

Author: Dräc

http://drac.site.chez.tiscali.fr/Tutorials%20Programming%20PureBasic/indexTutorials_en.htm

Version: 22 May 2005

Converted to PDF by Techjunkie






- OOP
- PureB Language

PureBasic Programming



Object-Oriented Programming (29)

Title	0	Format	Translation	Last Modification
■ Why OOP in PureBasic ?		HTML		09/05/2005
■ Object concepts		HTML		09/05/2005
➔ Notion of Object				09/05/2005
➔ Notion of Class				09/05/2005
➔ Notion of Instance				09/05/2005
➔ Encapsulation				09/05/2005
➔ Inheritance				09/05/2005
➔ The Overload				09/05/2005
➔ Notion of Abstract Class				09/05/2005
■ First Implementation of the concepts		HTML		09/05/2005
➔ Concrete Class and Abstract Class				09/05/2005
➔ Instanciation				09/05/2005
➔ Encapsulation				09/05/2005
➔ Inheritance				09/05/2005
➔ Overload				09/05/2005
➔ Conclusion				09/05/2005
■ Interface instruction		HTML		09/05/2005
➔ Initialization				09/05/2005
■ Second Implementation of the concepts		HTML		09/05/2005
➔ Notion of Object Interface				09/05/2005
➔ Object Instanciation and Object Constructor				15/05/2005
➔ Object Initialization				15/05/2005
➔ Object Destructor				09/05/2005
➔ Memory Allocations				15/05/2005
➔ Encapsulation				15/05/2005
➔ Inheritance				15/05/2005
➔ Get() and Set() objet methods				15/05/2005
■ Synthesis and notation		HTML		09/05/2005

■ Conclusion	HTML	 	09/05/2005
■ Appendix	HTML	 	15/05/2005
➡ Optimisation: Get() and Set() methods			15/05/2005
➡ Use of the Linked lists			15/05/2005



PureBasic Language (0)

Titre	0	Format	Translation	Last Modification
-------	---	--------	-------------	-------------------

PureBasic and the Object-Oriented Programming

or the OOP demystified

Why OOP in PureBasic ?


It can be surprising to try realizing in PureBasic, which is a procedural language, an object-oriented representation, as far as numerous languages adapted to the OOP exist.


But the fact that some programming languages are called "Object" and the other one are not, translates only the existence of supplementary keywords, which facilitate the writing of these programs. So, object-oriented languages enrich semantics but not modify at all the compilation aspects with regard to a none-object language. They are only adding a layer over this last one.

Thus we can implement completely the concepts in PureBasic but the price is a rigor both of development and of notation. It is there the immediate advantage of object-oriented languages.

Nevertheless, the application of the object-oriented concepts in PureBasic offer, besides the possibility of scheduling according to OO philosophy, the interest to reveal certain underlying mechanisms of the object-oriented languages keywords.

The Object-Oriented Programming introduces concepts as the object, the inheritance or the polymorphism. Now, we are going to see how the main concepts can be realized in PureBasic. But before it, these concepts must be defined.

 [Contents](#)

[Next Page](#) 

[\[1-2-3-4-5-6-7-8\]](#)

[Top of the page](#)

PureBasic and the Object-Oriented Programming

Object concepts

The notion of Object

An object has an internal state:

- The state is represented by the value of each inner components at a given moment,
- A component is either a value or another object.

It uses and gives services:

- It interacts with the outside through functions called methods.

It is unique (notion of identity)

An object can be seen at two levels:

- By the services which it returns: external view (specification). It is User side,
- By the way that services are implemented within it : internal view (implementation). It is the Developer side.

From the point of view of the developer, the object is thus a contiguous memory area containing information: variables called **attributes** and functions called **methods**.

The fact that functions are called methods of an object, means the fact that they are appropriate to the object and allow the manipulation of the attributes of this object.

The notion of Class

It is about an extension of the type notion found in PureBasic.

In a given context, several objects can have the same structure and the same behavior. Then, they can be grouped together in the same "Class".

From the point of view of the developer, the Class defines the content of an object of this Class: the nature of its attributes (type of each variables) and its methods (Names, implementation).

If the type of a variable is an integer, the type of an object is its Class.

The notion of instance

An instance is an object defined from a Class.

Such a process is called the instantiation.

It corresponds to the assignment of variables in PureBasic.

The object is **normally initialized** at this time.

The encapsulation

In theory, the manipulation of the object attributes should be able only through the methods. This technique, which allows making visible to the user only a part of the object, is called encapsulation.

The encapsulation advantage is to guarantee the integrity of the attributes. Indeed, the developer is the only one who, through the methods given to the user, manages the modifications allowed to an object.

At our level, it is at least what it shall be retain about encapsulation concept.

The inheritance

The inheritance allows defining new Classes by using already existing Classes.

From the point of view of the developer, it means being able to add attributes, methods and even to modify some methods, to an existing Class in order to define another one.

There are two kinds of inheritances:

- The simple inheritance: the new Class is defined from a single existing Class
- The multiple inheritance: the new Class is defined from several existing Classes

The multiple inheritance is complex to implement and it will not be approached here. Thus, the document deals only with the simple inheritance.

Terminology:

The Class which inherits from another Class, is often called Child Class

The Class which gives its inheritance to a Child Class is often called Mother Class.

The overload

A method is overloaded if it realizes different actions according to the nature of the aimed objects.

Let us take an example:

The following objects: circle, rectangle and triangle are geometrical forms.

We can define for these objects the same Class with the given name: Form.

Thus, above objects are instances of the Class Form.

If we want to display objects, the Class Form needs to have a "Draw" method.

So endowed, every object has a "Draw" method to display themselves. Now, this method could not be the same, as we want to display a circle or a rectangle.

