

Objective Caml version 3.08.3

```

(* ----- *)
(* Rein funktionale Punkt-Objekte mit einer move-Methode: Der ERGEBNISTYP der *)
(* Methode move stimmt mit dem Typ des Objekts überein (im Gegensatz zu einer *)
(* binären Methode, deren ARGUMENTTYP mit dem Typ des Objekts übereinstimmt). *)
(* ----- *)

# class mv_point (x0, y0) =
object
  val x = x0
  val y = y0
  method x_coord = x
  method y_coord = y
  method move (dx, dy) = new mv_point (x + dx, y + dy)
  method to_string =
    "(" ^ string_of_int x ^ ", " ^ string_of_int y ^ ")"
end;;

class mv_point :
  int * int ->
  object
    val x : int
    val y : int
    method move : int * int -> mv_point
    method to_string : string
    method x_coord : int
    method y_coord : int
  end

# let (p0: mv_point) = new mv_point (1,2);;

val p0 : mv_point = <obj>

# let (p1: mv_point) = p0#move (1,1);;

val p1 : mv_point = <obj>

# p0#to_string;;

- : string = "(1, 2)"

# p1#to_string;;

- : string = "(2, 3)"

(* ----- *)
(* Durch Vererbung soll Farbe zu den mv_points hinzugefügt werden. Der naive *)
(* Ansatz besteht darin, die Methode move unverändert zu übernehmen. Dann geht *)
(* natürlich die Farbe bei move verloren. *)
(* ----- *)

# class mv_colored_point (x0, y0) c =
object (self)
  inherit mv_point (x0, y0) as super
  method color = c
  method to_string =
    super#to_string ^ " [" ^ self#color ^ "]"
end;;

class mv_colored_point :
  int * int ->
  string ->
  object
    val x : int
    val y : int
    method color : string
    method move : int * int -> mv_point
    method to_string : string
    method x_coord : int
    method y_coord : int
  end

# let pc0 = new mv_colored_point (1,2) "blue";;

val pc0 : mv_colored_point = <obj>

# (pc0 :> mv_point);;

```

```

- : mv_point = <obj>

# let pc1 = pc0#move (1,1);;

val pc1 : mv_point = <obj>

(* ----- *)
(* Wenn man die Farbe beibehalten will, muss man move neu definieren. Dann hat *)
(* man aber das gleiche Problem wie bei binären Methoden: Der Typ der Methode *)
(* move wird beim Überschreiben verändert, und das ist nicht erlaubt. *)
(* ----- *)

# class mv_colored_point (x0, y0) c =
object (self)
  inherit mv_point (x0, y0) as super
  method color = c
  method move (dx, dy) =
    new mv_colored_point (x + dx, y + dy) c
  method to_string =
    super#to_string ^ " [" ^ self#color ^ "]"
end;;

Characters 5-247:
..... mv_colored_point (x0, y0) c =
object (self)
  inherit mv_point (x0, y0) as super
  method color = c
  method move (dx, dy) =
    new mv_colored_point (x + dx, y + dy) c
  method to_string =
    super#to_string ^ " [" ^ self#color ^ "]"
end..
The expression "new mv_colored_point" has type
int * int -> string -> mv_colored_point
but is used with type int * int -> string -> mv_point
Type
mv_colored_point =
  < color : string; move : int * int -> mv_point; to_string : string;
    x_coord : int; y_coord : int >
is not compatible with type
mv_point =
  < move : int * int -> mv_point; to_string : string; x_coord : int;
    y_coord : int >
Only the first object type has a method color

(* ----- *)
(* Die Lösung des Problems sieht ähnlich aus wie bei binären Methoden: Anstel- *)
(* le des festen Typs mv_point benutzt man eine Typvariable 'a für den Typ des *)
(* Objekts und definiert die Methode move so, dass sie den Typ int * int -> 'a *)
(* hat. Dafür benötigt man die Schreibweise {<x = ...; y = ...>}, mit der man *)
(* eine Kopie des aktuellen Objekts mit neu besetzten Instanzvariablen erhält. *)
(* ----- *)

# class mv_point (x0, y0) =
object (self: 'a)
  val x = x0
  val y = y0
  method x_coord = x
  method y_coord = y
  method move (dx, dy): 'a = {<x = x + dx; y = y + dy>}
  method to_string =
    "(" ^ string_of_int x ^ ", " ^ string_of_int y ^ ")"
end;;

class mv_point :
int * int ->
object ('a)
  val x : int
  val y : int
  method move : int * int -> 'a
  method to_string : string
  method x_coord : int
  method y_coord : int
end

(* ----- *)
(* Die Typvariable 'a muss man dabei gar nicht explizit angeben. Sie wird vom *)

```

```

(* Typinferenz-Algorithmus gefunden, weil {<x = ...; y = ...>} per Definition *)
(* den gleichen Typ wie das Objekt selbst hat. *)
(* ----- *)

# class mv_point (x0, y0) =
object
  val x = x0
  val y = y0
  method x_coord = x
  method y_coord = y
  method move (dx, dy) = {<x = x + dx; y = y + dy>}
  method to_string =
    "(" ^ string_of_int x ^ ", " ^ string_of_int y ^ ")"
end;;

class mv_point :
  int * int ->
  object ('a)
    val x : int
    val y : int
    method move : int * int -> 'a
    method to_string : string
    method x_coord : int
    method y_coord : int
  end

(* ----- *)
(* Jetzt kann man durch Vererbung Farbe hinzufügen. Dabei muss man move noch *)
(* nicht einmal überschreiben, weil die Schreibweise {<x= ...; y = ...>} schon *)
(* das Gewünschte liefert. *)
(* ----- *)

# class mv_colored_point (x0, y0) c =
object (self)
  inherit mv_point (x0, y0) as super
  method color = c
  method to_string =
    super#to_string ^ " [" ^ self#color ^ "]"
end;;

class mv_colored_point :
  int * int ->
  string ->
  object ('a)
    val x : int
    val y : int
    method color : string
    method move : int * int -> 'a
    method to_string : string
    method x_coord : int
    method y_coord : int
  end

# let pc0 = new mv_colored_point (1,2) "blue";;

val pc0 : mv_colored_point = <obj>

# let pc1 = pc0#move (1,1);;

val pc1 : mv_colored_point = <obj>

(* ----- *)
(* Anders als bei binären Methoden entsteht hier sogar eine Subtyp-Beziehung *)
(* durch die Vererbung. *)
(* ----- *)

# (pc1 :> mv_point);;

- : mv_point = <obj>

#

```