

```
(*
```

CS51 Lab 17
Objects and Classes

Objective:

This lab provides practice with object-oriented programming: the creation of classes, interfaces, inheritance, subtyping, and dynamic dispatch. *)

```
(*=====
```

Part 1: Flatland

Suppose we want to model a two-dimensional world called Flatland populated by creatures of geometric shapes (<https://url.cs51.io/flatland>). In particular, we want:

- o All shapes to have some notion of **area**, so that when we meet any new shape, we can easily calculate the shape's area.
- o To know the shape's **location** in Flatland space.
- o To support many different kinds of shapes.

How can we solve this problem?

There are several approaches. As a first attempt we might use algebraic data types. (Later, we'll see this isn't quite ideal.)

First, we'll define a point to store (x, y) coordinates. *)

```
type point = float * float ;;
```

(* Now we'll define `'shape_adt'`, an algebraic data type for shapes, with the ability to represent some shapes: a `'Square'`, a `'Rect'`, and a `'Circle'`. Each shape will have some aspects that together specify its location and size, as follows:

Square -- a single point for the location of its lower-left corner and an edge length.

Rect -- a single point for the location of its lower-left corner, a width, and a height.

Circle -- a single point for the location of its center and a radius.

```
*)
```

```
type shape_adt =
  | Square of point * float
  | Rect of point * float * float
  | Circle of point * float ;;
```

(*.....
Exercise 1A: Given the definitions above, write a function `'area_adt'` that accepts a `'shape_adt'` and returns a `'float'` representing the area of the shape. (You can review the area calculations for the various shapes in Section B.4 of the textbook.)
.....*)

```
let area_adt (s : shape_adt) : float =
  failwith "area_adt not implemented" ;;
```

(*.....
Exercise 1B: Write a function `'list_area_adt'` that, given a list of elements of type `'shape_adt'`, returns a list of areas corresponding to

each shape.

.....*)

```
let list_area_adt (lst : shape_adt list) : float list =
  failwith "list_area_adt not implemented" ;;
```

(*=====

Part 2: Interfaces, Classes, Objects

Why is implementing shapes as an ADT not ideal?

Suppose you travel to Flatland and meet a new shape, 'Triangle'. What must change above to support the new shape?

The type definition of shape needs to change to include 'Triangle':

```
type shape_adt =
  | ...
  | Triangle of point * point * point
```

and the area function (and more generally, any function that used a match statement on 'shape_adt') would need to change to include the 'Triangle' case:

```
let area (s : shape_adt) : float =
  match s with
  | ...
  | Triangle ... -> ...
```

You've seen this problem before in the previous lab.

Thus, extending our world of shapes tends to break a lot of different parts of the code before it starts to work again. In real production code, we may not always have access to all elements (we may not have access to the type definition, for example, or the area function), so adding new functionality in this manner may be difficult or impossible.

As you might imagine, declaring all possible shapes up front is an equally poor idea.

Using algebraic data types gives us a **closed** definition for all possible types in this world; this means that we must know all possible variants at the time of type definition.

We can resolve this difficulty with object-oriented programming.

Below, we've created a class type (or interface). Interfaces define a new type and define methods for us to interact with this new type.

Once we have defined this class type, we can create new shapes by defining classes that implement the shape interface. *)

```
class type shape =
  object
    (* The area of this shape *)
    method area : float

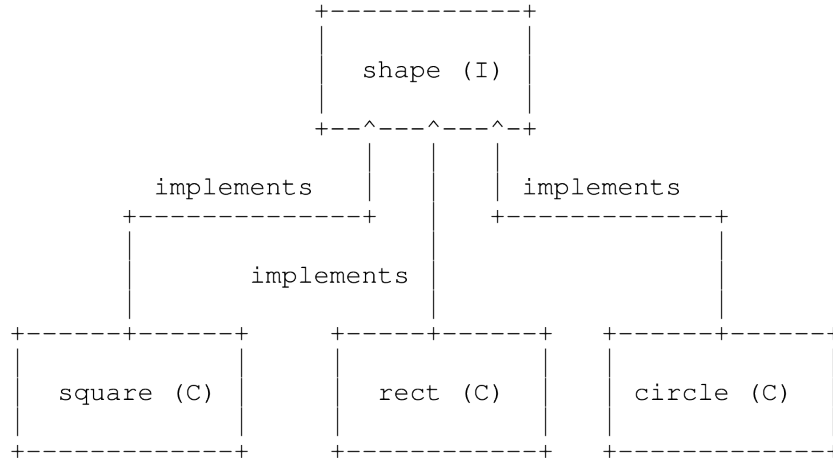
    (* The lower-left corner and the upper-right corner of the
       box that bounds the shape *)
    method bounding_box : point * point

    (* The center point of this shape *)
    method center : point
```

```
(* Translates the shape by the offset specified in the point *)
method translate : point -> unit
```

```
(* Dilates the shape by the scale factor *)
method scale : float -> unit
end ;;
```

