# Old School Object Oriented Toolkit ⋆⋆⋆

## Topics

- Objects, Polymorphic variants, Labels, Modules
- Graphics

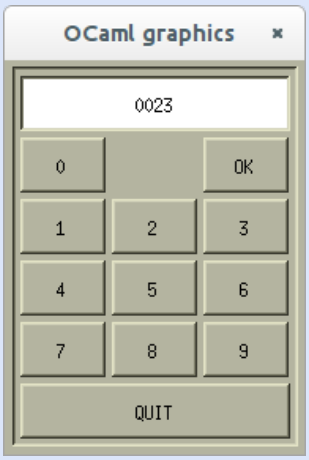
## introduction

In this series of exercises, we will build an old school object oriented GUI toolkit in OCaml.

The goal is to review the basic principles of toolkits, and to learn how to implement them in OCaml mixing functional, imperative and object oriented styles.

After the first exercises, you should be able to run the following program. It defines a higher order function that protects a function application with a PIN code using a visual interface, and applies it to a Hello World program. The arguments are taken in two steps, so the function can be partially applied and the PIN will be checked only before the first application.

```
let check_pin pin =
  let checked = ref false in
  fun f x -> if not !checked then begin
      let main_widget () =
        let field = new field "0000" in
        let quit = object
          inherit button "QUIT"
          method on_click () = exit 2
        end in
        let test_pin = (object
          inherit button "OK"
          method on_click () =
            if field#value = pin then raise Exit
        end)#coerce in
        let digit n = (object
          inherit button (string_of_int n)
          method on_click () =
            field#set_value (field#value ^ string_of_int n)
        end)#coerce in
        let vbox = new vbox [] in
        vbox#add quit ;
        vbox#add (new hbox [ digit 0 ; (new label "")#coerce ; test_pin ]) ;
        vbox#add (new hbox [ digit 1 ; digit 2 ; digit 3 ]) ;
        vbox#add (new hbox [ digit 4 ; digit 5 ; digit 6 ]) ;
        vbox#add (new hbox [ digit 7 ; digit 8 ; digit 9 ]) ;
        vbox#add field ;
        vbox#add (new label "ENTER YOUR PIN") ;
        new frame vbox in
      render main_widget ; checked := true
    end ; f x
```



```
let () =
  let checked_print = check_pin "1234" print_endline in
  checked_print "Hello World" ;
  checked_print "Hello World"
```

## Exercise 1 – Renderer

We give the code of the render in a file `nineties.ml`. It takes the form of a function that runs a main component in a window, and returns once the event handler of a component has raised `Exit`.

The following types are used for communication between the rendered and the components.

```
type point =
  { x : int ; y : int }
type size =
  { w : int ; h : int }
type event =
  | Enter | Leave
  | Grab_focus | Loose_focus
  | Down of point | Move of point | Up of point
  | Key of char
exception Redraw
exception Exit
```

The component has the following type:

```
class type component = object
  method minimum_size : unit -> size
  method natural_size : unit -> size
  method focusable : bool
  method layout : point -> size -> unit
  method draw : unit -> unit
  method react : event -> bool
  method find : point -> component list
  method coerce : component
end
```

**Question 1.1** – Read the code of the renderer (in appendix) to understand how it works, and provide a documented interface to the `Nineties` module.

**Question 1.2** – Document the methods of the component types, explaining the relations between them. Give the invariants that component implementations can assume / must respect for each method.

**Question 1.3** – Explain the goal of `coerce`.

## Exercise 2 – Widgets

**Question 2.1** – Derive from `component` a `widget` that is not focusable and does not rect to events, and fills its layour rectangle with a background color when drawn. Try the renderer on a widget.

**Question 2.2** – Derive from `widget` a `label` that simply draws a given text, centered in its rectangle.

**Question 2.3** – We will now derive `widgets` that react to events. For this, we will implement simple traits that introduce an instance variable and an updater to be called by the `react` method of the

widget.

