

Objective Caml version 3.08.3

```
(* ----- *)
(* Klassen dürfen Parameter haben. Diese werden mit fun eingeführt ... *)
(* ----- *)

# class movable_point =
fun ((x_init, y_init): float * float) ->
object
  val mutable x = x_init
  val mutable y = y_init
  method x_coord = x
  method y_coord = y
  method move (dx, dy) = (x <- x +. dx; y <- y +. dy)
  method show = print_string ("(" ^ string_of_float x ^ ", " ^ string_of_float y ^ ")")
end;;

class movable_point :
float * float ->
object
  val mutable x : float
  val mutable y : float
  method move : float * float -> unit
  method show : unit
  method x_coord : float
  method y_coord : float
end

(* ----- *)
(* oder (syntaktischer Zucker) mit der traditionellen Parameterschreibweise: *)
(* ----- *)

# class movable_point ((x_init, y_init): float * float) =
object
  val mutable x = x_init
  val mutable y = y_init
  method x_coord = x
  method y_coord = y
  method move (dx, dy) = (x <- x +. dx; y <- y +. dy)
  method show = print_string ("\n(" ^ string_of_float x ^ ", " ^ string_of_float y ^ ")")
end;;

class movable_point :
float * float ->
object
  val mutable x : float
  val mutable y : float
  method move : float * float -> unit
  method show : unit
  method x_coord : float
  method y_coord : float
end

(* ----- *)
(* Um einzelne Objekte zu kreieren, muss man alle Parameter mitgeben. *)
(* ----- *)

# let p0 = new movable_point (0.,0.);;

val p0 : movable_point = <obj>

# let p1 = new movable_point (0.,0.);;

val p1 : movable_point = <obj>

# p0#move (1.,1.);;

- : unit = ()

# p0#show;;

(1., 1.)- : unit = ()

# p1#show;;

(0., 0.)- : unit = ()
```

```

(* ----- *)
(* Ohne Parameter erhält man nur eine Funktion, die noch Parameter erwartet. *)
(* ----- *)

# new movable_point;;

- : float * float -> movable_point = <fun>

(* ----- *)
(* new darf man nicht weglassen, denn Klassen sind keine first class citizens. *)
(* ----- *)

# movable_point;;

Characters 0-13:
movable_point;;
^^^^^^^^^^^^^^
Unbound value movable_point

(* ----- *)
(* Der Klassenname dient auch als Name für den Typ der erzeugten Objekte, also *)
(* ist movable_point eine Abkürzung für den Objekttyp <x_coord: float; ...>. *)
(* ----- *)

# (p0: movable_point);;

- : movable_point = <obj>

# (p0: <x_coord: float; y_coord: float; move: float * float -> unit; show: unit>);;

- : movable_point = <obj>

(* ----- *)
(* Ein Beispiel für Vererbung: Methode show wird überschrieben, Methode color *)
(* wird hinzugefügt. Bei der Implementierung von show wird die Methode show *)
(* der Klasse movable_point benutzt. Dazu muss man dem "geerbten Objekt" einen *)
(* Namen geben, z.B. den Namen super. *)
(* ----- *)

# class movable_colored_point ((x,y,c): float * float * string) =
object
  inherit movable_point (x, y) as super
  method show = (super#show; print_string (" [ ^ c ^ ")))
  method color = c
end;;

class movable_colored_point :
float * float * string ->
object
  val mutable x : float
  val mutable y : float
  method color : string
  method move : float * float -> unit
  method show : unit
  method x_coord : float
  method y_coord : float
end

# let q0 = new movable_colored_point (0.,0., "blue");;

val q0 : movable_colored_point = <obj>

# let q1 = new movable_colored_point (1.,1., "green");;

val q1 : movable_colored_point = <obj>

# q0#move (1.,1.);;

- : unit = ()

# q0#show;;

(1., 1.) [blue]- : unit = ()

# q1#show;;

