Designing Programs with Class

Sam Tobin-Hochstadt and David Van Horn

April 8, 2013

This book introduces the fundamental elements of class-based program design.
The book is also available in PDF form here.

Contents

Preface

This book is a **draft** textbook to accompany Fundamentals of Computer Science II (Honors): Introduction to Class-Based Program Design at Northeastern University.[1] It began its life as a series of course notes from the Spring 2011 incarnation of the class, evolved during Spring 2012, and is now being revised throughout the Spring 2013 semester.

The "book" is very much a work in progress, which means there are large omissions and numerous errors. Moreover, it's certain to change as the course progresses. Your patience, feedback, and bug reports are greatly appreciated. The book and its accompanying software are maintained on Github at the following URL:

`https://github.com/dvanhorn/dpc`

---

[1]`http://www.ccs.neu.edu/course/cs2510h/`

Design Recipes

It puts a lot of emphasis on something called The Design Recipe, which sounded hokey to me at first, but when I was shown what The Design Recipe was, I realized, even as someone who's been programming for 35 years, that rather than code a solution to something complex in say, an hour, after 15-20 minutes, when I thought I was a third done, I was actually and suddenly completely done, sometimes almost feeling like magic. It basically puts a high emphasis on thinking and designing first, writing test cases very early, picking the right program structure, and then filling in the blanks and expanding them in a few places. Following the Design Recipe, not only was I often "suddenly done," but because my tests were written before I started coding the meat of a solution, I had (and continue to have) high confidence that my program works and will continue to work (or alert me quickly to new errors) should I make any "enhancements" down the road.

— Geoffrey S. Knauth, 2011

The main focus of this book is the *design process* that leads from problem statements to well-organized solutions, oriented around the concept of *objects*. We make extensive use of explicit design guidelines formulated as a number *program design recipes* as described in *How to Design Programs*. The most general form of the recipe for designing programs is given in figure 1.

1. Problem Analysis & Data Definition
2. Contract, Purpose & Effect Statements, Header
3. Examples
4. Template
5. Code
6. Tests

Figure 1: The Design Recipe for Programs

The objective of the design recipe is to provide a *technique* that guides programmers—

novices and professionals alike—with a systematic approach to problem solving. By reasoning systematically, programmers can eliminate the incidental complexities of writing software and instead focus their creative energy on what is essentially difficult about a particular problem. A great programmer is first a master technician; they continue to hone and internalize their skills with each day of programming. As the chef Jacques Pépin said, [2] speaking of another field that values the idea of recipes, to master the technique, you have no choice: "you have to repeat, repeat, repeat, repeat until it becomes part of yourself."

---

[2]*New York Times*, "There's the Wrong Way and Jacques Pepin's Way" October 18, 2011.

The Choice of Language and Environment

> *Another lesson we should have learned from the recent past is that the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages.*
>
> — Edsger W. Dijkstra "The Humble Programmer", Turing Award Lecture, 1972

In support of this book, we have a developed a series of modest programming languages that emphasize the principles of object-oriented design. We use these languages throughout the first part of the book. They are deliberately not industrial strength programming languages, complete with full-featured libraries, convenience mechanisms, and the usual idiosyncrasies that accompany "real" languages. Instead we have designed a progression of simple and consistent languages which embody the common core of modern object-oriented languages. The idea is that by focusing on the conceptual basis of object-oriented programming, students can apply their design knowledge regardless of whatever linguistic context they happen to find themselves in down the road.

In the second part, we transition to Java, a widely used industrial language that has been developed over more than 15 years. The move to Java allows us to explore the application of the principles introduced in the first portion of the course in the context of a practical language.

As of this draft of the book, the course software has been developed and tested with version 5.2 of Racket. To install the course software, you will first need to install Racket:

```
http://racket-lang.org/
```

Once installed, launch DrRacket, the development environment that ships with the Racket system. Click on the File|Install PLT File, and then enter the URL:

```
http://www.ccs.neu.edu/course/cs2510h/class-system-latest.plt
```

After the PLT file has been installed, select Language|Choose Language... and select the "Use the language declared in the source" option. You can now write programs in any of the languages included in the course software by writing #`lang class/`N as the first line of a file, where N is a number in 0, 1, 2, etc. To find out more about the languages, use Help Desk and search for `class/0` to get started.

The Parts of the Book

Acknowledgments

Part I

Basic Design with Objects

Chapter 1

Objects = Data + Function

One of the key concepts behind so-called *object-oriented programming* (OOP) is the notion of an *object*. An object is a new kind of value that can, as a first cut, be understood as a pairing together of two familiar concepts: data and function.

- **An object is like a structure** in that it has a fixed number of fields, thus an object (again, like a structure) can represent compound data. But unlike a structure, an object contains not just data, but *functionality* too;

- **An object is like a (set of) function(s)** in that it has behavior—it *computes*; it is not just inert data.

This suggests that objects are a natural fit for well-designed programs since good programs are organized around data definitions and functions that operate over such data. An object, in essence, packages these two things together into a single programming apparatus. This has two important consequences:

1. **You already know how to design programs oriented around objects.**

   Since objects are just the combination of two familiar concepts that you already use to design programs, you already know how to design programs around objects, even if you never heard the term "object" before. In short, the better you are at programming with functions, the better you will be at programming with objects.

2. **Objects enable new kinds of abstraction and composition.**

   Although the combination of data and function may seem simple, objects enable new forms of abstraction and composition. That is, objects open up new approaches to the construction of computations. By studying these new approaches, we can distill new design principles. Because we understand objects are just the combination of data and function, we can understand how all of these principles apply in the familiar context of programming with functions. In short, the better you are at programming with objects, the better you will be at programming with functions.

In this chapter, we will explore the basic concepts of objects by revisiting a familiar program, first organized around data and functions and then again organized around objects.

1.1   Functional rocket

In this section, let's develop a simple program that animates the lift-off of a rocket.

The animation will be carried out by using the `big-bang` system of the `2htdp/universe` library. For an animation, `big-bang` requires settling on a representation of *world states* and two functions: one that renders a world state as an image, and one that consumes a world state and produce the subsequent world state.

Generically speaking, to make an animation we must design a program of the form:

```
(big-bang <world0>           ; World
          (on-tick <tick>)   ; World -> World
          (to-draw <draw>))  ; World -> Scene
```

where `World` is a data definition for world states, `<tick>` is an expression whose value is a `World -> World` function that computes successive worlds and `<draw>` is an expression whose value is a `World -> Scene` function that renders a world state as an image.

For the purposes of a simple animation, the world state can consist of just the rocket:

```
;; A World is a Rocket.
```

The only relevant piece of information that we need to keep track of to represent a rocket lifting off is its height. That leads us to using a single number to represent rockets. Since rockets only go up in our simple model, we can use non-negative numbers. We'll interpret a non-negative number as meaning the distance between the ground and the (base of the) rocket measured in *astronomical units* (AU):

```
;; A Rocket is a non-negative Number.
;; Interp: distance from the ground to base of rocket in AU.
```

This dictates that we need to develop two functions that consume `Rocket`s:

```
;; next : Rocket -> Rocket
;; Compute next position of the rocket after one tick of time.

;; render : Rocket -> Scene
;; Render the rocket as a scene.
```

Let's take them each in turn.

1.1.1 The `next` function

For `next`, in order to compute the next position of a rocket we need to settle on the amount of elapsed time a call to `next` embodies and how fast the rocket rises per unit of time. For both, we define constants:

```
(define CLOCK-SPEED  1/30) ; SEC/TICK
(define ROCKET-SPEED 1)    ; AU/SEC
```

The `CLOCK-SPEED` is the rate at which the clock ticks, given in seconds per tick, and `ROCKET-SPEED` is the rate at which the rocket lifts off, given in AU per second. We use these two constants to define a third, computed, constant that gives change in the rocket's distance from the ground per clock tick:

```
(define DELTA (* CLOCK-SPEED ROCKET-SPEED)) ; AU/TICK
```

We can now give examples of how `next` should work. We are careful to write test-cases in terms of the defined constants so that if we revise them later our tests will still be correct:

```
(check-expect (next 10) (+ 10 DELTA))
```

Now that we have develop a purpose statement, contract, and example, we can write the code, which is made clear from the example:

```
;; next : Rocket -> Rocket
;; Compute next position of the rocket after one tick of time.
(check-expect (next 10) (+ 10 DELTA))
(define (next r)
  (+ r DELTA))
```

1.1.2 The `render` function

The purpose of `render` is visualize a rocket a scene. Remember that rockets are represented by the distance between the ground and their base, so a rocket at height 0 should sitting at the bottom of a scene. We want it to look something like:

```
> (render 0)
```
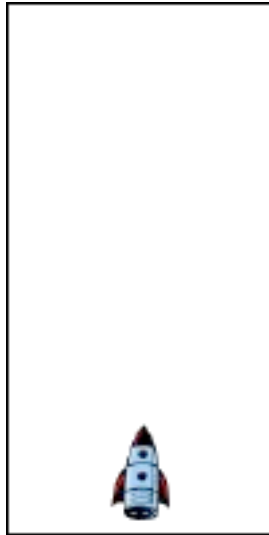
To do so we need to settle on the size of the sceen and the look of the rocket. Again, we define constants for this. We use the 2htdp/image library for constructing images.

```
(define ROCKET        )  ; Use rocket key to insert the rocket here.
(define WIDTH 100)    ; PX
(define HEIGHT 200)   ; PX
(define MT-SCENE (empty-scene WIDTH HEIGHT))
```

You can copy and paste the rocket image from this program, or you can access the image as follows:

```
> (bitmap class/0/rocket.png)
```

Since we may want to draw rockets on scenes other than the MT-SCENE, let's develop a helper function:

```
;; draw-on : Rocket Scene -> Scene
;; Draw rocket on to scene.
(define (draw-on r scn) ...)
```

allowing us to define render simpy as:

```
;; render : Rocket -> Scene
```

```
;; Render the rocket as a scene.
(define (render r)
  (draw-on r MT-SCENE))
```

Recall that a rocket is represented by the distance from the ground to its **base**. On the other hand, the 2htdp/image library works in terms of *pixels* (PX) and *graphics coordinates*. We need draw-on to establish the mapping between AU. For simplicity, we assume 1 PX equals 1 AU. Using overlay/align/offset, the draw-on functions places the rocket on the scene on the center, bottom of the scene, offset vertically by the height of the rocket:

```
;; draw-on : Rocket Scene -> Scene
;; Draw rocket on to scene.
(define (draw-on r scn)
  (overlay/align/offset "center" "bottom"
                        ROCKET
                        0 (add1 r)
                        scn))
```

1.1.3   Lift off

With these functions in place, let's launch a rocket:

```
;; Lift off!
(big-bang 0
          (tick-rate CLOCK-SPEED)
          (on-tick next)
          (to-draw render))
```

Our complete BSL program is:

```
(require 2htdp/image)
(require 2htdp/universe)

; A World is a Rocket.

; A Rocket is a non-negative Number.
; Interp: distance from the ground to base of rocket in AU.

(define CLOCK-SPEED  1/30) ; SEC/TICK
(define ROCKET-SPEED 1)    ; AU/SEC
(define DELTA (* CLOCK-SPEED ROCKET-SPEED)) ; AU/TICK
```



```
(define ROCKET       ) ; Use rocket key to insert the rocket here.
```

```
(define WIDTH 100)   ; PX
(define HEIGHT 200)  ; PX
(define MT-SCENE (empty-scene WIDTH HEIGHT))

; next : Rocket -> Rocket
; Compute next position of the rocket after one tick of time.
(check-expect (next 10) (+ 10 DELTA))
(define (next r)
  (+ r DELTA))

; render : Rocket -> Scene
; Render the rocket as a scene.
(define (render r)
  (draw-on r MT-SCENE))

; draw-on : Rocket Scene -> Scene
; Draw rocket on to scene.
(check-expect (draw-on 0 (empty-scene 100 100))
              (overlay/align/offset "center" "bottom"
                                    ROCKET
                                    0 1
                                    (empty-scene 100 100)))
(define (draw-on r scn)
  (overlay/align/offset "center" "bottom"
                        ROCKET
                        0 (add1 r)
                        scn))

; Lift off!
(big-bang 0
          (tick-rate CLOCK-SPEED)
          (on-tick next)
          (to-draw render))
```

1.2   Object-oriented rocket

Now let's redevelop this program only instead of using data and functions, we'll use objects.

   You'll notice that there are two significant components to the rocket program. There is the *data*, which in this case is a number representing the distance the rocket has traveled, and the *functions* that operate over that class of data, in this case next and render.

   This should be old-hat programming by now. But in this book, we are going to explore a new programming paradigm that is based on *objects*. As a first approximation, you can think of an *object* as the coupling together of the two significant components

of our program (data and functions) into a single entity: an object.

Since we are learning a new programming language, you will no longer be using BSL and friends. Instead, select Language|Choose Language... in DrRacket, then select the "Use the language declared in the source" option and add the following to the top of your program:

```
#lang class/0
```

The constants of the rocket program remain the same, so our new program still includes a set of constant definitions:

```
(define CLOCK-SPEED  1/30) ; SEC/TICK
(define ROCKET-SPEED 1)    ; AU/SEC
(define DELTA (* CLOCK-SPEED ROCKET-SPEED)) ; AU/TICK
```

```
(define ROCKET        ) ; Use rocket key to insert the rocket here.
(define WIDTH 100)   ; PX
(define HEIGHT 200)  ; PX
(define MT-SCENE (empty-scene WIDTH HEIGHT))
```

A set of objects is defined by a *class*, which determines the number and name of fields and the name and meaning of each behavior that every object is the set contains. By analogy, while an object is like a structure, a class definition is like a structure definition.

1.2.1 A class of rockets

The way to define a class is with `define-class`:

```
(define-class rocket%
  (fields dist))
```

This declares a new class of values, namely `rocket%` objects. (By convention, we will use the `%` suffix for the name of classes.) For the moment, `rocket%` objects consist only of data: they have one *field*, the `dist` between the rocket and the ground.

Like a structure definition, this class definition defines a new kind of data, but it does not make any particular instance of that data. To make a new instance of a particular class, i.e. an object, you use the `new` syntax, which takes a class name and expressions that produce a value for each field of the new object. Since a `rocket%` has one field, `new` takes the shape:

```
> (new rocket% 7)
(new rocket% 7)
```

This creates a `rocket%` representing a rocket with height 7.

In order to access the data, we can invoke the `dist` accessor method. Methods are like functions for objects and they are called by using the `send` form like so:

```
> (send (new rocket% 7) dist)
7
```

This suggests that we can now re-write the data definition for `Rockets`:

```
;; A Rocket is a (new rocket% NonNegativeNumber)
;; Interp: distance from the ground to base of rocket in AU.
```

1.2.2   The `next` and `render` methods

To add functionality to our class, we define *methods* using the `define` form. In this case, we want to add two methods `next` and `render`:

```
;; A Rocket is a (new rocket% NonNegativeNumber)
;; Interp: distance from the ground to base of rocket in AU.
(define-class rocket%
  (fields dist)

  ;; next : ...
  (define (next ...) ...)

  ;; render : ...
  (define (render ...) ...))
```

We will return to the contracts and code, but now that we've seen how to define methods, let's look at how to *apply* them in order to actually compute something. To call a defined method, we again use the `send` form, which takes an object, a method name, and any arguments to the method:

```
(send (new rocket% 7) next ...)
```

This will call the `next` method of the object created with `(new rocket% 7)`. This is analogous to applying the `next` function to 7 in the section 2.1 section. The elided code (`...`) is where we would write additional inputs to the method, but it's not clear what further inputs are needed, so now let's turn to the contract and method headers for `next` and `render`.

When we designed the functional analogues of these methods, the functions took as input the rocket on which they operated, i.e. they had headers like:

```
;; next : Rocket -> Rocket
;; Compute next position of the rocket after one tick of time.

;; render : Rocket -> Scene
;; Render the rocket as a scene.
```

But in an object, the data and functions are packaged together. Consequently, the method does not need to take the world input; that data is already a part of the object

and the values of the fields are accessible using accessors. In other words, methods have an implicit input that does not show up in their header—it is the object that has called the method. That value, since it is not available as an explicit parameter of the method, is made available through the `this` variable. We likewise revise the purpose statements to reflect the fact "the rocket" is the object calling the method, so we instead write "this rocket", emphasizing that `this` refers to a rocket.

That leads us to the following method headers:

`rocket%`

```
;; next : -> Rocket
;; Compute next position of this rocket after one tick of time.
(define (next) ...)

;; render : -> Scene
;; Render this rocket as a scene.
(define (render) ...)
```

The `rocket%` box is our way of saying that this code should live in the `rocket%` class.

Since we now have contracts and have seen how to invoke methods, we can now formulate test cases:

`rocket%`

```
;; next : -> Rocket
;; Compute next position of this rocket after one tick of time.
(check-expect (send (new rocket% 10) next)
              (new rocket% (+ 10 DELTA)))
(define (next) ...)

;; render : -> Scene
(check-expect (send (new rocket% 0) render)
              (overlay/align/offset "center" "bottom"
                                    ROCKET
                                    0 1
                                    MT-SCENE))
(define (render) ...)
```

Finally, we can write the code from our methods:

`rocket%`

```
(define (next)
  (new rocket% (+ (send this dist) DELTA)))

(define (render)
  (send this draw-on MT-SCENE))
```

Just as in the functional design, we choose to defer to a helper to draw a rocket on to the empty scene, which we develop as the following method:

rocket%

```
;; draw-on : Scene -> Scene
;; Draw this rocket on to scene.
(define (draw-on scn)
  (overlay/align/offset "center" "bottom"
                        ROCKET
                        0 (add1 (send this dist))
                        scn))
```

At this point, we can construct `rocket%` objects and invoke methods.
Examples:

```
> (new rocket% 7)
(new rocket% 7)
> (send (new rocket% 7) next)
(new rocket% 211/30)
> (send (new rocket% 80) render)
```
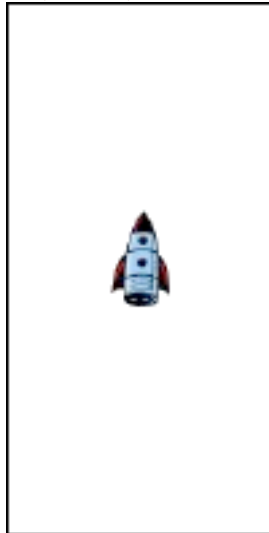

1.2.3   A big-bang oriented to objects

It's now fairly easy to construct a program using `rocket%` objects that is of the generic
form of a `big-bang` animation:

```
(big-bang <world0>            ; World
          (on-tick <tick>)    ; World -> World
          (to-draw <draw>))   ; World -> Scene
```

We can again define a world as a rocket:

```
;; A World is a Rocket.
```

We now need to construct `Rocket -> Rocket` and `Rocket -> Scene` functions—but the work of these functions is already taken care of by the `next` and `render` methods. Thus we construct simple functions that call the appropriate method on the given rocket:

```
(require 2htdp/universe)
(big-bang (new rocket% 0)
          (on-tick (λ (r) (send r next)))
          (to-draw (λ (r) (send r render)))))
```

This creates the desired animation, but something should stick out about the above code. The `big-bang` system works by giving a piece of data (a number, a position, an image, an object, etc.), and a set of functions that operate on that kind of data. That sounds a lot like... *an object!* It's almost as if the interface for `big-bang` were designed for, but had to fake, objects.

Now that we have objects proper, we can use a new `big-bang` system has an interface more suited to objects. To import this OO-style `big-bang`, add the following to the top of your program:

```
(require class/universe)
```

In the functional setting, we had to explicitly give a piece of data representing the state of the initial world and list which functions should be used for each event in the system. In other words, we had to give both data and functions to the `big-bang` system. In an object-oriented system, the data and functions are already packaged together, and thus the `big-bang` form takes a single argument: an object that both represents the initial world and implements the methods needed to handle system events such as `to-draw` and `on-tick`.

