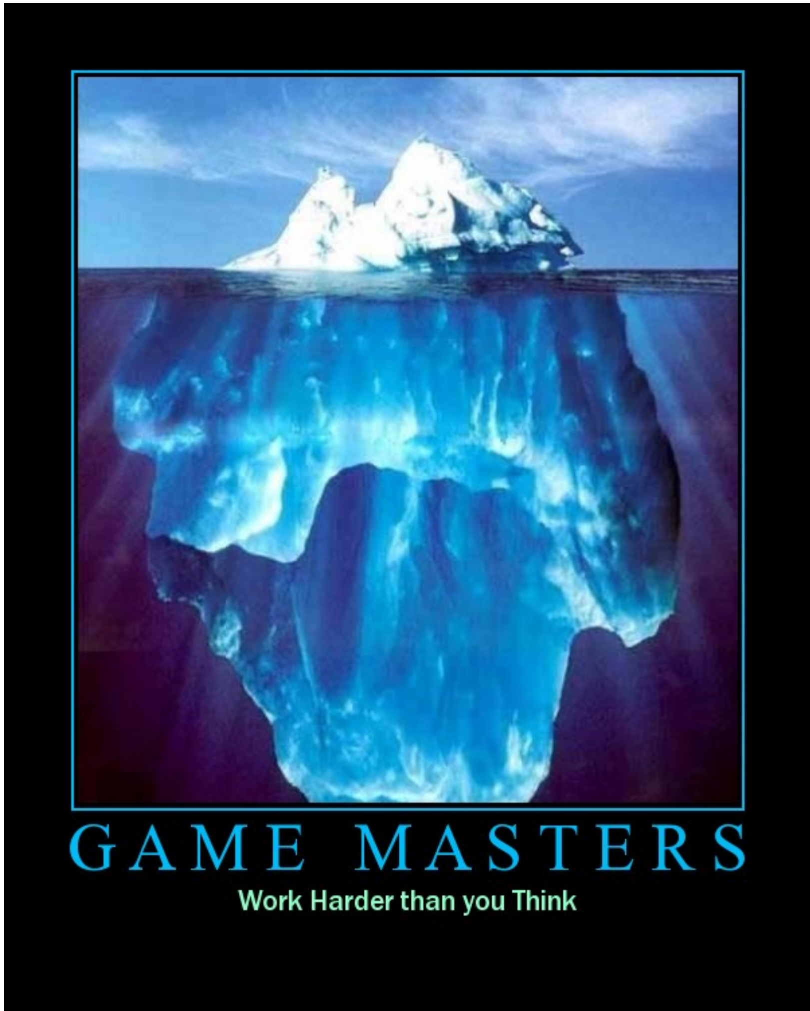
Functional Programming

Introduction To Cool

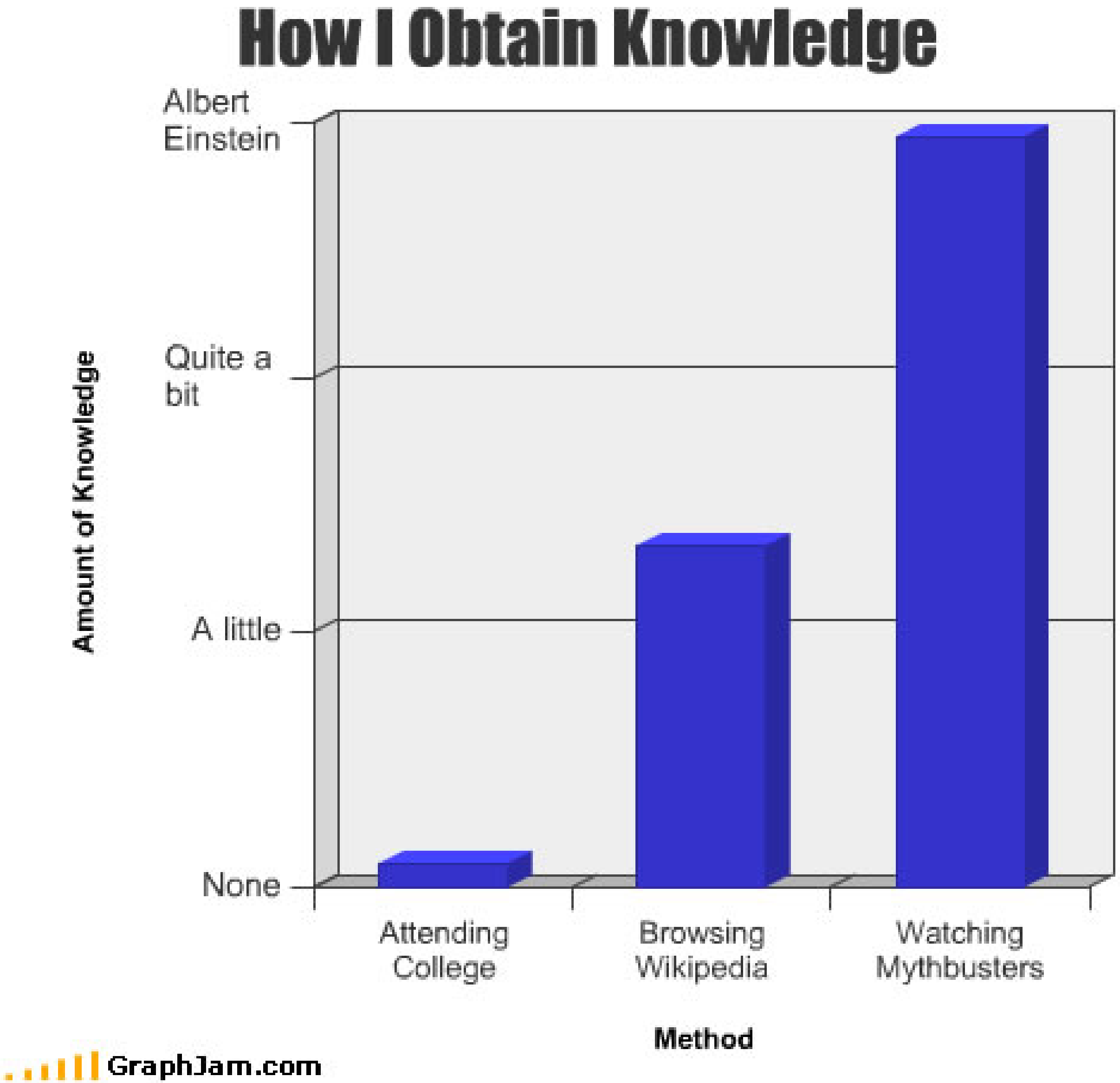
Cunning Plan

- ML Functional Programming
  - Fold
  - Sorting
- Cool Overview
  - Syntax
  - Objects
  - Methods
  - Types



Administrivia

- Credits
- Office Hours
- What was the conclusion of *Speedcoding*?



This is my final day

- ... as your ... *companion* ... through Ocaml and Cool. After this we start the compiler project.

One-Slide Summary

- Functions and type inference are **polymorphic** and operate on more than one type (e.g., List.length works on int lists and string lists).
- **fold** is a powerful higher-order function (like a swiss-army knife or duct tape).
- **Cool** is a Java-like language with classes, methods, private fields, and inheritance.

Higher-Order Functions

- Function are first-class values
  - Can be used whenever a value is expected
  - Notably, can be passed around
  - Closure captures the environment
  - let rec map f lst = match lst with
  - | [] -> []
  - | hd :: tl -> f hd :: map f tl
  - val map : (α -> β) -> α list -> β list
  - let offset = 10 in
  - let myfun x = x + offset in
  - val myfun : int -> int
  - map myfun [1;8;22] = [11;18;32]
- Extremely powerful programming technique
  - General iterators
  - Implement abstraction