```

```

(1., 1.) [green]- : unit = ()

(* ----- *)
(* Der Objekttyp movable_colored_point ist ein Subtyp von movable_point, d.h. *)
(* in diesem Fall ist durch Vererbung eine Subtyp-Beziehung entstanden. Diese *)
(* Verbindung zwischen Vererbung und Subtyping ist nicht immer vorhanden, ein *)
(* Gegenbeispiel findet man in der Sitzung "Binäre Methoden". *)
(* ----- *)

# (q0 :> movable_point);;

- : movable_point = <obj>

(* ----- *)
(* Die Subtyp-Beziehung erlaubt uns, vernünftig mit Listen von movable_points *)
(* und movable_colored_points zu arbeiten. *)
(* ----- *)

# let ps: movable_point list = [p0; p1];;

val ps : movable_point list = [<obj>; <obj>]

# let qs: movable_colored_point list = [q0; q1];;

val qs : movable_colored_point list = [<obj>; <obj>]

(* ----- *)
(* List.iter (fun p -> p#show) ist auf Listen von Objekten anwendbar, die min- *)
(* destens eine Methode show besitzen, also sowohl auf ps als auch auf qs. *)
(* ----- *)

# List.iter (fun p -> p#show);;

- : < show : unit; ... > list -> unit = <fun>

# List.iter (fun p -> p#show) ps;;

(1., 1.)
(0., 0.)- : unit = ()

# List.iter (fun p -> p#show) qs;;

(1., 1.) [blue]
(1., 1.) [green]- : unit = ()

(* ----- *)
(* List.iter step ist auf Listen von Objekten anwendbar, die mindestens eine *)
(* Methode move besitzen, also auch auf beide Listen. *)
(* ----- *)

# let step p = p#move (1.,1.);;

val step : < move : float * float -> 'a; .. > -> 'a = <fun>

# List.iter step;;

- : < move : float * float -> unit; ... > list -> unit = <fun>

# List.iter step ps;;

- : unit = ()

# List.iter step qs;;

- : unit = ()

(* ----- *)
(* Mittels coercion kann man movable_points und movable_colored_points in EINE *)
(* Liste aufnehmen und darauf die oben genannten Funktionen anwenden. Dann re- *)
(* agiert jedes Objekt "richtig" auf die Nachrichten, d.h. so wie es in seiner *)
(* Klasse festgelegt ist (Stichwort: dynamic dispatch). Ein movable_point lie- *)
(* fert bei Nachricht show nur seine Koordinaten, ein movable_colored_point *)
(* liefert auch seine Farbe. *)
(* ----- *)

```

```
# let pqs = [p0; p1; q0; q1];;
```

Characters 19-21:

```
let pqs = [p0; p1; q0; q1];;
```

This expression has type `movable_colored_point` but is here used with type `movable_point`

Only the first object type has a method `color`

```
# let pqs = [p0; p1; (q0 :> movable_point); (q1 :> movable_point)];;
```

```
val pqs : movable_point list = [<obj>; <obj>; <obj>; <obj>]
```

```
# List.iter (fun p -> p#show) pqs;;
```

```
(2., 2.)
(1., 1.)
(2., 2.) [blue]
(2., 2.) [green]- : unit = ()
```

```
# List.iter step pqs;;
```

```
- : unit = ()
```

```
# List.iter (fun p -> p#show) pqs;;
```

```
(3., 3.)
(2., 2.)
(3., 3.) [blue]
(3., 3.) [green]- : unit = ()
```

```
(* ----- *)
(* Auch Mehrfach-Vererbung ist möglich: Man kann sich "inherit" so vorstellen, *)
(* dass einfach die Instanzvariablen und Methoden der beerbten Klasse an der *)
(* entsprechenden Stelle einkopiert werden. Also können (mehrfache) inherits *)
(* zur Überschreibung von Methoden führen, d.h. wenn ein Methodename in ver- *)
(* schiedenen beerbten Klassen auftritt, so überlebt nur die letzte dieser Me- *)
(* thoden, weil die vorhergehenden von ihr überschrieben wurden. *)
(* ----- *)
```

```
# class colored_point ((x,y,c): float * float * string) =
object
  method x_coord = 0.
  method y_coord = 0.
  method color = "blue"
end;;
```

```
class colored_point :
  float * float * string ->
  object
    method color : string
    method x_coord : float
    method y_coord : float
  end
```

```
(* ----- *)
(* O'Cam1 gibt eine Warnung aus, wenn Methoden durch Vererbung überschrieben *)
(* wurden. Anhand der Warnung kann man kontrollieren, ob unbeabsichtigte Über- *)
(* schreibungen stattgefunden haben. *)
(* ----- *)
```

```
# class movable_colored_point ((x,y,c): float * float * string) =
object
  inherit colored_point (x,y,c)
  inherit movable_point (x,y) as super
  method show = (super#show; print_string (" [" ^ c ^ "]))
end;;
```