So to launch our rocket, we simply do the following:

```
(big-bang (new rocket% 0))
```

In order to handle events, we need to add the methods `on-tick` and `to-draw` to `rocket%`:

<div align="right">`rocket%`</div>

```
;; on-tick : -> World
;; Tick this world
(define (on-tick) ...)

;; to-draw : -> Scene
;; Draw this world
(define (to-draw) ...)
```

These methods, for the moment, are synonymous with `next` and `render`, so their code is simple:

<div align="right">`rocket%`</div>

```
(define (on-tick) (send this next))
(define (to-draw) (send this render))
```

Our complete program is:

```
#lang class/0
(require 2htdp/image)
(require class/universe)

; A World is a Rocket.

; A Rocket is a (new rocket% NonNegativeNumber).
; Interp: distance from the ground to base of rocket in AU.

(define CLOCK-SPEED  1/30) ; SEC/TICK
(define ROCKET-SPEED 1)    ; AU/SEC
(define DELTA (* CLOCK-SPEED ROCKET-SPEED)) ; AU/TICK
```



```
(define ROCKET       ) ; Use rocket key to insert the rocket here.
(define WIDTH 100)    ; PX
(define HEIGHT 200)   ; PX
(define MT-SCENE (empty-scene WIDTH HEIGHT))

(define-class rocket%
  (fields dist)

  ; next : -> Rocket
  ; Compute next position of this rocket after one tick of time.
  (check-expect (send (new rocket% 10) next)
                (new rocket% (+ 10 DELTA)))
  (define (next)
    (new rocket% (+ (send this dist) DELTA)))

  ; render : -> Scene
  ; Render this rocket as a scene.
  (check-expect (send (new rocket% 0) render)
                (overlay/align/offset "center" "bottom"
                                      ROCKET
                                      0 1
                                      MT-SCENE))
  (define (render)
    (send this draw-on MT-SCENE))

  ; draw-on : Scene -> Scene
```

```
; Draw this rocket on to scene.
(define (draw-on scn)
  (overlay/align/offset "center" "bottom"
                        ROCKET
                        0 (add1 (send this dist))
                        scn))

; on-tick : -> World
; Tick this world
(define (on-tick) (send this next))

; to-draw : -> Scene
; Draw this world
(define (to-draw) (send this render)))

; Lift off!
(big-bang (new rocket% 0))
```

You've now seen the basics of how to write programs with objects.

1.3 A Brief History of Objects

Objects are an old programming concept that first appeared in the late 1950s and early 1960s just across the Charles river at MIT in the AI group that was developing Lisp. Simula 67, a language developed in Norway as a successor to Simula I, introduced the notion of classes. In the 1970s, Smalltalk was developed at Xerox PARC by Alan Kay and others. Smalltalk and Lisp and their descendants have influenced each other ever since. Object-oriented programming became one of the predominant programming styles in the 1990s. This coincided with the rise of graphical user interfaces (GUIs), which objects model well. The use of object and classes to organize interactive, graphical programs continues today with libraries such as the Cocoa framework for Mac OS X.

1.4 Exercises

1.4.1 Complex, with class

For this exercise, you will develop a class-based representation of class-based representation ofcomplex numbers, which are used in several fields, including: engineering, electromagnetism, quantum physics, applied mathematics, and chaos theory.

A *complex number* is a number consisting of a *real part* and an *imaginary part*. It can be written in the mathematical notation *a+bi*, where *a* and *b* are real numbers, and *i* is the standard imaginary unit with the property $i^2 = \longrightarrow 1$.

Complex numbers are so useful, it turns out they are included in the set of numeric values that Racket supports. The Racket notationRacket notation for writing down

complex numbers is 5+3i, where this number has a real part of 5 and an imaginery part of 3; 4-2i has a real part of 4 and imaginary part of -2. (Notice that complex numbers *generalize* the real numbers since any real number can be expressed as a complex number with an imaginery part of 0.) Arithmetic operations on complex numbers work as they should, so for example, you can add, subtract, multiply, and divide complex numbers. (One thing you can't do is *order* the complex numbers, so < and friends work only on real numbers.)

Examples:

```
; Verify the imaginary unit property.
> (sqr (sqrt -1))
-1
> (sqr 0+1i)
-1
; Arithmetic on complex numbers.
> (+ 2+3i 4+5i)
6+8i
> (- 2+3i 4+5i)
-2-2i
> (* 2+3i 4+5i)
-7+22i
> (/ 2+3i 4+5i)
23/41+2/41i
; Complex numbers can't be ordered.
> (< 1+2i 2+3i)
<: contract violation
  expected: real?
  given: 1+2i
  argument position: 1st
  other arguments...:
   2+3i
; Real numbers are complex numbers with an imaginary part of 0,
; so you can perform arithmetic with them as well.
> (+ 2+3i 2)
4+3i
> (- 2+3i 2)
0+3i
> (* 2+3i 2)
4+6i
> (/ 2+3i 2)
1+3/2i
> (magnitude 3+4i)
5
```

Supposing your language was impoverished and didn't support complex numbers, you should be able to build them yourself since complex numbers are easily represented as a pair of real numbers—the real and imaginary parts.

Design a structure-based data representation for `Complex` values. Design the functions `=?`, `plus`, `minus`, `times`, `div`, `sq`, `mag`, and `sqroot`. Finally, design a utility function `to-number` which can convert `Complex` values into the appropriate Racket complex number. Only the code and tests for `to-number` should use Racket's complex (non-real) numbers and arithmetic since the point is to build these things for yourself. However, you can use Racket to double-check your understanding of complex arithmetic.

For mathematical definitions of complex number operations, see the Wikipedia entries on complex numbers and the square root of a complex number.

```
> (define c-1  (make-cpx -1 0))

> (define c0+0 (make-cpx 0 0))

> (define c2+3 (make-cpx 2 3))

> (define c4+5 (make-cpx 4 5))

> (=? c0+0 c0+0)
#t
> (=? c0+0 c2+3)
#f
> (=? (plus c2+3 c4+5)
      (make-cpx 6 8))
#t
```

Develop a class-based data representation for `Complex` values. Add accessor methods for extracting the `real` and `imag` parts. Develop the methods `=?`, `plus`, `minus`, `times`, `div`, `sq`, `mag`, `sqroot` and `to-number`.
Examples:

```
; Some example Complex values.
> (define c-1  (new complex% -1 0))

> (define c0+0 (new complex% 0 0))

> (define c2+3 (new complex% 2 3))

> (define c4+5 (new complex% 4 5))

; Verify the imaginary unit property.
> (send c-1 mag)
1
> (send c-1 sqroot)
(new complex% 0 1)
> (send (send (send c-1 sqroot) sq) =? c-1)
#t
```

```
> (send (send (new complex% 0 1) sq) =? c-1)
#t
; Arithmetic on complex numbers.
> (send c0+0 =? c0+0)
#t
> (send c0+0 =? c2+3)
#f
> (send (send c2+3 plus c4+5) =?
        (new complex% 6 8))
#t
> (send (send c2+3 minus c4+5) =?
        (new complex% -2 -2))
#t
> (send (send c2+3 times c4+5) =?
        (new complex% -7 22))
#t
> (send (send c2+3 div c4+5) =?
        (new complex% 23/41 2/41))
#t
> (send (new complex% 3 4) mag)
5
```

1.4.2   Circles

For this exercise, you will develop a structure-based representation of circles and functions that operate on circles, and then develop a class-based representation of circles.

A *circle* has a radius and color. They also have a position, which is given by the coordinates of the center of the circle (using the graphics coordinates system).

1. **The `circ` structure and functions.**

   Design a structure-based data representation for `Circle` values.

   Design the functions `=?`, `area`, `move-to`, `move-by`, `stretch`, `draw-on`, `to-image`, `within?`, `overlap?`, and `change-color`.

   Here are a few examples to give you some ideas of how the functions should work (note you don't necessarily need to use the same structure design as used here).

   First, let's define a few circles we can use:

   ```
   > (define c1 (make-circ 25 "red" 100 70))

   > (define c2 (make-circ 50 "blue" 90 30))

   > (define c3 (make-circ 10 "green" 50 80))
   ```
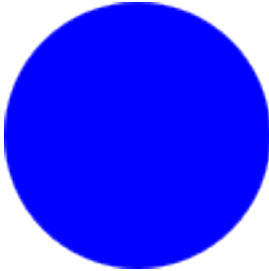
A `(make-circ R C X Y)` is interpreted as a circle of radius `R`, color `C`, and centered at position (`X`,`Y`) in graphics-coordinates.

The `to-image` function turns a circle into an image:

> `(to-image c1)`

> `(to-image c2)`

> `(to-image c3)`

While the `draw-on` function draws a circle onto a given scene:

> `(draw-on c1 (empty-scene 200 200))`
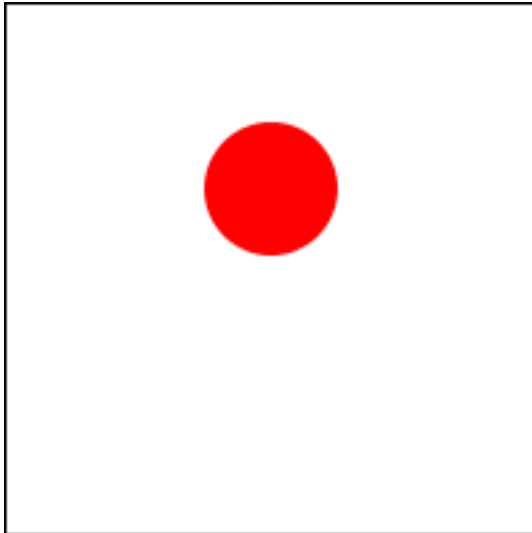
> `(draw-on c2 (empty-scene 200 200))`
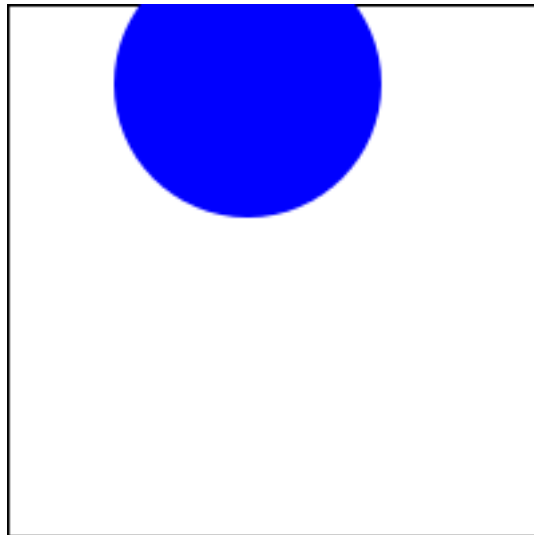
> `(draw-on c3 (empty-scene 200 200))`
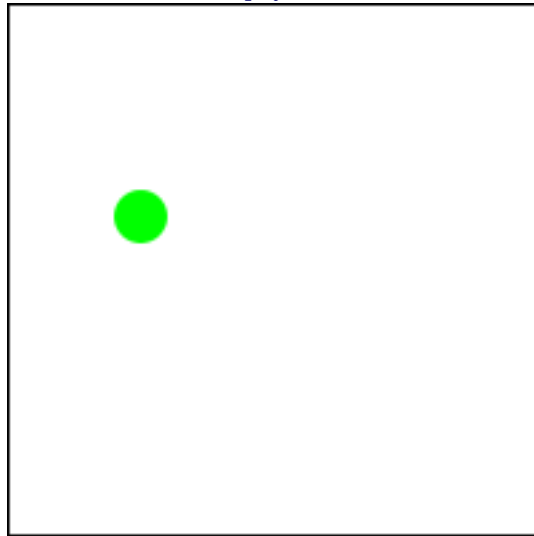
> `(draw-on c1 (draw-on c2 (draw-on c3 (empty-scene 200 200))))`
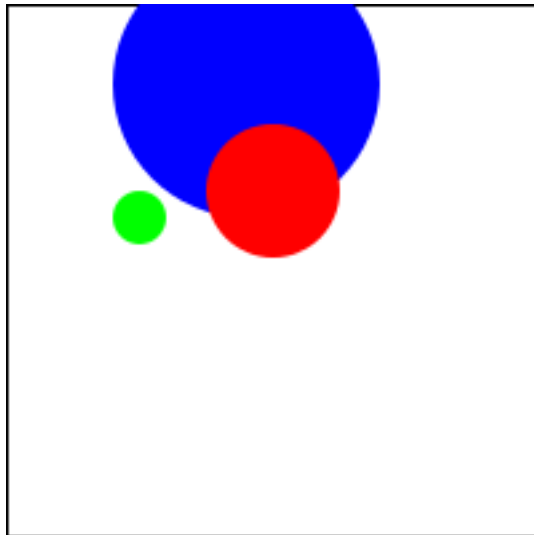
The `area` function computes the area of a circle:

```
> (area c1)
1963.4954084936207
> (area c2)
7853.981633974483
> (area c3)
314.1592653589793
```

The `move-to` function moves a circle to be centered at the given coordinates:

```
> (draw-on (move-to c1 100 100) (empty-scene 200 200))
```
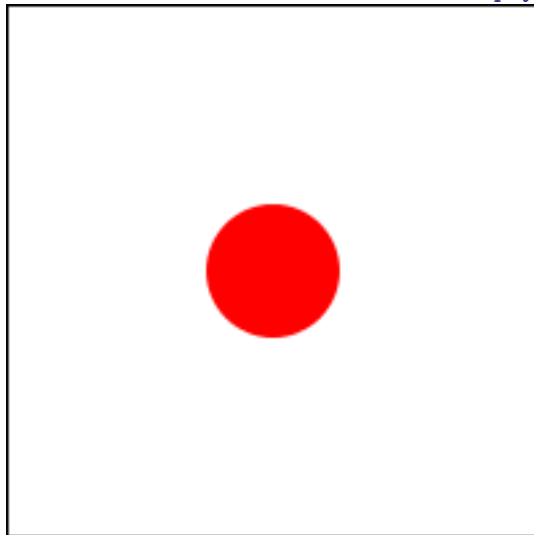
While `move-by` moves a circle by the given change in coordinates:

```
> (draw-on (move-by c1 -30 20) (empty-scene 200 200))
```
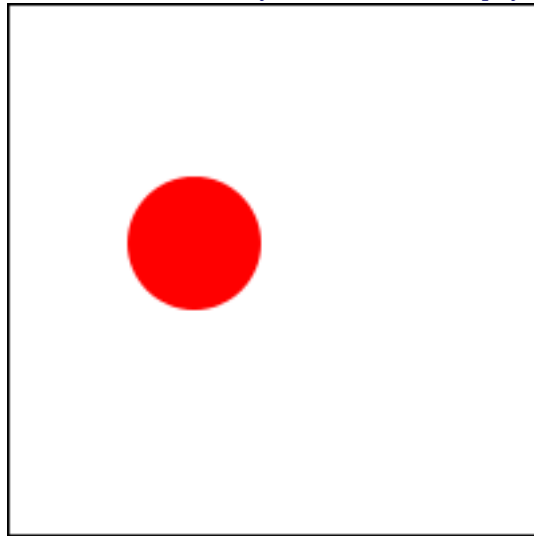
The `within?` function tells us whether a given position is located within the circle; this includes any points on the edge of the circle:

```
> (within? c1 (make-posn 0 0))
#f
> (within? c1 (make-posn 110 80))
#t
```

The `change-color` function produces a circle of the given color:

```
> (to-image (change-color c1 "purple"))
```

The `=?` function compares two circle for equality; two rectangles are equal if they have the same radius and center point—we ignore color for the purpose of equality:

```
> (=? c1 c2)
#f
> (=? c1 c1)
#t
> (=? c1 (change-color c1 "purple"))
#t
```

The `stretch` function scales a circle by a given factor:

```
> (draw-on (stretch c1 3/2) (empty-scene 200 200))
```
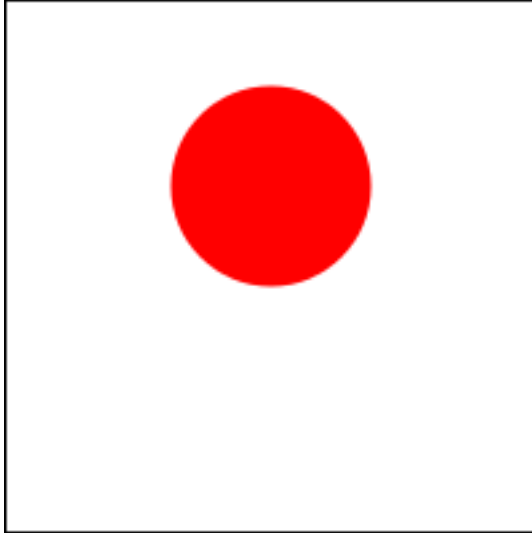

The `overlap?` function determines if two circles overlap at all:

```
> (overlap? c1 c2)
#t
> (overlap? c2 c1)
#t
> (overlap? c1 c3)
#f
```

2. **The `circ%` class.**

   Develop a class-based data representation for `Circle` values. Develop the *methods* corresponding to all the functions above.

   The methods should work similar to their functional counterparts:

   ```
   > (define c1 (new circ%  25 "red" 100 70))

   > (define c2 (new circ% 50 "blue" 90 30))

   > (send c1 area)
   1963.4954084936207
   > (send c1 draw-on (empty-scene 200 200))
   ```
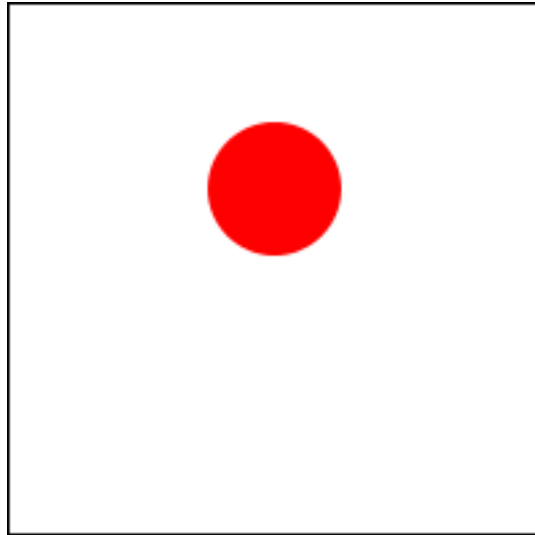
Chapter 2

Classes of Objects: Data Definitions

One of the most important lessons of *How to Design Programs* is that the structure of code follows the structure of the data it operates on, which means that the structure of your code can be derived *systematically* from your data definitions. In this chapter, we see how to apply the design recipe to design data represented using classes as well as operations implemented as methods in these classes.

We've seen various kinds of data definitions:

1. Atomic: numbers, images, strings, ...

2. Compound: structures, posns, ...

3. Enumerations: colors, key events, ...

4. Unions: atoms, ...

5. Recursive unions: trees, lists, matryoshka dolls, s-expressions, ...

6. Functions: infinite sets, sequences, ...

Each of these kinds of data definitions can be realized with objects. In this chapter, we'll examine how each the first five are implemented with a class-based design. We'll return to representing functions later.

2.1   Atomic and Compound Data

We already saw how to program with the object equivalent of atomic data in the chapter 2 chapter. If you worked through the section 2.4.1 exercise, you've already seen how to program with compound data, too.

Stepping back, we can see that the way to represent some fixed number N of data is with a class with N fields. For example, a position can be represented by a pair (x,y) of real numbers:

```
;; A Posn is (new posn% Real Real)
(define-class posn%
  (fields x y))
```

Methods can compute with any given arguments and the object that calling the method, thus the template for a `posn%` method is:

```
;; posn%-method : Z ... -> ???
(define (posn%-method z ...)
  (... (send this x) (send this y) z ...))
```

Here we see that our template lists the available parts of the `posn%` object, in particular the two fields `x` and `y`.

2.2   Enumerations

An *enumeration* is a data definition for a finite set of possibilities. For example, we can represent a traffic light like the ones on Huntington Avenue with a finite set of symbols, as we did in Fundies I:

```
;; A Light is one of:
;; - 'Red
;; - 'Green
;; - 'Yellow
```

Following the design recipe, we can construct the template for functions on `Lights`:

```
;; light-function : Light -> ???
(define (light-function l)
  (cond [(symbol=? 'Red l) ...]
        [(symbol=? 'Green l) ...]
        [(symbol=? 'Yellow l) ...]))
```

Finally, we can define functions over `Lights`, following the template.

```
;; next : Light -> Light
;; Next light after the given light
(check-expect (next 'Green) 'Yellow)
(check-expect (next 'Red) 'Green)
(check-expect (next 'Yellow) 'Red)
(define (next l)
  (cond [(symbol=? 'Red l) 'Green]
        [(symbol=? 'Green l) 'Yellow]
        [(symbol=? 'Yellow l) 'Red]))
```

That's all well and good for a function-oriented design, but we want to design this using classes, methods, and objects.

