UMass Boston Computer Science

CS450 High Level Languages

High Level Comparison: FP vs OOP

~~Tuesday, February 24, 2026~~

Thursday, February 26, 2026

Logistics

- HW2
  - Grades out
- HW3
  - ~~due: Tue 2/24 11am EST~~
- HW4
  - out: Tue 2/24 11am EST
  - due: Tue 3/3 11am EST

(needed for HW 4!)

Randomness

[bracketed args] = optional

```
(random k [rand-gen]) → exact-nonnegative-integer?
  k : (integer-in 1 4294967087)
  rand-gen : pseudo-random-generator?
           = (current-pseudo-random-generator)
```

When called with an integer argument *k*, returns a random exact integer in the range `0` to *k*-1.

Optional arg Default value

```
(random min max [rand-gen]) → exact-integer?
  min : exact-integer?
  max : (integer-in (+ 1 min) (+ 4294967087 min))
  rand-gen : pseudo-random-generator?
           = (current-pseudo-random-generator)
```

When called with two integer arguments *min* and *max*, returns a random exact integer in the range *min* to *max*-1.

"random" is not random???

A **pseudorandom number generator** (**PRNG**), also known as a **deterministic random bit generator** (**DRBG**),[1] is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's *seed*

Not "secure"!
e.g., for generating passwords

vs

A **cryptographically secure pseudorandom number generator** (**CSPRNG**) or **cryptographic pseudorandom number generator** (**CPRNG**) is a pseudorandom number generator (PRNG) with properties that make it suitable for use in cryptography.

(needed for HW 4!)

Random Functions: Same Recipe (almost)!

```
;; A Velocity is a non-negative integer
;;          Represents:  pixels/tick change in a ball coordinate
(define MAX-VELOCITY 10)
```

```
;; random-velocity : -> Velocity
;; returns a random velocity between 0 and MAX-VELOCITY
(define (random-velocity)
  (random MAX-VELOCITY))
```

Random functions don't need **Examples** (but **Purpose Stmt** more important now)

Functions (with **side-effects**) can have zero args!

```
(check-true (< (random-velocity) MAX-VELOCITY))
(check-true (>= (random-velocity) 0))
(check-true (integer? (random-velocity)))
(check-pred (λ (v) (and (integer? v)
                        (< v MAX-VELOCITY)
                        (>= v 0)))
            (random-velocity))
```

Can still Test! Just less precise

Previously

Kinds of Data Definitions

- Basic data
  - E.g., **numbers, strings,** etc
- Intervals
  - Data that is from a **range of values,** e.g., [0, 100)
- Enumerations
  - Data that is **one of a list of possible values**, e.g., “green”, “red”, “yellow”
- Itemizations
  - Data value that can be from a **list of possible other data definitions**
  - E.g., **either** a **string** or **number** (Generalizes enumerations)
- Compound Data
  - Data that is a **combination of values from other data definitions**

Combo of …

(extremely common, see hw5 and up)

Itemization of Compound Data – Example

```
;; A Shape is one of:
;; - Rect
;; - Circ
```

```
;; A Rect is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; Represents: a rectangle of the specified width, height, color
;; A Circ is a (mk-Circ [r : Num] [c : Color])
;; Represents: a circle of the specified radius and color
```

Itemization of Compound Data – template

```
;; A Shape is one of:
;; - Rect
;; - Circ
```

Template looks like **Data Def,** i.e., **number of cases**, etc

```
;; A Rect is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; Represents: a rectangle of the specified width, height, color
;; A Circ is a (mk-Circ [r : Num] [c : Color])
;; Represents: a circle of the specified radius and colors
```

```
;; shape-fn : Shape -> ???
(define (shape-fn sh)
 (cond
  [(Rect? sh) … (Rect-fn sh) … ]
  [(Circ? sh) … (Circ-fn sh) … ]))
```

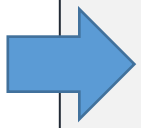
Template should call **other Templates,** when needed

Itemization of Compound Data – function!

```
;; A Shape is one of:
;; - Rect
;; - Circ
```

```
;; A Rect is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; Represents: a rectangle of the specified width, height, color
;; A Circ is a (mk-Circ [r : Num] [c : Color])
```

