Unlock the Power
Within Your Data

IDEAScript Best Practices Guide

IDEAScript Best Practices Guide

A CaseWare IDEA Support Document

CaseWare IDEA Inc. is a privately held software development and marketing company, with offices in Toronto and Ottawa, Canada, related companies in The Netherlands and China, and CaseWare IDEA Partners serving over 90 countries. CaseWare IDEA Inc. is a subsidiary of CaseWare International Inc., the world leader in business-intelligence software for auditors, accountants, and systems and financial professionals.

IDEA is distributed under an exclusive license by:

CaseWare IDEA Inc.
1400 St. Laurent Blvd., Suite 500
Ottawa, ON K1K 4H4
Canada
1-800-265-4332
idea.caseware.com

Publish Date: May 22, 2019

Contents

Overview

When programming it is best to use a standard that defines the coding style and conventions, so that when the code is shared amongst multiple people it looks the same and the flow is similar. This is not just true for groups of programmers but the individual programmer as well. This document is meant to be a guide for creating coding standards and as an outline of best practices for IDEAScript.

The guide will be divided into two sections. The first section is related to programming style in general, such as naming conventions, code structure, comments and other items. The second section is related to best practices in IDEAScript so that your macros will be less prone to hard to find errors.

Section 1: Programming Style

Naming Conventions

When you are programming your IDEAScript macros it is best to use the same method for naming variables, subroutines, databases, etc. This will make it easier to remember what a name means when you come back to a macro or for others who are looking at your macro understand what the name means. This guide uses three styles for naming variables. Some suggested options are:

- Camel Casing – this means that the first word in a variable name is lower case and each additional word uses initial capitalization (e.g., fileName or numRecords).
- Pascal Casing – this is similar to Camel Casing however, all words use initial capitalization (e.g., DirectExtraction or KeyValueExtraction).
- Block Casing – this style refers to all variables being in upper case and words are separated by an underscore (e.g., FILE_PATH or OUTPUT_LOCATION).

Variables and Constants

The first rule when naming a variable or constant is that the name is meaningful. For example, if you create a variable that holds a file name "fileName" is much better than "fn". The former makes the purpose of the variable much clearer to anyone who may need to examine or edit the script. Variable names should also typically be nouns.

When you are creating a variables or constants in your macros these are suggested best practices.

- Local variables should use Camel Casing
- Global variables should use Pascal Casing
- Constants should use Block Casing

The only exception to these rules should be loop variables. Loop variables should be a single character and use lower case. In general most programmers have the variable for the outer variable start at i and each nested loop use the next letter as in the following example.

```
For i = 1 To 100 - 1
For j = i + 1 To 100
Next j
Next i
```

Functions and Subroutines

Similar to naming variables, it is best to use a Function or Subroutine name which clearly describes its desired function. For example, a function named "DirectExtraction" is much better than "DE". It describes the purpose of the function and it is easy to tell by the name what purpose it serves. Where variables and constants are usually nouns functions and subroutines are either verbs or short sentences. In larger macros it is also best to include what the result of the function is or what input it requires. For example, ExtractHighValueCredits is more explanatory than DirectExtraction as to the panned behavior of the Function or Subroutine. Functions and subroutines should use Pascal Casing.

Dialogs

Dialogs should use similar naming conventions to the other items already discussed. They should be named so that their purpose is clear to the reader. It is common to append the word Dialog before or after its purpose such as CreditLimitDialog or MainDialog. The recommendation is to use Pascal Casing for dialogs.

Dialog Functions

Dialog functions are special functions that are called when a dialog is created, a control gains focus, or the user clicks a button. The parameters for this function use default values. The function name should use Pascal Casing and it should reflect its purpose. For example, CreditLimitDialogHandler or MainDialogHandler are good examples following the best practice for naming dialogs.

Controls

Dialogs are made up of a number of controls. Typically when naming a control it is prefaced with a two or three letter code that defines the control type. A suggested list of codes follows. Controls should be named used the Camel Casing methodology.