There are two obvious possibilities. First, we could create a `light%` class, with a field holding a `Light`. However, this fails to use classes and objects to their full potential. Instead, we will design a class for each state the traffic light can be in. Each of the three classes will have their own implementation of the `next` method, producing the appropriate `Light`.

```
#lang class/0
;; A Light is one of:
;; - (new red%)
;; - (new green%)
;; - (new yellow%)

(define-class red%
  ;; next : -> Light
  ;; Next light after red
  (check-expect (send (new red%) next) (new green%))
  (define (next)
    (new green%)))

(define-class green%
  ;; next : -> Light
  ;; Next light after green
  (check-expect (send (new green%) next) (new yellow%))
  (define (next)
    (new yellow%)))

(define-class yellow%
  ;; next : -> Light
  ;; Next light after yellow
  (check-expect (send (new yellow%) next) (new red%))
  (define (next)
    (new red%)))
```

If you have a `Light`, `L`, how do you get the next light?

```
(send L next)
```

Note that there is no use of `cond` in this program, although the previous design using functions needed a `cond` because the `next` function has to determine *what kind of light is the given light*. However in the object-oriented version there's no use of a `cond` because we ask an object to call a method; each kind of light has a different `next` method that knows how to compute the appropriate next light. Notice how the purpose statements are revised to reflect knowledge based on the class the method is in; for example, the `next` method of `yellow%` knows that this light is yellow.

2.3   Unions and Recursive Unions

Unions are a generalization of enumerations to represent infinite families of data. One example is *binary trees*, which can contain arbitrary other data as elements. We'll now look at how to model binary trees of numbers, such as:

```
7           6                 8
           / \               / \
          8   4             2   1
             / \
            3   2
```
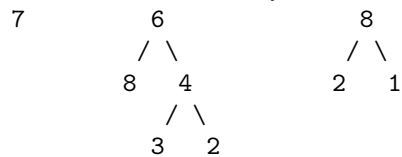
How would we represent this with classes and objects?

```
#lang class/0
;;    +- - - - - - - - - - - - - +
;;    | +- - - - - - - - - - - + |
;;    V V                       | |
;; A BT is one of:              | |
;; - (new leaf% Number)         | |
;; - (new node% Number BT BT)   | |
;;                      |  +- - -+ |
;;                      +- - - - --+
(define-class leaf%
  (fields number))

(define-class node%
  (fields number left right))

(define ex1 (new leaf% 7))
(define ex2 (new node% 6
                  (new leaf% 8)
                  (new node% 4
                       (new leaf% 3)
                       (new leaf% 2))))
(define ex3 (new node% 8
                  (new leaf% 2)
                  (new leaf% 1)))
```

We then want to design a method `count` which produces the number of numbers stored in a BT.

Here are our examples:

```
(check-expect (send ex1 count) 1)
(check-expect (send ex2 count) 5)
(check-expect (send ex3 count) 3)
```

Next, we write down the templates for methods of our two classes.

The template for `leaf%`:

leaf%

```
;; count : -> Number
;; count the number of numbers in this leaf
(define (count)
  (... (send this number) ...))
```

The template for `node%`:

node%

```
;; count : -> Number
;; count the number of numbers in this node
(define (count)
  (send this number) ...
  (send (send this left) count) ...
  (send (send this right) count) ...)
```

Now we provide a definition of the `count` method for each of our classes.

leaf%

```
;; count : -> Number
;; count the number of numbers in this leaf
(define (count)
  1)
```

node%

```
;; count : -> Number
;; count the number of numbers in this node
(define (count)
  (+ 1
     (send (send this left) count)
     (send (send this right) count)))
```

Next, we want to write the `double` function, which takes a number and produces two copies of the BT with the given number at the top. Here is a straightforward implementation for `leaf%`:

leaf%

```
;; double : Number -> BT
;; double this leaf and put the number on top
(define (double n)
  (new node%
       n
       (new leaf% (send this number))
       (new leaf% (send this number))))
```

Note that `(new leaf% (send this number))` is just constructing a new `leaf%` object just like the one we started with. Fortunately, we have a way of referring to ourselves, using the identifier `this`. We can thus write the method as:

leaf%

```
;; double : Number -> BT
;; double this leaf and put the number on top
(define (double n)
  (new node% n this this))
```

Since these two methods are so similar, you may wonder if they can be abstracted to avoid duplication. We will see how to do this in a subsequent class.

For node%, the method is very similar:

node%

```
;; double : Number -> BT
;; double this node and put the number on top
(define (double n)
  (new node% n this this))
```

The full BT code is now:

```
#lang class/0
;;    +- - - - - - - - - - - - - +
;;    | +- - - - - - - - - - - + |
;;    V V                       | |
;; A BT is one of:              | |
;; - (new leaf% Number)         | |
;; - (new node% Number BT BT)   | |
;;                        |   +- - -+ |
;;                        +- - - - --+
(define-class leaf%
  (fields number)
  ;; count : -> Number
  ;; count the number of numbers in this leaf
  (define (count)
    1)

  ;; double : Number -> BT
  ;; double the leaf and put the number on top
  (define (double n)
    (new node% n this this)))

(define-class node%
  (fields number left right)
  ;; count : -> Number
  ;; count the number of numbers in this node
  (define (count)
    (+ 1
       (send (send this left) count)
       (send (send this right) count)))
```

```
  ;; double : Number -> BT
  ;; double the node and put the number on top
  (define (double n)
    (new node% n this this)))

(define ex1 (new leaf% 7))
(define ex2 (new node% 6
                 (new leaf% 8)
                 (new node% 4
                      (new leaf% 3)
                      (new leaf% 2))))
(define ex3 (new node% 8
                 (new leaf% 2)
                 (new leaf% 1)))

(check-expect (send ex1 count) 1)
(check-expect (send ex2 count) 5)
(check-expect (send ex3 count) 3)

(check-expect (send ex1 double 5)
              (new node% 5 ex1 ex1))
(check-expect (send ex3 double 0)
              (new node% 0 ex3 ex3))
```

2.4 Revisiting the Rocket

2.4.1 Landing and taking off

Let's now revise our section 2.2 program so that the rocket first descends toward the ground, lands, then lifts off again. Our current representation of a world is insufficient since it's ambiguous whether we are going up or down. For example, if the rocket is at 42, are we landing or taking off? There's no way to know. We can revise our data definition to included a representation of this missing information. As we hear this revised description, the idea of a union data definition should jump out: "a rocket is either landing or taking off." Let's re-develop our program with this new design.

Our revised class definition is then:

```
;; A World is a Rocket.

;; A Rocket is one of:
;; - (new takeoff% Number)
;; - (new landing% Number)

;; Interp: distance from the ground to base of rocket in AU,
;; either taking off or landing.
```

```
(define-class takeoff%
   (fields dist)
   ...)
(define-class landing%
   (fields dist)
   ...)
```

The signatures for our methods don't change, however we now have two sets of methods to implement: those for rockets taking off, and those for landing rockets.

First, let's make some test cases for the `next` method. We expect that a rocket taking off works just as before:

```
(check-expect (send (new takeoff% 10) next)
              (new takeoff% (+ 10 DELTA-Y)))
```

However, when landing we expect the rocket to be descending toward the ground. For simplicity, let's specify the rocket descends as fast as it ascends:

```
(check-expect (send (new landing% 100) next)
              (new takeoff% (- 100 DELTA-Y)))
```

There is an important addition case though. When the rocket is descending and gets close to the ground, we want it to land. So when the rocket is descending and less than `DELTA-Y` units from the ground, we want its next state to be on the ground, ready to lift off:

```
(check-expect (send (new landing% (sub1 DELTA-Y)) next)
              (new takeoff% 0))
```

Based on these examples, we can now define the `next` method in the `landing%` and `takeoff%` classes:

<div align="right">

takeoff%

</div>

```
;; next : -> Rocket
;; Compute next position of this ascending rocket after one tick of time.
(define (next)
  (new takeoff% (+ (send this dist) DELTA-Y)))
```

<div align="right">

landing%

</div>

```
;; next : -> Rocket
;; Compute next position of this descending rocket after one tick of time.
(define (next)
  (cond [(< (send this dist) DELTA-Y) (new takeoff% 0)]
        [else (new landing% (- (send this dist) DELTA-Y))]))
```

Now let's turn to the remaining methods such as `render`. When rendering a rocket, it's clear that it doesn't matter whether the rocket is landing or taking off; it will be

drawn the same. This leads to having two *identical* definitions of the `render` method in both `takeoff%` and `landing%`. Since the method relies upon the helper method `draw-on`, we likewise have two identical definitions of `draw-on` in `takeoff%` and `landing%`.

`landing%` and `takeoff%`

```
;; render : -> Scene
;; Render this rocket as a scene.
(define (render)
  (send this draw-on MT-SCENE))

; draw-on : Scene -> Scene
; Draw this rocket on to scene.
(define (draw-on scn)
  (overlay/align/offset "center" "bottom"
                        ROCKET
                        0 (add1 (send this dist))
                        scn))
```

This duplication of code is unsettling, but for now let's just live with the duplication. We could abstract the code by defining a *function* and calling the function from both methods, but as we try to focus on object-oriented designs, let's instead recognize there's a need for an object-oriented abstraction mechnaism here and revisit the issue later in the chapter on chapter 13.

We can experiment and see ascending rockets climb and descending rockets land:

Examples:

```
> (send (new landing% 5) next)
(new landing% 149/30)
> (send (new takeoff% 5) next)
(new takeoff% 151/30)
> (send (new landing% 0) next)
(new takeoff% 0)
> (send (new landing% (quotient HEIGHT 2)) render)
```
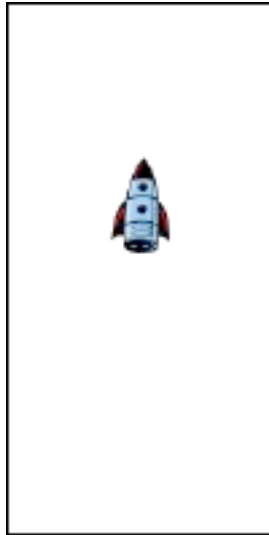
Implementing the needed methods for a `big-bang` animation is straightforward:

`landing%` and `takeoff%`

```
(define (on-tick) (send this next))
(define (to-draw) (send this render))
```

And to run the animation, just start `big-bang` with a landing rocket:

```
(big-bang (new landing% HEIGHT))
```

2.4.2   Adding a satellite

Let's now add an orbiting satellite. To do so, let's first forget about rockets and make an satellite animation. The satellite is represented by a class with a single field—a number giving the distance from the date line, which we'll draw at the left of the screen, to the center of the satellite. When the satellite gets to the edge of the screen, it will wrap around starting over again at the date line.

```
(define CLOCK-SPEED  1/30) ; SEC/TICK
(define SATELLITE-SPEED 1) ; AU/SEC
(define DELTA-X (* CLOCK-SPEED SATELLITE-SPEED)) ; AU/TICK

(define SATELLITE (circle 30 "solid" "red"))
(define WIDTH 100)   ; PX
(define HEIGHT 200)  ; PX
(define SATELLITE-Y (quotient HEIGHT 4))
(define MT-SCENE (empty-scene WIDTH HEIGHT))

;; A World is a Satellite.
```

```
;; A Satellite is a (new satellite% Number).
;; Interp: distance in AU from date line to center of satellite.

(define-class satellite%
  (fields dist)

  ;; next : -> Satellite
  ;; Move this satellite distance travelled in one tick.
  (define (next)
    (local [(define n (+ (send this dist) DELTA-X))]
      (new satellite% (cond [(> n WIDTH) (- n WIDTH)]
                            [else n]))))))
```

Drawing the satellite is a little more tricky that the rocket because the satellite can appear on both the left and right side of the screen as it passes over the date line. A simple trick to manage this is to draw *three* satellites, each a full "day" behind and ahead of the current satellite, thus when the satellite is just past the date line, the day ahead image appears on the right, and when the satellite approaches the end of the day, the day behind satellite appears on the left. To accomodate this, we define a helper method that draws the satellite at given day offsets.

satellite%

```
;; render : -> Scene
;; Render this satellite as a scene.
(define (render)
  (send this draw-on MT-SCENE))

;; draw-on : Scene -> Scene
;; Draw this satellite on scene.
(define (draw-on scn)
  (send this draw-on/offset -1
        (send this draw-on/offset 0
              (send this draw-on/offset 1
                    MT-SCENE))))

;; draw-on/offset : Number Scene -> Scene
;; Draw this satellite on scene with given day offset.
(define (draw-on/offset d scn)
  (place-image SATELLITE
               (+ (send this dist) (* d WIDTH))
               SATELLITE-Y
               scn))
```

Examples:

```
> (send (new satellite% 0) next)
```

```
(new satellite% 1/30)
> (send (new satellite% (quotient WIDTH 2)) render)
```
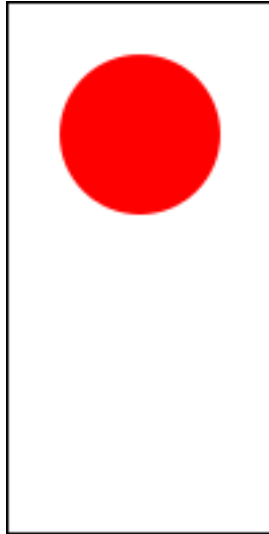


```
> (send (new satellite% 0) render)
```
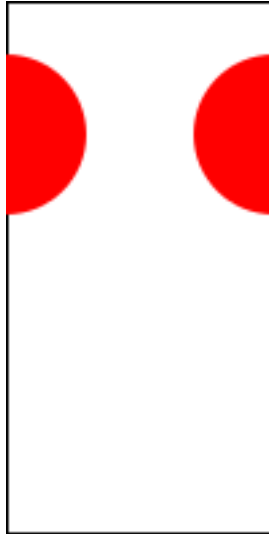

We can now add the needed methods to animate satellites with `big-bang`:

`satellite%`

```
(define (on-tick) (send this next))
(define (to-draw) (send this render))
```

And then animate a satellite with:

```
(big-bang (new satellite% 0))
```

Now we have animations of rockets and of satellites, but putting the pieces together is simple. We need to revise our data definition. Let's make a new class of compound that *contains* a rocket and a satellite and implement the methods needed to make an animation:

```
;; A World is a (new space% Rocket Satellite).
(define-class space%
  (fields rocket satellite)

  (define (on-tick)
    (new space%
         (send (send this rocket) next)
         (send (send this satellite) next)))

  (define (to-draw)
    (send (send this rocket) draw-on
          (send (send this satellite) draw-on
                MT-SCENE))))
```

Example:

```
> (send (new space%
             (new landing% (quotient HEIGHT 2))
             (new satellite% (quotient WIDTH 3)))
        to-draw)
```
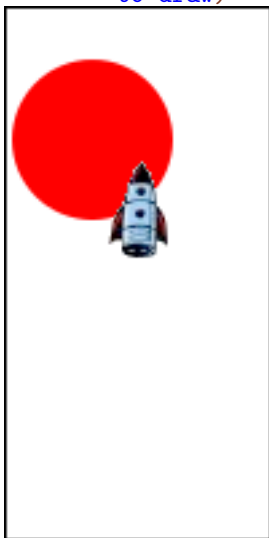

Finally, to animate the whole thing, we just call `big-bang` with an initial `space%` object:

```
(big-bang (new space%
```

```
                    (new landing% HEIGHT)
                    (new satellite% 0)))
```

2.5   Exercises

2.5.1   Lists of Numbers

Design classes to represent lists of numbers. Implement the methods `length`, `append`, `sum`, `prod`, `contains?`, `reverse`, `map`, and `max`. Note that `max` raises some interesting design decisions in the case of the empty list. One solution is to define the `max` of the empty list as negative infinity, `-inf.0`, a number smaller than every other number (except itself). Another solution is to only define `max` for non-empty lists of numbers.

2.5.2   Home on the Range

A *range* represents a set of numbers between two endpoints. To start with, you only need to consider ranges that *include* the smaller endpoint and *exclude* the larger endpoint—such ranges are called *half-open*. For example, the range [3,4.7) includes all of the numbers between 3 and 4.7, including 3 but *not* including 4.7. So 4 and 3.0000001 are both in the range, but 5 is not. In the notation used here, the "[" means include, and the ")" means exclude.

- Design a representation for ranges and implement the `in-range?` method, which determines if a number is in the range. For example, the range [3,7.2) includes the numbers 3 and 5.0137, but not the numbers -17 or 7.2.

- Extend the data definition and implementation of ranges to represent ranges that *exclude* the low end of the range and *include* the high end, written (lo,hi].

- Add a `union` method to the interface for ranges and implement it in all range classes. This method should consume a range and produces a new range that includes all the numbers in this range *and* all the numbers in the given range.

    You may extend your data definition for ranges to support this method.

    Don't worry if your initial design duplicates code; you can abstract later.

Chapter 3

Classes of Objects: Interface Definitions

In this chapter, we take an alternative perspective on defining sets of objects; we can characterize objects not just by their construction, as done with a data definition, but also by the methods they support. We call this characterization an interface definition. As we'll see, designing to interfaces leads to generic and extensible programs.

3.1   Lights, revisited

Let's take another look at the `Light` data definition we developed in section 3.2. We came up with the following data definition:

```
;; A Light is one of:
;; - (new red%)
;; - (new green%)
;; - (new yellow%)
```

We started with a `next` method that computes the successor for each light. Let's also add a `draw` method and then build a `big-bang` animation for a traffic light.

```
#lang class/0
(require 2htdp/image)
(define LIGHT-RADIUS 20)

(define-class red%
  ;; next : -> Light
  ;; Next light after red
  (check-expect (send (new red%) next) (new green%))
  (define (next)
    (new green%))
```

```
  ;; draw : -> Image
  ;; Draw this red light
  (check-expect (send (new red%) draw)
                (circle LIGHT-RADIUS "solid" "red"))
  (define (draw)
    (circle LIGHT-RADIUS "solid" "red")))

(define-class green%
  ;; next : -> Light
  ;; Next light after green
  (check-expect (send (new green%) next) (new yellow%))
  (define (next)
    (new yellow%))

  ;; draw : -> Image
  ;; Draw this green light
  (check-expect (send (new green%) draw)
                (circle LIGHT-RADIUS "solid" "green"))
  (define (draw)
    (circle LIGHT-RADIUS "solid" "green")))

(define-class yellow%
  ;; next : -> Light
  ;; Next light after yellow
  (check-expect (send (new yellow%) next) (new red%))
  (define (next)
    (new red%))

  ;; draw : -> Image
  ;; Draw this yellow light
  (check-expect (send (new yellow%) draw)
                (circle LIGHT-RADIUS "solid" "yellow"))
  (define (draw)
    (circle LIGHT-RADIUS "solid" "yellow")))
```
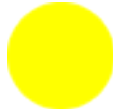
We can now create and view lights:

```
> (send (new green%) draw)
```



```
> (send (new yellow%) draw)
```

```
> (send (new red%) draw)
```


To create an animation we can make the following world:

```
(define-class world%
  (fields light)
  (define (tick-rate) 5)
  (define (to-draw)
    (send (send this light) draw))
  (define (on-tick)
    (new world% (send (send this light) next)))))

(require class/universe)
(big-bang (new world% (new red%)))
```

At this point, let's take a step back and ask the question: *what is essential to being a light?* Our data definition gives us one perspective, which is that for a value to be a light, that value must have been constructed with either `(new red%)`, `(new yellow%)`, or `(new green%)`. But from the world's perspective, what matters is not how lights are constructed, but rather what can lights compute. All the world does is call methods on the light it contains, namely the `next` and `draw` methods. We can rest assured that the light object understands the `next` and `draw` messages because, by definition, a light must be one of `(new red%)`, `(new yellow%)`, or `(new green%)`, and each of these classes defines `next` and `draw` methods. But it's possible we could relax the definition of what it means to be a light by just saying what methods an object must implement in order to be considered a light. We can thus take a constructor-agnostic view of objects by defining a set of objects in terms of the methods they understand. We call a set of method signatures (i.e., name, contract, and purpose statement) an *interface*.

3.2 A light of a different color

So let's consider an alternative characterization of lights not in terms of *what they are*, but rather *what they do*. Well a light does two things: it can render as an image and it can transition to the next light; hence our *interface definition* for a light is:

```
;; An ILight implements
;; next : -> ILight
;; Next light after this light.
;; draw : -> Image
;; Draw this light.
```

Now it's clear that every `Light` is an `ILight` because every `Light` implements the methods in the `ILight` interface, but we can imagine new kinds of implementations of

the `ILight` interface that are not `Light`s. For example, here's a class that implements
the `ILight` interface:

```
;; A ModLight is a (new mod-light% Natural)
;; Interp: 0 = green, 1 = yellow, otherwise red.
(define-class mod-light%
  (fields n)
  ;; next : -> ILight
  ;; Next light after this light.
  (define (next)
    (new mod-light% (modulo (add1 (send this n)) 3)))

  ;; draw : -> Image
  ;; Draw this light.
  (define (draw)
    (cond [(= (send this n) 0)
           (circle LIGHT-RADIUS "solid" "green")]
          [(= (send this n) 1)
           (circle LIGHT-RADIUS "solid" "yellow")]
          [else
           (circle LIGHT-RADIUS "solid" "red")])))
```

Now clearly a `ModLight` is never a `Light`, but every `ModLight` is an `ILight`.
Moreover, any program that is written for `ILight`s will work *no matter what imple-mentation we use*. So notice that the world program only assumes that its `light` field
is an `ILight`; this is easy to inspect—the world never assumes the light is constructed
in a particular way, it just calls `next` and `draw`. Which means that if we were to start
our program off with

```
(big-bang (new world% (new mod-light% 2)))
```

it would work exactly as before.