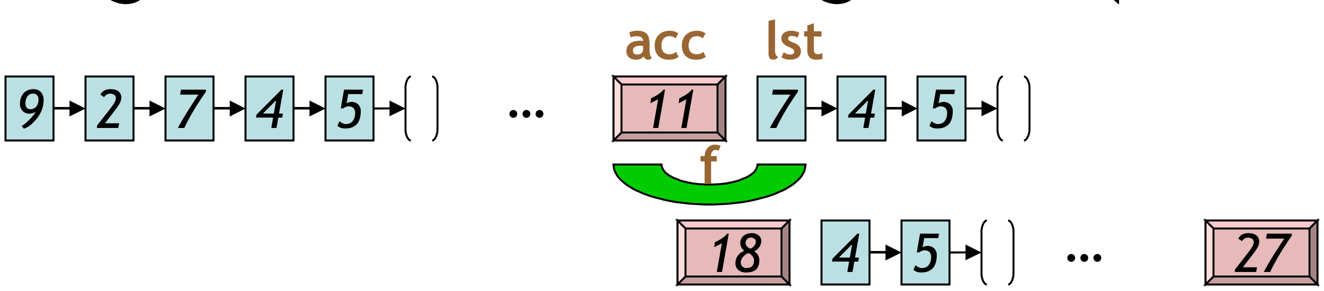
f is itself a function!

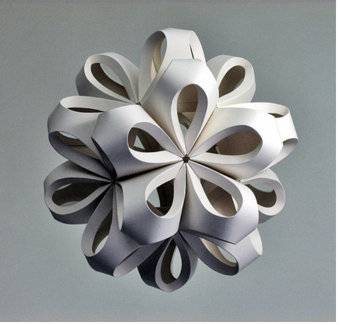
The Story of Fold

- We've seen **length** and **map**
- We can also imagine …
  - **sum** [1; 5; 8] = 14
  - **product** [1; 5; 8] = 40
  - **and** [true; true; false] = false
  - **or** [true; true; false] = true
  - **filter** (fun x -> x>4) [1; 5; 8] = [5; 8]
  - **reverse** [1; 5; 8] = [8; 5; 1]
  - **mem** 5 [1; 5; 8] = true
- Can we build all of these?

The House That Fold Built

- The **fold** operator comes from Recursion Theory (Kleene, 1952)
  - let rec **fold** f acc lst = match lst with
  - | [] -> acc
  - | hd :: tl -> fold f (f acc hd) tl
  - **val fold : (α -> β -> α) -> α -> β list -> α**
- Imagine we're summing a list (f = addition):





Folding Quiz Show

- Consider this mysterious function:

Starting accumulator value

```
let mystery lst = fold (fun acc elt -> acc + 1) 0 lst
```

Evaluating this yields next accumulator value

- One paper, work out:
  - **mystery** [8 ; 6 ; 7]
  - **mystery** [“five” ; “three” ; “oh” ; “nine”]
- What is **mystery** computing?

It's Lego Time

- Let's build things out of Fold!
  - **length** lst = fold (fun acc elt -> acc + 1) 0 lst
  - **sum** lst = fold (fun acc elt -> acc + elt) 0 lst
  - **product** lst= fold (fun acc elt -> acc * elt) 1 lst
  - **and** lst = fold (fun acc elt -> acc & elt) true lst
- How would we do **or**?
- How would we do **reverse**?

Tougher Legos

- Examples:
  - **reverse** lst = <u>fold</u> (fun acc e -> acc @ [e]) [] lst
    - Note typing: **(acc : α list) (e : α)**
  - **filter** keep_it lst = <u>fold</u> (fun acc elt ->
  - if keep_it elt then elt :: acc else acc) [] lst
  - **mem** wanted lst = <u>fold</u> (fun acc elt ->
  - acc || wanted = elt) false lst
    - Note typing: **(acc : bool) (e : α)**
- How do we do **map**?
  - Recall: map (fun x -> x +10) [1;2] = [11;12]
  - Let's write it on the board …

Map From Fold

- let **map** myfun lst =

  fold (fun acc elt -> (myfun elt) :: acc) [] lst
  - Types: **(myfun : α -> β)**
  - Types: **(lst : α list)**
  - Types: **(acc : β list)**
  - Types: **(elt : α)**
- How do we do **sort**?
  - **(sort : (α * α -> bool) -> α list -> α list)**

