OOP & NEOBOOK

Object oriented programming (OOP) has many advantages:

- Object oriented code can be reused in several projects in a fast way
- Object oriented code is clearly structured and thus highly readable
- Objects can easily be extended with new functionality
- Objects with a clearly defined interface are easy to use.

Neobook's scripting language is not strong in its OOP support as Java or C++. However, I think it's still a good idea to understand how objects are used, especially how they are used in 'not true' OOP languages such as Javascript. Javascript is an object-based language, meaning that you can modularize your code with functions or objects. In Neobook we can use the ideas of Javascript developers, at least to some extent.

1. What is an object?

The structure of an object (or class) can be defined as a collection of

- properties, i.e. things that describes the features of an object;
- methods, i.e. functions that do some operations on the object properties.

2. Creating an object

The function is the central OOP concept in Javascript, which could be easily be applicated in Neobook's scripting language.

2.1. Creating an object in Javascript

Defining a custom JavaScript object is similar to defining the so called constructor function of Javascript. The syntax of an object with two parameters and one method is as follows:

```
function object(parameter1, parameter2)
{
 this.property1 = parameter1;
 this.property2 = parameter2;
 this.method1 = memberfunction1;
}
```

2.2. Creating an object in Neobook

Creating a custom Neobook object is quite similar to constructing a Javascript object. The syntax is as follows:

```
"[FunctionName]" "[parameter1]" "[parameter2]"
```

The body of the Neobook function can be defined as a collection of properties which are -as in the JavaScript example- initialized by parameters and methods. The syntax of an object with two parameters and one method is as follows:

```
SetVar "[this.property1]" "[%parameter1]"
SetVar "[this.property2]" "[%parameter2]"
GoSub "memberfunction1"
```

Instead of the subroutine GoSub we can use of course inline Neobook functions (functions within the object). I prefer to use GoSub; so the function body is more clearly defined and readable.

3. Creating an instance of an object

The way an instance of an object is created in Javascript, can be simulated in Neobook.

3.1. Creating an instance of an object in Javascript

The following will create a new instance of the mcMath object with the name myMath:

```
myMath = new mcMath (3,2);
```

To access the properties (with x = property1 and y = property2)

```
myMath.x (> value is 3)
myMath.y (> value is 2)
myMath.Product (> value is 6)
```

3.2. Creating an instance of an object in Neobook

Creating an instance of an object as in Javascript is in Neobook not possible. However we can define an InstanceParameter (IP).

```
Call "[FunctionName]" "[IP]"  "[parameter1]" "[parameter2]"
```

The body of the Neobook function can be defined as:

```
SetVar "[[%IP].property1]" "[%parameter1]"
SetVar "[[%IP].property2]" "[%parameter2]"
GoSub "memberfunction1"
```

The following will create a new instance of the mcMath object with the name myMath:

```
Call "mcMath" "myMath" "3" "2"
```

To access the properties (with x = property1 and y = property2):

```
[myMath.x] (> value is 3)
[myMath.y] (> value is 2)
[myMath.Product] (> value is 6)
```

The memberfunction of mcMath -inline or as subroutine- is as follows:

```
Math "[myMath.x] * [myMath.y]" "2" "[myMath.Product]"
```

4. Examples of Neobook coding

4.1. Unstructured Neobook code

Unstructured programming results in a sequence of variable definitions and action commands:

```
SetVar "[myMath.x]" "3"
SetVar "[myMath.y]" "2"
Math "[myMath.x] * [myMath.y]" "2" "[myMath.Product]"
SetVar "[myMath.Result]" "[myMath.Result][myMath.x] * [myMath.y] = [myMath.Product][#10][#13]"

SetVar "[myMath.x]" "14"
SetVar "[myMath.y]" "3"
Math "[myMath.x] * [myMath.y]" "2" "[myMath.Product]"
SetVar "[myMath.Result]" "[myMath.Result][myMath.x] * [myMath.y] = [myMath.Product][#10][#13]"

SetVar "[myMath.x]" "5"
SetVar "[myMath.y]" "2"
Math "[myMath.x] * [myMath.y]" "2" "[myMath.Product]"
SetVar "[myMath.Result]" "[myMath.Result][myMath.x] * [myMath.y] = [myMath.Product][#10][#13]"

SetVar "[myMath.x]" "16"
SetVar "[myMath.y]" "8"
Math "[myMath.x] * [myMath.y]" "2" "[myMath.Product]"
SetVar "[myMath.Result]" "[myMath.Result][myMath.x] * [myMath.y] = [myMath.Product][#10][#13]"
```

4.2. Procedural Neobook code

Procedural programming with subroutines results in a better code:

```
:myMathOperation
 Math "[myMath.x] * [myMath.y]" "2" "[myMath.Product]"
 SetVar "[myMath.Result]" "[myMath.Result][myMath.x] * [myMath.y] = [myMath.Product][#10][#13]"
Return

SetVar "[myMath.x]" "3"
SetVar "[myMath.y]" "2"
GoSub "myMathOperation"

SetVar "[myMath.x]" "14"
SetVar "[myMath.y]" "3"
GoSub "myMathOperation"

SetVar "[myMath.x]" "5"
SetVar "[myMath.y]" "2"
GoSub "myMathOperation"

SetVar "[myMath.x]" "16"
SetVar "[myMath.y]" "8"
GoSub "myMathOperation"
```

However, it can be done better and more easily.

4.3. OOP Neobook code

Let's make an OOP alternative to the code of 4.1 and 4.2:

1. We define a Neobook object mcMath with an InstanceParameter (IP), two parameters, one inline memberfunction (Math) and one variable definition ([[%IP].Result]).

```
Math "[%parameter1] * [%parameter2]" "2" "[[%IP].Product]"
SetVar "[[%IP].Result]" "[[%IP].Result][%parameter1] * [%parameter2] = [[%IP].Product][#10][#13]"
```

2. We invoke this object by a Call command:

```
Call "mcMath" "myMath" "[parameter1]" "[parameter2]"
```

3. The OOP alternative to the code of 4.1 and 4.2 is as follows:

```
Call "mcMath" "myMath" "3" "2"
Call "mcMath" "myMath" "14" "3"
Call "mcMath" "myMath" "5" "2"
Call "mcMath" "myMath" "16" "8"
```

So, the OOP alternative consists of four easy and clearly defined Call commands!

5. OOP: slowing down your Neobook application?

One problem with using custom objects is that function calls and subroutines make your application slower than straight code. Normally, the time difference between using custom objects or not is imperceptible to the end user. However, this could be a potential side effect.

6. Always OOP?

In general I prefer code that is readable and easy to modify. In addition, I like the way of coding that results in less, not in more code. As someone said:

```
"Adding objects to your code is like adding salt to a dish: use a little, and it's a savory
seasoning; add too much and it utterly ruins the meal."
```

Nevertheless, it's always worth considering reasons to use an object oriented approach, looking for a balance between quality and quantity of Neobook code.