Abnormal programming with InterSystems

Dmitriy Maslennikov 5 November 2018

Caché, Studio, InterSystems IRIS

I bet that not everyone familiar with InterSystems Caché knows about Studio extensions for working with the source code. You can actually use the Studio to create your own type of source code, compile it into interpretable (INT) and object code, and sometimes even add code completion support. That is, theoretically, you can make the Studio support any programming language that will be executed by the DBMS just as well as Caché ObjectScript. In this article, I will give you a simple example of writing programs in Caché Studio using a language that resembles JavaScript. If you are interested, please read along.

If you go to the SAMPLES namespace, you will find an example of working with user-defined file types. The example suggests opening a document of the “Example User Document (.tst)” type, and there is only one file of this type called TestRoutine.TST, which, in fact, is generated on the go. The class required for working with this file type is called *Studio.ExampleDocument*. Let’s not get into this example too deeply and create our own instead. The ".JS" file type is already being used in the Studio and JavaScript that we want to support is not exactly the original JavaScript. Let’s call it CacheJavaScript and the file type will be ".CJS". To start off, create a *%CJS.StudioRoutines* class as a subclass of the *%Studio.AbstractDocument* class and add the support of the new file type to it.

```
/// The extension name, this can be a comma separated list of extensions if this clas
s supports more than one
Projection RegisterExtension As %Projection.StudioDocument(DocumentDescription = "Cac
héJavaScript Routine", DocumentExtension = "cjs", DocumentIcon = 1, DocumentType = "J
S");
```

- DocumentDescription — displayed as the type description in the open file window in the list of filters;
- DocumentExtension — the extension of the files that will be processed by this class;
- DocumentIcon — the icon number starts from zero; the following icons are available:
  
- DocumentType — this type will be used for code and error highlighting; the following types are available:
  - INT — Cache Object Script INT code
  - MAC — Cache Object Script MAC code
  - INC — Cache Object Script macro include
  - CSP — Cache Server Page
  - CSR — Cache Server Rule
  - JS — JavaScript code
  - CSS — HTML Style Sheet
  - XML — XML document
  - XSL — XML transform
  - XSD — XML schema
  - MVB — Multivalue Basic mvb code
  - MVI — Multivalue Basic mvi code

We will now implement all the necessary methods for supporting the new source code type in the Studio.

`ListExecute` and **`ListFetch`** methods are used for obtaining a list of files available in the namespace and for showing them in the open file dialogue.

```
ClassMethod ListExecute(ByRef qHandle As %Binary, Directory As %String, Flat As %Bool
```

```
ean, System As %Boolean) As %Status
{
    Set qHandle=$listbuild(Directory,Flat,System,"")
    Quit $$$OK
}

ClassMethod ListFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd As %I
nteger = 0) As %Status [ PlaceAfter = ListExecute ]
{
    Set Row="",AtEnd=0
    If qHandle="" Set AtEnd=1 Quit $$$OK
    If $list(qHandle)'=""||($list(qHandle,4)=1) Set AtEnd=1 Quit $$$OK
    set AtEnd=1
    Set rtnName=$listget(qHandle,5)
    For {
        Set rtnName=$order(^rCJS(rtnName))    Quit:rtnName=""
        continue:$get(^rCJS(rtnName,«LANG»))'=«CJS»
        set timeStamp=$zdatetime($get(^rCJS(rtnName,0)),3)
        set size=+$get(^rCJS(rtnName,0,«SIZE»))
        Set Row=$listbuild(rtnName_".cjs",timeStamp,size,"")
        set AtEnd=0
        set $list(qHandle,5)=rtnName
        Quit
    }
    Quit $$$OK
}
```

We will store the description of the programs in the **^rCJS** global, and the **ListFetch** method will traverse this global to return strings containing the following: name, date, and size of the found file. In order for the results of being displayed in the dialogue, you need to create an **Exists** method that checks whether a file with such a name exists.

```
/// Return 1 if the routine 'name' exists and 0 if it does not.
ClassMethod Exists(name As %String) As %Boolean
{
    Set rtnName = $piece(name,".",1,$length(name,".")-1)
    Set rtnNameExt = $piece(name,".",$length(name,"."))
    Quit $data(^rCJS(rtnName))&&($get(^rCJS(rtnName,«LANG»))=$zconvert(rtnNameExt,«U»
))
}
```