Do nothing which is of no use.
- **Miyamoto Musashi**, 1584-1645

Sorting Examples

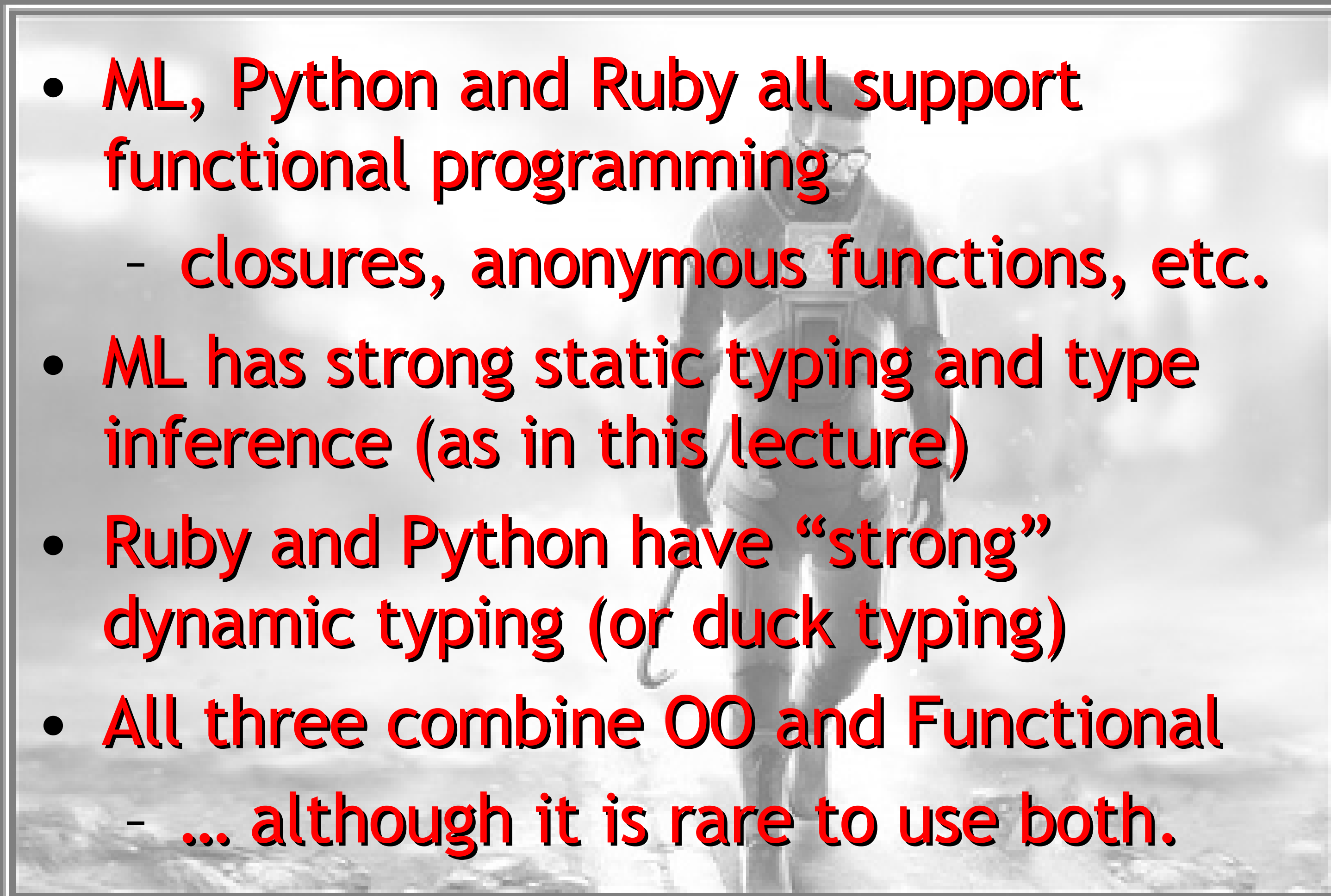
- langs = [“fortran”; “algol”; “c”]
- courses = [216; 333; 415]
- sort (fun a b -> a < b) langs
  - [“algol”; “c”; “fortran”]
- sort (fun a b -> a > b) langs
  - [“fortran”; “c”; “algol”]
- sort (fun a b -> strlen a < strlen b) langs
  - [“c”; “algol”; “fortran”]
- sort (fun a b -> match is_odd a, is_odd b with
- | true, false -> true (* odd numbers first *)
- | false, true -> false (* even numbers last *)
- | _, _ -> a < b (* otherwise ascending *)) courses
  - [333 ; 415 ; 216]

