Article
Maks Atygaev · Sep 6, 2017 5m read

Open Exchange

DeclarativeCOS — Declarative Programming in Caché

The **DeclarativeCOS** project is a heartfelt cry about programming in the COS language.

The purpose of the project is to draw attention of the public to improving the inner core of COS.

The idea of the project is the support of a laconic syntax for cycles and collections.

So what is this laconic something that I have come up with? Welcome to the examples below!

Examples

The key concept underlying the project is the declarative approach to writing code. You need to specify WHAT should be used and HOW.

I have personally always longed for a simple operator/command/magic spell in the COS terminal that would allow me to show a collection on the screen exactly the way I wanted to. Now we have two useful goodies to work with: **zforeach** and **$zjoin**!

```
>set words = ##class(%ListOfDataTypes).%New()
>do words.Insert("Hello")
>do words.Insert("World!")

>zforeach $zbind(words, "io:println")
Hello
World!
```

I should say a few words about the **$zbind** function. First of all, COS can be extended with custom commands and functions, which is described in detail in the corresponding *documentation* and this *article* on the developers ' community portal.

This function creates an instance of the Binder class. Its purpose is to bind a collection to a function that should be applied to each element of the collection. In this case, we use a standard function from **DeclarativeCOS** called **«io:println»**, which runs a simple command for the given value of **value**:

```
>w value,!
```

The **zforeach** command works with an instance of the Binder class by traversing the collection and applying a function to each element.

$zjoin — creates a string from a collection by joining its elements and adding a delimiter between them.

Example: create a date based on a day, month and year using " / " as a delimiter.

```
>s numbers = ##class(%ListOfDataTypes).%New()
>d numbers.Insert("04")
```

```
>d numbers.Insert("03")
>d numbers.Inset("2017")

>w $zjoin(numbers, " / ")
04 / 03 / 2017
```

$zmap — creates a new collection from the elements of the original collection and applies the specified function to each of its elements.

Example: convert every number to hex.

```
>set numbers = ##class(%ListOfDataTypes).%New()
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))

>write "[" _ $zjoin(numbers, ", ") _ "]"
[82, 12, 27]

>set hexNumbers = $zmap(numbers, "examples:toHex")

>write "[" _ $zjoin(hexNumbers, ", ") _ “]”
[52, C, 1B]
```

$zfind — finds the first element of a collection for which the specified function returns $$$YES. Otherwise, returns a null string.

Example: find a prime number.

```
>set numbers = ##class(%ListOfDataTypes).%New()
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))

>set primeNumber = $zfind(numbers, "examples:isPrime")

>write "[" _ $zjoin(numbers, ", ") _ "]"
[69, 41, 68]

>write "Prime number: " _ $select(primeNumber="":"<not found>", 1:primeNumber)
Prime number: 41
```

$zfilter — creates a new collection on the basis of the original collection, but only using elements for which the specified function returns $$$YES. If there are no such elements, it returns an empty collection.

Example: select odd numbers.

```
>set numbers = ##class(%ListOfDataTypes).%New()
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))

>set filteredNumbers = $zfilter(numbers, "examples:isOdd")

>write "[" _ $zjoin(numbers, ", ") _ "]"
[22, 71, 31]
```

```
>write "[" _ $zjoin(filteredNumbers, ", ") _ "]"
[71, 31]
```

$zexists — checks whether the collection has at least one element for which the specified function returns $$$YES.

Example: check whether the collection contains even numbers.

```
>set numbers = ##class(%ListOfDataTypes).%New()
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))
>do numbers.Insert($random(100))

>set hasEvenNumbers = $zexists(numbers, "examples:isEven")

>write "[" _ $zjoin(numbers, ", ") _ "]"
[51, 56, 53]

>write "Collection has" _ $case(hasEvenNumbers, 1:" ", 0:" no ") _ "even numbers"
Collection has even numbers
```

$zcount —count the number of elements in the collection for which the specified function returns $$$YES.

Example: count the number of palindromes.

```
>set numbers = ##class(%ListOfDataTypes).%New()
>do numbers.Insert($random(1000))
>do numbers.Insert($random(1000))
>do numbers.Insert($random(1000))

>set palindromicNumbersCount = $zcount(numbers, "examples:isPalindromic")

>write "[" _ $zjoin(numbers, ", ") _ "]"
[715, 202, 898]

>write "Count of palindromic numbers: " _ palindromicNumbersCount
Count of palindromic numbers: 2
```

Installation

To install DeclarativeCOS, all you need to do is to download two files from the project ' s GitHubepository:

install.base.xml — just classes. No Z-functions.
install.advanced.xml — %ZLANG routines that add z-functions.

How to use it

- Inherit the class from DeclarativeCOS.DeclarativeProvider.
- Implement the class method.
- Mark this method with the @Declarative annotation.
- Use the *z-functions* of DeclarativeCOS.
- Feel the bliss.

Detailed description

Step 1. Inherit the class from DeclarativeCOS.DeclarativeProvider.

```
Class MyPackage.IO extends DeclarativeProvider
{
}
```

Step 2. Implement the class method.

```
Class MyPackage.IO extends DeclarativeProvider
{

ClassMethod println(value As %String)
{
    w value,!
}

}
```

Step 3. Mark this method with the @Declarative annotation.

```
Class MyPackage.IO extends DeclarativeProvider
{

/// @Declarative("myIO:myPrintln")
ClassMethod println(value As %String)
{
    w value,!
}

}
```

Step 4. Use the z-functions of DeclarativeCOS.

```
>s words = ##class(%Library.ListOfDataTypes).%New()
>d words.Insert("Welcome")
>d words.Insert("to")
>d words.Insert("DeclarativeCOS!")

>zforeach $zbind(words, "myIO:println")
```

Step 5. Feel the bliss!

```
Welcome
to
DeclarativeCOS!
```

How it works

The DeclarativeCOS project uses the **^DeclarativeCOS** global to save information about the methods marked with the *@Declarative(declarativeName)* annotation.

Every such method is saved to the global in the following form:

```
set ^DeclarativeCOS(declarativeName) = $lb(className, classMethod)
```

For instance, for the io:println function:

```
set ^DeclarativeCOS("io:println") = $lb("DeclarativeCOS.IO", "println")
```

Each time we use the io:println function, the search is made in the global, then the $classmethod function calls the original method (DeclarativeCOS.IO # println) for the set value.

Conclusion

DeclarativeCOS is a contribution to the new Caché ObjectScript, the language that really helps developers to quickly write laconic, well-readable and reliable code. Feel free to dive into the ocean of criticism, support and opinions in the comments section under this post!)

Disclaimer: this article and my comments express my personal opinion only and have no relation to the official position of the InterSystems corporation.

#ObjectScript #Caché
Check the related application on InterSystems Open Exchange