The objects of the same Class use the same "Draw" method, but the implementation of the method will be specific to the object nature (Rectangle, Triangle).

The "Draw" method is overloaded: for the user, displaying a circle or a rectangle, it is made in the same way.

From the point of view of the developer, the methods implementation needs to be different.

Instead of overloaded method, we can speak also about polymorph method (having several forms).

Notion of abstract Class

As shown above, a Class includes the definition of both the attributes and the methods of an object.

Let us suppose that we cannot give the implementation of one of the methods of the Class. The method is only a name without code. We speak then about abstract method.

A Class containing at least an abstract method is qualified as abstract Class.

We can wonder the reason to be of an abstract class, because an object of a such Class cannot be create. Abstract Classes allow defining [Object Classes](#) considered by opposition as concrete. The link between them is made by inheritance where the concrete Class takes care in giving the missing implementations to the inherited abstract methods.

Thus, abstract Classes have an interface responsibility, because they describe the generic specification of all the Classes which inherits from them.

PureBasic and the Object-Oriented Programming

Concepts Implementation

In the next section, we present how the object concepts can be implemented in PureBasic.

This implementation doesn't refer to what is programmed in object-oriented languages. Furthermore, the goal of an implementation is to be improved or to be adapted to the need.

Thus, we propose here one of these implementations with its own advantages and limits.

First Implementation

Concrete Class and Abstract Class

As seen, the Class defines the contents of an object:

- Its attributes (each variable type)
- Its methods (Names, implementation)

For example, if we want to represent Rectangle objects and to display them on the screen, we shall define a Class Rectangle including a **Draw()** method.

The Class Rectangle could have the following construction:

```
Structure Rectangle
  *Draw.l
  x1.l
  x2.l
  y1.l
  y2.l
EndStructure

Procedure Draw_Rectangle(*this.Rectangle)
...
EndProcedure
```

where x1, x2, y1 and y2 are four attributes (diametrically opposite point coordinates of the rectangle) and *Draw is a pointer referencing to the drawing function which displays Rectangles.

*Draw is here a function pointer used to contain the address of the wished function: **@Draw_Rectangle()**.

By using **CallFunctionFast()**, a such referenced function can be used.

Thus, we see that the proposed Structure is completely adapted to the Class notion:

We find on it the definition of the object attributes: x1, x2, y1 and y2 are here Long variables.

We find on it the definition of the object method: here the **Draw()** function thanks to a function pointer.

If a so defined Class is followed by the method implementations (in our example it is the **Draw_Rectangle()** Procedure/EndProcedure block statement), the Class will be a concrete Class.

In the opposite, the Class will be a abstract Class.



**this always refers to the object on which the method must be applied. This notation is applied in our example with the method Draw_Rectangle().*

Instanciation

Now, to create an object called Rect1 from the Class Rectangle, write:

```
Rect1.Rectangle
```

To initialize it, simple write:

```
Rect1\Draw = @Draw_Rectangle()
Rect1\x1 = 0
Rect1\x2 = 10
Rect1\y1 = 0
Rect1\y2 = 20
```

Next, to draw Rect1 object, do:

```
CallFunctionFast(Rect1\Draw, @Rect1)
```

Encapsulation

In this implementation, the encapsulation does not exist, simply because there is no means to hide the attributes or the methods of such an object.

By writing Rect1\x1, the user can access to the attribute x1 of the object. This way was used to initialize the object.

We shall see in the second implementation, how that can change.

Although importing, this notion is not the most essential to practice OOP.

Inheritance

Now we want to create a new Class Rectangle with the supplementary capability to Erase rectangles from the screen.

We can perform the new Class Rectangle2 by using the existing Class Rectangle and by providing to it a new method called `Erase()`.

A Class being a Structure, we are going to take advantage of the extension property from structure. So, the new Class Rectangle2 can be:

```
Structure Rectangle2 Extends Rectangle
*Erase.l
EndStructure

Procedure Erase_Rectangle(*this.Rectangle2)
...
EndProcedure
```

The Class Rectangle2 includes as well members of the previous Class Rectangle and the new `Erase()` method. The object instantiation from this new Class gives:

```
Rect2.Rectangle2
```

```
Rect2\Draw = @Draw_Rectangle()
Rect2\Erase = @Erase_Rectangle()
Rect2\x1 = 0
Rect2\x2 = 10
Rect2\y1 = 0
Rect2\y2 = 20
```

To use **Draw()** and **Erase()** methods of Rect2, we shall proceed in the same way as previously.

That shows that Rectangle2 Class inherited properties of Class Rectangle.



The inheritance is a category of polymorphism. The object Rect2 can be seen also as an Object from the Class Rectangle. For this, just don't use the Erase() method! By inheritance, the object carries several forms: those of the objects coming from the Mothers Classes. It is named as inheritance polymorphism.

Overload

During the initialization of an object, the function pointers are initialized by assigning to them the method addresses, which are convenient for the object.

So, for an object Rect from Class Rectangle, by writing:

```
Rect1\Draw = @Draw_Rectangle()
```

we can use the **Draw()** method as following:

```
CallFunctionFast(Rect1\Draw, @Rect1)
```

Now, imagine that it is possible to implement another method for a rectangle display (by using a different algorithm than for the first method).

Let us call this implementation as **Draw_Rectangle2()**:

```
Procedure Draw_Rectangle2(*this.Rectangle)
...
EndProcedure
```

It is possible to initialize our object Rect1 with this new method without effort:

```
Rect1\Draw = @Draw_Rectangle2()
```


To use the method, it shall be writed again:

```
CallFunctionFast(Rect1\Draw, @Rect1)
```

In a hand ([Draw_Rectangle\(\)](#) method) as in the other hand ([Draw_Rectangle2\(\)](#) method) the use of the Rect1 method is strictly identical.

