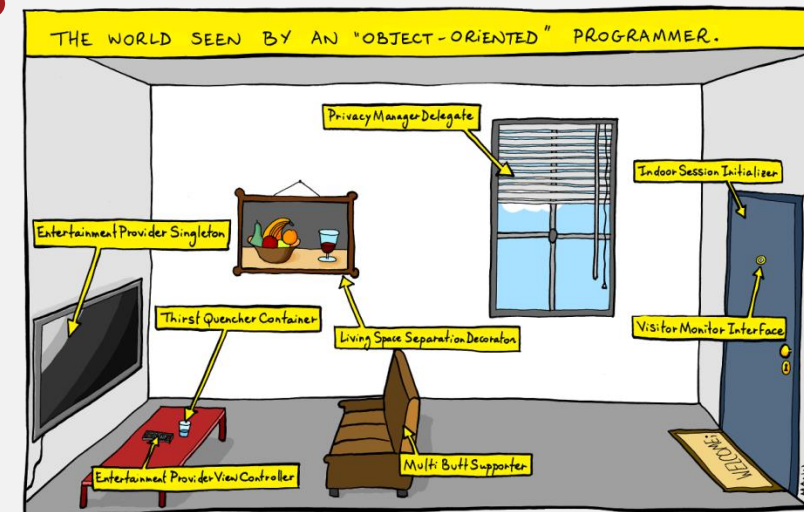


UMass Boston Computer Science  
**CS450 High Level Languages** (section 2)

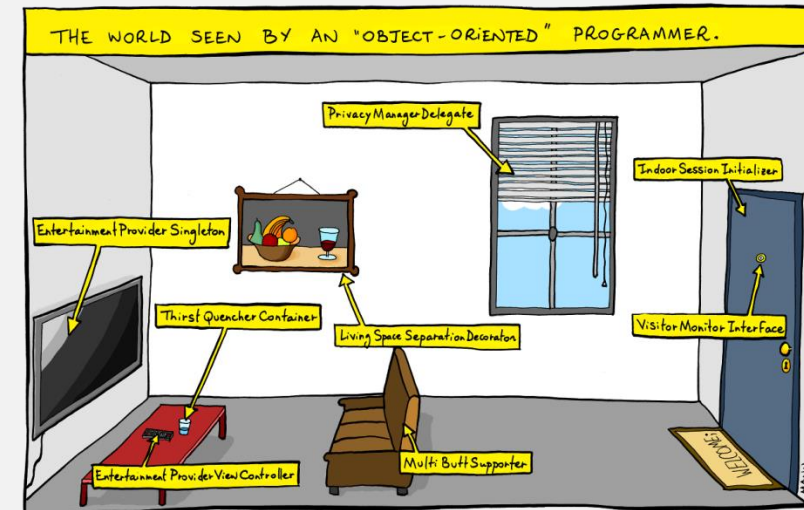
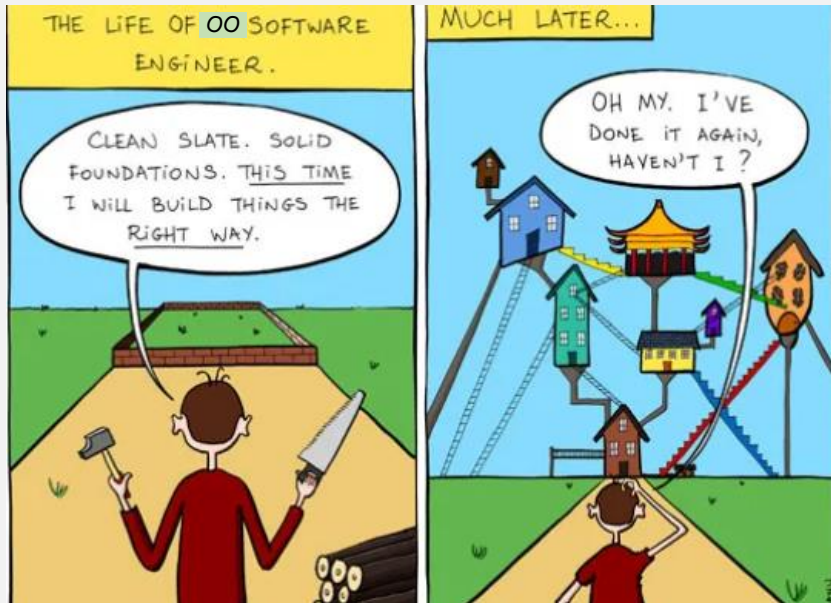
# How To Design ... OO Programs?