The **TimeStamp** will return the date and time of the program. The result is also shown in the file open dialogue.

```
/// Return the timestamp of routine 'name' in %TimeStamp format. This is used to dete
rmine if the routine has
/// been updated on the server and so needs reloading from Studio. So the format shou
ld be $zdatetime($horolog,3),
/// or "" if the routine does not exist.
ClassMethod TimeStamp(name As %String) As %TimeStamp
{
    Set rtnName = $piece(name,".",1,$length(name,".")-1)
    Set timeStamp=$zdatetime($get(^rCJS(rtnName,0)),3)
    Quit timeStamp
}
```

We will now need to load the program and save the changes in the file. The text of the program, line by line, is stored in the same **^rCJS** global.

```
/// Load the routine in Name into the stream Code
Method Load() As %Status
{
    set source=..Code
    do source.Clear()
    set pCodeGN=$name(^rCJS(..ShortName,0))
    for pLine=1:1:$get(@pCodeGN@(0),0) {
        do source.WriteLine(@pCodeGN@(pLine))
    }
    do source.Rewind()
    Quit $$$OK
}

/// Save the routine stored in Code
Method Save() As %Status
{
    set pCodeGN=$name(^rCJS(..ShortName,0))
    kill @pCodeGN
    set @pCodeGN=$ztimestamp
    Set ..Code.LineTerminator=$char(13,10)
    set source=..Code
    do source.Rewind()
    WHILE '(source.AtEnd) {
        set pCodeLine=source.ReadLine()
        set @pCodeGN@($increment(@pCodeGN@(0)))=pCodeLine
    }
    set @pCodeGN@(«SIZE»)=..Code.Size
    Quit $$$OK
}
```

Here comes the most interesting part: compilation of our program. We will compile into INT code and therefore have full compatibility with Caché. This article is just an example, which is why I used just a small fraction of the capabilities of CachéJavaScript: declaration of variables (`var`), reading (`read`), and data output (`println`).

```
/// CompileDocument is called when the document is to be compiled
/// It has already called the source control hooks at this point
Method CompileDocument(ByRef qstruct As %String) As %Status
{
    Write !,«Compile: „,..Name
    Set compiledCode=##class(%Routine).%OpenId(..ShortName_“.INT»)
    Set compiledCode.Generated=1
    do compiledCode.Clear()

    do compiledCode.WriteLine(" ;generated at "_$zdatetime($ztimestamp,3))
    do ..GenerateIntCode(compiledCode)

    do compiledCode.%Save()
    do compiledCode.Compile()
    Quit $$$OK
}

Method GenerateIntCode(aCode) [ Internal ]
{
    set varMatcher=##class(%Regex.Matcher).%New("[ \t]*(var[ \t]+)?(\w[\w\d]*)[ \t]*(
```

```
\=[ \t]*(.*))?")
    set printlnMatcher=##class(%Regex.Matcher).%New("[ \t]*(?:console\.log|println)\(
([^\)]+)\)?")
    set readMatcher=##class(%Regex.Matcher).%New("[ \t]*read\((.*)\,(.*)\)")

    set source=..Code
    do source.Rewind()
    while 'source.AtEnd {
        set tLine=source.ReadLine()

        set pos=1
        while $locate(tLine,"(([^\'\""\;\r\n]|[\'\""][^\'\""]*[\'\""])+)",pos,pos,tCo
de) {
            set tPos=1
            if $zstrip(tCode,"*W")="" {
                do aCode.WriteLine(tCode)
                continue
            }
            if varMatcher.Match(tCode) {
                set varName=varMatcher.Group(2)
                if varMatcher.Group(1)'="" {
                    do aCode.WriteLine($char(9)_«new „_varName)
                }
                if varMatcher.Group(3)'=“» {
                    set expr=varMatcher.Group(4)
                    set expr=..Expression(expr)
                    do:expr'="" aCode.WriteLine($char(9)_«set „_varName_“ = „_expr)
                }
                continue

            } elseif printlnMatcher.Match(tCode) {
                set expr=printlnMatcher.Group(1)
                set expr=..Expression(expr)
                do:expr'=“» aCode.WriteLine($char(9)_«Write „_expr_“,!»)

            } elseif readMatcher.Match(tCode) {
                set expr=readMatcher.Group(1)
                set expr=..Expression(expr)
                set var=readMatcher.Group(2)
                do:expr'="" aCode.WriteLine($char(9)_«read „_expr_“,»_var_",!")
            }
        }
    }
}

ClassMethod Expression(tExpr) As %String
{
    set matchers($increment(matchers),«matcher»)="(?sm)([^\'\""]*)\+[ \t]*(?:\""([^\"
"]*)\""|\'([^\']*)\')([^\'\""]*)"
    set matchers(matchers,«replacement»)="$1_""$2$3""$4"

    set matchers($increment(matchers),«matcher»)="(?sm)([^\'\""]*)(?:\""([^\""]*)\""|
\'([^\']*)\')[ \t]*\+([^\'\""]*)"
    set matchers(matchers,«replacement»)="$1""$2$3""_$4"

    set matchers($increment(matchers),«matcher»)="(?sm)([^\'\""]*)(?:\""([^\""]*)\""|
\'([^\']*)\')([^\'\""]*)"
    set matchers(matchers,«replacement»)="$1""$2$3""$4"
```

```
    set tResult=tExpr
    for i=1:1:matchers {
        set matcher=##class(%Regex.Matcher).%New(matchers(i,«matcher»))
        set replacement=$get(matchers(i,«replacement»))

        set matcher.Text=tResult

        set tResult=matcher.ReplaceAll(replacement)
    }

    quit tResult
}
```

