CSE305 Week 12: New Issues in OOP

The issues stated at the end of the last lecture:

1. Is there a notion of **object** that is *separate* from the notion of **class**? Can one have objects without classes?
2. Is there support for **singleton objects**, like `Clientele` in the "`Customer`" example?
3. Are constructor arguments distinguished as class arguments from other fields? (Scala and OCaml: yes)
4. Are **accessor** features of classes distinguished in other ways? (get/set in Scala, C#)
5. Is class extension treated as *specialization* or as *generalization*? (Continuing discussion from before spring break, we will use thuis as one "angle" on OCaml...)

These lead right away into some other issues that apply also in the older OO languages. And among the issues listed by Sebesta, let's recall his #2 (meaning section 12.3.2) "**Are Subclasses Subtypes**?" His page there is very meager and uses only the ghostly Ada for an example; OCaml will help us flesh it out. Let's jump in.

Objects Versus Records in OCaml

An object in OCaml is just like a record---except for lots of differences. Here are thematic ones:

1A. The *exact type* of a record is what matters. The type has to be declared before a record of that type can be created. The type of a record is the set of *both names and types* of its fields. Since functions are first-class objects, they can be fields. Here is an example:

```
type myrec = { start:int ; f: int->int ; name:string };;
let x = { name="double" ; f = (function n -> 2*n); start=1 };;
```

Ocaml echoes the first line as `type myrec = { start:int ; f: int->int ; name:string}` and then replies: `val x : myrec = {start = 1; f = <fun>; name = "double"}`. Then

```
x.f(x.start);;
```

gives 2. Is `f` like a method? Really not. You can't do `x.f(start)`, and this gives an error too:

```
let z = { start=1; f=(function n->n*start); name="mulbystart" };;   (*Error*)
```

You can define an ordinary function with a record as parameter, and OCaml will even infer the record type as the most recent one that is applicable (well, you should really explicitly give the parameter the desired record type):

```
let g y = y.f(y.start);;   (* better: let g(y:myrec) = y.f(y.start);; *)
```

OCaml replies `val g : myrec -> int = <fun>`. Now for the point of what we mean by needing the exact type for compatibility, let's define what looks like an "extending type" of `myrec`:

```
type otherrec = { start: int; f: int->int ; name: string; diff: float };;
let other = { start=3; f = (function n -> n*n); name="square"; diff = -3.0 };;
g(other);;  (* Error *)
```

Ocaml's error reads like usual: "Error: This expression has type `otherrec` but an expression was expected of type `myrec`." This comes back even though you *can* execute `g(other)` to have value $3^2 = 9$. Languages like Python that allow this kind of call are said to use **duck typing** ("if it walks like a duck and quacks like a duck, it counts as a duck"). Ruby is another such language.

1B. Whereas, OCaml objects obey a *subtype compatibility* rule. It would allow passing an *object* like `other` to a function like `g` originally defined for an object type `myobj` that is like `myrec`.
Before we get into this, we note six other differences:

2. OCaml object types are encased in `< ... >` not `{ ... }`. Perhaps this is to remind you that their types obey a kind of '<' relation.

3. The fields and their names are part of the record type. But the fields in OCaml objects are not part of the type, and in that sense have non-public visibility. You can, however, define a getter method that returns the value of a given field---and if the field is `mutable`, also a setter method.

4. OCaml records provide matchable structure; OCaml objects do not.

5. An OCaml object does not need a type declaration. It may be the body of a constructor definition that is simultaneous with defining the object---as with classes in Scala.

6. OCaml record members cannot see other fields of the same record. The best you can do is try to define a record object recursively:

```
let rec u = { start=1; f=(function n->n*u.start); name="nxstart"; diff=5.0 };;
```

OCaml accepts this and replies `val u : otherrec = {start = 1; f = <fun>; name = "nxstart"; diff = 5.}`. Weird, and I haven't seen a case where this is useful, but this works into something many of us have seen about objects: Some languages like Python require prefixing `self` to access a field. Java and C++ (etc.) do not, while Scala is midway in sometimes needing a user-defined "self"-name in syntax like "`class Foo (...) { Outer =>` " before the class body. OCaml uses `self` and calls this "Open Recursion"---I guess by analogy to the above. (?)