Monday, December 11, 2023



# Logistics

- HW 9 in
  - ~~due: Sun 12/10 11:59 pm EST~~
- HW 10 out (Shapes!)
  - due: Sun 12/17 11:59 pm EST



# A Simple OO Example: Shapes

```
interface Shape  
Image render();
```

```
class Circle
```

```
Num radius;  
Color col;
```

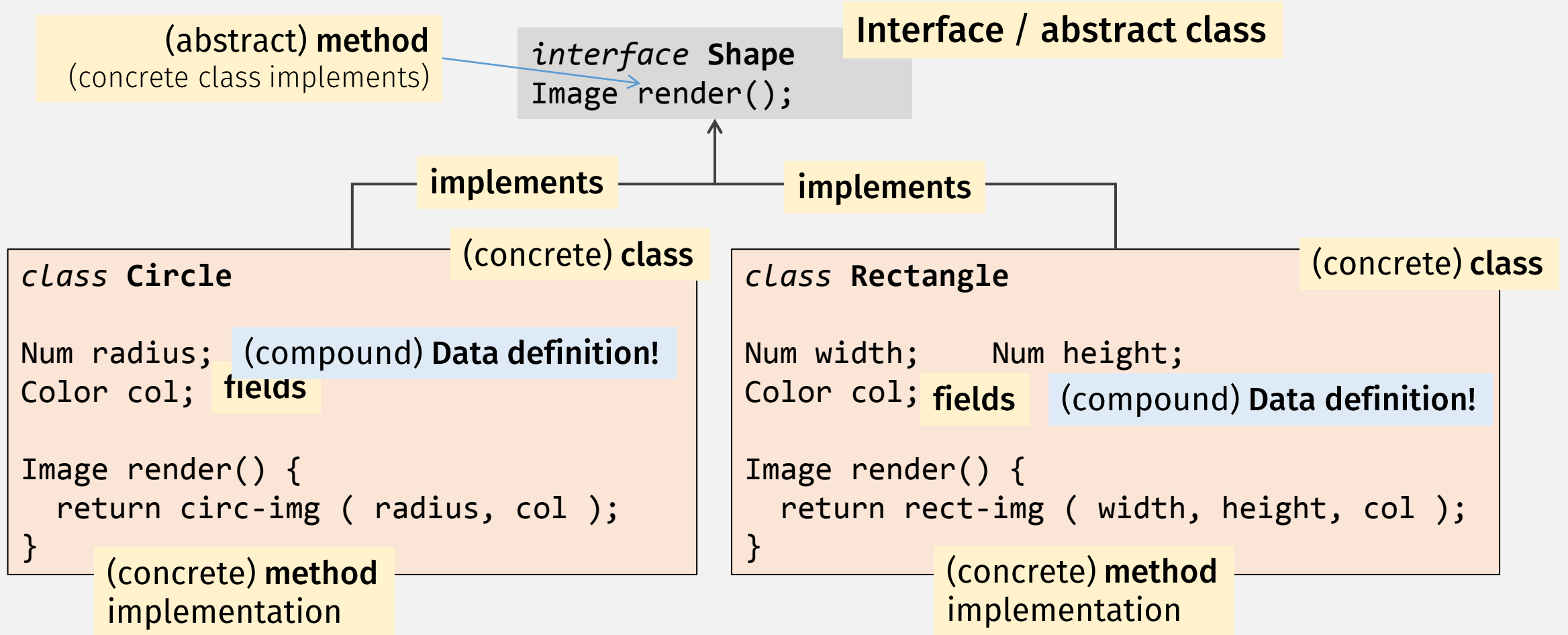
```
Image render() {  
    return circ-img ( radius, col );  
}
```

```
class Rectangle
```

```
Num width;    Num height;  
Color col;
```

```
Image render() {  
    return rect-img ( width, height, col );  
}
```

