CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 14/15?

Object-Oriented Programming

Don't Believe the Hype

OOP provides concise ways to build extensible software and exploit a sometimes-useful analogy between interaction of objects in physical systems and interaction of software parts.

It also raises tricky semantic and style issues that require careful PL investigation. (Good thing we're doing it near the end!)

Personally, I am skeptical about:

- The coding factor (X lines/day of accessor methods!)
- The Barnes&Noble factor (certified insane in X days!)
- The intro factor (by week X you can write your own class!)

If it takes an advanced degree to understand objects, I've got the right audience!

So what is OOP?

It seems to look like this... That's a lot; what's the essence?

```
class Point1 extends Object {
 int x;
 int  get_x() { x }
 unit set_x(int y) { self.x = y }
 int distance(Point1 p) { p.get_x() - self.get_x() }
 constructor() { x = 0; }
}
class Point2 extends Point1 {
 int y;
 int get_y() { y }
 int get_x() { 34+super.get_x(); }
 constructor() { super(); y=0; }
}
```

OOP can mean many things

- An ADT (private fields)
- Subtyping
- Inheritance, method/field extension, method override
- Implicit this/self
- Dynamic dispatch
- All the above (plus constructor(s)) with 1 class declaration

Let's consider how OO each of these is...

Side question: Is "good design" many combinable constructs or one "do it all" construct?

OO as ADT-focused

Object/class members (fields, methods, constructors) often have *visibilities* and "more private" is sort of "more abstract"

What code can invoke a method/access a field? Other methods in same object, other methods in same class, a subclass, within some other boundary (e.g., a package), any code, ...

With just classes, the only other way to hide a member is upcasting. With *interfaces* (which are more like record types), we can hide members more selectively:

```
interface I { int distance(Point1 p); }
class Point1 { ... I f() { return self; } ... }
```

Last lecture we saw objects are a bad match for "strong binary methods" (we'll come back to this)

Records with private fields

Let's assume all fields are visible only to self object. Then objects are just a record of methods with private state.

So far, this is no need for a fancy OO language:

```
let point1_constructor () =
    let x = ref 0 in
    let rec self =
    { get_x  () = !x;
       set_x   y = x := y;
       distance p = p.get() - self.get() } in
    self
```

But there's more to it (haven't considered inheritance yet)

Subtyping

Most class-based OO languages "confuse" classes and types:

- If C is a class, then C is a type.
- If C extends D (via declaration), then $C \leq D$.
- Subtyping is (only) the reflexive, transitive closure of this.

Is this novel? If C adds members, that's width subtyping.

By this is "by name" subtyping. If classes $C1$ and $C2$ are *incomparable in the class hierarchy* they are *incomparable types*, even if they have the same members.

We will definitely revisit this "subclassing is subtyping" assumption! (For now, it restricts subtyping *and* subclassing!)

Subtyping, continued

If C extends D and *overrides* a method of D, what restrictions should we have?

- Argument types contravariant (assume less about arguments)
- Result type covariant (provide more about result)

Many “real” languages are even more restrictive.

Some bend over backward to be more flexible. (Don’t!)

It’s good we studied this in a simpler setting.

Inheritance and Override

A subclass *inherits* the fields and methods of its superclass. It can override some methods and have special "super" (a.k.a. resend) calls.

This isn't hard in ML either, if everything is visible:

```
let point1_constructor () =
    let rec self =
    { x           = ref 0
       get_x  () = !x;
       set_x   y = x := y;
       distance p = p.get() - self.get() } in self
```

continued...

Continued...

```
let point2_constructor () =
    let r = point1_constructor () in
    let rec self =
    { x = r.x;
       get_x () = 34 + r.get();
       dinstance = r.distance;
       y = ref 0;
       ... } in self
```

Fields visible only in subclasses requires multiple abstractions (doable).

Also have to change point2 code when point1 changes, but often true in OO too ("fragile base class" issues).

Then what is it?

I claim class-based objects are poor (maybe okay) ADTs, same old subtyping, and a little syntactic sugar for extension and override.

So what is that makes OO different in an intellectually interesting way?

Answer: The "late" binding of self and the dynamic dispatch that results.

The difference between `point2_constructor()` and an object of class `Point2` is in the behavior of `distance`.

More on late binding

Late-binding, dynamic dispatch, and open recursion are all closely related ideas. The simplest example I know:

Functional (even still $O(n)$) vs. OO (even now $O(1)$):

```
let c1() = let rec r = {
  even i = if i > 0 then r.odd  (i-1) else true;
  odd  i = if i > 0 then r.even (i-1) else false} in r
let c2() = let r1 = c1() in
  let rec r = {even = r1.even; odd i = i % 2 == 1} in r
class C1 {
  int even(int i) {if i>0 then odd(i-1)  else true;}
  int odd(int i)  {if i>0 then even(i-1) else false;}}
class C2 extends C1 {
  int odd(int i) {i % 2 == 1} }
```

Political Spin

“Call your congresspeople. Tell them `C2` should have the right to change even by overriding odd. It’s a question of code reuse and you deserve better subclasses.”

“Call your congresspeople. Tell them `C1.even` shouldn’t break whenever `C2` decides to write a bad `odd`. You deserve quality code, regardless of subclasses.”

Meanwhile, public television has a “boring” documentary about quiet behind-the-scenes work to *understand* the approaches and how the shortcomings of each can be compensated.

More about OO to come. . .