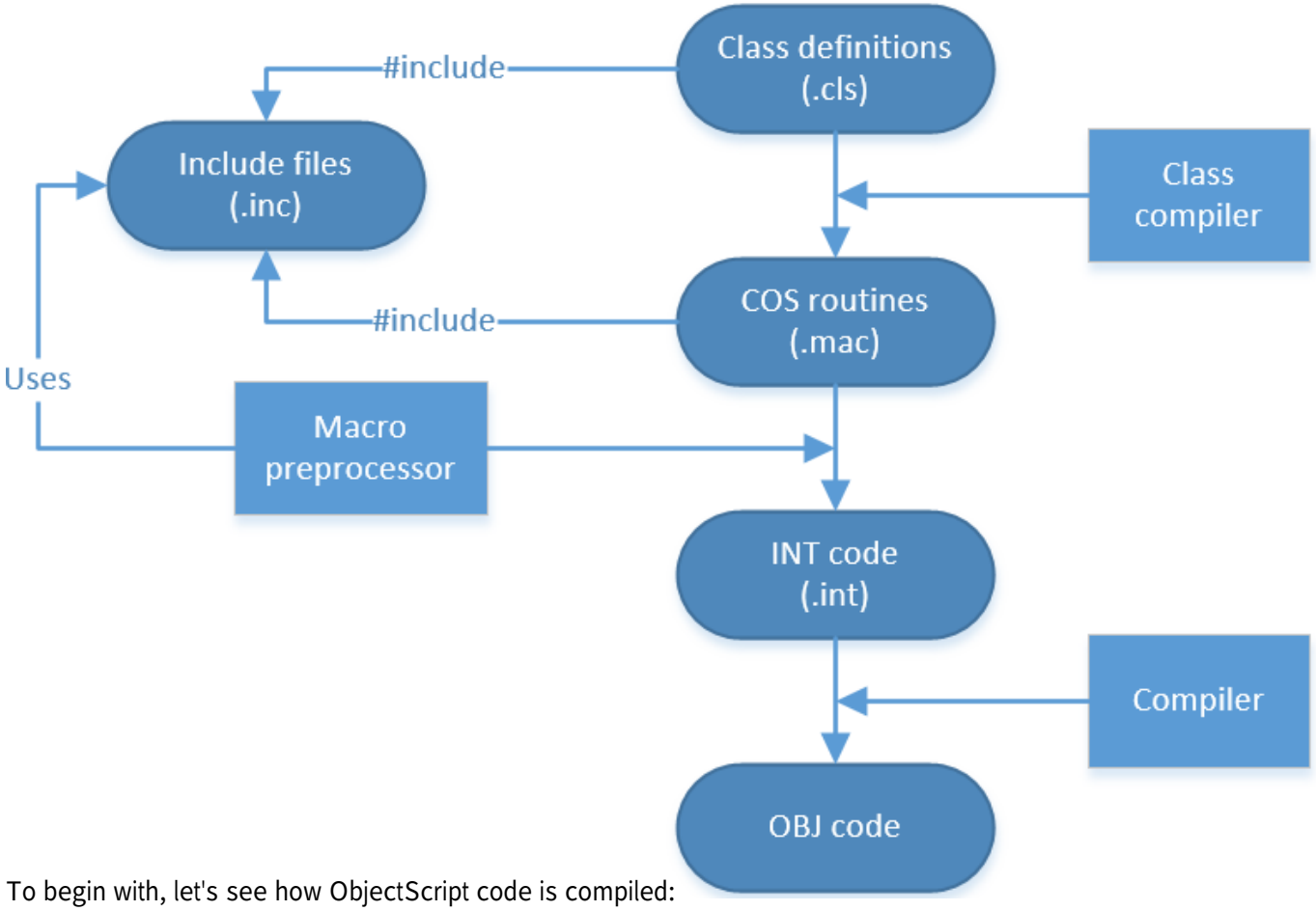
Article
Eduard Lebedyuk · Oct 18, 2016 7m read

# Macros in the InterSystems Caché

In this article I would like to tell you about macros in InterSystems Caché. A macro is a symbolic name that is replaced with a set of instructions during compilation. A macro can " unfold " in various instruction sets each time it is called, depending on the parameters passed to it and activated scenarios. This can be both static code and the result of ObjectScript execution. Let's take a look at how you can use them in your application.