Control Type	Prefix	Sample Name
OK Button	btn	btnOK
Cancel Button	btn	btnCancel
Button	btn	btnEvaluateEquation
Check Box	chk	chkIncludePercentage
Static Text	lbl	lblFirstName
Edit Box	txt	txtFirstName

Control Type	Prefix	Sample Name
Group Box	grp	grpDatabaseOptions
List Box	lst	lstFieldNames
Combo Box	cbo	cboDatabaseNames
Drop Down Combo Box	ddl	ddlOptions
Picture	pic	picVendorImage

Parameters

Parameters are similar to variables. They should clearly define what input or output your function or subroutine requires. For example, creditLimit is much clearer than cl. Similar to variables and subroutines parameters are typically nouns that describe their purpose in the function or subroutine. As well, parameters should use Camel Casing.

On Error GoTo

When you use On Error GoTo to add error checking to your macros you must specify a label that the macro will jump to if an error occurs. The label should always have a meaningful name. It is also best practice to use Pascal Casing for labels. It is also good practice to add a word to the label to describe its purpose. Error, Catch or Handle all are good for this. For example, CheckActiveDatabaseError and CatchActiveDatabaseError are a good label for a function called CheckActiveDatabase.

Comments

Comments should be included in any macro that you create. It is considered best practice to have a comment block at the start of the macro, called the header, a comment at the start of each function or subroutine and inline comments that describe the logic of the code where it may not be clear what the macro is doing at that point. Comments in your code make it much easier for you or others to debug the macro later.

Header

At the start of each macro there should be a header which holds the name of the macro and describes the purpose of the macro It should include the author, date the macro was created, contact details if you will be sharing the macro, a list of inputs and/or outputs, and a change log. Included below are two samples for header comments with slightly different styles.

```
' ____________________________________________________________________
' IDEAScript:   Disk Check
' Author:       IDEA Support
' Date:         2015/05/16
' Purpose:      Browse a directory for all sub directories and/or files.
' Contact Info: ideasupport@caseware.com
'
' This macro takes a starting directory and file mask then
' browses the directory structure and creates a database with all
' sub directories and optionally files in those sub directories.
'
' (c) CaseWare IDEA Inc., All Rights Reserved.
' Disclaimer:  This macro is provided as is without any warranties.
' ____________________________________________________________________


'*************************************************************************

'***IDEAScript: Cross Join
'***Author:     Peter Smith
'***Date:       December 5, 2011
'***Purpose:    This script was developed based on a request from the IDEA
'               forum. The script will take two files and join all records
'               together.
'               So if you have one file with 2 records and another with
'               5 records your final file will have 10 records (2 x 5) and
'               all information will be joined between each file
'***Modified:   April 1, 2012 - Redid the menu so that multiple files can
'               be selected.
'***Modified:   April 4, 2012 - Added functionality to display sub
'               directories in the menu

'*************************************************************************
```

Functions

As with the macro itself, each function or subroutine should have a comment at the start of it. Included in these comments are the name, a list of parameters, the return value, and a description of the function. If the function or subroutine requires an input database or creates an output database it is best to include these details in the comments as well. See the following example of a comments section for a function.

```
'************************************************************

'***Function Name: Cat_Num_Char
'***Parameter:     (number As Double, fieldLength As Double)
'***Input:number   (i.e., 123), fieldLength for leading zeros (i.e., 10)
'***Output:        String of numbers (i.e., "0000000123")

'************************************************************
```

Inline Comments

Inline comments are used to make it clear what the purpose of items in the macro are, to the reader. They should be used to help the reader understand complex processes. They can also be used to give a more detailed description of a variable. There are also two ways that this type of commenting can be done.

They can either be appended to the end of a line of code or they can be included above a section of code. It is preferred that for variables they are included after the declaration and for blocks of code just above. Here are samples of both cases. Notice that for variable the comments are placed a few tabs from the variable and are lined up.

```
Dim strListSeparator As String     'The list separator from regional settings
Dim strDecimalSeparator As String  'The decimal separator from regional settings
Dim strThousandSeparator As String 'The thousand separator from regional settings

' Checks to see if initialization succeeded. If initialization failed display a
' message and end the macro.
If init = False Then

   ' If the resource strings were not read in correctly dispaly a message in English
   ' otherwise display the translated string based on language
   If GetResourceString("INIT_ERROR") = "" Then
      MsgBox "There was a problem initializing the macro."
   Else
      MsgBox GetResourceString("INIT_ERROR")
   End If
   Stop
End If
```

Layout

Your macros should all use the same layout consistently. This will help organize the code so it is easier to find items and improves readability. Both your macros and your functions and subroutines should use the same formats.