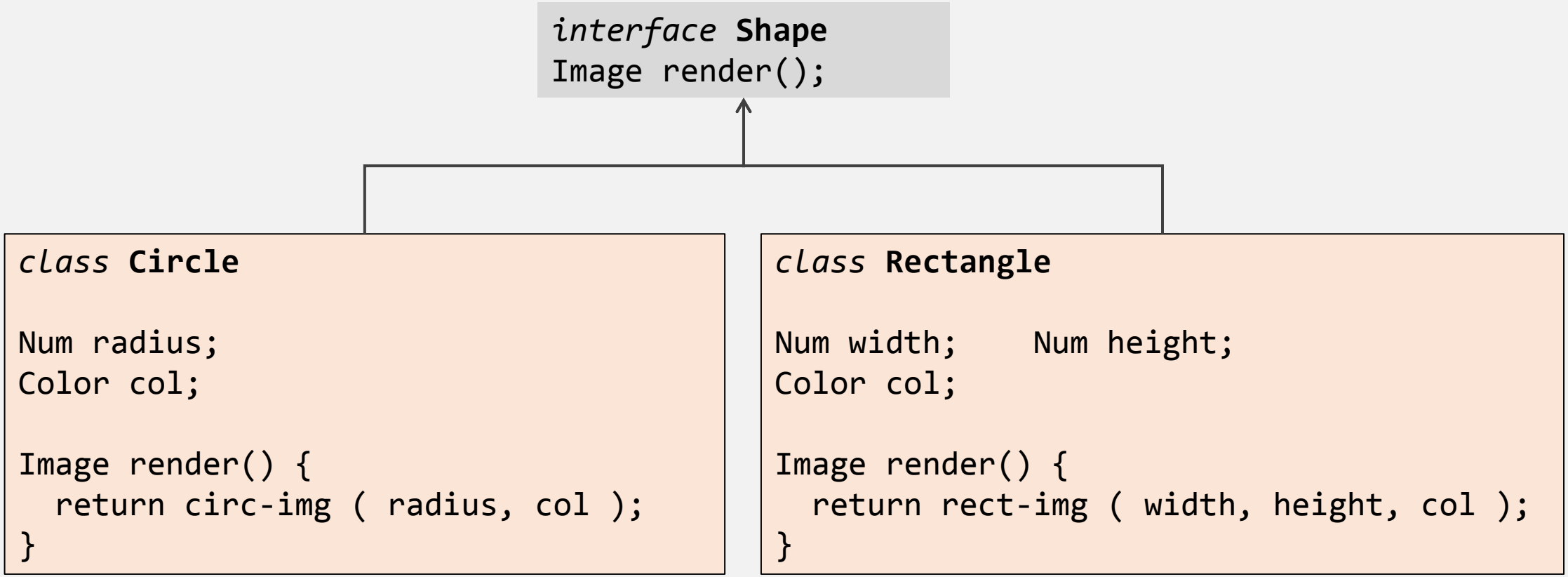
```
;; shape-fn : Shape -> ???
(define (shape-fn sh)
 (cond
  [(Rect? sh) … (Rect-fn sh) … ]
  [(Circ? sh) … (Circ-fn sh) … ]))
```