(* This shape interface can have multiple classes implementing it, as so:



In the graphic above, an 'I' denotes a class type (interface) whereas a 'C' denotes a concrete implementation of that interface, a class. Concrete classes implement an interface class type, which are denoted by arrows and the label 'implements'.

Below, we'll create these classes that implement the 'shape' class type.

A class is a specification for how to build objects; you might think of a class as a blueprint and an instance of the class as a specific building created with that blueprint. (A class type in this analogy might be a set of architectural regulations that blueprints need to satisfy.)

Classes may include:

- o Definitions of instance variables and methods. Each object, or instance, of a class has its own copy of instance variables.
- o Information on how to construct and initialize objects.
- o Scope information about what to hold private.

Here, the arguments to 'rect' represent *constructor* arguments: values necessary to initialize the object.

Notice that the type of the 'rect' class is 'shape', or more properly, the 'rect' class implements the 'shape' interface.

.....
 Exercise 2A: Implement the 'rect' class. To think about: How do we store the values provided by the constructor? Keep in mind that the 'scale' and 'translate' methods may need to modify aspects of the object.
*)

```
class rect (initial_pos : point)
  (initial_width : float)
  (initial_height : float)
```

```

    : shape =
object (this)

  (* instance variables that store the rectangle's properties *)
  val mutable pos = initial_pos  (* lower-left corner of rectangle *)
  val mutable width = initial_width
  val mutable height = initial_height

  method area : float =
    failwith "rect area method not implemented"

  method bounding_box : point * point =
    failwith "rect bounding_box method not implemented"

  method center : point =
    failwith "rect center method not implemented"

  (* Destructively update `pos` to translate the shape by the values
     given by `vector`. *)
  method translate (vector : point) : unit =
    failwith "rect translate method not implemented"

  (* Scale the width and height of the rectangle from the lower-left
     corner by the scale factor `k`. *)
  method scale (k : float) : unit =
    failwith "rect scale method not implemented"
end ;;

(*.....
Exercise 2B: Implement the `circle` class.
.....*)

class circle (initial_center : point)
              (initial_radius : float)
              : shape =
object
  val mutable center = initial_center
  val mutable radius = initial_radius

  method area : float =
    failwith "circle area method not implemented"

  method bounding_box : point * point =
    failwith "circle bounding_box method not implemented"

  method center : point =
    failwith "circle center method not implemented"

  (* Destructively update position to translate the shape by the
     values given by `vector`. *)
  method translate (vector : point) : unit =
    failwith "rect translate method not implemented"

  (* Scale the radius by scale factor `k` without moving its
     center. *)
  method scale (k : float) : unit =
    failwith "circle scale method not implemented"

end ;;

(*.....
Exercise 2C: Implement the `square` class. Notice how similar it is to
`rect`!
.....*)

```

```

class square (initial_pos : point) (initial_side : float) : shape =
  object (this)
    method area : float =
      failwith "square area method not implemented"

    method bounding_box : point * point =
      failwith "square bounding_box method not implemented"

    method center : point =
      failwith "square center method not implemented"

    (* Destructively update 'pos' to translate the shape by the values
       given by 'vector'. *)
    method translate (vector : point) : unit =
      failwith "square translate method not implemented"

    (* Scale the sides of the square from the lower-left corner. *)
    method scale (k : float) : unit =
      failwith "square scale method not implemented"
  end ;;

```

(* Recall one of the original motivations for these exercises. We wanted to create a single 'area' function that returns the area of any shape. Let's discover how easy this is with our objects.

.....
 Exercise 2D: Create a function called 'area' that accepts a shape object and returns a 'float' of the area for that shape. Hint: If your definition isn't truly trivial, you're missing the point here.
*)

```

let area (s : shape) : float =
  failwith "area not implemented" ;;

```

(*.....
 Exercise 2E: Create a list of instantiated shapes called 's_list'.
 The list should contain, in order:

1. a 'rect' at (1, 1) with width 4 and height 5
2. a 'circle' at (0, -4) with radius 10
3. a 'square' at (-3, -2.5) with size 4.3

.....*)

```

let s_list = [] ;;

```

(* A DIGRESSION about a common confustication: As you might recall, lists can only contain objects of the same type. Why does the type system not show an error with your answer to 2E? What is the type of 's_list'?