For macros the header is always at the very start of the macro and should always be followed by Option Explicit statement. Next will be any global variables, constants or types. Use of Global Variables will be discussed in more detail later on in this document. Following these declarations will be your Sub Main and then all of your other functions and subroutines.

Functions and subroutines will be similar to macros. There should be a comment block for the function, the function declaration, followed by any local variable declarations.

Indentation, Spacing and Tabs

The last item in programming style is indentation, spacing and tabs. Just as you would do when writing text, you should be consistent in the use of indentation, spacing and tabs throughout your macros. This leads to better readability and helps the reader follow the flow of your macro better. It is recommended that you use tabs instead of spaces when indenting items. This is more important when you are using a font that is not a monospace font such as Calibri, Times New Roman, etc. unlike Courier New or Consolas. This ensures that starting character positions in the editor line up.

Functions and Subroutines

To be consistent the starting word, Function or Sub, and the ending word, End, should be in the first column. Each line of code, other than code that is organized in blocks, should start one tab in. As mentioned earlier variable declarations should be first. Between variable declarations and the first line of code or comment there should be at least one blank line.

Code Blocks

Code blocks refer to either sets of statements that are surrounded by either a condition or loop structure or code for a similar task. You should always include at least one blank line after each block. The way indentation and spacing is handled between code should be the same throughout all of your macros.

Each code block should start at the same tab location. If the code block is a multi-line statement, like an If/Then statement or For/Next statement, then the code that belongs inside that block starts one tab in from the tab of first line of the block. Each nested block is handled the same way, moving one more tab further to the right with each nested block. The end of each block should line up with the beginning of that block.

The following function demonstrates this concept. The variable declarations start one tab in from the left margin. Between the variable declarations and the first line of code to be executed there is a blank line. Another blank line was included as there is a new code block, a For/Next loop. In the For/Next loop the code has been indented another tab to indicate that it is a "child" of the For/Next code block. Following that is a new statement followed by a blank

line as after that is a new code block, an If/Then statement. The inner code for the If/Then statement is again indented once more as it is child code. The position of the End If and Next I match the position of the starting code. Finally there is a blank line between the For/Next loop and setting the return value for the function. This sort of structure allows the reader to better follow the flow and read through the code.

```
Function MyFunctyon(ByVal loopLimit As Integer) As Integer
   Dim i As Integer
   Dim sum As Integer

   sum = 0

   For i = 1 To loopLimit
      sum = sum + 1

      If i = 100 Then
         MyFunction = sum
         Exit sum
      End If
   Next i

   MyFunction = sum
End Function
```

Here now as an example where best practice was not use, making it much harder to follow the flow of the code and the programmers thought process.

```
Function RegisterResults() As Boolean
   Dim Filenum As Integer
   Dim i As Integer
   Dim sLine As String

Filenum = FreeFile
   If fso.FileExists(Client.WorkingDirectory & "ResultFile.dat") Then
   Open Client.WorkingDirectory & "ResultFile.dat" For Input As Filenum
      Line Input #Filenum, sourceFile
      Line Input #Filenum, amount
      Line Input #Filenum, jeNo
      Line Input #Filenum, rowID
         While Not EOF(Filenum)
         i = i + 1
         Line Input #Filenum, sLine
         resultFiles(i) = Trim(sLine)
         Wend
      Close Filenum
      RegisterResults = True
   End If
End Function
```

Without the extra spaces and consistent indentation, it is harder to follow the flow of this relatively simple function.

Select Case

This is slightly different code block. The first and last lines should line up with code at the same nesting level. For example, if it was at the start of a function it should line up with the variable declarations. Each Case section included should be one tab in from the start of the Select Case statement. The code executed for that case should be another tab in from that. If a code block is included inside of a case it should follow the same rules as above. See the following example.

```
Function HandleOptionsDialog(ControlID$, Action%, SuppValue%)
   Select Case Action%
      Case 1
         DlgEnable "txtSearch", 0
         DlgEnable "txtKeyword", 0
      Case 2
         If ControlID$ = "btnSetSearch" Then
            DlgText "txtSearch", "txtInput"
         End If
   End Select
End Function
```

On Error

When On Error is used it should be separated by blank lines. The label for the GoTo statement should be at the left margin of the editor. This helps better understand the flow of the function. See the following example.

```
Function CheckActiveDatabase() As Boolean
   Dim db As Object

   On Error GoTo CheckActiveDatabaseError

   Set db = Client.CurrentDatabase
   Set db = Nothing

   CheckActiveDatabase = True
   Exit Sub

CheckActiveDatabaseError:
   CheckActiveDatabase = False
End Function
```

Section 2: Programming Guidelines

Following the guidelines below will help you write stronger macros, avoid common pitfalls, and prevent hard to find bugs.