## *Compilation*




To begin with, let's see how ObjectScript code is compiled:

1. The class compiler uses class definitions to generate MAC code
2. In some cases, the compiler uses classes as a basis for generating additional classes. You can see these classes in the studio, but you should not change them. This happens, for example, while generating classes that define web services and clients
3. The class compiler also generates a class descriptor used by Caché at runtime
4. The preprocessor (also referred to as macro preprocessor, MPP) uses INC files and replaces macros. Besides, it also processes embedded SQL in ObjectScript routines
5. All of these changes take place in the memory; the user's code remains unchanged
6. After that, the compiler creates INT code for ObjectScript routines. This layer is known as intermediate code. All access to data on this level is provided via globals
7. INT code is compact and can be read by a human. To view it in the studio, press Ctrl+Shift+V.

8. INT code is used for generating OBJ code
9. OBJ code is used by the Caché virtual machine. Once it's generated, CLS/MAC/INT code is no longer needed and can be deleted (for example, if we want to ship a product without the source code)
10. If the class is persistent, the SQL compiler will create corresponding SQL tables

## Macros

As I mentioned before, a macro is a symbolic name that is replaced by the preprocessor with a set of instructions. A macro is defined with the help of the #Define command followed by the name of the macro (perhaps with a list of arguments) and its value:

```
#Define Macro[(Args)] [Value]
```

Where can macros be defined? Either in the code or in standalone INC files containing only macros. The necessary files are included into classes at the very beginning of class definitions using the Include MacroFileName command – this is the main and preferred method of including macros into classes. Macros included this way can be used in any part of a class. You can use the #Include MacroFileName command to include an INC file with macros into MAC routines or the code of particular class methods.

Note, that method generators require #Include inside their own body if you want to use macros at compile time or use of IncludeGenerator keyword in a class.

To make macro available in studio autocomple, add `///` on a previous line:

```
///
#Define Macro[(Args)] [Value]
```

## Examples

*Example 1*

Let's jump to some examples now, and why don't we start with the standard " Hello World " message? COS code:

```
Write "Hello, World!"
```

We'll create a macro called HW that will write this line:

```
#define HW Write "Hello, World!"
```

All we need to do now is to write $$$HW ($$$ for calling the macro, then its name):

```
ClassMethod Test()
{
     #define HW Write "Hello, World!"
     $$$HW
}
```

It will be converted into the following INT code during compilation:

```
zTest1() public {
     Write "Hello, World!" }
```

The following text will be shown in the terminal when this method is called:

```
Hello, World!
```

*Example 2*

Let's use variables in the following example:

```
ClassMethod Test2()
{
     #define WriteLn(%str,%cnt) For ##Unique(new)=1:1:%cnt { ##Continue
         Write %str,! ##Continue
     }

     $$$WriteLn("Hello, World!",5)
}
```

Here the %str string is written %cnt time. The names of variables must start with %. The ##Unique(new) command creates a new unique variable in the generated code, while the ##Continue command allows us to continue defining the macro on the next line. This code converts into the following INT code:

```
zTest2() public {
     For %mmmu1=1:1:5 {
         Write "Hello, World!",!
     } }
```

The terminal will show the following:

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

*Example 3*

Let's proceed to the more complex examples. ForEach operator can be very useful for iterating through globals, let's create it:

```
ClassMethod Test3()
{
    #define ForEach(%key,%gn) Set ##Unique(new)=$name(%gn) ##Continue
    Set %key="" ##Continue
    For { ##Continue
        Set %key=$o(@##Unique(old)@(%key)) ##Continue
        Quit:%key=""

    #define EndFor    }

       Set ^test(1)=111
       Set ^test(2)=222
       Set ^test(3)=333
```

```
        $$$ForEach(key,^test)
            Write "key: ",key,!
            Write "value: ",^test(key),!
        $$$EndFor
}
```

Here is how it looks in INT code:

```
zTest3() public {
        Set ^test(1)=111
        Set ^test(2)=222
        Set ^test(3)=333
        Set %mmmu1=$name(^test)
        Set key=""
        For {
            Set key=$o(@%mmmu1@(key))
            Quit:key=""
            Write "key: ",key,!
            Write "value: ",^test(key),!
        } }
```

What is going on in these macros?