3.3   Representation inpedendence and extensibility

We've now developed a new concept, that of an *interface*, which is a collection of
method signatures. We say that an object *is* an instance of an interface whenever it
implements the methods of the interface.

The idea of an interface is already hinted at in the concept of a union of objects since
a function over a union of data is naturally written as a method in each class variant of
the union. In other words, to be an element of the union, an object must implement all
the methods defined for the union—the object must implement the union's interface.
But interfaces are about more than just unions. By focusing on interfaces, we can see
there are two important engineering principles that can be distilled even from this small
program:

1. Representation independence

   As we've seen with the simple world program that contains a light, when a program is written to use only the methods specified in an interface, then the program is *representation independent* with respect to the interface; we can swap out any implementation of the interface without changing the behavior of the program.

2. Extensibility

   When we write interface-oriented programs, it's easy to see that they are *extensible* since we can always design new implementations of an interface. Compare this to the construction-oriented view of programs, which defines a set of values once and for all.

These points become increasingly important as we design larger and larger programs. Real programs consist of multiple interacting components, often written by different people. Representation independence allows us to exchange and refine components with some confidence that the whole system will still work after the change. Extensibility allows us to add functionality to existing programs without having to change the code that's already been written; that's good since in a larger project, it may not even be possible to edit a component written by somebody else.

Let's look at the extensiblity point in more detail. Imagine we had developed the `Light` data definition and its functionality along the lines of *HtDP*. We would have (we omit `draw` for now):

```
;; A Light is one of:
;; - 'Red
;; - 'Green
;; - 'Yellow

;; next : Light -> Light
;; Next light after the given light
(check-expect (next 'Green) 'Yellow)
(check-expect (next 'Red) 'Green)
(check-expect (next 'Yellow) 'Red)
(define (next l)
  (cond [(symbol=? 'Red l) 'Green]
        [(symbol=? 'Green l) 'Yellow]
        [(symbol=? 'Yellow l) 'Red]))
```

Now imagine if we wanted to add a new kind of light—perhaps to represent a blinking yellow light. For such lights, let's assume the next light is just a blinking yellow light:

```
(check-expect (next 'BlinkingYellow) 'BlinkingYellow)
```

That's no big deal to implement *if we're allowed to revise* `next`—we just add another clause to `next` handle `'BlinkingYellow` lights. But what if we can't? What if

`next` were part of a module provided as a library? Well then life is more complicated; we'd have to write a new function, say `fancy-next`, that handled blinking lights and used `next` for all non-blinking lights. And while that gets us a new function with the desired behavior, that won't do anything for all the places the `next` function is used. If we're able to edit the code that uses `next`, then we can replace each use of `next` with `fancy-next`, but what if we can't...? Well then we're just stuck. If we cannot change the definition of `next` or all the places it is used, then it is not possible to extend the behavior of `next`.

Now let's compare this situation to one in which the original program was developed with objects and interfaces. In this situation we have an interface for lights and several classes, namely `red%`, `yellow%`, and `green%` that implement the `next` method. Now what's involved if we want to add a variant of lights that represents a blinking yellow light? We just need to write a class that implements `next`:

```
;; Interp: blinking yellow light
(define-class blinking-yellow%
  ;; next : -> ILight
  ;; Next light after this blinking yellow light.
  (check-expect (send (new blinking-yellow%) next)
                (new blinking-yellow%))
  (define (next) this))
```

Notice how we didn't need to edit `red%`, `yellow%`, or `green%` at all! So if those things are set in stone, that's no problem. Likewise, programs that were written to use the light interface will now work even for blinking lights. We don't need to edit any uses of the `next` method in order to make it work for blinking lights. This program is truly extensible.

3.4   Sharing Interfaces

```
;; A Posn implements
;; move-by : Real Real -> Posn
;; move-to : Real Real -> Posn
;; dist-to : Posn -> Real

;; A Segment implements
;; draw-on : Scene -> Scene
;; move-by : Real Real -> Segment
;; move-to : Real Real -> Segment
;; dist-to : Posn -> Real
```

Notice that any object that is a `Segment` is also a `Posn`.

3.5   Exercises

3.5.1   Beings, Zombies, and You

Q: I've designed a single interface `Being` that subsumes both `Zombie` and `Player` in the current assignment. Do I still have to design a `Zombie` and `Player` interface?

A: Yes. There are a couple reasons for this. One is that there really are some differences between the operations that should be supported by a player versus a zombie. For example, zombies eat brains; players don't. Another is that, as you are probably noticing, much of this course is about interface specification and implementation. As we build larger and larger programs, interfaces become a much more important engineering tool. An interface can be viewed as a contract—an agreement on the terms of engagement—between the *implementor* and the *consumer* of a software component. In this assignment, even though you are acting simultaneously as both of these parties, we are asking you to write down the agreement you are making between the world program that uses zombies and players and the classes that implement zombies and players. Part of our agreement with you, is that you'll write separate specifications; so that's what you need to do.

That said, if really believe that there should be a single uniform interface that all zombies *and* players should adhere to, you can write a `Being interface and program to it`.

Revise your Zombie! program.

3.5.2   Super Zombie!

Revise your design of the Zombie game to include a `Zombie` and `Player` interface. Implement a `live-zombie%` and `dead-zombie%` class that both implement your `Zombie` interface; implement a `player%` class that implements your `Player` interface.

Design a `world%` class for playing the Zombie game that interacts with the zombie and player objects only according to the interfaces you've designed, i.e. the `world%` class should work for *any* objects that correctly implement your zombie and player interfaces.

3.5.3   Modulo Zombie!

Using your interface design from the previous problem, design a `modulo-player%` class and a `modulo-live-zombie%` class that implement the `Player` and `Zombie` interfaces, respectively.

These alternative implementations should behave as follows: the player and the zombies may now "wrap around" on the screen. If a player goes off the top of the screen, they should re-appear on the bottom; if the go off of the left side, they should appear on the right, etc., and likewise for the zombies. When calculating in which direction they should go, the player and zombies should take into account the possibility of wrapping around the screen. So for example, if the player is on the far right side and there is a zombie on the far left side, the zombie should head left to wrap around the screen and quickly arrive upon the player and feast upon his or her brains. Similarly

if the mouse is on the very top and the player is on the very bottom, the player should move down to get to the top quickly.

If you need to make changes to your interface design to accommodate these new game requirements, you must re-implement your solution to problem 2 in order to satisfy the revised interfaces. In the end, the interfaces used and your implementation of the `world%` class be the same in both problem 2 and 3.

3.5.4   Mixed Zombie!

Experiment with different combinations of your classes from the previous exercises (only the player can wrap around; only the zombies can wrap around; some of the zombies and the player; some of the zombies, but not the player, etc.) until you find a combination you like best. Write down an expression that launches the game using this combination.

Chapter 4

Parameterized Data and Interfaces

4.1 Parametric data

Consider the parametric data definition for lists we studied last semester:

```
;; A [List X] is one of:
;; - empty
;; - (cons X [List X])
```

Recall this is really not just a single data definition, but a family of data definitions. We can obtain a different member of this family by plugging in some data definition for X; `List X` works just like a function—apply it to arguments results in its definition, but with `X` replaced by the argument. Unlike with a function, the arguments are not values, but *data definitions*. So for example, plugging in `Number` for `X`, written `[List Number]` results in:

```
;; A [List Number] is one of:
;; - empty
;; - (cons Number [List Number])
```

Applying `List` to `String` results in:

```
;; A [List String] is one of:
;; - empty
;; - (cons String [List String])
```

The process by which we obtained the parameterized data definition, `List X`, is one of abstraction. Looking back, we *started* with the non-parameterized data definitions:

```
;; A ListNumber is one of:
```

```
;; - empty
;; - (cons Number ListNumber)

;; A ListString is one of:
;; - empty
;; - (cons String ListString)
```

These definitions are so similar, it's natural to want to abstract them to avoid repreating this nearly identical code again and again. It's easy to see what is different: `Number` vs `String`, and that's exactly how we arrived at `[List X]`.

We can repeat the process for objects. Let's start with the non-parameterized data definitions we've been working worth:

```
;; A ListNumber is one of:
;; - (new emtpty%)
;; - (new cons% Number ListNumber)

;; A ListString is one of:
;; - (new empty%)
;; - (new cons% String ListString)

(define-class empty%)
(define-class cons%
  (fields first rest))
```

Abstracting out the data definition of elements results in:

```
;; A [List X] is one of:
;; - (new empty%)
;; - (new cons% X [List X])
```

4.2 Parametric interfaces

We have now developed a parametric data definition for lists, focusing on the representation of lists. If instead we focused the bevahiors of lists, we would arrive at a *parametric interface*:

```
;; A [List X] implements:
;; - cons : X -> [List X]
;;   Cons given value on to this list.
;; - first : -> X
;;   Get first element of this non-empty list.
;; - rest : -> [List X]
;;   Get the reset of this non-empty list.
```

We can implement the interface as follows:

```
;; A (new empty%) implements [List X]
(define-class empty%
  (define (cons x) (new cons% x this)))

;; A (new cons% X [List X]) implements [List X]
(define-class cons%
  (fields first rest)
  (define (cons x) (new cons% x this)))
```

4.3 Parameteric methods

We can design further methods:

```
;; A [List X] implements:
;; - cons : X -> [List X]
;;   Cons given value on to this list.
;; - first : -> X
;;   Get first element of this non-empty list.
;; - rest : -> [List X]
;;   Get the reset of this non-empty list.
;; - length : -> Natural
;;   Get the length of this list.
;; - append : [List X] -> [List X]
;;   Append this to given list.
;; - reverse : -> [List X]
;;   Reverse this list of elements.
;; - map : [X -> Y] -> [List Y]
;;   Map given function over this list.
;; - filter : [X -> Boolean] -> [List X]
;;   Select elements that satisfy the given predicate.
;; - foldr : [X Y -> Y] Y -> Y
;;   Fold right over this list.
;; - foldl : [X Y -> Y] Y -> Y
;;   Fold left over this list.
```

Again, the model of a parametric interface is as a *function* of classes of data. In this interface, X is the parameter, bound at the point of "A [List X] implements". That variable occurs several times within the definition and is replaced by the argument of List. But on closer inspection, there are other variables that are not bound, e.g. Y in:

```
;; - map : [X -> Y] -> [List Y]
```

To be precise, there's really another, implicit, parameter that doesn't range over the whole interface, but just the map method. We can make this implicit parameter in the contract notation by adding a class variable at the level of map:

```
;; - map [Y] : [X -> Y] -> [List Y]
```

So for example, we might have a `[List Number]`, in which case, the object has the `map` method:

```
;; - map [Y] : [Number -> Y] -> [List Y]
```

Even though the `X` has been replaced by `Number`, the `Y` parameter remains and only takes on a specific meaning from the function argument of `map`. So if `ns` is a `List Number`, we apply map to a `[Number -> String]` function, its contract is interpreted with `Y` replaced by `String`:
Examples:

```
> (define ns (new cons% 3 (new cons% 4 (new empty%))))

> (ns . map number->string)
(new cons% "3" (new cons% "4" (new empty%)))
```

4.4   Exercises

4.4.1   Parametric Lists

Design an implementation of the `[List X]` interface given in this chapter.

Chapter 5

Solidifying what we've done

So far, we've seen a number of different ways of specifying the creation and behavior of the data we work with. At this point, it's valuable to take a step back and consider all of the concepts we've seen.

5.1   Data Definitions

A data definition defines a *class*[1] of values by describing how instances of that data are constructed. In this book, we focus on particular new kind of value: *objects*, and data definitions will primarily define a class of objects. New kinds of objects are made with `define-class`, while instances of a class are made with a class constructor, written (`new class-name% arg ...`).

Data definitions can be built out of primitive data, such as `Numbers`, `Strings`, `Images`, etc., but also compound data can be represented with objects containing data. Data definitions can also be formed as the (possibly recursive) union of other data definitions. For example:

```
;; A ListofImage is one of:
;; - (new empty%)
;; - (new cons% Image ListofImage)
```

Data definitions may be parameterized, meaning a family of similar data definitions is simultaneously defined by use of variable parameters that range over other classes of values. For example:

```
;; A [Pair X Y] is one of:
;; - (new pair% X Y)
```

Here the `Pair` family of data definition is parameterized over classes of values `X` and `Y`.

---

[1]Here we mean "class" in the general sense of "a set of values," not to be confused with the concept of a "class" as definied by the `define-class` form.

5.2   Interface Definitions

Another way to define a class of values is by way of an *interface definition*. Unlike a data definition, which focuses on how data is represented, an interface defines a set of values by the operations that set of values support. Interfaces provide a means for defining a set of values independent of representation.

For example:

```
;; A [List X] implements

;; - empty : -> [List X]
;;    Produce an empty list
;; - cons : X -> [List X]
;;    Produce a list with the given element at the front.
;; - empty? : -> Boolean
;;    Determine if this list is empty.
;; - length : -> Number
;;    Count the elements in this list
;; ... and other methods ...
```

There are several important aspects of this interface defintion to note. First, it lists all of the methods that can be used on an `[List X]`, along with their contracts and purpose statements. Mere method names are not enough—with just a method name you have no idea how to use a method, or what to use it for. Second, interface defintions can have parameters (here X), just like data defintions. Third, there is no description of how to *construct* an `[List X]`. That's the job of data defintions that implement this interface.

Of course, just like data defintions don't have to be named, interface defintions don't have to be named either. If you need to describe an interface just once, it's fine to write the interface right there where you need it.

5.3   Contracts

Contracts describe the appropriate inputs and outputs of functions and methods. In the past, we've seen many contracts that refer to data defintions. In this class, we've also seen contracts that refer to interface defintions, like so:

```
;; [IList Number] -> [IList Number]
```

When describing the contract of a function or method, it's almost always preferable to refer to an interface definition instead of a data defintion that commits to a specific representation.

5.4   Design Recipe

Interfaces change the design recipe in one important way. In the Template step, we take an inventory of what is available in the body of a function or method. When designing

a method, we have the following available to us:

- The fields of this object, accessed with accessor methods,

- The methods of this object, accessed by calling them,

- And the operations of the arguments, which are given by their *interfaces*.

For example, if a method takes an input `a-list` which is specified in the contract to be an `IList`, then we know that `(send a-list empty?)`, `(send a-list length)`, and so on.

5.5 Design Choices

Q: Which is considered a better design: a union with two variants, or a single variant with a Boolean field that indicates "which part of the union this data belongs to"? For example, is it better to have a `live-zombie%` and `dead-zombie%` class or a single `zombie%` class with a `dead?` field.

A: One of the themes of this and last semester is that *once you have settled on choice for the representation of information in your program, the structure of your program follows the structure of that data*. We've trained you to systematically derive code structure from data structure; there's a recipe—you don't even have to think. That's great because it frees up significant quantities of the most precious and limited resource in computing: your brain. But unfortunately, that recipe kicks in only after you've chosen how information will be represented, i.e. after you've written down data definitions. Much of the hard work in writing programs, and where your creative energy and brain power is really needed, is going from information to representation. There is no recipe for this part of program design. You need to develop the ability to analyze problems, take an account of what knowledge needs to be represented to solve a problem, and practice making decisions about how that knowledge can be represented computationally. The good news is you don't have to be struck by divine inspiration to manage this step. Program design is a process of iterative refinement: make some choices, follow the recipe. You will discover ways to improve your initial choices, so go back and revise, then carry out the changes in code structure those design decision entail. Rinse and repeat.

This is a long winded way of saying: there is no universal "right" answer to this question. It will depend on the larger context. That said, there are some important things to take into account for this particular question. It is much easier to add new variants to a union than it is to create Boolean values other than `true` and `false`. Good program design is often based on anticipating future requirements. If you think it's possible that there might be some other kind of zombie down the road, the union design will be less painful to extend and maintain.

5.6   Exercises

5.6.1   JSON, Jr.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write; it is based on a subset of the JavaScript Programming Language. There are a large number of huge corpora of data available on the internets. In order to write programs that can make use of this data, you'll need to design a data representation of JSON values.

A JSON value can take on the following forms:[2]

- An *object* is an unordered set of name/value pairs.

- An *array* is an ordered collection of values.

- A *value* can be a string, or a number, or true or false or null, or an object or an array. These structures can be nested.

This definition is written in the terminology of JSON, so don't confuse a JSON object with an object in the `class` language. Likewise, don't confuse JSON strings with `class` strings.

To start things off simply, let's focus on a subset of JSON we'll call "JSON, Jr." which consists solely of strings and arrays. Using the notation of JSON, the following are examples of JSON, Jr. values:

- `"this is JSON, Jr."`

- `[ "this is JSON", "Jr." ]`

- `[ [], [ "So", "is" ], "this" ]`

The first example is just a string. The second is an array of two JSON elements, which are both strings. The third is another array, but of three elements: the first is an array of zero elements, the second is an array of two strings, and the third is a string.

1. Design an object-based data representation for JSON, Jr values.

2. Design a method for counting the number of strings in a JSON, Jr. value.

5.6.2   JSON

Revise the program you developed in the previous problem to handle all of JSON.

Design the following methods for JSON values:

- A method for counting the number of numbers in a JSON value.

- A method for summing all the numbers in a JSON value.

---

[2]This definition is drawn directly from `http://json.org/`

- A method for finding the length of the longest array in a JSON value.

- A method for computing the nesting depth of a JSON value.

Design the following methods for JSON objects:

- A method that works on JSON object values that takes a string and produces the JSON value associated with that string in the object, or `false` if no such value exists.

- A method that counts the number of name/value pairs in an object.

- A method that extends an object by adding a given name/value pair to an object.

- A method that restricts an object by subtracting a given name/value pair from an object.

Design the following methods for JSON arrays:

- A method of computing the length of the array.

- A method for indexing the ith element of an array.

- A method for reversing an array.

Design the following functions for randomly generating JSON values:

- Given a nesting depth, compute a random JSON value of at most that nesting depth. (It must be the case that if called repeatedly, eventual this function will produce a JSON value of *exactly* the given nesting depth, but it may not always produce a value nested so deep.)

Design an alternative data representation of JSON values that uses a subset of S-Expressions to model JSON values, which we'll call JSEN (JSON S-Expression Notation).

Design a function for converting from a JSEN representation to the object representation of that value. Design a method for JSON values that produces their JSEN representation.

Part II

Schemes of a Larger Design

Chapter 6

Larger system design: Snakes on a plane

So far, we have introduced a lot of important new concepts such as interfaces and data and method inheritance for class-based abstraction. In this chapter, we are going to see these concepts being applied in the context of a larger program design. It's a game we've all seen and designed before; we're going to develop a class-based version of the Snake Game.

6.1 Information in the Snake Game

Our first task in designing the Snake Game is to take an account of the information that our program will need to represent. This includes:

- a system of coordinates

- a snake, which has

    - direction

    - segments

- food

- a world

6.2 The world

Let's start by designing a minimal `world%` class. We will iteratively refine it later to add more an more features. For now, let's just have the snake move.

```
;; A World is a (new world% Snake Food).
(define-class world%
  (fields snake food)

  (define (on-tick)
    (new world%
         (send (send this snake) move)
         (send this food)))

  (define (tick-rate) 1/8)

  (define (to-draw)
    (send (send this food) draw
          (send (send this snake) draw MT-SCENE))))
```

6.3   Coordinate interface

Let's focus on the system of coordinates. There are really two coordinate systems we will need to represent; one is necessitated by the animation system we are using, the other by the logic of the Snake Game. We are *consumers* of the `big-bang` animation system, which uses a pixel-based, graphics-coordinate system (meaning the origin is at the Northwest corner). This is part of `big-bang`'s interface, which we don't have the power to change, and therefore we have to communicate to `big-bang` using pixels in graphics-coordinates. In general, when we design programs, the interfaces of libraries we use impose obligations on our code.

On the other hand, using pixel-based graphics-coordinates is probably not the best representation choice for the information of the Snake Game. We'll be better off if we design our own representation and have our program translate between representations at the communication boundaries between our code and `big-bang`. Let's use a grid-based coordinate system where the origin is the Southwest corner. We can define a mapping between the coordinate systems by defining some constants such as the grid size and the size of the screen:

```
#lang class/0
(define WIDTH  32) ; in grid units
(define HEIGHT 32) ; in grid units
(define SIZE   16) ; in pixels / grid unit
(define WIDTH-PX  (* SIZE WIDTH))  ; in pixels
(define HEIGHT-PX (* SIZE HEIGHT)) ; in pixels
```

This defines that our game is logically played on a 32x32 grid, which we will render visually as a 512x512 pixel image. This are the values we cooked up in class, which results in the following grid for our game:

As an exercise, try to write an expression that produces this image.