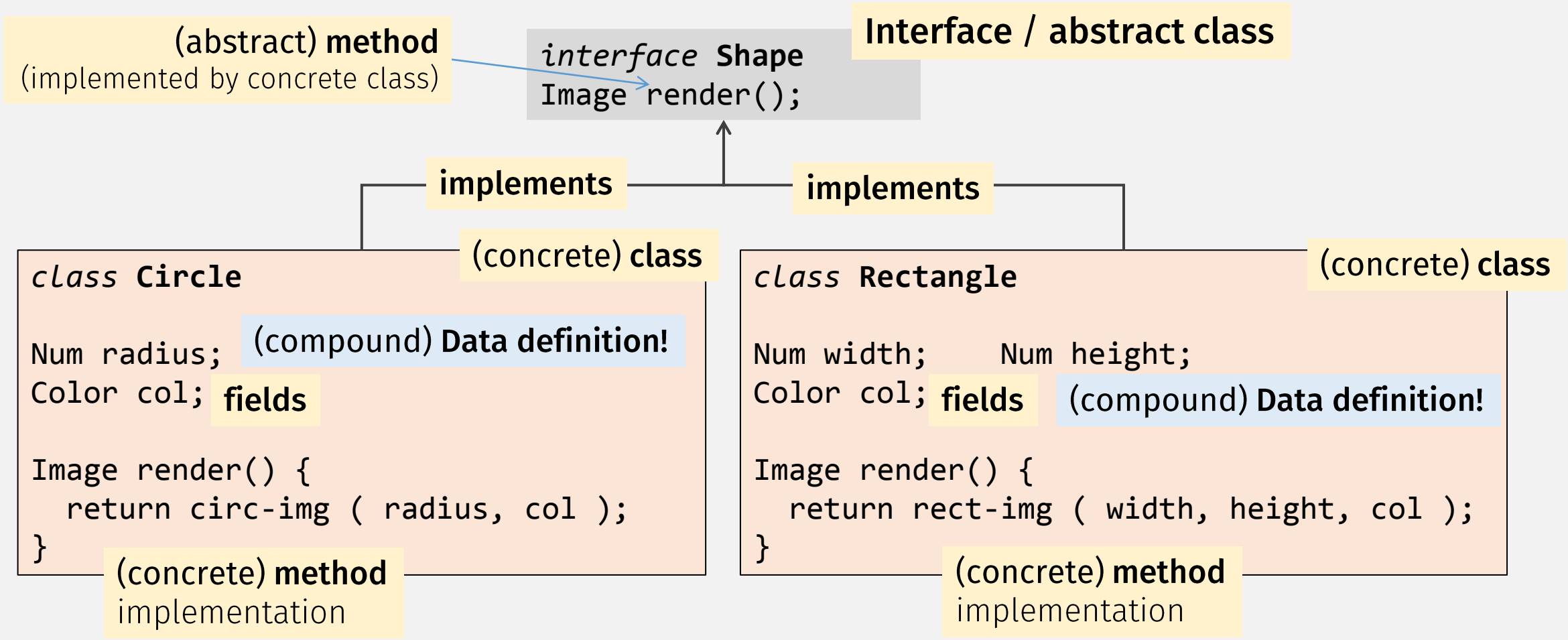
```
;; render : Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (rect-img sh)]
  [(Circ? sh) (circ-img sh)]))
```

Feels familiar …?

A Simple OO Example: Shapes



A Simple OO Example: Terminology



CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have **fields** but separate fns (to process data)

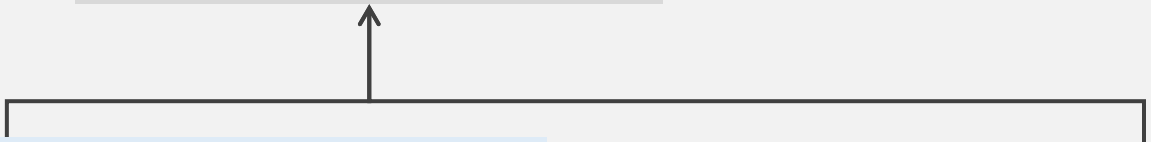
OO Programming

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

A Simple OO Example: Compare to CS450

(itemization) **Data definition**

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



(itemization) **Data definition** item

```
class Circle

Num radius;
Color col;

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

(compound) **Data definition**

function implementation (one **cond** clause) for **Shape** data (split up)

(itemization) **Data definition** item

```
class Rectangle

Num width;    Num height;
Color col;

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

(compound) **Data definition**

function implementation (one **cond** clause) for **Shape** data (split up)

CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have fields but separate fns (to 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	OO Programming
• **Compound data** (`struct`) have fields but separate fns (to process data)	• **Compound data** (`class`) group fields and methods together!
• **Itemization** Data Defs: explicitly defined	• **Itemization** Data Defs: implied by interface / class definitions
• **Functions** organized by the kind of data they process! ⇐Same principle⇒	• **Methods** organized by the kind of data they process!

1 function,
1 task, … processes
1 data definition!

