18

Extension and object-oriented programming

Think of your favorite graphical user interface (GUI). It probably has various WIDGETS – buttons, checkboxes, textboxes, radio buttons, menus, icons, and so forth. These widgets might undergo various operations – we might want to draw them in a window, click on them, change their location, remove them, highlight them, select from them. Each of these operations seems like a function. We'd organize functions like this:

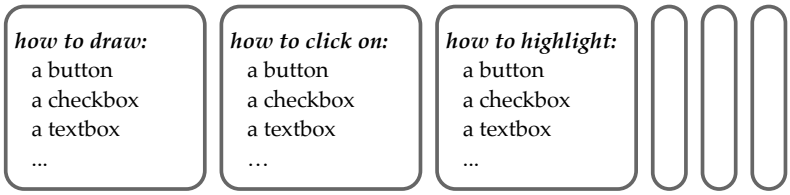

Figure 18.1: Function-oriented organization of widget software

But new widgets are being invented all the time. Every time a new widget type is added, we'd have to change every one of these functions. Instead, we might want to organize the code a different way:

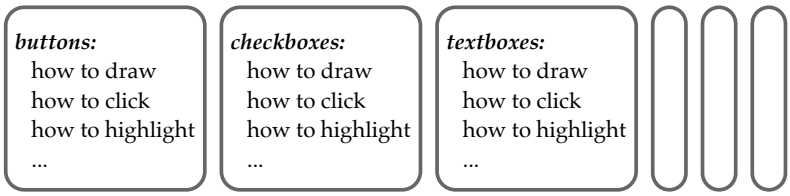

Figure 18.2: Object-oriented organization of widget software

This way, adding a new widget doesn't affect any of the existing ones. The changes are localized, and therefore likely to be much more reliably added. We are carving the software at its joints, following the edict of decomposition.

This latter approach to code organization, organizing by "object"

rather than by function, is referred to as OBJECT-ORIENTED. It's probably no surprise that the rise in popularity of object-oriented programming tracks the development of graphical user interfaces; as seen above, it's a natural fit. In particular, the idea of object-oriented programming was popularized by the Smalltalk programming language and system, which pioneered many of the fundamental ideas of graphical user interfaces that we are now accustomed to – windows, icons, menus, buttons. Smalltalk with its graphical user interface was developed in the early 1970's at Xerox PARC by Alan Kay, Adele Goldberg, Dan Ingalls, and others (Figure 18.3). Steve Jobs, seeing the Smalltalk environment in a 1979 visit to Xerox PARC, immediately imported the ideas into Apple's Lisa and Macintosh computers, thereby disseminating and indeed universalizing the ideas.

In this chapter, we introduce object-oriented programming, a programming paradigm based on organizing functionalities (in the form of methods) together with the data that they operate on, as opposed to the functional paradigm, which organizes functionalities (in the form of functions) separate from the corresponding data.

Figure 18.3: Alan Kay, Adele Goldberg, and Dan Ingalls, developers of the influential Smalltalk language, a pioneering object-oriented language, with an innovative user interface based on graphical widgets and direct manipulation.

18.1 Drawing graphical elements

To motivate such a reorganization, consider a program to draw graphical elements on a window. We'll start by organizing the code in a function-oriented, not object-oriented, style.

Positions in the window can be captured with a `point` data type:

```
# type point = {x : int; y : int} ;;
type point = { x : int; y : int; }
```

We might want data types for the individual kinds of graphical elements – rectangles, circles, squares – each with its own parameters specifying pertinent positions, sizes, and the like:

```
# type rect = {rect_pos : point;
#              rect_width : int; rect_height : int} ;;
type rect = { rect_pos : point; rect_width : int; rect_height :
  int; }

# type circle = {circle_pos : point; circle_radius : int} ;;
type circle = { circle_pos : point; circle_radius : int; }

# type square = {square_pos : point; square_width : int} ;;
type square = { square_pos : point; square_width : int; }
```

We can think of a *scene* as being composed of a set of these *display elements*:

```
# type display_elt =
#   | Rect of rect
```

```
#   | Circle of circle
#   | Square of square ;;
type display_elt = Rect of rect | Circle of circle | Square of
  square

# type scene = display_elt list ;;
type scene = display_elt list
```

In order to make use of these elements to actually draw on a screen, we'll make use of the OCaml `Graphics` module, which you may want to familiarize yourself with before proceeding. (We rename the module `G` for brevity.)

```
# module G = Graphics ;;
module G = Graphics
```

