Where Classes Come From

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 8.1

Goals of this lesson

- Learn about classes, objects, fields, and interfaces
- Learn how these ideas are expressed in the Racket object system
- Translate from Data Definitions to classes and interfaces

Functions in a space-invader world

```
 World = Heli * Bombs * ..other stuff..
-----
 world-after-tick            : World -> World
 world-after-mouse-event : World Number Number MouseEvent -> World
 world-to-scene              : World Scene -> Scene

 Heli = Posn * ..other stuff..
-----
 heli-after-tick             : Heli -> Heli
 heli-after-mouse-event  : Heli Number Number MouseEvent -> Heli
 heli-to-scene               : Heli Scene -> Scene

 Bombs = ListOf<Bomb>
-----
 bombs-after-tick            : Bombs -> Bombs
 bombs-after-mouse-event : Bombs Number Number MouseEvent -> Bombs
 bombs-to-scene              : Bombs Scene -> Scene

 Bomb = Posn * Radius
-----
 bomb-after-tick             : Bomb -> Bomb
 bomb-after-mouse-event  : Bomb Number Number MouseEvent -> Bomb
 bomb-to-scene               : Bomb Scene -> Scene
```

Place an image of the helicopter on the given scene

Observe Redundancies

```
World = Heli * Bombs * ..other stuff..
----
world-after-tick         : World -> World
world-after-mouse-event  : World Number Number MouseEvent -> World
world-to-scene           : World Scene -> Scene

Heli = Posn * ..other stuff..
----
heli-after-tick          : Heli -> Heli
heli-after-mouse-event   : Heli Number Number
heli-to-scene            : Heli Scene -> Scene

Bombs = ListOf<Bomb>
-----
bombs-after-tick         : Bombs -> Bombs
bombs-after-mouse-event  : Bombs Number Numbe
bombs-to-scene           : Bombs Scene -> Sce

Bomb = Posn * Radius
-----
bomb-after-tick          : Bomb -> Bomb
bomb-after-mouse-event   : Bomb Number Number Mo
bomb-to-scene            : Bomb Scene -> Scene
```

all functions for world have name world- and take World as first arg

all functions for Heli have name heli- and take Heli as first arg

all functions for Bombs have name bombs- and take Bombs as first arg

all functions for Bomb have name bomb- and take Bomb as first arg

but result types can be different

Can we abstract on this?

Want to write World, etc. only once

```
 World = Heli * Bombs * ..other stuff..
-----
 world-after-tick           : World -> World
 world-after-mouse-event : World Number Number MouseEvent -> World
 world-to-scene             : World Scene -> Scene

Heli = Posn * ..other stuff..
-----
 heli-after-tick            : Heli -> Heli
 heli-after-mouse-event  : Heli Number Number MouseEvent -> Heli
 heli-to-scene              : Heli Scene -> Scene

Bombs = ListOf<Bomb>
-----
 bombs-after-tick           : Bombs -> Bombs
 bombs-after-mouse-event : Bombs Number Number MouseEvent -> Bombs
 bombs-to-scene             : Bombs Scene -> Scene

Bomb = Posn * Radius
-----
 bomb-after-tick            : Bomb -> Bomb
 bomb-after-mouse-event  : Bomb Number Number MouseEvent -> Bomb
 bomb-to-scene              : Bomb Scene -> Scene
```

These are *classes!*

Class = Data Definition + Methods

```
{World = Heli * Bombs * ..other stuff..
-----
       after-tick          :       -> World
       after-mouse-event   :       Number Number MouseEvent -> World
       to-scene            :       Scene -> Scene}

{Heli = Posn * ..other stuff..
-----
      after-tick           :      -> Heli
      after-mouse-event    :      Number Number MouseEvent -> Heli
      to-scene             :      Scene -> Scene}

{Bombs = ListOf<Bomb>
-----
       after-tick          :       -> Bombs
       after-mouse-event   :       Number Number MouseEvent -> Bombs
       to-scene            :       Scene -> Scene}

{Bomb = Posn * Radius
-----
      after-tick           :      -> Bomb
      after-mouse-event    :      Number Number MouseEvent -> Bomb
      to-scene             :      Scene -> Scene}
```

The Racket Class System

- We will use full PLT Racket
- #lang racket
- example: 08-1-space-invaders.rkt

Testing in an object-oriented world

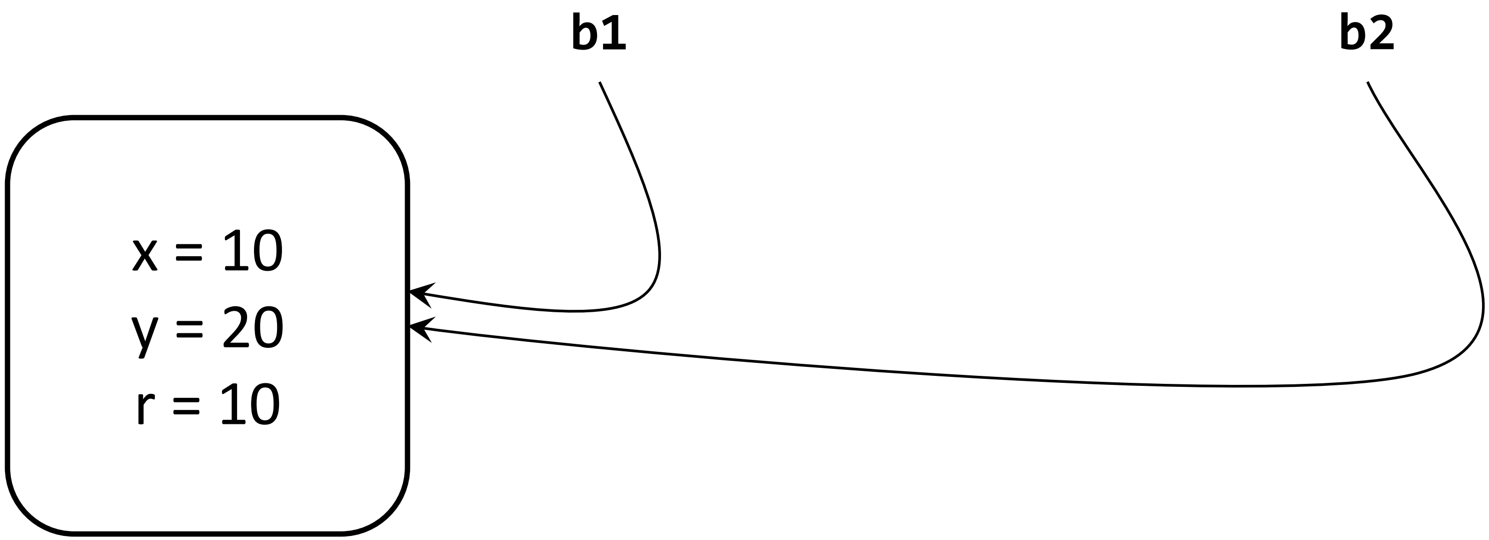
- Objects have identity!

