# SEP # :

---

Title: Object Oriented Programming under Scilab
Version: 1.0
Author: Y. Collette
Review:
Commented:
State: Draft
Scilab-Version: 5.0
Vote: "No: discussions and suggestions"
Created: Monday, 22 September 2008

---

## Abstract

This SEP proposes to add some new functionalities to Scilab so as to be able to build classes.

## Rationale

Today, in scilab, an object oriented programming is nearly feasible. To do so, we can use mlist and overloading to emulate object oriented programming. The type of the mlist defines the name of the classe and we can store functions and data in the mlist. For example:

```
class_A_inst = mlist(['class_A','text_to_print','print', ...
                      'set','get'], ...
                     ['none',[],[],[]]);
```

Now, let's define the methods for class_A:

```
function res = print()
printf('%s\n', class_A_inst('text_to_print'));
endfunction
function res = get()
res = class_A_inst('text_to_print');
endfunction
function set(text)
class_A_inst('text_to_print') = text;
endfunction
```

Now, we complete the class:

```
class_A_inst('print') = print;
class_A_inst('set') = set;
class_A_inst('get') = get;
```

Now, if I wanted to create a new class and change the data contained in my new class:

```
class_B_inst = class_A_inst;
class_B_inst.set('new_text');
```

The fist problem: only the data contained in class_A_inst has been changed.

So, the first requirement to have the possibility to build objects is to have a 'this' operator which will refer to the current data structure. Here is the new definition of class_A_inst:

```
function res = print()
printf('%s\n', this('text_to_print'));
endfunction
function res = get()
res = this('text_to_print');
endfunction
function set(text)
this('text_to_print') = text;
endfunction
```

Now, we complete the class:

```
class_A_inst('print') = print;
class_A_inst('set') = set;
class_A_inst('get') = get;
```

Now, if I wanted to create a new class and change the data contained in my new class:

```
class_B_inst = class_A_inst;
class_B_inst.set('new_text');
```

Now, only the content of class_B_inst has been changed because 'this' refers to the current data structure which is now class_B_inst.

An operator 'parent' which refers to the first inherited class should be added too.

What about overloading of operators. Currently, it's not possible to overload everything in scilab. Only the primitives can be overloaded. Let's take an example of the possibility to overload macros. I store a mesh in a mlist and I want to be able to merge 2 meshes via the '+' operator.

```
Mesh_1 = mlist(['mesh',....],[....]);
Mesh_2 = mlist(['mesh',....],[....]);
```

Now, we define the '+' operator for 'mesh' mlist:

```
function res = %mesh_a_mesh(a,b)
// I merge the meshes ...
...
endfunction
```

We use now the operator '+':

```
Mesh_3 = Mesh_1 + Mesh_2;
```

This is currently not possible under scilab because overloading is not applicable to macros.

A problem to solve: what will happen if a '_' appears in the name of the macros ?

It should be 'easy' to solve because, for binary operators, you will always have a odd number of '_' character. And for unary operators ? The operator is always placed at the end of the function name.

Let's talk about inheritage.

To inherit from a class, it's like merging with some rules all the data from a mlist and be able to extend the content of merged class.

We define a 'merge' function which will take all the fields of mlists and create a new mlist.

```
class_A_inst = mlist(['class_A','set','get', ...
                      'common_field','class_A_field'], ...
                      [[],[],1,2]);
class_B_inst = mlist(['class_B','set','get', ...
                      'common_field','class_B_field'], ...
                      [[],[],3,4]);
class_C_inst = mlist(['class_C','set','get', ...
                      'class_C_field'], ...
                      [[],[],5]);
```

For class_A and class_B, 'set' and 'get' allow to view / modify 'common_field'.