0,31	1,31	2,31	3,31	4,31	5,31	6,31	7,31	8,31	9,31	10,31	11,31	12,31	13,31	14,31	15,31	16,31	17,31	18,31	19,31	20,31	21,31	22,31	23,31	24,31	25,31	26,31	27,31	28,31	29,31	30,
0,30	1,30	2,30	3,30	4,30	5,30	6,30	7,30	8,30	9,30	10,30	11,30	12,30	13,30	14,30	15,30	16,30	17,30	18,30	19,30	20,30	21,30	22,30	23,30	24,30	25,30	26,30	27,30	28,30	29,30	30,
0,29	1,29	2,29	3,29	4,29	5,29	6,29	7,29	8,29	9,29	10,29	11,29	12,29	13,29	14,29	15,29	16,29	17,29	18,29	19,29	20,29	21,29	22,29	23,29	24,29	25,29	26,29	27,29	28,29	29,29	30,
0,28	1,28	2,28	3,28	4,28	5,28	6,28	7,28	8,28	9,28	10,28	11,28	12,28	13,28	14,28	15,28	16,28	17,28	18,28	19,28	20,28	21,28	22,28	23,28	24,28	25,28	26,28	27,28	28,28	29,28	30,
0,27	1,27	2,27	3,27	4,27	5,27	6,27	7,27	8,27	9,27	10,27	11,27	12,27	13,27	14,27	15,27	16,27	17,27	18,27	19,27	20,27	21,27	22,27	23,27	24,27	25,27	26,27	27,27	28,27	29,27	30,
0,26	1,26	2,26	3,26	4,26	5,26	6,26	7,26	8,26	9,26	10,26	11,26	12,26	13,26	14,26	15,26	16,26	17,26	18,26	19,26	20,26	21,26	22,26	23,26	24,26	25,26	26,26	27,26	28,26	29,26	30,
0,25	1,25	2,25	3,25	4,25	5,25	6,25	7,25	8,25	9,25	10,25	11,25	12,25	13,25	14,25	15,25	16,25	17,25	18,25	19,25	20,25	21,25	22,25	23,25	24,25	25,25	26,25	27,25	28,25	29,25	30,
0,24	1,24	2,24	3,24	4,24	5,24	6,24	7,24	8,24	9,24	10,24	11,24	12,24	13,24	14,24	15,24	16,24	17,24	18,24	19,24	20,24	21,24	22,24	23,24	24,24	25,24	26,24	27,24	28,24	29,24	30,
0,23	1,23	2,23	3,23	4,23	5,23	6,23	7,23	8,23	9,23	10,23	11,23	12,23	13,23	14,23	15,23	16,23	17,23	18,23	19,23	20,23	21,23	22,23	23,23	24,23	25,23	26,23	27,23	28,23	29,23	30,
0,22	1,22	2,22	3,22	4,22	5,22	6,22	7,22	8,22	9,22	10,22	11,22	12,22	13,22	14,22	15,22	16,22	17,22	18,22	19,22	20,22	21,22	22,22	23,22	24,22	25,22	26,22	27,22	28,22	29,22	30,
0,21	1,21	2,21	3,21	4,21	5,21	6,21	7,21	8,21	9,21	10,21	11,21	12,21	13,21	14,21	15,21	16,21	17,21	18,21	19,21	20,21	21,21	22,21	23,21	24,21	25,21	26,21	27,21	28,21	29,21	30,
0,20	1,20	2,20	3,20	4,20	5,20	6,20	7,20	8,20	9,20	10,20	11,20	12,20	13,20	14,20	15,20	16,20	17,20	18,20	19,20	20,20	21,20	22,20	23,20	24,20	25,20	26,20	27,20	28,20	29,20	30,
0,19	1,19	2,19	3,19	4,19	5,19	6,19	7,19	8,19	9,19	10,19	11,19	12,19	13,19	14,19	15,19	16,19	17,19	18,19	19,19	20,19	21,19	22,19	23,19	24,19	25,19	26,19	27,19	28,19	29,19	30,
0,18	1,18	2,18	3,18	4,18	5,18	6,18	7,18	8,18	9,18	10,18	11,18	12,18	13,18	14,18	15,18	16,18	17,18	18,18	19,18	20,18	21,18	22,18	23,18	24,18	25,18	26,18	27,18	28,18	29,18	30,
0,17	1,17	2,17	3,17	4,17	5,17	6,17	7,17	8,17	9,17	10,17	11,17	12,17	13,17	14,17	15,17	16,17	17,17	18,17	19,17	20,17	21,17	22,17	23,17	24,17	25,17	26,17	27,17	28,17	29,17	30,
0,16	1,16	2,16	3,16	4,16	5,16	6,16	7,16	8,16	9,16	10,16	11,16	12,16	13,16	14,16	15,16	16,16	17,16	18,16	19,16	20,16	21,16	22,16	23,16	24,16	25,16	26,16	27,16	28,16	29,16	30,
0,15	1,15	2,15	3,15	4,15	5,15	6,15	7,15	8,15	9,15	10,15	11,15	12,15	13,15	14,15	15,15	16,15	17,15	18,15	19,15	20,15	21,15	22,15	23,15	24,15	25,15	26,15	27,15	28,15	29,15	30,
0,14	1,14	2,14	3,14	4,14	5,14	6,14	7,14	8,14	9,14	10,14	11,14	12,14	13,14	14,14	15,14	16,14	17,14	18,14	19,14	20,14	21,14	22,14	23,14	24,14	25,14	26,14	27,14	28,14	29,14	30,
0,13	1,13	2,13	3,13	4,13	5,13	6,13	7,13	8,13	9,13	10,13	11,13	12,13	13,13	14,13	15,13	16,13	17,13	18,13	19,13	20,13	21,13	22,13	23,13	24,13	25,13	26,13	27,13	28,13	29,13	30,
0,12	1,12	2,12	3,12	4,12	5,12	6,12	7,12	8,12	9,12	10,12	11,12	12,12	13,12	14,12	15,12	16,12	17,12	18,12	19,12	20,12	21,12	22,12	23,12	24,12	25,12	26,12	27,12	28,12	29,12	30,
0,11	1,11	2,11	3,11	4,11	5,11	6,11	7,11	8,11	9,11	10,11	11,11	12,11	13,11	14,11	15,11	16,11	17,11	18,11	19,11	20,11	21,11	22,11	23,11	24,11	25,11	26,11	27,11	28,11	29,11	30,
0,10	1,10	2,10	3,10	4,10	5,10	6,10	7,10	8,10	9,10	10,10	11,10	12,10	13,10	14,10	15,10	16,10	17,10	18,10	19,10	20,10	21,10	22,10	23,10	24,10	25,10	26,10	27,10	28,10	29,10	30,
0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9	8,9	9,9	10,9	11,9	12,9	13,9	14,9	15,9	16,9	17,9	18,9	19,9	20,9	21,9	22,9	23,9	24,9	25,9	26,9	27,9	28,9	29,9	30,
0,8	1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8	9,8	10,8	11,8	12,8	13,8	14,8	15,8	16,8	17,8	18,8	19,8	20,8	21,8	22,8	23,8	24,8	25,8	26,8	27,8	28,8	29,8	30,
0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7	9,7	10,7	11,7	12,7	13,7	14,7	15,7	16,7	17,7	18,7	19,7	20,7	21,7	22,7	23,7	24,7	25,7	26,7	27,7	28,7	29,7	30,
0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6	9,6	10,6	11,6	12,6	13,6	14,6	15,6	16,6	17,6	18,6	19,6	20,6	21,6	22,6	23,6	24,6	25,6	26,6	27,6	28,6	29,6	30,
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5	16,5	17,5	18,5	19,5	20,5	21,5	22,5	23,5	24,5	25,5	26,5	27,5	28,5	29,5	30,
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4	10,4	11,4	12,4	13,4	14,4	15,4	16,4	17,4	18,4	19,4	20,4	21,4	22,4	23,4	24,4	25,4	26,4	27,4	28,4	29,4	30,
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3	11,3	12,3	13,3	14,3	15,3	16,3	17,3	18,3	19,3	20,3	21,3	22,3	23,3	24,3	25,3	26,3	27,3	28,3	29,3	30,
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2	11,2	12,2	13,2	14,2	15,2	16,2	17,2	18,2	19,2	20,2	21,2	22,2	23,2	24,2	25,2	26,2	27,2	28,2	29,2	30,
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1	11,1	12,1	13,1	14,1	15,1	16,1	17,1	18,1	19,1	20,1	21,1	22,1	23,1	24,1	25,1	26,1	27,1	28,1	29,1	30,
0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0	12,0	13,0	14,0	15,0	16,0	17,0	18,0	19,0	20,0	21,0	22,0	23,0	24,0	25,0	26,0	27,0	28,0	29,0	30,

But for the sake of our notes, let's develop the game for a much smaller grid that is rendered on a larger scale. It will be easy to change our definitions at the end in order to recover the original design. If done properly, all of test cases will remain unaffected

by the change.

```
(define WIDTH   8) ; in grid units
(define HEIGHT  8) ; in grid units
(define SIZE   32) ; in pixels / grid unit
```

This defines that our game is logically played on a 8x8 grid, which we will render visually as a 256x256 pixel image. The grid for our game now looks like:

0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7
0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1
0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0

Now we need to consider the interface for coordinates (henceforth, the term "coordinate" refers to *our* representation of coordinates in the Snake Game, not the graphics-coordinates of `big-bang`). What do we need to do with coordinates? Here's a coarse first approximation:

- Compare two coordinates for equality.

- Draw something at a coordinate on to a scene.

- Move a coordinate.

- Determine whether a coordinate is on the board.

This list suggest the following interface for coordinates:

```
;; A Coord implements

;; same-pos? : Coord -> Boolean
```

```
;; Is this coordinate at the same position as the given one?

;; draw : Scene -> Scene
;; Draw this coordinate on the scene.

;; move : -> Coord
;; Move this coordinate.

;; on-board? : -> Boolean
;; Is this coordinate on the board?
```

This is a good place to start, but as we start thinking about *what* these methods should do and *how* we might write them, some issues should come to mind. For example, in thinking about the what: what should be drawn when the `draw` method is invoked? Perhaps we want the coordinate "to just know" what should be drawn, which suggests that when we implement coordinates they should contain data representing what to draw. Perhaps we want to tell the `draw` method what to draw, which suggests we should revise the contract to include an argument or arguments that represent what to draw. For the time-being, let's decide that the coordinate will know what to draw.

In the thinking about the how: how do you imagine the draw coordinate will be written? Assuming we know what to draw, the next question is how will the method know where to draw it? We have a coordinate, which is a grid coordinate, but will need to use `place-image` to actually draw that image on the given scene. But `place-image` works in the pixel-based graphics-coordinate system. We need to be able to convert a coordinate to a pixel-based graphics-coordinate in order to write the `draw` method, but there is nothing in the interface that gives us that capability, *and the interface is all we will have to work with*. This suggests we should revise the interface to include this needed behavior.

Similarly, if we consider how to write the `same-pos?` method, we will want to compare the *x*- and *y*-components of the given coordinate with the *x*- and *y*-components of this coordinate. Again, there is nothing in the interface as given that allows this, so we need to revise.

Now consider the `move` method. How can we write it? What do we expect to happen? There's not enough information to know—should the coordinate move up? Down? Right three and down seven? Hard to say. While we might expect a coordinate to know how to draw itself, we cannot expect a coordinate to know which way to move itself. This suggests we need to add inputs to the method that represent this needed information. For the purposes of our game, a position needs to be able to move one grid unit in one of four directions. Let's design the representation of a direction and a directional input to the `move` method.

Interface design is incredibly important, especially when, unlike in our current situation, it is not easy to revise in the future. Modern computer systems are littered with detritus of past interface design choices because interfaces are difficult and expensive, if not impossible, to change. As an example, the developers of the UNIX operating system, which was developed in the 1970s and is now the basis of both Linux and Mac OS, made the choice to save characters and call the operation that creates a file

"CREAT". Forty years later, I'm writing these notes on a portable computer while flying from Boston to Houston. My machine, which weighs far less than any computer that ran UNIX in the 70's, has not one, but two 2.66 GHz processors and 8 gigs of RAM: unimaginable computing resources in the 70s. And yet, when my OS wants to make a new file, it calls the "CREAT" function—not because that missing "E" is a computational extravangance I cannot afford, far from it, but because it is simply too difficult a task to realize a redesign of the interface between my computer and its operating system. The unforunate thing about interfaces is that when they change, all parties that have agreed to that interface must change as well. Too many people, programs, and devices have agreed to the UNIX interface to make changing "CREAT" to "CREATE" worthwhile.

You won't be able to make the perfect interface on the first try, but the closer you get, the better your life will be in the future.

Revising our interface as described above, we arrive at the following:

```
;; A Coord implements:

;; same-pos? : Coord -> Boolean
;; Is this coordinate at the same position as the given one?

;; draw : Scene -> Scene
;; Draw this coordinate on the scene.

;; move : Dir -> Coord
;; Move this coordinate in the given direction.

;; on-board? : -> Boolean
;; Is this coordinate on the board?

;; {x,y} : -> Nat
;; The {x,y}-component of grid-coordinate.

;; {x-px,y-px} : -> Nat
;; The {x,y}-component of pixel-graphics-coordinate.
```

We haven't designed `Dir` data definition for representing direction; let's take care of that quickly. In our game, a direction is one of four possibilities, i.e. it is an enumeration. We could use a class-based enumeration, but for the sake of simplicity, let's just use strings and say that:

```
;; A Dir is one of:
;; - "left"
;; - "right"
;; - "up"
;; - "down"
```

This representation has the nice property of being a subset of `big-bang`'s KeyEvent

representation, so we can rely on the coincidence and handle the "up" key event by moving in the "up" direction without need to convert between representations.

6.4 An implementation of coordinates: segments

At this point, we've flushed out enough of the initial design of the coordinate interface we can now start working on an implementation of it. There are two components that will implement the coordinate interface: segments and food. Let's start with segments.

```
;; A (new seg% Int Int) is a Coord
;; Interp: represents a segment grid-coordinate.
(define-class seg%
  (fields x y)
  ...)
```

Our template for `seg%` methods is:

```
;; ? ... -> ?
(define (seg-template ...)
  (... (send this x) ... (send this y) ...))
```

We've now made a data definition for segments and committed ourselves to implementing the interface. This obligates us to implement all of the methods in `coord%`. We've decided to implement the `coord%` using a class with an `x` and `y` field. This satisfies part of our implementation right off the bat: we get an `x` and `y` method by definition. Let's now do `same-pos?`:

`seg%`

```
(check-expect (send (new seg% 0 0) same-pos? (new seg% 0 0)) true)
(check-expect (send (new seg% 0 0) same-pos? (new seg% 1 0)) false)
(define (same-pos? c)
  (and (= (send this x) (send c x))
       (= (send this y) (send c y))))
```

And now `draw`:

`seg%`

```
(check-expect (send (new seg% 0 0) draw MT-SCENE)
              (place-image (square SIZE "solid" "red")
                           (* 1/2 SIZE)
                           (- HEIGHT-PX (* 1/2 SIZE))
                           MT-SCENE))
(define (draw scn)
  (place-image (square SIZE "solid" "red")
               (send this x-px)
               (send this y-px)
               scn))
```

And now `move`:

seg%

```
(check-expect (send (new seg% 0 0) move "up")    (new seg%  0  1))
(check-expect (send (new seg% 0 0) move "down")  (new seg%  0 -
1))
(check-expect (send (new seg% 0 0) move "left")  (new seg% -1  0))
(check-expect (send (new seg% 0 0) move "right") (new seg%  1  0))
(define (move d)
  (cond [(string=? d "up")
         (new seg% (send this x) (add1 (send this y)))]
        [(string=? d "down")
         (new seg% (send this x) (sub1 (send this y)))]
        [(string=? d "left")
         (new seg% (sub1 (send this x)) (send this y))]
        [(string=? d "right")
         (new seg% (add1 (send this x)) (send this y))]))
```

And now `on-board?`:

seg%

```
(check-expect (send (new seg% 0  0) on-board?) true)
(check-expect (send (new seg% 0 -1) on-board?) false)
(check-expect (send (new seg% 0 (sub1 HEIGHT)) on-board?) true)
(check-expect (send (new seg% 0 HEIGHT) on-board?) false)
(define (on-board?)
  (and (<= 0 (send this x) (sub1 WIDTH))
       (<= 0 (send this y) (sub1 HEIGHT))))
```

And finally, the `x-px` and `y-px` methods:

seg%

```
(check-expect (send (new seg% 0 0) x-px) (* 1/2 SIZE))
(check-expect (send (new seg% 0 0) y-px) (- HEIGHT-PX (* 1/2 SIZE)))
(define (x-px)
  (* (+ 1/2 (send this x)) SIZE))
(define (y-px)
  (- HEIGHT-PX (* (+ 1/2 (send this y)) SIZE)))
```

That completes all of the obligations of the `seg%` interface.

6.5   Another implementation of coordinates: food

Food is another implementation of the `coord<%>` interface, and it is largely similar to the `seg%` class, which suggests that `seg%` and `food%` may be good candidates for abstraction, but that's something to worry about later. For now, let's implement `food%`. Since we've already been through the design of `seg%`, we'll do `food%` quickly:

```
;; A Food is a (new food% Nat Nat) is a Coord
(define-class food%
  (fields x y)

  (define (same-pos? c)
    (and (= (send this x) (send c x))
         (= (send this y) (send c y))))

  (define (draw scn)
    (place-image (square SIZE "solid" "green")
                 (send this x-px)
                 (send this y-px)
                 scn))

  (define (move d)
    (cond [(string=? d "up")
           (new food% (send this x) (add1 (send this y)))]
          [(string=? d "down")
           (new food% (send this x) (sub1 (send this y)))]
          [(string=? d "left")
           (new food% (sub1 (send this x)) (send this y))]
          [(string=? d "right")
           (new food% (add1 (send this x)) (send this y))]))

  (define (on-board?)
    (and (<= 0 (send this x) (sub1 WIDTH))
         (<= 0 (send this y) (sub1 HEIGHT))))

  (define (x-px)
    (* (+ 1/2 (send this x)) SIZE))
  (define (y-px)
    (- HEIGHT-PX (* (+ 1/2 (send this y)) SIZE))))
```

You'll notice that this class definition is nearly identical to the definition of `seg%`. The key differences are in `move` and `draw`. We'll hold off on abstracting for now.

6.6 Representing the snake

What information needs to be represented in a snake?

- Direction

- Segments

What are the operations we need to perform on snakes?

```
;; A Snake implements:

;; move : -> Snake
;; Move this snake in its current direction.

;; grow : -> Snake
;; Grow this snake in its current direction.

;; turn : Dir -> Snake
;; Turn this snake in the given direction.

;; draw : Scene -> Scene
;; Draw this snake on the scene.
```

Here's a possible data definition:

```
;; A (new snake% Dir [Listof Seg]) is a Snake
(define-class snake%
  (fields dir segs))
```

But after a moment of reflection, you will notice that a snake with no segments doesn't make sense—a snake should always have at least one segment. Moreover, we need to settle on an interpretion of the order of the list; either the front of the list is interpreted as the front of the snake or the rear of the list is interpreted as the front of the snake. Together, non-emptiness and order let us determine which element is the head of the snake.