A Bomb

x = 10
y = 20
r = 10

One bomb or two?

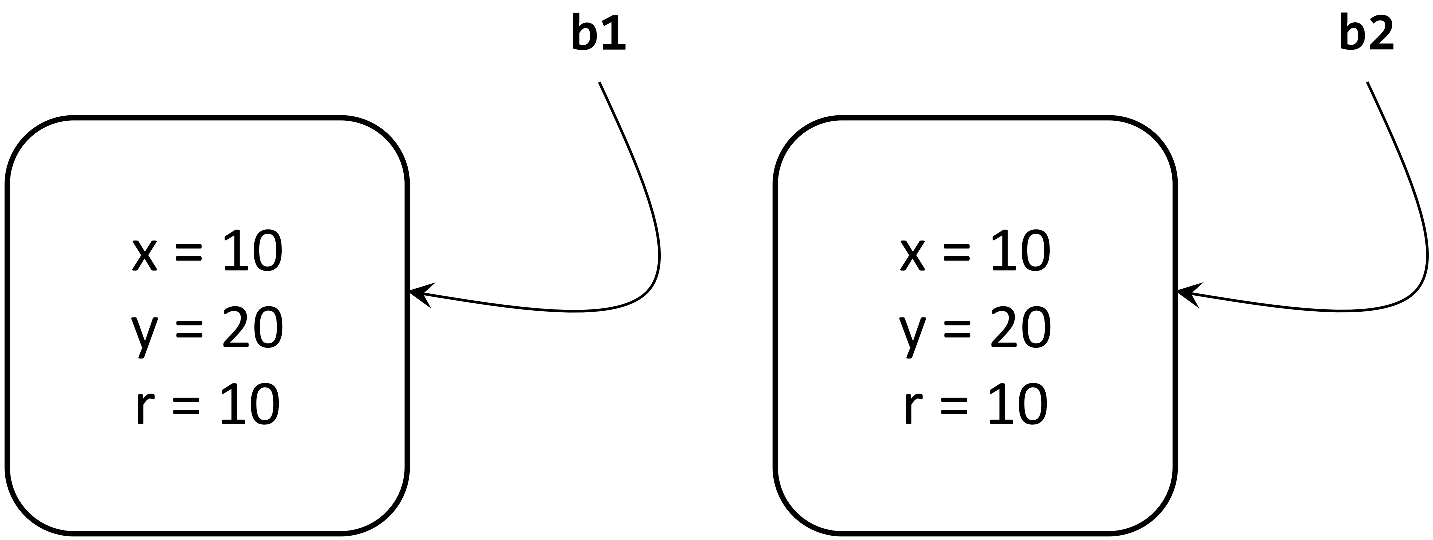
`(define b1 (make-bomb))` `(define b2 b1)`



`(check-equal? b1 b2)` ➔ `true`

One bomb or two?

`(define b1 (make-bomb))` `(define b2 (make-bomb))`



`(check-equal? b1 b2) -> false`

Workaround: Test Properties instead

- Add getter methods to look at properties of the object.
- This is OK now, but is considered bad OO design in general
- For unit tests only
- Make sure these have restricted visibility
  - different OO languages have different techniques for this
  - we're just not going to worry about it
- **`bomb-similar?`**
  - 08-2-with-getters.rkt

Redundancy #2

All these classes have the same methods!

```
{World = Heli * Bombs * ..other stuff..
-----
      after-tick           :      -> World
      after-mouse-event    :      Number Number MouseEvent -> World
      to-scene             :      Scene -> Scene}

{Heli = Posn * ..other stuff..
-----
     after-tick            :     -> Heli
     after-mouse-event     :     Number Number MouseEvent -> Heli
     to-scene              :     Scene -> Scene}

{Bombs = ListOf<Bomb>
-----
      after-tick           :      -> Bombs
      after-mouse-event    :      Number Number MouseEvent -> Bombs
      to-scene             :      Scene -> Scene}

{Bomb = Posn * Radius
-----
     after-tick            :     -> Bomb
     after-mouse-event     :     Number Number MouseEvent -> Bomb
     to-scene              :     Scene -> Scene}
```

and the methods have similar contracts

Let's mark the differences

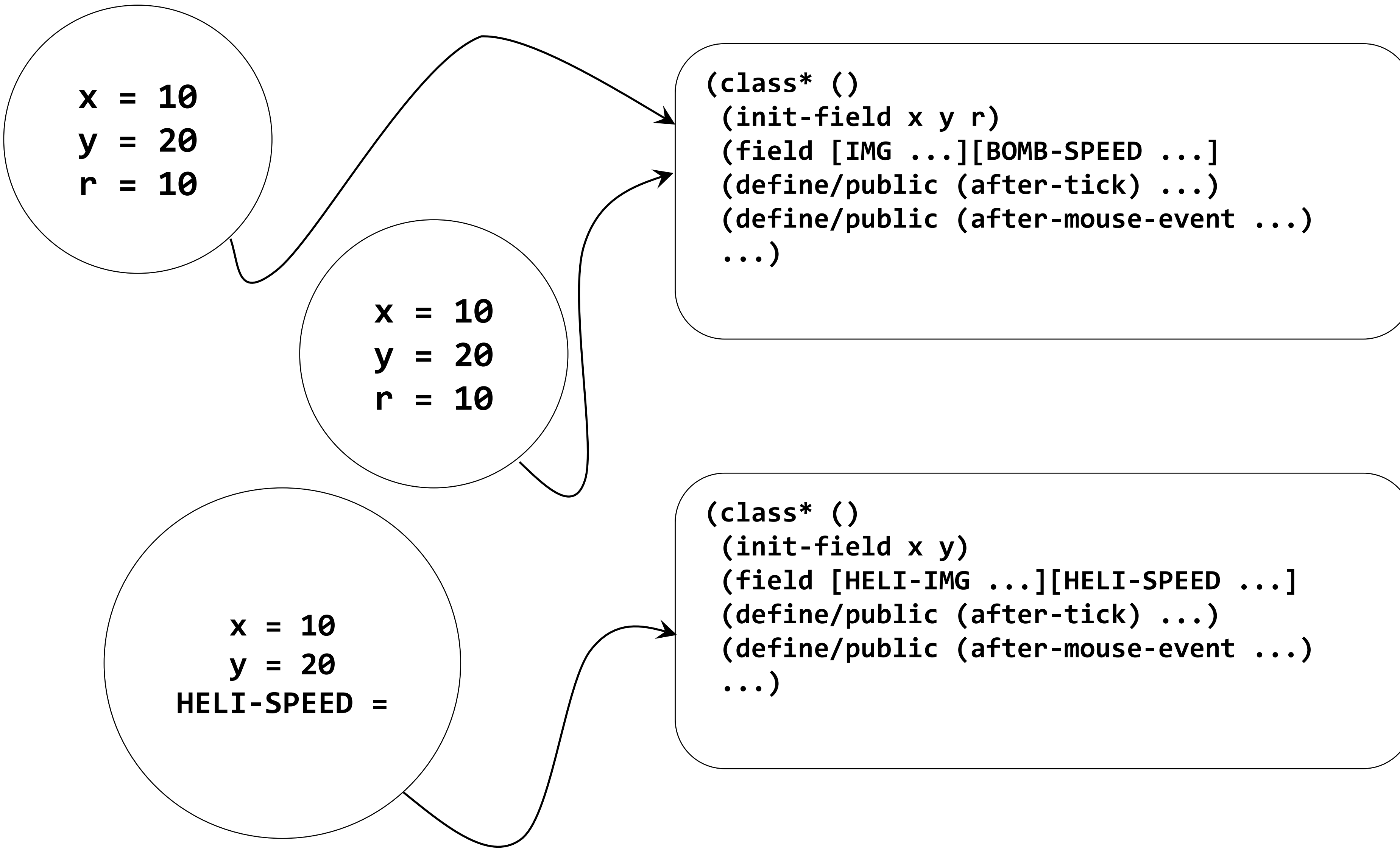
```
{World = Heli * Bombs * ..other stuff..
-----
       after-tick          :       -> World
       after-mouse-event   :       Number Number MouseEvent -> World
       to-scene            :       Scene -> Scene}

{Heli = Posn * ..other stuff..
-----
      after-tick           :       -> Heli
      after-mouse-event    :       Number Number MouseEvent -> Heli
      to-scene             :       Scene -> Scene}

{Bombs = ListOf<Bomb>
-----
       after-tick          :       -> Bombs
       after-mouse-event   :       Number Number MouseEvent -> Bombs
       to-scene            :       Scene -> Scene}

{Bomb = Posn * Radius
-----
      after-tick           :       -> Bomb
      after-mouse-event    :       Number Number MouseEvent -> Bomb
      to-scene             :       Scene -> Scene}
```

