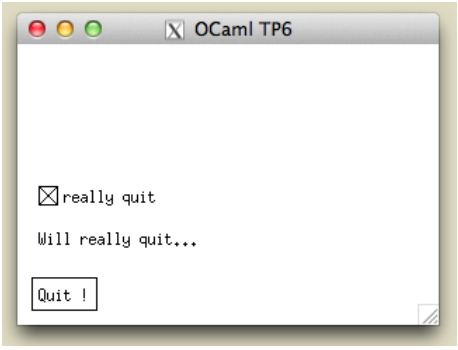
TP 6 : programmation orienté objet en OCaml

Samuel Hornus

30 avril 2014

1 Description du TP

La plupart des applications sur PC présentent une interface graphique : divers éléments sont affichés à l'écran et répondent interactivement aux actions de l'utilisateur. On trouve typiquement des boutons, des menus, des cases à cocher, des barres de défilement, des zones de texte éditable ou pas, etc... Ces élements de base d'une interface graphique son souvent appelés *widgets* (contraction de *window+gadget*). Vus du côté programmeur, ces widgets se prêtent particulièrement bien à une implémentation sous forme de hiérarchie de classes.
Dans ce TP, vous allez ébaucher une telle hiérarchie, pour aboutir à une interface similaire à celle illustrée ci-dessous.



On peut voir : une case à cocher nommée, un champ de texte non éditable mais modifiable par programmation, et un bouton. Un clic sur le bouton quitte l'aplication uniquement si la case est cochée. Un clic sur la case ou sur le bouton modifie le champ de texte pour indiquer à l'utilisateur ce qu'il se passe.

2 Points et rectangles

Nous avons besoin d'un peu de géométrie pour pouvoir dessiner les *widgets* : des points et des rectangles.

2.1 La classe `point`

Dans un fichier `widget.ml`, implémentez la classe `point` :

```
class point :
int -> int -> object
  method x : int
  method y : int
  method string : string
end
```

de telle sorte que :

```
# let p = new point 3 7;;
point created (3, 7)
val p : point = <obj>
# p # x;;
- : int = 3
# p # string;;
- : string = "(3, 7)"
```

2.2 La classe `rect`

Dans le fichier `widget.ml`, implémentez la classe `rect` :

```
class rect :
point -> point -> object
  val pos : point
  method pos : point
  method width : int
  method height : int
  method string : string
end
```

de telle sorte que :

```
# let r = new rect p (new point 20 10);;
point created (20, 10)
rect created [(3, 7); (20, 10)]
val r : rect = <obj>
# r # pos # string;;
- : string = "(3, 7)"
# r # height;;
- : int = 10
# r # string;;
- : string = "[(3, 7); (20, 10)]"
```

La variable membre `pos` représente le coin en bas à gauche du rectangle et le deuxième point passé en argument à la onstruction de l'objet représente la taille du rectangle (largeur, hauteur). Plus tard, vous ajouterez d'autres méthodes à la classe `rect`.

3 Une classe abstraite pour les widgets

Nous passons maintenant à la classe dont tous les widget vont hériter. La variable membre `bounds` est un rectangle qui encode la position et la taille du *widget* dans la fenêtre. La méthode virtuelle `do_click` implémente la ou les actions a effectuer lorsque l'utilisateur clique sur le *widget*. La méthode `draw` sera chargée du dessin du *widget*. Enfin, la méthode `handle_click` prend un argument `p` de type `point` et doit

- vérifier si le point `p` (position du clic) est dans le rectangle `bounds`,
- appeller la méthode `do_click` le cas échéant.

```
class virtual widget :
rect -> object
  val mutable bounds : rect
  method virtual do_click : point -> unit
  method virtual draw : unit
  method handle_click : point -> unit
end
```

3.1 Compléter la class `rect`

Ajoutez la méthode `contains : point -> bool` à la class `rect`.

```
# r # contains p;;
- : bool = false
# r # contains (new point 10 10);;
point created (10, 10)
- : bool = true
```

3.2 La classe `widget`

La variable membre `bounds` et la méthode `handle_click` ne sont pas virtuelles. Il faut donc donner leur définition : on implémentera cette classe comme suit :

```
class virtual widget r =
object (self)
  val mutable bounds = (r : rect)
  method virtual do_click : point -> unit
  method virtual draw : unit
  method handle_click (p : point) =
    (* complétez cette méthode *)
end
```

Ainsi, les classes dérivées n'auront plus qu'à implémenter les méthodes `do_click` et `draw`.

4 Le module `Graphics`

Prenez un dizaine de minutes pour lire la documentation du module `Graphics` : cliquez-moi.

5 Le fichier `main.ml`

Lisez, comprenez et copiez le code ci-dessous dans un fichier `main.ml`

```
(* passe à |true| pour quitter l'application *)
let finish = ref false
(* la liste des widgets présents dans la fenêtre *)
let widgets = ref []
(* raccourci pour créer un point *)
let point x y = new Widget.point x y
(* raccourci pour créer un rectangle *)
let make_rect left bottom width height =
  new Widget.rect (point left bottom)
                  (point width height)
(* ajoute un widget à la liste *)
let add_widget w =
          widgets := (w :> Widget.widget)::!widgets
let create_interface () =
  ()
let draw () =
  ()
let handle_mouse (s : Graphics.status) =
  finish := true
let _ =
  Graphics.open_graph " 300x200+400+20";
  Graphics.set_window_title "OCaml TP6";
  Graphics.set_font "-*-helvetica-medium-r-*-*-24-*";
  create_interface ();
  while not !finish do
    draw();
    let s = Graphics.wait_next_event
              [Graphics.Button_up]
    in handle_mouse s
  done;
  Graphics.close_graph ();
  print_endline "Bye bye"
```

