UMass Boston Computer Science

CS450 High Level Languages (section 2)

Functional Progamming vs OOP

Wednesday, December 11, 2024

(last lecture!)

Logistics

- **HW 14 out** (extra credit)
  - Use your **Example** and **Test writing skills** to …
  - … **find** and **submit bug reports** for **`#lang 450lang`** !
    - **Up to 4 reports (20 points)**
    - 8 + 6 + 4 + 2 points
  - due: **Wed 12/18 12pm (noon) EST**

(last lecture!)

Last Time

There's Nothing Special About OOP!

- A typical (`interface` and `class`es) **OOP program** is just a **specific data definition / function design choice!**
  - imposed by the language!

- Data definition:
  - **itemization** of **compound data** ...
  - ... where **processing functions are grouped with other data fields!**

- Function design:
  - **Function to process this itemization data is split into separate** "`methods`" (one for each kind of item in the itemization)

Last Time

A Simple OO Example: Compare to CS450

Data definition: Itemization of compound data

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

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image
```

```
;; A Circle is a (circ Num Color)
;; fields are radius and color
```

```
class Circle                                  itemization item

Num radius;                                   Compound data fields
Color col;                                    (struct circ [r col])

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

```
;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
```

```
class Rectangle                               itemization item

Num width;      Num height;                   Compound data fields
Color col;                                    (struct rect [w h col])

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

Last Time

A Simple OO Example: Compare to CS450

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

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image
```

```
class Circle

Num radius;
Color col;

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

```
class Rectangle

Num width;    Num height;
Color col;

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

(one cond clause of a) **Shape**-processing function, as a (hidden) field!

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

(one cond clause of a) **Shape**-processing function, as a (hidden) field!

In OO langs, this "**dispatch**" function is implicitly written for you

Calls item-specific implementations

Last Time

A Simple OO Example: as structs !

(required) method, as field

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

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

```
class Circle

Num radius;
Color col;

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

"implements" interface

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

```
;; render-circ : Circle -> Image
(define (render-circ this) …)
```

```
class Rectangle

Num width;    Num height;
Color col;

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

"implements" interface

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

```
;; render-rect : Rectangle -> Image
(define (render-rect this) … )
```

In OO langs, every method implicitly has a class instance arg ("**this**"!)

Last Time

OO-style Constructors … with structs!

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

manually write alternate Shape constructors, with explicit method impls

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

default

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

```
;; render-circ : Circle -> Image
(define (render-circ this) …)
```

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

(method arg optional, with default)

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

```
;; render-rect : Rectangle -> Image
(define (render-rect this) … )
```

Last Time

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)]))
```

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

OO-Style “dispatch”

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

```
;; render-circ : Circle -> Image
(define (render-circ this) …)
```

```
;; render-rect : Rectangle -> Image
(define (render-rect this) … )
```

OO vs CS450 Comparison

OO Programming

- `interface` + `class` imply specific (Itemization-of-compound) Data Def
- `class` (compound data) has fields and methods together!
- `class` constructor implicitly adds method impls to created object
- data value to process is implicit method arg
- Implicit itemization **dispatch**

CS 450 Design Recipe

- Explicitly define any kind of Data Def
- `struct` (compound data) fields typically do not include functions
- data processing function is separate definition
- data value to process is explicit function arg
- Explicit itemization **dispatch** (cond)

OO vs CS450 "OO"-Style Comparison

OO Programming

- `interface` + `class` imply specific (Itemization-of-compound) Data Def
- `class` (compound data) has fields and methods together!
- `class` constructor implicitly adds method impls to created object
- data value to process is implicit method arg
- Implicit itemization **dispatch**

CS 450 "OO-style" Design Recipe

- Explicitly define (itemization-of-compound) Data Def
- Include methods in `struct` (compound data) fields
- Define additional constructor with explicit method args
- data value to process is explicit ~~function~~ "method" arg
- Define explicit OO-style **dispatch**

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])
```

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

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)
```

ata Definition

Defs (and ot

ed methods

ct" struct (w

dispatch fu