The general pattern

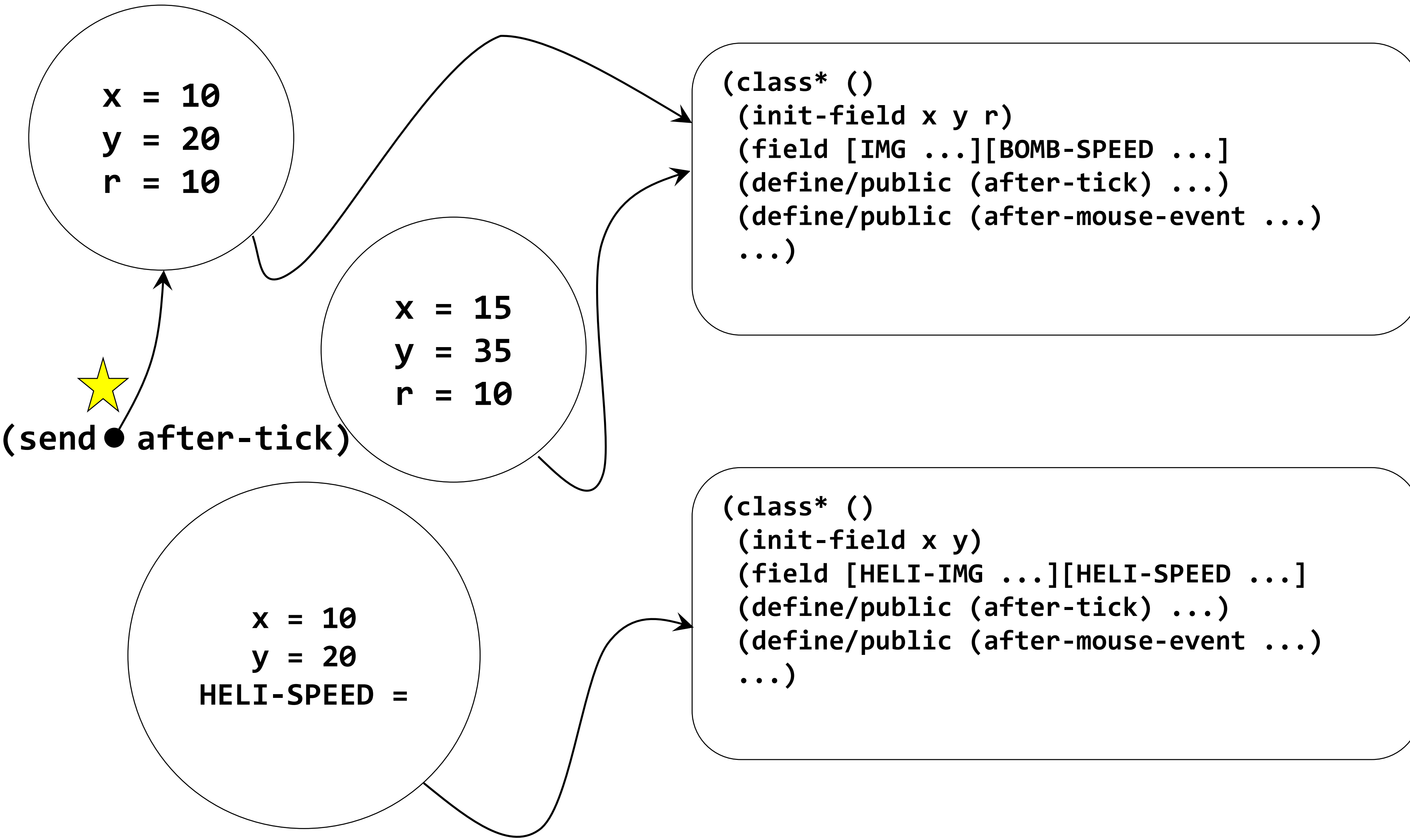
```
{Foo = ...
 -----
        after-tick          :       -> Foo
        after-mouse-event   :       Number Number MouseEvent -> Foo
        to-scene            :       Scene -> Scene}
```

- We call this an *interface.*
- An interface describes a group of classes with the same functions on them.
- World, Heli, Bombs, Bomb all *implement* this interface

