Workshop: How to Design Class Hierarchies

Viera K. Proulx

July 19, 2004

Goals of the Workshop

Program design in the context of object-oriented language (Java), with the focus on the design of class hierarchies to represent data and on designing methods for these hierarchies.

The Daily Routine

We start with a morning lecture at 8:00 am each morning and follow with a lab. The lunch break is from noon till 1:15 pm. The afternoon starts with a lecture, again followed by more lab time. The goal is to wrap up each day by 5:00 pm.

We will be working on Unix workstations, using ProfessorJ languages within the DrScheme programming environment. Each of you will get an individual account. You should save your data on a diskette (provided) or e-mail it to your home account.

1 Monday Morning:

1.1 Goals

Recall the main points from How to Design Programs. The goal is to understand how the well structured data definition determines the structure of the program.

- Data definitions: primitive types, structures, unions
- Design recipe for data definitions:
  structure, unions, self-referential data definitions
- Design recipe for functions:

  1. analyze the problem & identify the classes of data (with examples)
  2. formulate a purpose statement and a contract
  3. make up "functional" examples
  4. create the template: what information exists to compute the desired result?
  5. program
  6. turn the examples into a test suite

- Templates:
  - for problem alternatives
  - for structure arguments
  - for union arguments
  - for self-referential arguments

1.2 Example: Data definition for structures

```
;; A Posn is a structure: Number Number
;; (define posn (x y) --- this definition already exists in DrScheme

;; An Auto is a structure: String Number Number
(define-struct auto (model tank-size mpg))

;; Examples: defining data using the constructors
(define p1 (make-posn 10 20))
(define p2 (make-posn 30 20))

(define auto1 (make-auto "Ford" 15 20))
(define auto2 (make-auto "VW" 12 15))

;; Retrieving fields of structures using the selectors:
(equal? 10 (posn-x p1))
(= 20 (posn-y p2))

(equal? "Ford" (auto-model auto1))
;(= "VW" (auto-model auto2)) Note: = compares only numbers
(= 15 (auto-tank-size auto1))
(= 15 (auto-mpg auto2))
```

1.3 Example: Data definition for unions

```
;; Data definition
;; A Shape is either
;;  - a Circle
;;  - a Rectangle
;;  - a Dot

;; We also need structure definitions for the Circle, the Rectangle, and the Dot

;; A Circle is a structure: Posn Number
(define-struct circle (center radius))


;; A Rectangle is a structure: Posn Number Number
(define-struct rect (nw width height))

;; A Dot is a Posn (alternately, could be a struct with one field that
;;   is a Posn)

;; Examples:
(define cir1 (make-circle p1 10))
(define cir2 (make-circle p2 15))

(define rect1 (make-rect p1 20 30))
(define rect2 (make-rect p2 20 10))

(define dot1 p1)
(define dot2 p2)

;; Examples of shapes:
;; cir1, cir2, rect1, tect2, dot1, and dot2 are examples of shapes
```

1.4 Example: Data definition for self-referential data

```
;; A List-of-Auto is either
;;   - the empty list, or
;;   - (cons a loa) where a is an Auto and loa is a List-of-Auto

;; Examples:
(define mt-loa empty)
(define loa1 (cons auto1 mt-loa))
(define loa2 (cons (make-auto "Audi" 15 10) loa1))
(define loa3 (cons auto2 loa2))

;; A NeLoP (Non-empty List of Posn is either
;;   - (cons Posn empty)
;;   - (cons p lop) where p is a Posn and lop is a NeLoP

;; Examples:
(define nelop1 (cons p1 empty))
(define nelop2 (cons p2 nelop1))
(define nelop3 (cons (make-posn 30 50) nelop2))

;; Test the data construction:
(equal? loa1 (cons (make-auto "Ford" 15 20) empty))
(equal? loa3 (cons (make-auto "VW" 12 15) loa2))

(equal? nelop1 (cons (make-posn 10 20) empty))
(equal? nelop3 (cons (make-posn 30 50)
                     (cons (make-posn 30 20)
                           nelop1)))

;; Selectors for cons lists
(equal? auto1 (first loa1))
(equal? loa2 (rest loa3))

(equal? p1 (first nelop1))
(equal? (make-posn 30 50) (first nelop3))
(equal? nelop1 (rest nelop2))
```

1.5 Examples: Data definitions for mutually referential data

```
;; A Combo is a structure of Shapes Shapes
(define-struct combo (top bottom))

;; A Shapes is either
;;  - a Circle
;;  - a Rectangle
;;  - a Combo

;; Examples of Shapes
;; remember earlier definitions of cir1, cir2, rect1,and rect2

(define combo1 (make-combo cir1 cir2))
(define combo2 (make-combo rect1 combo1))
(define combo3 (make-combo combo2 rect2))

;; Examples of selectors:
(equal? combo2 (combo-top combo3))
(equal? rect2 (combo-bottom combo3))
(equal? cir1 (combo-top combo1))
(equal? cir2 (combo-bottom combo1))
```

1.6 Example: Design recipe for functions

```
;; Purpose and Contract:
;; compute the distance a vehicle can travel on one tank,
;; given its size and the mpg consumption
;; Number Number -> Number
;; (define (range t-size mpg)

;; Examples:
;; (range 20 10) "should be" 200
;; (range 15 20) "should be" 300

;; Template:
;; ... t-size... ...mpg...

;; Program:
(define (range t-size mpg)
  (* t-size mpg))

;; Tests:
(= (range 20 10) 200)
(= (range 15 20) 300)
```

1.7 Example: Design recipe: functions with structure args

```
;; Purpose and Contract:
;; compute the distance an auto can travel on one tank of fuel
;; Auto -> Number
;; (define (auto-range a)

;; Examples:
;; (auto-range auto1) "should be" 300
;; (auto-range auto2) "should be" 180

;; Template:
;; ... (auto-model a) ...
;; ... (auto-tank-size a) ...
;; ... (auto-mpg a) ...

;; Program:
(define (auto-range a)
  (* (auto-tank-size a) (auto-mpg a)))

;; Tests
(= (auto-range auto1) 300)
(= (auto-range auto2) 180)
```

Example: one `Auto` argument

```
;; Purpose and Contract:
;; determine whether an auto can travel the desired distance on one tank of gas
;; Auto Number -> Boolean
;; (define (reach? a dist)

;; Examples:
;; (reach? auto1 200) "should be" true
;; (reach? auto1 320) "should be" false
;; (reach? auto2 200) "should be" false

;; Template:
;; ... (auto-model a) ...
;; ... (auto-tank-size a) ...
;; ... (auto-mpg a) ...

;; ... (auto-range a) ...

;; Program:
(define (reach? a dist)
  (< dist (auto-range a)))

;; Tests
(equal? (reach? auto1 200) true)
(equal? (reach? auto1 320) false)
(equal? (reach? auto2 200) false)
```

Example: two `Auto` arguments

```
;; Purpose and Contract:
;; determine whether one auto can travel farther than another
;; Auto Auto -> Boolean
;; (define (better-range a1 a2)...

;; Examples:
;; (better-range auto1 auto2) "should be" true
;; (better-range auto2 auto1) "should be" false

;; Template:
;; ... (auto-model a1) ...     ... (auto-model a2) ...
;; ... (auto-tank-size a1) ... ... (auto-tank-size a2) ...
;; ... (auto-mpg a1) ...       ... (auto-mpg a2) ...

;; ... (auto-range a1) ...     ... (auto-range a2) ...

;; Program:
(define (better-range? a1 a2)
  (> (auto-range a1) (auto-range a2)))

;; Tests
(equal? (better-range? auto1 auto2) true)
(equal? (better-range? auto2 auto1) false)
```

Example: two `Posn` arguments

```
;; Contract and Purpose
;; Posn Posn -> Number
;; compute the distance between p1 and p2
;; (define (dist p1 p2) ...

;; Examples:
;; (= (dist p1 p2) 20)
;; (= (dist p2 p3) 30)
;; (= (dist p3 (make-posn 50 50)) 20)

;; Template:
;; ... (posn-x p1) ... (posn-x p2)...
;; ... (posn-x p2) ... (posn-y p2) ...

;; Program:
(define (dist p1 p2)
  (+ (abs (- (posn-x p1) (posn-x p2)))
     (abs (- (posn-y p1) (posn-y p2)))))

;; Tests:
(= (dist p1 p2) 20)
(= (dist p2 p3) 30)
(= (dist p3 (make-posn 50 50)) 20)
```

1.8 Example: Design recipe for functions (using union arguments)

```
;; Contract and Purpose
;; Shape -> Number
;; compute the distance of a shape s from the origin
;; (define (distTo0 s) ...

;; Examples:
;; (= (distTo0 cir1) 30)
;; (= (distTo0 cir2) 50)
;; (= (distTo0 rect1) 30)
;; (= (distTo0 rect2) 50)
;; (= (distTo0 dot1) 30)
;; (= (distTo0 dot2) 80)

;; Template:
;; ... (cond
;;       [(circle? s) ...]
;;       [(rect? s)   ...]
;;       [(dot? s)    ...]

;; But, looking at the structures for each kind of shape we get:
;; ... (cond
;;       [(circle? s) ...(circle-center s) ... (circle-radius s) ...]
;;       [(rect? s)   ...(rect-nw s) ... (rect-width s) ... (rect-height s) ...]
;;       [(dot? s)    ...(posn-x s) ... (posn-y s) ...]

;; Program:
(define (distTo0 s)
  (cond
   [(circle? s) (dist (circle-center s) (make-posn 0 0))]
   [(rect? s)   (dist (rect-nw s) (make-posn 0 0))]
   [(posn? s)    (dist s (make-posn 0 0))] ))


;; Tests:
(= (distTo0 cir1) 30)
(= (distTo0 cir2) 50)
(= (distTo0 rect1) 30)
(= (distTo0 rect2) 50)
(= (distTo0 dot1) 30)
(= (distTo0 dot2) 50)
```

1.9 Example: Design recipe for functions (using self-referential arguments)

Example: counting the number of items in a list

```
;; A List-of-Auto is either
;;   - the empty list, or
;;   - (cons a loa) where a is an Auto and loa is a List-of-Auto

;; count the number of autos in the loa list
;; List-of-Auto -> Number
;; (define (count loa) ...

;; Examples:
;; (= (count mt-loa) 0)
;; (= (count loa1) 1)
;; (= (count loa2) 2)
;; (= (count loa3) 3)

;; Template:
;; (cond
;;   [(empty? loa) ...]
;;   [else ... (first loa) ...
;;         ... (count (rest loa)) ...]

;; Program:
(define (count loa)
  (cond
   [(empty? loa) 0]
   [else (+ 1 (count (rest loa)))]))

;; Tests:
(= (count mt-loa) 0)
(= (count loa1) 1)
(= (count loa2) 2)
(= (count loa3) 3)
```

