CSEP505: Programming Languages
Lecture 10: Object-Oriented Programming; Course Wrap-Up

Dan Grossman
Autumn 2016

Onto OOP

Now let's talk about (class-based) object-oriented programming

- What's different from what we have been doing
  - Boils down to one important thing
- How do we define it (will stay informal)
- Supporting extensibility
- Some "issues" not handled well

Won't have time for: "more advanced OOP topics"

- Multiple inheritance, static overloading, multimethods, …
- I, at least, have "no regrets" about "making room for Haskell"

OOP the sales pitch

OOP lets you:

1. Build some extensible software concisely
2. Exploit an intuitive analogy between interaction of physical entities and interaction of software pieces

It also:

- Raises tricky semantic and style issues worthy of careful PL study
- Is more complicated than functions
  - Does *not necessarily mean* it's worse

So what is OOP?

OOP "looks like this" pseudocode, but what is the *essence*?

```
class Pt1 extends Object {
  int x;
  int  get_x() { x }
  unit set_x(int y) { self.x = y }
  int  distance(Pt1 p) { p.get_x() – self.get_x() }
  constructor() { x = 0 }
}
class Pt2 extends Pt1 {
  int y;
  int get_y() { y }
  int get_x() { 34 + super.get_x() }
  constructor() { super(); y = 0 }
}
```


Class-based OOP

In (pure) *class-based OOP:*

1. Every value is an *object*
2. Objects communicate via *messages* (handled by *methods*)
3. Objects have their own [private] *state*
4. Every object is an instance of a *class*
5. A class describes its instances' behavior

Pure OOP

- Can make "everything an object" (cf. Smalltalk, Ruby, …)
  - Just like "everything a function" or "everything a string" or …

```
class True extends Boolean {
  myIf(x,y) { x.m() }
}
class False extends Boolean {
  myIf(x,y) { y.m() }
}

e.myIf((new Object() { m() {…}),
       (new Object() { m() {…}))
```

- Essentially *identical* to the lambda-calculus encoding of Booleans
  - Closures are just objects with one method, perhaps called "apply", and a private field for the environment

OOP can mean many things

Why is this *approach* such a popular way to structure software?

- Implicit **`self`**/**`this`** ?
- An ADT (private fields)?
- Inheritance: method/field extension, method override?
- Dynamic dispatch?
- Subtyping? [will do types *after* the rest, like earlier in course]
- All the above (plus constructor(s)) in one (class) definition

Design question: Better to have small orthogonal features or one "do it all" feature?

Anyway, let's consider how "unique to OO" each is…

OOP as ADT-focused

Fields, methods, constructors often have *visibilities*

What code can invoke a member/access a field?

- Methods of the same object?
- Methods defined in same class?
- Methods defined in a subclass?
- Methods in another "boundary" (package, assembly, friend, …)
- Methods defined anywhere?

Hiding concrete representation matters, in any paradigm

- For simple examples, objects or modules work fine
- But OOP struggles with *binary methods*…

Simple Example

```
type int_stack
val empty : int_stack
val push : int ->
           int_stack ->
           int_stack

push 42 empty
```

```
class IntStack {
 … // fields
  int  push(Int i) {…}
  constructor() { …}
  …
}
new IntStack().push(42);
```


Binary-Method Example

A "bag" supporting *"choose" an element uniformly at random*

```
type choose_bag
val single : int ->
              choose_bag
val union : choose_bag ->
            choose_bag ->
            choose_bag
val choose : choose_bag ->
              int
```

```
class ChooseBag {
 … // fields
 constructor(Int i){…}
 ChooseBag union
     (ChooseBag that){…}

 Int choose()  {…}
```

- Various ML implementations work fine (e.g., use an **`int list`**)
- Pure OOP implementation with private-to-object fields *impossible*
  - Fix: widen the interface (although clients shouldn't use it)

Inheritance & override

Subclasses:

- *Inherit* superclass' members
- Can *override* methods
- Can use **`super`** calls

Can we code this up in OCaml/F#/Haskell?

