Objects

object: to feel distaste for something
– Webster’s Dictionary

Prof. Clarkson
Fall 2017

Today’s music: Kung Fu Fighting by CeeLo Green
Review

Currently in 3110: Advanced topics
• Futures
• Monads

Today:
• What is an object?
• Implement/encode objects in OCaml
Question: What is an object?

A. Objects are entities that combine state, behavior, and identity.
B. Objects have state and behavior.
C. Objects encapsulate data and operations.
D. An object is a data structure encapsulating some internal state and offering access to this state to clients with a collection of methods.
E. None of the above
Question: What is an object?

A. Objects are entities that combine state, behavior, and identity. [Wikipedia]
B. Objects have state and behavior. [Oracle]
C. Objects encapsulate data and operations. [Carrano & Prichard]
D. An object is a data structure encapsulating some internal state and offering access to this state to clients with a collection of methods. [Pierce]
E. None of the above
What are key features of OOP?

1. Encapsulation
2. Subtyping
3. Inheritance
4. Dynamic dispatch
   • (Classes?)
   • ...
1. Encapsulation

- Object has *internal state*
- Object's *methods* can inspect and modify that state
- Clients cannot directly access state except through methods
2. Subtyping

- *Type* of an object involves the names and types of its methods
- Object of type t can be used in place of an object of type t' if t is a *subtype* of t'
- Subtyping depends on names and types of methods
3. Inheritance

- Objects *inherit* some of their behavior
- Usually, behavior associated with *classes* – templates from which objects can be constructed
- *Subclassing* derives new classes from old classes – add new methods – *override* implementations of old methods – inherit other old methods
4. Dynamic dispatch

- Recall: when invoking `toString` you always get the "right" implementation
- Method that is invoked on an object is determined at run-time rather than at compile-time
  - $dynamic = run-time$
  - $dispatch = invocation$
- Special keyword: `this` or `self`
  - Always in scope inside a method
  - Always bound to the receiving object of a method invocation
Object encoding

• **Rest of this lecture:** encode objects in OCaml
• **Purpose:** *understand* OOP features better by approximating them in OCaml
• **Non-purpose:** *exactly* model Java objects in all their rich details
• **Non-purpose:** use the OCaml object system to mimic Java objects
class Counter {
    protected int x = 0;
    public int get() { return x; }
    public void inc() { x++; }
}
1. ENCAPSULATION
Objects as records

• A Java object is a collection of named values
• An OCaml record is also a collection of named values
• So we could try something like:

```java
{ x = 0;
  get = ...;
  set = ...;
}
```

• But that would fail to provide encapsulation of x
Encapsulation of private state

• Idea: use let-binding to hide the state

```ocaml
let x = ref 0 in {
    get = (fun () -> !x);
    inc = (fun () -> x := !x+1);
}
```

• A closure is created for each "method"
  – Closure has x in its environment
  – Protected "field" is hidden by the let-binding
  – Record exposes only the "methods"
Object type

- Type of the object we just created:
  ```
  type counter = {
    get : unit -> int;
    inc : unit -> unit;
  }
  ```
- Note: x is not exposed in type
Method invocation

• Given an "object":

```plaintext
let c : counter =
    let x = ref 0 in {
        let get = (fun () -> !x);
        let inc = (fun () -> x := !x+1);
    }
```

• We can invoke "methods" with field accesses:

```plaintext
c.inc(); c.inc(); c.get()
```

• Note: the parens are the unit value
Functions with objects

• OCaml functions can manipulate objects:
  ```ocaml
  let inc3 (c:counter) =
  c.inc(); c.inc(); c.inc()
  ```

• OCaml functions can construct new objects:
  ```ocaml
  let new_counter = fun () ->
      let x = ref 0 in {
        let get = (fun () -> !x); in
        let inc = (fun () -> x := !x+1); in
      }
  let c = new_counter()
  let one = c.inc(); c.get()
  ```
2. SUBTYPING
Subtype of Counter

class ResetCounter extends Counter {
    public void reset() {
        x = 0;
    }
}
Direct encoding of ResetCounter

type reset_counter = {
    get : unit -> int;
    inc : unit -> unit;
    reset : unit -> unit;
}

let new_reset_counter () =
  let x = ref 0 in {
    get = (fun () -> !x);
    inc = (fun () -> x:=!x+1);
    reset = (fun () -> x:=0);
  }

we're duplicating code from new_counter :-(
let's come back to that
Call function with a subtype

```plaintext
let rc = new_reset_counter()
inc3 rc (* won't work! wrong arg type *)

let counter__of__reset_counter
  (rc : reset_counter) : counter =
{
  get = rc.get;
  inc = rc.inc;
}
inc3 (counter__of__reset_counter rc)
```
Explicit coercion