We can write a function to draw a display element of whatever variety by dispatching (matching) based on the variant of the `display_elt` type:[1]

```
# let draw (d : display_elt) : unit =
#     match d with
#     | Rect r ->
#         G.set_color G.black;
#         G.fill_rect (r.rect_pos.x - r.rect_width / 2)
#                     (r.rect_pos.y - r.rect_height / 2)
#                     r.rect_width r.rect_height
#     | Circle c ->
#         G.set_color G.black;
#         G.fill_circle c.circle_pos.x c.circle_pos.y
#                       c.circle_radius
#     | Square s ->
#         G.set_color G.black;
#         G.fill_rect (s.square_pos.x - s.square_width / 2)
#                     (s.square_pos.y - s.square_width / 2)
#                     s.square_width s.square_width ;;
val draw : display_elt -> unit = <fun>
```

[1] All of the subtractions of half the widths and heights is because the Graphics module often draws graphics based on the lower left hand corner position, instead of the center of the graphic that we're using.

and use it to draw an entire scene on a fresh canvas:

```
# let draw_scene (s : scene) : unit =
#   try
#     G.open_graph "";         (* open the canvas *)
#     G.resize_window 200 300; (* erase and resize *)
#     List.iter draw s;        (* draw the elements *)
#     ignore (G.read_key ())   (* wait for a keystroke *)
#   with
#     exn -> (G.close_graph () ; raise exn) ;;
val draw_scene : scene -> unit = <fun>
```

Let's test it on a simple scene of a few rectangles and circles:

```
# let test_scene =
#     [ Rect {rect_pos = {x = 0; y = 20};
```

```
#             rect_width = 15; rect_height = 80};
#       Circle {circle_pos = {x = 40; y = 100};
#               circle_radius = 40};
#       Circle {circle_pos = {x = 40; y = 140};
#               circle_radius = 20};
#       Square {square_pos = {x = 65; y = 160};
#               square_width = 50} ] ;;
val test_scene : display_elt list =
  [Rect {rect_pos = {x = 0; y = 20}; rect_width = 15; rect_height =
  80};
   Circle {circle_pos = {x = 40; y = 100}; circle_radius = 40};
   Circle {circle_pos = {x = 40; y = 140}; circle_radius = 20};
   Square {square_pos = {x = 65; y = 160}; square_width = 50}]

# draw_scene test_scene ;;
- : unit = ()
```

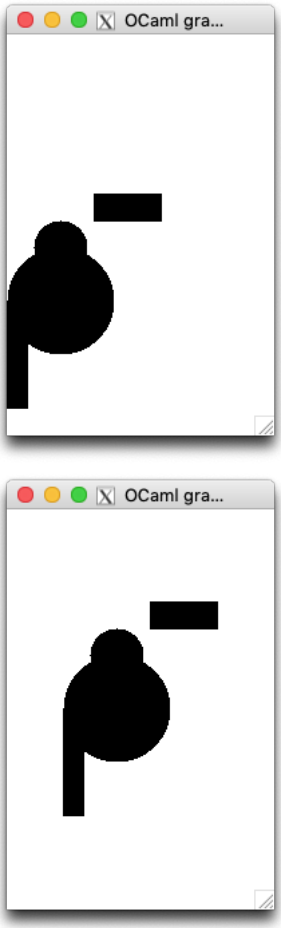

Figure 18.4: (a) A test scene. (b) The test scene translated.

A window pops up with the scene (Figure 18.4(a)).

Sadly, the scene is not centered very well in the canvas. Fortunately, it's easy to add functionality in the functional programming paradigm: just add functions. We can easily add functions to translate a display element or a scene by a given amount in the x and y directions.

```
# let translate (p : point) (d : display_elt) : display_elt =
#   let vec_sum {x = x1; y = y1} {x = x2; y = y2} =
#     {x = x1 + x2; y = y1 + y2} in
#   match d with
#   | Rect r ->
#       Rect {r with rect_pos = vec_sum p r.rect_pos}
#   | Circle c ->
#       Circle {c with circle_pos = vec_sum p c.circle_pos}
#   | Square s ->
#       Square {s with square_pos = vec_sum p s.square_pos} ;;
val translate : point -> display_elt -> display_elt = <fun>

# let translate_scene (p : point) : scene -> scene =
#   List.map (translate p) ;;
val translate_scene : point -> scene -> scene = <fun>
```

Using these, we can translate the scene to center it before drawing:

```
# draw_scene (translate_scene {x = 42; y = 50} test_scene) ;;
- : unit = ()
```

to get the depiction in Figure 18.4(b).