Here's our revised data definition:

```
;; A (new snake% Dir (cons Seg [Listof Seg])) is a Snake
(define-class snake%
  (fields dir segs)
  ...)
```

An alternative data definition that might be worth considering is:

```
;; A Snake is a (new snake% Dir Seg [Listof Seg])
(define-class snake%
  (fields dir head segs))
```

But for the time being let's stick with the former one.
Now let's implement the interface. Here's the template:

```
;; ? ... -> ?
(define (snake-template ...)
  (send this dir) ... (send this segs) ...)
```

The `move` method works by moving the head of the snake and dropping the last element of the list of segments:

```
(check-expect (send (new snake% "right" (list (new seg% 0 0))) move)
              (new snake% "right" (list (new seg% 1 0))))
```

snake%

```
(define (move)
  (new snake%
       (send this dir)
       (cons (send (first (send this segs)) move (send this dir))
             (all-but-last (send this segs)))))
```

This relies on a helper function, `all-but-last`, which is straightforward to write (recall that `segs` is a non-empty list):

```
(check-expect (all-but-last (list "x")) empty)
(check-expect (all-but-last (list "y" "x")) (list "y"))

;; (cons X [Listof X]) -> [Listof X]
;; Drop the last element of the given list.
(define (all-but-last ls)
  (cond [(empty? (rest ls)) empty]
        [else (cons (first ls)
                    (all-but-last (rest ls)))]))
```

The `grow` method is much like `move`, except that no element is dropped from the segments list:

snake%

```
(check-expect (send (new snake% "right" (list (new seg% 0 0))) grow)
              (new snake% "right" (list (new seg% 1 0)
                                        (new seg% 0 0))))
(define (grow)
  (new snake%
       (send this dir)
       (cons (send (first (send this segs)) move (send this dir))
             (send this segs))))
```

Now let's write the `turn` method:

snake%

```
(check-expect (send (new snake% "left" (list (new seg% 0 0))) turn "up")
              (new snake% "up" (list (new seg% 0 0))))
(define (turn d)
  (new snake% d (send this segs)))
```

And finally, `draw`:

snake%

```
(check-expect (send (new snake% "left" (list (new seg% 0 0))) draw MT-
SCENE)
              (send (new seg% 0 0) draw MT-SCENE))
(define (draw scn)
  (foldl (λ (s scn) (send s draw scn))
         scn
         (send this segs)))
```

As this method shows, functions and methods can co-exist nicely in a single language.

6.7   Seeing the world

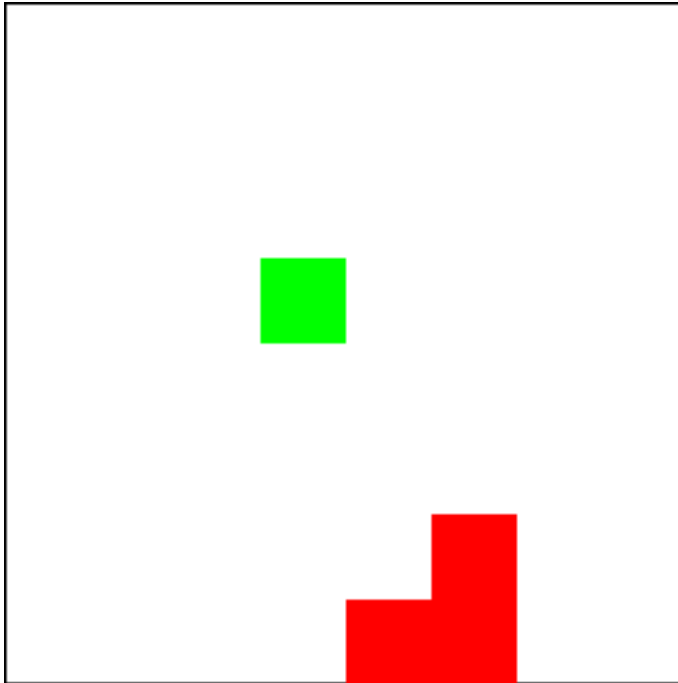At this point we have a working but incomplete system and we can interact with it in the interactions window:
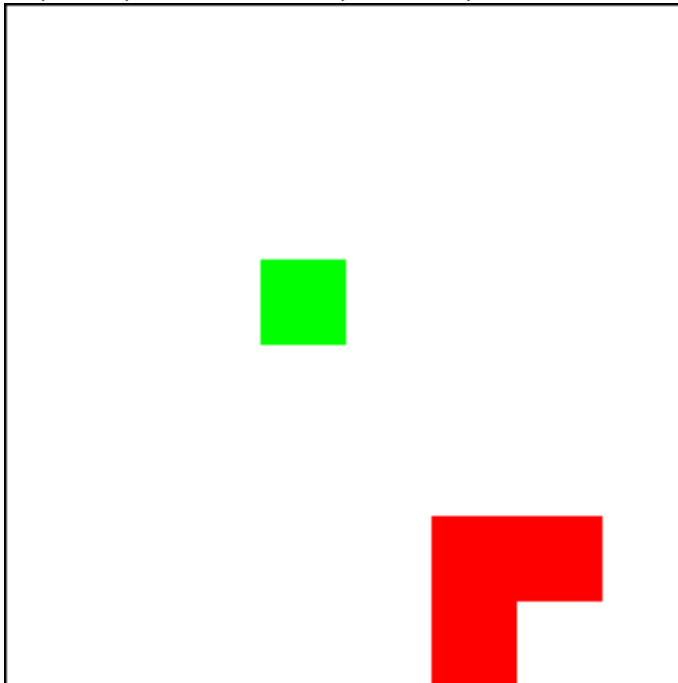
Examples:

```
> (define w0 (new world%
                  (new snake%
                       "right"
                       (list (new seg% 5 1)
                             (new seg% 5 0)
                             (new seg% 4 0)))
                  (new food% 3 4)))

> (send w0 to-draw)
```
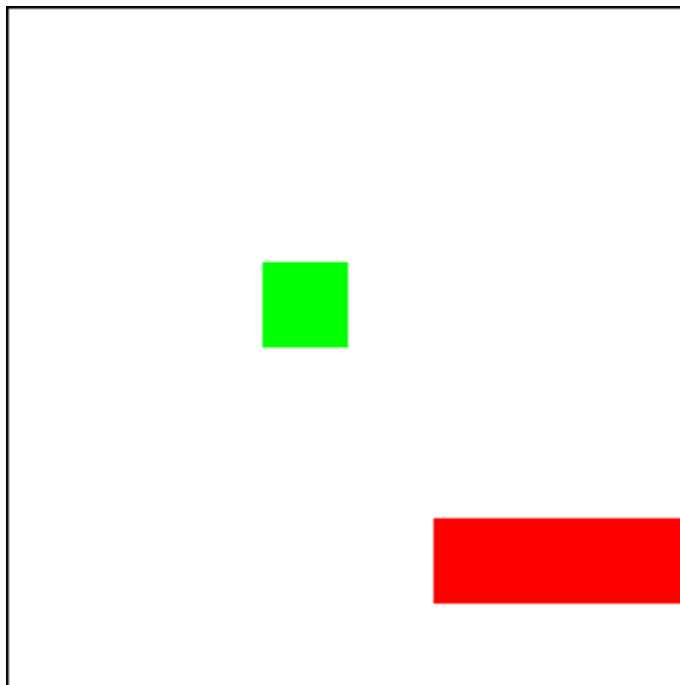
```
> (send (send w0 on-tick) to-draw)
```



```
> (send (send (send w0 on-tick) on-tick) to-draw)
```

We'll leave it at this point and further the refine the program in the future.

6.8   The whole ball of wax

```
#lang class/0
(require 2htdp/image)
(require class/universe)

(define WIDTH   8) ; in grid units
(define HEIGHT  8) ; in grid units
(define SIZE   32) ; in pixels / grid unit
(define WIDTH-PX  (* SIZE WIDTH))  ; in pixels
(define HEIGHT-PX (* SIZE HEIGHT)) ; in pixels
(define MT-SCENE (empty-scene WIDTH-PX HEIGHT-PX))

;; A World is a (new world% Snake Food).
(define-class world%
  (fields snake food)

  (define (on-tick)
    (new world%
         (send (send this snake) move)
```

```
          (send this food)))

  (define (tick-rate) 1/8)

  (define (to-draw)
    (send (send this food) draw
          (send (send this snake) draw MT-SCENE)))))

;; A Coord implements:

;; same-pos? : Coord -> Boolean
;; Is this coordinate at the same position as the given one?

;; draw : Scene -> Scene
;; Draw this coordinate on the scene.

;; move : Dir -> Coord
;; Move this coordinate in the given direction.

;; on-board? : -> Boolean
;; Is this coordinate on the board?

;; {x,y} : -> Nat
;; The {x,y}-component of grid-coordinate.

;; {x-px,y-px} : -> Nat
;; The {x,y}-component of pixel-graphics-coordinate.

;; A Dir is one of:
;; - "left"
;; - "right"
;; - "up"
;; - "down"

;; A (new seg% Int Int) is a Coord
;; Interp: represents a segment grid-coordinate.
(define-class seg%
  (fields x y)
  (check-expect (send origin same-pos? (new seg% 0 0)) true)
  (check-expect (send origin same-pos? (new seg% 1 0)) false)
  (define (same-pos? c)
    (and (= (send this x) (send c x))
         (= (send this y) (send c y))))
  (define (draw scn)
    (place-image (square SIZE "solid" "red")
                 (send this x-px)
```

```
                    (send this y-px)
                    scn))
  (define (move d)
    (cond [(string=? d "up")
           (new seg% (send this x) (add1 (send this y)))]
          [(string=? d "down")
           (new seg% (send this x) (sub1 (send this y)))]
          [(string=? d "left")
           (new seg% (sub1 (send this x)) (send this y))]
          [(string=? d "right")
           (new seg% (add1 (send this x)) (send this y))]))
  (define (on-board?)
    (and (<= 0 (send this x) (sub1 WIDTH))
         (<= 0 (send this y) (sub1 HEIGHT))))
  (define (x-px)
    (* (+ 1/2 (send this x)) SIZE))
  (define (y-px)
    (- HEIGHT-PX (* (+ 1/2 (send this y)) SIZE))))

(check-expect (send (new seg% 0 0) draw MT-SCENE)
              (place-image (square SIZE "solid" "red")
                           (* 1/2 SIZE)
                           (- HEIGHT-PX (* 1/2 SIZE))
                           MT-SCENE))
(check-expect (send (new seg% 0 0) move "up")    (new seg%  0  1))
(check-expect (send (new seg% 0 0) move "down")  (new seg%  0 -
1))
(check-expect (send (new seg% 0 0) move "left")  (new seg% -1  0))
(check-expect (send (new seg% 0 0) move "right") (new seg%  1  0))
(check-expect (send (new seg% 0  0) on-board?) true)
(check-expect (send (new seg% 0 -1) on-board?) false)
(check-expect (send (new seg% 0 (sub1 HEIGHT)) on-board?) true)
(check-expect (send (new seg% 0 HEIGHT) on-board?) false)
(check-expect (send (new seg% 0 0) x-px) (* 1/2 SIZE))
(check-expect (send (new seg% 0 0) y-px) (- HEIGHT-PX (* 1/2 SIZE)))

;; A Food is a (new food% Nat Nat) is a Coord
(define-class food%
  (fields x y)

  (define (same-pos? c)
    (and (= (send this x) (send c x))
         (= (send this y) (send c y))))

  (define (draw scn)
    (place-image (square SIZE "solid" "green")
```

```
                    (send this x-px)
                    (send this y-px)
                    scn))

  (define (move d)
    (cond [(string=? d "up")
            (new food% (send this x) (add1 (send this y)))]
          [(string=? d "down")
            (new food% (send this x) (sub1 (send this y)))]
          [(string=? d "left")
            (new food% (sub1 (send this x)) (send this y))]
          [(string=? d "right")
            (new food% (add1 (send this x)) (send this y))]))

  (define (on-board?)
    (and (<= 0 (send this x) (sub1 WIDTH))
         (<= 0 (send this y) (sub1 HEIGHT))))

  (define (x-px)
    (* (+ 1/2 (send this x)) SIZE))
  (define (y-px)
    (- HEIGHT-PX (* (+ 1/2 (send this y)) SIZE))))

;; A Snake implements:

;; move : -> Snake
;; Move this snake in its current direction.

;; grow : -> Snake
;; Grow this snake in its current direction.

;; turn : Dir -> Snake
;; Turn this snake in the given direction.

;; draw : Scene -> Scene
;; Draw this snake on the scene.

;; A (new snake% Dir Seg [Listof Seg]) is a Snake
(define-class snake%
  (fields dir segs)
  (define (move)
    (new snake%
         (send this dir)
         (cons (send (first (send this segs)) move (send this dir))
               (all-but-last (send this segs)))))
```

```
  (define (grow)
    (new snake%
         (send this dir)
         (cons (send (first (send this segs)) move (send this dir))
               (send this segs))))

  (define (turn d)
    (new snake% d (send this segs)))

  (define (draw scn)
    (foldl (λ (s scn) (send s draw scn))
           scn
           (send this segs))))

(define origin (new seg% 0 0))
(check-expect (send (new snake% "right" (list (new seg% 0 0))) move)
              (new snake% "right" (list (new seg% 1 0))))
(check-expect (send (new snake% "right" (list (new seg% 0 0))) grow)
              (new snake% "right" (list (new seg% 1 0)
                                        (new seg% 0 0))))
(check-expect (send (new snake% "left" (list (new seg% 0 0))) turn "up")
              (new snake% "up" (list (new seg% 0 0))))
(check-expect (send (new snake% "left" (list (new seg% 0 0))) draw MT-
SCENE)
              (send (new seg% 0 0) draw MT-SCENE))

(check-expect (all-but-last (list "x")) empty)
(check-expect (all-but-last (list "y" "x")) (list "y"))

;; (cons X [Listof X]) -> [Listof X]
;; Drop the last element of the given list.
(define (all-but-last ls)
  (cond [(empty? (rest ls)) empty]
        [else (cons (first ls)
                    (all-but-last (rest ls)))]))

(big-bang (new world%
               (new snake%
                    "right"
                    (list (new seg% 5 1)
                          (new seg% 5 0)
                          (new seg% 4 0)))
               (new food% 3 4)))
```

6.9 Exercises

6.9.1 Different representation of Snakes

Consider the alternative data definition suggested for Snakes:

```
; A Snake is a (new snake% Dir Seg [Listof Seg])
(define-class snake%
  (fields dir head segs))
```

Revise the Snake Game to use this definition and carry out all the changes it implies.

6.9.2 Zombie!

Design and develop an interactive game called *Zombie!*. In this game, there are a number of zombies that are coming to eat your brains. The object is simple: stay alive. You can maneuver by moving the mouse. The player you control always moves toward the mouse position. The zombies, on the other hand, always move toward you. If the zombies ever come in contact with you, they eat your brains, and you die. If two zombies happen to come in to contact with each other, they will mistakenly eat each other's brain, which it turns out is fatal to the zombie species, and so they both die. When a zombie dies, the zombie flesh will permanently remain where it is and should any subsequent zombie touch the dead flesh, they will try to eat it and therefore die on the spot. Survive longer than all the zombies, and you have won the game.

(This game is based on the *Attack of the Robots!* game described in Land of Lisp. Unlike the Land of Lisp version, this game is graphical and interactive rather than text-based. Hence, our game doesn't suck.)

Once you have a working version of the game, add the following feature: whenever the user does a mouse-click, the player should be instantly teleported to a *random* location on the screen.

6.9.3 Primum non copy-and-paste

A natural design for the Zombie game is to have a `Zombie` and `Player` class of data. But you may find your first iteration of the Zombie game duplicates a lot of code between these classes. In fact, the `Zombie` and `Player` classes have more in common than apart. It may even be tempting to pursue an unnatural design in which there is only a single class of data, which must consist of an additional *bit*, which is interpreted as signifying "zombieness" versus "playerness." Down that path waits shame, defeat, and a brittle design that makes babies cry.

To recoil at the prospect of copy-and-paste is commendable, but we shouldn't throw the crying babies out with the bathwater. Let's step back and ask ourselves if this dilemma is really inescapable.

First, let's consider the information that needs to be represented in a game. When you look at the game, you see several things: live zombies, dead zombies, a player, and a mouse. That might lead you to a representation of each of these things as a separate class, in which case you may later find many of the methods in these classes

are largely identical. You would like to abstract to avoid the code duplication, but thus far, we haven't seen any class-based abstraction mechanisms. So there are at least two solutions to this problem:

1. Re-consider your data definitions.

   Program design is an iterative process; you are continually revising your data definitions, which induces program changes, which may cause you to redesign your data definitions, and so on. So when you find yourself wanting to copy and paste lots of code, you might want to reconsider how you're representing information in your program. In the case of zombie, you might step back and see that although the game consists of a player, dead zombies, live zombies, and a mouse, these things have much in common. What *changes over time* about each of them is their position. Otherwise, what makes them different is how they are rendered visually. But it's important to note that way any of these things are rendered *does not change over the course of the game*—a dead zombie is *always* drawn as a gray dot; a live zombie is *always* drawn as a green dot; etc. Taking this view, we can represent the position of each entity uniformly using a single class. This avoids duplicating method definitions since there is only a single class to represent each of these entities.

2. Abstract using the functional abstraction recipe of last semester.

   Just because we are in a new semester and studying a new paradigm of programming, we should not throw out the lessons and techniques we've previously learned. In particular, since we are writing programs in a multi-pararadigm language—one that accomodates both structural and functional programming *and* object-oriented programming—we can mix and match as our designs necessitate. In this case, we can apply the recipe for *functional abstraction* to the design of identical or similar methods, i.e. two methods with similar implementations can be abstracted to a single point of control by writing a helper function that encapsulates the common code. The method can then call the helper function, supplying as arguments values that encapsulate the differences of the original methods.

   Revise your Zombie! program.

6.9.4   Space Invaders!

For this exercise, you will design and develop (a pared down version of) the classic game of *Space Invaders*. In this game, there are a number of space aliens that are descending from the top of the screen. They move left to right and then down at uniform speed. The player controls a laser canon that can be moved left or right along the bottom of the screen. The player can fire the laser, which shoots straight up. If the laser hits an alien, the alien dies. If any alien makes it to the bottom of the screen (or hits the cannon), the player loses. If the player destroys all the invaders, the player wins.

To get a sense of the game, you can play this online version of the game. Your version doesn't need to have all of the features of the online game; in particular, you

don't need levels, different kinds of aliens, protective bunkers, shooting aliens, or the mysterious red alien that flies across the top of the screen. You don't need to keep score and the player only needs to have one life. Of course, *you can* implement all of these nice features, but remember, you're graded for your program design, not making a cool video game. So whatever you add, make sure it's well designed.

Chapter 7

Universe

In this chapter, we're going to start looking at the design of multiple, concurrently running programs that communicate we each other. We will use the universe system as our library for communicating programs.

7.1 A look at the Universe API

The basic universe concept is that there is a "universe" program that is the adminstrator of a set of world programs. The universe and the world programs can communicate with each other by sending messages, which are represented as S-Expressions.

So far we have focused on the design of single programs; we are now going to start looking at the design of communicating systems of programs.

In addition to these notes, be sure to read the documentation on section ???.

7.2 Messages

A message is represented as an S-Expression. Here is there is their data definition:

An S-expression is roughly a nested list of basic data; to be precise an S-expression is one of:

- a string,

- a symbol,

- a number,

- a boolean,

- a char, or

- a list of S-expressions.

97

The way that a world program sends a message to the universe is by constructing a package:

```
;; A Package is a (make-package World SExp).
```

The world component is the new world just like the event handler's produced for single world programs. The s-expression component is a message that is sent to the universe.

7.3   Simple world

As a simple example, let's look at a world program that counts up and sends messages to a universe server as it counts. In this simple example there is only one world that communicates with the server, and the server does nothing but receive the count message (it sends no messages back to the world).

Let's start with the counting world program, which does not communicate with any server, it just counts:

```
#lang class/0
(require class/universe)
(require 2htdp/image)

;; Scene is 300x100 pixels
(define WIDTH 300)
(define HEIGHT 100)

;; A CounterWorld is a (new cw% Natural)
;; and implements
;; - tick-rate : -> Number
;;    Tick rate for counting.
;; - on-tick : -> CounterWorld
;;    Increment counter world state.
;; - to-draw : -> Scene
;;    View counter world state as a scene.
(define-class cw%
  (fields n)
  (define (tick-rate) 1)
  (define (on-tick)
    (new cw% (add1 (send this n))))
  (define (to-draw)
    (overlay (text (number->string (send this n)) 40 "red")
             (empty-scene WIDTH HEIGHT))))

;; Run, program, run!
(big-bang (new cw% 0))
```

When you run this program, you see the world counting up from zero.

7.4 Simple world, broadcasting to server

Now let's modify our program so that it does some simple communication with a server. As an initial design, we'll make a program that simply notifies a server as it counts. In other words, our program will only engage in one-way communication by broadcasting data to the server.

So what exactly do we want to broadcast? If we want to communicate the current state of the world, we may be tempted to try to communicate the current `Counter-World` value; but remember that the current world state is an *object*, which isn't included in the data definition of messages. Moreover, the current state includes not only data, but functionality: functionality for handling tick events, viewing the current count as an image, etc.—all of which are things the server doesn't really need. To communicate the essence of what state the `CounterWorld` is in, all we really need to send is the current count: a number, which fortunately does fall under the set of message values.

