
Usage	FormulaField1 = DATA_XML.getXMLValue("\PARAMETER\NAME[1]");
Input	<PARAMETER> <NAME>param1</NAME> <NAME>param2</NAME> <PARAMETER>
Output	param1

Method	<COLUMN_NAME>.getXMLValue(String xPathQuery, String defaultValue)
Return	String
Usage	FormulaField1 = DATA_XML.getXMLValue("\PARAMETER\VALUE", "NONE");
Input	<PARAMETER> <NAME>param1</NAME> <NAME>param2</NAME> <PARAMETER>
Output	NONE

Fields with CSV data

Method	<COLUMN_NAME>.getCSVValue(int columnIndex)
Return	String
Usage	FormulaField1 = DATA_CSV. getCSVValue(2);
Input	John,29,USA,302765
Output	29

Method	<COLUMN_NAME>.getCSVValue(int columnIndex, String separator, String enclosedBy, String defaultValue)
Return	String
Usage	FormulaField1 = DATA_CSV. getCSVValue(2,"|","\"","NONE");
Input	John|29|USA|302765
Output	29

Method	<COLUMN_NAME>.getCSVValues()
Return	String array
Usage	FormulaField1 = DATA_CSV. getCSVValues();
Input	John,29,USA,302765
Output	[John,29,USA,302765]

Method	<COLUMN_NAME>.getCSVValues(String separator, String enclosedBy)
Return	String array
Usage	FormulaField1 = DATA_CSV. getCSVValues(“|”,”\””);
Input	John|29|USA|302765
Output	[John,29,USA,302765]

Fields with Regular Expression data

Method	<COLUMN_NAME>.getREGValue(String regexPattern)
Return	String
Usage	FormulaField1 = DATA_REG. getREGValue (“(\b(?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)((?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)(\d+)(\^A)(\[[^\]]*\])(\^A)(200|500)(\^A)(\d+)(\^A)([a-zA-Z0-9-_]+)(\^A?)(1|0|-)(\^A?)(-|\d+)(\^(\d+)?)(\^(\d+)?)(\^(\d+(\.\d+))?)(\^(\d+(\.\d+))?)(\^(\d+(\.\d+))?)(\^ASID\^B(\d+))”);
Input	10.70.110.94^A10.71.173.21^A9077^A[17/Jan/2011:18:37:27 +0800]^A200^A55^A7rd65tzq-mida-iptw-wl98-a1c8tji1wm56^1^0^716^^600.0^59.99999999994^65.99999999994^ASID^B10100
Output	10.70.110.94

Method	<COLUMN_NAME>.getREGValue(String regexPattern, int groupIndex)
Return	String
Usage	FormulaField1 = DATA_REG. getREGValue (“(\b(?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)((?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)(\d+)(\^A)(\[[^\]]*\])(\^A)(200|500)(\^A)(\d+)(\^A)([a-zA-Z0-9-_]+)(\^A?)(1|0|-)(\^A?)(-|\d+)(\^(\d+)?)(\^(\d+)?)(\^(\d+(\.\d+))?)(\^(\d+(\.\d+))?)(\^(\d+(\.\d+))?)(\^ASID\^B(\d+))”, 7);

Input	10.70.110.94^A10.71.173.21^A9077^A[17/Jan/2011:18:37:27 +0800]^A200^A55^A7rd65tzq-mida-iptw-wl98-a1c8tji1wm56^1^0^716^^600.0^59.99999999994^65.99999999994^ASID^B10100
Output	17/Jan/2011:18:37:27 +0800

Method	<COLUMN_NAME>.getREGValue(String regexPattern, int groupIndex, String defaultValue)
Return	String
Usage	FormulaField1 = DATA_REG. getREGValue (“(\b(?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)((?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)(\d+)(\^A)(\[[^\]]*\])(\^A)(200|500)(\^A)(\d+)(\^A)([a-zA-Z0-9-_]+)(\^A?)(1|0|-)(\^A?)(-|\d+)(\^(\d+)?)(\^(\d+)?)(\^(\d+(\.\d+))?)(\^(\d+(\.\d+))?)(\^(\d+(\.\d+))?)(\^ASID\^B(\d+))”,40,”Not Found”);
Input	10.70.110.94^A10.71.173.21^A9077^A[17/Jan/2011:18:37:27 +0800]^A200^A55^A7rd65tzq-mida-iptw-wl98-a1c8tji1wm56^1^0^716^^600.0^59.99999999994^65.99999999994^ASID^B10100
Output	Not Found

