The Stack Class

class Stack {
    class Entry {
        Object elt; Entry next;
        Entry(Object e, Entry n) { elt = e; next = n; }
    }
    private Entry theStack;
    void push(Object e) {
        theStack = new Entry(e, theStack);
    }
    Object pop() {
        if (theStack == null) throw new NoSuchElementException();
        Object temp = theStack.elt;
        theStack = theStack.next;
        return temp;
    }
}

Writing a “Stack” in OCaml: Take 1

module type STACK =
    sig
        type 'a stack
        val new_stack : unit -> 'a stack
        val push : 'a stack -> 'a -> unit
        val pop : 'a stack -> 'a
    end

module Stack : STACK =
    struct
        type 'a stack = 'a list ref
        let new_stack () = ref []
        let push x = this := (x::!this) in
        let pop () = match !this with
            [] -> failwith "Empty stack"
        | (h::t) -> this := t; h
        in
            (push, pop)
    end

Writing a “Stack” in OCaml: Take 2

let new_stack () =
    let this = ref [] in
    let push x = this := (x::!this) in
    let pop () = match !this with
        [] -> failwith "Empty stack"
    | (h::t) -> this := t; h
    in
        (push, pop)

# let s = new_stack();;
val s : ('a -> unit) * (unit -> '_a) = (fun, fun)
# fst s 3;;
- : unit = ()
# snd s ();;
- : int = 3
Relating Objects and Closures

• An object...
  – Is a collection of fields (data)
  – ...and methods (code)
  – When a method is invoked, it is passed an implicit this parameter it can use to access fields

• A closure...
  – Is a pointer to an environment (data)
  – ...and a function body (code)
  – When a closure is invoked, it is passed its environment it can use to access variables

Encoding Objects with Lambda

• We can apply this transformation in general

\[
\text{class C}\{\text{f1 ... fn; m1 ... mn;}\}
\]

– becomes

\[
\text{let make () = } \\
\text{let f1 = ... in } \\
\text{... } \\
\text{let fn = ... in } \\
\text{(fun ... , (* body of m1 *))} \\
\text{... } \\
\text{fun ... , (* body of mn *))}
\]

– make () is like the constructor
  – the closure environment contains the fields

Recall a Useful Higher-Order Function

• Can we encode this in Java?

\[
\text{let rec map f = function} \\
\text{}[] -> [] \\
\text{}(h::t) -> (f h)::(map f t)
\]
A Map Method for Stack

• To write a map method, we need some way of passing a function into another function
  – We can do that with an object with a known method

  ```java
  public interface Function {
    Object eval(Object arg);
  }
  ```

A Map Method for Stack, cont’d

• Here are two classes which both implement this `Function` interface:

  ```java
  class AddOne implements Function {
    Object eval(Object arg) {
      Integer old = (Integer) arg;
      return new Integer(old.intValue() + 1);
    }
  }

  class MultTwo implements Function {
    Object eval(Object arg) {
      Integer old = (Integer) arg;
      return new Integer(old.intValue() * 2);
    }
  }
  ```

A Map Method for Stack, cont’d

class Stack {
  class Entry {
    Object elt; Entry next;
    Entry(Object x, Entry n) { elt = x; next = n; }
    Entry map(Function f) {
      if (next == null) return new Entry(f.apply(elt), null);
      else return new Entry(f.apply(elt), next.map(f));
    }
  }
  Entry theStack;

  Stack map(Function f) {
    Stack s = new Stack();
    s.theStack = theStack.map(f);
    return s;
  }
}

A Map Method for Stack, con’t.

• Then to apply the function, we just do

  ```java
  Stack s = ...;
  Stack t = s.map(new AddOne());
  Stack u = s.map(new MultTwo());
  ```

– We make a new object that has a method that performs the function we want
– This is sometimes called a `callback`, because `map` “calls back” to the object passed into it
– But it’s really just a higher-order function, written more awkwardly
Relating Closures and Objects

```java
let app f x = f x

fun b -> a + b

let add a b = a + b;
let f = add 3;
app f 4;
```

-Encoding Lambda with Objects

- We can apply this transformation in general
  ```java
  ...(fun x -> (* body of fn *)) ...
  let h f ... = ...f y...
  ```

- F is the interface to the callback
- G represents the particular function

Code as Data

- The key insight in all of these examples is to treat code as if it were data
  - Higher-order functions allow code to be passed around the program
  - As does object-oriented programming
- This is a powerful programming technique
  - And it can solve a number of problems quite elegantly
- Closures and objects are related
  - Both of them allow data to be associated with higher-order code as its passed around (but we can even get by without this)