Example: computing the totals of items in a list

```
;; compute the total of the mpg consumption for all cars int the loa list
;; List-of-Auto -> Number
;; (define (total-mpg loa)...

;; Examples:
;; (= (total-mpg mt-loa) 0)
;; (= (total-mpg loa1) 20)
;; (= (total-mpg loa2) 30)
;; (= (total-mpg loa3) 45)

;; Template:
;; (cond
;;   [(empty? loa) ...]
;;   [else ... (first loa)
;;             ... (auto-model (first loa)) ...
;;             ... (auto-tank-size (first loa))...
;;             ... (auto-mpg (first loa)) ...
;;         ... (total-mpg (rest loa)) ...]

;; Program:
(define (total-mpg loa)
  (cond
   [(empty? loa) 0]
   [else (+ (auto-mpg (first loa)) (total-mpg (rest loa)))]))

;; Tests:
(= (total-mpg mt-loa) 0)
(= (total-mpg loa1) 20)
(= (total-mpg loa2) 30)
(= (total-mpg loa3) 45)
```

Example: using helper functions

```
;; compute the average mpg for all cars in the loa list
;; List-of-Auto -> Number
;; (define (avg-mpg loa)

;; Notice: we will really need two functions:

;; compute the total of the mpg consumption for all cars int the loa list
;; List-of-Auto -> Number
;; (define (total-mpg loa)...

;; count the number of autos in the loa list
;; List-of-Auto -> Number
;; (define (count loa) ...

;; Need to make a decision, how to handle the empty case
;; Decide to produce average to be 0

;; Examples:
;; (= (avg-mpg mt-loa) 0)
;; (= (avg-mpg loa1) 20)
;; (= (avg-mpg loa2) 15)
;; (= (avg-mpg loa3) 15)

;; Template:
;; (cond
;;   [( empty? loa) ...]
;;   [else  ... (first loa) ...
;;          ... (rest loa) ...]

;; Notice, we cannot apply avg-mpg to the (rest loa)
;; we use the wish list helper functions instead

;; Program:
(define (avg-mpg loa)
  (cond
    [(empty? loa) 0]
    [else (/ (total-mpg loa) (count loa))]))

;; Tests for the avg-mpg function:
(= (avg-mpg mt-loa) 0)
(= (avg-mpg loa1) 20)
(= (avg-mpg loa2) 15)
(= (avg-mpg loa3) 15)
```

Example: Design recipe for self-referential data

We are using the earlier definitions of `Circle` and `Rectangle`, and we leave it to the reader to design the helper functions `(c-within c p)` and `(r-within r p)` that determine whether the `Posn p` is within the `Circle c` or `Rectangle r` respectively.

```
;; A Combo is a structure of Shapes Shapes
(define-struct combo (top bottom))

;; A Shapes is either
;;  - a Circle
;;  - a Rectangle
;;  - a Combo

;; Examples of Shapes
;; remember earlier definitions of cir1, cir2, rect1,and rect2

(define combo1 (make-combo cir1 cir2))
(define combo2 (make-combo rect1 combo1))
(define combo3 (make-combo combo2 rect2))


;; determine whether p is within the shapes cs
;; Shapes -> Boolean
;; (define (within? cs p) ...

;; Examples (to be used as tests after the program is written)
;; (equal? (within? combo1 (make-posn 30 10)) true)
;; (equal? (within? combo1 (make-posn 30 50)) false)
;; (equal? (within? combo3 (make-posn 50 25)) true)
;; (equal? (within? combo2 (make-posn 50 25)) false)

;; Template:
;;  (cond
;;    [(circle? cs) ...]
;;    [(rect? cs)   ...]
;;    [else ... (within? (combo-top cs) p) ...
;;          ... (within? (combo-bottom cs) ...]

;; Program:
(define (within? cs p)
  (cond
   [(circle? cs) (c-within? cs p)]
   [(rect? cs)   (r-within? cs p)]
   [else (or (within? (combo-top cs) p)
             (within? (combo-bottom cs) p))]))
```

2 Monday Afternoon:

2.1 Goals

Introduction to the design recipe for class-based data definitions: class definition, class diagram, the constructor, examples of the instances of data.

1. simple classes
2. classes with containment
3. unions of classes

2.2 Example: Definition of the class `Posn`

```
/*
+-------+
| Posn  |
+-------+
| int x |
| int y |
+-------+
*/

// to represent one point on  a canvas
class Posn {
  int x;
  int y;

  Posn(int x, int y) {
    this.x = x;
    this.y = y;
  }
}

/* Interactions:
Examples of constructing instances of Posn

Posn p1 = new Posn(10, 20);
Posn p2 = new Posn(20, 20);

Examples of the use of the field selectors for the class Posn

p1.x  -- expected: 10
p1.y  -- expected: 20
p2.x  -- expected: 20
p2.y  -- expected: 20
*/
```

2.3 Example: Definition of the class `Book` - version 1

```
/*
+-------------------+
| Book              |
+-------------------+
| String title      |
| String authorName |
| int price         |
| int year          |
+-------------------+
*/

// to represent a book
class Book {
  String title;
  String authorName;
  int price;
  int year;

  Book(String title, String authorName, int price, int year) {
    this.title = title;
    this.authorName = authorName;
    this.price = price;
    this.year = year;
  }
}

/* Interactions:
Examples of the use of the constructor for the class Book

Book b1 = new Book("HtDP", "Matthias", 60, 2001);
Book b2 = new Book("Beach Music", "Conroy", 20, 1996);

Examples of the use of the field selectors for the class Book

b1.title       -- expected: "HtDP"
b1.authorName  -- expected: "Matthias"
b1.price       -- expected: 60
b1.year        -- expected: 2001

b2.title       -- expected: "Beach Music"
b2.authorName  -- expected: "Pat Conroy"
b2.price       -- expected: 20
b2.year        -- expected: 1996
*/
```

2.4 Example: Definition of the class `Author`

```
/*
+-------------+
| Author      |
+-------------+
| String name |
| int dob     |
+-------------+
*/

// to represent an author of a book
class Author {
  String name;
  int dob;

  Author(String name, int dob) {
    this.name = name;
    this.dob = dob;
  }
}

/*Interactions:
Examples of the use of the constructor

Author mf = new Author("Matthias", 1900);
Author pc = new Author("Pat Conroy", 1940);
Author eap = new Author("Annie", 1046);

Examples of the use of the field selectors for the class Author

mf.name  -- expected: "Matthias"
mf.dob   -- expected: 1900

pc.name  -- expected: "Pat Conroy"
pc.dob   -- expected: 1940
*/
```

2.5 Example: Definition of the class `Circle`

```
/*
+-------------+      +-------+
| Circle      |  +-->| Posn  |
+-------------+  |   +-------+
| Posn center |--+   | int x |
| int radius  |      | int y |
+-------------+      +-------+
*/

// to represent a circle
class Circle {
  Posn center;
  int radius;

  Circle(Posn center, int radius) {
    this.center = center;
    this.radius = radius;
  }
}

/* Intreractions:
Examples of the use of the constructor for the class Circle

Circle c1 = new Circle(new Posn(10, 20), 10);
Circle c2 = new Circle(new Posn(20, 40), 15);

Examples of the use of the field selectors for the class Circle

c1.center         -- expected: new Posn(10, 20)
c1.radius         -- expected: 10

c2.center         -- expected: new Posn(20, 40)
c2.radius         -- expected: 15

c1.center.x       -- expected 10
c1.center.y       -- expected 20
c2.center.x       -- expected 20
c2.center.y       -- expected 40
*/
```

2.6 Example: Definition of the class `Rect`

```
/*
+------------+      +-------+
| Rect       |  +-->| Posn  |
+------------+  |   +-------+
| Posn nw    |--+   | int x |
| int width  |      | int y |
| int height |      +-------+
+------------+
*/

// to represent a rectangle on canvas
class Rect {
  Posn nw;
  int width;
  int height;

  Rect(Posn nw, int width, int height) {
    this.nw = nw;
    this.width = width;
    this.height = height;
  }
}

/* Interactions:
Examples of the use of the constructor for the class Rect

Rect r1 = new Rect(new Posn(10, 20), 40, 30);
Rect r2 = new Rect(new Posn(20, 40), 10, 50);

Exampples of the use of the selectors for the class Rect

r1.nw          -- expected: new Posn(10, 20)
r1.width       -- expected: 40
r1.height      -- expected: 30

r2.nw          -- expected: new Posn(20, 40)
r2.width       -- expected: 10
r2.height      -- expected: 50

r1.nw.x        -- expected 10
r1.nw.y        -- expected 20
r2.nw.x        -- expected 20
r2.nw.y        -- expected 40
*/
```

2.7 Example: Definition of the class `Book` - version 2

```
/*
+---------------+
| Book          |      +-------------+
+---------------+  +-->| Author      |
| String title  |  |   +-------------+
| Author author |- +   | String name |
| int price     |      | int dob     |
| int year      |      +-------------+
+---------------+
*/

// to represent an author of a book
class Author {
  String name;
  int dob;

  Author(String name, int dob) {
    this.name = name;
    this.dob = dob;
  }
}

// to represent a book with author
class Book {
  String title;
  Author author;
  int price;
  int year;

  Book(String title, Author author, int price, int year) {
    this.title = title;
    this.author = author;
    this.price = price;
    this.year = year;
  }
}

/* Interactions:
Examples of the use of the constructor for the class Author

Author mf = new Author("Matthias",1900);
Author pc = new Author("Pat Conroy", 1940);
```

```
Examples of the use of the constructor for the class Book - version 2

Book b1 = new Book("HtDP", mf, 60, 2001);
Book b2 = new Book("Beach Music", pc, 20, 1996);

Examples of the use of the selectors in the class Book - version 2

b1.title          -- expected: "HtDP"
b1.author         -- expected: new Author("Matthias",1900)
b1.author.name    -- expected: "Matthias"
b1.author.dob     -- expected: 1900
b1.price          -- expected: 60
b1.year           -- expected: 2001

b2.title          -- expected: "Beach Music"
b2.author         -- expected: new Author("Pat Conroy", 1940)
b2.author.name    -- expected: "Pat Conroy"
b2.author.dob     -- expected: 1940
b2.price          -- expected: 20
b2.year           -- expected: 1996
*/
```