Method	<COLUMN_NAME>.getREGValues(String regexPattern)
Return	String array
Usage	FormulaField1 = DATA_REG. getREGValues (“(\b(?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)((?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)(\d+)(\^A)”);
Input	10.70.110.94^A10.71.173.21^A9077^A
Output	[10.70.110.94,^A, 10.71.173.21,^A,9077,^A]

Method	<COLUMN_NAME>.getREGValues(String regexPattern, String groupIndexes)
Return	String array
Usage	FormulaField1 = DATA_REG. getREGValues (“(\b(?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)((?:[0-9]{1,3}\.){3}[0-9]{1,3})(\^A)(\d+)(\^A)”,”1,3,5”);
Input	10.70.110.94^A10.71.173.21^A9077^A
Output	[10.70.110.94, 10.71.173.21,9077]

Fields with Fixed Length Strings data

Method	<COLUMN_NAME>.getFLSValue (String fixedLengthPattern)
Return	String
Usage	FormulaField1 = DATA_FLS. getFLSValue (2-3,8-11);
Input	1Mr29USJohn
Output	MrJohn

Method	<COLUMN_NAME>.getFLSValue (String fixedLengthPattern, String defaultValue)
Return	String
Usage	FormulaField1 = DATA_FLS. getFLSValue (14-17, “Not Found”);
Input	1Mr29USJohn
Output	Not Found

Method	<COLUMN_NAME>.getFLSValues (String fixedLengthPattern)
Return	String array
Usage	FormulaField1 = DATA_FLS. getFLSValues (-1:2-3,8-11:4-5:6-7);
Input	1Mr29USJohn
Output	[1,MrJohn,29,US]

2 Script Editor

Intellicus provides facility to write scripts for field properties and field events using the Script Editor. This enables you to define your own constructs for report generation. To get the **Script Editor** dialog box, click Add Script under Advanced tab of Parameter Object Editor or select Scripting from Tools menu in Studio application.

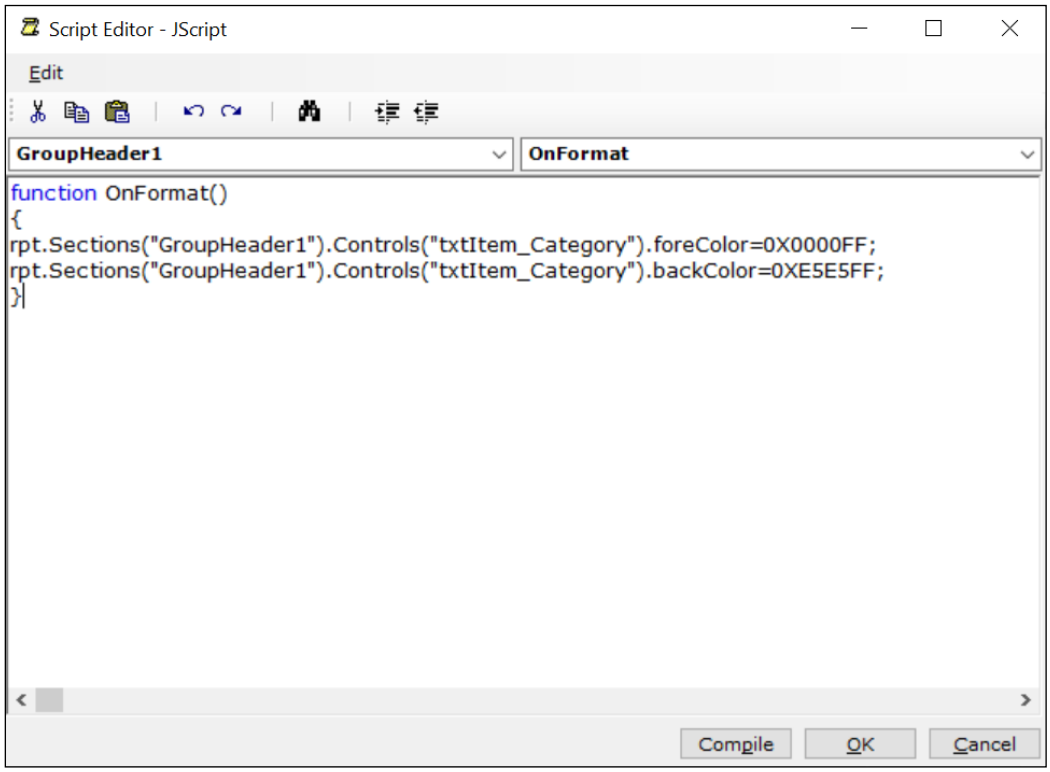

Figure 4: Script Editor

Context-Sensitive Help

The Script Editor also provides context-sensitive help that assists in correct code (syntax) formation. As you type the code in the Script Editor, the context-sensitive help keeps popping up selection list of various fields and objects that may fit into the syntax.