You can view the generated INT code for each compiled program or class. To do that, you will need to write a **`GetOther`** method. It's pretty simple — its purpose is to return a comma-delimited list of programs that were generated for the source code.

```
/// Return other document types that this is related to.
/// Passed a name and you return a comma separated list of the other documents it is
related to
/// or "" if it is not related to anything. Note that this can be passed a document o
f another type
/// for example if your 'test.XXX' document creates a 'test.INT' routine then it will
 also be called
/// with 'test.INT' so you can return 'test.XXX' to complete the cycle.
ClassMethod GetOther(Name As %String) As %String
{
    Set rtnName = $piece(Name,".",1,$length(Name,".")-1)_".INT"
    Quit:##class(%Routine).%ExistsId(rtnName) rtnName
    Quit ""
}
```

We implemented a method of blocking a program so that just one developer at a time could edit a program or class on the server.

Don't forget about writing a method for deleting programs.

```
/// Delete the routine 'name' which includes the routine extension
ClassMethod Delete(name As %String) As %Status
{
    Set rtnName = $piece(name,".",1,$length(name,".")-1)
    Kill ^rCJS(rtnName)
    Quit $$$OK
}

/// Lock the current routine, default method just unlocks the ^rCJS global with the n
ame of the routine.
/// If it fails then return a status code of the error, otherwise return $$$OK
Method Lock(flags As %String) As %Status
{
    Lock +^rCJS(..Name):0 Else Quit $$$ERROR($$$CanNotLockRoutine,..Name)
    Quit $$$OK
}

/// Unlock the current routine, default method just unlocks the ^rCJS global with the
 name of the routine
Method Unlock(flags As %String) As %Status
```