We can now revise the `on-tick` method to not only produce the new state of the world, but additionally a message to be sent to the server:

`cw%`

```
(define (on-tick)
  (make-package (new cw% (add1 (send this n)))
                (add1 (send this n))))
```

Thinking about the message protocol for this simple scenario, the client and server communications will look like this:

```
          Client                      Server
-------------------------             ---------

Event     State     Message

Bang    (new cw% 0)           =========>
Tick    (new cw% 1)    1      --------->
Tick    (new cw% 2)    2      --------->
Tick    (new cw% 3)    3      --------->
...         ...       ...         ...
```

Here, the `==>` arrow indicates the world registering with the server. This is the one time communication that establishes a dialogue (really, in this case, a monologue) between the client and server. After registering with the server, the client will send its current count, indicated wtih `->` arrows, as it ticks along.

In this table we show the state of the client and the events that occur. With each event, which potentially changes the state, we show what message is sent to the server. At this point the server is opaque—we don't know or really care what state the server is in and we assume that the server doesn't send any message back to the client.

Now to register this program with a universe server, we need to implement a `register` method that produces a string that is the IP address of the server. (Since we're going to run the universe and world on the same computer, we will use `LOCALHOST` which is bound to the address of our computer.)

`cw%`

```
;; register : -> String
;; IP address of server
(define (register) LOCALHOST)
```

Now when you run this program you will see the world program try to connect to the universe, but since we have not written—much less run—the server, it cannot find the universe. After a few tries, it gives up and continues running without communicating with the universe.

Our complete client is:

```
#lang class/0
(require class/universe)
(require 2htdp/image)

;; Scene is 300x100 pixels
(define WIDTH 300)
(define HEIGHT 100)

;; A CounterWorld is a (new cw% Natural)
;; and implements
;; - register : -> String
;;    IP address of server
;; - tick-rate : -> Number
;;    Tick rate for counting.
;; - on-tick : -> (make-package CounterWorld Number)
;;    Increment counter world state, broadcast to server.
;; - to-draw : -> Scene
;;    View counter world state as a scene.
(define-class cw%
  (fields n)
  (define (register) LOCALHOST)
  (define (tick-rate) 1)
  (define (on-tick)
    (make-package (new cw% (add1 (send this n)))
                  (add1 (send this n))))
  (define (to-draw)
    (overlay (text (number->string (send this n)) 40 "red")
             (empty-scene WIDTH HEIGHT))))

;; Run, program, run!
(big-bang (new cw% 0))
```

7.5   Simple universe, receiving broadcasts

Now let's write a simple server that receives the message from the counter world client. We could write the server in the same file as the client, but since these are really two

separate programs that talk to each other, let's emphasize that by writing the server in its own tab.

A universe program is similar to a world program: it's a program that responds to events. The difference is in the kind of events that can occur. The most important events are already shown in our protocol diagram:

- a new world starts communicating with the server,

- a world sends a message.

When these events occur, the server reacts by calling the appropriate method, in this case `on-new` and `on-msg`.

We'll be working with the OO-style universe, but you should read the documentation for `2htdp/universe` and translate over the concepts to our setting as you've done for `big-bang`.

As it turns out, if you leave these methods off, the universe library will do something sensible, namely nothing. So for our simple counter program, the following works:

```
#lang class/0
(require class/universe)
(define-class cu%)


;; Run, server, run!
(universe (new cu%))
```

This is equivalent to doing the following:

```
#lang class/0
(require class/universe)

(define-class cu%
  ;; IWorld -> Bundle
  (define (on-new iw) this)

  ;; IWorld S-Expr -> Bundle
  (define (on-msg iw m) this))
```

When a world registers, the `on-new` method is called with an `IWorld` value. An `IWorld` value opaquely represents a world, that is you do not have the ability to examine the contents of the value, but you can compare it for equality with other `IWorld` values using `iworld=?`.

When a world sends message, the `on-msg` method is called with the IWorld representing the world that sent the message and the S-Exp message that was sent.

In both cases, the method must either produce a universe or a bundle:

```
;; A Bundle is a (make-bundle Universe [Listof Mail] [Listof IWorld]).
;; A Mail is a (make-mail IWorld S-Exp).
```

A bundle signals communication to some set of worlds. Within a bundle, the universe component is the new state of the universe; the list of mail is a list of messages that will be sent back to the worlds (more on this in a moment), and the list of worlds are worlds that the server has chosen to disconnect from.

For the purposes of our example, the universe maintains no state (the class has no data). When a new world registers, we do nothing, and when a world sends a message, we also do nothing, send nothing in response, and disconnect no worlds.

Running this program launches the universe server, making it ready to receive registrations from worlds. After starting the universe server, if we switch back to the world program tab and run it, we'll see that it successfully registers with the universe and the universe console reports that the world signed up with it.

If we examine the server side of the diagram considered above, we see:

```
 Client                          Server
--------                 -----------------------
                         Message   Event    State
                                   Bang     (new cu%)
          =========>              Join     (new cu%)
          --------->  1            OnMsg    (new cu%)
          --------->  2            OnMsg    (new cu%)
          --------->  3            OnMsg    (new cu%)
                         ...        ...      ...
```

From the perspective of the server, the client is opaque—all we can observe is the messages sent from the client.

The server, as written, will actually work if more than one client connect to the server. To try it out, just run two clients at the same time. When multiple clients connect, we may see interaction like this:

```
 A       B                       Server
-----   -----            -----------------------
                         Message   Event    State
                                   Bang     (new cu%)
          =========>              Join     (new cu%)
          --------->  1            OnMsg    (new cu%)
          --------->  2            OnMsg    (new cu%)
 ================>              Join     (new cu%)
 ---------------->  1            OnMsg    (new cu%)
 ---------------->  2            OnMsg    (new cu%)
          --------->  3            OnMsg    (new cu%)
 ---------------->  3            OnMsg    (new cu%)
                         ...        ...      ...
```

If we would like to rule out this kind of interaction and have the server listen only to one client, that's easy to do. The idea is that we can develop a server that accepts no new clients. Such a server would have the following `on-new` method:

```
(define (on-new iw)
  (make-bundle this empty (list iw)))
```

Now we can have the server start in the accepting universe state that will allow a

client to join, but as soon as one does, it transitions to the unaccepting state that rejects all new clients:

```
#lang class/0
(require class/universe)

;; A Server is one of:
;; - (new accept%)
;; - (new reject%)

(define-class accept%
  (define (on-new iw)
    (new reject%)))

(define-class reject%
  (define (on-new iw)
    (make-bundle this empty (list iw))))

;; Run, server, run!
(universe (new accept%))
```

The interaction described above would now look like:

```
A         B                    Server
-----     -----          -----------------------
                         Message   Event   State
                                   Bang    (new accept%)
          ========>                Join    (new reject%)
          --------->  1            OnMsg   (new reject%)
          --------->  2            OnMsg   (new reject%)
===============>                   Join    (new reject%)
<===============                   Disconn (new reject%)
          --------->  3            OnMsg   (new reject%)
                         ...       ...     ...
```

We can imagine more sophisticated scenerios such as one where we want to enable peer-to-peer communication. As a simple example, let's build a system where one client can broadcast messages to another. This server will have to involve some data since it needs to remember who to broadcast mail to when it receives a message. The initial state of the server is waiting for the broadcaster to join, after which it waits for the broadcastee. Once both parties have joined, every time the broadcaster sends a message, the server relays it to the broadcastee:

```
#lang class/0
(require class/universe)

;; A Server is one of:
;; - (new wait-caster%)
;; - (new wait-castee%)
```

```
;; - (new relay% IWorld)

(define-class wait-caster%
  (define (on-new iw)
    (new wait-castee%)))

(define-class wait-castee%
  (define (on-new iw)
    (new relay% iw)))

(define-class relay%
  (fields castee)
  (define (on-new iw)
    (make-bundle this empty (list iw)))
  (define (on-msg iw msg)
    (make-bundle this (make-mail (send this castee) msg) empty)))

;; Run, server, run!
(universe (new wait-caster%))
```

Notice how the `relay%` class contains an `IWorld` that it broadcasts to when it receives a message. It also rejects any new clients that try to connect.

An example interaction for this serve is:

```
A         B                   Server
-----     -----               ----------------------
                              Message  Event   State
                                       Bang    (new wait-caster%)
          ========>                    Join    (new wait-castee%)
          --------->  1                OnMsg   (new wait-castee%)
          --------->  2                OnMsg   (new wait-castee%)
================>                      Join    (new relay% A)
          --------->  3                OnMsg   (new relay% A)
<---------------  3
          --------->  4                OnMsg   (new relay% A)
<---------------  4
                              ...      ...     ...
```

Notice how in this example, B, the broadcaster, starts sending messages before another client has joined. Those messages are simply ignored. After a second client joins and the server goes into the relay state, subsequent messages will be sent to A, the broadcastee.

7.6   Simple world, receiving messages from the server

This should be re-written to use the state pattern.

In the example considered so far, the server sends messages to a client, but that requires the client is willing to receive such messages. Let's look at how to adapt our simple

broadcasting client to one that listens for message from the universe server.

This client is in one of two states: it either hasn't received a message yet, so it doesn't know what the current "count" is yet, or it has received at least one message and so it knows the most recent count, relayed from the counting to world to the server to this client. The listener client doesn't need to generate any of its own events so we drop the `on-tick` handler and instead add a `on-receive` method that handles incoming messages from the server.

```
#lang class/0
(require class/universe)
(require 2htdp/image)

;; Scene is 300x100 pixels
(define WIDTH 300)
(define HEIGHT 100)

;; A ListenerWorld is one of:
;; - (new wait-first%)          Interp: waiting for first msg.
;; - (new wait-next% Number)    Interp: waiting for next msg.
;; and implements
;; - to-draw : -> Scene
;;   View counter world state as a scene.
;; - on-receive : Number -> ListenerWorld
;;   Receive new counter state from the universe.

(define-class wait-first%
  (define (register) LOCALHOST)
  (define (on-receive msg) (new wait-next% msg))
  (define (to-draw)
    (overlay (text "Waiting" 40 "red")
             (empty-scene WIDTH HEIGHT))))

(define-class wait-next%
  (fields n)
  (define (on-receive msg) (new wait-next% msg))
  (define (to-draw)
    (overlay (text (number->string (send this n)) 40 "red")
             (empty-scene WIDTH HEIGHT))))

;; Run, program, run!
(big-bang (new wait-first%))
```

We can now run the clients and server and observe the communication, but to do so, you have to be careful about the order in which you start things. Our server *assumes* the first client to connect is the broadcaster. This is really a flaw in our design, and we can look at eliminating it, but as a work around just be sure to first start the server, then the counter world, and then the listener world.

7.7   Rules of engagement: protocols and enforcement

There are a couple subtle points worth noting about the server and the possible interactions it allows.

First, if the broadcaster and broadcastee don't join in that order, nothing works as intended. While it may be reasonable to think we can control the order for simple experiments, when our programs are released into the wild and we have *no control* over when and where the clients run, this isn't a reasonable assumption to make. We should refine our program to allow the clients to join in any order.

If the clients can join in any order, a client must identify their role by sending a message after joining. But now the protocol, even for this simple scenario, becomes fairly complicated. You can imagine a client joins but waits a long time to identify their role. Another client may join in the interim and immediately identify their role. What if two clients want to broadcast? What if two clients want to listen?

Second, if the broadcastee ever decides to send a message, rather than just passively listening to what's sent its way, we will treat this message as though it came from the broadcaster (and thus send it back to the broadcastee). That is, the following scenario is possible:

```
  A        B                   Server
-----    -----          ----------------------
                        Message   Event   State
                                  Bang    (new wait-caster%)
         ========>                Join    (new wait-castee%)
 ===============>                 Join    (new relay% A)
         -------->  1             Msg     (new relay% A)
 <---------------  1
         -------->  2             Msg     (new relay% A)
 <---------------  2
 --------------->  7             Msg     (new relay% A)
 <---------------  7

                        ...       ...     ...
```

This brings up the interesting issue of *protocols*, that is, what are the proper rules of conduct for the clients' and server's interaction. First and foremost, it's important to establish and document the rules of interaction. After doing so, we may want to *enforce* protocols, either on the client or server side, or both.

Let's try to develop broadcast and listener clients and a server that address these two issues. In this design, the clients' changes are relatively straightforward. They start by sending an initial message identifying their role, either `'broadcast` or `'listen`.

The server is more involved. Here we model *connections* with worlds as either being an unknown connection, a broadcast connection, or a broadcastee connection. The server starts in a state with no connections. When a world joins, it transitions to a single, unknown connection state. When a second world joins, it transitions to a two-connection state. Any time after a connection has been made the server accepts role messages and sends them to the connections. If the connection is in an unknown state and the role came from that connections' underlying world, the connection is assigned the requested role. The only issue remaining with this server is handling the case of

two clients want to both broadcast or both listen, which we just ignore for now.

```
#lang class/0
;; A Server is
;; - (new wait%)
;; - (new one% Conn)
;; - (new two% Conn Conn)

(define-class wait%
  (define (on-new iw)
    (new one% (new unknown% iw))))

(define-class one%
  (fields conn)
  (define (on-new iw)
    (new two% (send this conn) (new unknown% iw)))
  (define (on-msg iw msg)
    (cond [(role? msg)
           (new one% (send (send this conn) ident msg))]
          [else this])))

(define-class two%
  (fields conn1 conn2)
  (define (on-msg iw msg)
    (cond [(role? msg)
           (new two%
                (send (send this conn1) ident msg)
                (send (send this conn2) ident msg))]
          [else
           (make-bundle this
                        (append (send (send this conn1) mail msg)
                                (send (send this conn2) mail msg))
                        empty)])))

;; A Conn is one of:
;; - (new unknown% IWorld)
;; - (new caster% IWorld)
;; - (new castee% IWorld)
;; implements
;; - mail : Msg -> [Maybe Mail]
;;   Maybe send mail to this connection.
;; - ident : IWorld Role -> Conn
;;   Assign given role for this connection.

;; A [Maybe X] is one of:
;; - empty
```

```
;; - (list X)

;; An Role is one of 'broadcast or 'listen
(define (role? x) (or (eq? 'broadcast) (eq? 'listen)))

(define-class unknown%
  (fields iw)
  (define (mail msg) empty)
  (define (ident iw role)
    (cond [(iworld=? (send this iw) iw)
           (cond [(eq? role 'broadcast)
                  (new caster% (send this iw))]
                 [else
                  (new castee% (send this iw))])]
          [else this])))

(define-class caster%
  (fields iw)
  (define (mail msg) empty)
  (define (ident iw role) this))

(define-class castee%
  (fields iw)
  (define (mail msg)
    (list (make-mail (send this iw) msg)))
  (define (ident iw role) this))
```

Now that we've seen the basics of communicating programs, let's build something a little more substantial.

7.8   Exercises

7.8.1   Tron

For this assignment, you will design and develop the Tron Lightcycle game. The basic idea of this game, if you haven't seen the movie, is that two players move around the screen, leaving a trail of where each of the players has been. If a player runs into the trail that they or the other player has left behind, or into a wall, they lose. If the two players simultaneously hit each other, or both hit a wall or trail at the same time, the game is a tie.

Here's a flash game where you can play the game yourself online.

1. **Distributed Tron**

   The first version of Tron you will develop is a distributed one, using the `class/universe` library. You will need to design two separate parts of the program:

The server: this will accept connections from two clients, communicate with the clients via messages indicating the directions the clients want to move, and then send back updated information about the positions of both players and the trails on the board.

The client: you should only need to *implement* one client, but you will *run* two of them, one for each player. The client will draw the world to the screen, recieve messages from the server and update the world state in response, allow the user to input their desired direction (probably via the arrow keys) and communicate this direction to the server.

Once you've implemented both the server and client, you'll be able to play against your friends and classmates over the network.

2. **Computer Tron**

   In this part of the assignment, you'll implement a new kind of client—a computer player. This player will, like the regular client, display the world as well as send and recieve messages to and from the server. However, it won't take input from the user; instead it will make decisions itself based on the state of the board.

   There is no requirement for any particular behavior for your computer player— you can have it behave randomly, behave dumbly, or be the world's best tron player. We won't grade your assignment based on its playing choices, but we encourage you to go wild with your choices of how the computer player behaves.

Chapter 8

Guess my number

In this chapter, we'll take an in-depth look at small, but interesting distributed game: the "Guess my Number" game.

8.1 One Player Guess my Number

Let's start by considering a slimmed-down version of guess my number in which there is just one player, the client, who tries to guess the number the server is thinking of.

8.1.1 The GmN server

In this simplified version of the game, there is not much the server needs to do:

- it should remember what number it is thinking of,

- and it should respond to guesses made by the player.

From the server's point of view, the interactions look like the following, supposing the server is thinking of 5:

```
 Client              Server
--------             ---------
                     Message
         ========>
         --------->
         <--------- "too small"
         ---------> 9
         <--------- "too big"
         ---------> 6
         <--------- "too big"
         ---------> 5
         <--------- "just right"
```

In order to respond with "too big", "too small", or "just right", the state of the server will need to include the number that the server has in mind. Thus a natural representation of the state of the server is an object with a single field that contains the number, and an `on-msg` method that will respond to a guess made by the player:

```
(define-class thinking-of%
  (fields n)
  ;; on-msg : IWorld SExp -> Universe
  ;; Mail response to guess from given world
  (define (on-msg iw msg) ...))
```

In support of `on-msg`, let's design a method that consumes a guess (a real number) and produces either the string `"too small"`, `"too big"`, or `"just right"` depending on whether the guess is smaller, bigger, or equal to the number the server contains.

`thinking-of%`

```
;; A Response is one of:
;; - "too big"
;; - "too small"
;; - "just right"

;; guess : Real -> Response
;; Respond to a given guess
(check-expect ((new thinking-of% 7) . guess 5) "too small")
(check-expect ((new thinking-of% 7) . guess 9) "too big")
(check-expect ((new thinking-of% 7) . guess 7) "just right")
(define (guess m) ...)
```

The final step of writing the code is trivial at this point, so we can move on to the `on-msg` method:

`thinking-of%`

```
;; on-msg : IWorld SExp -> Universe
;; Mail response to guess from given world
(check-expect ((new thinking-of% 7) . on-msg iworld1 "Bogus")
              (new thinking-of% 7))
(check-expect ((new thinking-of% 7) . on-msg iworld1 5)
              (make-bundle (new thinking-of% 7)
                           (list (make-mail iworld1 "too small"))
                           empty))
(define (on-msg iw msg) ...)
```

Again the code is trivial once the initial design work is complete.

8.1.2   The GmN Client

The client program will register with the server and allow the user to propose guesses which are sent to the server. The response of "too small", "too larg", or "just right" is

shown to the user and they can propose more guesses if desired. For the moment, let's just focus on guessing a single digit to make things simple. We'll look at mult-digit guesses later. In this simplified setting the world can be in one of two states: the client is accepting guesses, or it has a guess and it is waiting for the server to respond to that guess.

So we arrive at the interface definitions:

```
;; A Client is one of:
;; - Waiting
;; - Accepting
;;
;; A Waiting implements:
;; - to-draw : -> Scene
;; - on-receive : SExp -> Client
;;
;; An Accepting implements:
;; - to-draw : -> Scene
;; - on-key : SExp -> Client
```

So from the client's perspective, interactions with the server will look like the following:

```
    Client                      Server
------------------            ---------

Event    State
Bang     Accepting  =========>
Key "3"  Waiting    ---------> 3
Msg      Accepting  <--------- "too small"
Key "9"  Waiting    ---------> 9
Msg      Accepting  <--------- "too big"
Key "6"  Waiting    ---------> 6
Msg      Accepting  <--------- "too big"
Key "5"  Waiting    ---------> 5
Msg      Accepting  <--------- "just right"
```

On further reflection, you should discover that there are in fact two different kinds of `Accepting` states the client could be in: one in which no guess has been made—so client is waiting to accept what will be the initial guess, and another in which a guess has been and a response has been received from the server about that guess. In this case, we want the client to display the guess and the server's response while waiting for the next guess. Based on this analysis, it's clear we will need two implementations of `Accepting` with different behaviour and data:

```
;; A (new no-guess%) implements Accepting
(define-class no-guess%
  (define (register) ...)
  (define (to-draw) ...)
  (define (on-key ke) ...))
```

```
;; A (new waiting% Number) implements Waiting
(define-class waiting%
  (fields n)
  (define (to-draw) ...)
  (define (on-receive msg) ...))

;; A (new inform% Number String) implements Accepting
(define-class inform%
  (fields n msg)
  (define (to-draw) ...)
  (define (on-key msg) ...))
```

Here is the interactions diagram, revised slightly to be more precise about the state of the client:

```
      Client                               Server
----------------------------             ---------

Event    State
Bang     (new no-guess%)      =========>
Key "3"  (new waiting% 3)     ---------> 3
Msg      (new inform% 3 ...)  <--------- "too small"
Key "9"  (new waiting% 9)     ---------> 9
Msg      (new inform% 9 ...)  <--------- "too big"
...
```
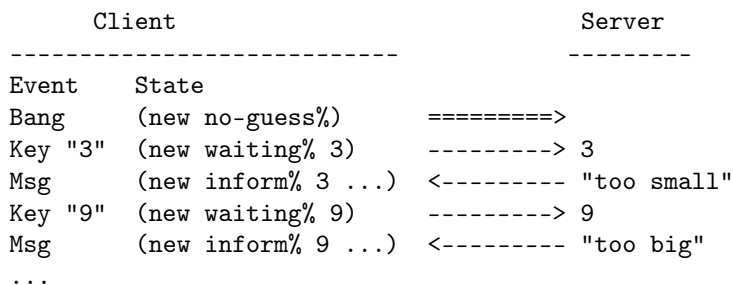
First, let's fix the dimensions of the background image and make a function for displaying strings:

```
(define MT-SCENE (empty-scene 400 400))
;; String -> Image
(define (txt str)
  (text str 40 'red))
```

The `no-guess%` class represents the initial state of the client and should display a message to the user to make a guess. When a key is pressed in this state, if it's numeric, that number becomes the new guess. Otherwise the key is ignored. Some examples:

no-guess%

```
(check-expect ((new no-guess%) . to-draw)
              (overlay (txt "Take a guess") MT-SCENE))
(check-expect ((new no-guess%) . on-key "h")
              (new no-guess%))
(check-expect ((new no-guess%) . on-key "7")
              (make-package (new waiting% 7) 7))
```

The remaining work of writing the code is easy:

no-guess%

```
(define (to-draw)
  (overlay (txt "Take a guess") MT-SCENE))

(define (on-key ke)
  (local [(define n (string->number ke))]
    (cond [(number? n)
           (make-package (new waiting% n) n)]
          [else this])))
```

The `string->number` function is being used to test for numeric key events—it works by producing `false` when given a string that cannot be converted to a number, otherwise it converts the string to a number.

