

Objective Caml version 3.06

```
(* Weitere OO-Konzepte *)

(* Initialisierung *)

(* Beim Kreieren eines neuen Objekt soll stets ein Seiteneffekt ausgefuehrt werden.
   Dazu gibt es ein Konzept: Initializer *)

# class point (x_init, y_init) =
object
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method moveto (a, b) = x <- a; y <- b
  method rmoveto (dx, dy) = x <- x + dx; y <- y + dy
  method distance () = sqrt (float (x*x + y*y))
  method to_string () =
    "(" ^ string_of_int x ^ ", " ^ string_of_int y ^ ")"
end;;

      class point :
int * int ->
object
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method rmoveto : int * int -> unit
  method to_string : unit -> string
end

# class verbose_point coord =
object
  inherit point coord
  initializer
    print_string ("Neuer Punkt an Stelle (" ^ string_of_int x ^ ", " ^ string_of_int y ^ ")\n\n"
)
end;;

      class verbose_point :
int * int ->
object
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method rmoveto : int * int -> unit
  method to_string : unit -> string
end

# let p = new verbose_point (1,2);;
Neuer Punkt an Stelle (1,2)

val p : verbose_point = <obj>

(* Klassen-Variablen *)

(* Manchmal benötigt man eine "globale Buchführung" in einer Klasse,
   z.B. wenn man mitzählen will, wieviele Objekte konstruiert wurden.

   Dies gelingt, weil man in Klassendeklarationen auch die let- und die
   Funktionsschreibweise benutzen darf. *)

# class count_point =
let counter = ref 0
in fun coord ->
object
  inherit point coord
  initializer
    counter := !counter + 1
```

```

method number = print_string ("Es wurden bisher " ^ string_of_int (!counter) ^ " Punkte krei
ert.\n\n")
end;;

class count_point :
int * int ->
object
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method number : unit
  method rmoveto : int * int -> unit
  method to_string : unit -> string
end

# let p0 = new count_point (1,1);;
val p0 : count_point = <obj>

# let p1 = new count_point (1,2);;
val p1 : count_point = <obj>

# let p2 = new count_point (2,2);;
val p2 : count_point = <obj>

# p2#number;;
Es wurden bisher 3 Punkte kreiert.

- : unit = ()

(* Unschön: Methode number gehört zum einzelnen Objekt.
   Gibt es "statische Methoden" wie in Java? *)

(* Mehrfach-Vererbung *)

# class virtual printable () =
object (self)
  method virtual to_string: unit -> string
  method print () = print_string (self#to_string ())
end;;

class virtual printable :
unit ->
object
  method print : unit -> unit
  method virtual to_string : unit -> string
end

# class virtual geometric () =
object
  method virtual area: unit -> int
end;;

class virtual geometric :
unit -> object method virtual area : unit -> int end

# class rectangle ((p1,p2): point * point) =
object
  inherit printable ()
  inherit geometric ()
  val llc = p1
  val urc = p2
  method to_string () =
    "(" ^ p1#to_string () ^ ", " ^ p2#to_string () ^ ")"
  method area () =
    (abs (urc#get_x - llc#get_x) * abs (urc#get_y - llc#get_y))
end;;

class rectangle :
point * point ->
object
  val llc : point
  val urc : point
  method area : unit -> int
  method print : unit -> unit
  method to_string : unit -> string
end

```

```

# let p0 = new point (1,2);;
val p0 : point = <obj>

# let p1 = new point (3,4);;
val p1 : point = <obj>

# let r = new rectangle (p0, p1);;
val r : rectangle = <obj>

# r#print ();;
((1, 2),(3, 4))- : unit = ()

# r#area ();;
- : int = 4

(* Mehrfachvererbung kann zu seltsamen Effekten fuehren, wenn die beiden Oberklassen
die gleichen Instanzvariablen haben. Solche Instanzvariablen gibt's dann in der
Unterklasse doppelt, und manche Methoden arbeiten auf der einen, andere Methoden
auf der anderen Instanzvariablen. *)

# class point (x_init, y_init) =
object
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method moveto (dx, dy) = x <- x + dx; y <- y + dy
  method distance () = sqrt (float (x*x + y*y))
  method to_string () =
    "(" ^ string_of_int x ^ ", " ^ string_of_int y ^ ")"
end;;

      class point :
int * int ->
object
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method to_string : unit -> string
end

# class colored_point (x_init, y_init) c =
object (self)
  inherit point (x_init, y_init) as super
  val color = c
  method get_color = color
  method to_string () =
    super#to_string () ^ " [" ^ self#get_color ^ "]"
end;;

      class colored_point :
int * int ->
string ->
object
  val color : string
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_color : string
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method to_string : unit -> string
end

# class eq_point (x_init, y_init) =
object (self: 'a)
  inherit point (x_init, y_init) as super
  method equal (p: 'a) =
    x = p#get_x && y = p#get_y
end;;

      class eq_point :
int * int ->
object ('a)
  val mutable x : int
  val mutable y : int

```