So adding functionality is easy. What about adding new types of data, new display elements? Suppose we want to add a textual display element to place some text in the scene.

```
# type text = {text_pos : point;
#              text_title : string} ;;
type text = { text_pos : point; text_title : string; }
```

We'll have to modify the `display_elt` data type to incorporate text elements:

```
# type display_elt =
#   | Rect of rect
#   | Circle of circle
#   | Square of square
#   | Text of text ;;
type display_elt =
    Rect of rect
  | Circle of circle
  | Square of square
  | Text of text
```

Now the `draw` function complains (unsurprisingly) of an inexhaustive match:

```
# let draw (d : display_elt) : unit =
#     match d with
#     | Rect r ->
#         G.set_color G.black;
#         G.fill_rect (r.rect_pos.x - r.rect_width / 2)
#                     (r.rect_pos.y - r.rect_height / 2)
#                     r.rect_width r.rect_height
#     | Circle c ->
#         G.set_color G.black;
#         G.fill_circle c.circle_pos.x c.circle_pos.y
#                       c.circle_radius
#     | Square s ->
#         G.set_color G.black;
#         G.fill_rect (s.square_pos.x - s.square_width / 2)
#                     (s.square_pos.y - s.square_width / 2)
#                     s.square_width s.square_width ;;
Lines 2-16, characters 0-29:
 2 | match d with
 3 | | Rect r ->
 4 | G.set_color G.black;
 5 | G.fill_rect (r.rect_pos.x - r.rect_width / 2)
 6 | (r.rect_pos.y - r.rect_height / 2)
...
13 | G.set_color G.black;
14 | G.fill_rect (s.square_pos.x - s.square_width / 2)
15 | (s.square_pos.y - s.square_width / 2)
16 | s.square_width s.square_width...
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Text _
val draw : display_elt -> unit = <fun>
```

We'll have to augment it to handle drawing text. Ditto for the `translate` function. In fact, every function that manipulates display elements will have to be changed. If we're going to be adding new types of elements to display, translate, and the like, this will get unwieldy quickly. But there's a better way – objects.

18.2 Objects introduced

What do we care about about display elements? That they can be drawn. That's it. We want to abstract away from all else.

We'll define a data type, an abstraction, `display_elt`, that is a record with a single field called `draw` that stores a drawing function.

```
# type display_elt = {draw : unit -> unit} ;;
type display_elt = { draw : unit -> unit; }
```

Then rectangles, circles, squares, and texts are just ways of building display elements with that drawing functionality.

Take rectangles for example. A rectangle is a `display_elt` whose `draw` function displays a rectangle. We can establish a `rect` function that builds such a display element given its initial parameters – position, width, and height:

```
# let rect (p : point) (w : int) (h : int) : display_elt =
#   { draw = fun () ->
#             G.set_color G.black ;
#             G.fill_rect (p.x - w/2) (p.y - h/2) w h } ;;
val rect : point -> int -> int -> display_elt = <fun>
```

Similarly with circles and squares:

```
# let circle (p : point) (r : int) : display_elt =
#   { draw = fun () ->
#             G.set_color G.black;
#             G.fill_circle p.x p.y r } ;;
val circle : point -> int -> display_elt = <fun>

# let square (p : point) (w : int) : display_elt =
#   { draw = fun () ->
#             G.set_color G.black ;
#             G.fill_rect (p.x - w/2) (p.y - w/2) w w } ;;
val square : point -> int -> display_elt = <fun>
```

Now to draw a display element, we just extract the draw function and call it. The display element data object knows how to draw itself.

```
# let draw (d : display_elt) = d.draw () ;;
val draw : display_elt -> unit = <fun>
```

If we want to add a new display element, a text, say, we just have to provide a way to draw such a thing. No other code (`draw`, `draw_scene`) needs to change.

```
# let text (p : point) (s : string) : display_elt =
#    { draw = (fun () ->
#                let (w, h) = G.text_size s in
#                G.set_color G.black;
#                G.moveto (p.x - w/2) (p.y - h/2);
#                G.draw_string s) } ;;
val text : point -> string -> display_elt = <fun>
```

Of course, we'd probably want display elements to have more functionality than just drawing themselves – for instance, moving them to a new position, querying and changing their color, and much more. Let's start with these.

```
# type display_elt =
#   { draw : unit -> unit;
#     set_pos : point -> unit;
#     get_pos : unit -> point;
#     set_color : G.color -> unit;
#     get_color : unit -> G.color } ;;
type display_elt = {
  draw : unit -> unit;
  set_pos : point -> unit;
  get_pos : unit -> point;
  set_color : G.color -> unit;
  get_color : unit -> G.color;
}
```