A Simple OO Example: Compare to CS450

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
```

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

```
class Circle

Num radius;
Color col;

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

(struct Circ [r col])

```
class Rectangle

Num width;    Num height;
Color col;

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

(struct Rect [w h col])

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

"cond" template is method "**dispatch**" – same as OO !

"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)

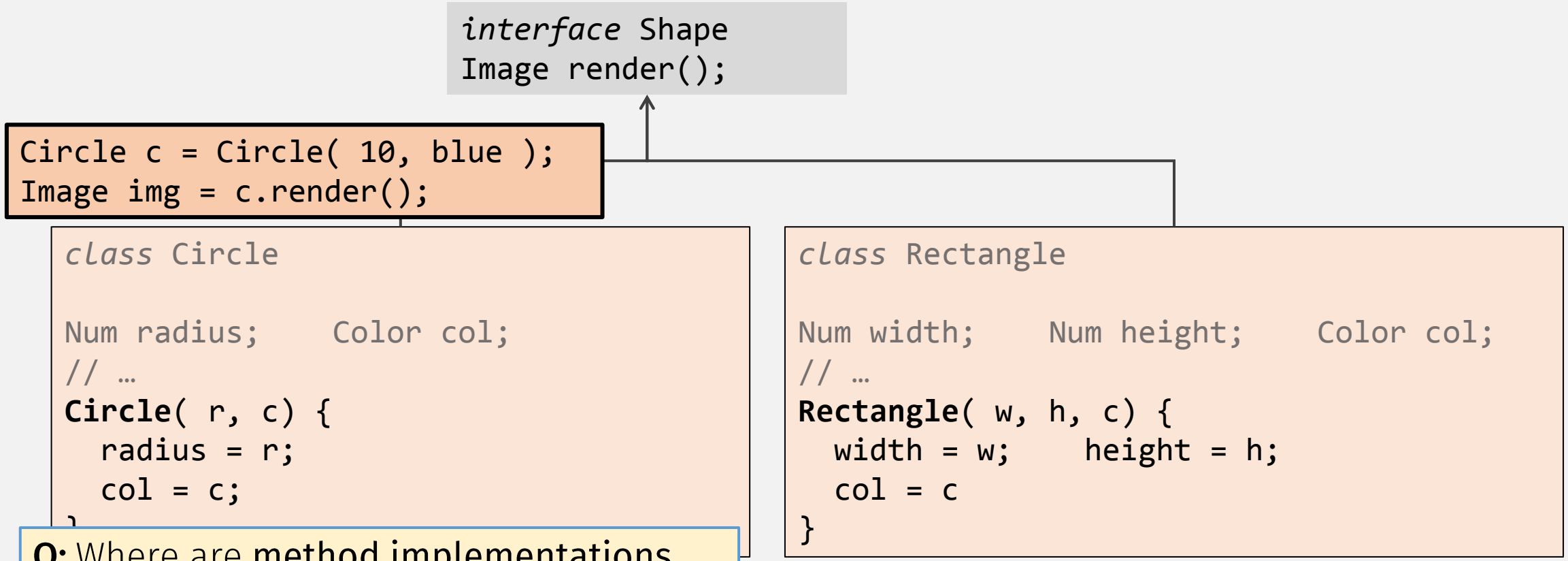
```
;; (explicit) render: Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (rect-img sh)]
  [(Circ? sh) (circ-img sh)]))
```

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**

```
;; (implicit) render: Shape -> Image
Image render (Shape sh)
  if (sh instanceof Rectangle) { rect-img(sh); }
  else if (sh instanceof Circle) { circ-img(sh); }
```

A Simple OO Example: Constructors



Q: Where are **method implementations** for an object instance **"stored"?**

A: It's another (hidden) **field** (see "**method table**")**!**

CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have fields but separate fns (to 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 ???**

Can we do this?

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**

Not for hw!

OO-style Constructors … with structs!

Shape "dispatch" function

```
;; render : Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (rect-img sh)]
  [(Circ? sh) ((Shape-render-method sh) sh)]))
```

(don't need to call **circ-img** directly anymore?)

Compare to OO Syntax:

```
sh.Shape-render-method
```

"get" the **method** (field)

(to more resemble OO, make method an **optional argument**, with **default**)

Shape "interface" definition

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

Method implementation (as a field)

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

Super struct

Circ constructor must be given 3 args

Shape constructors

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

default

Q: Where are **method implementations** for an object **instance** "stored"?

A: It's another (hidden) **field!**

Not for hw!

OO-style dispatch … with structs!

Shape "dispatch" function (450-style)

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

OO-Style "dispatch"

Methods are just fields that are functions!

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

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

Same as OO (different syntax): `sh.Shape-render-method ( this )`

Redundant argument?

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

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

"**this**" is implicit in some langs (JAVA), explicit in others (PYTHON, RACKET)

CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have **fields** but separate fns (to 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

CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have fields but separate fns (to process data)
- **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

Summary

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!

1 function,
1 task, … processes
1 data definition!

- 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)

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)

Not for hw!

OO vs CS450 "OO"-Style Comparison

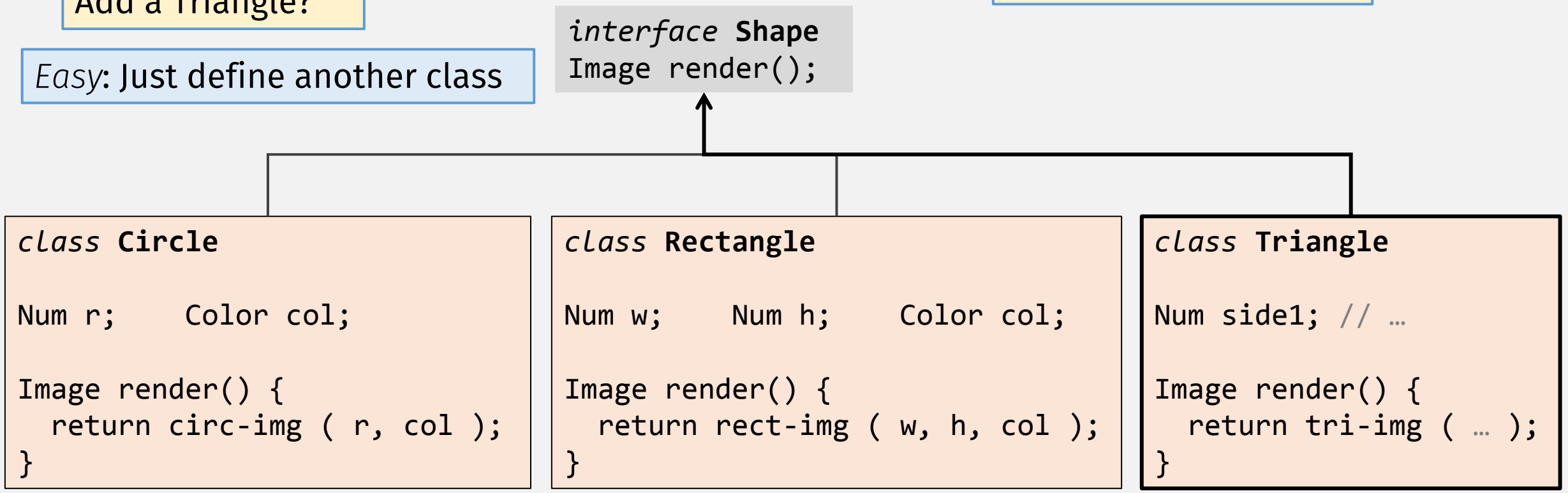
OO Programming	CS 450 "OO-style" Design Recipe
• `interface` + `class` imply specific (Itemization-of-compound) Data Def	• Explicitly define (itemization-of-compound) Data Def
• `class` (compound data) has fields and methods together!	• Include methods in `struct` (compound data) fields
• `class` constructor implicitly adds method impls to created object	• Define additional constructor with explicit method args
• data value to process is implicit method arg	• data value to process is explicit ~~function~~ "method" arg
• Implicit itemization **dispatch**	• Define explicit OO-style **dispatch**

A Simple OO Example: Extensions?

Add a Triangle?

Easy: Just define another class

Add a rotate method?



A Simple OO Example: Extensions?

Add **rotate** method?

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

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

```
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:

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

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-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,

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

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-Circ Num Color)
;; fields are radius and color
(struct Circ [r col])
;; A Triangle is a (mk-Tri … )
;; fields are …
(struct Tri [ … ])
```

```
;; 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!)

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

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

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-Circ Num Color)
;; fields are radius and color
(struct Circ [r col])
;; A Triangle is a (mk-Tri … )
;; fields are …
(struct Tri [ … ])
```

Shapes, CS450 style

Add a Triangle?

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

Add a **`rotate`** function?

Easy!: Just define another function!

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

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-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 a **new "item"** to **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** and **define another `struct`**
    - **Explicit "dispatch" function**(s) must be manually updated with another **cond** clause

Add a **new operation** for **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
  ```

Demo ...
  ```