Code Review 8: April 10, 2020

Object-Oriented Programming

Hakeem Angulu, hangulu@college.harvard.edu

Learning Goals

- Understand the components of the Object-Oriented Paradigm
- Compare those components to functional programming
- Be able to use inheritance to prevent redundancy/code repetition

---

Terminology

Object-Oriented Programming is our latest programming paradigm, following our study of functional programming, imperative programming, and lazy programming.

An object is something that has data stored in **fields** (frequently called instance variables) and has **methods**: functions that can be called on the object and can interact with the data in the object's fields.

Fields can be mutable or immutable, and methods are able to read both sorts of fields but can only modify mutable fields. Methods are similar to regular functions, with the notable exception that methods are allowed to take zero arguments (you might think of the object on which the method is invoked as always being an implicit argument).

Classes

While **objects** are concrete things with real data, a **class** is a **recipe or a blueprint** for creating new objects. A class describes all of the fields and methods that objects of that class have.

As an example, here's a camel class which encapsulates the two important properties of a camel:

```
class camel (humps : int) =
  object
    val humps = humps
    method say : string = Printf.sprintf "I am a %d-hump camel" humps
  end
```

This class is a blueprint for camels in general. To get a object representing one real camel, we need to instantiate the class to create an object. To this end, classes provide special methods called constructors that take some parameters and return a new instance of the class.

The class has a **constructor** called `camel` that takes an `int` as an argument, representing the number of humps the camel has. To instantiate the `camel` object, one would do the following:

```
let my_camel = new camel 1
```

Note that you do not need a class to create an object. An object can be created directly as such:

```
let my_camel =
  object
    val humps = humps
    method say : string = Printf.sprintf "I am a %d-hump camel" humps
  end
```

To invoke a method on the `camel` object, use the following:

```
>> my_camel#say
<< I am a 1-hump camel
```

Exercise 1: The module and class implementations of a stack are below. What are the main differences between the two?

```
module MakeStack (Element: SERIALIZE)
    : (STACK with type element = Element.t) =
  struct
    exception Empty
    type element = Element.t
    type stack = element list
    let empty () : stack = []
    let push (el: element) (s: stack) : stack = el :: s
    let top (s: stack) : element = ...
    let pop (s: stack) : stack = ...
  end ;;

let s = MakeStack(IntSerialize).empty ;;
```

```
class ['a] stack init =
  object(this)
    val mutable internal : 'a list = [init]
    method push e =
      internal <- e :: internal;
      ()
    method top () =
      match internal with
```

```
      | [] -> None
      | h :: _t -> Some h
    method pop () =
      match internal with
      | [] -> None
      | h :: t ->
        internal <- t;
        Some h
  end ;;

let s = new stack 5 ;;
```

1. Type annotation is explicitly required in the object-oriented version, but that type annotation can be polymorphic. This is in constrast to the functor, which requires a parameterization with another module to be polymorphic.
2. At least in this implementation, a module-based stack is immutable, whereas an class-based stack is mutable.
3. Types cannot be declared within a class – we're stuck leaving the implementation of the object-based stack exposed to the client.

Exercise 2: Below is the implementation of a `course` class. Change that implementation to modify the course's title and professor.

```
class course (t : string) (p : string) =
  object
    val title = t
    val motto = p
    method title : string = title
    method professor : string = professor
  end ;;
```

```
class course (t : string) (p : string) =
  object
    val mutable title = t
    val mutable professor = p
    method title : string = title
    method professor : string = professor

    method update_title (t : string) : unit =
      title <- t
    method update_prof = (p : string) : unit =
      professor <- p
  end ;;
```

Inheritance

Inheritance allows you to write a new class by only adding or changing the methods that are different. This is an excellent way to avoid repeating code.

The class that you are extending or modifying is called the **superclass** or **parent class**, and the new class being defined is called the **subclass** or **child class**.