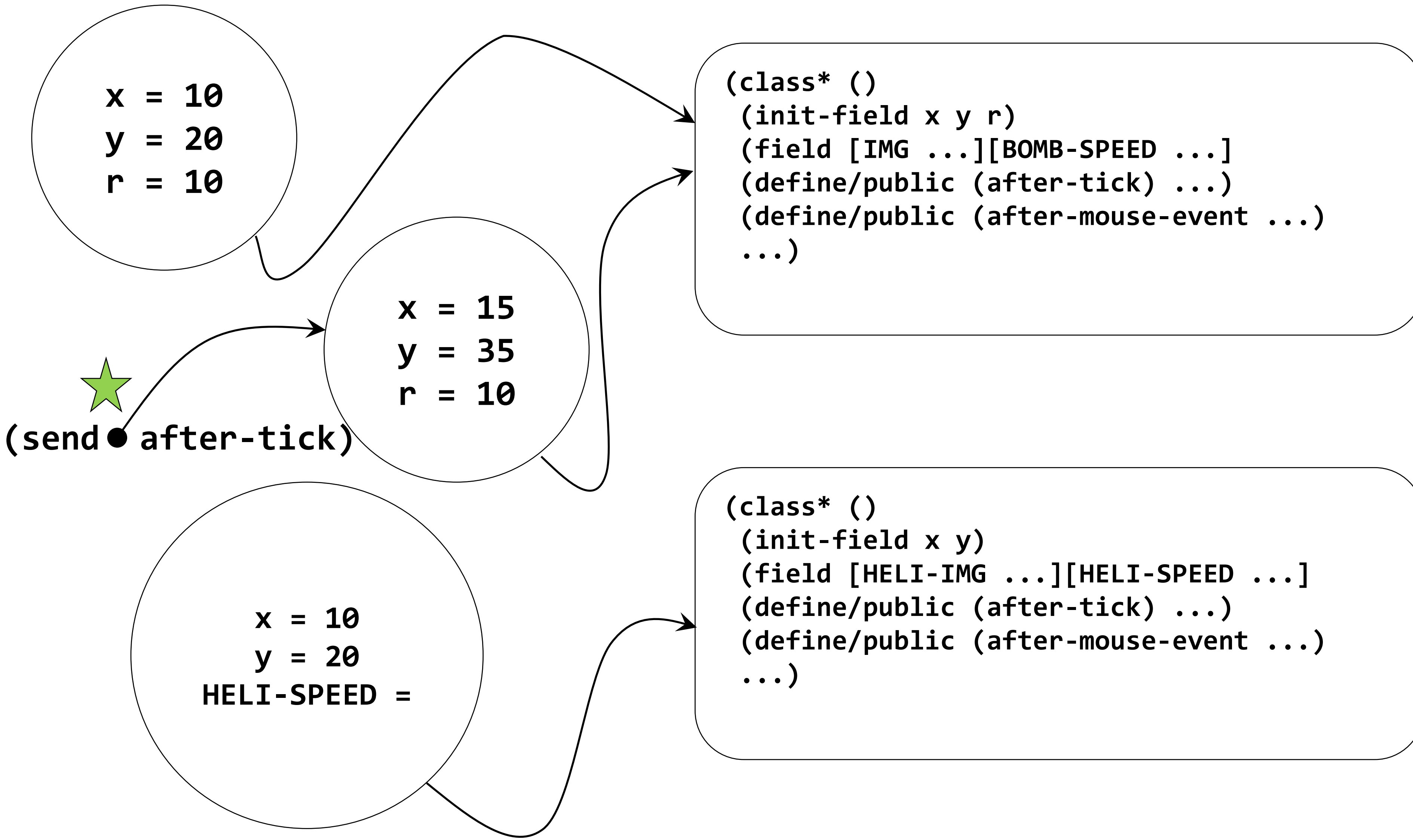
Every object knows its class



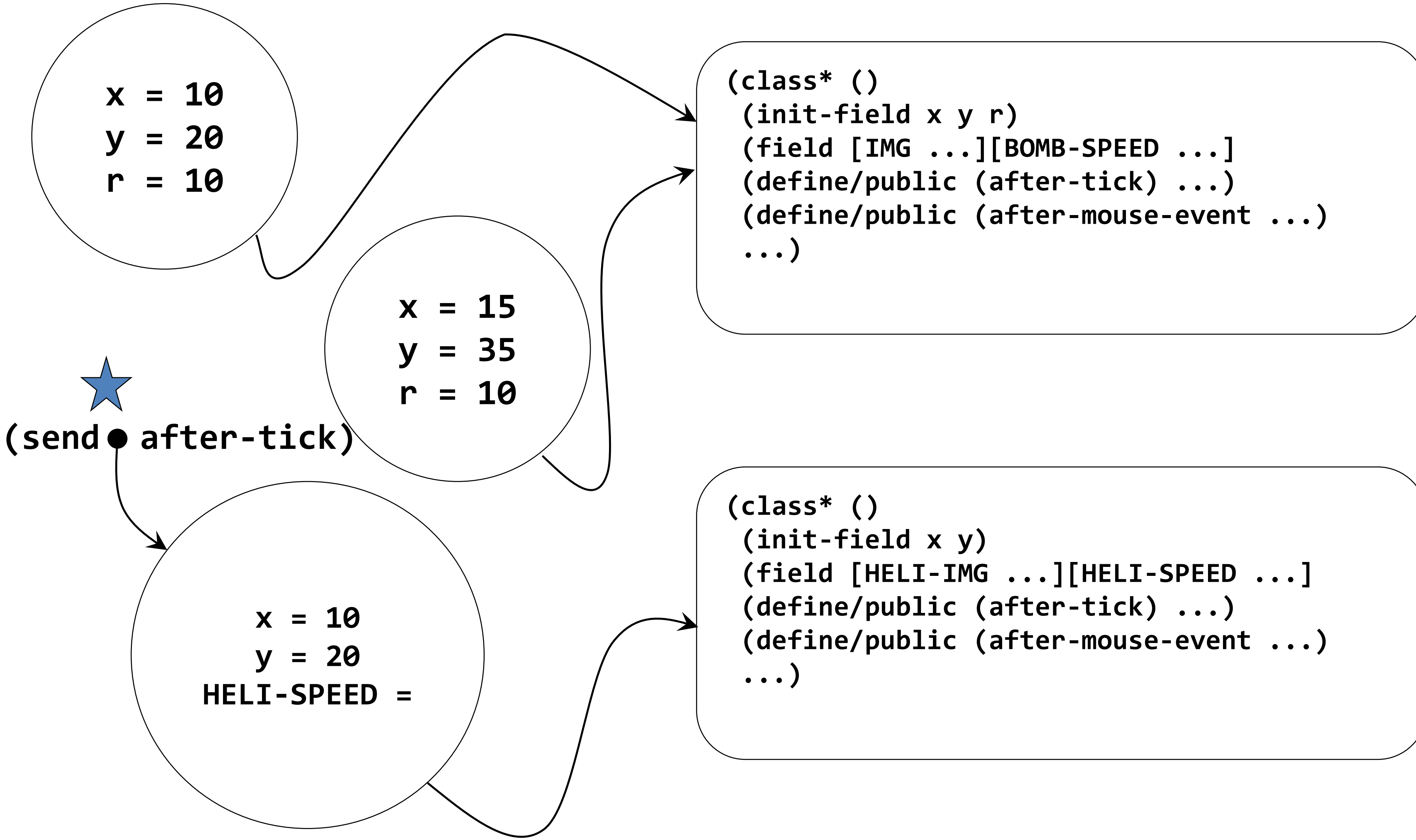
Every object knows its class



Every object knows its class



Every object knows its class



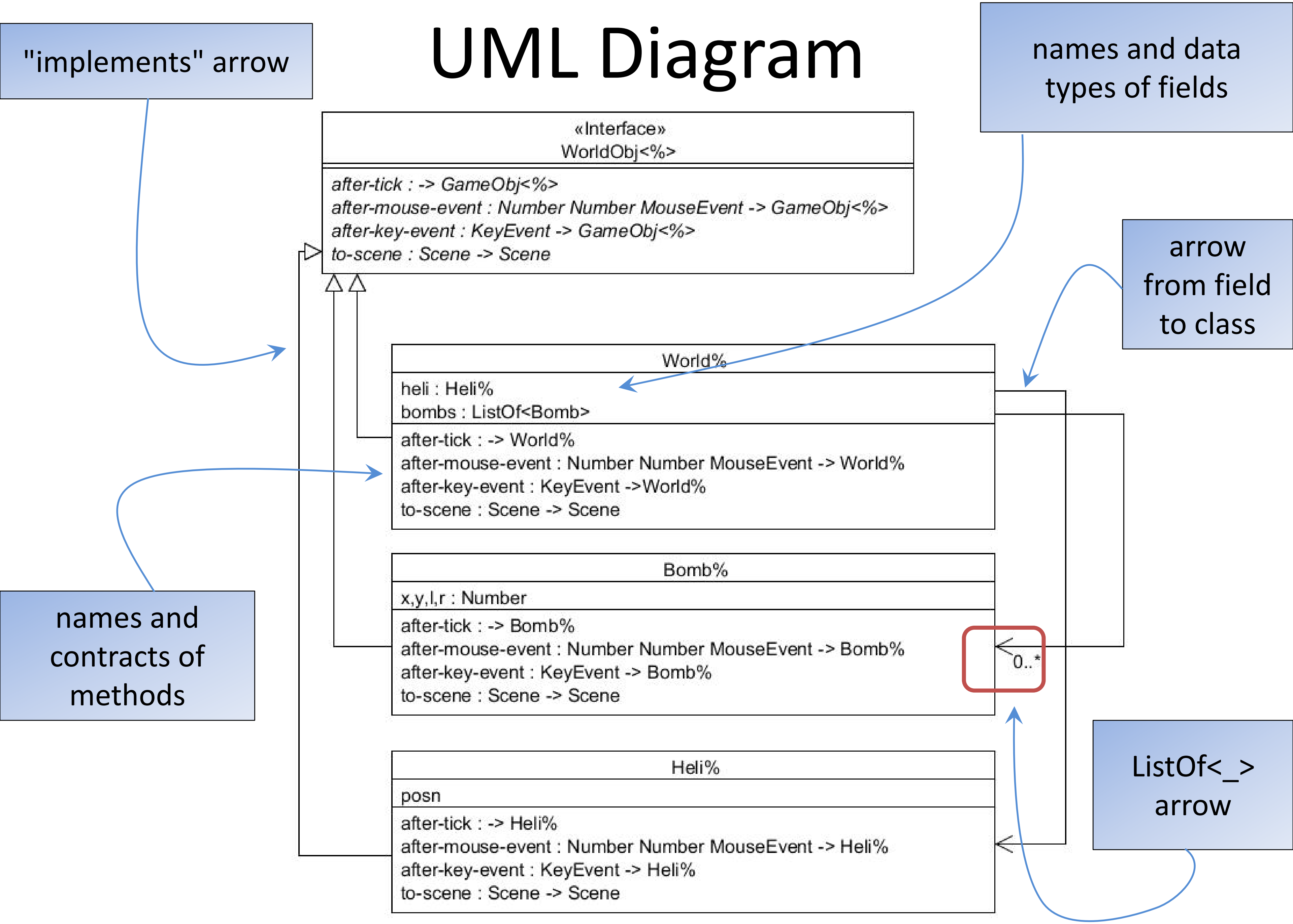
From Data Definitions to Classes and Interfaces

- Compound Data ➔ Class
- Enumeration/Mixed Data ➔ Interface
  - each variant is a class that implements that interface
- The class diagram lays out the variants
- example: [08-3-with-interfaces.rkt]

Recipe for converting from data defs to interfaces and classes

Recipe
1. Define an interface for each kind of enumeration data
2. Define a class for each kind of compound data. The class must implement the interface
3. In the interface, add a method for each function that follows the template
4. In each class, add an init-field for each field in the struct
5. In each class, add a method for each function in the interface. Define the method by taking the appropriate **cond**-line from your function, and replace: • each selector by a field reference • each function call by a **send**

UML Diagram


"implements" arrow
names and data types of fields
arrow from field to class
names and contracts of methods
ListOf<_> arrow
«Interface»
WorldObj<%>
after-tick : -> GameObj<%>
after-mouse-event : Number Number MouseEvent -> GameObj<%>
after-key-event : KeyEvent -> GameObj<%>
to-scene : Scene -> Scene
World%
heli : Heli%
bombs : ListOf<Bomb>
after-tick : -> World%
after-mouse-event : Number Number MouseEvent -> World%
after-key-event : KeyEvent ->World%
to-scene : Scene -> Scene
Bomb%
x,y,l,r : Number
after-tick : -> Bomb%
after-mouse-event : Number Number MouseEvent -> Bomb%
after-key-event : KeyEvent -> Bomb%
to-scene : Scene -> Scene
0..*
Heli%
posn
after-tick : -> Heli%
after-mouse-event : Number Number MouseEvent -> Heli%
after-key-event : KeyEvent -> Heli%
to-scene : Scene -> Scene