1. Write the name of the global to a new variable %mmmu1 ($name function)
2. The key assumes the initial empty string value
3. Iteration cycle starts
4. Next value to the key is assigned using indirection and the $order function
5. Post-condition is used to check if the key has assumed a "" value; if it has, the iteration is completed and the cycle ends
6. Arbitrary user code is executed – in this case, key and value output
7. The cycle closes

The terminal shows the following when this method is called:

```
key: 1
value: 111
key: 2
value: 222
key: 3
value: 333
```

If you are using lists and arrays inherited from the %Collection.AbstractIterator class, you can write a similar iterator for it.

*Example 4*

Yet another capability of macros is the execution of arbitrary ObjectScript code on the compilation stage and substitution of its results instead of a macro. Let's create a macro for showing the compilation time:

```
ClassMethod Test4()
{
      #Define CompTS ##Expression("""Compiled: " _ $ZDATETIME($HOROLOG) _ """,!")
```

```
      Write $$$CompTS
}
```

Which transforms into the following INT code:

```
zTest4() public {
      Write "Compiled: 18.10.2016 15:28:45",! }
```

The terminal will display the following line when this method is called:

```
Compiled: 18.10.2015 15:28:45
```

The ##Expression executes the code and substitutes the result. The following elements of the ObjectScript language can be used for input:

- Strings: "abc"
- Routines: $$Label^Routine
- Class methods: ##class(App.Test).GetString()
- COS functions: $name(var)
- Any combination of these elements

*Example 5*

Preprocessor directives #If, #ElseIf, #Else, #EndIf are used for selecting the source code during compilation depending on the value of the expression following a directive. For example, this method:

```
ClassMethod Test5()
{
    #If $SYSTEM.Version.GetNumber()="2016.2.0" && $SYSTEM.Version.GetBuildNumber()="7
36"
        Write "You are using the latest released version of Caché"
    #ElseIf $SYSTEM.Version.GetNumber()="2017.1.0"
        Write "You are using the latest beta version of Caché"
    #Else
        Write "Please consider an upgrade"
    #EndIf
}
```

Will be compiled into the following INT code in Caché version 2016.2.0.736:

```
zTest5() public {
    Write "You are using the latest released version of Caché"
}
```

And the following will be shown in the terminal:

```
You are using the latest released version of Caché
```

If we use Caché downloaded from the beta-portal, the compiled INT code will look differently:

```
zTest5() public {
    Write "You are using the latest beta version of Caché"
```

```
}
```

The following will be shown in the terminal:

```
You are using the latest beta version of Caché
```

Older versions of Caché will compile the following INT code with a suggestion to update the program:

```
zTest5() public {
    Write "Please consider an upgrade"
}
```

The terminal will show the following text:

```
Please consider an upgrade
```

This capability may come in handy, for example, in situations where you want to ensure compatibility of the client application with older and newer versions, where new Caché features may be used. Preprocessor directives #IfDef, #IfNDef serve the same purpose by verifying the existence or absence of a macro, respectively.

## *Conclusions*

Macros can make your code more readable by simplifying frequently used constructions and help you implement some of your application's business logic on the compilation stage, thus reducing the load at runtime.

## *What's next?*

In my next article, I will tell you about a more practical example of using macros in an application – a logging system.

## *Links*

- About compilation
- List of preprocessor directives
- List of system macros
- Class with examples from this article
- Part 2: Logging

#Beginner #Best Practices #Caché #Compiler #Terminal #ObjectScript #Tips & Tricks