7. Instead of the dot, OCaml objects use **#** to access fields.

Objects in OCaml

The official OCaml source https://v2.ocaml.org/manual/objectexamples.html leads off with classes, as does section 18.2 of the book by Stuart Schieber used as reference before spring break.   The other reference given at the end of my Week 11 notes, https://dev.realworldocaml.org/classes.html, leads with a nice integer stack `istack` example having a class, but the previous chapter---

https://dev.realworldocaml.org/objects.html

---starts with a simpler version of the same example without a class. So let's go there. (I've condensed some lines and changed indentation.) This example shows mutability right away:

```
let s = object
   val mutable v = [0; 2]   (* see below for general construction *)

   method pop = match v with
      | hd :: tl ->
           v <- tl;
           Some hd
      | [] -> None

   method push hd = v <- hd :: v
end;;
```

OCaml responds with the type of the object:

```
   val s : < pop : int option; push : int -> unit > = <obj>
```

The type does not have a given name, though you can do "type foo = " followed by that to give it a name. It has to be a lowercase name, an issue we will "fix" by residing objects inside a module.

The `pop` method uses sequencing to execute a side effect on the stack, which is represented as a list called `v` with top at the left. The `push` method has body that only executes a side effect and has return type `unit`. This is in the manner of a `void` method in C/C++/Java but `unit` is more versatile. The thing to note first is that the field `v` is *not* part of the type.

This version of a stack ADT does not raise an error for trying to pop an empty stack. You could make it throw an exception---but where to define the exception? does it make sense as a field member of the object? *hmmm...* Before we get there, let's continue with the site's examples:

```
s#pop;;
- : int option = Some 0     (* stack now has just [2] *)
s#push 4;;
- : unit = ()               (* stack now has [4;2] *)
s#pop;;
- : int option = Some 4     (* stack back to [2] *)
```

Before we move on with the site's next example of giving the object a constructor, let's see the point about type compatibility in action. Without referencing the type or any stack object at all, let's define a function that illustrates the point about subtyping:

```
let iggypop(x) = if x#pop = Some 2 then "Two!" else "Not Two!";;
```

Ocaml says: `val iggypop : < pop : int option; .. > -> string = <fun>`. Again, it gives an object type without giving the type a name. This type only specifies that there has to be a zero-parameter `pop` method that returns `int option`. Will it work on our stack object `s` from a type that defines more stuff? Unlike with the analogous record example, yes:

```
iggypop(s);;
```

This works! Anything that "pops like a duck" is compatible with the parameter `x`. More notable, with the above example (either with or without doing the above three lines), if you repeat the call

```
iggypop(s);;
```

the answer comes out different. No referential transparency here. The *sinfulness* of this is maybe why the official OCaml site wards off using objects in its preamble, but there's much to like. A way to understand it is by analogy to `interfaces` in Java or C# or `traits` in Scala. In these languages, one can write an interface/trait called "`Poppable`" specifying just the presence of a pop() function (returning `int` or could be generic). Then the `iggypop` function could be written---

```
String iggypop(Poppable x) { return (x.pop().equals(new Integer(2)) ? "Two!" : "Not Two!"); }
```

---and any object whose class implements `Poppable` could be passed to it. OCaml infers the interface(s) that an object can implement automatically---as was remarked also about OCaml modules and their signature types. This is IMPHO the best way to understand traditionally what otherwise comes off as "duck typing"---where an important difference from Python is that OCaml can tell that things are **type-safe** at compile time, thus avoiding run-time errors.

Objects With Construction, and Syntax of Bodies

Again from the "RealWorldOCaml" page:

```
(** int list -> object of this type *)
let stack init = object
   val mutable v = init

   method pop = match v with
      | hd :: tl ->
          v <- tl;
          Some hd
      | [] -> None

   method push hd = v <- hd :: v
end;;

let s = stack [3; 2; 1];;
s#push 4;;
```