Notice that display elements now (apparently) must have mutable state. Their position and color can be modified over time. We'll implement this state by creating appropriate references, called `pos` and `color`, respectively, that are generated upon creation of an object and are specific to it. Here, for instance, is the `circle` function to create a circular display element object:

```
# let circle (p : point) (r : int) : display_elt =
#   let pos = ref p in
#   let color = ref G.black in
#   { draw = (fun () -> G.set_color (!color);
#                       G.fill_circle (!pos).x (!pos).y r);
#     set_pos = (fun p -> pos := p);
#     get_pos = (fun () -> !pos);
#     set_color = (fun c -> color := c);
#     get_color = (fun () -> !color) } ;;
val circle : point -> int -> display_elt = <fun>
```

The scoping is crucial. The definitions of `pos` and `color` are within the scope of the `circle` function. Thus, new references are generated each time `circle` is invoked and are accessible only to the record structure (the object) created by that invocation.[2] Similarly, we'll want a function to create rectangles and text boxes, each with its own state and functionality as specified by the `display_elt` type.

```
# let rect (p : point) (w : int) (h : int) : display_elt =
#   let pos = ref p in
#   let color = ref G.black in
#   { draw = (fun () ->
#             G.set_color (!color);
#             G.fill_rect ((!pos).x - w/2) ((!pos).y - h/2)
#                         w h);
#     set_pos = (fun p -> pos := p);
```

[2] Recall the similar idea of local, otherwise inaccessible, persistent, mutable state first introduced in the `bump` function from Section 15.3, and reproduced here:

```
# let bump =
#   let ctr = ref 0 in
#   fun () ->
#     ctr := !ctr + 1;
#     !ctr ;;
val bump : unit -> int = <fun>
```

```
#     get_pos = (fun () -> !pos);
#     set_color = (fun c -> color := c);
#     get_color = (fun () -> !color) };;
val rect : point -> int -> int -> display_elt = <fun>

# let text (p : point) (s : string) : display_elt =
#   let pos = ref p in
#   let color = ref G.black in
#   { draw = (fun () ->
#             let (w, h) = G.text_size s in
#             G.set_color (!color);
#             G.moveto ((!pos).x - w/2) ((!pos).y - h/2);
#             G.draw_string s);
#     set_pos = (fun p -> pos := p);
#     get_pos = (fun () -> !pos);
#     set_color = (fun c -> color := c);
#     get_color = (fun () -> !color) } ;;
val text : point -> string -> display_elt = <fun>
```

What we've done is to generate a wholesale reorganization of the display element code, organizing it not by functionality (with a `draw` function, a `set_pos` function, and so forth), but instead by variety of "object" bearing that functionality. We've organized the code in an OBJECT-ORIENTED manner.

Think of a table (as in Table 18.1) that describes for each functionality (draw, move, getting and setting color) and each class of object (rectangle, circle, text) the code necessary to carry out that functionality for that class of object. We can organize the code by functionality, packaging the rows into functions; this is the function-oriented paradigm. Alternatively, we can organize the code by class of object, packaging the columns into objects; this is the object-oriented paradigm.

	rectangle	*circle*	*text*

draw	`G.set_color (!color);`
	`G.fill_rect (!pos).x`
	`(!pos).y w h`
		`G.set_color (!color)`
		`G.fill_circle (!pos).x`
		`(!pos).y r`
			`G.set_color (!color);`
			`G.moveto (!pos).x`
			`(!pos).y;`
			`G.draw_string s`

move	`pos := p`	`pos := p`	`pos := p`
set color	`color := c`	`color := c`	`color := c`
get color	`!color`	`!color`	`!color`

Table 18.1: The matrix of functionality (rows) and object classes (columns) for the display elements example. The code can be organized by row – function-oriented – or by column – object-oriented.

Which is the better approach? The edict of decomposition appeals to cutting up software at its joints. Which of row or column constitutes the natural joints will vary from case to case. It is thus a fundamental

design decision as to whether to use a function- or object-oriented structuring of code. If you expect a need to add additional columns with regularity, whereas adding rows will be rare, the object-oriented approach will fare better. Conversely, if new rows, new functionality, will be needed over a relatively static set of classes of data, the function-oriented approach is preferable.

18.3 Object-oriented terminology and syntax

The object-oriented programming paradigm that we've reconstructed here comes with its own set of terminology. First, the data structure that encapsulates the various bits of functionality – here implemented as a simple record structure – is an OBJECT. The various components providing the functionality are its METHODS, and the state variables (like `color` and `pos`) its INSTANCE VARIABLES. The specification of what methods are provided by an object (like `display_elt`) is its CLASS INTERFACE, and the creation of an object is specified by its CLASS (like `circle` or `text`).

We create an object by INSTANTIATING the class, in this example, the `circle` class,

```
# let circle1 = circle {x = 100; y = 100} 50 ;;
val circle1 : display_elt =
  {draw = <fun>; set_pos = <fun>; get_pos = <fun>; set_color =
  <fun>;
   get_color = <fun>}
```

which satisfies the `display_elt` class interface.