2.8 Example: Definition of the classes that represent the union of shapes

A Shape is on of:

- a Circle: consisting of a center and a radius, where the center is a Posn and the radius is an integer
- a Rectangle: consisting of a nw, a width, and a height, where the nw is a Posn, and the width and the height are integers
- a Dot: consisting of a location, where the location is a Posn

```
/*
                        +--------+
                        | AShape |
                        +--------+
                        +--------+
                           / \
                           ---
                            |
          ---------------------------------------
          |                 |                   |
  +-------------+    +------------+    +---------------+
  | Circle      |    | Rect       |    | Dot           |
  +-------------+    +------------+    +---------------+
  | Posn center |-+  | Posn nw    |-+  | Posn location |-+
  | int radius  | |  | int width  | |  +---------------+ |
  +-------------+ |  | int height | |                    |
    +-------+     |  +------------+ |                    |
    | Posn  |<----+-----------------+--------------------+
    +-------+
    | int x |
    | int y |
    +-------+
*/

// to represent one point on  a canvas
class Posn {
  int x;
  int y;

  Posn(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

```
// to represent geometric shapes
abstract class AShape {
}

// to represent a circle
class Circle extends AShape {
  Posn center;
  int radius;

  Circle(Posn center, int radius) {
    this.center = center;
    this.radius = radius;
  }
}

// to represent a rectangle
class Rect extends AShape {
  Posn nw;
  int width;
  int height;

  Rect(Posn nw, int width, int height) {
    this.nw = nw;
    this.width = width;
    this.height = height;
  }
}

// to represent a dot
class Dot extends AShape {
  Posn location;

  Dot(Posn location) {
    this.location = location;
  }
}

/* Interactions:
Examples of the use of the constructors for the classes representing shapes
Circle c1 = new Circle(new Posn(10, 20), 15);
Circle c2 = new Circle(new Posn(40, 20), 10);
Rect r1   = new Rect(new Posn(10, 20), 20, 40);
Rect r2   = new Rect(new Posn(40, 20), 50, 10);
Dot d1    = new Dot(new Posn(10, 20));
Dot d2    = new Dot(new Posn(40, 20));
```

```
Examples of the use of field selectors for sub-classes of AShape

c1.center      -- expected: new Posn(10, 20)
c1.center.x    -- expected: 10
c1.center.y    -- expected: 20
c1.radius      -- expected: 15

c2.center      -- expected: new Posn(40, 20)
c1.center.x    -- expected: 40
c2.center.y    -- expected: 20
c2.radius      -- expected: 10

r1.nw          -- expected: new Posn(10, 20)
r1.nw.x        -- expected: 10
r1.nw.y        -- expected: 20
r1.width       -- expected: 20
r1.height      -- expected: 40

r2.nw          -- expected: new Posn(40, 20)
r2.nw.x        -- expected: 40
r2.nw.y        -- expected: 20
r2.width       -- expected: 50
r2.height      -- expected: 10

d1.location    -- expected: new Posn(10, 20)
d1.location.x  -- expected: 10
d1.location.y  -- expected: 20

d2.location    -- expected: new Posn(40, 20)
d2.location.x  -- expected: 40
d2.location.y  -- expected: 20
*/
```

2.9 Example: Definition of the class hierarchy that represents tourist attractions

A hotel has information for the guests about several kinds of attractions in the city. These can be museums, shows, and restaurants. For each of these attractions, they have the name, the location and the phone number. For restaurants there is also the information about the kind of food they serve and the price range (cheap, moderate, expensive). For each show the hotel has the show title and the time the show starts. For the museum, the available information describes the kind of museum (aquarium, natural history, a gallery, etc.) and the hours when it is open. **Write a data definition for these classes.**

```
/*
                        +---------------+     +-------+
                        | Attraction    |  +->| Posn  |
                        +---------------+  |  +-------+
                        | String name   |  |  | int x |
                        | Posn Location |--+  | int y |
                        | String phone  |     +-------+
                        +---------------+
                               / \
                               ---
                                |
          ----------------------------------------------
          |                     |                      |
  +-------------+    +-------------------+    +-----------------+
  | Museum      |    | Restaurant        |    | Show            |
  +-------------+    +-------------------+    +-----------------+
  | String kind |    | String foodType   |    | String showName |
  | int opens   |    | String priceRange |    | int startTime   |
  | int closes  |    +-------------------+    +-----------------+
  +-------------+
*/

// to describe a city tourist attraction
abstract class Attraction {
  String name;
  Posn Location;
  String phone;
}

// to describe a museum
class Museum extends Attraction {
  String kind;
  int opens;
  int closes;
```

```
  Museum(String name, Posn Location, String phone,
         String kind, int opens, int closes) {
    this.name = name;
    this.Location = Location;
    this.phone = phone;
    this.kind = kind;
    this.opens = opens;
    this.closes = closes;
  }
}

// todescribe a restaurant
class Restaurant extends Attraction {
  String foodType;
  String priceRange;

  Restaurant(String name, Posn Location, String phone,
             String foodType, String priceRange) {
    this.name = name;
    this.Location = Location;
    this.phone = phone;
    this.foodType = foodType;
    this.priceRange = priceRange;
  }
}

// to describe a show
class Show extends Attraction {
  String showName;
  int startTime;

  Show(String name, Posn Location, String phone,
       String showName, int startTime) {
    this.name = name;
    this.Location = Location;
    this.phone = phone;
    this.showName = showName;
    this.startTime = startTime;
  }
}

// We leave it to the reader to add the examples of instances
```

3 Tuesday Morning:

3.1 Goals

Design recipes for defining classes that represent self-referential data and mutually-referential data. Designing complex class hierarchies.

3.2 Example: Data definitions for a list of books

A List of Books is one of:

- an MtLoB - an empty list of books
- a ConsLoB consisting of a *fst* (a Book) and *rst* (a List of Books)

```
// to represent a book
class Book {
  String title;
  String authorName;
  int price;
  int year;

  Book(String title, String authorName, int price, int year) {
    this.title = title;
    this.authorName = authorName;
    this.price = price;
    this.year = year;
  }
}

/*        +------+
          | ALoB |<------------+
          +------+             |
          +------+             |
             / \               |
             ---               |
              |                |
      ----------------         |
      |             |          |
  +-------+    +-----------+    |
  | MtLoB |    | ConsLoB   |    |
  +-------+    +-----------+    |
  +-------+    | Book fst  |    |
               | ALoB rst  |----+
               +-----------+

*/
```

```
// to represent a list of books
abstract class ALoB {
}

// to represent an empty list of books
class MtLoB extends ALoB {

  MtLoB() {
  }
}

// to represent a constructed list of books
class ConsLoB extends ALoB {
  Book fst;
  ALoB rst;

  ConsLoB(Book fst, ALoB rst) {
    this.fst = fst;
    this.rst = rst;
  }
}

/* Interactions:
Examples of the use of the constructor for the class Book

Book b1 = new Book("HtDP", "Matthias", 60, 2001);
Book b2 = new Book("Beach Music", "Conroy", 20, 1996);

Examples of the use of constructors for the subclasses of ALoB

MtLoB mtLoB  = new MtLoB();
ConsLoB lob1 = new ConsLoB(b1, mtLoB);
ConsLoB lob2 = new ConsLoB(b2, lob1);


Examples of the use of the field selectors for the subclasses of ALoB

lob1.fst       -- expected: new Book("HtDP", "Matthias", 60, 2001)
lob1.rst       -- expected: new MtLoB()

lob2.fst       -- expected: new Book("Beach Music", "Pat Conroy", 20, 1996)
lob2.rst       -- expected: lob1

// Note: lob1.fst.title, lob1.fst.name, etc. is not legal!!! WHY??
*/
```

3.3 Example: Data definitions for a list of files

A File consists of a name (String) and size (int).

A List of Files is one of

- an MtLoF - an empty list of files
- a ConsLoF consisting of a *fst* (a File) and *rst* (a List of Files)

```
// to represent one file in a computer system
class File {
  String name;
  int size;

  File(String name, int size) {
    this.name = name;
    this.size = size;
  }
}

/* Interactions:
Examples of the use of the constructor for the class File

File f1 = new File("MyTrip", 1000);
File f2 = new File("hmwk1", 200);
File f3 = new File("hmwk3", 350);

/*
          +------+
          | ALoF |<-------------+
          +------+              |
          +------+              |
             / \                |
             ---                |
              |                 |
      ----------------          |
      |              |          |
  +-------+     +----------+    |
  | MtLoF |     | ConsLoF  |    |
  +-------+     +----------+    |    +-------------+
  +-------+     | File fst |----+--->| File        |
                | ALoF rst |----+    +-------------+
                +----------+         | String name |
                                     | int size    |
                                     +-------------+
*/
```

```
// to represent a list of files
abstract class ALoF {
}

// to represent an empty list of files
class MtLoF extends ALoF {

  MtLoF() {
  }
}

// to represent a constructed list of files
class ConsLoF extends ALoF {
  File fst;
  ALoF rst;

  ConsLoF(File fst, ALoF rst) {
    this.fst = fst;
    this.rst = rst;
  }
}

/* Interactions:
Examples of the use of the constructors for the subclasses of ALoF

MtLoF mtlof  = new MtLoF();
ConsLoF lof1  = new ConsLoF(f1, mtlof);
ConsLoF lof2  = new ConsLoF(f2, lof1);
ConsLoF lof3  = new ConsLoF(f3, lof2);

Examples of the use of the selectors for the subclasses of ALoF
lof1.fst    -- expected: f1
lof1.rst    -- expected: mtlof
lof2.fst    -- expected: f2
lof2.rst    -- expected: lof1
lof3.fst    -- expected: f3
lof3.rst    -- expected: lof2
*/
```

3.4 Example: Data definitions for a combination of shapes

A Shape is on of:

- a Circle: consisting of a center and a radius, where the center is a Posn and the radius is an integer
- a Rectangle: consisting of a nw, a width, and a height, where the nw is a Posn, and the width and the height are integers
- a Combo: consisting of a top and a bottom, where the top and the bottom are each a Shape

Note: For clarity, we start the names of *abstract* classes with the letter *A*.