By the only line " `CallFunctionFast (Rect1\Draw, @Rect1)` ", it is not possible for us to distinguish the [Draw\(\)](#) method that the Rect1 object really uses.

To know this, it is necessary to go back to the initialization of the object.


The notion of function pointer allows the overloading of the [Draw\(\)](#) method.


One limitation. The use of the `CallFunctionFast()` instruction implies to pay attention among the parameter numbers.

Conclusion:

In this first implementation, we have an object able to answer to the main object-oriented concepts with some limitations.

In fact we have just put the bases for realizing a more complete object, this thanks to the PureBasic **Interface** instruction.

 [Contents](#)

[Next Page](#) 

[[1](#)-[2](#)-**[3](#)**-[4](#)-[5](#)-[6](#)-[7](#)-[8](#)]

[Top of the page](#)

PureBasic and the Object-Oriented Programming

Interface instruction

Syntax:

```
Interface <Name1> [Extends <Name2>]
  [Procedure1]
  [Procedure2]
  ...
EndInterface
```

The PureBasic Interface instruction allows grouping under the same Name (< Name1 > in the above box) various procedures.

Ex :

```
Interface My_Object
  Procedure1(x1.i, y1.i)
  Procedure2(x2.i, y2.i)
EndInterface
```

By declaring an element of My_Object type, the access to its procedures looks like this:
The statement is made in the same way as for a Structure:

```
Object.My_Object
```

To use the object functions, write directly:

```
Object\Procedure1(10, 20)
Object\Procedure2(30, 40)
```

Using the Interface instruction leads to a very practical and pleasant notation to operate a procedure.
By writing " Object\Procedure1(10, 20) ", the Procedure1() from Object is called.
This notation is typical of the Object-oriented Programming.

Initialization:

To variable statements, follows normally the variable initialization.
It's the same when we declare an element from an Interface.

Contrary to all expectations, it is not enough to give the name of a procedure inside the Interface/EndInterface block to make a reference to the implementation of this procedure, e.i. to refer to the Procedure/EndProcedure block of the required procedure.

In fact, you can rename procedures in a Interface/EndInterface block, by giving the names you want for the procedures that you go to use.

Then, how to connect this new name with the true procedure?

As for overloaded methods, the solution is in the function addresses.

It is necessary to see names, into the Interface/EndInterface block, as function pointers in which the required function addresses are attributed.

However, to initialize the function pointers of an Interface specified element, it is necessary to process differently than for a Structure specified element.

In fact, it is not possible to initialize individually each field defined by an Interface, because remember that Object \Procedure1() means a procedure call.

The initialization comes true indirectly by giving to the element, the address of a variable consisted of function pointers previously initialized.

Ex: by resuming the Interface My_Object, let us consider the following Structure describing the function pointers:

```
Structure My_Methods
*Procedure1.l
*Procedure2.l
EndStructure
```

and the associated initialized variable:

```
Methods.My_Methods
Methods\Procedure1 = @My_Procedure1()
Methods\Procedure2 = @My_Procedure2()
```

where **My_Procedure1()** and **My_Procedure2()** are the required procedure implementations.

Then, the initialization of Object from Interface My_Object will look like this:

```
Object.My_Object = @Methods
```

Next, by writing

```
Object\Procedure2(30, 40)
```

the Object **Procedure2()** function is called according **My_Procedure2()** implementation.



When an Interface specified element is declared, it is essential to initialize it before using element procedures. Thus it is advised to initialize the element at statement time.



The composition of the Structure, which describing the function pointers, has to be the exact representation of the Interface composition. It has to contain the same number of fields and to respect the order of them to insure the right assignation between names and addresses of each function. It is only on these conditions that the element will be correctly initialized.

To summarize, using an Interface correspond into:

- an Interface describing the required procedures to use,
- a Structure describing the function pointers,
- a Structured variable initialized with the required function addresses.

It is also:

- benefit from an object-oriented notation
- be able to rename easily the procedures



[Contents](#)

[Next Page](#)



[\[1-2-3-4-5-6-7-8\]](#)

[Top of the page](#)

PureBasic and the Object-Oriented Programming

Second Implementation

In our first implementation, object concepts was translated in a way more or less extend. Now, we are presenting how this first implementation can be improved thanks to the use of the Interface instruction.

Notion of Object interface:

The purpose of encapsulation is first to make visible, to the user, part of an object contents. The visible part of the contents is called **interface**, the other hidden part is called **implementation**.

Thus, the object interface is the only input-output access that the user has to act on an object.

It is the responsibility that we are going to give in our use of the Interface instruction.

The Interface instruction allows us to group under the same Name, all or any of the methods of an object that the user will have the access right.

Object Instanciation and Object Constructor

To use an interface, it is at first to arm with:

1. an Interface describing the required methods,
2. a Structure describing the pointers of the corresponding functions,
3. a structured and initialized variable.

Stage 1, consists in specify the object Interface, which is not a difficulty. Just name methods.

Stages 2 and 3 are linked. In our object approach, we already have the adapted Structure: it is the one who describes the Class of an object.

Moreover, the Interface and the Class of an object are close: both contain function pointers.

Simply, the Interface instruction does not contain the attributes of the Class but only all or any of the methods of the Class.

It is thus completely possible to use the Class of an object to initialize the Interface. This step is the most natural. Let us remind that the interface is the visible part of the Class of an object, it is thus normal that the Class determines the Interface.

To see how to process, let us resume the Class Rectangle2 which provided the **Draw()** and **Erase()** methods.