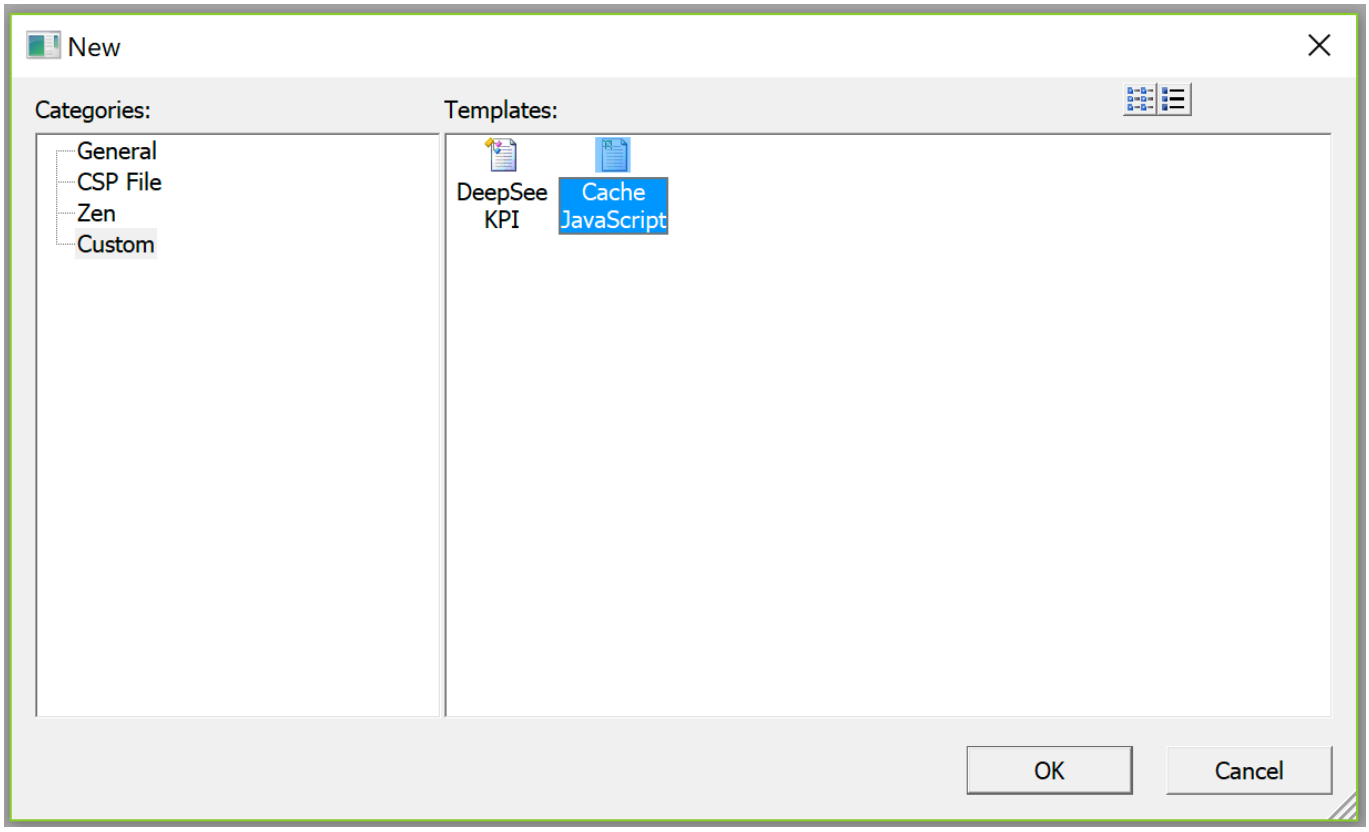
```
{
    Lock -^rCJS(..Name)
    Quit $$$OK
}
```

All right, we have written a class that allows us to work with our type of programs. However, we cannot write such a program just yet. Let's fix it. The Studio enables you to define templates and there are 3 ways of doing it: a simple CSP file of a particular format, a CSP class inherited from the **%CSP.StudioTemplateSuper** class, and, finally, a ZEN page inherited from **%ZEN.Template.studioTemplate**. In our case, we will use the last option for simplicity. Templates can be of 3 types as well: for creating new objects, just code templates, and add-ins, which generate no output.
In our case, we will need a template for creating new objects. Let's make a new class called **%CJS.RoutineWizard**. Its content is pretty simple – you will need to describe a field for entering the program's name, then describe the name of the new program and its mandatory content for the Studio in the **%OnTemplateAction** method.