- No because of field-name reuse and lack of subtyping
  - But ignoring that we can get *close*…

(More than) records of functions

If OOP = functions + private state, we already have it

- But it's more (e.g., inheritance)

```
type pt1 = {get_x    : unit -> int;
            set_x    : int  -> unit;
            distance : pt1  -> int}
let pt1_constructor () =
  let x = ref 0 in
  let rec self = {
   get_x    = (fun() -> !x);
   set_x    = (fun y -> x := y);
   distance = (fun p -> p.get_x() +self.get_x())
  } in
  self
```

Almost OOP?

```
let pt1_constructor () =
  let x = ref 0 in
  let rec self = {
    get_x    = (fun() -> !x);
    set_x    = (fun y -> x := y);
    distance = (fun p -> p.get_x()+self.get_x())
  } in self
(* note: field reuse precludes type-checking *)
let pt2_constructor () = (* extends Pt1 *)
  let r = pt1_constructor () in
  let y = ref 0 in
  let rec self = {
    get_x    = (fun() -> 34 + r.get_x());
    set_x    = r.set_x;
    distance = r.distance;
    get_y    = (fun() -> !y);
  } in self
```


Problems

Small problems:

- Have to change `pt2_constructor` whenever `pt1_constructor` changes
- But OOPs have tons of "fragile base class" issues too
  - Motivates C#'s version support
- No direct access to "private fields" of superclass

Big problem:

- Distance method in a `pt2` doesn't behave how it does in OOP
- We do not have late-binding of `self` (i.e., dynamic dispatch)

The essence

Claims so far:

Class-based objects are:

- So-so ADTs
- Some syntactic sugar for extension and override

And:

- The essence of OOP (versus records of closures) is a fundamentally different rule for what `self` maps to in the environment

More on late-binding

Late-binding, dynamic-dispatch, and open-recursion are all essentially synonyms

The simplest example I know:

```
let c1 () =
  let rec r = {
    even = (fun i -> i=0  || r.odd (i-1));
    odd  = (fun i -> i<>0 && r.even (i-1))
  } in r

let c2 () =
  let r1 = c1 () in
  let rec r = {
    even = r1.even; (* still O(n) *)
    odd  = (fun i -> i % 2 == 1)
  } in r
```


More on late-binding

Late-binding, dynamic-dispatch, and open-recursion all related issues (nearly synonyms)

The simplest example I know:

```
class C1 {
  int even(int i) { i=0  || odd  (i-1)) }
  int odd(int i)  { i!=0 && even (i-1)) }
}

class C2 extends C1 {
  // even is now O(1)
  int odd(int i) {i % 2 == 1}
}
```


The big debate

Open recursion:

- Code reuse: improve `even` by just changing `odd`
- Superclass has to do less extensibility planning

Closed recursion:

- Code abuse: break `even` by just breaking `odd`
- Superclass has to do more abstraction planning

Reality: Both have proved very useful; should probably just argue over "the right default"

Our plan

- Dynamic dispatch is the essence of OOP
- How can we define/implement dynamic dispatch?
  - Basics, not super-optimized versions (see P501)
- How do we use/misuse overriding?
  - Functional vs. OOP extensibility
  - Revenge of binary methods
- Types for objects
  - Our prior study of subtyping mostly suffices
  - Subclasses vs. subtypes

Defining dispatch

Methods "compile down" to functions taking **`self`** as an extra argument
  - Just need **`self`** bound to "the right thing"

Approach #1:
- Each object has 1 "code pointer" per method
- For **`new C()`** where C extends D:
  - Start with code pointers for D (recursive definition!)
  - If C adds m, add code pointer for m
  - If C overrides m, change code pointer for m
- **`self`** bound to the (whole) object in method body

Defining dispatch

Methods "compile down" to functions taking self as an extra argument
  - Just need **`self`** bound to "the right thing"

Approach #2:
- Each object has 1 run-time tag
- For new C() where C extends D:
  - Tag is C
- **`self`** bound to the object
- Method call to m reads tag, looks up (tag,m) in a global table

Which approach?

- The two approaches are very similar
  - Just trade space for time via indirection
- vtable pointers are a fast encoding of approach #2
- This "definition" is low-level, but with overriding, simpler models are probably wrong

Our plan

- Dynamic dispatch is the essence of OOP
- How can we define/implement dynamic dispatch?
  - Basics, not super-optimized versions (see P501)
- How do we use/misuse overriding?
  - Functional vs. OOP extensibility
  - Revenge of binary methods
- Types for objects
  - Our prior study of subtyping mostly suffices
  - Subclasses vs. subtypes