Figure 5: Context-Sensitive Help

Accessing Fields

You can access the report fields in 'rpt.Fields' collections. You can use this collection to write code in the Script Editor to access fields (controls) in the layout pane. Each event in the Script Editor has a specific purpose; you should not write a code that does not pertain to the object / event under which it has been written.

To add a new code under the Data Initialize event of the Intellicus Report Layout, the syntax is:

rpt.fields.add "<MyField>";

Warning: Make sure that the added field does not already exist; else, a fatal error will occur.

Code script for the value property of the fields can be given under 'OnFetchData' event of Intellicus Report Layout.

rpt.fields("SomeFieldName").value="<SomeValue>";

Accessing Layout Objects

The layout objects are the controls that are added to the report layout region. [See also: Working with Layout Editor, Chapter 3], you can access these objects through control's collection members of the sections collection.

rpt.sections("Detail").Controls("imgLogo").visible = false;

Important: You will not be able to access the database using code (scripts).

Compiling Scripts

After typing in the script, you can compile the script to make sure it will run without error and you will be able to achieve the results that you want, using the script. To compile the script, click the **Compile** button available on bottom-left side of the dialog box.

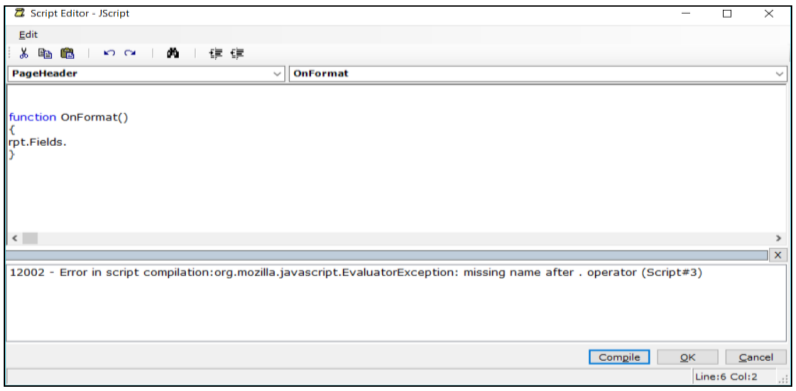

Figure 6: Syntax Error in the script

If the script has any syntax error, it is listed in a pane opening between script pane and buttons. You can remove the errors and click the **Compile** button to make sure the script is error-free.

Find and Replace

Script Editor dialog box offers Find and replace functionality. Click the Find button on the toolbar of Script Editor or press Ctrl + F on the keyboard to switch on the functionality.

You have options to **search up** and **search down**. Selecting **Match Case** will conduct a case sensitive search. Selecting **Match whole word only** will not find the words where the search string is part of a word. A click on **Find Next** button will start search.