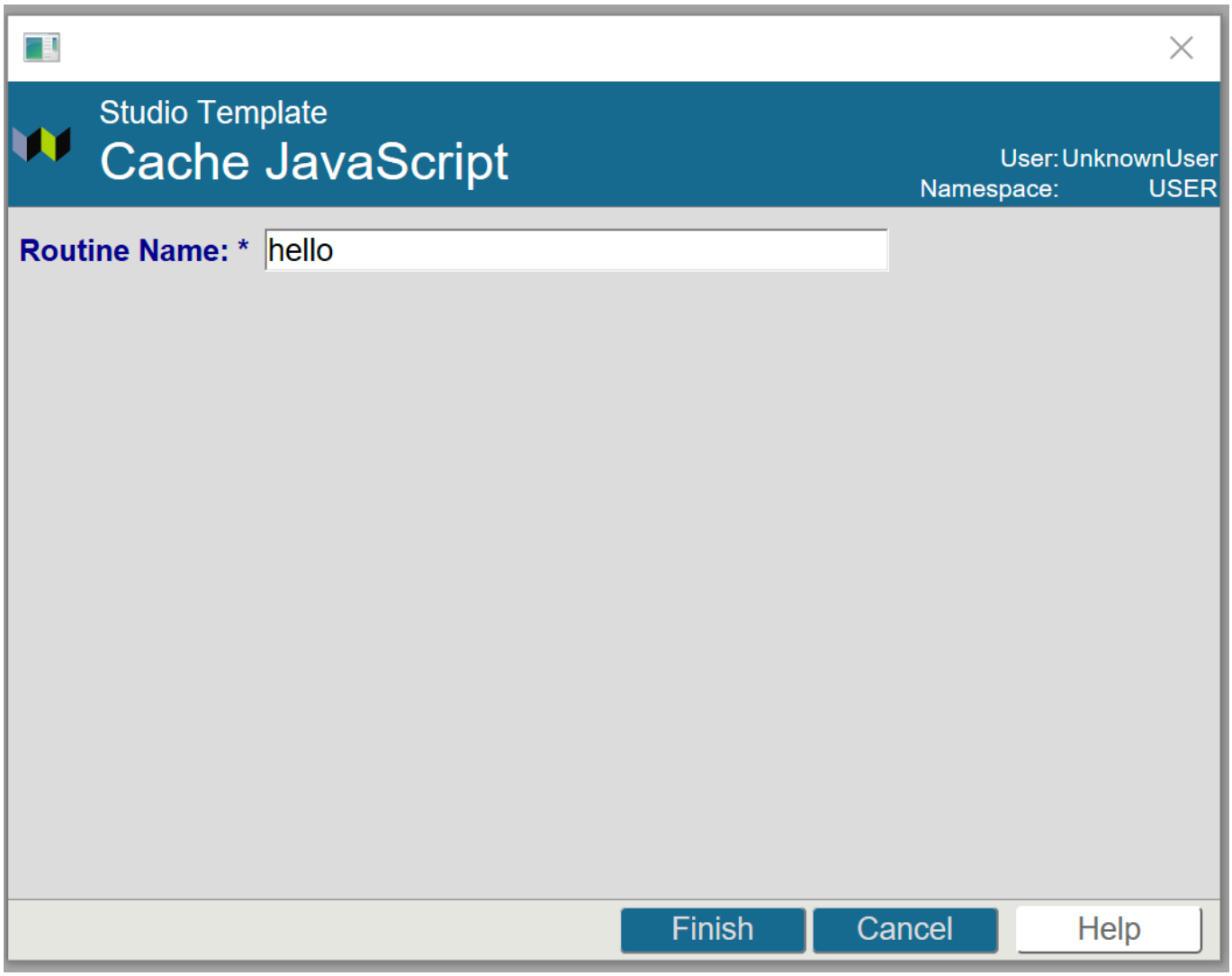
```
/// Studio Template:

/// Create a new Cache JavaScript Routine.
Class %CJS.RoutineWizard Extends %ZEN.Template.studioTemplate [ StorageStrategy = ""
]
{

Parameter TEMPLATENAME = "Cache JavaScript";

Parameter TEMPLATETITLE = "Cache JavaScript";

Parameter TEMPLATEDESCRIPTION = "Create a new Cache JavaScript routine.";

Parameter TEMPLATETYPE = "CJS";

/// What type of template.
Parameter TEMPLATEMODE = "new";

/// If this is a TEMPLATEMODE="new" then this is the name of the tab
/// in Studio this template is dispayed on. If none specified then
/// it displays on 'Custom' tab.
Parameter TEMPLATEGROUP As STRING;

/// This XML block defines the contents of the body pane of this Studio Template.
XData templateBody [ XMLNamespace = "http://www.intersystems.com/zen" ]
{

}

/// Provide contents of description component.
Method %GetDescHTML(pSeed As %String) As %Status
{
    Quit $$$OK
}

/// This is called when the template is first displayed;
/// This provides a chance to set focus etc.
ClientMethod onstartHandler() [ Language = javascript ]
{
    // give focus to name
    var ctrl = zenPage.getComponentById('ctrlRoutineName');
    if (ctrl) {
```

```
        ctrl.focus();
        ctrl.select();
    }
}

/// Validation handler for form built-into template.
ClientMethod formValidationHandler() [ Language = javascript ]
{
    var rtnName = zenPage.getComponentById('ctrlRoutineName').getValue();

    if ('' == rtnName) {
        return false;
    }

    return true;
}

/// This method is called when the template is complete. Any
/// output to the principal device is returned to the Studio.
Method %OnTemplateAction() As %Status
{
    Set tRoutineName = ..%GetValueByName("RoutineName")

    Set %session.Data("Template","NAME") = tRoutineName_".CJS"
    Write "// "_tRoutineName,!
    Quit $$$OK
}

}
```