Here is an example that creates a child class of the `camel` parent class, updating it to allow for camels that breathe fire.

```
class first_breathing_camel (humps : int) =
  object
    inherit camel humps
    method breathe_fire : string = "rawr"
  end ;;
```

The objects of the class `first_breathing_camel` have access to all of the methods of `camel` objects, through inheritance.

When modifying a class instead of simply adding to it, you do the following:

```
class mean_camel (humps : int) =
  object
    inherit camel humps as super
    method! say = super#say ^ " and I love C++."
  end ;;
```

Note that to modify the `say` method, you need to use the `method!` keyword instead of the `method` keyword, and to access the `say` method of the parent class, you need to inherit it as `super`.

Exercise 3: Create a new `course` class, `detailed_course`, with all the methods of the old (mutable) one, but also a field for the number of students enrolled and a method for changing that number.

```
class detailed_course (t : string) (p : string) (n : int) =
  object
    inherit course t p as super
    val mutable num_students = n
    method update_num (n : int) : unit =
      num_students <- n
  end ;;
```

Class Types and Subtyping

Like module signatures, class types serve two purposes:

1. Class types act as interfaces: contracts or promises of the behavior that a class implements.

2. Class types allow programmers to control which parts of a class are accessible. It's considered good design to only expose what is necessary in the class type.

Subtyping is used to describe the following situation: suppose that an object `a` has all of the functionality of an object `b` (and possibly more). This means that anywhere that someone is expecting `b`, it would be safe to give them `a` since `a` implements all of `b`'s functionality. Here, we say that `a` is a subtype of `b` and that `b` is a supertype of `a`.

The following is an example of that situation, using 2D and 3D points:

```
class point_two (x : int) (y : int) =
  object
    method x = x
    method y = y
  end ;;

class point_three (x : int) (y : int) (z : int) =
  object
    method x = x
    method y = y
    method z = z
  end ;;

let get_x (p : point_two) = p#x in
let my_point = new point_three 1 2 3 in
get_x (my_point :> point_two)
```

This code, which passes an object from the point_three class to a function expecting a point_two class is safe because 3D points are a subtype of 2D points. Note that explicitly using the subtype coercion operator, `:>`, is necessary in the final line.

Note that subtyping and inheritance are different but related concepts: an object is subtype of another if it contains all of the same functionality of that object (or more) – essentially, an object `a` that is a subtype of `b` has inherited `b`'s interface.

The implementations of the methods and fields in `a` can be completely independent of those in `b`, as long as they have equivalent signatures.

In inheritance, `a` has not only inherited the signature, but has inherited the implementation as well: the implementation of the functions in `a` are no longer independent of the implementations in `b`. Inheritance is definitely a popular way of creating subtypes, but inheritance alone does not encapsulate all of the nuances of the idea of a subtype.

Exercise 4: Write the class for a zombie object. You must include the following fields: 1. the zombie's name 2. the number of brains the zombie has eaten 3.

whether or not the zombie can move quickly 4. whether the zombie is alive (undead) or dead (dead)

Think about which fields should be mutable. If fields should be mutable, create methods to mutate them.

```
class zombie (n: string) (brains: int) (fast : bool) (undead : bool) =
object
  val name = n
  val mutable brain_count = brains
  val fast = fast
  val mutable undead = undead

  method get_name = name
  method get_brain_count = brain_count
  method get_speed =
    if fast then "fast" else "slow"
  method get_undead =
    if undead then "undead" else "dead"

  method eat_brain : unit =
    brain_count <- brain_count + 1
  method be_killed : unit =
    undead <- False
end ;;
```

Exercise 5: Write a function that kills (makes an undead zombie dead) zombies if their brain count is higher than 3.

```
let zombie_slayer zombie =
  if zombie#get_brain_count > 3 then zombie#be_killed else () ;;
```

The solutions to all of the above exercises will be available on Sunday at noon.

For this and future code reviews, please do not hesitate to reach out to me with any questions or concerns. My email can be found at the top of this document.