Overriding and hierarchy design

- Subclass writer decides what to override to modify behavior
  - Often-claimed, unchecked style issue: overriding should *specialize behavior*
- But superclass writer typically knows what will be overridden
- Leads to notion of abstract methods (must-override)
  - Classes w/ abstract methods can't be instantiated
  - Does not add expressiveness
  - Adds a static check
  - C++ calls this "pure virtual"

Overriding for extensibility

```
class Exp { // a PL example; constructors omitted
 abstract Exp interp(Env);
 abstract Typ typecheck(Ctxt);
 abstract Int toInt();
}
class IntExp extends Exp {
 Int i;
 Value interp(Env e) { self }
 Typ typecheck(Ctxt c) { new IntTyp() }
 Int toInt() { i }
}
class AddExp extends Exp {
 Exp e1; Exp e2;
 Value interp(Env e) {
   new IntExp(e1.interp(e).toInt().add(
              e2.interp(e).toInt())) }
 Int toInt() { throw new BadCall() }
 // typecheck on next page
}
```


Example cont'd

- We did addition with "pure objects"
  - Int has a binary add method
- To do **`AddExp::typecheck`** the same way, assume equals is defined appropriately (structural on **`Typ`**):

```
Type typecheck(Ctxt c) {
  e1.typecheck(c).equals(new IntTyp()).ifThenElse(
  e2.typecheck(c).equals(new IntTyp()).ifThenElse(
  (fun () -> new IntTyp()),
  (fun () -> throw new TypeError())),
  (fun () -> throw new TypeError()))
}
```

- Pure "OOP" avoids **`instanceof IntTyp`** and if-statements

More extension

- Now suppose we want **`MultExp`**
  - No change to existing code, unlike OCaml!
  - In OCaml, can "prepare" with "Else of 'a" constructor [not shown]
- Now suppose we want a **`toString`** method
  - Must change all existing classes, unlike OCaml!
  - In OOP, can "prepare" with a "Visitor pattern" [not shown]
- Extensibility has many dimensions – most require forethought!

The Grid

- You know it's an important idea if I take the time to draw a picture ☺

	interp	typecheck	toString	…
IntExp	Code	Code	Code	Code
AddExp	Code	Code	Code	Code
MultExp	Code	Code	Code	Code
…	Code	Code	Code	Code

1 new class

1 new function

Back to MultExp

- Even in OOP, **`MultExp`** is easy to add, but you'll *copy* the typecheck method of **`AddExp`**
- Or maybe **`MultExp`** extends **`AddExp`**, but that's a *kludge*
- Or maybe *refactor* into **`BinaryExp`** with subclasses **`AddExp`** and **`MultExp`**
  - So much for not changing existing code
  - Awfully heavyweight approach to a helper function

Our plan

- Dynamic dispatch is the essence of OOP
- How can we define/implement dynamic dispatch?
  - Basics, not super-optimized versions (see P501)
- How do we use/misuse overriding?
  - Functional vs. OOP extensibility
  - Revenge of binary methods
- Types for objects
  - Our prior study of subtyping mostly suffices
  - Subclasses vs. subtypes

The equals mess

- *Equals* is very common and important (cf. Java, C#, …)
- But it's a binary method and does not work well when combined with subclassing and overriding
- Summarize an hour-long lecture (!!) in a sophomore-level course* (CSE331) in the next 5 minutes…
- [Focus on Java, which I know better]

*It's *not* the == vs. .equals lecture – that's in an earlier course

Acknowledgments for slides 31-36: CSE331 instructors, particularly Michael D. Ernst

How equals should behave

Documented *contract* for subclasses of class Object is sensible: "reflexive, symmetric, transitive" [and more, not shown here]

Reflexive `a.equals(a) == true`
- Confusing if an object does not equal itself

Symmetric `a.equals(b)` ⇔ `b.equals(a)`
- Confusing if order-of-arguments matters

Transitive `a.equals(b)` ∧ `b.equals(c)` ⇒ `a.equals(c)`
- Confusing again to violate centuries of logical reasoning

`Object.equals` method

```
public class Object {
  public boolean equals(Object o) {
    return this == o;
  }
  …
}
```

- Implements reference equality
- Subclasses can override to implement a different equality
- But library includes a *contract* **`equals`** should satisfy
  - Reference equality satisfies it
  - So should *any* overriding implementation
  - Balances flexibility in notion-implemented and what-clients-can-assume even in presence of overriding