Java uses Inner Classes for this.

Partial Application and Currying

- let myadd x y = x + y
- **val myadd : int -> (int -> int)**
- myadd 3 5 = 8
- let addtwo = myadd 2
  - How do we know what this means? We use referential transparency! Basically, just substitute it in.
- **val addtwo : int -> int**
- addtwo 77 = 79
- Currying: "if you fix some arguments, you get a function of the remaining arguments"

int * int -> int would also work, but ...



- ML, Python and Ruby all support functional programming
  - closures, anonymous functions, etc.
- ML has strong static typing and type inference (as in this lecture)
- Ruby and Python have “strong” dynamic typing (or duck typing)
- All three combine OO and Functional
  - … although it is rare to use both.

MULTIFUNCTIONALTY

One tool. One million uses.

Q: Music (182 / 842)

- The *man in Brussels* gives the singer what type of sandwich in the 1982 **Men At Work** hit **Down Under**?

Q: Movie Music (420 / 842)

- In a 1995 Disney movie that has been uncharitably referred to as "Hokey-Hontas", the Stephen Schwartz lyrics *"what I love most about rivers is: / you can't step in the same river twice"* refer to the ideas of which Greek philosopher?

Q: Cartoons (694 / 842)

- In this 1986 Marvel cartoon series, young businesswoman Jerrica Benton turns into a "truly outrageous" rock star with the help of her hologram-projecting computer Synergy.

Cool Overview

- Classroom Object-Oriented Language
- Design to
  - Be implementable in one semester
  - Give a taste of implementing modern features
    - Abstraction
    - Static Typing
    - Inheritance
    - Memory management
    - And more ...
  - But many “grungy” things are left out

A Simple Example

```
class Point {
    x : Int <- 0;
    y : Int <- 0;
};
```

- Cool programs are sets of class definitions
  - A special **Main** class with a special method **main**
  - Like Java
- **class** = a collection of fields and methods
- Instances of a class are **objects**

Cool Objects

```
class Point {
    x : Int <- 0;
    y : Int; (* use default value *)
};
```

- The expression “new Point” creates a new object of class Point
- An object can be thought of as a record with a slot for each attribute (= field)

