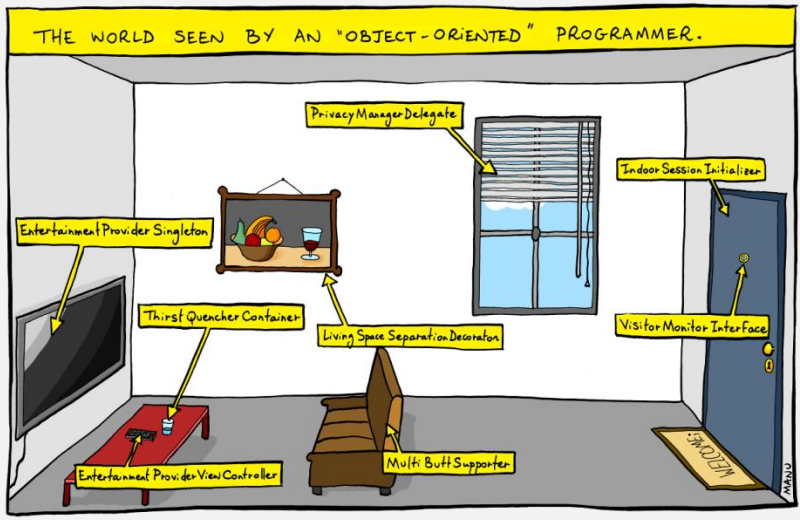
(last lecture!)

UMass Boston Computer Science

CS450 High Level Languages

High Level Comparison: FP vs OOP

Tuesday, May 13, 2025



(last lecture!)

Logistics

- HW13 **extended**
  - ~~due: Tue 5/13 11am EST~~
  - due: Thurs 5/15 11am EST

- HW 14 out (extra credit)
  - Use your **Example** and **Test writing skills** to …
  - … **find** and **submit bug reports** for **`#lang 450lang`** !
    - "bug" = does not match specification
    - Up to 4 reports (20 points)
    - 8 + 6 + 4 + 2 points
  - due: Tue 5/20 11am EST (no late)
  - **Respectful reports only!**

Kinds of Data Definitions

Previously (Lecture 6)

- 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., <u>either</u> a **string** or **number** (Generalizes enumerations)
- Compound Data
  - Data that is a **combination of values from other data definitions**

Combo of ...

HW7! (and onwards)

Itemization of Compound Data - Example

```
;; A Shape is one of:
;; - (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; - (mk-Circ [r : Num] [c : Color])
;; interp: fields are radius and color
;; Represents a shape to be drawn on a canvas
```

Itemization of Compound Data - Template

```
;; A Shape is one of:
;; - (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; - (mk-Circ [r : Num] [c : Color])
;; interp: fields are radius and color
;; Represents a shape to be drawn on a canvas
```

```
;; shape-fn : Shape -> ???
(define (shape-fn sh)
 (cond
  [(Rect? sh) … (rect-h sh) … (rect-w sh) … (rect-c sh) … ]
  [(Circ? sh) … (circ-r sh) … (circ-c sh) … ]))
```

Itemization of Compound Data – 2nd way

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

```
;; A Rectangle is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; A Circle is a (mk-Circ [r : Num] [c : Color])
;; interp: fields are radius and colors
```

Itemization of Compound Data – template

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

```
;; A Rectangle is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; A Circle is a (mk-Circ [r : Num] [c : Color])
colors
```

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

Itemization of Compound Data – function!

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

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

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

(abstract) **method**
(concrete class implements)

Interface / abstract class

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

implements

implements

(concrete) **class**

```
class Circle

Num radius;
Color col;

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

(compound) **Data definition!**

fields

(concrete) **method** implementation

(concrete) **class**

```
class Rectangle

Num width;    Num height;
Color col;

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

fields

(compound) **Data definition!**

(concrete) **method** implementation

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

(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, 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	OO Programming
• **Compound data** (`struct`) have fields, separate fns 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!	• **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
}
```

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

```
class Rectangle

Num width;    Num height;
Color col;

Image render() {
                g ( 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)]))
```

method "**dispatch**" - OO does the same!

"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

```
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 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, 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) (rect-img sh)]
  [(Circ? sh) (circ-img sh)]))
```

Shape "interface" definition

Method implementation (as a field)

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

```
(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

(make method an optional argument, with default)

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

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

Then create same definitions for **rect** …

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

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

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!

- 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)

Summary

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
```

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

```
class Circle

Num radius;
Color col;

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

itemization item

Compound data fields

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

```
class Rectangle

Num width;   Num height;
Color col;

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

itemization item

Compound data fields

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

Summary

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!

(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)]))
```

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

Calls item-specific implementations

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])
```

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

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

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

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 );
}
```

```
class Rectangle

Num width;    Num height;
Color col;

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

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

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

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

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

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

(method arg optional, with default)

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

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

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

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

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))
```

struct “getter”

```
;; 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	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?

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

```
class Circle

Num r;    Color col;

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

```
class Rectangle

Num w;    Num h;    Color col;

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

```
class Triangle

Num side1; // …

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

A Simple OO Example: Extensions?

Add **rotate** method?

Hard!: must update interface <u>and</u> 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
;; 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])
```

```
;; 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!)

```
;; 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)]
  [(tri? sh)  (render-tri sh) ]))
```

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
;; 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])
```

```
;; 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 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`

Can't spell!

Too much hair!

Thank you for a great semester!

create a picture for the last lecture in CS450