Correct overriding

```
public class Duration {
   public int min, sec;
   public boolean equals(Object o) {
      if(! o instanceof Duration)
        return false;
      Duration d = (Duration) o;
       return this.min==d.min && this.sec==d.sec;
    }
}
```

- Reflexive: Yes
- Symmetric: Yes, even if **`o`** is not a **`Duration`**!
  - (Assuming **`o`**'s **`equals`** method satisfies the contract)
- Transitive: Yes, similar reasoning to symmetric

But then you are stuck

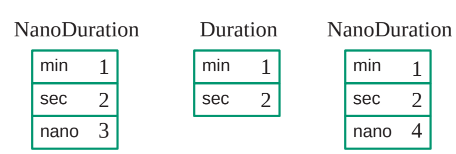
- Only "correct" for the contract approach below is "ignore nanoseconds", which is probably not what you want

```
class NanoDuration extends Duration {
  public int nano;
  public NanoDuration(int min, int sec, int nano){
    super(min,sec);
    this.nano = nano;
  }
  public boolean equals(Object o) { ????? }
  …
}
```

- Any use of nanoseconds breaks symmetry or transitivity or both
  - When comparing a mix of Duration and NanoDuration
- Can change Duration's equals to be "false" for any subclass of Duration, but that's not what you want [for other subclasses]

The gotchas

```
Duration d1 = new NanoDuration(1, 2, 3);
Duration d2 = new Duration(1, 2);
Duration d3 = new NanoDuration(1, 2, 4);
d1.equals(d2);
d2.equals(d3);
d1.equals(d3);
```



Haskell's Eq

- The Eq typeclass in Haskell has no such issues because it is about polymorphism and overloading, *not* about subclassing
- **`(==) :: Eq a => a -> a -> Bool`**
- For example, the **`String`** instance provides a function
  `(==) :: String -> String -> Bool`
- You can (and probably should) program this way in OOP
  - Recall "explicit dictionary"
  - C++ says "functors" others say "function objects" or add "good old lambdas"
  - Caller passes in an **`a -> a -> Bool`**

Our plan

- Dynamic dispatch is the essence of OOP
- How can we define/implement dynamic dispatch?
  - Basics, not super-optimized versions (see P501)
- How do we use/misuse overriding?
  - Functional vs. OOP extensibility
  - Revenge of binary methods
- Types for objects
  - Our prior study of subtyping *mostly* suffices
  - Subclasses vs. subtypes

Typechecking

Remember "my religion":

To talk about types, first discuss "what are we preventing"

1. In pure OOP, stuck if we need to interpret **`v.m(v1,…,vn)`** and **`v`** has no **`m`** method (taking *n* args)
   - "No such method" error
2. Also if ambiguous: multiple methods with same name and there is no "best choice"
   - "No best match" error
   - Arises with static overloading and multimethods [omitted]

Subtyping

Most class-based OOP languages purposely "confuse" classes & types
- If C is a class, then C is a type
- If C extends D (via declaration) then C $\leq$ D
- Subtyping is reflexive and transitive

Novel subtyping?
- New members in C "just" width subtyping
- "Nominal" (by name) instead of structural
- What about override…

Subtyping, continued

- If C extends D, overriding m, what do we need:
  - Arguments contravariant (assume less)
  - Result covariant (provide more)
- Many "real" languages are more restrictive
  - Often in favor of static overloading
- Some languages (e.g., Eiffel, TypeScript) try to be more flexible
  - At expense of run-time checks/casts

Good we studied this in a simpler setting!
- Little new to say – just "records of [immutable] methods"

The One Difference

- In the subclass' override, the method can soundly assume **`self`** is an instance of the subclass

```
class A {
  Int m1() { 42 }
}
class B extends A {
  Int x;
  Int m2() { 73 }
  Int m1() { x + m2() }
}
```

- So **`self`** is like "an implicit argument" but unlike the other arguments it is covariant
- This is sound because callers cannot "choose what **`self`** is"
  - If they could, they could cast to supertype and pass a **`self`** that is an instance of the supertype
- This "special treatment of " is *exactly* why trying to "do OOP" in a statically typed language without OOP support works poorly

Subtyping vs. subclassing

- Often convenient confusion: C a subtype of D if and only if C a subclass of D
- But more subtypes are sound
  - If A has every field and method that B has (at appropriate types), then subsume B to A
  - Java-style interfaces help, but require explicit annotation
- And fewer subtypes could allow more code reuse…