When you've completed this exercise, you might notice that the type reported for this list is 'rect list'. Why is that, especially since not all the elements of the list are rectangles (!), and all elements of a list are supposed to be of the same type? The actual **type** associated with the elements of the list is an "object type", as described in Real World OCaml <https://dev.realworldocaml.org/objects.html>, in particular, something like:

```

< area : float;
  bounding_box : point * point;
  center : point;
  scale : float -> unit;
  translate : point -> unit >

```

The 'rect', 'circle', and 'square' objects are **all** of this type. But the ocaml REPL tries to be helpful in printing a more

evocative name for the type. By treating the class names as synonyms for the object types, the type of the list can be interpreted as a 'rect list' or 'circle list' or 'square list'. The REPL uses the class of the first element in the list as the type name to use, thus 'rect list'. Had the elements been in another order, the type might have been abbreviated as 'circle list' or 'square list'. END DIGRESSION *)

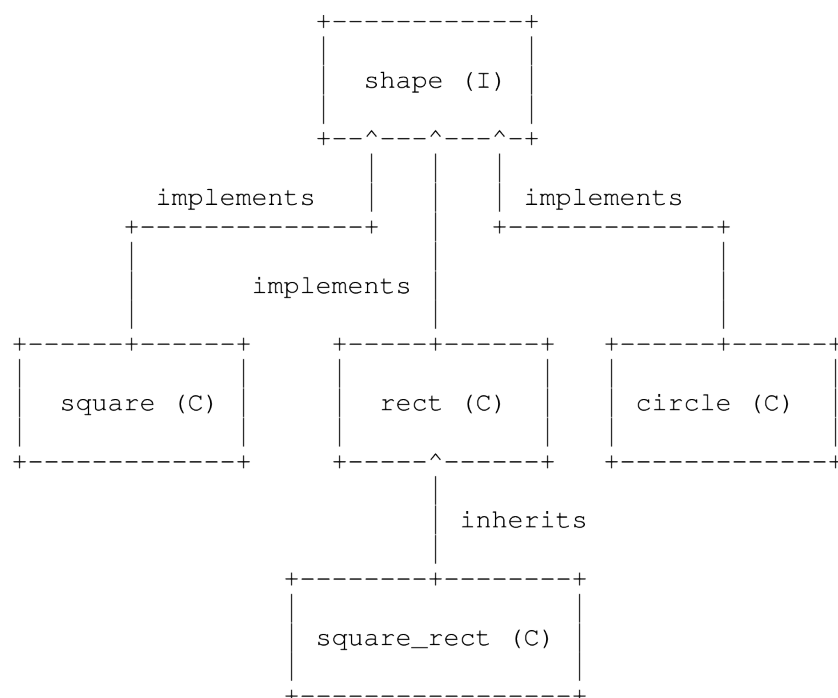
(*=====

Part 3: Representation, Inheritance

You might have noticed that 'rect' and 'square' are very similar representations, or implementations, of a class. They both use a 'point' to represent a lower-left corner, and both take side length(s). A square, as you know, is a rectangle with the additional constraint that its width is equal to its height.

We can reimplement 'square' to **inherit** from 'rect', and thus rely on 'rect''s existing representation.

In the end, we'll have this revised type hierarchy:



.....

Exercise 3A: Implement the 'square_rect' class which inherits all of its methods from its parent.

Note: In this and some later problems, we comment out the original stub code because it won't even compile yet until you finish the problem. Consequently, you're code won't compile on Gradescope until you complete these problems.

.....*)

```
(* UNCOMMENT AND COMPLETE
class square_rect (p : point) (s : float) : shape = ...
*)
```

(*.....

Exercise 3B: Now, implement a 'square_center_scale' class that inherits from 'square_rect', but **overrides** the 'scale' method so that the center (rather than the lower-left corner) of the square

stays in the same place.

You may find this tricky because *you don't have access to the instance variables of the class that you inherit from*. (Why is this?) A hint: First scale, then translate the center back to its original position.

.....*)

(* UNCOMMENT AND COMPLETE

class square_center_scale (p: point) (s: float) : shape = ...