x	y
0	0

Methods

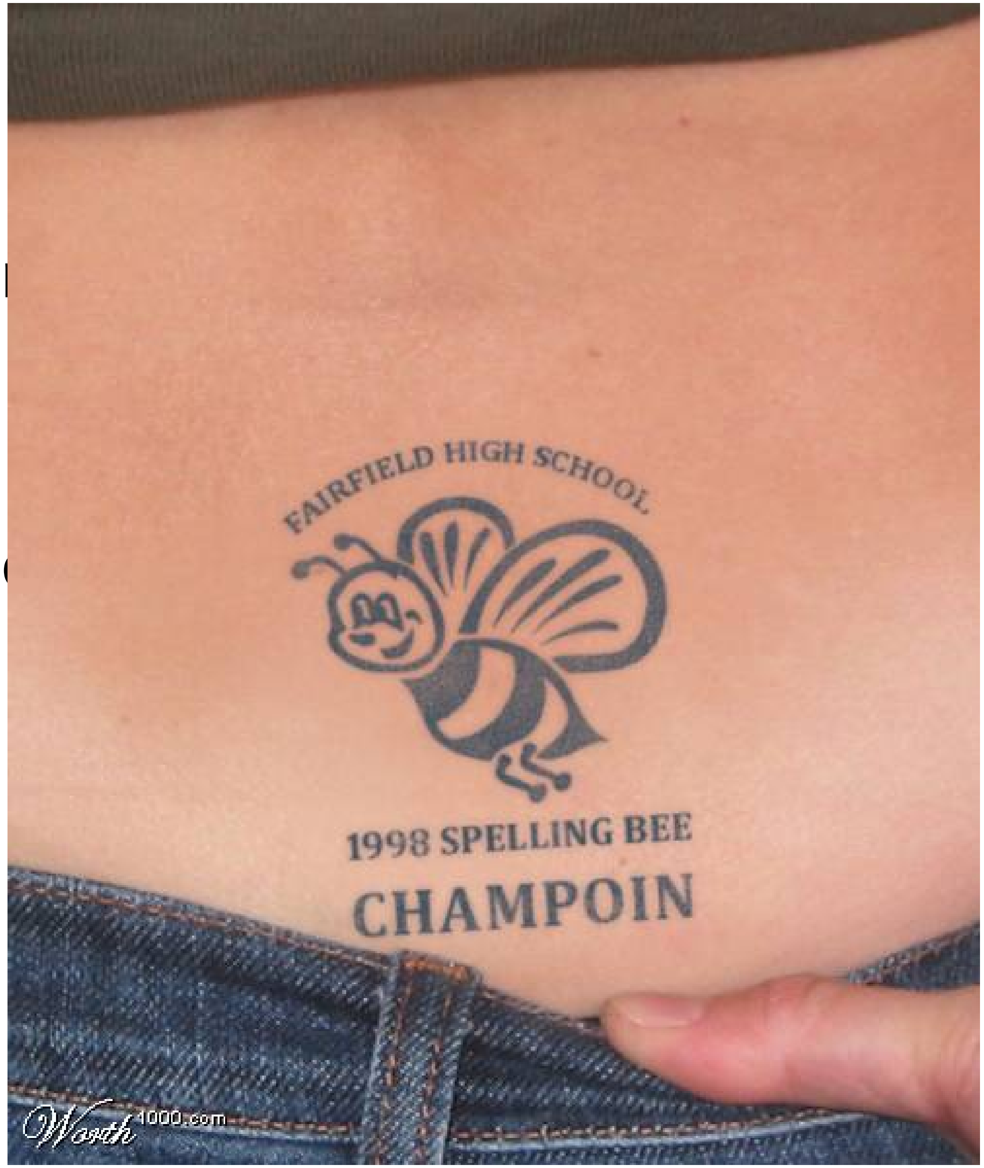
```
class Point {
     x : Int <- 0;
     y : Int <- 0;
     movePoint(newx : Int, newy : Int) : Point {
          { x <- newx;
            y <- newy;
            self;
          } -- close block expression
     }; -- close method
}; -- close class
```

- A class can also define methods for manipulating its attributes
- Methods refer to the current object using **self**

Aside: Semicolons

```
class Point {
      x : Int <- 0;
      y : Int <- 0;
      movePoint(newx : I
            { x <- newx;
              y <- newy;
              self;
            } -- close bl
      }; -- close method
}; -- close class
```

Yes, it's
somewhat arbitrary.
Still, don't get it wrong.



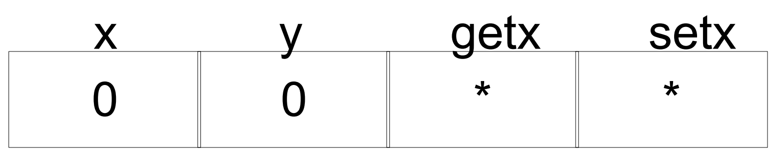
Information Hiding

- Methods are **global**
- Attributes are **local** (private) to a class
  - They can *only* be accessed by *that class's methods*