Following the example below, define two traits `hover_state_updater` and `pressed_state_updater`.

```
type focus_state = Focused | Unfocused
class focus_state_updater = object
  val mutable focus_state = Unfocused
  method update_focus_state = function
    | Grab_focus -> focus_state <- Focused ; true
    | Loose_focus -> focus_state <- Unfocused ; true
    | _ -> false
end
```

**Question 2.4** – Derive from `label`, `hover_state_updater` and `pressed_state_updater` a button that is hilighted when hovered darkened when pressed. The button should be a virtual class that requires its derivates to implement an `on_click` method that is called when the button is clicked.

**Question 2.5** – Derive from `label` or directly from `widget`, and `focus_state_updater` an input `field` class.

The class must define two methods `value` and `set_value` so that the text can be accessed from outside the object.

## Exercise 3 – Containers

**Question 3.1** – Derive from `component` or `widget` a single widget container `frame` that takes a widget and wraps it in a frame.

**Question 3.2** – Write two container classes `vbox` (resp. `hbox`) that take a list of widgets and display them in a column (resp. row).

Equip these container with a polymorphic `add` method that cam append any subtype of component.

## Exercise 4 – Views

**Question 4.1** – Write a virtual parametric class `[ 'a ] list_box` that displays a list of values in a dynamic vertical box. For this, the class requires a `to_component : 'a -> component` method, and provides primitives to edit the list.

## Exercise 5 – Functional builders

**Question 5.1** – Write an interface for the toolkit that makes the component type abstract. For this, you have to provide builder functions (since you cannot call `new` on non public object types). You will also have to provide functional wrappers for subtype specific methods (e.g. `vbox#add`).

**Question 5.2** – Rewrite the example from the introduction using this functional API.

**Question 5.3** – Use labels and polymorphic variants to add options to the various widgets and containers.

## Exercise 6 – To go further

A lot of aspects could be enhanced with this toolkit.

- Widgets such as checkboxes, radio buttons, etc.
- Advanced containers such as scrollers, panners, etc.
- Enhance the distribution of widgets inside boxes. This can be done using glues, weights, maiximal size or a combination of all.
- Switch to a better low level layer than Graphics.
- Enhance the look and feel.

## Appendix: code of the renderer

```
let render (main : unit -> component) =
  let open Graphics in
  open_graph "" ;
  auto_synchronize false ;
  let main = main () in
  let origin = { x = 0 ; y = 0 } in
  let redraw = ref true in
  let exit = ref false in
  let rec loop prev_size prev_hovered prev_focused prev_st =
    let m = main # minimum_size () in
    let size = { w = size_x () ; h = size_y () } in
    if size <> prev_size then redraw := true ;
    if size.w < m.w || size.h < m.h then begin
      resize_window (max m.w size.w) (max m.h size.h) ;
      redraw := true ;
      loop size prev_hovered prev_focused prev_st
    end else begin
      begin try main#layout origin size with Redraw -> redraw := true end ;
      let st = wait_next_event [ Poll ; Key_pressed ] in
      let point = { x = st.mouse_x ; y = st.mouse_y } in
      let hovered = main#find point in
      let focused = ref prev_focused in
      let react o ev =
        try o#react ev with
        | Redraw -> redraw := true ; true
        | Exit -> exit := true ; true in
      List.iter (fun c ->
          if not (List.exists (fun o -> Oo.(id c = id o)) hovered) then
            ignore (react c Leave))
        prev_hovered ;
      List.iter (fun c ->
          if not (List.exists (fun o -> Oo.(id c = id o)) prev_hovered) then
            ignore (react c Enter))
        hovered ;
      let first_to_react ev =
        let rec descend = function
          | o :: os -> if not (react o ev) then descend os
          | [] -> ()
        in descend hovered in
      begin match prev_st, st with
```

```
      | { button = false}, { button = true ; mouse_x ; mouse_y } ->
        first_to_react (Down point)
      | { button = true}, { button = false ; mouse_x ; mouse_y } ->
        first_to_react (Up point) ;
        let rec focus = function
          | c :: cs when c#focusable ->
            begin match prev_focused with
              | None -> ignore (react c Grab_focus)
              | Some prev_c when Oo.id c = Oo.id prev_c -> ()
              | Some prev_c ->
                ignore (react prev_c Loose_focus) ;
                ignore (react c Grab_focus)
            end ;
            focused := Some c
          | _ :: cs -> focus cs
          | [] ->
            begin match prev_focused with
              | None -> ()
              | Some c ->
                ignore (react c Loose_focus) ;
            end ;
            focused := None
        in focus hovered
      | { mouse_x = prev_x ; mouse_y = prev_y},
        { mouse_x ; mouse_y } ->
        if mouse_x <> prev_x || mouse_y <> prev_y then
          first_to_react (Move point)
    end ;
    if st.keypressed then begin
      ignore (wait_next_event [ Key_pressed ]) ;
      match !focused with
      | None -> ()
      | Some o -> ignore (react o (Key st.key))
    end ;
    if !redraw then
      (clear_graph () ; main#draw () ; synchronize ()) ;
    ((* OS yield *) try ignore (Unix.select [] [] [] 0.01) with _ -> ()) ;
    redraw := false ;
    if not !exit then loop size hovered !focused st
  end in
let size = main # natural_size () in
resize_window size.w size.h ;
loop size [] None (wait_next_event [ Poll ; Key_pressed ]) ;
close_graph ()
```