# A Simple OO Example: Terminology



# CS450 vs OO Comparison

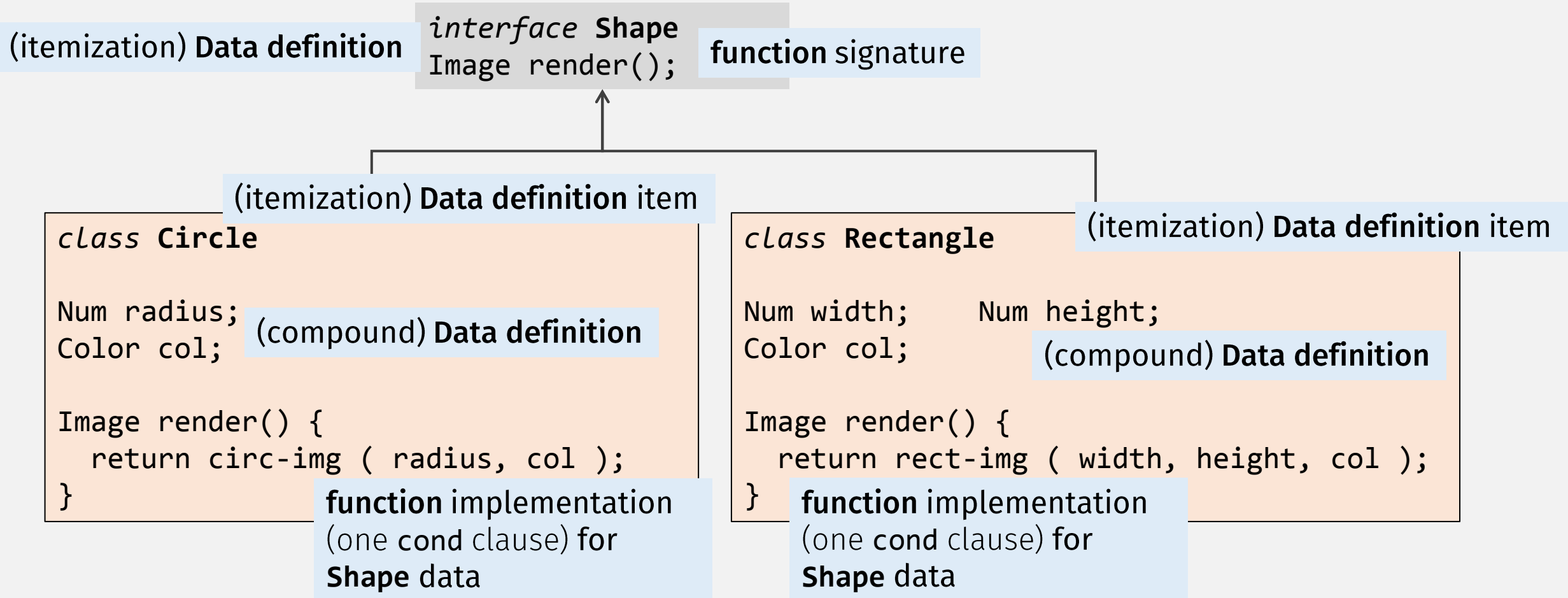
## CS 450 Design Recipe

- **Compound data** (struct) have fields, separate fns process data

## OO Programming

- **Compound data** (class) group fields and methods together!

# A Simple OO Example: Compare to CS450



# CS450 vs OO Comparison

## CS 450 Design Recipe

- **Compound data** (struct) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined

## OO Programming

- **Compound data** (class) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions

# CS450 vs OO Comparison

## CS 450 Design Recipe

- **Compound data** (struct) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!

## OO Programming

- **Compound data** (class) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!