The corresponding Class is the following one

```
Structure Rectangle2
*Draw.I
*Erase.I
x1.I
x2.I
y1.I
y2.I
EndStructure

Procedure Draw_Rectangle(*this.Rectangle2)
...
EndProcedure

Procedure Erase_Rectangle(*this.Rectangle2)
...
EndProcedure
```

The associated Interface is the following:

```
Interface Rectangle
Draw()
Erase()
EndInterface
```

Because we want that the user employ only the interface, it is not any more necessary to create an object directly from the Class Rectangle2.

The object will thus be create by writing:

```
Rect.Rectangle
```

instead of Rect.Rectangle2 .

However, you should not forget to connect the Interface to the Class.
For this, it is necessary to initialize the Rect object during the statement.
Correction made, the good instruction to assign the object is the following one:

```
Rect.Rectangle = New_Rect(0, 10, 0, 20)
```

New_Rect() is a function which realizes the initialization operation.
What we already know about it, it is the returned value which is the memory address containing the function addresses to be used by the interface.

Here is the body of the **New_Rect()** function:

```
Procedure New_Rect(x1.l, x2.l, y1.l, y2.l)
*Rect.Rectangle2 = AllocateMemory(SizeOf(Rectangle2))

*Rect \Draw = @Draw_Rectangle()
*Rect \Erase = @Erase_Rectangle(

*Rect \x1 = x1
*Rect \x2 = x2
*Rect \y1 = y1
*Rect \y2 = y2

ProcedureReturn *Rect
EndProcedure
```

This function assigns a memory area of the object Class size.
It initializes then the methods and the attributes of the object.
It ends by retrieving the memory area address.
Because we find at the beginning of this memory area the addresses of **Draw()** and **Erase()** functions, the interface is effectively initialized.

To access to the Rect object methods, just write:

```
Rect\Draw()
Rect\Erase()
```

Then, it is checked that:

- Class Rectangle2 allows the object Interface initialization.
- Rect, declared thanks the interface, is an object of the Class Rectangle2 which can use the **Draw()** and the **Erase()** methods.

Thus the Interface instruction and the **New_Rect()** function performe the instantiation of an Rect object from the Class Rectangle2.

The **New_Rect()** function is called as the object **Constructor** of the Class Rectangle2.



*All the Methods implementations (blocks Procedure/EndProcedure) have to contain, as first argument, the *this pointer of the object to which the function must be applied. In contrast, the *this argument does not appear any more at the Interface level. In fact, as this instruction allows us to write Rect\Draw(), we know that the Draw() method is involved by the Rect object: there is no ambiguity. Everything takes place as if the object Rect was "aware" of its state.*



The Constructor could receive, as supplementary parameters, the function addresses which implement the methods. It is nothing here because we know the methods implementation: it is that from the class. On the other hand we do not know the initial state that the user wants to give to the object. Thus, it is possible that the Constructor contains parameters for the attributes initialization. It is the present case, the New_Rect() required entries are the coordinates (x1, y1) and (x2, y2) of the diametrically opposite points of the rectangle.

Object Initialization

After having assigned the required memory area for an object, the Constructor initializes the various members of the object (methods and attributes).

This operation can be isolated in a specific procedure, which the Constructor will call.

This precaution allows distinguishing the memory allocation and the object initialization. This will be very useful to achieve afterward the concept of inheritance, because a single memory allocation is sufficient, but several initializations will be necessary.

However we shall separate the initialization of the methods and that of the attributes. Indeed, the methods implementation depends on the class, while the attributes initialization depends on the object itself (see previous remark ➡).

In our example, the two initialization procedures will be implemented as:

```
Procedure Init_Mthds_Rect(*Rect.Rectangle2)
*Rect \Draw = @Draw_Rectangle()
*Rect \Erase = @Erase_Rectangle()
EndProcedure

Procedure Init_Mbers_Rect(*Rect.Rectangle2, x1.l, x2.l, y1.l, y2.l)
*Rect\x1 = x1
*Rect\x2 = x2
*Rect\y1 = y1
*Rect\y2 = y2
EndProcedure
```

and the Constructor became:

```

Procedure New_Rect(x1.i, x2.i, y1.i, y2.i)
*Rect = AllocateMemory(SizeOf(Rectangle2))
Init_Mthds_Rect(*Rect)
Init_Mbers_Rect(*Rect, x1, x2, y1, y2)
ProcedureReturn *Rect
EndProcedure

```

Object Destructor

Always associated to an object Constructor is the object Destructor.

During the construction of an object, a memory area was assigned to store the method and attribute definitions.

When an object is not useful any more, it must be destroyed to free the computer memory.

This process is made by using a specific function called **Destructor** of the object.

In our example of Class Rectangle2 objects, the Destructor is:

```

Procedure Free_Rect(*Rect)
FreeMemory(*Rect)
EndProcedure

```

and will be used as:

```
Free_Rect(Rect2)
```



The Destructor can be seen as a method of the object. But to avoid weighing down the object and keeping homogeneity with the Constructor, we have preferred to see it as a function of the Class.

Delete an object by its Destructor, means releasing the memory area which contains the object information (methods to use and attributes states) but not deleting the object infrastructure.

So, in our example, having made:

Free_Rect(Rect2)

Rect2 can reuse without specifying its type again:

Rect2 = New_Rect(0, 10, 0, 20)

Rect2\Dessiner()

Definitely, when an object instantiation is realized, as hereafter:

Rect2. Rectangle

the life cycle of object Rect2 is subjected to the same rules as those of the variables because Rect2 is at first a variable: it is a structured variable containing the function pointers of the object methods. (See also the reminder which follows)

Small reminder: the life cycle of a variable is linked to the life cycle of the program part where the variable is declared:



- *If the variable is declared inside a procedure, its life cycle will be linked to that of the procedure, which is equal to the function time of use.*
- *If the variable is declared outside any procedure, in the program's main core, its life cycle is linked to that of the program.*

Memory Allocations

In every new instantiation, the Constructor has to allocate dynamically a memory area according to the size of the information describing the object.