# Old School Object Oriented Toolkit



**Solution to question 1.2**

```
(** The base type of graphical components. *)
class type component = object

  (** The minimal rendering space that the renderer should allow to
      this graphical component for display to be correct. This is the
      unshrinkable space taken by a widget chrome, the minimal size at
      which a picture is readable, etc. It is called at the beginning
      of each rendering pass to check that the window is big
      enough. *)
  method minimum_size : unit -> size

  (** The ideal rendering space that the renderer should allow to this
      graphical component for display to be perfect. The exact text
      size of a text field, the pixel size of a picture, etc. This
      also the size allowed by default when the window is first
      opened. *)
  method natural_size : unit -> size

  (** Tells if the component should be taken into account when
      electing the focused component after a click. *)
  method focusable : bool

  (** At each rendering pass, after the {minimum_size} function is
      called, the main component's layout method is called with the
      origin and window size. These coordinates are the ones to use
      for this rendering pass. The drawing code should not write out
      of this rectangle. Container components should reflect this pass
      order and invariant locally by calling the methods of their
      subcomponents and giving them mutually exclusive
      sub-rectangles. These invariants are however not checked. *)
  method layout : point -> size -> unit

  (** The drawing code. Called at the end of a pass, when {layout} or
      react raised {Redraw}, or on an external event. Should draw
      directly in the graphics window using the rectangle given to
      {layout}. *)
  method draw : unit -> unit

  (** Reacts to an event (return [true]) or ignore it (return [false]).
      Can raise {Redraw} to trigger a display at the end of the pass. *)
  method react : event -> bool

  (** Returns the lists of components under a pointer. The elements
```

```
      should be in order of preference for event dispatching. The
      rendering engine will feed mouse events to the {react} methods of
      these components until one of them returns [true]. This is also
      used for focus selection. When a click happens, the first
      {focusable} component will receive the focus. *)
  method find : point -> component list

  (** A utility method to coerce elaborated widgets and container to
      this base component type. *)
  method coerce : component
end
```

**Solution to question 1.3**

It should return the same object with the base component type. Specific widgets will probably define more public methods, so coercions will be needed and this methods simplifies them.

**Solution to exercise 2**

Some auxiliary code:

```
let padding = 5
let palette =
  Graphics.[| white ;
  rgb 230 230 200 ;
  rgb 210 210 185 ;
  rgb 180 180 160 ;
  rgb 120 120 100 ;
  rgb 50 50 40 ;
  black |]
let raised_border origin size =
  let open Graphics in
  set_color palette.(1) ;
  moveto origin.x origin.y ;
  rlineto 0 size.h ; rlineto size.w 0 ;
  set_color palette.(5) ;
  rlineto 0 (-size.h) ; rlineto (-size.w) 0 ;
  set_color palette.(2) ;
  moveto (origin.x + 1) (origin.y + 1) ;
  rlineto 0 (size.h - 2) ; rlineto (size.w - 2) 0 ;
  set_color palette.(4) ;
  rlineto 0 (-size.h + 2) ; rlineto (-size.w + 2) 0
let lowered_border origin size =
  let open Graphics in
  set_color palette.(5) ;
  moveto origin.x origin.y ;
  rlineto 0 size.h ; rlineto size.w 0 ;
  set_color palette.(1) ;
  rlineto 0 (-size.h) ; rlineto (-size.w) 0 ;
  set_color palette.(4) ;
  moveto (origin.x + 1) (origin.y + 1) ;
  rlineto 0 (size.h - 2) ; rlineto (size.w - 2) 0 ;
  set_color palette.(2) ;
  rlineto 0 (-size.h + 2) ; rlineto (-size.w + 2) 0
```