# A Simple OO Example: Compare to CS450

```
interface Shape
Image render();
```

```
;; A Shape is one of:
;; - (rect Num Num Color)
;; interp: fields are width, height, color
;; - (circ Num Color)
;; interp: fields are radius and color
;; Represents a shape to be drawn on a canvas
```

*class* Circle

```
Num radius;
Color col;
```

(struct circ [r col])

```
Image render() {
  return circ-img (radius col);
}
```

*class* Rectangle

```
Num width;
Color col;
```

(struct rect [w h col])

```
Image render() {
  return rect-img (width height col);
}
```

```
;; render: Shape -> Image
(define (render sh)
  (cond
    [(rect? sh) (render-rect sh)]
    [(circ? sh) (render-circ sh)]))
```

method "dispatch"

"abstract" implementation

"concrete" implementations

# CS450 vs OO Comparison

## CS 450 Design Recipe

- **Compound data** (struct) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)

## OO Programming

- **Compound data** (class) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**

# A Simple OO Example: Constructors

```
interface Shape  
Image render();
```

```
Circle c = Circle( 10, blue );  
Image img = c.render();
```

```
class Circle  
  
Num radius;    Color col;  
// ...  
Circle( r, c ) {  
    radius = r;  
    col = c;  
}
```

```
class Rectangle  
  
Num width;    Num height;    Color col;  
// ...  
Rectangle( w, h, c ) {  
    width = w;    height = h;  
    col = c  
}
```

**Q:** Where are method implementations for an obj instance “stored”?

**A:** It’s another (hidden) field (see “method table”)!

# CS450 vs OO Comparison

## CS 450 Design Recipe

- **Compound data** (struct) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)
- **Struct Constructor** explicitly includes method defs ???

## OO Programming

- **Compound data** (class) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**
- **Object Constructor** implicitly includes method defs

# OO-style Constructors ... with structs!

Shape “dispatch” function

```
;; render : Shape -> Image
(define (render sh)
  (cond
    [(rect? sh) (render-rect sh)]
    [(circ? sh) (render-circ sh)]))
```

Shape “interface” definition

```
(struct Shape [render-method])
```

```
(struct circ Shape [r col])
```

```
(struct rect Shape [w h col])
```

Method implementation  
(as a field)

Shape constructors

(make method an optional argument, with default)

```
(define (mk-circ r col
                [circ-render-fn render-circ])
  (circ circ-render-fn r col))
```

```
(define (mk-rect w h col
                [rect-render-fn render-rect])
  (rect rect-render-fn w h col))
```

**Q:** Where are method implementations for an obj instance “stored”?

**A:** It’s another (hidden) field!

# CS450 vs OO Comparison

## CS 450 Design Recipe

- **Compound data** (struct) has (possibly function) fields!
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)
- **Struct Constructor** explicitly includes method defs

## OO Programming

- **Compound data** (class) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**
- **Object Constructor** implicitly includes method defs

# OO-Style Dispatch ... with structs!

450-style “dispatch” function

```
;; render : Shape -> Image
(define (render sh)
  (cond
    [(rect? sh) (render-rect sh)]
    [(circ? sh) (render-circ sh)]))
```

OO-Style “dispatch”



```
;; render : Shape -> Image
(define (render sh)
  ((shape-render-method sh) sh))
```

```
(define c (mk-circ 10 "blue"))
(define img (render c))
```

equivalent

Shape OO-style “interface and “class” definitions

```
(struct Shape [render-method])
```

```
(struct circ Shape [r col])
```

```
(struct rect Shape [w h col])
```

**Q:** But object itself must be argument to methods?

**A:** In OO langs, it’s a hidden argument (see “**this**”)!

```
Circle c = Circle( 10, blue );
Image img = c.render(); // equiv to render(c) !
```

# OO-Style Dispatch ... with structs!

450-style "dispatch" function

```
;; render : Shape -> Image
(define (render sh)
  (cond
    [(rect? sh) (render-rect sh)]
    [(circ? sh) (render-circ sh)]))
```

