```
(*
                          CS51 Lab 16
                   Object-Oriented Programming
*)

(*
Objectives:

  Compare the effectiveness of function-oriented and object-oriented
  programming in various situations.
  Gain practice implementing objects in OCaml.
 *)

    (* An aside: In this lab, you'll use a `Point` module for x-y
    points, provided in the files `point.ml` and `point.mli`. You may
    want to look over `point.mli` to get a sense of the
    interface. (There's no need to look at `point.ml`. Do you
    understand why?) *)

open Point ;;

(*======================================================================
Part 1: Function-oriented vehicles

In this part, you will revisit algebraic data types to model some
vehicles, implementing functionality in a function-oriented approach,
with comparison to object-oriented implementations to come in Part 2.

We start with a `vehicle` type that can be a `Bus`, a `Car`, or a
`Truck`, each constructed of a `point` that represents its current x-y
position and a `float` that represents the energy stored in the tank
(measured in units of gallons of gas, a useful unit even for electric
vehicles). *)

type vehicle =
  | Bus of point * float
  | Car of point * float
  | Truck of point * float ;;

(*......................................................................
Exercise 1: Define a function `get_efficiency` that takes a `vehicle`
and returns its efficiency in units of miles per gallon (mpg). For
purposes here, buses get 20. mpg, cars get 30. mpg, and trucks get
15. mpg. (Notice that these values are `float`s.)
......................................................................*)

let get_efficiency _ = failwith "get_efficiency not implemented" ;;

(*......................................................................
Exercise 2: Write a function `get_energy` that returns the amount of
energy a vehicle has available. (Recall that the amount of energy is
a component of the `vehicle` data type.)
......................................................................*)

let get_energy _ = failwith "get_energy not implemented" ;;

(*......................................................................
Exercise 3: Write a function `get_pos` that returns the x-y position
of a vehicle as a `point`.
......................................................................*)

let get_pos _ = failwith "get_pos not implemented" ;;

(*......................................................................
Exercise 4: Let's define a function that allows these vehicles to
```

travel somewhere. Write a `go` function that takes a vehicle, a distance (a `float`), and a direction (an angle in radians represented by a `float`), and returns updated information about the vehicle -- its new position after traveling that distance in that direction, and its energy reduced accordingly.

Some important points:

- Your function should return a new `vehicle` with the updated position and energy. (Since the `point` and `vehicle` types are not mutable, you can't update the information "in place".)
- Calling the function with a negative distance should raise an `Invalid_argument` exception.
- Fortunately, you don't need to know how to do the calculation of the change in position, since the `Point` module can handle it for you; see the `Point.offset` function.
- Assume that a vehicle can only go as far as its energy will carry it, so if the `distance` is farther than that, the vehicle will go as far as its energy allows and stop with no remaining energy.

```
.....................................................................*)

let go _ = failwith "not implemented" ;;

(* At this point, you should be able to model a vehicle's movement
like this:

      # open Lab16 ;;
      # let mach5 = Car ((0., 0.), 5.) ;;
      val mach5 : Lab16.vehicle = Car ((0., 0.), 5.)
      # get_pos mach5 ;;
      - : Point.point = (0., 0.)
      # let mach5 = go mach5 5. 0. ;;
      val mach5 : Lab16.vehicle = Car ((5., 0.), 4.83333333333333304)
      # let mach5 = go mach5 5. (Float.pi) ;;
      val mach5 : Lab16.vehicle =
        Car ((0., 6.12323399573676628e-16), 4.66666666666666607)
      # get_pos mach5 ;;
      - : Point.point = (0., 6.12323399573676628e-16)
 *)

(*======================================================================
Part 2: Object-oriented vehicles

Having implementing a few functions to be performed on vehicles
defined as an algebraic data type, you might be getting weary of all
these match statements you need to add. Consider what would happen if
we added a new kind of `vehicle` -- say, a `motorcycle` -- to the mix?
We update the `vehicle` variant type definition to include motorcycles
simply enough, but suddenly we get a bunch of nonexhaustive match
cases in all the functions we implemented matching on vehicles. This
in itself is a source of motivation to pursue a better strategy. This
example is a small case, but consider what would happen if you had
dozens of instances in your code where you match to the `vehicle`
variant type. For each of these different locations, perhaps even in
different files that you have never seen or touched (if working in a
group), the extra match cases would need to be added.

We named our type `vehicle`. What do vehicles have in common? In our
case, they have a certain amount of energy, an efficiency with which
they use that energy, and the ability to move in two dimensional
space. We can provide that functionality as a _class_. When we need an
actual instance of that class, we can create an object based upon its
```

```
structure. We've provided the skeleton of a `vehicle_class` class
(called `vehicle_class` rather than `vehicle` because we already
called the algebraic data type `vehicle` and OCaml doesn't allow the
shared name). Notice the syntax for defining a class, as well as the
field variables and methods.

The `vehicle_class` class constructor takes several arguments:

  capacity -- the maximum amount of energy (in gallons of gas) that
  can be stored in the vehicle's tank/battery

  efficiency -- the vehicle's efficiency in mpg

  initial_energy -- the vehicle's initial amount of energy

  initial_pos -- the initial position of the vehicle

The class has several instance variables to store these values, as
well as an odometer field (initially 0.) to track the total number of
miles driven.

(Note that some fields are mutable while some are not. Do you see
why?) *)

class vehicle_class (capacity: float)
                    (efficiency : float)
                    (initial_energy : float)
                    (initial_pos : point) =
  object (this)
    val capacity = capacity
    val efficiency = efficiency
    val mutable energy = initial_energy
    val mutable pos = initial_pos
    val mutable odometer = 0.

    (*..............................................................
    Exercise 5: To this vehicle class, add methods `get_distance`,
    `get_pos`, and `get_energy` which return the current distance
    traveled, position, and remaining energy, respectively.
    ..............................................................*)

    method get_distance : float =
      failwith "get_distance method not implemented"

    method get_pos : point =
      failwith "get_pos method not implemented"

    method get_energy : float =
      failwith "get_energy method not implemented"

    (*..............................................................
    Exercise 6: Now add a method to your vehicle class called `go`
    which takes a `float` for the distance the vehicle wants to travel
    and a `float` for the direction it should head. The actual
    distance traveled should be dependent on the energy available as
    in the implementation of the `go` function above. Instead of
    returning a new object, you should simply be updating the fields
    of the same object. Notice how this differs from the way you did
    it before.
    ..............................................................*)

    method go _ =
      failwith "go method not implemented"

    (*..............................................................
```

```
    Exercise 7: Since we'll eventually run out of energy, it would be
    useful for a vehicle to recharge or fill the tank. Define a `fill`
    method that resets the energy to whatever the vehicle's capacity
    is.
    ..............................................................*)

    method fill : unit =
      failwith "fill method not implemented"
  end ;;

(* At this point, you should be able to model a vehicle's movement
like this:

      # let mach5 = new vehicle_class 5. 30. 5. (0., 0.) ;;
      val mach5 : Lab16.vehicle_class = <obj>
      # mach5#get_pos ;;
      - : Point.point = (0., 0.)
      # mach5#go 5. 0. ;;
      - : unit = ()
      # mach5#get_pos ;;
      - : Point.point = (5., 0.)
      # mach5#go 5. Float.pi ;;
      - : unit = ()
      # mach5#get_pos ;;
      - : Point.point = (0., 6.12323399573676628e-16)
      # mach5#get_energy ;;
      - : float = 4.66666666666666607
      # mach5#fill ;;
      - : unit = ()
      # mach5#get_energy ;;
      - : float = 5.
 *)

(*======================================================================
Part 3 Inheritance

Let's say that we know certain facts about the energy and efficiency
corresponding to buses, trucks, and cars as we did above and thus
would like to conveniently be able to refer to them in more detail
than just as vehicles, for whatever purpose we may have down the road
(so to speak). Instead of implementing a completely new class for each
of the types of vehicles, we can make use of code we already wrote in
the `vehicle_class` definition by taking advantage of inheritance. Try
this out for yourself below.

......................................................................
Exercise 8: Define a `car` class, taking advantage of inheritance by
inheriting from the `vehicle_class` class.  Objects of the `car` class
should have the energy efficiency and capacity as in this table:

           efficiency  capacity
    Car        30.        100.
    Truck      15.        150.
    Bus        20.        200.

Hint: Because of the power of inheritance, your definition of the car
class should require only a few lines of code.
......................................................................*)

class car (initial_energy : float) (initial_pos : point) =
  object
    (* implement the car class here *)
  end ;;

(*......................................................................
```

```
Exercise 9: Now, define a `truck` class similarly to the `car` class,
but with the truck's specifications given in Part 1.
..............................................................*)

class truck (initial_energy : float) (initial_pos : point) =
  object
    (* implement the truck class here *)
  end ;;

(*..............................................................
Exercise 10: Finally, define the `bus` class. Rather than merely
inherit all the functionality from the vehicle class, we'll add
additional functionality that makes a bus a bus.

A bus should be able to pick up and drop off passengers. To implement
this functionality, give your bus class additional instance variables:

    `seats` -- the number of seats on the bus, which defines the
       maximum number of passengers it can accommodate

    `passengers` -- the number of passengers currently on the bus

(You'll want to think about whether these fields should be mutable or
not.)

The class should allow for methods `pick_up` and `drop_off`, which
both take the number of passengers to perform the action on and return
a `unit`. Keep in mind you can't drop off more passengers than you
currently have, so in this case you can just reset your passenger
count to 0. A bus also offers finite seating (50 seats, say)
and this rule should be enforced as well. So, if 70 people try to
board the bus at the same time, only the first 50 will be able to.

Furthermore, when a bus goes in for a fill-up, it will behave as a
vehicle, but first needs to drop off all its passengers. Override the
`fill` method to implement this functionality.
..............................................................*)

class bus (initial_energy : float) (initial_pos : point) (seats : int) =
  object
    (* Complete the implementation of the `bus` class here. Here are
       some things you'll likely want to add to any inherited instance
       variables and methods:

              val mutable passengers
              val seats = seats

              method get_passengers : int
              method get_seats : int
              method pick_up : int -> unit
              method drop_off : int -> unit
   *)
  end ;;
```