Interfaces Open Up New Possibilities

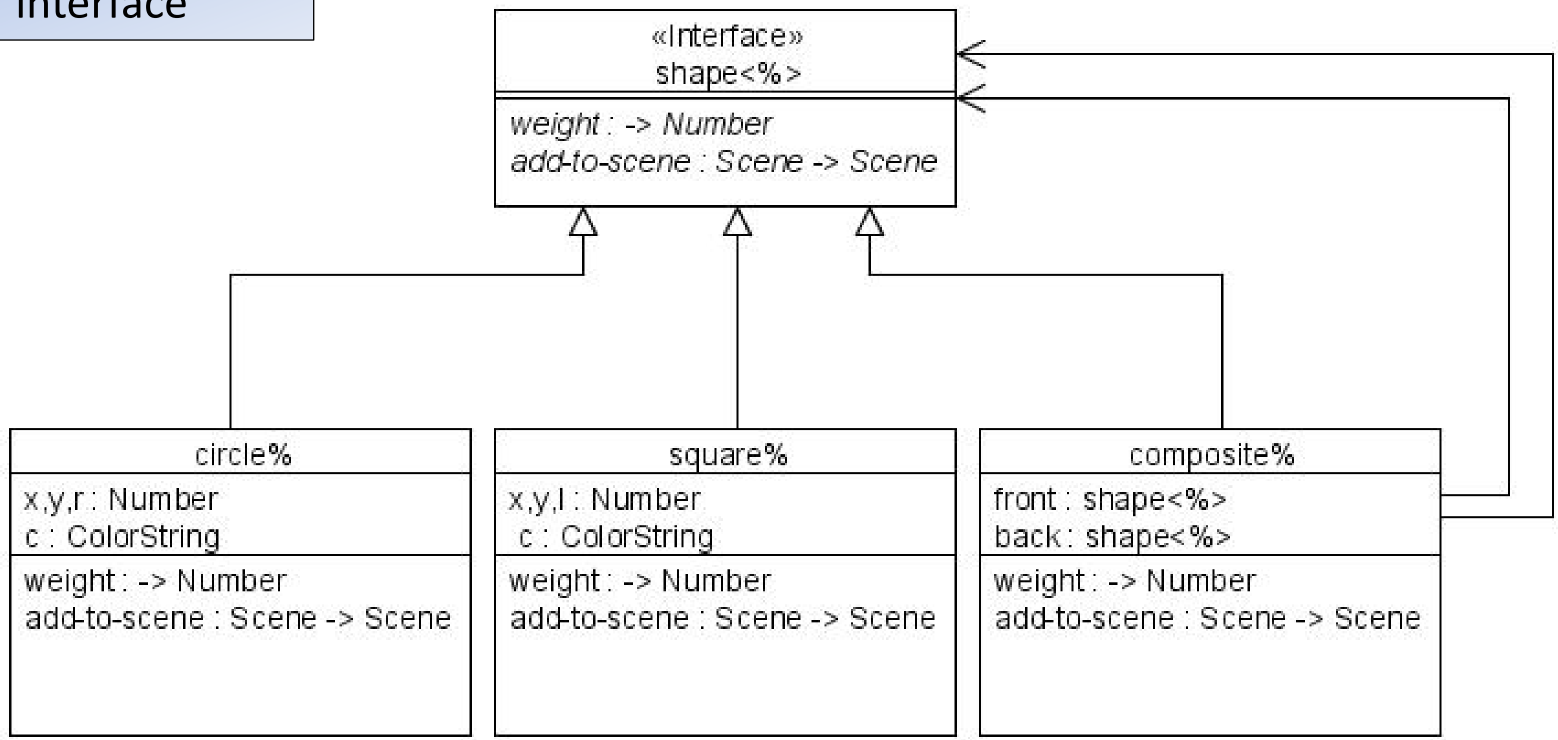
- In space-invaders, when you sent a message you always knew exactly what class the target was in
- If you know what interface an object has, you can send a message to it, even if you don't know its class.

"static dispatch"

"dynamic dispatch"

UML Class Diagram

We don't know the class of front or back, only their interface



From Functional to OOP: Take 1

- Example:
  - 08-4-shapes-functional.rkt
  - 08-5-shapes-with-separate-functions.rkt
  - 08-6-shapes-with-interfaces.rkt

Self-Referential Data

```
(define composite%
  (class* object% (shape<%>)
    (init-field front   ; Shape, the shape in front
                back)   ; Shape, the shape in back

    (super-new)

    ;; struct decomp
    (define/public (weight) (+ (send front weight)
                               (send back weight)))

    ;; all we know here is that front and back implement shape<%>.
    ;; we don't know if they are circles, squares, or other
  composites!

    ;; SD + acc [scene]
    (define/public (add-to-scene scene)
      (send front add-to-scene
            (send back add-to-scene scene)))

   ))
```

This is called the *composite* pattern

Recursion!

Doesn't care what kind of shape these are: it just works!

The Big Picture: Functional

my-circle-weight

my-square-weight

my-composite-weight

my-circle-add-to-scene

my-square-add-to-scene

my-composite-add-to-scene

define weight:

my-circle-weight

my-square-weight

my-composite-weight

define add-to-scene:

my-circle-add-to-scene

my-square-add-to-scene

my-composite-add-to-scene

The Big Picture: Classes

my-circle-weight

my-square-weight

my-composite-weight

my-circle-add-to-scene

my-square-add-to-scene

my-composite-add-to-scene

class circle:

my-circle-weight

my-circle-add-to-scene

class square:

my-square-weight

my-square-add-to-scene

class composite:

my-composite-weight

my-composite-add-to-scene

Functional vs. OO organization

Functional:	Square	Circle	Composite
weight			
add-to-scene			

OO:	Square	Circle	Composite
weight			
add-to-scene			

Adding a new data variant

Functional:	Square	Circle	Composite	Triangle
weight				!!!
add-to-scene				!!!

OO:	Square	Circle	Composite	Triangle
weight				!!!
add-to-scene				!!!

Adding a new data variant in the functional model: requires editing in multiple places
Adding a new data variant in the class model: requires editing in only one place

Adding a new operation

Functional:	Square	Circle	Composite
weight			
add-to-scene			
move	!!!	!!!	!!!

OO:	Square	Circle	Composite
weight			
add-to-scene			
move	!!!	!!!	!!!

Adding a new operation in the functional model: requires editing in only one place
Adding a new operation in the class model: requires editing in multiple places

Recipe for converting from data defs to interfaces and classes

Recipe
1. Define an interface for each kind of enumeration data
2. Define a class for each kind of compound data. The class must implement the interface
3. In the interface, add a method for each function that follows the template
4. In each class, add an init-field for each field in the struct
5. In each class, add a method for each function in the interface. Define the method by taking the appropriate **cond**-line from your function, and replace: • each selector by a field reference • each function call by a **send**

Summary

- We've seen how classes and interfaces arise from related sets of functions
- We've seen how to translate from data definitions to classes and interfaces
- We've seen how these ideas are expressed in the Racket object system