Variables

Option Explicit

The first line of code for all macros should always be Option Explicit. This statement forces the programmer to define all variables, functions, subroutines, and so forth before using them. There are a number of reasons why this is important. What is most important is that this gives you control over a variable's scope and life time. This prevents confusing variables between functions and subroutines as well as making it easier to track changes of values in debugging.

Local Variables vs. Global Variables

It is often considered bad practice to use global variables. The use of global variables lead to messy design, cause hard to find bugs during testing and can make future changes and maintenance to a macro much more difficult. This is especially true when using Object type variables, more on those later. While this is considered to be the best practice there may be times where it is more practical to use global variables.

Defining Variables

Variables are created using the Dim statement. While it is valid syntax to dimension multiple variables on the same line this should never be done. Always dimension each variable on a line of its own. The reason for this is particular to IDEAScript, not a programming guideline in general. Defining multiple variables on the same line has the potential to lead to errors in IDEAScript which may be very hard to identify.

Variable Types

When you are dimensioning variables you should assign them a type rather than allowing IDEAScript to determine the type. Having strongly typed variables helps in the debugging process as most times the compiler will catch that you are using a variable of the wrong type. However, there may be cases where it will not detect types correctly and the error message is misleading, so some care is required.

The other note is to be careful with numeric data types due to maximum and minimum values associated with these types. If you choose a numeric type and a value assigned is outside of the limit of allowed values an error will occur. Keep these following guidelines in mind when selecting numeric data types.

For loops and counter is it is best to use and data type without decimals, like Integer or Long, for both speed and space reasons. The data type require the lowest amount of memory and process quicker than other numeric data types.

Double offers the widest range of numeric values, from very small to very large. It should be used when you will be unsure what numeric limits your macro will cover, you are dealing with decimal values, or you want to prepare for the possibility that values will grow over time.

Integer has possibility the lowest range of values that can be assigned to it. This type is usually associated with return values or cases. Use with care as it is easy to cause an overflow error with this data type.

Object Variables

Object is a special variable type. Object variables are assigned values using the Set keyword. These types of variables are assigned memory that must be freed when you are done using them. This done by setting the variable to 'Nothing'. There are several things to remember when using Object variables. Below are a set of guidelines for working with Object type variables. Not following these can lead to very hard to resolve and inconsistent bugs.

1. You must assign all variables of Object type to 'Nothing' when you are finished.
2. If you plan to assign a new value to an object variable, you should set it to Nothing first so any reserved resources are released
3. When setting variables to Nothing you should always use the last created first released methodology

Here is a sample function that demonstrates all three principals.

```
Function FindHighValueAccounts

   ' Define variables
   Dim database As Object
   Dim task As Object
   Dim dbName As String

   ' Set database object and create summarization task
   Set database = Client.OpenDatabase("IDEA Journal Entries.IMD")
   Set task = database.Summarization

   ' Set output database name
   dbName = "Account Summarization.IMD"

   ' Define task settings and run the task
   task.AddFieldToSummarize "ACCT_NUM"
   task.AddFieldToTotal "AMOUNT"
   task.OutputDBName = dbName
   task.CreatePercentField = FALSE
   task.StatisticsToInclude = SM_SUM
   task.PerformTask

   ' The variable task was created with database so it must be set to Nothing
   ' before the database variable. They also need to be set to Nothing before
   ' running the next task.
   Set task = Nothing
   Set database = Nothing
```

```
    ' Set database object and create extaction task
    Set database = Client.OpenDatabase("Account Summarization.IMD")
    Set task = database.Extraction

    ' Set output database name
    dbName = "High Value Accounts.IMD"

    ' Define task settings and run the task task.IncludeAllFields
    task.AddExtraction dbName, "", "@Abs(AMOUNT_SUM) >= 5000"
    task.CreateVirtualDatabase = False
    task.PerformTask 1, database.Count

    Set task = Nothing
    Set database = Nothing
End Function
```

Parameters

ByVal vs ByRef

When defining parameters you have the option of using ByVal, by value, or ByRef, by reference, for each parameter. Unlike other languages where ByVal is the default, IDEAScript uses ByRef by default. Programmers who have experience in other languages should keep this in mind.