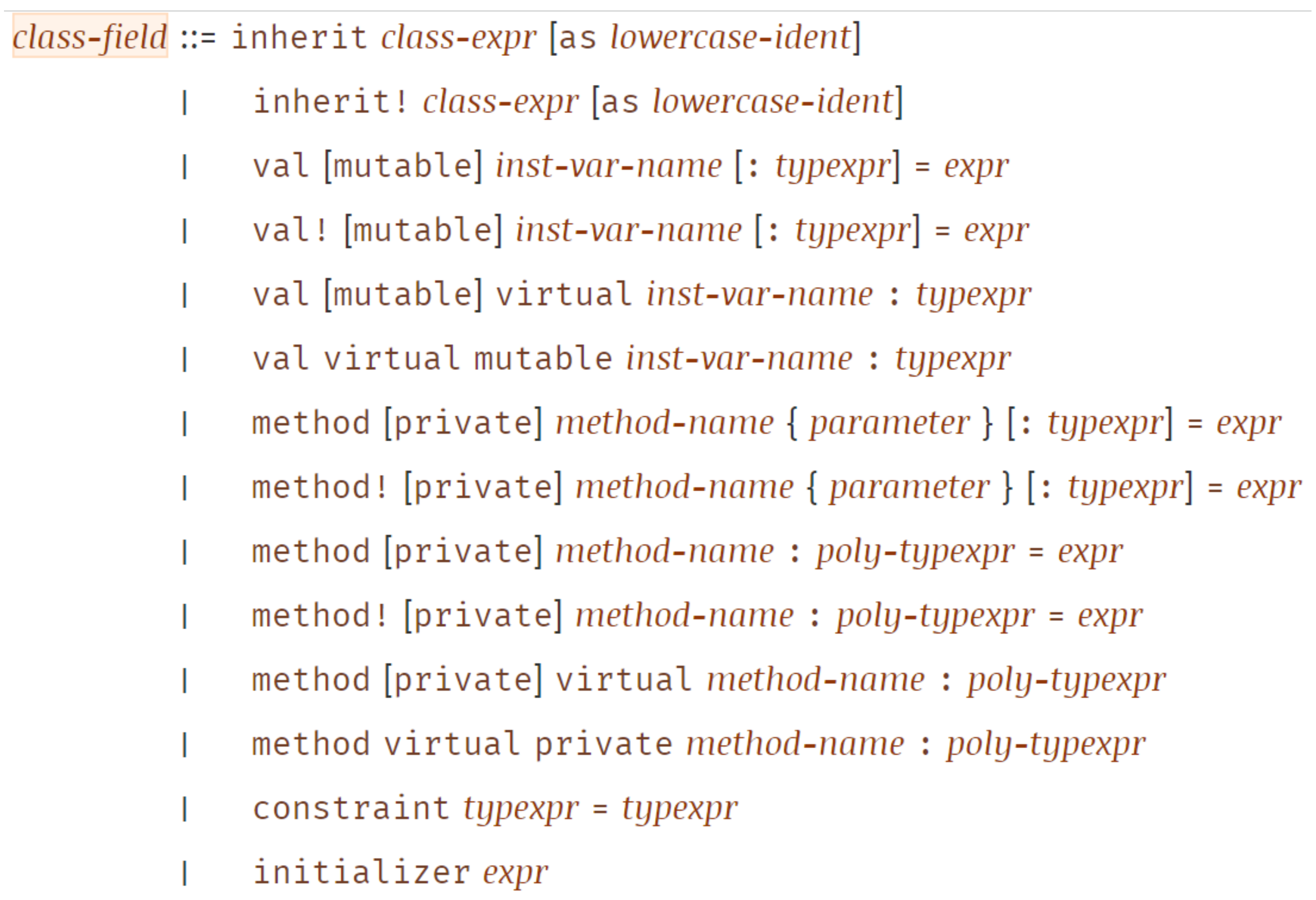
We can think of stack as the name of the class and the name of the constructor. But it is just an ordinary function. Its body is an "object expression"---which is the last option in the official OCaml grammar for expressions and which has forms like so:

```
object-expr     ::=
                 |   new class-path
                 |   object class-body end
                 |   expr # method-name
                 |   inst-var-name
                 |   inst-var-name <- expr
                 |   {< [ inst-var-name [= expr] { ; inst-var-name [= expr] } [;] ] >}
```

Note that the last part involves braces too and has completely optional contents---what's up with that? Except for noting that the third line says that a method invocation is an expression, let's focus on the second line:

```
class-body ::=   [( pattern [: typexpr] )] { class-field }
```

The braces are EBNF braces, not class curly braces like in C/C++/Java/C#/etc. The rest of the syntax---you need only treat lines 3 and 7 and maybe the last line as necessary---is:



```
class-field ::= inherit class-expr [as lowercase-ident]
             |   inherit! class-expr [as lowercase-ident]
             |   val [mutable] inst-var-name [: typexpr] = expr
             |   val! [mutable] inst-var-name [: typexpr] = expr
             |   val [mutable] virtual inst-var-name : typexpr
             |   val virtual mutable inst-var-name : typexpr
             |   method [private] method-name { parameter } [: typexpr] = expr
             |   method! [private] method-name { parameter } [: typexpr] = expr
             |   method [private] method-name : poly-typexpr = expr
             |   method! [private] method-name : poly-typexpr = expr
             |   method [private] virtual method-name : poly-typexpr
             |   method virtual private method-name : poly-typexpr
             |   constraint typexpr = typexpr
             |   initializer expr
```

This needs some "translation from French"---as the OCaml site warns, "Note that the relationship between object, class and type in OCaml is different than in mainstream object-oriented languages such as Java and C++, so you shouldn't assume that similar keywords mean the same thing."