**Solution to question 2.1**

```
class virtual widget = object (self)
  val mutable origin = { x = 0 ; y = 0 }
  val mutable size = { w = 1 ; h = 1 }
  method layout new_origin new_size =
    origin <- new_origin ;
    size <- new_size
  method find { x ; y }  =
    if x >= origin.x && y >= origin.y
       && x <= origin.x + size.w && y <= origin.y + size.h then
      [ self#coerce ]
    else []
  method draw () =
    let open Graphics in
    set_color palette.(3) ;
    fill_rect origin.x origin.y size.w size.h
  method minimum_size () =
    { w = 1 ; h = 1 }
  method natural_size () =
    self # minimum_size ()
  method focusable =
    false
  method react _ =
    false
  method coerce =
    (self :> component)
end
```

Solution to question 2.2

```
class label text = object (self)
  inherit widget
  method minimum_size () =
    let w, h = Graphics.text_size "..." in
    { w = w + 2 * padding ; h = h + 2 * padding }
  method natural_size () =
    let w, h = Graphics.text_size text in
    { w = w + 2 * padding ; h = h + 2 * padding }
  method draw () =
    let open Graphics in
    let tw, th = Graphics.text_size text in
    moveto
      (origin.x + (size.w - tw) / 2)
      (origin.y + (size.h - th) / 2) ;
    set_color palette.(6) ;
    draw_string text
end
```

Solution to question 2.3

```
type hover_state = Inside | Outside
class hover_state_updater = object
  val mutable hover_state = Outside
  method update_hover_state = function
    | Enter -> hover_state <- Inside ; true
    | Leave -> hover_state <- Outside ; true
```

```
    | _ -> false
end
type pressed_state = Pressed | Released
class pressed_state_updater = object
  val mutable pressed_state = Released
  method update_pressed_state = function
    | Down _ -> pressed_state <- Pressed ; true
    | Leave | Up _ -> pressed_state <- Released ; true
    | _ -> false
end
```

Solution to question 2.4

```
class virtual button text = object (self)
  inherit label text as label
  inherit hover_state_updater
  inherit pressed_state_updater
  method draw () =
    let open Graphics in
    match pressed_state with
    | Released ->
      if hover_state = Inside then begin
        set_color palette.(2) ;
        fill_rect origin.x origin.y size.w size.h
      end ;
      raised_border origin size ;
      label#draw ()
    | Pressed ->
      lowered_border origin size ;
      label#draw ()
  method virtual on_click : unit -> unit
  method react ev =
    if List.mem true
        [ self#update_pressed_state ev ;
          self#update_hover_state ev ;
          match ev with
          | Up _ -> self#on_click () ; true
          | _ -> false ]
    then raise Redraw else false
end
```

Solution to question 2.5

```
class field text = object (self)
  inherit widget
  inherit focus_state_updater
  val mutable text = text
  method value =
    text
  method set_value v =
    text <- v
  method minimum_size () =
    let w, h = Graphics.text_size "...|" in
    { w = w + 2 * padding ; h = h + 2 * padding }
  method natural_size () =
```

```
    let w, h = Graphics.text_size (text ^ "|") in
    { w = w + 2 * padding ; h = h + 2 * padding }
  method draw () =
    let open Graphics in
    set_color palette.(0) ;
    fill_rect origin.x origin.y size.w size.h ;
    lowered_border origin size ;
    let tw, th = Graphics.text_size text in
    moveto
      (origin.x + (size.w - tw) / 2)
      (origin.y + (size.h - th) / 2) ;
    set_color palette.(6) ;
    match focus_state with
    | Focused -> draw_string (text ^ "|")
    | Unfocused -> draw_string text
  method focusable = true
  method react ev =
    if List.mem true
        [ self#update_focus_state ev ;
          match ev with
          | Key '\b' ->
            if String.length text > 1 then
              text <- String.sub text 0 (String.length text - 1) ;
            true
          | Key c ->
            text <- text ^ String.make 1 c ;
            true
          | _ -> false ]
    then raise Redraw else false
end
```