The DD➔OO Recipe

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 8.2

Goals of this lesson

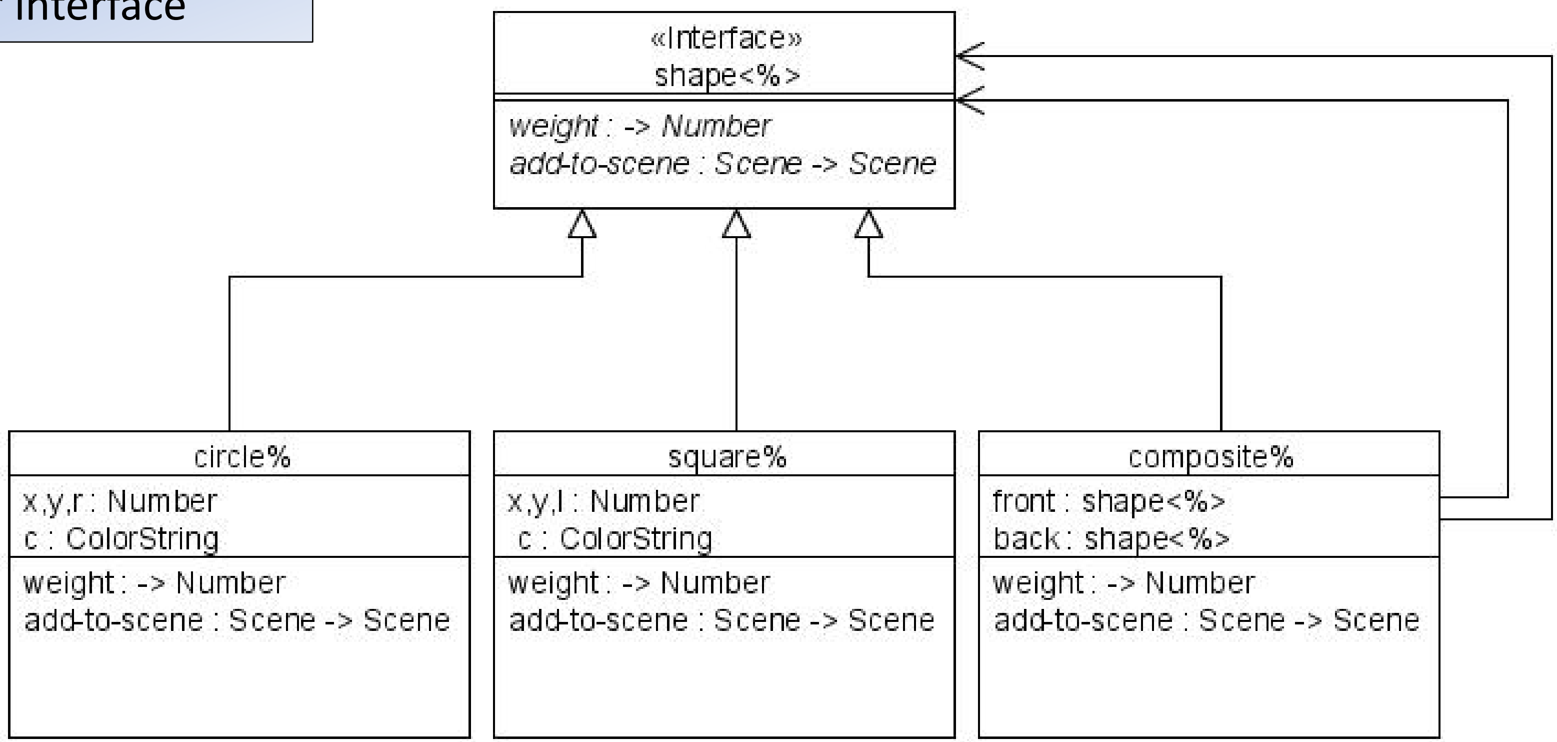
- Review how to go from data definitions to classes and interfaces
- Learn how to go from function definitions to method definitions (in more detail)

From Data Definitions to Classes and Interfaces

- Compound Data → Class
- Enumeration Data → Interface
- An interface captures a common API.

UML Class Diagram

We don't know the class of front or back, only their interface



Example: File System Interface

```
(define FileSystem<%>
  (interface ()
    open
    close
    read
    write
    ; ...
    ))
```

```
(define NTFS%
  (class* object% (FileSystem<%>)
    (init-field
     ntfs-param1 ntfs-param2)
    (define/public (open) ...)
    (define/public (close) ...)
    (define/public (read) ...)
    (define/public (write) ...)))
```

```
(define GFS%
  (class* object% (FileSystem<%>)
    (init-field
     gfs-param1 gfs-param2)
    (define/public (open) '...)
    (define/public (close) '...)
    (define/public (read) '...)
    (define/public (write) '...)))
```

```
(define mydiskC
  (new NTFS%
       [ntfs-param1 'a-value1]
       [ntfs-param2 'another-value]))
```

```
(define my-network-fs
  (new GFS%
       [gfs-param1 'a-different-value]
       [gfs-param2 'yet-another-value]))
```

Functional vs. OO organization: shapes

```
;; A Shape is one of
;; -- (make-circle ...)
;; -- (make-square ...)
;; -- (make-circle ...)
;; -- (make-composite .)
```

Functional:	Square	Triangle	Circle	Composite
weight				
render				
translate				

```
(define shape<%>
  (interface ()
    ;; -> Number
    weight

    ;; Scene -> Scene
    render

    ;; Num Num -> shape<%>
    translate ))
```

OO:	Square	Triangle	Circle	Composite
weight				
render				
translate				

Recipe for converting from data defs to interfaces and classes

Converting from Data Definitions to Interfaces and Classes
1. Define an interface for each kind of enumeration data.
2. Define a class for each kind of compound data. In the class, put in an init-field for each field in the struct.
3. Convert (make-whatever ...) to (new whatever% [field1 ...][field2 ...])
4. For each function that follows the template, add a method to the interface
5 . Convert functions to methods

Example

```
(define-struct foo
  (first-one left right))
(define-struct bar (lo hi))

;; Data Definition
;; A Baz is one of
;; -- (make-foo
        Number Baz Baz)
;; -- (make-bar
        Number Number)
```

```
(define baz<%>
  (interface ()
    ...
    ))

(define foo%
  (class* object% (baz<%>)
    (init-field
     first-one left right)
    ...
    (super-new)))

(define bar%
  (class* object% (baz<%>)
    (init-field lo hi)
    ...
    (super-new)))
```

Creating Objects

Replace

```
(make-bar 12 13)
```

by

```
(new bar% [lo 12][hi 13])
```

From function calls to method calls

Instead of saying

```
(baz-fn a-baz n)
```

say

```
(send a-baz fn n)
```

From Function Definitions to Method Definitions

Converting a Function Definition to a Method Definition
1. Add function name to interface
2. Pull out the parts
3. Change selectors to field references
4. Change function calls to method calls
5. Put method definitions into classes

Turning a function into a method

Example:

```
(define (baz-mymin b n)
  (cond
    [(foo? b) (min
                (foo-first-one b)
                (baz-mymin (foo-left b) n)
                (baz-mymin (foo-right b) n))]
    [(bar? b) (min
                 n (bar-lo b) (bar-hi b))]))
```