Pour tester, compiler comme suit :
`ocamlc graphics.cma widget.ml main.ml`

6 Un widget pour les boutons

Implémentez, dans le fichier `widget.ml`, la classe `button` qui hérite bien entendu de la classe `widget`. Les termes en noir ci-dessous sont héritées et ne doivent donc pas être (ré)implémenté.

```
class button :
string -> point -> object
  val mutable bounds : rect
  method do_click : point -> unit
  method draw : unit
  method handle_click : point -> unit
  method label : string
end
```

Le 1er paramètre, de type `string` est le *label* du

bouton. Le second paramètre est le point en bas à gauche du bouton. La taille du bouton sera calculée automatiquement en fonction de son *label* (le module `Graphics` vous aidera). La méthode `do_click` se contentera pour l'instant d'afficher un message avec `print_endline "..."`. La méthode `draw` doit dessiner le bouton, comme illustré par le bouton « Quit ! » sur l'image en §1.

7 Tester les boutons

7.1 Instancier

Modifiez la fonction `create_interface` (`main.ml`) de façon à instancier un bouton, et à l'ajouter dans la liste globale des *widgets*.

7.2 Dessiner l'interface graphique

Modifiez la fonction `draw` (`main.ml`) de façon à
- effacer le contenu de la fenêtre.
- invoquer la méthode `draw` de tous les *widgets* contenus dans la liste globale `!widgets`.

7.3 Gérer les clics souris

Modifiez la fonction `handle_mouse` (`main.ml`) de façon à invoquer la méthode `handle_click` de tous les *widgets* contenus dans la liste globale `!widgets` (avec un paramètre correctement construit).

OK, tout (ou presque) est en place maintenant... il ne reste plus qu'à implémenter quelques *widgets* supplémentaires.

8 Case à cocher

```
class checkBox :
string -> bool -> point -> object
  val mutable active_bounds : rect
  val mutable checked : bool
  method checked : bool
end
```

La classe `checkBox` **doit hériter** de la classe `button`, **mais** nous n'avons pas ré-écrit toutes les variables et méthodes héritées dans l'interface de classe ci-dessus.
La classe prend trois arguments pour construire une case à cocher. Le 1er et le 3ème correspondent au 1er et 2nd de la classe `button`. Le second paramètre est un booléen indiquant l'état initial de la case à cocher (cochée ou pas). Cet état devra être stocker dans la variable membre mutable `checked`. La variable membre `active_bounds` sera de type `rect` et décrit la forme (carrée) de la case à cocher proprement dite, sans son label (voir la figure en §1).
La méthode `do_click` doit changer la valeur de la variable `checked` et invoquer la méthode `do_click` de la classe parente `button`.
La méthode `draw` doit dessiner la case de deux façons bien distinctes selon la valeur de la variable membre `checked` pour informer l'utilisateur de l'interface graphique de l'état de la case à cocher.

9 *Text field*

Le dernier *widget* que vous allez implémenter est un champ de texte. La classe `textField` doit hériter directement de la classe `widget` et prend en paramètre un rectangle, comme pour `widget`.
Ci-dessous, nous n'indiquons que les variables et méthodes nouvelles :

```
class textField :
rect -> object
  val mutable text : string
  method setText : string -> unit
end
```

On doit donc pouvoir utiliser la méthode `setText` pour modifier le texte affiché par un *widget* de type `textField`. (N'oubliez pas d'implémenter les méthodes `do_click` et `draw`.)

10 Donner vie à l'aplication

10.1 L'interface

Modifiez la fonction `create_interface` (`main.ml`) de façon à reproduire l'interface illustrée en §1. (Nhésitez pas à apporter une touche personnelle).

10.2 Personnaliser le comportement des boutons

Ajouter la variable et la méthode suivantes à la classe `button` :
- `val mutable callback : button -> unit` est une variable membre qui stocke une fonction de type `button -> unit`
- `method setCallback : (button -> unit) -> unit` qui servira à changer le « callback » du bouton.

Ensuite, modifiez la méthode `do_click` ainsi :

```
method do_click (p : point) = callback
(self :> button)
```

Lorsque cette méthode est invoquée, le type de `self` peut être le type de n'importe quelle classe dérivée de `button` (par exemple `checkBox`, `button`

ou tout autre classe dérivée de `button`). La coercion `self :> button` sert donc à s'assurer que l'on passe bien un argument de type `button` à la fonction `callback`.

10.3 Utiliser le mécanisme des *callbacks*

Dans la fonction `create_interface` (`main.ml`), créez :

• un *callback* (c'est-à-dire une fonction `button -> unit`) qui vérifie si la case à cocher est cochée, et change la valeur de `finish` en `true` le cas échéant. Dans les deux cas (cochée ou pas), la fonction doit modifier le texte affiché par le champ de texte créé en §10.1, pour décrire ce qu'il se passe.
Associer ce *callback* au bouton que l'on a crée en §7.1.

• un *callback* (c'est-à-dire une fonction `button -> unit`) qui vérifie si la case à cocher est cochée ou pas et modifie le texte affiché par le champ de texte créé en §10.1, pour décrire ce qu'il se passe.
Associer ce *callback* à la case à cocher que l'on a crée en §10.1.