Characters 115-134:

Warning: the following methods are overridden by the inherited class:

```
x_coord y_coord
  inherit movable_point (x,y) as super
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
class movable_colored_point :
  float * float * string ->
  object
```

```

    val mutable x : float
    val mutable y : float
    method color : string
    method move : float * float -> unit
    method show : unit
    method x_coord : float
    method y_coord : float
end

(* ===== *)
(* Polymorphe Klassen *)
(* ===== *)

(* ----- *)
(* Klassen dürfen auch Typparameter haben. Die folgende stack-Klasse hat den *)
(* Elementtyp des stacks als Parameter, d.h. den Typ des formalen Parameters x *)
(* der Methode push. *)
(* ----- *)

# class ['a] stack =
object
  val mutable s = []
  method push (x: 'a) = s <- x :: s
  method pop = s <- List.tl s
  method top = List.hd s
end;;

class ['a] stack :
object
  val mutable s : 'a list
  method pop : unit
  method push : 'a -> unit
  method top : 'a
end

(* ----- *)
(* Das stack-Objekt s hat zunächst den Typ '_a stack, der bei der ersten push- *)
(* Nachricht auf einen konkreten Typ festgelegt wird. *)
(* ----- *)

# let s = new stack;;

val s : '_a stack = <obj>

# s#push 1;;

- : unit = ()

# s#top;;

- : int = 1

# s#pop;;

- : unit = ()

# s#push true;;

Characters 7-11:
s#push true;;
  ^^^^
This expression has type bool but is here used with type int

# s;;

- : int stack = <obj>

(* ----- *)
(* Alternativ kann man den Typ eines neuen stack-Objekts selbst festlegen. *)
(* ----- *)

# let s: int stack = new stack;;

val s : int stack = <obj>

# s#push;;

- : int -> unit = <fun>

```

```
(* ----- *)
(* Die formalen Typparameter einer polymorphen Klasse dürfen nicht weggelassen *)
(* werden (im Gegensatz zu den formalen Typparametern einer polymorphen Funk- *)
(* tion, die man gar nicht hinschreiben kann). *)
(* ----- *)
```

```
# class stack =
object
  val mutable s = []
  method push x = s <- x :: s
  method pop = s <- List.tl s
  method top = List.hd s
end;;
```

Characters 5-132:

```
..... stack =
object
  val mutable s = []
  method push x = s <- x :: s
  method pop = s <- List.tl s
  method top = List.hd s
end..
```

Some type variables are unbound in this type:

```
class stack :
  object
    val mutable s : 'a list
    method pop : unit
    method push : 'a -> unit
    method top : 'a
  end
```

The method push has type 'a -> unit where 'a is unbound

```
(* ----- *)
(* Darüber hinaus müssen die Typparameter an den "passenden" Stellen im Objekt *)
(* auftauchen. Andernfalls können durch die Typinferenz NEUE Typvariablen ent- *)
(* stehen, die nicht gebunden sind. *)
(* ----- *)
```

```
(* ----- *)
(* Die O'Caml-Fehlermeldung ist in diesem Falle unsinnig, sie müsste lauten: *)
(* The method push has type 'b -> unit where 'b is unbound. *)
(* ----- *)
```

```
# class ['a] stack =
object
  val mutable s = []
  method push x = s <- x :: s
  method pop = s <- List.tl s
  method top = List.hd s
end;;
```

Characters 5-137:

```
..... ['a] stack =
object
  val mutable s = []
  method push x = s <- x :: s
  method pop = s <- List.tl s
  method top = List.hd s
end..
```

Some type variables are unbound in this type:

```
class ['a] stack :
  object
    val mutable s : 'b list
    method pop : unit
    method push : 'b -> unit
    method top : 'b
  end
```

The method push has type 'a -> unit where 'a is unbound

```
#
```