# CMSC 330: Organization of Programming Languages

Functional Programming in Object-Oriented Languages

## Iteration

- Goal: Loop through all objects in an aggregate

```
class Node { Element elt; Node next; }
Node n = ...;
while (n != null) { ...; n = n.next; }
```

- Problems:
  - Depends on implementation details
  - Varies from one aggregate to another



## Iterators in Java

```
public interface Iterator<A> {
  // returns true if the iteration has more elements
  public boolean hasNext();

  // returns the next element in the iteration
  public A next() throws NoSuchElementException;
}
```

- Advantages
  - The implementation of iterators is not exposed
  - Generic for many different aggregates
  - Supports multiple traversal strategies
  - In Java, iterators can be used explicitly, or implicitly via the enhanced for (also called foreach) loop



## Writing an “Iterator” in OCaml

```
let iterate l =
  let cur_list = ref l in
  ((fun () -> (!cur_list) != []),
   (fun () ->
     let temp = List.hd (!cur_list) in
       cur_list := List.tl (!cur_list);
       temp))
```

```
# let iterate l = ... ;;
val iterate : 'a list -> (unit -> bool) * (unit -> 'a) = <fun>
# let (has_next, next) = iterate [1; 2; 3];;
val has_next : unit -> bool = <fun>
val next : unit -> int = <fun>
```

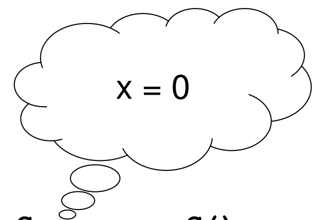
# Relating Objects and Closures

- 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



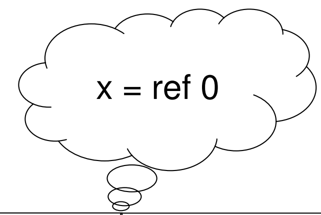
# Relating Objects and Closures (cont'd)

```
class C {
  int x = 0;
  void setX(int y) { x = y; }
  int getX() { return x; }
}
```




```
C c = new C();
c.setX(3);
int y = c.getX();
```

```
let make () =
 let x = ref 0 in
   ( (fun y -> x := y),
     (fun () -> !x) )
```




```
fun y -> x := y | fun () -> !x
```

```
let (set, get) = make ();;
set 3;;
let y = get ();;
```



# Encoding Objects in General

- We can apply this transformation in general

```
class C { f1 ... fm; m1 ... mn; }
```

  - becomes

```
let make () =
 let f1 = ... in
 ...
 let fm = ... in
 ( fun ... , (* body of m1 *)
   ...
   fun ...,  (* body of mn *)
 )
```

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



# Recall a Useful Higher-Order Function

```
let rec map f = function
  [] -> []
| (h::t) -> (f h)::(map f t)
```

- Can we encode these in Java?

# 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

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



# A Map Method for Stack, con't.

- Here are two classes which both implement this Function interface:

```
class AddOne implements Function {
  Object eval(Object arg) {
    return new Integer(((Integer) arg) + 1);
  }
}
```

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



# A Map Method for Stack, con't.

```
class Stack {  /* ignore type parameter for now... */
  ...
  private Entry theStack;
  Stack map(Function f) {
    Stack s = new Stack();
    map_helper(f, s, theStack);
    return s;
  }
  void map_helper(Function f, Stack s, Entry e) {
    if (e != null) {
      map_helper(f, s, e.next);
      s.push(f.eval(e.elt));
  }
}
```



# A Map Method for Stack, con't.

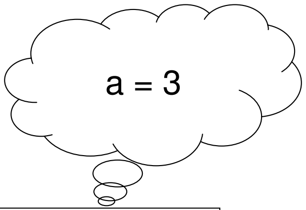
- Then to apply the function, we just do

```
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

```
let app f x = f x
```




```
fun b -> a + b
```

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

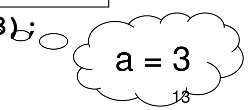
```
interface F {
  Object eval(Object y);
}
class C {
   static Object app(F f, Object x) {
     return f.eval(x);
   }
}
```

```
class G implements F {
  int a;

  G(int a) { this.a = a; }

  Object eval(Object y) {
    return a + (Integer) y;
  }
}
```

```
F adder = new G(3);
C.app(adder, 4);
```






# Encoding Functions with Objects

- We can apply this transformation in general

```
...(fun x -> (* body of f_n *)) ...
let h f ... = ...f y...
```

  - becomes

```
interface F { Object eval(Object x); }
class G implements F {
  Object eval(Object x) { /* body of f_n */ }
}
class C {
  Typ h(F f, ...) {
     ...f.eval(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)