Object oriented system with late binding typically use a "vtable"-approach: the first variable in each object is a pointer to a table, which contains the methods as function pointers. This vtable may contain some other informations, too.

Object --> vtable-ptr        -->  instvars
instance var                 methods
instance var                 xt
instance var                 xt
...                          ...

So first, let's declare methods:

: method ( m v -- m' v ) Create  over , swap cell+ swap
       DOES> ( ...) o -- ... ) @ over @ + @ execute ;

During method declaration, the number of methods and instance variables is on the stack (in address units). method creates one method and increments the method number. To execute a method, it takes the object, fetches the vtable pointer, adds the offset, and executes the xt stored there. Each method takes the object it is invoked from as top of stack parameter. The method itself should consume that object.

Now, we also have to declare instance variables

: var ( m v size -- m v' ) Create  over , +
       DOES> ( o -- addr ) @ + ;

Same as above, a word is created with the current offset. Instance variables can have different sizes (cells, floats, doubles, chars), so all we do is take the size and add it to the offset. If your machine has alignment restrictions, put the proper aligned or faligned before the variable, it will adjust the variable offset. That's why it is on the top of stack.

We need a starting point (the empty object) and some syntactic sugar:

Create object 1 cells , 2 cells ,
: class ( class -- class methods vars ) dup 2@ ;

Now, for inheritance, the vtable of the parent object has to be copied, when a new, derived class is declared. This gives all the methods of the parent class, which can be overridden, though.

: end-class ( class methods vars -- )
       Create  here >r , dup , 2 cells ?DO [''] noop , 1 cells +LOOP
cell+ dup cell+ r> rot @ 2 cells /string move ;

The first line creates the vtable, initialized with noops. The second line is the inheritance mechanism, it copies the xts from the parent vtable.

We still have no way to define new methods, let's do that now:

: defines ( xt class -- ) ' >body @ + ! ;

To allocate a new object, we need a word, too:

: new ( class -- o ) here over @ allot swap over ! ;

And sometimes derived classes want to access the method of the parent object. There are two ways to achieve this with this OOF: first, you could use named words, and second, you could look up the vtable of the parent object.

: :: ( class "name" -- ) ' >body @ @ compile, ;

An Example#

Nothing can be more confusing than a good example, so here is one. First let's declare a text object (further called button), that stores text and position:

object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button

Now, implement the two methods, draw and init:

: noname ( o -- ) r
  r@ x @ r@ y @ at-xy r@ text @ r> len @ type ;
button defines draw
: noname ( addr u o -- ) r
  0 r@ x ! 0 r@ y ! r@ len ! r> text ! ;
button defines init

For inheritance, we define a class bold-button, with no new data and no new methods.

button class
end-class bold-button

: bold 27 emit '" [1m" ;
: normal 27 emit '" [0m" ;

: noname bold [ button :: draw ] normal ; bold-button defines draw

And finally, some code to demonstrate how to create objects and apply methods:

button new Constant foo
s" thin foo" foo init
page
foo draw
bold-button new Constant bar
"fat bar" bar init
1 bar y!
bar draw

Larger Example#

Gerry Jackson wrote a larger example, LexGen. This example extends Mini-OOF slightly, but it shows that with these small changes, Mini-OOF can be used for real-world applications.

Mini-OOF for VolksForth#

\ Mini-OOF by Bernd Paysan

CR .( loading Mini-OOF ... )

: METHOD ( m v -- m' v )
  CREATE OVER , SWAP 2+ SWAP
  DOES> ( ... o -- ... )
  @ OVER @ + @ EXECUTE ;

: VAR ( m v size -- )
  CREATE OVER , +
  DOES> ( o -- addr )
  @ + ;

: CLASS ( class -- class methods vars )
  DUP 2@ ;

: END-CLASS ( -- class methods vars )
  CREATE HERE >R , DUP ,
  4 ?DO '[' NOOP , 2 +LOOP
  2+ DUP 2+ R> ROT @ 4 /STRING MOVE ;

: DEFINES ( xt class -- )
  ' >BODY @ + ! ;

: NEW ( class -- o )
  HERE OVER @ ALLOT SWAP OVER ! ;

: :: ( class "name" -- )
  ' >BODY @ + @ , ;

CREATE OBJECT 2 , 4 ,
CR .( Mini-OOF loaded. )

Mini-OOF Example#

\ Mini OOF Example

\needs class INCLUDE" D:MINIOOF.F"

CR .( loading OOF Example )

CR .( creating object class "animal" )

object class
  2 var sound
2 var color
2 var kind
method init
method say
method present
end-class animal

CR .( Implementing Methods )

: m-say ( o -- )
  cr." it says "
  sound @ COUNT TYPE ;
'
  m-say animal defines say

: m-present ( o -- )
  cr." This animal is a "
  DUP color @ COUNT TYPE BL EMIT
  kind @ COUNT TYPE." !" ;
'
  m-present animal defines present

: m-init ( say color kind o -- )
  >R
  R@ SOUND !
  R@ COLOR !
  R> KIND ! ;
'
  m-init animal defines init

CR .( creating animal objects )

animal new constant dog
animal new constant cat
animal new constant eagle

CR .( initializing objects )

: S>A DROP 1- ; ( convert string to address )

S" MAMAL" S>A
S" BLACK" S>A
S" BARK BARK" S>A dog init

S" MAMAL" S>A
S" SILVER" S>A
S" MEOW MEOW" S>A cat init

S" BIRD" S>A
S" BROWN" S>A
S" ARK ARK" S>A eagle init

CR .( now lets the objects speak )

CR
dog present dog say
cat present cat say
eagle present eagle say
.( Finish! )