```
/*
                        +-------------+
                        | AComboShape |<-------------------------+
                        +-------------+                          |
                        +-------------+                          |
                              / \                                |
                              ---                                |
                               |                                 |
          ----------------------------------------               |
          |                |                    |               |
  +-------------+    +------------+    +--------------------+    |
  | Circle      |    | Rect       |    | Combo              |    |
  +-------------+    +------------+    +--------------------+    |
  | Posn center |    | Posn nw    |    | AComboShape top    |----+
  | int radius  |    | int width  |    | AComboShape bottom |----+
  +-------------+    | int height |    +--------------------+
                     +------------+
*/

// to represent a combination of shapes
abstract class AComboShape {
}

// to represent a circle on the canvas
class Circle extends AComboShape {
  Posn center;
  int radius;

  Circle(Posn center, int radius) {
    this.center = center;
    this.radius = radius; }
}
```

```
// to represent a rectangle on a canvas
class Rect extends AComboShape {
  Posn nw;
  int width;
  int height;

  Rect(Posn nw, int width, int height) {
    this.nw = nw;
    this.width = width;
    this.height = height; }
}

// to represent a combination of two shapes
class Combo extends AComboShape {
  AComboShape top;
  AComboShape bottom;

  Combo(AComboShape top, AComboShape bottom) {
    this.top = top;
    this.bottom = bottom;
  }
}

// Intreractions:
Examples of the use of the constructors for the subclasses of AComboShape

Circle c1    = new Circle(new Posn(10, 20), 15);
Circle c2    = new Circle(new Posn(40, 20), 10);
Rect r1      = new Rect(new Posn(10, 20), 20, 40);
Rect r2      = new Rect(new Posn(40, 20), 50, 10);
Combo combo1 = new Combo(c1, r1);
Combo combo2 = new Combo(r2, combo1);
Combo combo3 = new Combo(combo2, c2);

// Challenge: paint this shape
// assume the colors are c1: red, c2: blue, r1: green, r2: yellow
```

3.5 Example: Data definitions for a list of files and directories

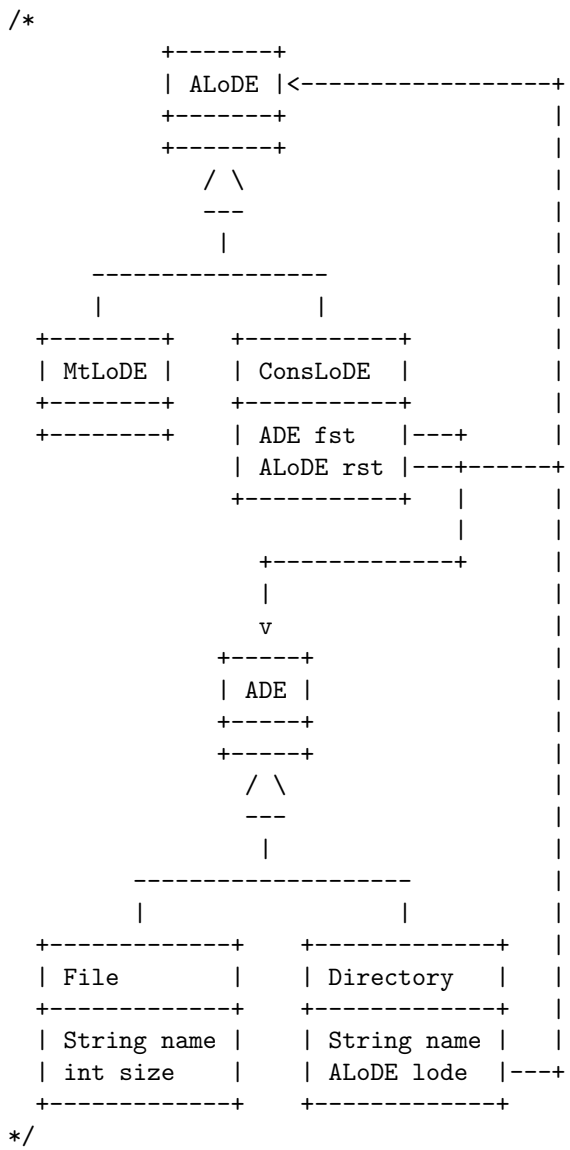
Data definition:

- A File consists of a name (String) and a size (Number)
- A Directory consists of a name (String) and a lode (ALoDE)
- An ADE (directory entry) is either
  - a File
  - a Directory
- ALoDE (a list of directory entries) is either
  - MtLoDE – an empty list of directory entries
  - ConsLoDe consisting of a fst (ADE) and a rst (ALoDE)

Challenge1: Can you design the class diagram for this class hierarchy? (Do not peak on the next page!!!)

```
// to represent one directory entry
abstract class ADE {
}

// to represent a file
class File extends ADE {
  String name;
  int size;

  File(String name, int size) {
    this.name = name;
    this.size = size; }
}

// to represent a directory
class Directory extends ADE {
  String name;
  ALoDE lode;

  Directory(String name, ALoDE lode) {
    this.name = name;
    this.lode = lode; }
}

// to represent a list of directory entries
abstract class ALoDE {
}

// to represent an empty list of directory entries
class MtLoDE extends ALoDE {

  MtLoDE() {
  }
}

// to represent a cosntructed list of directory entries
class ConsLoDE extends ALoDE {
  ADE fst;
  ALoDE rst;

  ConsLoDE(ADE fst, ALoDE rst) {
    this.fst = fst;
    this.rst = rst; }
}
```

```
/* Interactions:
Examples of files:

File f1 = new File("MyPics", 1000);
File f2 = new File("MySong", 1500);
File f3 = new File("hmwk1", 300);

Examples of ALoDE

MtLoDE mtlode  = new MtLoDE();
ConsLoDE lode1 = new ConsLoDE(f1, mtlode);
ConsLoDE lode2 = new ConsLoDE(f2, lode1);
ConsLoDE lode3 = new ConsLoDE(f3, mtlode);

Examples of Directories

Directory d1 = new Directory("Work", lode3);
Directory d2 = new Directory("Play", lode2);
Directory dir = new Directory("Main Dir",
                              new ConsLoDE(d1,
                                           new ConsLoDE(d2,
                                           mtlode)));
*/
```


/*
ALoDE
MtLoDE
ConsLoDE
ADE fst
ALoDE rst
ADE
File
String name
int size
Directory
String name
ALoDE lode
*/

4 Tuesday Afternoon:

4.1 Goals

Design recipes for methods for simple classes and classes with containment.

Design recipe for methods:

1. analyze the problem & identify the classes of data (with examples)
2. formulate a purpose statement and a contract
3. make up examples of the method invocation - with expected outcomes
4. create the template: what information exists to compute the desired result?
5. program
6. turn the examples into a test suite

We will think carefully about the structure of the templates for each kind of problems:

- simple class – no arguments or arguments of primitive type only
- simple class – argument is another instance of this class
- class with containment – no argument or a primitive argument
- class with containment – methods already defined for the contained object
- class with containment – argument is another instance of this class
- unions of classes – no arguments or arguments of primitive type only
- unions of classes – methods already defined for this union

This leads to deciding whether the method for a union of classes should be defined as an `abstract` method or a concrete method in the `abstract` class.

4.2 Example: Compute the distance of a `Posn` to the origin

1. Data Analysis and Problem Analysis

No additional data needed, as the coordinates of the origin are known. We will compute the distance in city blocks, rather than finding the length of a direct line between the origin and the `Posn` that invokes the method.

2. Purpose and Contract/Header:

```
// compute the distance of this Posn to the origin
int distTo0() { ... }
```

3. Examples:

```
Posn p1 = new Posn(10, 20);
Posn p2 = new Posn(20, 20);
// p1.distTo0()  -- expected: 30
// p2.distTo0()  -- expected: 40
```

4. Template:

The first varsion of the template is the same for all methods in a class.

```
// ** DRAFT TEMPLATE ** Edit as needed.
??? mmm() {
  ... this.x ...
  ... this.y ...
}
```

5. Program:

```
// compute the distance of this Posn to the origin
int distTo0() {
  return this.x + this.y;
}
```

6. Tests:

We use the Test Tool, or write the tests in the Interactions Box:

```
/* Interactions:
p1.distTo0() == 30
p2.distTo0() == 40
*/
```

4.3 Example: Determine whether one `Posn` is closer to the origin than another one

1. Data Analysis and Problem Analysis

The method is invoked by one `Posn`. We need another `Posn` as an argument for this method. No additional information is needed.

2. Purpose and Contract/Header:

```
// determine whether this Posn is closer to the origin
// than the given Posn
boolean closer(Posn that){...}
```

3. Examples:

```
p1.closer(p2)  -- expected: true
p2.closer(p1)  -- expected: false
```

4. Template:

```
boolean closer(Posn that) {
  ... this.x ...          ... that.x ...
  ... this.y ...          ... that.y ...
  ... this.distTo0() ... ... that.distTo0() ...
}
```

5. Program:

```
// compute the distance from this Posn to the given Posn
boolean closer(Posn that){
  return this.distTo0() < that.distTo0();
}
```

6. Tests:

```
p1.closer(p2) == true
p2.closer(p1) == false
```

4.4 Example: Determine whether a book costs less than the given amount

1. Data Analysis and Problem Analysis

The book in question invokes the method. The only additional information needed is the desired amount. We expect it to be an integer.

2. Purpose and Contract/Header:

```
// determine whether this book is cheaper than the given amount
boolean cheaperThan(int amount){...}
```

3. Examples:

```
b1.cheaperThan(100)  -- expected: true
b2.cheaperThan(10)   -- expected: false
```

4. Template:

```
boolean cheaperThan(int amount){...}
  ... this.title ...
  ... this.author ...
  ... this.price ...
  ... this.year ...
}
```

5. Program:

```
boolean cheaperThan(int amount){
  return this.price < amount;
}
```

6. Tests:

```
b1.cheaperThan(100) == true
b2.cheaperThan(10)  == false
```

4.5 Example: Determine which of two books has been published earlier

1. Data Analysis and Problem Analysis

We need two books - this book which invokes the method and that book which will be the argument for the method.

2. Purpose and Contract/Header:

```
// determine whether this book was published before that book
boolean olderThan(Book that){...}
```

3. Examples:

```
b1.olderThan(b2)  -- expected: false
b2.olderThan(b1)  -- expected: true
```

4. Template:

We need to add to the basic template the fields of `that` book.