When we make use of a method, for instance, the `set_pos` method,

```
# circle1.set_pos {x = 125; y = 125} ;;
- : unit = ()
```

we are said to INVOKE the method

It should be clear that the object-oriented programming paradigm can be carried out in any programming language with the abstractions that we've relied on here, basically, first-class functions, lexical scoping, and mutable state. But, as with other programming paradigms we've looked at, providing some syntactic sugar in support of the paradigm can be quite useful. OCaml does just that. Indeed, the "O" in "OCaml" indicates that the language was developed as an extension to the Caml language that added syntactic support for object-oriented programming.

The object-oriented syntax extensions in OCaml are summarized in Table 18.2.

The display element example can thus be stated in colloquial OCaml as follows. We start with the `display_elt` class interface:

Concept	*Syntax*
Class interfaces	`class type` ⟨*interfacename*⟩ `= ...`
Class definition	`class` ⟨*classname*⟩ ⟨*args*⟩ `= ...`
Object definition	`object ... end`
Instance variables	`val (mutable)` ⟨*varname*⟩ `= ...`
Methods	`method` ⟨*methodname*⟩ ⟨*args*⟩ `= ...`
Instance variable update	`... <- ...`
Instantiating classes	`new` ⟨*classname*⟩ ⟨*args*⟩
Invoking methods	⟨*object*⟩#⟨*methodname*⟩ ⟨*args*⟩

Table 18.2: Syntactic extensions in OCaml supporting object-oriented programming.

```
# class type display_elt =
#   object
#     method draw : unit
#     method set_pos : point -> unit
#     method get_pos : point
#     method set_color : G.color -> unit
#     method get_color : G.color
#   end ;;
class type display_elt =
  object
    method draw : unit
    method get_color : G.color
    method get_pos : point
    method set_color : G.color -> unit
    method set_pos : point -> unit
  end
```

and define some classes that satisfy the interface:

```
# class circle (p : point) (r : int) : display_elt =
#   object
#     val mutable pos = p
#     val mutable color = G.black
#     method draw = G.set_color color;
#                   G.fill_circle pos.x pos.y r
#     method set_pos p = pos <- p
#     method get_pos = pos
#     method set_color c = color <- c
#     method get_color = color
#   end ;;
class circle : point -> int -> display_elt

# class rect (p : point) (w : int) (h : int) : display_elt =
#   object
#     val mutable pos = p
#     val mutable color = G.black
#     method draw = G.set_color color;
#                   G.fill_rect (pos.x - w/2) (pos.y - h/2)
#                   w h
#     method set_pos p = pos <- p
```

```
#     method get_pos = pos
#     method set_color c = color <- c
#     method get_color = color
#   end ;;
class rect : point -> int -> int -> display_elt
```

Now we can use these to create and draw some display elements. We create a new circle,

```
# let _ = G.open_graph "";
#         G.clear_graph ;;
- : unit -> unit = <fun>
# let b = new circle {x = 100; y = 100} 40 ;;
val b : circle = <obj>
```

but nothing appears yet until we draw the element.

```
# let _ = b#draw ;;
- : unit = ()
```

(Notice that invoking the method doesn't require the application to a `unit`. In the object-oriented syntax, method invocation with no arguments can be implicit in this way.) The circle now appears, as in Figure 18.5(a).

We can erase the object by setting its color to white and redrawing it (Figure 18.5(b)).

```
# let _ = b#set_color G.white;
#         b#draw ;;
- : unit = ()
```

We move it to a new position and change its color (Figure 18.5(c)).

```
# let _ = b#set_pos {x = 150; y = 150};
#         b#set_color G.red;
#         b#draw ;;
- : unit = ()
```

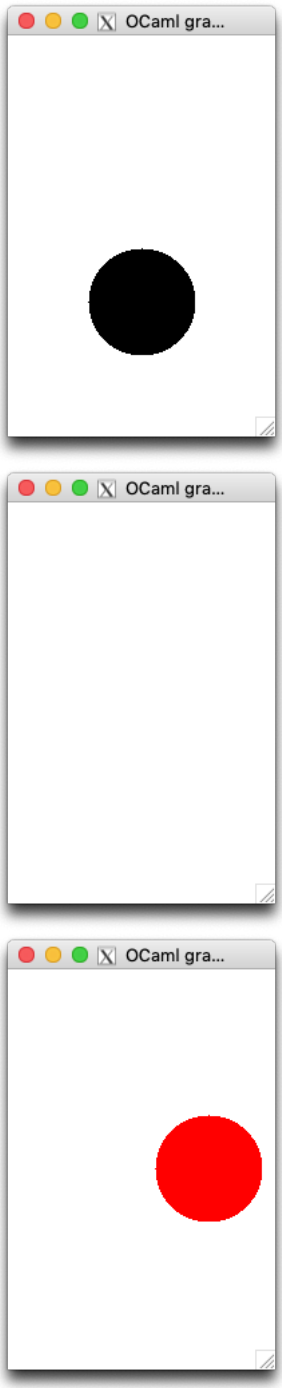

Figure 18.5: A circle appears (a) and disappears (b). It moves and reappears with a changed color (c).