• Upcast: use an explicit function call to *coerce* value of subtype into value of supertype
  – This is an actual compilation technique used in some high-performance compilers

• Digression:
  – Wouldn't be needed if OCaml supported subtyping on records
  – Basic idea: `{x:int; y:int}` can be used wherever `{x:int}` is expected
    – aka *row polymorphism*
  – Problem: efficient implementation; can't just compile records into tuples

• Downcasts: would require run-time type information and a more painful low-level encoding than we have here
3. INHERITANCE
Duplicated code

• **Problem:** duplicated code between objects
• **Solution:** classes

• What is a *class*?
  Data structure holding methods. Can be:
  • *instantiated* to yield a new object
  • *extended* to yield a new class

• We want to reuse method code when possible
  ...even if the representation of internal state changes
  ...let's *parameterize on representation type*
type counter_rep = {
    x : int ref;
}

let counter_class = fun (r:counter_rep) -> {
    get = (fun () -> !(r.x));
    inc = (fun () -> (r.x := !(r.x) + 1));
}

let new_counter () =
    let r = {x = ref 0} in
    counter_class r
What is a class?

• A function
  – from internal rep of object state
  – to record of methods, all of which use that shared state
  – i.e., a way of generating related objects

• Not a type!
  – Many languages conflate types and classes
ResetCounter with inheritance

let reset_counter_class =
  fun (r:counter_rep) ->
    let super = counter_class r in {
      get = super.get;
      inc = super.inc;
      reset = (fun () -> r.x := 0)
    }

let new_reset_counter () =
  let r = {x=ref 0} in
  reset_counter_class r
Implementing inheritance: Code

reset_counter_class

– first creates an object of the superclass with the same internal state as its own
– the resulting parent object is bound to super
– then creates a new object with same internal state
– copies (inherits) the implementations of get and inc from superclass
– provides its own implementation of new methods
Another subtype of Counter

class BackupCounter extends ResetCounter {
    protected int b = 0;
    public void backup() { b = x; }
    public void reset() { x = b; }
}

...adds method and a new field

...overrides one method
BackupCounter with inheritance

```ocaml
type backup_counter = {
  get : unit -> int;
  inc : unit -> unit;
  reset : unit -> unit;
  backup : unit -> unit
}

type backup_counter_rep = {
  x : int ref;
  b : int ref;
}
```
Class for BackupCounter

```plaintext
let backup_counter_class (r : backup_counter_rep) =
  let super =
    reset_counter_class
    (counter_rep__of__backup_counter_rep r)
  in {
    get = super.get;
    inc = super.inc;
    reset = (fun () -> r.x := !(r.b));
    backup = (fun () -> r.b := !(r.x));
  }
```
Class for BackupCounter

let new_backup_counter () =
    let r = {x = ref 0; b = ref 0} in
    backup_counter_class r

let counter_rep__of__backup_counter_rep (r : backup_counter_rep) = {
    x = r.x;
}
CONCLUSION, AND A STORY
Closures vs. Objects

• We encoded objects in OCaml
  – Closures (i.e., first-class functions) were an essential part of that encoding
• In Java, closures can be encoded with objects
  – Evidence: in 2110 you might have seen that inner classes (like adapters for GUI buttons) capture variables from an outer scope
• For more discussion, see http://wiki.c2.com/?ClosuresAndObjectsAreEquivalent
Master, I have heard that objects are a very good thing. Is this true?

Foolish pupil. Objects are merely a pitiable substitute for closures.
Master, I have diligently studied the matter, and now understand that objects are truly a pitiable substitute for closures.
When will you learn? Closures are merely a pitiable substitute for objects.
Upcoming events

• [Wed] A5 due

*This is enlightening.*

**THIS IS 3110**