```
boolean olderThan(Book that){...}
  ... this.title ...    ... that.title ...
  ... this.author ...   ... that.title ...
  ... this.price ...    ... that.title ...
  ... this.year ...     ... that.title ...
}
```

5. Program:

```
boolean olderThan(Book that){...}
 return this.year < that.year;
}
```

6. Tests:

```
b1.olderThan(b2) == false
b2.olderThan(b1) == true
```

4.6 Example: Compute the distance of a rectangle to the origin

1. Data Analysis and Problem Analysis

The rectangle which invokes the method has all the information needed to solve the problem.

2. Purpose and Contract/Header:

```
// compute the distance of this Rect to the origin
int distTo0() { ... }
```

3. Examples:

```
Rect r1 = new Rect(new Posn(10, 20), 40, 30);
Rect r2 = new Rect(new Posn(20, 20), 10, 50);
...
r1.distTo0()  -- expected: 30
r2.distTo0()  -- expected: 40
```

4. Template:

We notice that a method `distTo0()` is already defined in the class `Posn` already. We add the method invocation of `distTo0()` by `Posn nw` to the template.

```
// compute the distance of this Rect to the origin
int distTo0() {
  ... this.nw ...
      ... this.nw.x ...
      ... this.nw.y ...
      ... this.nw.distTo0() ...
  ... this.width ...
  ... this.height ...
}
```

5. Program:

```
// compute the distance of this Rect to the origin
int distTo0() {
  return this.nw.distTo0();
}
```

6. Tests:

```
r1.distTo0()  == 30
r2.distTo0()  == 40
```

4.7 Example: Determine which of two books was written by an earlier author

1. Data Analysis and Problem Analysis

The problem deals with two books - this book and that book.

2. Purpose and Contract/Header:

```
// determine whether this book was written by an earlier author
// than that book
boolean earlier(Book that) { ... }
```

3. Examples:

```
Author mf = new Author("Matthias", 1900);
Author pc = new Author("Pat Conroy", 1940);
Book b1 = new Book("HtDP", mf, 60, 2001);
Book b2 = new Book("Beach Music", pc, 20, 1996);
...
b1.earlier(b2)  -- expected: true
b2.earlier(b1)  -- expected: false
```

4. Template:

The template includes the field selectors for both `this` book and for `that` book, as well as the field selectors for the respective authors.

```
boolean earlier(Book that) { ... }
  ... this.title ...            ... that.title ...
  ... this.author ...           ... that.author ...
      ... this.author.name ...      ... that.author.name ...
      ... this.author.dob ...       ... that.author.dob ...
  ... this.price ...            ... that.price ...
  ... this.year ...             ... that.year ... }
```

5. Program:

```
// determine whether this book was written by an earlier author
// than that book
boolean earlier(Book that) {
  return this.author.dob < that.author.dob;  }
```

6. Tests:

```
b1.earlier(b2) == true
b2.earlier(b1) == false
```

4.8 Example: Compute the distance of a shape from the origin

1. Data Analysis and Problem Analysis

We know how to compute the distance of a rectangle and a dot to the origin. For circle, we compromise and compute the distance of the center to the origin. Because we will have this method for every shape, we notify the `abstract` class `AShape` of its availability. By defining an `abstract` method we require that every subclass implements this method. Therefore, we only need a purpose statement for the `abstract` method.

2. Purpose and Contract/Header:

```
// compute the distance of this Shape to the origin
 abstract int distTo0();
```

We develop each method separately, but show them here concurrently.

3. Examples:

```
AShape c1 = new Circle(new Posn(10, 20), 15);
AShape c2 = new Circle(new Posn(40, 20), 10);
AShape r1 = new Rect(new Posn(10, 20), 20, 40);
AShape r2 = new Rect(new Posn(40, 20), 50, 10);
AShape d1 = new Dot(new Posn(10, 20));
AShape d2 = new Dot(new Posn(40, 20));
...
c1.distTo0()  -- expected: 30
c2.distTo0()  -- expected: 60
r1.distTo0()  -- expected: 30
r2.distTo0()  -- expected: 60
d1.distTo0()  -- expected: 30
d2.distTo0()  -- expected: 60
```

4. Template:

There is a separate template for each of the three subclasses.

```
/* TEMPLATE for the Circle class:
int distTo0() {
  ... this.center ...
      ... this.center.x ...
      ... this.center.y ...
  ... this.radius ... }  */
```

```
/* TEMPLATE for the Rect class:
 int distTo0() {
   ... this.nw ...
       ... this.nw.x ...
       ... this.nw.y ...
   ... this.width ...
   ... this.height ... }  */

/* TEMPLATE for the Dot class:
 int distTo0() {
   ... this.location ...
       ... this.location.x ...
       ... this.location.y ... }  */
```

5. Program:

Again, we have three different methods, one for each class.

```
// PROGRAM for the class Circle:
int distTo0() {
  return this.center.distTo0(); }

// PROGRAM for the class Rect:
int distTo0() {
  return this.nw.distTo0(); }

// PROGRAM for the class Dot:
int distTo0() {
  return this.location.distTo0(); }
```

6. Tests:

```
c1.distTo0()  == 30
c2.distTo0()  == 60
r1.distTo0()  == 30
r2.distTo0()  == 60
d1.distTo0()  == 30
d2.distTo0()  == 60
```

5 Wednesday Morning:

5.1 Goals

Design recipes for methods for unions of classes. Design recipes for method for classes that represent self-referential data. Abstracting common data and common behavior in unions of classes.

5.2 Example: Counting the books in a list of books

1. Data Analysis and Problem Analysis

The only data we need is the list of books.

2. Purpose and Contract/Header:

```
// PURPOSE AND CONTRACT:
// count the number of books in this list
abstract int count();
```

We develop the methods in the two classes concurrently, starting with examples.

3. Examples:

```
Book b1 = new Book("HtDP", "Matthias", 60, 2001);
Book b2 = new Book("Beach Music", "Conroy", 20, 1996);
Book b3 = new Book("3 Com", "Remarque", 15, 1937);
ALoB mtlob = new MtLoB();
ALoB lob1 = new ConsLoB(b1, mtlob);
ALoB lob2 = new ConsLoB(b2, lob1);
ALoB lob3 = new ConsLoB(b3, lob2);
...
mtlob.count()  -- expected: 0
lob1.count()   -- expected: 1
lob2.count()   -- expected: 2
lob3.count()   -- expected: 3
```

4. Template:

There is a separate template for each of the two subclasses.

```
/* TEMPLATE for the MtLoB class:
int count() {
  ...  }  */

/* TEMPLATE for the ConsLoB class:
 int count() {
   ... this.fst ...
   ... this.rst.count() ... }  */
```

5. Program:

Again, we have two different methods, one for each class.

```
// PROGRAM for the class MtLoB:
int count() {
  return 0; }

// PROGRAM for the class ConsLoB:
int count() {
  return 1 + this.rst.count(); }
```

6. Tests:

```
mtlob.count()  == 0
lob1.count()   == 1
lob2.count()   == 2
lob3.count()   == 3
```

5.3 Example: Finding a book with some title in a list of books

1. Data Analysis and Problem Analysis

We need is the list of books and the title of the book to look for.

2. Purpose and Contract/Header:

```
// PURPOSE AND CONTRACT:
// determine whether a book with the given title is in this list
abstract boolean find(String title);
```

We develop the methods in the two classes concurrently, starting with examples.

3. Examples:

```
Book b1 = new Book("HtDP", "Matthias", 60, 2001);
Book b2 = new Book("Beach Music", "Conroy", 20, 1996);
Book b3 = new Book("3 Com", "Remarque", 15, 1937);
ALoB mtlob = new MtLoB();
ALoB lob1 = new ConsLoB(b1, mtlob);
ALoB lob2 = new ConsLoB(b2, lob1);
ALoB lob3 = new ConsLoB(b3, lob2);
...
mtlob.find("Beach Music")  -- expected: false
lob1.find("Beach Music")   -- expected: false
lob2.find("Beach Music")   -- expected: true
lob3.find("HtDP")          -- expected: true
```

4. Template:

There is a separate template for each of the two subclasses.

```
/* TEMPLATE for the MtLoB class:
boolean find(String title) {
  ...  }  */

/* TEMPLATE for the ConsLoB class:
boolean find(String title) {
  ... this.fst ...
      ... this.fst.title ...
      ... this.fst.author ...
      ... this.fst.price ...
      ... this.fst.year ...
  ... this.rst.find(title) ... }  */
```

5. Program:

Again, we have two different methods, one for each class.

```
// PROGRAM for the class MtLoB:
boolean find(String title) {
  return false; }

// PROGRAM for the class ConsLoB:
boolean find(String title) {
  return (title.equals(this.fst.title) || this.rst.find(title)); }
```

6. Tests:

```
mtlob.find("Beach Music")  == false
lob1.find("Beach Music")   == false
lob2.find("Beach Music")   == true
lob3.find("HtDP")          == true
```

5.4 Example: Making a list of expensive books

1. Data Analysis and Problem Analysis

We need is the list of books and the title of the book to look for.

2. Purpose and Contract/Header:

```
// PURPOSE AND CONTRACT:
// produce a list of all books in this list that cost more than
// the given price
abstract boolean costMoreThan(int amount);
```

We develop the methods in the two classes concurrently, starting with examples.

3. Examples:

We use the same data as in the previous two problems.

```
mtlob.costMoreThan(15)  -- expected: mtlob
lob1.costMoreThan(15)   -- expected: lob1
lob2.costMoreThan(25)   -- expected: lob1
lob3.costMoreThan(18)   -- expected: lob2
lob2.costMoreThan(70)   -- expected: mtlob
```

4. Template:

There is a separate template for each of the two subclasses.

```
/* TEMPLATE for the MtLoB class:
boolean costMoreThan(int amount) {
  ...  }  */

/* TEMPLATE for the ConsLoB class:
boolean costMoreThan(int amount) {
   ... this.fst ...
      ... this.fst.title ...
      ... this.fst.author ...
      ... this.fst.price ...

          if (this.fst.price < amount) ...
          else ...

      ... this.fst.year ...
  ... this.rst.find(title) ... }  */
```

5. Program:

Again, we have two different methods, one for each class.