Solution to question 3.1

```
class frame wrapped = object
  inherit widget as background
  method draw () =
    background#draw () ;
    let { x ; y } = origin in
    let { w ; h } = size in
    let origin = { x = x + padding ; y = y + padding } in
    let size = { w = w - 2 * padding ; h = h - 2 * padding } in
    lowered_border origin size ;
    wrapped#draw ()
  method find point =
    wrapped#find point
  method focusable =
    false
  method layout { x ; y } { w ; h } =
    background#layout { x ; y } { w ; h } ;
    wrapped#layout
      { x = x + 2 * padding ; y = y + 2 * padding }
      { w = w - 4 * padding ; h = h - 4 * padding }
  method minimum_size () =
    let { w ; h } = wrapped#minimum_size () in
```

```
    { w = w + 4 * padding ; h = h + 4 * padding }
  method natural_size () =
    let { w ; h } = wrapped#natural_size () in
    { w = w + 4 * padding ; h = h + 4 * padding }
  method react _ =
    false
end
```

**Solution to question 3.2**

```
class vbox components = object (self)
  val mutable components = (components :> component list)
  method add : 'a. (#component as 'a) -> unit = fun c ->
    components <- components @ [ (c :> component) ]
  method draw () =
    List.iter (fun c -> c#draw ()) components
  method find point =
    List.flatten (List.map (fun c -> c#find point) components)
  method focusable =
    false
  method layout { x ; y } { w ; h } =
    let len = List.length components in
    let dh = (h - padding * (len - 1)) / len in
    let rec loop offset = function
      | [] -> ()
      | c :: cs ->
        c#layout
          { x ; y = y + offset }
          { w ; h = dh } ;
        loop (offset + dh + padding) cs
    in loop 0 components
  method minimum_size () =
    let { w ; h } =
      List.fold_left
        (fun acc c ->
           let { w ; h } = c#minimum_size () in
           { w = max w acc.w ; h = h + acc.h + padding})
        { w = 0 ; h = 0 }
        components in
    { w = max 1 w ; h = max 1 (h - padding) }
  method natural_size () =
    let { w ; h } =
      List.fold_left
        (fun acc c ->
           let { w ; h } = c#natural_size () in
           { w = max w acc.w ; h = h + acc.h + padding})
        { w = 0 ; h = 0 }
        components in
    { w = max 1 w ; h = max 1 (h - padding) }
  method react (_ : event) =
    false
  method coerce =
    (self :> component)
end
```

```
class hbox components = object (self)
  val mutable components = (components :> component list)
  method add : 'a. (#component as 'a) -> unit = fun c ->
    components <- components @ [ (c :> component) ]
  method draw () =
    List.iter (fun c -> c#draw ()) components
  method find point =
    List.flatten (List.map (fun c -> c#find point) components)
  method focusable =
    false
  method layout { x ; y } { w ; h } =
    let len = List.length components in
    let dw = (w - padding * (len - 1)) / len in
    let rec loop offset = function
      | [] -> ()
      | c :: cs ->
        c#layout
          { x = x + offset ; y }
          { w = dw ; h } ;
        loop (offset + dw + padding) cs
    in loop 0 components
  method minimum_size () =
    let { w ; h } =
      List.fold_left
        (fun acc c ->
           let { w ; h } = c#minimum_size () in
           { h = max h acc.h ; w = w + acc.w + padding})
        { w = 0 ; h = 0 }
        components in
    { h = max 1 h ; w = max 1 (w - padding) }
  method natural_size () =
    let { w ; h } =
      List.fold_left
        (fun acc c ->
           let { w ; h } = c#natural_size () in
           { h = max h acc.h ; w = w + acc.w + padding})
        { w = 0 ; h = 0 }
        components in
    { h = max 1 h ; w = max 1 (w - padding) }
  method react (_ : event) =
    false
  method coerce =
    (self :> component)
end
```