1. Add function name to the interface

Add to baz interface:

```
(define baz<%>
  (interface ()
    mymin     ; -> Number
    ...
    ))
```

2. Pull out the parts

```
For a foo:
 (min
  (foo-first-one b)
  (baz-mymin (foo-left b) n)
  (baz-mymin (foo-right b) n))


For a bar:
 (min
   n (bar-lo b) (bar-hi b))])
```

3. Change selectors to field references

```
For a foo:
 (min
  first-one
  (baz-mymin left           n)
  (baz-mymin right          n))
```

```
For a bar:
 (min
   n lo hi)
```

4. Change function calls to method calls

```
For a foo:
 (min
  first-one
  (send left mymin          n)
  (send right mymin         n))



For a bar:
 (min
   n lo hi)
```

5. Put method definitions into classes

```
In class foo% :
 (define/public (mymin n)
   (min
    first-one
    (send left mymin          n)
    (send right mymin         n)))
```

```
In class bar% :
 (define/public (mymin n)
   (min
     n lo hi))
```

Summary

- We've seen how an interface captures a common API.
- We've seen how to convert a data definition into an interface and a set of class definitions.
- We've seen how to convert a function definition into an interface entry and a set of method definitions.

The Design Recipe using Classes

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 8.3

Goals of this lesson

- See how the design recipe and its deliverables appear in an object-oriented system

The Design Recipe

The Design Recipe
1. Information Analysis and Data Design
2. Contract and Purpose Statement
3. Examples
4. Design Strategy
5. Code
6. Tests

Information Analysis and Data Design

- Information is what lives in the real world
- Need to decide *what part* of that information needs to be represented as data.
- Need to decide *how* that information will be represented as data
- Need to document how to *interpret* the data as information

OO gives you a head start on this: What are the real-world objects you need to model? What real-world classes do they fall into? "Object-Oriented Analysis"

Representing Data

- Compound Data ➔ Class
- Enumeration Data ➔ Interface
  - The UML Diagram lists the variants.
  - You may want to list these in your code as well
- What about the interpretation?
  - Give a *purpose statement* with each class or interface
  - Give an *interpretation* (including a type) with each init-field

What happened to the template?

- The object system does all the cond's for you.
- All that's left for you to do is to write the right-hand side of each cond-line.
  - You can use fields instead of selectors.
  - So there's no need for a separate template! (Yay!)

Contract and Purpose Statement

- Contract and purpose statement go with the *interface*.
  - Each method in the interface has the same contract and the same purpose in each class.
  - That's the point of using an interface
- You may want to repeat the contract with the method definition for reference.

Examples

- Put these with the class or with the method, whichever works best.
- Phrase examples in terms of information (not data) whenever possible

Design Strategy

- Design strategy is part of implementation, not interface
- So write down design strategy with each *method definition*.

Examples of Design Strategies

```
;; domain knowledge
(define/public (weight) (* l l))
```

we're looking at the fields

```
;; functional combination
(define/public (volume)
  (* (send this height)
     (send this area)))
```

calling methods on **this**

Examples of design strategies

```
;; structural decomposition on this
(define/public (weight)
   (+ (send front weight)
      (send back weight)))

;; function composition
(define/public (volume other-obj)
   (* (send other-obj area)
      (send other-obj height)))
```

calling methods on fields
= calling functions on fields
= structural decomposition
  on **this**

calling methods on arguments
= calling functions on arguments
= function composition

Examples of Design Strategies

```
;; structural decomposition on mev
(define/public (after-mouse mx my mev)
  (cond
    [(mouse=? mev "button-down") ...]
    [(mouse=? mev "drag") ...]
    [(mouse=? mev "button-up") ...]
    [else ...]))
```

The MouseEvent template!

Examples of Design Strategies

```
;; structural decomposition on mev AND this
(define/public (after-mouse mx my mev)
  (cond
    [(mouse=? mev "button-down") ...]
    [(mouse=? mev "drag") (send x foo)]
    [(mouse=? mev "button-up") ...]
    [else ...]))
```

The MouseEvent template!

The "SD on this" idiom

Examples of design strategies

```
;; structural decomposition on this
(define/public (weight)
   (+ (send front weight)
      (send back weight)))
```

calling methods on fields
= calling functions on fields
= structural decomposition

```
;; structural decomposition on this +
  accumulator (scene)
(define/public (add-to-scene scene)
  (send front add-to-scene
    (send back add-to-scene scene)))
```

2nd argument changes!

Examples of Design Strategies

```
;; structural decomposition + higher-order function
   combination
(define/public (after-mouse-event x y evt)
      (new World%
           [heli (send heli after-mouse-event x y evt)]
           [bombs (map
                   (lambda (bomb)
                    (send bomb after-mouse-event x y evt))
                   bombs)]))
```

Examples of Design Strategies

```
(define AdjList-Graph%
  (class* object% (Graph<%>)
    (init-field
     entries) ; ListOf<(cons Node ListOf<Node>)>
 ...
;; generative recursion
 (define/public (path? src0 tgt)
      (local
        ((define (helper srcs seen)
           ;; ListOf<Node> ListOf<Node> -> Boolean
           ;; INVARIANT:
           ;; 0. srcs and seen each have no duplicates, and are disjoint
           ;; 1. there is a path in g from src0 to each of the
           ;; nodes in srcs or seen
           ;; 2. seen is a list of all the nodes that have been removed from srcs
           ;; PURPOSE: is there a path to tgt from any of the nodes in
           ;; srcs that does not go through any node in seen?
           ;; TERMINATION: the number of nodes in seen increases by 1 at each
           ;; call, so the number of nodes NOT in seen decreases by 1.
           (cond
             [(empty? srcs) false]
             [(node=? (first srcs) tgt) true] ...))
```

Need everything here!

Summary of Design Strategies in OO Code

Characteristic	Strategy
Calculations based on fields and arguments (no method calls)	Domain Knowledge
Method call on **this**	Function Composition
Method call on argument	Function Composition
Method call on field	Structural Decomposition on **this**
Method call on field (2nd argument changes)	Structural Decomposition on **this** + accumulator
Case analysis on an argument or field (must be mixed or compound data)	Structural Decomposition on the argument or field
Two kinds of structural decomposition	Show both strategies

Design Strategies → Patterns

- In OO world, the important design strategies are at the class level.
- Examples:
  - interpreter pattern (basis for our DD→OO recipe)
  - composite pattern (eg, composite shapes)
  - container pattern (we'll use this shortly)
  - template-and-hook pattern (later)

Tests

- Checking equality of objects is usually the wrong question.
- Instead, use **`check-equal?`** on *observable* behavior.
  - see last test in 8-3-with-interfaces.rkt
  - this illustrates use of *testing scenarios*

Summary

- The Design Recipe is still there, but the deliverables are in different places
- Testing is subtle
  - we'll have more to say about that