```
// PROGRAM for the class MtLoB:
ALoB costMoreThan(int amount){ return this; }

// PROGRAM for the class ConsLoB:
ALoB costMoreThan(int amount){
  if (this.fst.price < amount)
    return this.rst.costMoreThan(amount);
  else
    return new ConsLoB(this.fst, this.rst.costMoreThan(amount));
}
```

6. Tests:

We can no longer test two lists directly for equality. The Test Case tool understands how to compare two lists, even if they have been built at different times.

Note: This is a difficult topic, but students need to understand it to become good programmers. By constructing tests carefully from the beginning, students become aware of the difficulties and start asking the questions that expose the different kinds of equality.

5.5 Example: Computing the total size of all files in a list

1. Data Analysis and Problem Analysis

The only data we need is the list of files, because each file know its size.

2. Purpose and Contract/Header:

```
// PURPOSE AND CONTRACT:
// compute the total size of all files in this list
abstract int totalSize();
```

We develop the methods in the two classes concurrently, starting with examples.

3. Examples:

```
File f1 = new File("MyTrip", 1000);
File f2 = new File("hmwk1", 200);
File f3 = new File("hmwk3", 350);
ALoF mtlof = new MtLoF();
ALoF lof1  = new ConsLoF(f1, mtlof);
ALoF lof2  = new ConsLoF(f2, lof1);
ALoF lof3  = new ConsLoF(f3, lof2);
...
mtlof.totalSize()  -- expected: 0
lof1.totalSize()   -- expected: 1000
lof2.totalSize()   -- expected: 1200
lof3.totalSize()   -- expected: 1550
```

4. Template:

There is a separate template for each of the two subclasses.

```
/* TEMPLATE for the MtLoF class:
int totalSize() {
  ...  }
*/

/* TEMPLATE for the ConsLoF class:
int totalSize() {
  ... this.fst ...
      ... this.fst.name ...
      ... this.fst.size ...
  ... this.rst.totalSize() ...
}
*/
```

5. Program:

Again, we have two different methods, one for each class.

```
// PROGRAM for the class MtLoF:
int totalSize() {
  return 0; }

// PROGRAM for the class ConsLoF:
int totalSize() {
  return this.fst.size + this.rst.totalSize(); }
```

6. Tests:

```
mtlof.totalSize()  == 0
lof1.totalSize()   == 1000
lof2.totalSize()   == 1200
lof3.totalSize()   == 1550
```

5.6 Example: Is a `Posn` within a `AComboShape`

1. Data Analysis and Problem Analysis

The only data we need is this shape and the `Posn` in question.

2. Purpose and Contract/Header:

```
// PURPOSE AND CONTRACT:
// determine whether the given p is within this shape
abstract boolean within(Posn p);
```

We develop the methods in the three classes separately, starting from the class `Combo`, which is the simplest, then the class `Rect`, and finally the class `Circle`. For all examples we will use the following data:

```
AComboShape c1 = new Circle(new Posn(50, 20), 20);
AComboShape c2 = new Circle(new Posn(20, 20), 10);
AComboShape r1 = new Rect(new Posn(10, 20), 20, 40);
AComboShape r2 = new Rect(new Posn(40, 20), 50, 10);
AComboShape combo1 = new Combo(c1, r1);
AComboShape combo2 = new Combo(r2, combo1);
AComboShape combo3 = new Combo(combo2, c2);
```

Examples, Template, and Program for the class `Combo`:

3. Examples for the class Combo:

```
(combo1.within(new Posn(30, 30)) == true)
(combo3.within(new Posn(40, 30)) == true)
(combo3.within(new Posn(50, 50)) == false) // below the shape
(combo3.within(new Posn(40, 40)) == true)  // border of the shape
```

4. Template for the class Combo:

```
/* TEMPLATE for the method within in the class Combo:
// determine whether the given p is within this shape
boolean within(Posn p){ ... }
  ... this.top.within(p) ...
  ... this.bottom.within(p) ...
}
  */
```

5. Program for the class Combo:

We notice, that once we know whether a `Posn` is within either of the two shapes that comprise the `Combo` shape, the solution is easy.

```
  // PROGRAM for the class Combo:
  // determine whether the given p is within this shape
  boolean within(Posn p){
    return ( this.top.within(p) || this.bottom.within(p) );  }
```

Examples, Template, and Program for the class `Rect`:

3. Examples for the class Rect:

```
  (r1.within(new Posn(20, 20)) == true)
  (r2.within(new Posn(20, 20)) == false)    // left of the shape
  (r1.within(new Posn(50, 20)) == false)    // right of the shape
  (r2.within(new Posn(50, 10)) == false)    // above  the shape
  (r2.within(new Posn(60, 40)) == false)    // below the shape
```

4. Template for the class Rect:

```
/* TEMPLATE for the method within for the class Rect:
// determine whether the given p is within this shape
boolean within(Posn p){
  ... this.nw ...
      ... this.nw.x ...
      ... this.nw.y
  ... this.width ...
  ... this.height ...

  ... p.x ...
  ... p.y ...

  ... if ((p.x > this.nw.x) && (p.x < this.nw.x + this.width) ...
  ... if ((p.y > this.nw.y) && (p.y < this.nw.y + this.height) ...
}
*/
```

5. Program for the class Rect:

Writing the template we already realized that the relevant information is the relationship between the coordinates of the `Posn p` and the location and the size of the rectangle. The rest of the program follows easily.

```
  // PROGRAM for the class Rect:
  // determine whether the given p is within this shape
  boolean within(Posn p){
    return ((p.x >= this.nw.x) &&
            (p.x <= this.nw.x + this.width))
        && ((p.y >= this.nw.y) &&
            (p.y <= this.nw.y + this.height));  }
```

Examples, Template, and Program for the class `Circle`:

3. Examples for the class Circle:

```
  (c1.within(new Posn(40, 10)) == true)
  (c1.within(new Posn(20, 20)) == false)    // left of the shape
  (c1.within(new Posn(80, 20)) == false)    // right of the shape
  (c1.within(new Posn(20, 0))  == false)    // above  the shape
  (c1.within(new Posn(20, 40)) == false)    // below the shape
```

4. Template for the class Circle:

```
/* TEMPLATE for the method within for the class Circle:
// determine whether the given p is within this shape
boolean within(Posn p){
  ... this.center ...
      ... this.center.x ...
      ... this.center.y
  ... this.radius ...

  ... p.x ...
  ... p.y ...

  ... if ((p.x > this.center.x) && (p.x < this.center.x + this.radius) ...
  ... if ((p.y > this.center.y) && (p.y < this.center.y + this.radius) ...
}
*/
```

5. Program for the class Circle:

For the class `Circle` we compromise, and determine only whether the given `Posn p` is within the bounding square of our circle.

```
  // PROGRAM for the class Circle:
  // determine whether the given p is within this shape
  boolean within(Posn p){
    return ((p.x >= this.center.x - this.radius) &&
            (p.x <= this.center.x + this.radius))
        && ((p.y >= this.center.y - this.radius) &&
            (p.y <= this.center.y + this.radius));  }
```

6. Tests:

We had to wait with the examples until the method has been developed for all three classes. The alternative, is to provide each class with a *stub* of the method, that just returns an arbitrary value of the appropriate type, and then develop and test each method one step at a time. Of course, in that case, we

could not start with the method for the `Combo` class, because the test results for that method depend on the results of the methods for the classes `Circle` and `Rect`.

We now run as tests the examples introduced earlier.

5.7 Example: Sort the list of files by size

1. Data Analysis and Problem Analysis

The only data we need is the list of files, because each file know its size.

2. Purpose and Contract/Header:

```
////////////////////////////////////////////////////////////////
// PURPOSE AND CONTRACT:
// produce a list of all files in this list, sorted by size
ALoF sort() { ... }
```

We develop the methods in the two classes concurrently, starting with examples.

3. Examples:

```
File f1 = new File("MyTrip", 1000);
File f2 = new File("hmwk1", 200);
File f3 = new File("hmwk3", 350);
ALoF mtlof = new MtLoF();
ALoF lof1  = new ConsLoF(f1, mtlof);
ALoF lof2  = new ConsLoF(f2, lof1);
ALoF lof3  = new ConsLoF(f3, lof2);
...
mtlof.sort()  -- expected: mtlof
lof1.sort()   -- expected: lof1
lof2.sort()   -- expected: new ConsLoF(f2, new ConsLoF(f1 mtlof))
lof3.sort()   -- expected:
                 new ConsLoF(f2, new ConsLoF(f3, new ConsLoF(f1 mtlof)))
```

4. Template:

There is a separate template for each of the two subclasses.

```
/* TEMPLATE for the MtLoF class:
 ALoF sort() { ... }

*/

/* TEMPLATE for the ConsLoF class:
 ALoF sort() {
   ... this.fst ...
       ... this.fst.name ...
       ... this.fst.size ...
   ... this.rst.sort() ...
 }
 */
```

We observe, that, according to the contract, `this.rst.sort()` produces sorted rest of the list. Therefore, to complete the sort, all that is needed is to insert the `this.fst` item into the sorted rest of the list. Thus, we define a helper method:

2. Purpose and Contract/Header for the method `insert`:

```
/* PURPOSE AND CONTRACT:
// insert the given file into this sorted list
ALoF insert(File f) { ... }
```

and add it to the template for the `sort` method, so the template now is:

```
/* TEMPLATE for the ConsLoF class:
ALoF sort() {
  ... this.fst ...
      ... this.fst.name ...
      ... this.fst.size ...
  ... this.rst.sort() ...
  ... this.rst.insert(... a_file ...) ...
}
*/
```

5. Program:

Again, we have two different methods, one for each class.

```
// PROGRAM for the class MtLoF:
ALoF sort(){return this; }

// PROGRAM for the class ConsLoF:
ALoF sort(){
  return (this.rst.sort()).insert(this.fst);
}
```

We cannot run the tests until we implement and test the helper method `insert(File f)`.

3. Examples for the method `insert`:

```
mtlof.insert(f1)  -- expected: lof1
lof1.insert(f2)   -- expected: lof2
lof2.insert(f3)   -- expected:
   new ConsLoF(f2, new ConsLoF(f3, new ConsLoF(f1, mtlof)))
lof2.insert(new File("hmwk5", 1200))  -- expected:
   new ConsLoF(f2, new ConsLoF(f1, new ConsLoF(new File("hmwk5", 1200), mtlof)))
```

4. Template for the method `insert`:

There is a separate template for each of the two subclasses.