For that purpose, the Constructor can use the `AllocateMemory()` command associated with `FreeMemory()` command for the Destructor.

But it can be quite another command achieving dynamic memory allocation.
Under Windows OS, API can be directly used for example.

Standard PureBasic library proposes linked lists, which also allow to allocate dynamically some memory.

To know more about the way of using linked lists to implement an object, read the [Appendix](#) of this tutorial.

Encapsulation

Let us imagine now that we want to give to the user only access to the `Draw()` method of the Class Rectangle. We shall begin by defining the wished interface:

```
Interface Rectangle
Draw()
EndInterface
```

The new object instantiation stays the same:

```
Rect.Rectangle = New_Rect()
```

with,

```
Procedure Init_Mthds_Rect(*Rect.Rectangle2)
*Rect \Draw = @Draw_Rectangle()
*Rect \Erase = @Erase_Rectangle()
EndProcedure

Procedure Init_Mbers_Rect(*Rect.Rectangle2, x1.i, x2.i, y1.i, y2.i)
*Rect\x1 = x1
*Rect\x2 = x2
*Rect\y1 = y1
*Rect\y2 = y2
EndProcedure

Procedure New_Rect(x1.i, x2.i, y1.i, y2.i)
*Rect = AllocateMemory(SizeOf(Rectangle2))
Init_Mthds_Rect(*Rect)
Init_Mbers_Rect(*Rect, x1, x2, y1, y2)
ProcedureReturn *Rect
EndProcedure
```

...because the first function address is the `Draw()` method one.

Now, imagine that we want giving to the user only the access to the `Erase()` method. We shall begin by defining the new interface:

```

Interface Rectangle
  Effacer()
EndInterface

```

Nevertheless the new object instantiation cannot use the `New_Rect()` Constructor.
In the opposite case, the result would be identical to the previous case.

Thus, it is necessary to create a new Constructor able to return the adapted function address.

Hereafter one of them is given:

```

Procedure Init_Mthds_Rect2(*Rect.Rectangle2)
  *Rect \Draw = @Erase_Rectangle()
  *Rect \Erase = @Draw_Rectangle()
EndProcedure

Procedure Init_Mbers_Rect(*Rect.Rectangle2, x1.i, x2.i, y1.i, y2.i)
  *Rect\x1 = x1
  *Rect\x2 = x2
  *Rect\y1 = y1
  *Rect\y2 = y2
EndProcedure

Procedure New_Rect2(x1.i, x2.i, y1.i, y2.i)
  *Rect = AllocateMemory(SizeOf(Rectangle2))
  Init_Mthds_Rect2(*Rect)
  Init_Mbers_Rect(*Rect, x1, x2, y1, y2)
ProcedureReturn *Rect
EndProcedure

```

You notice that the function addresses were just inverted at the initialization level.
Certainly, it is not very elegant to allocate Draw field of the Rectangle2 Structure with an other function address.
If it allows to preserve the same Structure, that of the Class, it also underlines a matter:
The function pointer names are less interesting than their values!
To erase this non-problem, just rename the pointers of the Class as following:

```

Structure Rectangle2
  *Method1.i
  *Method2.i
  x1.i
  x2.i
  y1.i
  y2.i
EndStructure

```

In fact, the Interface and the Constructor are in charge to give a sense to these pointers:

- by giving them a name (task of the interface)
- by allocating them the adequate function addresses (task of the constructor)



In spite of this arrangement concerning the function pointer names, it remains more practical to keep an explicit name if hiding methods is not considered (what is the most common situation). That allows to modify a Mother Class without retouching the pointers numbering of the Child Classes.

Inheritance

As for the first inheritance concept implementation, we are going to take advantage that the Structure and Interface instructions have together to be extend thanks to the keyword `Extends`.

So, we shall pass from the Class `Rectangle1` which has a single `Draw()` method to...

Interface	Interface Rect1 Draw() EndInterface
Class	Structure Rectangle1 *Method1.l x1.l x2.l y1.l y2.l EndStructure Procedure Draw_Rectangle(*this.Rectangle1) ... EndProcedure Procedure Init_Mthds_Rect1(*Rect.Rectangle1) *Rect\Method1 = @Draw_Rectangle() EndProcedure
Constructor	Procedure Init_Mbers_Rect1(*Rect.Rectangle1, x1.l, x2.l, y1.l, y2.l) *Rect\x1 = x1 *Rect\x2 = x2 *Rect\y1 = y1 *Rect\y2 = y2 EndProcedure Procedure New_Rect1(x1.l, x2.l, y1.l, y2.l) *Rect = AllocateMemory(SizeOf(Rectangle1)) Init_Mthds_Rect1(*Rect) Init_Mbers_Rect1(*Rect, x1, x2, y1, y2) ProcedureReturn *Rect EndProcedure

...a Class `Rectangle2`, which have two methods: `Draw()` and `Erase()`, by writing:

Interface	Interface Rect2 Extends Rect1 Erase() EndInterface

Class	<pre> Structure Rectangle2 Extends Rectangle1 *Method2.l EndStructure Procedure Erase_Rectangle(*this.Rectangle1) ... EndProcedure Procedure Init_Mthds_Rect2(*Rect.Rectangle2) Init_Mthds_Rect1(*Rect) *Rect \Method2 = @Erase_Rectangle() EndProcedure </pre>
Constructor	<pre> Procedure Init_Mbers_Rect2(*Rect.Rectangle2, x1.l, x2.l, y1.l, y2.l) Init_Mbers_Rect1(*Rect, x1, x2, y1, y2) EndProcedure Procedure New_Rect2(x1.l, x2.l, y1.l, y2.l) *Rect = AllocateMemory(SizeOf(Rectangle2)) Init_Mthds_Rect2(*Rect) Init_Mbers_Rect2(*Rect, x1, x2, y1, y2) ProcedureReturn *Rect EndProcedur </pre>

Carrying out an inheritance consists in extending Interface and Class structure, but also in adapting the method and the attribut initializations

Both procedures `Init_Mthds_Rect2()` and `Init_Mbers_Rect2()` call respectively the initialization of the methods and to the initialization of the attributes of the Class Rectangle1 (`Init_Mthds_Rect1()` and `Init_Mbers_Rect1()`) rather than the Constructor `New_Rect1()`.