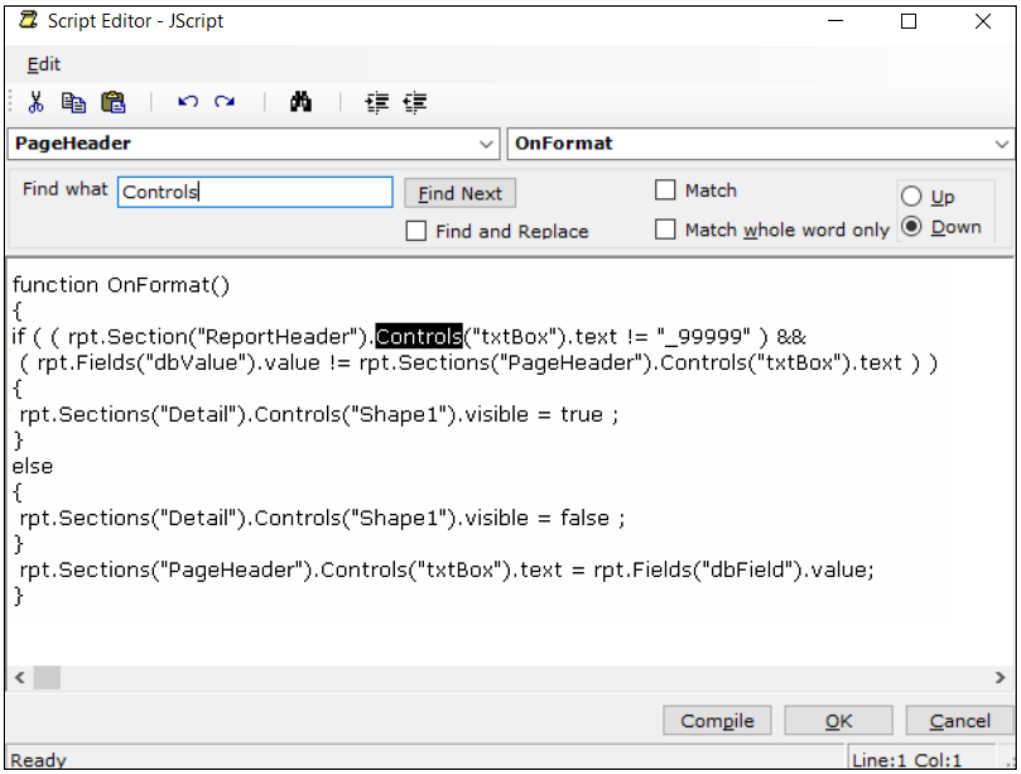

Figure 7: Find and Replace feature on Script Editor

If you want to carry out find and replace, select **Find and Replace** check box. Clicking **Replace** will replace the next occurrence of the search sting. Clicking **Replace All** will search for all the occurrences of the search string.

3 Examples

Conditional Formatting

You can achieve conditional formatting through Scripting too.

You can format a displayed row value if the values of that row satisfy a given condition. For example, if you need to compare the database field (say ‘dbfield’) with a field in the previous row, and encircle it in red if it is different, add a text box (say ‘text box’) in the report header and set its visible property (from the Properties list) as ‘False’.

Set its text property (from the Properties list) to any arbitrary value (say ‘-999’) that can never be attained by the field to be compared with.

Now, add a shape (say "shape1") around the ‘dbfield’ and set its color and shape to a red ellipse. The same can be achieved using the Script Editor as follows:

1. From the Tools menu, click Scripting.
2. Select object as Detail.
3. Select event as OnFormat.
4. Type the following JavaScript:

Object: Detail Event: OnFormat

Code:

```
function OnFormat()
   {
     if ( ( rpt.Section("ReportHeader").Controls("txtBox").text != "_99999" ) &&
          ( rpt.Fields("dbValue").value !=
rpt.Sections("PageHeader").Controls("txtBox").text ) )
               {
                 rpt.Sections("Detail").Controls("Shape1").visible = true ;
               }
               else
               {
                 rpt.Sections("Detail").Controls("Shape1").visible = false ;
               }

         rpt.Sections("PageHeader").Controls("txtBox").text =
rpt.Fields("dbField").value;
   }
```

Conditional Suppressing Of Rows

You can suppress the display of certain rows as per your requirement, like some column containing NULL values can be suppressed (hidden) from getting displayed on the report.

There are two methods to do this:

- Select the control and set the visible property (Property window) value as false, and assign a 0 (Zero) value to the height property (Property window) of the control.
- Go to **Tools** > **Scripting**; select the object as *Detail*, and the event as *OnFormat*, and write the following code:

Object: Detail Event: OnFormat

Code:

```
function OnFormat()
   {
     if ( rpt.Fields("Name").value == null )
     {
      rpt.Sections("Details").visible = false;

     }
     else
     {
       rpt.Sections("Details").visible = true;
     }

     if ( rpt.Fields("Name").value == null )
     {
      rpt.Sections("Details").height = 0;

     }
     else
     {
       rpt.sections("Details").height = 285;
     }

   }
```

Important: To dynamically change the height of a section through a program, the 'CanGrow' property (Properties list) of Detail Section should be set to 'False'.

If it is set to 'True', then the report section will override your (height) value to adjust the height of the section.

Conditional Calculation

You can calculate values in the report by giving conditions for calculation. For example, there are two fields in a report Account_type and Amount. There can be two account types say ‘A’ and ‘B’. If you want to sum ‘A’ and ‘B’ separately, write a JavaScript in the Script Editor as:

Object: Report Event: OnDataInitialize,

Code:

```
function OnDataInitialize()
{
   rpt.Fields.add("valueA");
   rpt.Fields.add("valueB");
   rpt.Fields.("valueA").value = 0;
   rpt.Fields.("valueB").value = 0;
}

function OnFetchData(eof)
{
   if ( rpt.Fields("accType").value == "A" )
   {
       rpt.Fields.("valueA").value =       parseFloat(rpt.Fields.("valueA").value))
+ rpt.Fields("Amt").value;      }
   else
   {
       rpt.Fields.("valueB").value = parseFloat(rpt.Fields.("valueB").value)) +
rpt.Fields("Amt").value;
   }

   rpt.Sections("gfDEPT").Controls("txtACC_A").dataValue =
rpt.Fields.("valueA").value;
   rpt.Sections("gfDEPT").Controls("txtACC_B").dataValue =
rpt.Fields.("valueB").value;
}
```

This script will add two new fields in the report containing summated values for ‘A’ and ‘B’ account types.