OO-Style "dispatch"



```
;; render : Shape -> Image
(define (render sh)
  ((shape-render-method sh) sh))
```

```
;; render-circ : Circle -> Image
(define (render-circ this)
  (match-define (circ r col) this)
  (circle r "solid" col)) ; 2htdp/image fn
```

```
;; render-rect : Rectangle -> Image
(define (render-rect this)
  (match-define (rect w h col) this)
  (rectangle w h "solid" col)) ; 2htdp/image fn
```



# CS450 vs OO Comparison

## CS 450 Design Recipe

- **Compound data** (struct) has (possibly function) fields!
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)
- **Constructor** explicitly includes method defs
- **Data to process** is explicit arg

## OO Programming

- **Compound data** (class) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**
- **Constructor** implicitly includes method defs
- **Data to process** ("this") is implicit arg

# How to Design ... OO-Style Programs

- For **Itemization Data Definition**

1. List Item Data Defs (and other prev data def parts)
2. Specify required methods
3. Define “abstract” struct (with # fields = # of methods)
4. Define explicit dispatch function(s) (one per method)

```
;; A Shape is one of:  
;; - Rectangle  
;; - Circle  
;; interp: Represents shape to draw on a canvas  
;; Required methods:  
;; - render : Shape -> Image
```

```
(struct Shape [render-meth])
```

```
;; render : Shape -> Image  
(define (render sh)  
  ((shape-render-meth sh) sh))
```

# How to Design ... OO-Style Programs

- For **Itemization Data Definition**
  1. List Item Data Defs (and other prev data def parts)
  2. Specify required methods
  3. Define “abstract” struct (with # fields = # of methods)
  4. Define explicit dispatch function(s) (one per method)
- For **each item:**
  1. Define separate Data def
  2. Define a struct, as substruct of “abstract” struct
  3. Define required methods
  4. Define constructor that includes method implementations

# How to Design ... OO-Style Programs

```
;; A Rectangle is a:  
;; (rect width : Num  
;;      height : Num  
;;      color : Color)
```

```
;; A Circle is a:  
;; (circ radius : Num  
;;      color : Color)
```

- For each **item**:

1. Define separate Data def
2. Define a struct, as substruct of "ab"
3. Define required methods
4. Define constructor that includes m

```
(struct rect Shape [w h col])  
(struct circ Shape [r col])
```

Data Definition

Defs (and other  
ed methods

ct" struct (w  
dispatch fu

```
;; render-circ : Circle -> Image  
(define (render-circ this)  
  (match-define (circ r col) this)  
  (circle r "solid" col)) ; 2htdp/image fn
```

```
;; render-rect : Rectangle -> Image  
(define (render-rect this)  
  (match-define (rect w h col) this)  
  (rectangle w h "solid" col)) ; 2htdp/image fn
```

```
;; constructors create shape "objects"  
(define (mk-rect w h col  
  [render-meth render-rect])  
  (rect render-meth w h col))  
(define (mk-circ r col  
  [render-meth render-circ])  
  (circ render-meth r col))
```

# HW 10 Preview: Drawing Shapes!

Use RACKET to create an OO-Style shape drawing app.

## Supports

- Rectangles!
- Circles!

- Repo: [cs450f23/lecture27-inclass](#)
- File: `hw10-<your last name>.rkt`

# In-class Coding 12/11: start hw10

Use RACKET to create an OO-Style shape drawing app.

```
Shape  
Image place-shape( Image canvas );
```

```
(struct Shape [place-method])  
(struct rect Shape [w h topleft])  
(struct circ Shape [r center])
```

## Circle

```
Num r;  
Posn center; // center coordinate  
  
// places circle into given canvas  
Image place-shape( Image canvas ) {  
  return  
    place-image( circ-img( r, ... ),  
                ... center  
                ... canvas );  
}
```

## Rectangle

```
Num w; Num h;  
Posn offset; // topleft corner  
  
// places rect into given canvas  
Image place-shape( Image canvas ) {  
  return  
    place-image ( rect-img(w, h, ...)  
                  ... offset  
                  ... canvas);  
}
```

**Submit your in-class work to github**