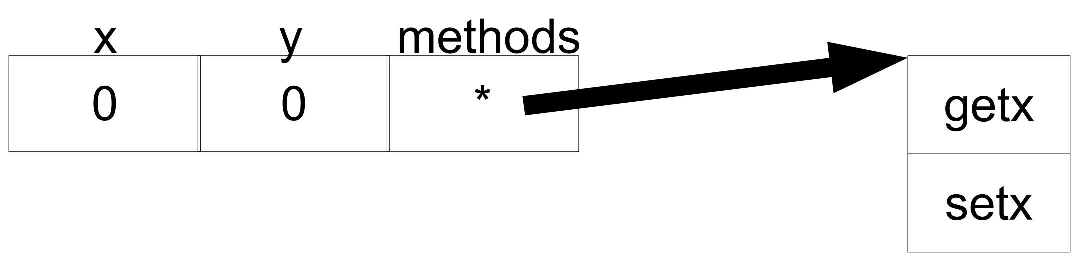
```
class Point {
      x : Int <- 0;
      y : Int <- 0;
      getx () : Int { x } ;
      setx (newx : Int) : Int { x <- newx };
};
```

Methods and Object Layout

- Each object knows how to access the code of its methods
- As if the object contains a slot pointing to the code



- In reality, implementations save space by sharing these pointers among instances of the same class



Inheritance

- We can extend points to color points using **subclassing** => **class hierarchy**

```
class ColorPoint extends Point {
      color : Int <- 0;
      movePoint(newx:Int, newy:Int) : Point {
            {  color <- 0;
               x <- newx; y <- newy;
               self;
            }
      };
};
```

Note references to fields x y –
They're defined in Point!

x	y	color	movePoint
0	0	0	*

Kool Types

- Every class is a **type**
- Base (built-in, predefined) classes:
  - **Int** for integers
  - **Bool** for booleans: true, false
  - **String** for strings
  - **Object** root of class hierarchy
- All variables must be declared
  - compiler infers types for expressions (like Java)

Cool Type Checking

- **x : Point;**
- **x <- new ColorPoint;**

- ... is well-typed if **Point** is an ancestor of **ColorPoint** in the class hierarchy
  - Anywhere a **Point** is expected, a **ColorPoint** can be used (Liskov, ...)
- Rephrase: ... is well-typed if **ColorPoint** is a **subtype** of **Point**
- **Type safety**: a well-typed program *cannot* result in run-time type errors

Method Invocation and Inheritance

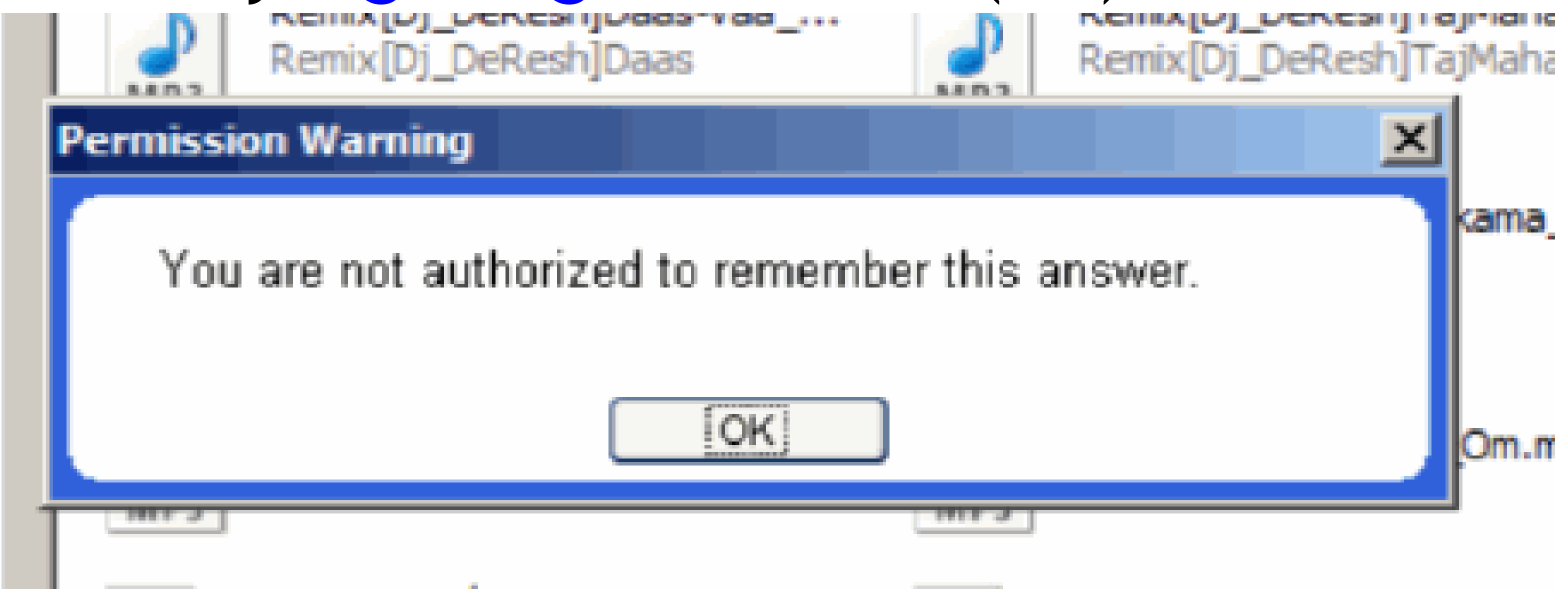
- Methods are invoked by (dynamic) **dispatch**
- Understanding dispatch in the presence of inheritance is a subtle aspect of OO
  - **p : Point;**
  - **p <- new ColorPoint;**
  - **p.movePoint(1,2);**
- p has static type Point
- p has dynamic type ColorPoint
- p.movePoint must invoke ColorPoint version

