CMSC 330: Organization of Programming Languages

Objects and Functional Programming

OOP vs. FP

- Object-oriented programming (OOP)
  - Computation as interactions between objects
  - Objects encapsulate state, which is usually mutable
    - Accessed / modified via object's public methods

- Functional programming (FP)
  - Computation as evaluation of functions
    - Mutable data discouraged; may be used to improve efficiency
  - Higher-order functions implemented as closures
    - Closure = function + environment

Relating Objects to Closures

- An object...
  - Is a collection of methods (code)
  - ...and fields (data)
  - When a method is invoked
    - an implicit `this` parameter is used to access object fields

- A closure...
  - Is a function body (code)
  - ...and an environment (data)
  - When a closure is invoked
    - the implicit environment is used to access (free) variables

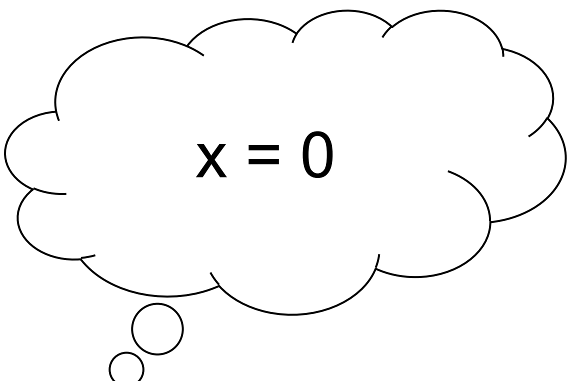
Relating Objects to Closures

Java

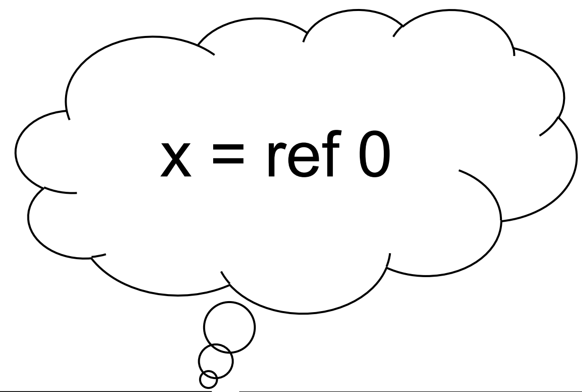
```java
class C {
  int x = 0;
  void set_x(int y) { x = y; }
  int get_x() { return x; }
}
```

OCaml

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




```java
C c = new C();
c.set_x(3);
int y = c.get_x();
```




```ocaml
fun y -> x := y
```

```ocaml
fun () -> !x
```

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

Encoding Objects with Closures

- We can apply this transformation in general

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

  - becomes

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

Tuple containing closures

(could use record instead)

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

Quiz 1: Is `Circle` Encoded Correctly?

```
class Circle {
  float r = 0;
  void set_r (float t) { r = t; }
  float get_r () { return r; }
  float area(){
        return 3.14 * r * r;}
}
```

```
C c = new Circle();
c.set_r(1.0);
float y = c.get_r();
c.area();
```

A. True
B. False

```
let make () =
 let r = 0.0 in
 ((fun y -> let r = y in ()),
  (fun () -> r),
  fun ()-> r *. r *. 3.14
 )
```

```
let (set_r, get_r, area) =
                    make ();;
set_r 1.0;;
let y = get_r();;
area();;
```

Quiz 1: Is `Circle` Encoded Correctly?

```
class Circle {
  float r = 0;
  void set_r (float t) { r = t; }
  float get_r () { return r; }
  float area(){
        return 3.14 * r * r;}
}
```

```
C c = new Circle();
c.set_r(1.0);
float y = c.get_r();
c.area();
```

A. True

B. False

```
let make () =
 let r = ref 0.0 in
 ((fun y -> let r := y in ()),
  (fun () -> !r),
  fun ()-> !r *. !r *. 3.14
 )
```

```
let (set_r, get_r, area) =
                     make ();;
set_r 1.0;;
let y = get_r();;
area();;
```

Relating Closures to Objects

- A closure is like an object with a designated **`eval()`** method
  - The type of **`eval`** corresponds to the type of the closure's function, **`T -> U`**

```
interface Func<T,U> {
  U eval(T x);
}
class G implements Func<T,U> {
  U eval(T x) { /* body of fn */ }
}
```

- Environment is stored as field(s) of **`G`**

Relating Closures to Objects

```
let add1 x = x + 1
```

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Add1 implements IntIntFun {
  Integer eval(Integer x) {
    return x + 1;
  }
}
```

```
add1 2;;
add1 3;;
```

```
new Add1().eval(2);
new Add1().eval(3)
```

Quiz 2: What does this evaluate to?

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Foo implements IntIntFun {
  Integer eval(Integer x) {
    return x * 2;
  }
}

new Foo(5);
```

A. 5
B. 10
C. 6
D. None of the above

Quiz 2: What does this evaluate to?

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Foo implements IntIntFun {
  Integer eval(Integer x) {
    return x * 2;
  }
}

new Foo(5);
```

A. 5
B. 10
C. 6
D. None of the above (should be called **`new Foo().eval(5)`**)

Relating Closures to Objects

```
let app_to_1 f = f 1
```

```
interface IntIntFunFun {
  Integer eval(IntIntFun x);
}
class AppToOne
  implements IntIntFunFun {
    Integer eval(IntIntFun f) {
      return f.eval(1);
  }
}
```

```
app_to_1 add1;;
```

```
new AppToOne().eval(new Add1());
```

Quiz 3: What does this evaluate to?

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Foo implements IntIntFun {
  Integer eval(Integer x) {
    return x * 2;
  }
}
interface IntIntFunFun {
  Integer eval(IntIntFun x);
}
class AppToFive
  implements IntIntFunFun {
    Integer eval(IntIntFun f) {
      return f.eval(5);
  }
}
```

```
new AppToFive().eval
              (new Foo());
```

A. 5
B. 10
C. 6
D. Error

Quiz 3: What does this evaluate to?

