Objects

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

You should be able to ...

In this lecture, we extend the idea of local state from last time to create a simple implementation of objects and discuss its limitations. We will also show the message dispatch model of objects, which allows for inheritance and virtual functions.
Your objectives:

- Be able to explain what an object is.
- Implement an object using records and HOFs.
- Implement an object using a message dispatcher.

Preliminaries

- We will use the following functions during our discussion:

```
let pi1 (x,y) = x
let pi2 (x,y) = y
let report (x,y) = print_string "Point: ";
                   print_int x;
                   print_string ",";
                   print_int y;
                   print_newline ()
let movept (x,y) (dx,dy) = (x+dx,y+dy)
```


Point

Here is an example of a point using local state.

```
let mktPoint myloc =
  let myloc = ref myloc in
    ( myloc,
      (fun () -> pi1 !myloc),
      (fun () -> pi2 !myloc),
      (fun () -> report !myloc),
      (fun dl -> myloc := movept !myloc dl) )
```

- This defines a tuple of functions that share a common state.
- It is cumbersome to use.

```
let (lref,getx,gety,show,move) = mktPoint (2,4);;
```

Improvement: Use Records

```
type point = {
  loc : (int * int) ref;  getx : unit -> int;
  gety : unit -> int;  draw : unit -> unit;
  move : int * int -> unit;
}
let mkrPoint newloc =
   let myloc = ref newloc in
   { loc = myloc;
     getx = (fun () -> pi1 !myloc);
     gety = (fun () -> pi2 !myloc);
     draw = (fun () -> report !myloc);
     move = (fun dl -> myloc := movept !myloc dl)}
```


Adding Self

By the way, this lecture is really about recursion.

```
let mkPoint newloc =
  let rec this =
  { loc = ref newloc;
    getx = (fun () -> pi1 !(this.loc));
    gety = (fun () -> pi2 !(this.loc));
    draw = (fun () -> report !(this.loc));
    move = (fun dl ->
         this.loc := movept !(this.loc) dl)}
 in this;;
```

We can store “this” explicitly in the record if we want.

Message Dispatching

Last time we said that an object is a kind of data that can *receive messages* from the program or other objects.

- Q: How do we normally represent messages?
- A: With strings!

Let a point object be a function that takes a string and returns an appropriate function matching that string.

- Question: Suppose p is our point object. What will be its type?

mkPoint

```
let mkPoint x y =
  let x = ref x in
  let y = ref y in
  fun st ->
    match st with
    | "getx" -> (fun _ -> !x)
    | "gety" -> (fun _ -> !y)
    | "movx" -> (fun nx -> x := !x + nx; nx)
    | "movy" -> (fun ny -> y := !y + ny; ny)
    |  _ -> raise (Failure "Unknown message.")
```

All methods now have to have type `int -> int`.

Subclassing

- Warmup exercise: How would we add a `report` method?
- Another one: How would we add `this` support?

Let's say we want a `fastpoint`, which moves twice as fast as the original point. What does it mean for `fastpoint` to be a *subclass* of `point`?

- `fastpoint` should respond to the same messages.
  - It may override some of them.
  - It may add its own.
  - It may **not** remove any methods.
- The `fastpoint` object will need access to some of the data in `point`.

Implementing

- Two entities involved: the superclass (`point`) and the subclass (`fastpoint`)
- `fastpoint` needs to create an instance of `point`.
- `point` construction needs to return the "public" data to `fastpoint`.
- `fastpoint` returns a dispatcher:
  - If the `fastpoint` dispatcher can handle a message, it does.
  - Otherwise, it *sends the message* to `point`.

Code: `point`

```
let mkSuperPoint x y =
  let x = ref x in
  let y = ref y in
  ((x,y),  (* This part returns the local state *)
  fun st ->
    match st with
    | "getx" -> (fun _ -> !x)
    | "gety" -> (fun _ -> !y)
    | "movx" -> (fun nx -> x := !x + nx; nx)
    | "movy" -> (fun ny -> y := !y + ny; ny)
    |  _ -> raise (Failure "Unknown message."));;
val mkSuperPoint : int -> int ->
   (int ref * int ref) * (string -> int -> int) = <fun>
```


Code: `fastpoint`

```
let mkFastpoint x y =
  let ((x,y),super) = mkSuperPoint x y in
  fun st ->
    match st with
    | "movx" -> (fun nx -> x := !x + 2 * nx; nx)
    | "movy" -> (fun ny -> y := !y + 2 * ny; ny)
    |  _ -> super st;;
```

- This technique is flexible; we can add methods very easily.
- But it's also slow. Imagine if we had a chain of 20 classes!

C++

- Methods and variables are kept in a table: a fixed location.
- "`this`" is an implicit argument, allowing only one copy of the function to be needed.
- Virtual methods are kept in a *vtable*, which counts as local data.

Local data for `point` or `fastpoint`:

`x`	*value of x*
`y`	*value of y*
`vtable`	*pointer to vtable*

Vtable for `point`:

`movx`	*pointer to point.movx*
`movy`	*pointer to point.movy*

(`fastpoint` vtable is similar.) `getx`, etc. is static.

Discussion

- Other languages (i.e., smalltalk) use a technique very similar to this one.
- Java uses the "every object is of type `Object`" technique.
- A strong type system makes it somewhat cumbersome to simulate objects. You either have to:
  - define a new type to encompass all objects, or
  - force all methods to have the same type.
- Important concept: *polymorphism* — when functions can operate on multiple types. (This is different than *overloading* — when multiple functions exist with the same name, but different inputs.)