18.4 Inheritance

The code we've developed so far violates the edict of irredundancy. The implementations of the `circle` and `rect` classes, for instance, are almost identical, differing only in the arguments of the class and the details of the `draw` method.

To capture the commonality, the object-oriented paradigm allows for definition of a class expressing the common aspects, from which both of the classes can INHERIT their behaviors. We refer to the class (or class type) that is being inherited from as the SUPERCLASS and the inheriting class as the SUBCLASS.

We'll define a `shape` superclass that can handle the position and color aspects of the more specific classes. Its class type is given by

```
# class type shape_elt =
#   object
#     method set_pos : point -> unit
#     method get_pos : point
#     method set_color : G.color -> unit
#     method get_color : G.color
#   end ;;
class type shape_elt =
  object
    method get_color : G.color
    method get_pos : point
    method set_color : G.color -> unit
    method set_pos : point -> unit
  end
```

and a simple implementation of the class is

```
# class shape (p : point) : shape_elt =
#   object
#     val mutable pos = p
#     val mutable color = G.black
#     method set_pos p = pos <- p
#     method get_pos = pos
#     method set_color c = color <- c
#     method get_color = color
#   end ;;
class shape : point -> shape_elt
```

Notice that the new `shape_elt` signature provides access to the four methods, but not directly to the instance variables used to implement those methods. The only access to those instance variables will be through the methods, an instance of the edict of compartmentalization that seems appropriate.

The `display_elt` class type can inherit the methods from `shape_elt`, adding just the additional `draw` method.

```
# class type display_elt =
#   object
#     inherit shape_elt
#     method draw : unit
#   end ;;
class type display_elt =
  object
    method draw : unit
    method get_color : G.color
    method get_pos : point
    method set_color : G.color -> unit
    method set_pos : point -> unit
  end
```

The `inherit` specification works as if the contents of the inherited superclass type were simply copied into the subclass type at that location in the code.

The `rect` and `circle` subclasses can inherit much of their behavior from the `shape` superclass, just adding their own `draw` methods. However, without the ability to refer directly to the instance variables, the `draw` method will need to call its own methods for getting and setting the position and color. We can add a variable to name the object itself, by adding a parenthesized name after the `object` keyword. Although any name can be used, by convention, we use `this` or `self`. We can then invoke the methods from the `shape` superclass with, for instance, `this#get_color`.

```
# class rect (p : point) (w : int) (h : int) : display_elt =
#   object (this)
#     inherit shape p
#     method draw =
#       G.set_color this#get_color;
#       G.fill_rect (this#get_pos.x - w/2)
#                   (this#get_pos.y - h/2)
#                   w h
#   end ;;
class rect : point -> int -> int -> display_elt

# class circle (p : point) (r : int) : display_elt =
#   object (this)
#     inherit shape p
#     method draw =
#       G.set_color this#get_color;
#       G.fill_circle this#get_pos.x this#get_pos.y r
#   end ;;
class circle : point -> int -> display_elt
```

Notice how the inherited `shape` class is provided the position argument `p` so its instance variables and methods can be set up properly.

Using inheritance, a `square` class can be implemented with a single inheritance from the `rect` class, merely by specifying that the width and height of the inherited rectangle are the same:

```
# class square (p : point) (w : int) : display_elt =
#   object
#     inherit rect p w w
#   end ;;
class square : point -> int -> display_elt
```

Exercise 180

Define a class `text : point -> string -> display_elt` for placing a string of text at a given point position on the canvas. (You'll need the `Graphics.draw_string` function for this.)

18.4.1 Overriding

Inheritance in OCaml allows for subclasses to override the methods in superclasses. For instance, we can implement a class of bordered

rectangles (rather than the filled rectangles of the `rect` class) simply by overriding the `draw` method:

```
# class border_rect (p : point)
#                   (w : int) (h : int)
#                 : display_elt =
#   object (this)
#     inherit rect p w h as super
#     method! draw = G.set_color this#get_color;
#                    G.fill_rect (this#get_pos.x - w/2 - 2)
#                                (this#get_pos.y - h/2 - 2)
#                                (w+4) (h+4) ;
#                    let c = this#get_color in
#                    this#set_color G.white ;
#                    super#draw ;
#                    this#set_color c
#   end ;;
class border_rect : point -> int -> int -> display_elt
```

Here, we've introduced the overriding `draw` method with `method!`, where the exclamation mark diacritic explicitly marks the method as overriding the superclass's `draw` method. Without that, OCaml will provide a helpful warning to the programmer in case the overriding was unintentional.