Those because, a Child Class object instantiation doesn't need to instantiate a Mother Class object, just to inherit methods and attributes.

On the other side, verification is made that the Child Class benefits immediately of any changes made on the Mother Class (adding a method or a variable).

Is the inheritance currently correct? No, because it does not allow the object of the Child Class (Rectangle2) to use the new `Erase()` method.

The reason take place on that function pointer `*Method2` does not follow the `*Method1` one.

By showing the explicit form of Class Rectangle2 Structure, we have:

```

Structure Rectangle2
*Method1.l
x1.l
x2.l
y1.l
y2.l
*Method2.l
EndStructure

```

instead of having the Structure below, authorizing a correct initialization of the interface:

```

Structure Rectangle2
*Method1.l
*Method2.l
x1.l
x2.l
y1.l
y2.l
EndStructure

```

Remind that a correct interface initialization needs function addresses, which follow each other (➡️⚠️).
 To resolve this problem, just group in a specific structure all the methods!
 The Class Structure needs just to have a pointer on this new structure as the following example shows:

Interface	Interface Rect1 Draw() EndInterface
Class	Structure Rectangle1 *Methods.l x1.l x2.l y1.l y2.l EndStructure Procedure Draw_Rectangle(*this.Rectangle1) ... EndProcedure Structure Methd_Rect1 *Method1.l EndStructure Procedure Init_Mthds_Rect1(*Mthds.Mthds_Rect1) *Mthd_Rect1\Method1 = @Draw_Rectangle() EndProcedure Mthds_Rect1. Mthds_Rect1 Init_Mthds_Rect1(@Mthds_Rect1)
Constructor	Procedure Init_Mbers_Rect1(*Rect.Rectangle1, x1.l, x2.l, y1.l, y2.l) *Rect\x1 = x1 *Rect\x2 = x2 *Rect\y1 = y1 *Rect\y2 = y2 EndProcedure Procedure New_Rect1(x1.l, x2.l, y1.l, y2.l) Shared Mthds_Rect1 *Rect.Rectangle1 = AllocateMemory(SizeOf(Rectangle1)) *Rect \Methods = @Mthds_Rect1 Init_Mbers_Rect1(*Rect, x1, x2, y1, y3) ProcedureReturn *Rect EndProcedur

The Methd_Rect1 structure describes all the function pointers of the Class methods.
 Follows the Methd_Rect1 variable statement and its initialization thanks to `Init_Mthds_Rect1()`.
 This collection groups the complete Class methods description.



The following expression
`Mthds_Rect1.Mthds_Rect1`
`Init_Mthds_Rect1(@Mthds_Rect1)`
can be condensate into
`Init_Mthds_Rect1(@Mthds_Rect1.Mthds_Rect1)`

The Rectangle1 structure, contains now a pointer "`*Methods`", initialized through the constructor by giving to it the Methd_Rect1 variable address.

The inheritance can be now performed correctly, because by extending the Methd_Rect1 Structure in a Methd_Rect2 new one, the addresses of function are going to follow each other:

Interface	Interface Rect2 Extends Rect1 Erase() EndInterface
Class	Structure Rectangle2 Extends Rectangle1 EndStructure Procedure Erase_Rectangle(*this.Rectangle2) ... EndProcedure Structure Methd_Rect2 Extends Methd_Rect1 *Method2.l EndStructure Procedure Init_Mthds_Rect2(*Mthds.Mthds_Rect2) Init_Mthds_Rect1(*Mthds) *Mthd_Rect2\Method2 = @Erase_Rectangle() EndProcedure Mthds_Rect2. Mthds_Rect2 Init_Mthds_Rect2(@Mthds_Rect2)
Constructor	Procedure Init_Mbers_Rect2(*Rect.Rectangle2 , x1.l, x2.l, y1.l, y2.l) Init_Mbers_Rect1(*Rect, x1, x2, y1, y2) EndProcedure Procedure New_Rect2(x1.l, x2.l, y1.l, y2.l) Shared Mthds_Rect2 *Rect.Rectangle2 = AllocateMemory(SizeOf(Rectangle2)) *Rect \Methods = @Mthds_Rect2 Init_Mbers_Rect2(*Rect, x1, x2, y1, y2) ProcedureReturn *Rect EndProcedure

In this example, the Rectangle2 Structure is empty, what is not a matter.
Two reasons in it:

- At first the *Methodes pointer needs to exist only once and this in the Mother Class.
- Then, no supplementary attributes have been added to it.

The fact of having extracted the methods initialization routine outside of the Constructor and having coupled it to the function pointers, which are available in a unique variable, has three advantages:



- *The function pointers, of the Class methods, are initialized once and not at each object instantiation,*
- *The objects have no more than one pointer towards the methods: it is a substantial gain of memory,*
- *All the objects referred towards the same function pointers, which guaranteed an identical behavior for all the objects of the same Class.*

Get() and Set() object methods

By Interface, it is possible to use only methods of an object.