Other Expressions

- Cool is an expression language (like Ocaml)
  - Every expression has a type and a value
  - Conditionals: if E then E else E fi
  - Loops: while E loop E pool
  - Case/Switch: case E of x : Type => E ; ... esac
  - Assignment: x <- E
  - Primitive I/O: out_string(E), in_string(), ...
  - Arithmetic, Logic Operations, ...
- Missing: arrays, floats, interfaces, exceptions
  - Plus: you tell me!

Cool Memory Management

- Memory is allocated every time "**new E**" executes
- Memory is deallocated automatically when an object is not reachable anymore
  - Done by a **garbage collector** (GC)



Course Project

- A complete **compiler**
  - Cool Source ==> Assembly Program
  - Optimizations = extra credit
  - Also no GC
- Split in 4 programming assignments (PAs)
- There is adequate time to complete assignments
  - But start early and follow directions
- PA2-4 ==> individual or teams (of max 2)

Ocaml Hint Marathon!

http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Hashtbl.html

- These are the key data structures for Ocaml.
- Let's say we want to use a hashtable to map task A to the set of tasks B it depends on.

```
let depends_on = Hashtbl.create 255 in
Hashtbl.add depends_on “a” “b” ;
let a_depends_on_what = Hashtbl.find_all
  depends_on “a” in
printf “a depends on %d tasks” (List.length
  a_depends_on_what)
```

Ocaml Hint Marathon!

- What does this code do?

```
let rec read_input () =
  try
    let a = read_line () in
    let b = read_line () in
    Hashtbl.add depends_on a b ;
    read_input ()
  with _ -> ()
in
read_input ()
```

Ocaml Hint Marathon!

- What does all this code do?

```
let not_finished a = not (Hashtbl.mem finished a) in
let no_remaining_deps a =
  (List.filter not_finished (Hashtbl.find_all depends_on a))
   = [ ] (* tricky *)
in
let not_yet_run = List.filter not_finished list_of_all tasks in
let ready_to_run = List.filter no_remaining_deps
   not_yet_run in
match List.sort compare ready_to_run with
| [] -> failwith "cycle"
| a :: rest -> output a ; Hashtbl.add finished a true
```

There is a "for" loop in Ocaml, but you almost never need it! Use higher-order functions!

Homework

- Wednesday: PA 0 due
- Thursday: Chapters 2.1 - 2.2
- Thursday: Dijkstra Paper

- Bonus for getting this far: questions about **fold** are very popular on tests! If I say “write me a function that does foozle to a list”, you should be able to code it up with fold.