When a subclass overrides the method of a superclass, the subclass may still want access to the superclass's version of the method. That's the case here, where the subclass's `draw` method needs to call the superclass's. In the presence of overriding, then, it becomes important to have a name for the superclass object so as to be able to call its methods. The inherited superclass can be given a name for this purpose by the `as` construct used above in the `inherit` specification. The variable following the `as` – conventionally `super` though any variable can be used – then names the superclass providing access to its version of any overridden methods.

18.5 Subtyping

Back in Section 18.1, we defined a scene as a set of drawable elements, so as to be able to iterate over a scene to draw each element. We can obtain that ability by defining a new function that draws a list of `display_elt` objects:

```
# let draw_list (d : display_elt list) : unit =
#   List.iter (fun x -> x#draw) d ;;
val draw_list : display_elt list -> unit = <fun>
```

We've put together a small scene (Figure 18.6), evocatively called `scene`, to test the process.[3]

[3] The type of the scene is displayed not, as one might expect, as `display_elt list` but as `border_rect` list. OCaml uses class names, not class type names, to serve the purpose of reporting typing information for objects. The elements of `scene` are instances of various classes (all consistent with class type `display_elt`). OCaml selects the first element of the list, which happens to be a `border_rect` instance, to serve as the printable name of the type. This quirk of OCaml reveals that the grafting of the "O" part of the language isn't always seamless.

Figure 18.6: A test scene.

```
let scene =
  (* generate some graphical objects *)
  let box = new border_rect {x = 100; y = 110} 180 210 in
  let l1 = new rect {x = 70; y = 60} 20 80 in
  let l2 = new rect {x = 135; y = 100} 20 160 in
  let b = new circle {x = 100; y = 100} 40 in
  let bu = new circle {x = 100; y = 140} 20 in
  let h = new rect {x = 150; y = 170} 50 20 in
  let t = new text {x = 100; y = 200} "The CS51 camel" in
  (* bundle them together *)
  let scene = [box; l1; l2; b; bu; h; t] in
  (* change their color and translate them *)
  List.iter (fun x -> x#set_color 0x994c00) scene;
  List.iter (fun o -> let {x; y} = o#get_pos in
                      o#set_pos {x = x + 50; y = y + 40})
            scene;
  (* update the surround color *)
  box#set_color G.blue;
  scene ;;
```

```
# scene ;;
- : border_rect list = [<obj>; <obj>; <obj>; <obj>; <obj>; <obj>;
  <obj>]
```

We can draw this scene in a fresh window using `draw_list`.

```
# let test scene =
#   try
#     G.open_graph "";
#     G.resize_window 300 300;
#     G.clear_graph ();
#     draw_list scene;
#     ignore (G.read_key ())
#   with
#     exn -> (G.close_graph (); raise exn) ;;
val test : display_elt list -> unit = <fun>

# test scene ;;
- : unit = ()
```

We defined `draw_list` to operate on `display_elt` lists. But there's no reason to be so specific. It ought to be the case that any object with a `draw` method should be able to participate in a scene. We can define a new class type of drawable elements

```
# class type drawable =
#   object
#     method draw : unit
```

```
#   end ;;
class type drawable = object method draw : unit end
```

and redefine `draw_list` accordingly:

```
# let draw_list (d : drawable list) : unit =
#   List.iter (fun x -> x#draw) d ;;
val draw_list : drawable list -> unit = <fun>
```

We've defined `drawable` as a SUPERTYPE of `display_elt`. It's a supertype because anything that can be done with a drawable can be done with a `display_elt`, but also potentially with other classes as well (namely, any that have a `draw` method). The idea is that an object with a "wider" interface (a subtype, like `display_elt`) can be used where an object with a "narrower" interface (a supertype, like `drawable`) is needed.

There is a family resemblance in this idea to polymorphism. Any function with a more polymorphic type (like `'a -> 'a list`, say) can be used where an object with a less polymorphic type (like `int -> int list`) is needed.

Exercise 181

Test out this polymorphism subtyping behavior in OCaml by defining two functions `mono : int -> int list` and `poly : 'a -> 'a list`, along with a function `need : (int -> int list) -> int list`. Then apply `need` to both `mono` and `poly`, thereby showing that `need` works with an argument of its required type (`int -> int list`) and also a subtype thereof (`'a -> 'a list`).

