Article
Maks Atygaev · Dec 11, 2016 4m read

Declarative development in Caché

Caché offers a number of methods for going through a collection and doing something with its elements. The easiest method uses a while-loop and lets you fulfill the task in an imperative manner. The developer needs to take care of the iterator, jumping to the next element and checking if the loop is within the collection.

But is it really what a developer should be concerned with?! A developer should be working on solving the problem at hand – quickly and producing code of the highest quality. It would be great to be able to just take a collection and apply a function to it that will perform the necessary operations on each element. No need to perform boundary checks, no need to create an iterator, no need to manually call a function for each element. This approach is called declarative programming.

> Declarative programming is when you write your code in such a way that it describes what you want to do, and not how you want to do it.
> (c) 1800-information

Let ' s now think how to solve the task declaratively, using built-in tools and capabilities of Caché.

In languages that support higher-order functions (like JavaScript), you can describe a function for processing a collection element and pass it as a parameter to another function to apply the passed function to each element.

```
[2, 3, 5, 7, 11, 13, 17].forEach(function(i) {
    console.log(i);
});
```

In this case, an anonymous function is created that outputs an element to the console. This function is passed as an argument to another function - forEach.

Unfortunately, Caché doesn ' t support higher-order functions that would provide a way to laconically accomplish your task. But let ' s think how we can implement this concept using Caché ' s standard means.

For starters, let ' take a look at a primitive implementation of a task that requires a loop to go through a collection with the subsequent output of each element.

```
set i = collection.Next("")

while (i '= "") {
    set item = collection.GetAt(i)

    w item,!

    set i = collection.Next(i)
}
```

To begin with, let ' s recall that Caché ObjectScript supports the OOP paradigm. And since it does, we should take a look at standard design patterns and try applying them to solve our problem. We need to go through the entire

collection and perform an action with each element. This makes me think about the Visitor pattern.

Let ' s define the **fp.Function** class with one abstract " execute " method.

```
Class fp.Function [ Abstract ] {
    Method execute(item As %Numeric) [ Abstract ] {}
}
```

Let us now define the implementation of this *" interface "* the **fp.PrintlnFunction** class.

```
Class fp.PrintlnFunction Extends (fp.Function, %RegisteredObject) {
    Method execute(item As %Numeric) {
        w item,!
    }
}
```

Okay, let ' s edit our original code a bit.

```
set function = ##class(fp.PrintlnFunction).%New()

set i = list.Next("")

while (i '= "") {
    set item = list.GetAt(i)

    do function.execute(item)

    set i = list.Next(i)
}
```

Let ' s now encapsulate the collection traversal algorithm. Let ' s create the **IterableStream** class.

```
Class fp.IterableStream Extends %RegisteredObject {
    Property iterator As %Collection.AbstractIterator [ Private ];

    Method %OnNew(iterator As %Collection.AbstractIterator) As %Status [ Private, Ser
verOnly = 1 ] {
        set ..iterator = iterator

        return $$$OK
    }

    Method forEach(function As Function) {
        set i = ..iterator.Next("")

        while (i '= "") {
            set item = ..iterator.GetAt(i)

            do function.execute(item)

            set i = ..iterator.Next(i)
        }
    }
}
```

The solution can now be presented in the following way:

```
do ##class(IterableStream).%New(list).forEach(##class(PrintlnFunction).%New())
```

You can encapsulate the algorithm of creating a wrapper for the while-loop. To do this, let ' s create the **Streams** class.

```
Class fp.Streams {
    ClassMethod on(iterator As %Collection.AbstractIterator) As IterableStream {
        return ##class(IterableStream).%New(iterator)
    }
}
```

You can then rewrite the solution in the following way:

```
do ##class(Streams).on(list).forEach(##class(PrintlnFunction).%New())
```

So, the problem has been solved declaratively. Yes, we have new classes. Yes, we have more code now. It must be noted, though, that the resulting code is more concise and transparent. It doesn ' t contain distractions but helps us concentrate on the problem being worked on.

If you imagine classes like **Function**, **Streams**, **IterableStream** in the place of standard Caché classes, you will only need to create the **PrintlnFunction** class.

And that was my two cents on declarative programming in Caché. Happy coding, everyone!

#Object Data Model #ObjectScript #Tips & Tricks #Caché