Non-subtyping example

`Pt2` ≤ `Pt1` is unsound here:

```
class Pt1 extends Object {
 int x;
 int  get_x() { x }
 bool compare(Pt1 p){ p.get_x() == self.get_x() }
}
class Pt2 extends Pt1 {
 int y;
 int  get_y() { y }
 bool compare(Pt2 p) { // override
      p.get_x() == self.get_x()
   && p.get_y() == self.get_y() }
}
```


What happened

- Could inherit code without being a subtype
- Cannot always do this
  - what if `get_x` called `self.compare` with a `Pt1`

  Possible solutions:
  - Re-typecheck `get_x` in subclass
  - Use a really fancy type system
  - Don't override `compare`
- Moral: Not suggesting "subclassing not subtyping" is useful, but the *concepts* of inheritance and subtyping are orthogonal

Now what?

- That's basic class-based OOP
  - Note: Not all OOPLs use classes (Javascript, Self, Cecil, …)
- Now I'd love to do some "fancy" stuff…
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

  … but we are out of time for the quarter! ☺ ☹
  … so let's wrap-up…

Victory Lap

A victory lap is an extra trip around the track
- By the exhausted victors (us) ☺

Review course goals
- Slides from Introduction and Course-Motivation

Some big themes and perspectives
- Stuff for five years from now more than for the final

Do your course evaluations!!!

Thanks!

- To you! (On top of your day jobs!)
- To John! (On top of your research!)
- To "Caryl and the kids who managed 9 bedtimes without me" ☺

Course [incomplete] summary

- Functional programming, datatypes, modularity, etc.
- Defining languages is hard but worth it
  - Interpretation vs. translation
  - Inference rules vs. a PL for the metalanguage
- Features we investigated
  - Mutable variables (and loops)
  - Higher-order functions, scope
  - Pairs and sums
  - Continuations
  - Monads
  - Typeclasses
  - Objects
- Types restrict programs (often a good thing (!) then counterbalanced via flavors of polymorphism)

[Now a few slides unedited from Lecture 1 that probably make a lot more sense now]

OCaml

- OCaml is an awesome, high-level language
- We'll use a small core subset that is well-suited to manipulating recursive data structures (like programs)
- Tutorial will demonstrate its *mostly functional* nature
  - Most data immutable
  - Recursion instead of loops
  - Lots of passing/returning functions
- Again, will support F# as a fine alternative

Last Motivation: "Fan Mail"

This class has changed the way I think about programming - even if I don't get to use all of the concepts we explored in OCaml (I work in C++ most of the time), understanding more of the theory makes a tremendous difference to how I go about solving a problem.

Picking a language

Admittedly, semantics can be far down the priority list:

- What libraries are available?
- What do management, clients want?
- What is the de facto industry standard?
- What does my team already know?
- Who will I be able to recruit?

But:

- Nice thing about class: we get to ignore all that ☺
- Technology *leaders* affect the answers
- Sound reasoning about programs *requires* semantics
  - Mission-critical code doesn't "seem to be right"
  - Blame: the compiler vendor or you?

Academic languages

Aren't academic languages worthless?

- Yes: fewer jobs, less tool support, etc.
  - But a lot has changed in the last decade
- No:
  - Knowing them makes you a better programmer
  - Java did not exist in 1993; what doesn't exist now
  - Eventual vindication (on the leading edge): garbage-collection, generics, function closures, iterators, universal data format, … (what's next?)
  - We don't conquer; we assimilate
    - And get no credit (fine by me)
  - Functional programming is "finally cool"-ish

“But I don’t do languages”

Aren’t languages somebody else’s problem?

- If you design an *extensible* software system or a *non-trivial API*, you'll end up designing a (small?) programming language!
- Another view: A language is an API with few functions but sophisticated data. Conversely, an interface is just a stupid programming language…

[Now 1.5 more slides]

Penultimate slide

- We largely avoided:
  - Subjective non-science (“I like curly braces”)
  - Real-world issues (“cool libraries / tricks in language X”)
- Focused on:
  - Concepts that almost every language has, including the next fad that doesn’t exist yet
  - Connections (objects and closures are different, but not totally different)
  - Reference implementations, not fast or industrial-strength ones
  - “Cool stuff” (e.g., Curry-Howard, laziness, …)

Questions?

Questions?

About PL, the exam, life, etc.?

[Oh, and reminder: do your course evaluation by Sunday midnight!]