Anything that's a `display_elt` or inherits from `display_elt` or satisfies the `display_elt` interface will have at least the functionality of a `drawable`. So they are subtypes of `drawable`.[4]

[4] There would seem to be a correlation between subclasses and subtypes. Of course, not all subtypes are subclasses; they may not be related by inheritance. But in a subclass, you have all the functionality of the superclass, plus you can add some more. So are subclasses always subtypes?

No. For instance, in the subclass, you could redefine a method to have a more restrictive signature. In that case, the subclass would not be a subtype; it would have a narrower interface (at least for that method), not a wider one.

The advantage of subtyping – allowing functions with a wider interface to be used where one with a narrower interface is called for – is just the advantage of polymorphism. It allows reuse of functionality, which redounds to the benefit of the edict of irredundancy. Rather than reimplement functions for the different interface "widths", we reuse them instead. We'll see that OCaml allows this kind of reuse, though with a little less automaticity than the reuse from polymorphism.

It ought to be the case, for instance, that, as `display_elt` is a subtype of `drawable`, our revision of `draw_scene` to apply to `drawable` objects ought to allow scenes composed of `display_elt` objects. Let's try it.

```
# let test scene =
#   try
#     G.open_graph "";
#     G.resize_window 300 300;
#     G.clear_graph ();
#     draw_list scene;
```

```
#     ignore (G.read_key ())
#   with
#     exn -> (G.close_graph (); raise exn) ;;
val test : drawable list -> unit = <fun>
```

The type of `test` shows that it now takes a `drawable list` argument. We apply it to our scene.

```
# test scene ;;
Line 1, characters 5-10:
1 | test scene ;;
          ^^^^^
Error: This expression has type border_rect list
       but an expression was expected of type drawable list
       Type
         border_rect =
           < draw : unit; get_color : G.color; get_pos : point;
             set_color : G.color -> unit; set_pos : point -> unit >
       is not compatible with type drawable = < draw : unit >
       The second object type has no method get_color
```

But the `draw_list` call no longer works. We've tripped over a limitation in OCaml's type inference. A subtype ought to be allowed where a supertype is needed, as it is in the case of polymorphic subtypes of less polymorphic supertypes. But in the case of class subtyping, OCaml is not able to perform the necessary type inference to view the subtype as the supertype and use it accordingly. We have to give the type inference system a hint.

We want the call to draw_list to view scene not as its `display_elt list` subtype but rather as the `drawable list` supertype. We use the `:>` operator to specify that view. The expression `scene :> drawable list` specifies `scene` viewed as a `drawable list`.

```
# let test scene =
#   try
#     G.open_graph "";
#     G.resize_window 300 300;
#     G.clear_graph ();
#     draw_list (scene :> drawable list);
#     ignore (G.read_key ())
#   with
#     exn -> (G.close_graph (); raise exn) ;;
val test : #drawable list -> unit = <fun>

# test scene ;;
- : unit = ()
```

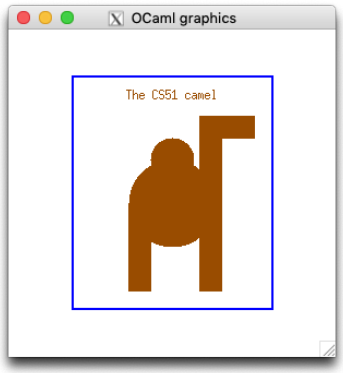

Figure 18.7: The rendered test scene.

Voila! The scene (Figure 18.7) appears. A little advice to the type inference mechanism has resolved the problem.

18.6 Problem section: Object-oriented counters

Here is a class type and class definition for "counter" objects. Each object maintains an integer state that can be "bumped" by adding an integer. The interface guarantees that only the two methods are revealed.

```
class type counter_interface =
  object
    method bump : int -> unit
    method get_state : int
  end ;;

class counter : counter_interface =
  object
    val mutable state = 0
    method bump n = state <- state + n
    method get_state = state
  end ;;
```

Problem 182

Write a class definition for a class `loud_counter` obeying the same interface that works identically, except that it also prints the resulting state of the counter each time the counter is bumped.

Problem 183

Write a class type definition for an interface `reset_counter_interface`, which is just like `counter_interface` except that it has an additional method of no arguments intended to reset the state back to zero.

Problem 184

Write a class definition for a class `loud_reset_counter` satisfying the `reset_counter_interface` that implements a counter that both allows for resetting and is "loud" (printing the state whenever a bump or reset occurs).

18.7 Supplementary material

- Lab 16: Object-oriented programming
- Lab 17: Objects and classes
- Problem set A.8: Force-directed graph drawing
- Problem set A.9: Simulating an infectious process