```
;; render-circ : Circle -> Image
(define (render-circ this) … )
```

```
;; render-rect : Rectangle -> Image
(define (render-rect this) … )
```

- 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 impls

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

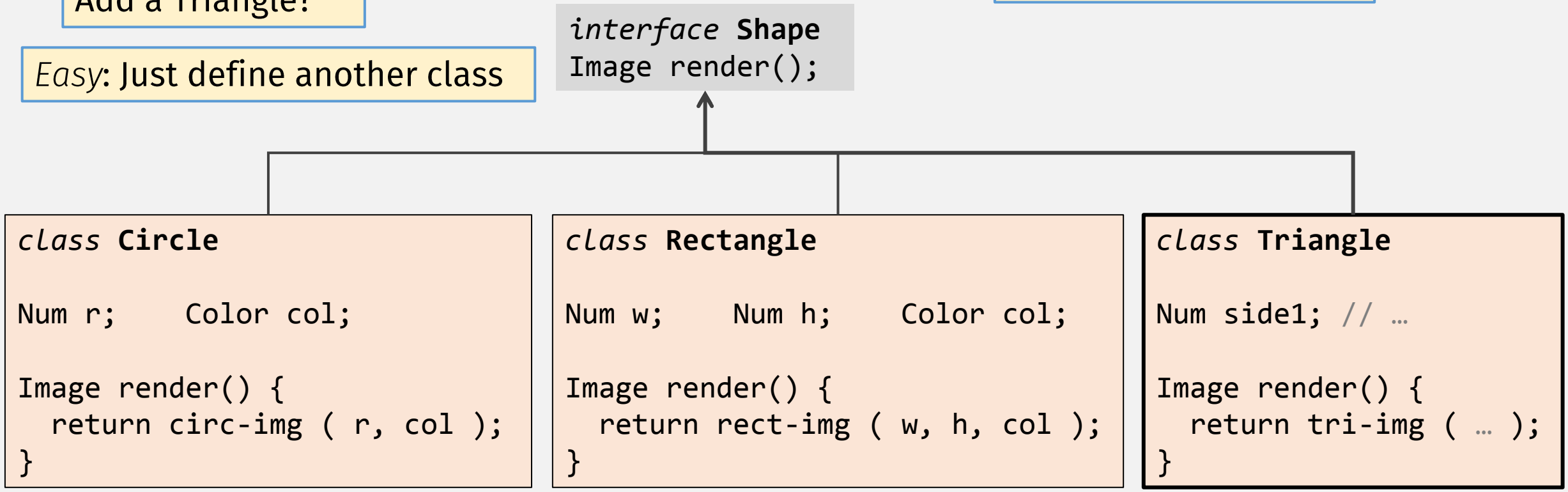
```
(define (mk-rect w h col
          [render
           render-rect])
  (rect render w h col))
(define (mk-circ r col)
          [render
           render-circ])
  (circ render r col))
```

A Simple OO Example: Extensions?

Add a Triangle?

Easy: Just define another class

Add a rotate method?

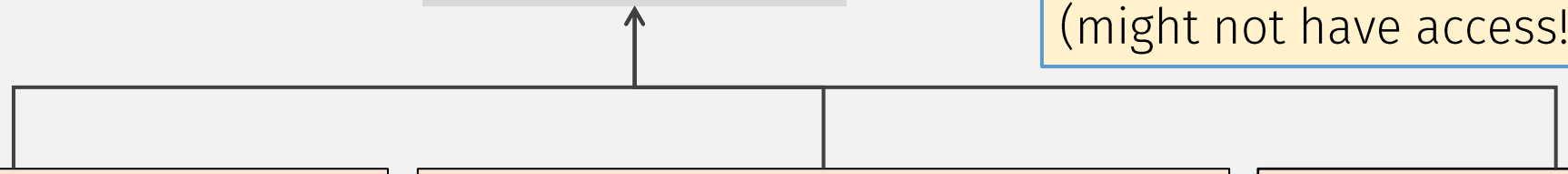


A Simple OO Example: Extensions?

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

Add **rotate** method?

Hard!: must update interface and every existing class (might not have access!)



```
class Circle

Num r;    Color col;

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

Circle rotate() { … }
```

```
class Rectangle

Num w;    Num h;    Color col;

Image render() {
  return rect-img ( w, h, col );
}

Rectangle rotate() { … }
```

```
class Triangle

Num side1; // …

Image render() {
  return tri-img ( … );
}

Triangle rotate() { … }
```

Shapes, CS450 style

Add a Triangle?

Hard!: must:
- update data def,
- define new struct,
- update every existing "dispatch" function (might not have access!)

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image
```

```
;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
(struct rect [w h col])
;; A Circle is a (circ Num Color)
;; fields are radius and color
(struct circ [r col])
```

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

Shapes, CS450 style

Add a Triangle?

Hard!: must:
- update data def,
- define new struct,
- update every existing "dispatch" function (might not have access!)

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; - Triangle
;; interp: Represents a shape image
```

```
;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
(struct rect [w h col])
;; A Circle is a (circ Num Color)
;; fields are radius and color
(struct circ [r col])
;; A Triangle is a (tri … )
;; fields are …
(struct tri [ … ])
```

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

Shapes, CS450 style

Add a `rotate` function?

Easy!: Just define another function!

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image
```

```
;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
(struct rect [w h col])
;; A Circle is a (circ Num Color)
;; fields are radius and color
(struct circ [r col])
```

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

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

FP vs OO Comparison

Add another "item" to an itemization data def, e.g., `Triangle`

- **OO**: *Easy*
  - Just define another class
    - Class methods only process that kind of item
    - Implicit "Dispatch" function(s) automatically updated
- **FP**: *Hard*
  - Must update data def, define another struct
  - Every explicit "dispatch" function must be manually updated with another `cond` clause

Add a new operation for an itemization data def, e.g., `rotate`

- **OO**: *Hard*
  - Must update interface, and add new method to every class that implements it
- **FP**: *Easy*
  - Just define another function

A better way? Mixins and classes as Results

(`class` “arithmetic”)

- A `Mixin` is a function, whose input and output is a `class`!

- Available in many languages:
  - Racket
  - JavaScript
  - Scala

- `(add-rotate-mixin class-without-rotate)`
  `=> class-with-rotate`

Thank you for a great semester!

In-class Coding 12/11: work on HW14!

Thank you for a great semester!

create a picture for the last lecture in CS450