That's it. You can now create your first program written in Caché JavaScript in the Studio.



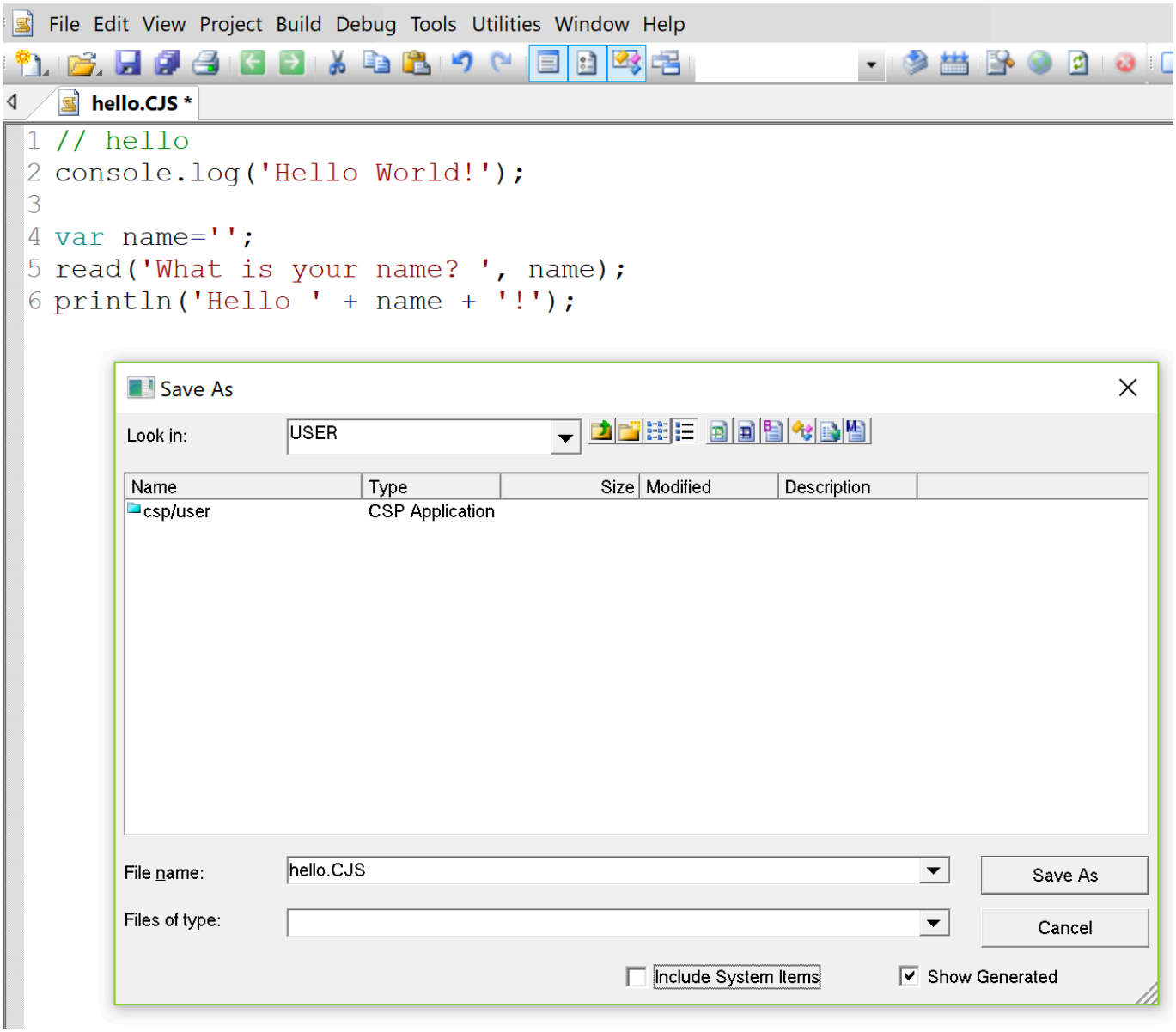
Let's call it "hello".