For class_C, 'set' and 'get' allow to view / modify 'class_C_field'.

```
function set_AB(variable)
this.common_field = variable;
endfunction
function res = get_AB(variable)
res = this.common_field;
endfunction
function set_C(variable)
this.class_C_field = variable;
endfunction
function get_C(variable)
this.class_C_field = variable;
endfunction
class_C_inst = merge(class_A_inst,class_B_inst);
class_A_inst('get') = get_AB;
class_B_inst('get') = get_AB;
class_A_inst('set') = set_AB;
class_B_inst('set') = set_AB;
class_C_inst('get') = get_C;
class_C_inst('set') = set_C;
```

Now, merge the 3 classes into class_C_inst:

```
class_C_inst = merge(class_C_inst, class_A_inst, class_B_inst);
```

The content of the new class:

```
class_C_inst = mlist(['class_C','set','get', ...
                      'common_field','class_A_field', ...
                      'class_B_field','class_C_field'],[...]);
```

What will be the content of the fields:

If the merge rule is “the first position in the variable list of the merge function dominates the other” then, the result is:

```
class_C_inst.set(6) → class_C_field is updated

class_C_inst.get() → class_C_field is returned

class_C_inst.common_field → 1 is returned (the value contained in
class_A_inst which is placed before class_B_inst in the list of
variables of the merge function).
```

The rule “the first position in the variable list of the merge function dominates the other” is maybe the best because it protects the data of class_C_inst.

A new operator should be added to allow to bypass this behavior. Something like public / private / protected. Let's restrict to only private / public.

'private' attribute will be used to protect the content of a field from being overwritten.

'public' attribute will be used to allow the content of a field to be overwritten.

Here are the new definitions of class_A_inst, class_B_inst and class_C_inst:

```
class_A_inst = mlist(['class_A','set','get', ...
                      'common_field','class_A_field'], ...
                      [[],[],1,2]);
class_B_inst = mlist(['class_B','set','get', ...
                      'common_field','class_B_field'], ...
                      [[],[],3,4]);
class_C_inst = mlist(['class_C','public set','public get', ...
                      'class_C_field'],[[],[],5]);
class_C_inst = merge(class_C_inst, class_A_inst, class_B_inst);
```

Because 'set' and 'get' of class_C_inst has been allowed to be overwritten, we will have the following behavior:

```
class_C_inst.set(6) → the content of 'common_field' is updated

class_C_inst.get() → the content of 'common_field' is returned.
Here, we will have the value 1 if no call to 'set' has been done
(the value of 'common_field' of class_A_inst).
```

By default all the members of the first variable of the merge function are 'private' the members of all the other variables are 'public'. The last class in the 'merge' function override the members of the others except the first one.

Something which will certainly helps to write objects is to allow to define the function stored in a structure directly. For example:

```
class_A_inst = mlist(['class_A','set','get', ...
                      'common_field','class_A_field'], ...
                      [[],[],1,2]);
function y = class_A_inst.get()
y = this.common_field;
endfunction
```

Now, sum-up the requirements for object oriented programming under scilab:

- have a 'this' operator to refer to the content of the current structure;
- have a 'parent' operator to refer to the content of the first inherited structure;
- have the possibility to overload macros;
- define a 'merge' function to deal with inheritage;
- define a 'public' keyword to allow a field of a structure to be overloaded during the merging phase.
- Define directly functions in structures: function y = class.get() ... endfunction

## Example Usage

```
class_geometry = mlist(['geom','public draw'],[]);
class_line = mlist(['line','set','get','draw',A,B],[[],[],0,0]);
class_square = mlist(['square','set','get', ...
                      'draw',side,center],[[],[],0,0]);
function set_line(A,B)
this.A = A;
this.B = B;
endfunction
function [A,B] = get_line()
A = this.A;
B = this.B;
endfunction
function draw_line()
X=0:0.1:10;
Y=A*X+B;
plot(X,Y,'k-');
endfunction
function set_square(side,center)
this.side = side;
this.center = center;
```

```
endfunction
function [side,center] = get_square()
side = this.side;
center = this.center;
endfunction
function draw_square()
xrect(center(1)-side(1)/2,center(2)+side(2)/2,side(1),side(2));
endfunction
class_line.set  = set_line;
class_line.get  = get_line;
class_line.draw = draw_line;
class_square.set  = set_square;
class_square.get  = get_square;
class_square.draw = draw_square;

shapes(1) = merge(class_geometry,class_line);
shapes(2) = merge(class_geometry,class_line);
shapes(3) = merge(class_geometry,class_line);
shapes(4) = merge(class_geometry,class_square);
shapes(5) = merge(class_geometry,class_square);
// class_geometry is here to define common methods for line and
square
for i=1:5
  shapes(i).draw();
end

another solution for class_geometry:

class_geometry = mlist(['geom','draw'],[]);
function draw_geometry()
parent.draw();
endfunction
```

## Changelog

1.0 — First version of the SEP.

## Copyright