It is best to use ByVal in most instances. This will prevent unintentional changes to the variable being passed as an argument to the function or subroutine. ByRef is best left if you need to pass objects or arrays between functions.

Using Object Variables as Parameters

IDEAScript allows you to programmatically interact with IDEA using different programming objects. The most common is the database object. If you will be using the same object in multiple functions it is considered best practice to create the object once and then pass it as a parameter to the other functions that require access to that object.

Code Reuse

Often you will find that you are using the same function in multiple scripts. It is considered a best practice to have a library or repository of these functions, where you can extract them and then paste them into a new macro as required. A good example is function that checks if there is a field with a specific name in a database.

On Error GoTo

Any number of unexpected things can happen when you run your macro. For example, a file could be missing, a field type could have changed, or an ODBC connection could have been lost. You should always design your macros to handle these errors. In order to handle errors, use the On Error GoTo construct. It is best practice to add error handling to each function and subroutine. Do not rely on a global error handler to catch all errors as in some instances this will not work as expected. You should also reset the error handler at the end of each function.

```
Function FindHighValueAccounts

   ' Define variables
   Dim database As Object
   Dim task As Object
   Dim dbName As String

   On Error GoTo CatchFindHighValueAccounts

   ' Set database object and create extaction task
   Set database = Client.OpenDatabase("Account Summarization.IMD")
   Set task = database.Extraction

   ' Set output database name
   dbName = "High Value Accounts.IMD"

   ' Define task settings and run the task
   task.IncludeAllFields
   task.AddExtraction dbName, "", "@Abs(AMOUNT_SUM) >= 5000"
   task.CreateVirtualDatabase = False
   task.PerformTask 1, database.Count

   Set task = Nothing
   Set database = Nothing

   On Error GoTo 0
   Exit Function

CatchFindHighValueAccounts:
   Set task = Nothing
   Set database = Nothing
   On Error GoTo 0
End Function
```

Working with Databases

Iterating Through a Database

Iterating through a database is much slower than having IDEA run a task on that database. Where possible use IDEA functions instead of iterating through a database. If you must iterate through a database the preferred way is as follows.

```
Function IterateDatabaseSample
   Dim database As Object
   Dim currRecordSet As Object
   Dim currRecord As Object
   Dim count As Long
   Dim i As Long
   Dim amount As Double
   Dim amountSum As Double

   Set database = Client.OpenDatabase("IDEA Journal Entries - OAG-GL Routine.IMD")
   Set currRecordSet = database.RecordSet ' Get the record set for the database
   Set currRecord = currRecordSet.ActiveRecord ' Set currRecord to be the active
   record

   ' Initialize variables
   count = currRecordSet.Count
   amountSum = 0

   ' Loop through all of the records sum the database on the amount field
   For i = 1 To count
      currRecordSet.GetAt(i)

      amount = currRecord.GetNumValueAt(7)
      amountSum = amountSum + amount
   Next i

   ' Display the result
   MsgBox amountSum

   ' Close the database
   database.Close

   ' Release objects
   Set record = Nothing
   Set recordSet = Nothing
   Set database = Nothing
End Function
```

You will notice that the currRecord variable is set just once outside of the loop. Doing this saves processing time as the Object variable is not constantly being set. Moving that line inside of the For/Next loop would dramatically increase the processing time for the function.

Field Manipulation

When working with IDEA fields, the most important thing to remember is that working with virtual fields is much faster than working with native fields. Wherever possible use virtual fields over native fields. If you are editing the equation of a virtual field, keep in mind that any existing indexes must be updated, in order to allow the change to be pushed through to the database. Whenever possible do not delete fields in your macro, as this is time consuming.

The main reason you should choose to use a native field instead of a virtual field is when that field will be used by other virtual fields. Each time a virtual field needs to use another virtual field, processing time increases. Another reason to use native fields is the complexity of the equation for the virtual field, which ties in with the previous point. Where the definition of the virtual field is complex, processing time when referencing that field will increase. For example, if you are using Regular Expressions then it is definitely better to use a native field, as the Regular Expression can be computationally expensive.