*)

(* Before we move on, consider: do you need to make any modifications to the 'area' function you wrote in Exercise 2D to support these new classes? *)

(*=====

Part 4: Subtyping Polymorphism and Dynamic Dispatch

As we wander more around Flatland, we discover that there are more four-sided shapes than we originally thought. We knew about Square and Rect, but we've also seen Rhombi, Trapezoids, and other four-sided creatures that are collectively called Quadrilaterals.

Since Square and Rect both like to identify themselves as Quadrilaterals, which also identify themselves as Shapes, we need to make Quadrilateral a *subtype* of Shape.

Below, we have defined a new class type (interface) called 'quad'. Notice that 'quad' has all of the methods in 'shape''s signature, but adds an additional method, 'sides', that returns the lengths of each of the four sides.

Since 'quad' can do everything that a shape can do (and thus, wherever we expect a 'shape', we can safely pass a 'quad'), we consider 'quad' a *subtype* of 'shape'. *)

```
class type quad =
```

```
  object
```

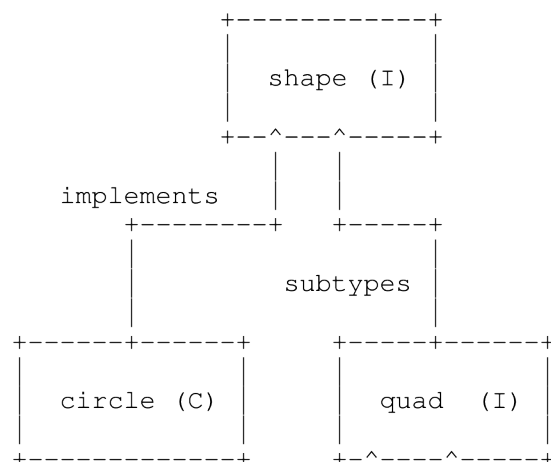
```
    inherit shape
```

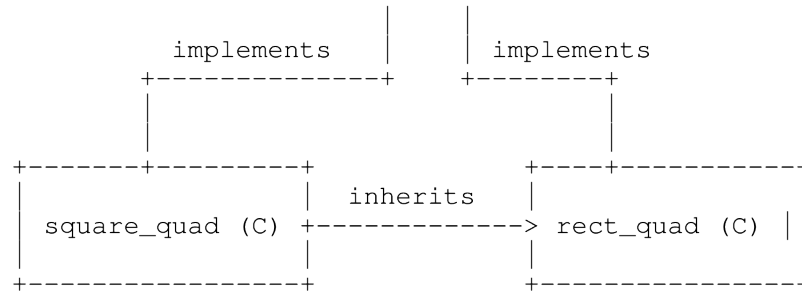
```
    (* returns the lengths of the four sides *)
```

```
    method sides : float * float * float * float
```

```
  end ;;
```

(* We can revise the type hierarchy as below, adding the 'quad' class type and some quadrilateral classes 'square_quad' and 'rect_quad', allowing us to drop 'square' and 'rect' but keeping 'circle'.





.....
 Exercise 4A: Write the class 'rect_quad', which represents a rectangle that implements a 'quad' class type. Hint: By taking advantage of existing classes, you should only need to implement a single method.
*)

```
(* UNCOMMENT AND COMPLETE
class rect_quad (initial_pos : point)
    (initial_width : float)
    (initial_height : float)
    : quad =

  object
  ...
end ;;
*)
```

(*.....
 Exercise 4B: Complete a class 'square_quad' that represents a square that implements a 'quad' class type. Hint: you shouldn't need to implement any methods!
*)

```
(* UNCOMMENT AND COMPLETE
class square_quad (p : point) (s : float) : quad =
  object
  end ;;
*)
```

(* Remember Exercise 2D, in which you implemented an area function for shapes? Amazingly, even though we have continued to create new shapes, due to subtyping and inheritance we can still rely on the existing implementation you already created! *)

(*.....
 Exercise 4C: Create an instance of 'square_quad' and name it 'sq'. Then, pass it to the area function to find out its area and store the result in a variable "a".

Hint: If you find yourself having problems with what ought to be a simple exercise, see the discussion of subtype coercion in Section 18.5.

.....*)

```
(* UNCOMMENT AND COMPLETE
let sq : quad = ... ;;

let a = ... ;;
*)
```

(*.....
 Exercise 4D: Write a function 'area_list' that accepts a list of shapes and returns a list of their areas.
*)


```
let area_list (shapes : shape list) : float list =  
  failwith "area_list not implemented" ;;
```