The source code in CachéJavaScript can look like this, for example:

```
// hello
console.log('Hello World!');

var name='';
read('What is your name? ', name);
println('Hello ' + name + '!');
```

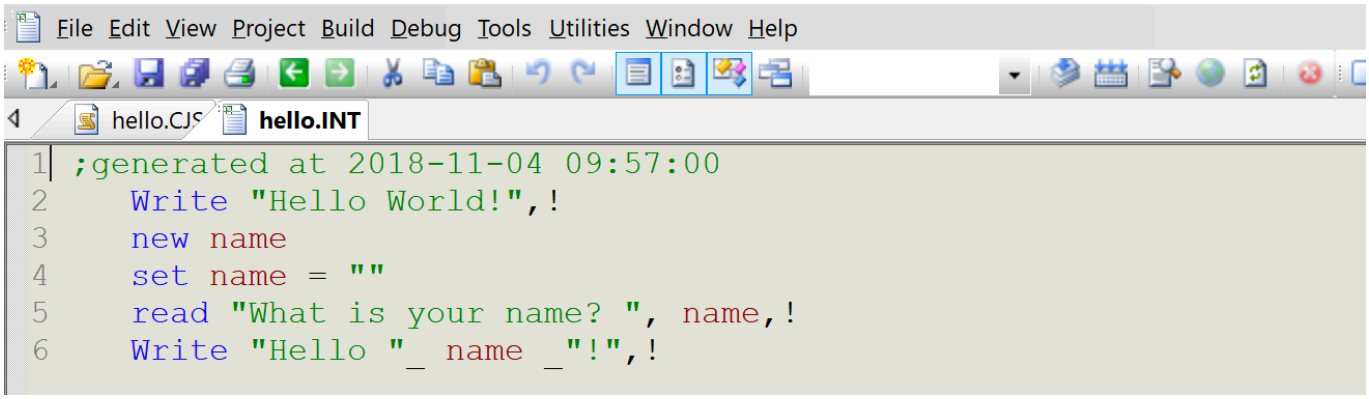
Let's save it.



After save and compile we will see that int code was generated compiled as well successfully, in the output:

```
Compilation started on 11/04/2018 12:57:00 with qualifiers 'ck-u'
Compile: hello.CJS
Compiling routine : hello.int
Compilation finished successfully in 0.034s.
```

Let's look at another source.



We can now run it in the terminal

```
USER>d ^hello
Hello World!
What is your name? daimor
Hello daimor!
```

This is how you can describe any language (to a certain extent, of course) that you like and use it to code the server-side business logic for the Caché/IRIS Data platform. There definitely will be problems with code highlighting if this language is not supported by the Studio. This example demonstrates the work with programs, but can definitely create Caché classes the same way. The possibilities are nearly limitless: you just need to write a lexical parser, a syntax parser, and a full-fledged compiler, then come up with the right mapping between all Caché system functions and specific constructs in the new language. Such programs can also be exported and imported with compilation, as it is done with any other programs in Caché.

Anyone willing to do it at home can download the source codes here in udl or xml.

- + 12
- 6
- 3
- 862
- 6

Dmitriy Maslennikov 5 November 2018

Reply

Source URL: https://community.intersystems.com/post/abnormal-programming-intersystems