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Foo implements IntIntFun {
  Integer eval(Integer x) {
    return x * 2;
  }
}
interface IntIntFunFun {
  Integer eval(IntIntFun x);
}
class AppToFive
  implements IntIntFunFun {
    Integer eval(IntIntFun f) {
      return f.eval(5);
  }
}
```

```
new AppToFive().eval
              (new Foo());
```

A. 5
B. 10
C. 6
D. Error

Relating Closures to Objects

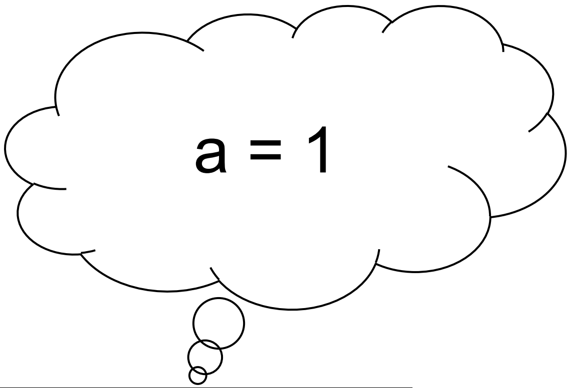
```
interface Func<T,U> {
  U eval(T x);
}
class Add1 implements Func<Integer,Integer> {
  public Integer eval(Integer x) {
    return x + 1;
  }
}
class AppToOne
  implements Func<Func<Integer,Integer>,Integer> {
    public Integer eval(Func<Integer,Integer> f) {
      return f.eval(1);
  }
}
```

```
app_to_1 add1;;
```

```
new AppToOne().eval(new Add1());
```

Relating Closures to Objects

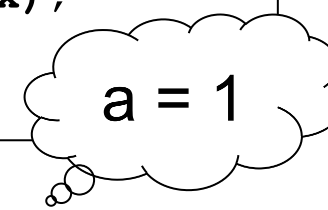
```
let add a b = a + b;;
```




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

```
let add1 = add 1;;
add1 4;;
```

```
class Add
  implements Func<Int,Func<Int,Int>> {
    private static class AddClosure
      implements Func<Int,Int> {
        private final Int a;
        AddClosure(Int a) {
          this.a = a;
        }
        Integer eval(Int b) {
          return a + b;
        }
    }
    Func<Int,Int> eval(Int x) {
      return new AddClosure(x);
    }
}
```




```
Func<Int,Int> add1 = new Add().eval(1);
add1.eval(4);
```

Encoding Closures with Objects

- We can apply this transformation in general

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

  - becomes

```
interface F<T,U> { U eval(T x); }
class G implements F<T,U> {
  U eval(T x) { /* body of fn */ }
}
class C {
  Typ1 h(F<Typ2,Typ3> f, ...) {
      ...f.eval(y)...
  }
}
```

  - F is the interface of a closure’s function
  - G represents the particular function

Quiz 4: Are these two versions equivalent?

```
let mult x y = x * y
let f = mult 2 in
f 3;;
```

A. True
B. False

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Mult implements IntIntFun {
  private int x;
  Mult(int x) { this.x = x }
  Integer eval(Integer y) {
    return x * y;
  }
}
Mult f = new Mult(2);
f.eval(3);
```

Quiz 4: Are these two versions equivalent?

```
let mult x y = x * y
let f = mult 2 in
f 3;;
```

A. True
B. False

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Mult implements IntIntFun {
  private int x;
  Mult(int x) { this.x = x }
  Integer eval(Integer y) {
    return x * y;
  }
}
Mult f = new Mult(2);
f.eval(3);
```

Java 8 Lambda Expressions

- Think this is a pain? The Java designers would agree!
  - So they introduced closures directly, in Java 8
- Writing **`x -> { … return e; }`** produces a closure, where **`x`** is the parameter, … is the body, and it concludes by returning **`e`**
  - If ... is empty, can just write e without return. For example, you can write: **`x -> x*2`**

Java 8 Closures

- Lambda expressions will produce closures
  - Free variables' values will be captured and stored in an environment

```
import java.util.function.Function;
public class Foo {
  public static
  Function<Integer,Integer> multby(Integer x) {
    return y -> x*y; // captures x's value
  }
  public static void main(String args[]) {
    Function<Integer,Integer> f = multby(3);
    System.out.println(f.apply(2));// prints 6
  }
}
```

Code as Data

- Closures and objects are related
  - Both of them allow
    - Data to be associated with higher-order code
    - Passing code around the program
- The key insight in all of these examples
  - Treat code as if it were data
    - Allowing code to be passed around the program
    - And invoked where it is needed (as callback)
- Approach depends on programming language
  - Higher-order functions (OCaml, Ruby, Lisp)
  - Function pointers (C, C++)
  - Objects with known methods (Java)

Code as Data

- This is a powerful programming technique
  - Solves a number of problems quite elegantly
    - Create new control structures (e.g., Ruby iterators)
    - Add operations to data structures (e.g., visitor pattern)
    - Event-driven programming (e.g., observer pattern)
  - Keeps code separate
    - Clean division between higher & lower-level code
  - Promotes code reuse
    - Lower-level code supports different callbacks