- "`virtual`" means the same as `abstract` in Java and `@abstractmethod` in Python.
- "`method!`" means that the method must override some superclass method---well, we haven't seen inheritance yet. Also, `val!` allows overriding a field and `inherit!` just means that the class or object being defined has to override something it inherits.
- The `initializer` part exactly means extra lines of code that some constructors have to do after initializing the class fields.
- A method with templated type variables `'a` etc. has to be given a type annotation saying so.

Example: Running an Iterator on a Fixed String

[show code Swi.ml (draft, hence only in ~regan/LANGUAGES/OCAML/ on "timberlake")]

Subtypes and Compatibility.

It is OK to state the definitions intuitively and operationally before grappling with the formal definition and its generally-useful ramifications.

Definition: An object `x` is **compatible** with a type `T` if whenever a function `f` is defined with a parameter `t` of type `T`, the call `f(x)` is legal.

Definition: A type `S` is **compatible** with a type `T` if every object of type `S` is compatible with `T`.

We may already greenlight the identification the notion of **subtype** with being compatible, and write `S < T` if so. It should maybe be `S <= T` since a type is always compatible with itself, but that could overlap other syntax. Sometimes by `S < T` we really mean that `S` is a proper subtype of `T`. The `<` relation is *not* symmetric---just like the relationship of derived class to base class does not go the other way around. Some mantras of OCaml that go with this:

- *Subclass* need not imply *subtype*.
- Classes don't have types in OCaml anyway; only their internal object has a type.
- Subset does not imply subtype either: `int` is not a subtype of `float` (the way it is in C/C++/Java etc.)

The basic types do not have any subtype relations in OCaml, except for the identity relation: `int` is a subtype of **int**, etc.. The only basis for the following two recursive definitions is the duck-typing of objects shown above:

Definition: A function type `C -> D` is a **subtype** of a function type `A -> B` if `D < B` and `A < C`.

The mantra here is: *wider arguments, narrower return.* In conventional OO languages this is what makes a legal override.

Definition: For object types `S` and `T`, we write `S < T` and say `S` is a **subtype** of `T` if for every method `foo` (or other visible member) in the type spec of `T`, `foo` is in the type spec of `S` by the same name, and the type of `S#foo` is a subtype of the type of `T#foo`.

More simply put, `S` has everything that `T` has, and for each of those shared things, the version in `S` is compatible with the version in `T`.

Example: `< pop : int option; push : int -> unit >` is a subtype of `< pop : int option; >`

That is why our stack type was compatible with the parameter of the function `iggypop`. Remember that "subtype" is meant inclusively: a type is compatible with itself.

A good traditional way to understand what is going on is to pose Sebesta's question: **Are subclasses subtypes?**

- On the face of it, a derived class inherits all the (fields and) methods of the base class, and only adds "more stuff". The use of `private` can complicate this picture, but that's the basic idea of why subclasses are treated as subtypes in traditional OO.

- Revisiting (at long last) the `Square` versus `Rectangle` example, the issue is that a Rectangle can have a mutator method `double_x()` that doubles one of its sides, but if a call `r.double_x()` is made when the `Rectangle` variable `r` holds a `Square` object `s`, this will violate the logical property of `s` being a `Square`. The call is still type-safe, however---provided the separate `x` and `y` fields of Rectangle were inherited without `private`.

- Where things get dicier is if the class defines a method like this:

```
class Rectange {
   ...
   bool nestedIn (Rectangle other)  {
      ...body using rectangle representation ...
   }
}

class Square extends Rectangle {
   @Override??
   bool nestedIn (Square other)  {
      ...body using square representation to take shortcuts...
   }
}
```

Now the problem is, what happens if we have a call

```
  r1.nestedIn(r2)
```

where `r1` and `r2` are variables of type Rectangle? The problem occurs when `r1` holds a `Square` but `r2` only holds a `Rectangle`.

- Because dynamic dispatch only consults the invoking object (run-time type lookup of parameters too could be really slow), if `Square.nestedIn` were taken as a legal override, it would bind that code.
- But running that code would crash because `r2` holds an object that does not have the `Square`-specific elements needed for the body.

The nub is that `Square.nestedIn` is not a legal override because its parameter needs to be at-least-as-wide, not at-least-as-narrow as the parameter of `Rectangle.nestedIn(Rectangle)`. In this case it would make sense to code `Square.nestedIn(Rectangle)` from the get-go. But now let's just move things up one step in the object hierarchy to `Rectangle extends Polygon`. Determining whether a rectangle is nested inside an arbitrary polygon is a nontrivial matehmatical task.