The interface encapsulates completely the object attributes.

To access attributes, either for examine or to modify them, it is necessary to have specific methods given to the user.

The methods allow to examine object attributes are called **Get()** methods.

The methods allow to modify object attributes are called **Set()** methods.

In our example of Class Rectangle1, if we want to examine the value of the attribute var2, we shall create following Get() method:

```
Procedure Get_var2(*this.Rectangle1)
ProcedureReturn *this\var2
EndProcedure
```

Also, to modify the value of the attribute var2, we shall write the following Set() method

```
Procedure Set_var2(*this.Rectangle1, value)
*this\var2 = value
EndProcedure
```

Because Get() and Set() methods exist only to allow the user to modify all or any of the object attributes, they are necessarily present in the Interface.



See the [Appendix](#) of the tutorial for Optimization, to study how the performances of Get() and Set() methods can be optimize during the run.



[Contents](#)

[Next Page](#)



[1-2-3-4-5-6-7-8]

[Top of the page](#)

PureBasic and the Object-Oriented Programming

Synthesis et notation

The implementation of an object appears under the following shape:

- An Interface,
- A Class (concrete / abstract) including the methods definition,
- A Constructor provided with a routine initializing attributes,
- A Destructor.

The following table summarize what is our object in PureBasic.


- The word Class refers to the name of the Class (ex: Methd_Class)
- The word Mother refers to the name of the Mother Class during an inheritance (ex: Methd_ MotherClass)
- The expressions between embraces { } are to be used during an inheritance


Interface	<pre> Interface <Interface> { Extends <MotherInterface> } Method1() [Method2()] [Method3()] ... EndInterface </pre>
Classe	<pre> Structure <Class> { Extends <MotherClass> } *Methods.l [Attribute1] [Attribute2] ... EndStructure Procedure Method1(*this.Class, [arg1]...) ... EndProcedure Procedure Method2(*this.Class, [arg1]...) ... EndProcedure ... Structure <Mthds_Class> { Extends <Mthds_MotherClass> } *Method1.l *Method2.l ... EndStructure Procedure Init_Mthds_Class(*Mthds.Mthds_Class) { Init_Mthds_MotherClass(*Mthds) } *Mthds\Method1 = @Method1() *Mthds\Method2 = @Method2() ... EndProcedure Mthds_Class.Mthds_Class Init_Mthds_Class(@Mthds_Class) </pre>

Constructor	<pre> Procedure Init_Mbers_Classe(*this.Class, [var1]...) { Init_Mbers_MotherClass(*this)} [*this\Attribute1 = var1] ... EndProcedure Procedure New_Class([var1]...) Shared Mthds_Class *this.Class = AllocateMemory(SizeOf(Class)) *this\Methods = @Mthds_Class Init_Mbers_Class(*this, [var1]...) ProcedureReturn *this EndProcedure </pre>
Destructor	<pre> Procedure Free_Class(*this) FreeMemory(*this) EndProcedur </pre>

Here is an example of a code where the inheritance is used:

Ex: [POO_Inheritance.pb](#)

 [Contents](#)

[Next Page](#) 

[[1](#)-[2](#)-[3](#)-[4](#)-[5](#)-**[6](#)**-[7](#)-[8](#)]

[Top of the page](#)

PureBasic and the Object-Oriented Programming

Conclusion

You will have understood that, if it is possible to adopt a Object-oriented Programming in PureBasic, a rigor of writing is required.


But once this task released, the object manipulation is excessively simple.


Now if the Objects languages bring a bigger flexibility in writing codes (according to a direct object concepts using), its organization is articulated on using methods and memory allocations, which leads to some performance loses.

Nevertheless, I hope that this tutorial will have allowed you, at first, to realize the underlying mechanisms in the OOP and to understand the concepts.

Even by keeping a procedural approach, we can now integrate these concepts somehow into codes. You can then benefit from the experience of the oriented-object domain to improve your own way of programming: notably Design Patterns brought adapted answers to a given conception problems.

It is up to you to make a good usage of all, by mixing procedural and object in wished proportions.

 [Contents](#)

[Next Page](#) 

[[1](#)-[2](#)-[3](#)-[4](#)-[5](#)-[6](#)-**[7](#)**-[8](#)]

[Top of the page](#)

PureBasic and the Object-Oriented Programming

Appendix

Optimisations

The paragraphs which follow deal with considerations which it is possible to adopt to improve, during the run of the program, the performances of our Object-oriented approach.

Optimisation: Get() and Set() methods:

By making frequent calls to Get() and Set() methods, it means many function calls and a loss of performance. For those who are in search of performance, there are two possibilities to accelerate the process:

Both consist in coupling a pointer to the object, the second solution bringing a supplementary layer to the first one.

First solution:

The pointer is specified by the Class Structure.

So, for an object Rect of the Class Rectangle1, write:

```
Rect.Rect = New_Rect()
*Rect.Rectangle1 = Rect
```

Access to the var2 attribute as:

```
*Rect\var2
```

It is then possible both to examine and to modify it.
This is the more simple solution to implement.

Second solution:

The first solution asks to work with two different typed elements: Rect and *Rect.

This second solution, suggests grouping these two elements in a StructureUnion block.

```
Structure Rect_
StructureUnion
Mthd.Rect
*Mbers.Rectangle1
EndStructureUnion
EndStructure
```

Creating an object from Class Rectangle1, means declaring the object thanks to this new Structure.
By modifying as a consequence the constructor, it gives:

```
New_Rect(@Rect.Rect_)
```

with,

```
Procedure New_Rect(*Instance.Rect_)
*Rect = AllocateMemory(SizeOf(Rectangle2))
Init_Rect1(*Rect)
Init_Rect2(*Rect)
*Instance\Mthd = *Rect
EndProcedure
```

To access to the `Draw()` method, write:

```
Rect\Mthd\Draw()
```

To access to the `var2` attribute, write:

```
Rect\Mbers\var2
```

This second solution has the advantage to have only a single element that can be deal as an object from which all the attributes are accessible from outside of the class.