```
/* TEMPLATE for the MtLoF class:
 ALoF insert(File f) { ... }

*/

/* TEMPLATE for the ConsLoF class:
 ALoF insert(File f) {
    ... this.fst ...
       ... this.fst.name ...
       ... this.fst.size ...
   ... this.rst.sort() ...
   ... this.rst.insert(f) ...
       ... f.name ...
       ... f.size ...

   ... if (f.size < this.fst.size) ...
       else ...
}
*/
```

5. Program:

Again, we have two different methods, one for each class.

```
// PROGRAM for the class MtLoF:
ALoF insert(File f) { return new ConsLoF(f, this); }

// PROGRAM for the class ConsLoF:
ALoF insert(File f) {
  if (f.size < this.fst.size)
    return new ConsLoF(f, this);
  else
    return new ConsLoF(this.fst, this.rst.insert(f));
}
```

6. Tests:

We first test the examples given for the `insert` method, then the examples developed for the `sort` method. We need to use the Test Case tool to compare the resulting lists.

5.8 Example: Modify class definitions for shapes, to have a common location field

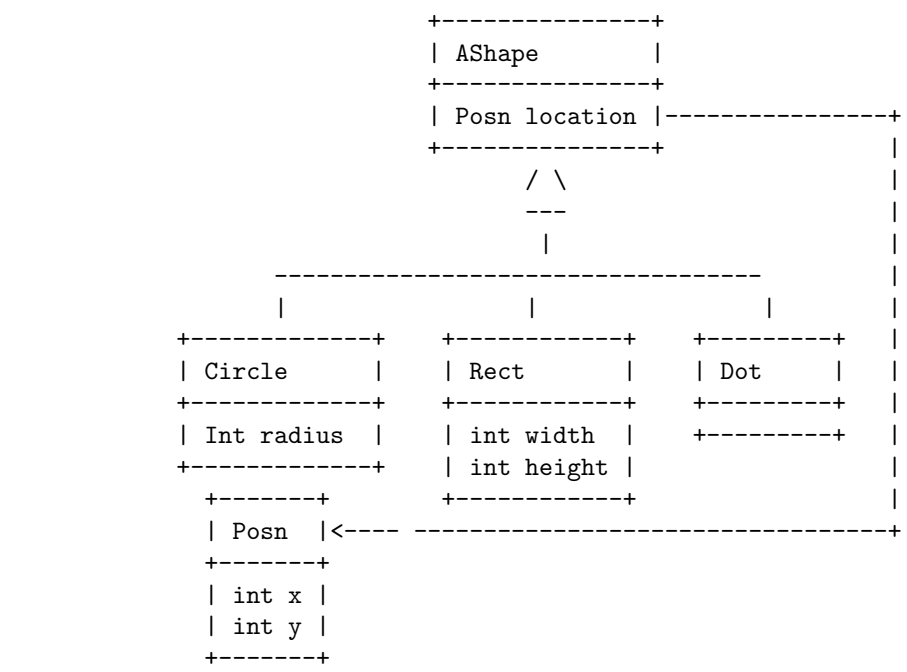
Earlier, we had the following data definition:

A Shape is on of:

- a Circle: consisting of a center and a radius, where the center is a Posn and the radius is an integer
- a Rectangle: consisting of a nw, a width, and a height, where the nw is a Posn, and the width and the height are integers
- a Dot: consisting of a location, where the location is a Posn

When computing the distance of a shape to the origin, we noticed that each of the three variants contained a field that in some manner represented the location of this shape. We may decide to give these fields the same name (`location` instead of `center` or `nw`). Furthermore, because this field is common to all shapes, and we expect any further variants of the shapes to have some distinct point that specifies its location, we can *lift* this field to be a field in the `abstract` class. The class diagram and the class definitions will then be:

```
/*
```




```
*/
```

```
// to represent one point on  a canvas
class Posn  ... we omit the rest of this definition ...

// to represent geometric shapes
abstract class AShape {
  Posn location;
}

// to represent a circle
class Circle extends AShape {
  int radius;

  Circle(Posn center, int radius) {
    this.location = location;
    this.radius = radius; }
}

// to represent a rectangle
class Rect extends AShape {
  int width;
  int height;

  Rect(Posn nw, int width, int height) {
    this.location = location;
    this.width = width;
    this.height = height; }
}

// to represent a dot
class Dot extends AShape {
  Posn location;

  Dot(Posn location) {
    this.location = location; }
}
```

We now want to compute the distance to origin for each of these shapes.

1. Data Analysis and Problem Analysis

We only need a purpose statement for the `abstract` method.

2. Purpose and Contract/Header:

```
  // compute the distance of this Shape to the origin
   abstract int distTo0();
```

We develop each method separately, but show them here concurrently. We only show the templates and the programs, as all the examples and tests are the same as before.

4. Template:

There is a separate template for each of the three subclasses.

```
  /* TEMPLATE for the Circle class:
  int distTo0() {
    ... this.location ...
        ... this.location.x ...
        ... this.location.y ...
    ... this.radius ... }  */

 /* TEMPLATE for the Rect class:
  int distTo0() {
    ... this.location ...
        ... this.location.x ...
        ... this.location.y ...
    ... this.width ...
    ... this.height ... }  */

 /* TEMPLATE for the Dot class:
  int distTo0() {
    ... this.location ...
        ... this.location.x ...
        ... this.location.y ... }  */
```

5. Program:

Again, we have three different methods, one for each class.

```
  // PROGRAM for the class Circle:
  int distTo0() {
    return this.location.distTo0(); }

  // PROGRAM for the class Rect:
  int distTo0() {
    return this.location.distTo0(); }

  // PROGRAM for the class Dot:
  int distTo0() {
    return this.location.distTo0(); }
```

Observing that the bodies of all three methods are the same, and do not rely on any information specific to these classes, we can *lift* the method to become

a concrete method in the abstract class. So, we end up with the abstract class defined as follows - with no methods in the concrete classes:

```
// to represent geometric shapes
abstract class AShape {
  Posn location;

  // compute the distance of this Shape to the origin
  int distTo0() {
    return this.location.distTo0(); }
}
```

6 Wednesday Afternoon:

6.1 Goals

Continue designing methods for self-referential and mutually referential data.

Introduce a simple drawing teachpack.

6.2 Simple Drawing

In order to draw the shapes like the circles, rectangles, and the combo shapes we have seen, we must have some place where the drawings will become visible. Doing so requires that we use a tool (teachpack) that provides the drawing surface and allows us to control its contents. We will first learn to draw basic shapes. We will then see how the drawing can change over the time (creating an animation), or in response to some events, such as a key press. These three features: the drawing canvas, the change over time, and the response to keyboard, mouse, or other events is at the core of many computer applications today.

To be able to see the drawing there must be a world (a stage, a canvas) where the drawing becomes visible. ProfessorJ provides such world through a library. To import the code form the library, start the code with the follwing import statement:

```
import draw.*;
```

This is the way we can use classes that have been defined elsewhere.

The colors

The library contains several classes such as `Red, Black, Blue, Yellow, Green, White` that represent colors for our drawings. So, if we want the circle to be shown in red, we must specify the red color as

```
new Red()
```

The `World`

The class `World` is the key class in this library. It provides a canvas into which we will draw, and methods that allow us to define what our program should do in response to various events, such as the user hitting a key or the timer advancing another tick ahead.

The class `World` contains the following useful methods for **drawing and erasing basic colored shapes**:

```
// draw a solid disk
boolean drawDisk(Posn center, int radius, Color c);

// draw a solid rectangle
boolean drawRect(Posn nw, int width, int height, Color c);
```

```
// draw a circle
boolean drawCircle(Posn center, int radius, Color c);

// draw a line
boolean drawLine(Posn from, Posn to, Color c);

// clear a solid disk
boolean clearDisk(Posn center, int radius, Color c);

// clear a solid rectangle
boolean clearRect(Posn nw, int width, int height, Color c);

// clear a circle
boolean clearCircle(Posn center, int radius, Color c);

// clear a line
boolean clearLine(Posn from, Posn to, Color c);
```

Before we start drawing, we create an instance of the `World`. We can then **show and destroy the canvas** that holds our drwaing by invoking the following methods:

```
// create a drawing canvas of the given size and show it
boolean start(int width, int height);

// destroy the drawing canvas
boolean stop();
```

Drawing shapes

To make it possible to draw shapes, we add *convenience* methods that will draw the shape instance in the given world to our classes that represent the geometry of shapes. For example, we may add the following methods to the class Circle:

```
  // draw this circle in the given World
  boolean draw(World w) {
    return w.drawDisk(this.center, this.radius, new Red());
  }

  // clear this circle in the given World
  boolean clear(World w) {
    return w.clearDisk(this.center, this.radius, new Red());
  }
```

A sample drawing interaction
The tests for these methods would be given in the Interactions Box as follows:

```
// make an example of a drawing world
World dw = new World();

// show the world as a canvas of the size 200, 400
dw.start(200, 400)

// make a big circle
Circle bigc = new Circle(new Posn(100, 200), 50);

// make a small circle
Circle smallc = new Circle(new Posn(100, 200), 25);

// draw the big circle in the given world
bigc.draw(dw)

// clear the small circle in the given world
smallc.clear(dw)
```

These methods return a `boolean` value `true` when the drawing succeeded. However, the only real test of the correctness of the desired effect is a visual inspection.

Alternately, we may change the definition of the class `Circle`, so that each instance of the class `Circle` contains a field that represents its color.

When creating and moving drawings, we can (and will) still test, whether the new *moved* object has been constructed as expected.

7 Thursday Morning:

7.1 Goals

Extended Lab Session: Design of an interactive game that responds to key events and the timer.

7.2 Event-Driven World

Next we learn to write programs that respond to user-initiated events, such as keystrokes. Our goal is to move the circle around the canvas in response to the four arrow keys: *up, down, left, right.* At each such keystroke we move the circle a fixed distance in the indicated direction.

To accomplish this, we need to add to the class `Circle` the method

```
// produce a circle moved 20 pixels in the direction given by the ke
// from this circle
Circle move(String ke){ ...
```

Exercise

Follow the design recipe to develop this method. Make sure you test it properly. The test (following the design recipe) determines whether the new circle is at the desired location. In addition, you can also include a visual verification of the correctness via the drawing effect, by drawing both, the original circle, and the moved circle.

Key events

We define a new class `KeyEventWorld` that extends `World`. This class has two fields, one that represents the circle to be drawn, and one that represents the bounding `Box` for our canvas:

```
// represent the world of a Circle, and the Bounding Box
class KeyEventWorld extends World {

   Circle c;
   Box  b;
...
```

This means, that an instance of the `World` knows about the `Circle` that will be drawn in its canvas, and the `Circle` knows about the `World` where to draw itself. Both also know the current size of the canvas that is represented by the bounding `Box`.

We first add the methods that will draw and clear the circle in this world:

```
// draw this world
boolean draw() {
  return this.c.draw(this);
}

// clear this world
boolean clear() {
  return this.c.clear(this);
}
```

Of course, the contents of the `World` for a game will be much more complex than just one circle. There may be a worm and the food for the worm to find, and a possible score box, indicating the current length of the worm, etc.)

Representing one move of the objects in the `World`

Next, we need to add to our `KeyWorld` class the method that will invoke the `move` method in the `Circle` class in response to each keystroke, with the `String` argument that represents that keystroke. The method is defined as follows:

```
// what happens when the player presses a key
World onKeyEvent(String ke) {

  // move the circle as given by the ke
  // make new world using the new circle
  // clear the old world and draw the new one
  // return the new world
  return this.oneStep(this.move(this.c.move(ke)));
}
```

The Body of this method invokes three different methods in two different classes. The inner-most argument is:

```
this.c.move(ke)
```

We already know that this method returns a circle that has been moved in the direction given by the `String ke`.

The circle that was produced by `this.c.move(ke)` becomes the argument for the method `move` in the class `KeyEventWorld` that is defined as follows:

```
// move the circle and return the new moved world
KeyEventWorld move(Circle newCircle) {

  // check whether the center of the new circle
  // is inside the canvas: if yes, return a new world
  if (this.b.inside(newCircle.center))
     return new KeyEventWorld(newCircle, this.b);
```

```
    // new circle is outside the bounds - the world ends
    else if (this.endOfTime())
       return this;

    // this will not happen, but else clause must be given
    else
       return this;
  }
```

The above method is the key to our *game*. We continue to move the circle, as long as it remains within the bounds of the canvas. Finally, the `oneStep` method clears the current world and replaces it with the new world, produced by the method `move` thus creating an animation:

```
  // clear this world and draw that one
  // return that if things go well, otherwise this
  KeyEventWorld oneStep(KeyEventWorld that) {
    if (this.clear() && that.draw())
       return that;
    else
       return this;
  }
```

The last thing that remains to be done is to construct an instance of the `KeyEventWorld` and start the world's timer. At the end of the class definitions, we add the following Interactions Box:

```
// construct an instance of a KeyEventWorld
KeyEventWorld w = new KeyEventWorld(
                              new Circle(new Posn(100, 200), 20),
                              new Box(200, 400));

// start the world, start the timer, draw the world
w.start(200,400) && w.bigBang(0.3) && w.draw()
```

Time-Driven World

There is only a small addition needed to allow the program to perform specific tasks at each tick of the clock. The clock speed is determined when the program starts, and then at each tick, the method designed to respond to the timer events is invoked. First, we add the method `onTick()` to the class `TimerWorld`, which is otherwise the same as our earlier `KeyEventWorld`:

```
  // what happens when the clock ticks
  World onTick() {
    return this.oneStep(this.move(this.c.moveRandom(20)));
  }
```

We decide to move the circle in a random direction on each tick. This behavior is defined in the method `moveRandom` in the class `Circle`, together with a helper method `int randomInt(int n)` as follows:

```
// produce a new circle moved by a random distance
Circle moveRandom(int n){
  return new Circle(new Posn(this.center.x + this.randomInt(n),
                             this.center.y + this.randomInt(n) ),
                    this.radius);
}

// helper method to generate a random number in the range -n to n
int randomInt(int n){
  return -n + (new Random().nextInt()) % (2 * n);
}
```

8 Thursday Afternoon:

8.1 Goals

Design recipe for abstractions:

1. abstracting over data type of an item in a collection: `Object`
2. abstracting over the behavior: interfaces

8.2 Example: A list of `Objects`

The following class hierarchy can represent an arbitrary list of

```
// to represent a list of arbitrary objects
abstract class ALoObj{

  // count the number of items in this list
  abstract int count();
}

// to represent an empty list of arbitrary objects
class MTLoObj extends ALoObj{

  MTLoObj(){ }

  int count(){ return 0; }
}

// to represent an empty list of arbitrary objects
class ConsLoObj extends ALoObj{
  Object fst;
  ALoObj rst;

  ConsLoObj(Object fst, ALoObj rst){
    this.fst = fst;
    this.rst = rst;
  }

  int count() {
    return 1 + this.rst.count();
  }
}
```

Make examples of instances of `Books` and lists of `Books` that are defined as lists of `Objects`.

Interface `IFilter`:

```
interface IFilter{
  // method that selects objects with some property
  boolean select();
}
```

Classes that implement the interface `IFilter`:

```
// class to represent a book
class Book implements IFilter{
  String title;
  Author author;
  int price;

  Book(String title, Author author, int price) {
    this.title = title;
    this.author = author;
    this.price = price;  }

  boolean equals(Object obj){
    return    (this.title.equals( ((Book) obj).title))
           && (this.author.equals( ((Book) obj).author))
           && (this.price == ((Book) obj).price); }

  boolean select(){
    return this.price < 10; }
}

// class to represent the author of a book
class Author implements IFilter{
  String name;
  int year;

  Author(String name, int year){
    this.name = name;
    this.year = year; }

  boolean equals(Object obj) {
    return    (this.name.equals(  ((Author) obj).name ))
           && (this.year == (  ((Author) obj).year ));
  }

  boolean select(){
    return this.year > 1940;
  }
```

```
  // determine whetheri this is a contemporary author
  boolean isContemporary(){
    return this.year > 1940;
  }
}
```

The interface `IFilter2`:

```
interface IFilter2{
  // method that selects objects with some property
  boolean select2(Object obj);
}
```

Classes that implement the interface `IFilter2`:

```
class CheapBook implements IFilter2{
  // determine whether the given book is cheap
  boolean select2(Object obj){
    return ((Book)obj).price < 10;
  }
}

class NewAuthor implements IFilter2{
  // determine whether the given book was written by contemporary author
  boolean select2(Object obj){
    return ( ((Book)obj).author.isContemporary() );
  }
}
```

Classes to represent a list of `Objects`

```
// to represent a list of arbitrary objects
abstract class ALoObj{

  // count the number of items in this list
  abstract int count();

  // determine whether this list contains the given object
  abstract boolean contains(Object obj);

  // remove the first occurrence of the given object from this list
  abstract ALoObj remove(Object obj);

  // determine whether there is a 'selected' object in this list
  abstract boolean orMap();
```

```
  // determine whether there is a 'selected' object in this list
  abstract boolean orMap2(IFilter2 filter);
}

//-----------------------------------------------------------------
// to represent an empty list of arbitrary objects
class MTLoObj extends ALoObj{

  MTLoObj(){ }

  int count(){ return 0; }

  boolean contains(Object obj){ return false; }

  ALoObj remove(Object obj){ return this; }

  boolean orMap(){ return false; }

  boolean orMap2(IFilter2 filter){ return false; }
}

//-----------------------------------------------------------------
// to represent an empty list of arbitrary objects
class ConsLoObj extends ALoObj{
  Object fst;
  ALoObj rst;

  ConsLoObj(Object fst, ALoObj rst){
    this.fst = fst;
    this.rst = rst;
  }

/*
Template:
...this.fst...
...this.rst...
...this.count()...
...this.contains(Object obj)...
...this.remove(Object obj)...
...this.orMap()...
*/

  int count() {
    return 1 + this.rst.count();
  }
```

```
  boolean contains(Object obj){
    return    (this.fst.equals(obj))
           || (this.rst.contains(obj));
  }

  ALoObj remove(Object obj){
    if (this.fst.equals(obj))
      return this.rst;
    else
      return new ConsLoObj(this.fst, this.rst.remove(obj));
  }

  boolean orMap(){
    return    ( ((IFilter)(this.fst)).select() )
           || (this.rst.orMap());
  }

  boolean orMap2(IFilter2 filter){
    return    filter.select2(this.fst)
           || (this.rst.orMap2(filter));
  }
}
```

Examples:

```
Author mf = new Author("Matthias", 1958);
Author bard = new Author("Shakespeare", 1716);

Book b1 = new Book ("HtDP", mf, 60);
Book b2 = new Book ("LL", mf, 18);
Book b3 = new Book ("Hamlet", bard, 7);

ALoObj mtlist = new MTLoObj();
ALoObj list1 = new ConsLoObj(b1, new ConsLoObj(b2, mtlist));
ALoObj list2 = new ConsLoObj(b3, list1);
```

Testing the `count()` method in `class ListofObj`

```
mtlist.count() == 0
list1.count() == 2
list2.count() == 3
```

Testing the `contains()` method in `class ListofObj`

```
mtlist.contains(b1) == false
```

```
list1.contains(b3) == false
list2.contains(b3) == true
list2.contains(b2) == true
```

Testing the `remove()` method in `class ListofObj`

```
mtlist.remove(b1).equals(mtlist)
list1.remove(b3).count()
list2.remove(b3).count()
```

Testing the `select()` method in `class Book`

```
b1.select() == false
b2.select() == true
```

Testing the `select()` method in `class Author`

```
mf.select() == true
bard.select() == false
```

Testing the `orMap()` method on lists of books

```
mtlist.orMap() == false
list1.orMap() == false
list2.orMap() == true
```

Testing the `orMap2()` method on lists of books

```
CheapBook cheapFilter = new CheapBook();

mtlist.orMap2(cheapFilter) == false
list1.orMap2(cheapFilter) == false
list2.orMap2(cheapFilter) == true

NewAuthor authorFilter = new NewAuthor();

mtlist.orMap2(authorFilter) == false
list1.orMap2(authorFilter) == true
list2.orMap2(authorFilter) == true
```

9 Friday Morning:

9.1 Goals

A big picture view of how this curriculum represents the key ideas of computing.

Topics not covered - a preview of how the curriculum continues to include programs with state, mutation, and traversals, and transitions to the full Java language:

- Function objects: Comparator, Selector, IObj2Boolean, etc.
- State - programming with effects
- Mutating structures
- Abstracting traversals: iterators

9.2 Example: ...