Opening and Closing Databases

Remember that it takes time and resources to open a database when calling Client.OpenDatabase. When processing the same multiple database in multiple functions it is best to pass that database as a parameter instead.

It is always important to close databases which are no longer needed. Leaving databases open can cause memory issues and issues caused by locked databases. It is better to call Close on the database object rather than to use the CloseDatabase function of the Client object.

The following function illustrates the best practice for opening and closing databases. The function opens the database, runs a few tests on the database and then closes it.

```
Function CloseDatabaseSample
   Dim database As Object

   Set database = Client.OpenDatabase("IDEA Journal Entries - OAG-GL Routine.IMD")

   Call ExtractHighValues(database)
   Call SearchMissingInformation(database)

   database.Close

   Set database = Nothing
End Function
```

Checking If a Field Exists

When running macros multiple times there will be instances where you will want to know if a field exists in the database. A field may have been appended as the result of a process, optionally be imported, and so on. The preferred way to do this is to use error trapping rather than iterating through all of the fields in a database and check the names of each field. The following function demonstrates the preferred way.

```
Function IsFieldInTheDatabase(ByRef db As Object, ByVal fieldName As String) As
Boolean
   Set table = db.TableDef
   On Error GoTo NotFound

   ' Does the field exist?
   Set field = table.GetField (fieldName)
   IsFieldInTheDatabase = TRUE
   Set table = Nothing
   Set field = Nothing

   ' Reset On Error
   On Error GoTo 0
   Exit Function
NotFound:

   ' Field wasn't found, return false
   IsFieldInTheDatabase = FALSE
   Set field = Nothing
   Set table = Nothing
   On Error GoTo 0
End Function
```

Limiting Data

While today's computers can work with 100% of a data set rather than samples or subsets it is always best to try and work with only the required data. To that end, try and limit the number of fields and records being analyzed. Limiting the size of the data set helps reduce computation and disk activity, with disk activity being the main bottleneck in processing large data sets. It is recommended to pre-process the database and extract records that are not exceptions to better process the exceptions. Another option is during import only bring in records in the period to be analyzed. Similarly, fields can be excluded from output databases. It is better the write the code to include specific fields instead of all fields in a database.

Testing

It goes without saying that all macros need be thoroughly tested. However, what exactly should be tested? In general testing will revolve around creating test cases and ensure that the proper output is returned and that the macro does not crash. It is also important that a macro is tested for speed as what might work fine on one computer can take much longer on a different computer.

Input

Macros should be tested against as wide a range of input as possible. Test cases should include valid and invalid input. These tests verify that macro will return the same results regardless of what input the user gives, even if the input is outside the scope of the macro. They also confirm that the macro does not crash when something unexpected is received. In addition to different values, different file sizes should also be tested. For example, a macro that runs well against a thousand records may behave much differently when run against a source file with millions of records.

Output

The output from the macro should be tested with different data sets which do and do not produce exceptions. It is just as important to indicate that there were no exceptions as there actually were exceptions. For example, in a perfect world all general ledger accounts should balance. When an account does not balance that is an exception and that exception needs to be recorded. There is also the possibility that all of the accounts do balance. That result also needs to be returned to the user. So, the macro should be tested for both scenarios and must receive correct and complete results. The macro must also handle the scenario where no exceptions were found.

Speed

The end goal should be that a macro returns the correct results and returns them as quickly as possible. This document has covered a few areas to assist in this area. Below is a high level list on ways to improve performance.

- Test exceptions rather than looping through all data. For example, when trying to determine if a field is in a database it is quicker to try and retrieve a field with that name rather than loop over all of the fields and testing until you find a match for the name or no match
- Pass objects as parameters rather than recreating them. if you will be running a few tasks against the same database, then open the database once and pass it as a parameter, as opposed to opening it repeatedly
- Keep non-essential code outside of loops. The preferred method of iterating over a database the code to set the currRecord variable was only done the once rather than multiple times
- Once the macro works as expected you may want to test field manipulation to see if you get better performance using virtual or native fields e.g. for appending fields with simple equations that do not rely on other virtual fields then a virtual field is the best way to approach, however, if the calculations are very complex and rely on other fields then native would be the best approach

CaseWare IDEA Inc.

1400 St. Laurent Blvd., Suite 500
Ottawa, ON K1K 4H4
Canada
1-800-265-4332

idea.caseware.com