It preserves also object-oriented notation, although it presents a supplementary level of fields.

The inconvenience concerns essentially the fact that it is necessary to maintain a new structure within the Class.

I give this exercise to the care of the interested reader 😊

Use of the Linked lists:

Linked lists can be employed to insure the dynamic memory allocation necessary for the object instantiation.

We should then benefit of list characteristics to manage more easily such objects.

We present here two possible implementations at Constructor and Destructor levels.

First possibility:

The first use of the Linked lists consists in transposing the initial studied implementation:

Constructor	<pre> NewList Instance_Class.Class() Procedure Init_Mbers_Class(*this.Class, [var1]...) { Init_Mbers_MotherClass(*this)} [*this\Attribute1 = var1] ... EndProcedure Procedure New_Class([var1]...) Shared Mthds_Class ; *this.Class = AllocateMemory(SizeOf(Class)) AddElement(Instance_Class()) *this.Class = @Instance_Class() *this\Methods = @Mthds_Class Init_Mbers_Class(*this, [var1]...) ProcedureReturn *this EndProcedure </pre>
Destructor	<pre> Procedure Free_Class(*this) ; FreeMemory(*this) ChangeCurrentElement(Instance_Class(), *this) Instance_Class()\Methods= 0 DeleteElement(Instance_Class(),1) EndProcedure </pre>

In comments are the original implementations.

Let us analyze now this transposition.

First of all, we declare the linked lists Instance_Class() of type **Class**, which handles to collect afterward the attributes and the methods of every object.

In the Constructor, the dynamic allocation command AllocateMemory() is replaced by AddElement(). Then, @Instance_Class() retrieve the memory address which is useful for us.



the following lines of the Constructor:
AddElement(Instance_Class())
*this.Class = @Instance_Class()
can be packed into:
*this.Class = AddElement(Instance_Class()) + 8

In the Destructor, FreeMemory() was replaced by DeleteElement(). However to make sure that the right object is deleted, it is necessary to point the right element in the list. ChangeCurrentElement() is used for that purpose.

At last, it shall note the presence of the line: " Instance_Class()\Methods = 0 "

It is in charge to force the zero setting of the methods pointer from the wished object to be deleted.

As reminder at the " **Object Destructor** " part of the tutorial, it is necessary to understand that only the allocated

memory area is freed, but that the variable Object().Interface is not deleted (➡ ⚠).

Thus, it is necessary to take care of the methods and of the attributes purge.

Our example limits the purge to the methods.

We notice that this implementation is heavier to write. Moreover, it will take more place in memory than the initial implementation because in every new instantiation, two pointers (those from the linked list) are added in memory besides the members of the object.

However, the way of using an object so implemented remains unchanged.

For single advantage, this implementation allows to count and to list easily the instantiated objects of the same class.

Just write following code to display all the instantiated objects of the Class:

```
Object.Interface
ForEach Instance_Class()
Object = @Instance_Class()
Object\Draw()
Next
```

Second possibility:

It is possible to see the linked list either as a group of the attributes and the methods of every object, but as a collection of objects.

Write the following implementation to perform it:

Constructeur	<pre>NewList Object.Intance() Procedure Init_Mbers_Class(*this.Class, [var1]...) { Init_Mbers_MotherClass(*this) [*this\Attribute1 = var1] ... EndProcedure Procedure New_Class([var1]...) Shared Mthds_Class *this.Class = AllocateMemory(SizeOf(Class)) *this\Methods = @Mthds_Class Init_Mbers_Class(*this, [var1]...) AddElement(Object()) Object() = *this EndProcedure</pre>
Destructeur	<pre>Procedure Free_Class() FreeMemory(Object()) DeleteElement(Object(),1) EndProcedure</pre>

This second implementation consists with resuming our initial implementation and with adding the objects management through linked list.

A so implemented object will be used in the following way:

```
New_Class(var1, var2...)
Object()\Method1()
```

In several object instanciations, it will be necessary to check that the current element of the list is the wished object.

In the opposite case, it will be necessary to select it, either by the SelectElement() instruction, or by ChangeCurrentElement() instruction.

As for the first solution, the main interest of this technique takes place in the easy enumeration of the objects of the same class, this time with more elegance:

```

ForEach Object()
Object()\Method1()
Next

```

However, objects managed through linked lists require more rigor.

It is not suitable to use directly on objects each of the linked lists functions.

Be writing DeleteElement(Object (), 1) instruction without taking care before about writing FreeMemory(Object ()) instruction, locks the memory deallocation to the end of the program!

Commonly, all the instructions altering a linked list must be avoided (except SwapElements() which is limited into switching existing elements).

It implies the following instructions:

- AddElement,
- ClearList,
- DeleteElement,
- InsertElement.

To use them, build methods that take care of the complete memory management of an object, as show with:

- **New_Classe()** for AddElement(),
- **Free_Classe()** for DeleteElement().

Conclusion

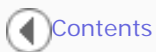
These implementations are interesting mainly if we want to list easily the objects from a same class.

Outside this reason, an object so implemented is rather complex to manipulate and require more rigor in coding than initially.

You will find here two examples of programs using the both exposed implementations:

[POO_LinkedList1.pb](#) using the first implementation

[POO_LinkedList2.pb](#) using the second implementation



[\[1-2-3-4-5-6-7-8\]](#)

[Top of the page](#)