The `waiting%` class represents the client waiting for a response from the server. To render this state, let's display the number that has been guessed. Since this class of objects doesn't have a `on-key` event, we are implicitly disallowing further guesses while waiting. If the server responds with a string message, the client transitions to a new accepting state. Some examples:

waiting%

```
(check-expect ((new waiting% 5) . to-draw)
              (overlay (txt "Guessed: 5") MT-SCENE))
(check-expect ((new waiting% 5) . on-receive "too small")
              (new inform% 5 "too small"))
(check-expect ((new waiting% 5) . on-receive 'something)
              (new waiting% 5))
```

All that's left is to write some code:

waiting%

```
(define (to-draw)
  (overlay (beside (txt "Guessed: ")
                   (txt (number->string (this . n))))
           MT-SCENE))

(define (on-receive msg)
  (cond [(string? msg)
         (new inform% (this . n) msg)]
        [else this]))
```

Finally, the `inform%` class represents clients that have guessed, received a response, and are now waiting for subsequent guesses.

By virtue of not having an `on-receive` method, a client in the accepting state will ignore message from the server (which should be considered an error on the server's part). Just like `no-guess%`, it should accept numeric key presses as a new guess and transition to the waiting state. To render the state, we should display the guess and the feedback from the server. For example:

```
(check-expect ((new inform% 7 "too small") . to-draw)
              (overlay (txt "Guessed: 7; too small") MT-SCENE))
(check-expect ((new inform% 7 "too small") . on-key "a")
              (new inform% 7 "too small"))
(check-expect ((new inform% 7 "too small") . on-key "9")
              (make-package (new waiting% 9) 9))
```

The code is just as easy as in the other classes:

<div align="right">

```
inform%
```

</div>

```
(define (to-draw)
  (overlay (beside (txt "Guessed: ")
                   (txt (number->string (this . n)))
                   (txt "; ")
                   (txt (this . msg)))
           MT-SCENE))

(define (on-key ke)
  (local [(define n (string->number ke))]
    (cond [(number? n)
           (make-package (new waiting% n) n)]
          [else this])))
```

Notice that the `on-key` method of `inform%` and `no-guess%` are *identical*. We'll discuss how to abstract such identical code in section ???.

Now we can play the game with:

```
(launch-many-worlds (big-bang (new no-guess%))
                    (universe (new thinking-of% (random 10))))
```

8.1.3   Many Players, One Number

Although we've developed this program under the simplifying assumption that there's only one client, the server works just as well when there are multiple clients. Under this scenario, all of the clients are trying to guess the one number the server is thinking of in parallel. For example, try this out:

```
(launch-many-worlds (big-bang (new no-guess%))
                    (big-bang (new no-guess%))
                    (big-bang (new no-guess%))
                    (universe (new thinking-of% (random 10))))
```

It would take more work and a redesign of the server if we wanted to have the server think of a number for each of the clients independently. We'll examine such a redesign later in the chapter, but first, let's look at how to implement a better client.

8.1.4 Guessing Big

It's not so fun to play guess my number when the numbers can only be between zero and nine. But note that this limitation exists only in the client. The server is perfectly capable of serving up any real number, but the client as currently designed will have a difficult time against (`new thinking-of%` 11). The good news is that the hard part—dealing with the protocol of messages—is behind us. It's a small matter of iterative refinement to make the client capable of playing larger numbers.

Looking back at our initial design, it should be clear that some of the peices we developed can still be used. In particular, the `waiting%` class is perfectly sufficient for dealing with numbers larger than 9. The problem is we have no good way of getting to that point from `no-guess%`. So let's reconsider the states of the client. It seems that if we want to accept multi-digit input, we need to have a new class of `Accepting` clients that has received some digits but is ready to accept more. We have to settle on some input to signify the end of digits, at which point a complete number has been given and can be shipped off to the server as a guess.

```
;; A (new continue% NumberString) implements Accepting
(define-class continue%
  (fields digits)
  (define (to-draw) ...)
  (define (on-key ke) ...))
```

The `digits` field will hold a string containing all of the digits entered so far (it will always be non-empty and can be converted to a number with `string->number`). Let's say that when the user presses the "Enter" key, the input is complete. If the user presses any key other than "Enter" or a digit, let's ignore it. To render a continue state, let's display "Guessing:" and the digits entered so far followed by an underscore to indicate that the client is waiting for more input.

We can now formulate some examples:

`continue%`

```
(check-expect ((new continue% "123") . to-draw)
              (overlay "Guessing: 123_" MT-SCENE))
(check-expect ((new continue% "123") . on-key "4")
              (new continue% "1234"))
(check-expect ((new continue% "123") . on-key "a")
              (new continue% "123"))
(check-expect ((new continue% "123") . on-key "\r")
              (make-package (new waiting% 123) 123))
```

Now for the code:

`continue%`

```
(define (to-draw)
  (overlay (beside (txt "Guessing: ")
                   (txt (this . digits))
```

```
                           (txt "_"))
                  MT-SCENE))

(define (on-key ke)
  (cond [(number? (string->number ke))
            (new continue% (string-append (this . digits) ke))]
        [(key=? "\r" ke)
            (local [(define n (string->number (this . digits)))]
              (make-package (new waiting% n) n))]
        [else this]))
```

We now need to go back and revise `no-guess%` and `inform%` to transition to `continue%` whenever a digit key is pressed:

```
(check-expect ((new no-guess%) . on-key "7")
              (new continue% "7"))
(check-expect ((new inform% 5 "too small") . on-key "7")
              (new continue% "7"))
```

The code in both cases is:

no-guess% and inform%

```
(define (on-key ke)
  (cond [(number? (string->number ke))
            (new continue% ke)]
        [else this]))
```

Now try this out:

```
(launch-many-worlds
  (big-bang (new no-guess%))
  (universe (new thinking-of% (random 1000))))
```

8.2 Two player guess my number

[FIXME this section is out of sync with previous sections and needs to be re-written.]

Now let's write a 2-player version of the game where one player thinks of a number and the other player guesses.

Here is the server:

```
#lang class/0
(require class/universe)

;; A Universe is a (new universe% [U #f Number] [U #f IWorld] [U #f IWorld]).
(define-class universe%
  (fields number
```

```
          picker
          guesser)

;; is the given world the picker?
(define (picker? iw)
  (and (iworld? (send this picker))
       (iworld=? iw (send this picker))))

;; is the given world the guesser?
(define (guesser? iw)
  (and (iworld? (send this guesser))
       (iworld=? iw (send this guesser))))

(define (on-new iw)
  (cond [(false? (send this picker))
         (make-bundle
          (new universe% false iw false)
          (list (make-mail iw "pick a number"))
          empty)]
        [(false? (send this guesser))
         (make-bundle
          (new universe% (send this number) (send this picker) iw)
          empty
          empty)]
        [else
         (make-bundle this empty (list iw))]))

(define (on-msg iw m)
  (cond [(and (picker? iw)
              (false? (send this number)))
         (make-bundle
          (new universe% m (send this picker) (send this guesser))
          empty
          empty)]
        [(picker? iw) ;; already picked a number
         (make-bundle this empty empty)]
        [(and (guesser? iw)
              (number? (send this number)))
         (make-bundle this
                      (list (make-mail iw (respond m (send this number))))
                      empty)]
        [(guesser? iw)
         (make-bundle this
                      (list (make-mail iw "no number"))
                      empty)]])))
```

```
;; Number Number -> String
(define (respond guess number)
  (cond [(< guess number) "too small"]
        [(> guess number) "too big"]
        [else "just right"]))

(universe (new universe% false false false))
```

The client stays the same! You can launch the two players with:

```
(launch-many-worlds
 (big-bang (new guess-world% "guess a number"))
 (big-bang (new guess-world% "guess a number")))
```

Part III

Abstraction with Objects

Chapter 9

Abstraction via Delegation

9.1 Constructor design issue in modulo zombie (Assignment 3, Problem 3)

Course staff solution for regular zombie game:

<div align="right">world%</div>

```
(define (teleport)
  (new world%
       (new player%
            (random WIDTH)
            (random HEIGHT))
       (this . zombies)
       (this . mouse)))
```

This has a significant bug: it always produces a plain `player%`, not a `modulo-player%`.

Bug (pair0MN):

<div align="right">modulo-player%</div>

```
(define (teleport)
  (new player%
       (* -1 (random WORLD-SIZE))
       (* -1 (random WORLD-SIZE))))
```

This has a similar bug: it always produces a plain `player%`, not a `modulo-player%`. However, it's in the the `modulo-player%` file, so there's an easy fix.

Lack of abstraction (pair0PQ):

<div align="right">modulo-player%</div>

```
;; warp : Real Real -> ModuloPlayer
;; change the location of this player to the given location
```

123

```
(define (warp x y)
  (new modulo-player%
       (this . dest-x)
       (this . dest-y)
       x y))
```

player%

```
;; warp : Real Real -> Player
;; change the location of this player to the given location
(define (warp x y)
  (new player%
       (this . dest-x)
       (this . dest-y)
       x y))
```

This works correctly (this is the fix for the bug in Pair0MN's solution), but it duplicates code.

We want to fix these bugs without duplicating code.

Possible solutions (suggested in class):

- Parameterize the `teleport` method with a class name. Unfortunately, this doesn't work because the class name in `new` is not an expression.

- Use `this` as the class name. This doesn't work because `this` is an *instance*, not a *class*.

The solution is to add a new method to the interface, which constructs a new instance of the appropriate class. So, we add this method to the `player%` class:

```
(define (move x y)
  (new player% x y))
```

And this method to the `modulo-player%` class:

```
(define (move x y)
  (new modulo-player% x y))
```

Here's an example of the technique in full. We start with these classes:

```
#lang class/1
(define-class s%
  (fields x y))

;; A Foo is one of:
;; - (new c% Number Number)
;; - (new d% Number Number)
```

```
(define-class c%
  (super s%)
  (define (make x y) (new c% x y))
  (define (origin) (new c% 0 0)))
(define-class d%
  (super s%)
  (define (make x y) (new d% x y))
  (define (origin) (new d% 0 0)))
```

Now we abstract the `origin` method to use `make`, and we can abstract `origin` to the superclass `s%`, since it becomes identical in both classes, avoiding the code duplication.

```
#lang class/1
(define-class s%
  (fields x y)
  (this . make 0 0))

;; A Foo is one of:
;; - (new c% Number Number)
;; - (new d% Number Number)

(define-class c%
  (super s%)
  (define (make x y)
    (new c% x y)))
(define-class d%
  (super s%)
  (define (make x y)
    (new d% x y)))

(new c% 50 100)
((new c% 50 100) . origin)
```

9.2 Abstracting list methods with different representations

Here is a parametric list interface definition:

```
;; ============================================================
;; Parametric lists

;; A [Listof X] implements
;;
;; cons : X -> [Listof X]
```

```
;; Add the given element to the front of the list.
;;
;; empty : -> [Listof X]
;; Produce the empty list.
;;
;; length : -> Nat
;; Count the number of elements in this list.
;;
;; append : [Listof X] -> [Listof X]
;; Append the given list to the end of this list.
;;
;; reverse : -> [Listof X]
;; Reverse the order of elements in this list.
;;
;; map : [X -> Y] -> [Listof Y]
;; Construct the list of results of applying the function
;; to elements of this list.
;;
;; filter : [X -> Boolean] -> [Listof X]
;; Construct the list of elements in this list that
;; satisfy the predicate.
;;
;; foldr : [X Y -> Y] Y -> Y
;; For elements x_0...x_n, (f x_0 ... (f x_n b)).
;;
;; foldl : [X Y -> Y] Y -> Y
;; For elements x_0...x_n, (f x_n ... (f x_0 b)).
```

Here's the usual implementation of a small subset of this interface, first for the recursive union implementation:

```
(define-class cons%
  (fields first rest)

  (define (cons x)
    (new cons% x this))

  (define (empty)
    (new empty%))

  (define (length)
    (add1 (this . rest . length)))

  (define (foldr c b)
    (c (this . first)
       (this . rest . foldr c b))))
```

```
(define-class empty%

  (define (cons x)
    (new cons% x this))

  (define (empty)
    this)

  (define (length)
    0)

  (define (foldr c b)
    b))
```

And for the wrapper list implementation:

```
(define-class wlist%
  (fields ls)

  (define (cons x)
    (new wlist% (ls:cons x (this . ls))))

  (define (empty)
    (new wlist% ls:empty))

  (define (length)
    (ls:length (this . ls)))

  (define (foldr c b)
    (ls:foldr c b (this . ls))))
```

None of these look the same, so how can we abstract? Our abstraction design recipe for using inheritance requires that methods look identical in order to abstract them into a common super class. But, for example, the length method looks like this for wlist%:

```
(define (length)
  (ls:length (this . ls)))
```

Like this for empty%:

```
(define (length)
  0)
```

And like this for cons%:

```
(define (length)
  (add1 (this . rest . length)))
```

Before we can abstract this method, we must make them all look the same. Fortunately, many list operations can be expressed using just a few simple operations, of which the most important is `foldr`. Here's an implementation of `length` which just uses `foldr` and simple arithmetic.

```
(define (length)
  (this . foldr (λ (a b) (add1 b)) 0))
```

Note that this isn't specific to any one implementation of lists—in fact, we can use it for any of them. This means that we can now abstract the method, creating a new `list%` class to share all of our common code:

```
(define-class list%
  (define (length)
    (this . foldr (λ (a b) (add1 b)) 0))
  ;; other methods here
  )
```

The only methods that need to be implemented differently for different list versions are `empty` and `cons`, because they construct new lists, and `foldr`, because it's the fundamental operation we use to build the other operations out of. It's also helpful to implementat `foldl`, since it's fairly complex to factor out.

9.3   Delegation

So far, we've seen multiple ways to abstract repeated code. First, in Fundies 1, we saw functional abstraction, where we take parts of functions that differ and make them parameters to the abstracted function. Second, in this class we've seen abstraction by using inheritance, where if methods in two related classes are identical, they can be lifted into one method in a common superclass.

However, can we still abstract common code without *either* of these mechanisms? Yes.

Consider the `class/0` language, *without* helper functions. We can write a binary tree class like this:

```
#lang class/0
;; A BT is one of:
;; - (new leaf% Number)
;; - (new node% Number BT BT)

;; double : Number -> BT
;; Double this tree and put the number on top.

(define-class leaf%
  (fields number)
```

```
    (define (double n)
      (new node% n this this)))

(define-class node%
  (fields number left right)

  (define (double n)
    (new node% n this this)))
```

Unfortunately, the `double` method is identical in both the `leaf%` and `node%` classes. How can we abstract this without using inheritance or a helper function?

One solution is to create a new class, and *delegate* the responsibility of doing the doubling to it. Below is an example of this:

```
(define-class helper%
  ;; Number BT -> BT
  ;; Double the given tree and puts the number on top.
  (define (double-helper number bt)
    (new node% number bt bt)))

(define tutor (new helper%))

(define-class leaf%
  (fields number)

  (define (double n)
    (tutor . double-helper n this)))

(define-class node%
  (fields number left right)

  (define (double n)
    (tutor . double-helper n this)))
```

The `helper%` class has just one method, although we could add as many as we wanted. We also need only one instance of `helper%`, called `tutor`, although we could create new instances when we needed them as well. Now the body of `double-helper` contains all of the doubling logic in our program, which might become much larger without needing duplicate code.

Chapter 10

Abstraction via Inheritance

10.1   Method inheritance with binary trees

We developed classes for representing binary trees and wrote a couple methods for
binary trees, but one of the troubling aspects of this code is the fact that the two imple-
mentations of double are identical:

```
#lang class/0

;; A BT is one of:
;; - (new leaf% Number)
;; - (new node% Number BT BT)

(define-class leaf%
  (fields number)
  ...
  ;; Number -> BT
  ;; double the leaf and put the number on top
  (define (double n)
    (new node% n this this)))

(define-class node%
  (fields number left right)
  ...
  ;; Number -> BT
  ;; double the node and put the number on top
  (define (double n)
    (new node% n this this)))
```

   If we think by analogy to the structural version of this code, we have something
like this:

```
#lang class/0
```

```
;; A BT is one of:
;; - (make-leaf Number)
;; - (make-node Number BT BT)
(define-struct leaf (number))
(define-struct node (number left right))

;; BT Number -> BT
;; Double the given tree and put the number on top.
(define (double bt n)
  (cond [(leaf? bt) (make-node n bt bt)]
        [(node? bt) (make-node n bt bt)]))
```

We would arrive at this code by developing the double function according to the design recipe; in particular, this code properly instantiates the template for binary trees. However, after noticing the duplicated code, it is straightforward to rewrite this structure-oriented function into an equivalent one that duplicates no code. All cases of the cond clause produce the same result, hence the cond can be eliminated, replaced by a single occurrence of the duplicated answer expressions:

```
;; BT Number -> BT
;; Double the given tree and put the number on top.
(define (double bt n)
  (make-node n bt bt))
```

But switching back to the object-oriented version of this code, it is not so simple to "eliminate the cond"—there is no cond! We would like to write this code just once, but the real question is *where*? The solution, in this context, is to *lift* the identical method defintions to a common *super* class. That is, we define a third class that contains the method shared among leaf% and node%:

```
(define-class bt%
  ;; -> BT
  ;; Double this tree and put the number on top.
  (define (double n)
    (new node% n this this)))
```

The double method can be removed from the leaf% and node% classes and instead these class can rely on the bt% definition of double, but to do this we must establish a relationship between leaf%, node% and bt%: we declare that leaf% and node% are *subclasses* of bt%, and therefore they *inherit* the double method; it is as if the code were duplicated without actually writing it twice:

```
(define-class leaf%
  (super bt%)
  (fields number)
  ;; -> Number
```

```
  ;; count the number of numbers in this leaf
  (define (count)
    1))

(define-class node%
  (super bt%)
  (fields number left right)
  ;; -> Number
  ;; count the number of numbers in this node
  (define (count)
    (+ 1
       (send (send this left) count)
       (send (send this right) count))))
```

10.2 The `class/1` language

To accommodate this new feature—*inheritance*—we need to adjust our programming language. We'll now program in `class/1`, which is a superset of `class/0`—all `class/0` programs are `class/1` programs, but not vice versa. The key difference is the addition of the `(super class-name)` form.

At this point we can construct binary trees just as before, and all binary trees understand the `double` method even though it is only defined in `bt%`:

```
> (new leaf% 7)
(new leaf% 7)
> (send (new leaf% 7) double 8)
(new node% 8 (new leaf% 7) (new leaf% 7))
> (send (send (new leaf% 7) double 8) double 9)
(new node% 9 (new node% 8 (new leaf% 7) (new leaf% 7)) (new node%
8 (new leaf% 7) (new leaf% 7)))
```

There are a couple other features of the `class/1` language that are worth knowing about. One is a trivial, but very handy shorthand form for writing `send`. The shorthand form is to write `(o . m)` to call method `m` on object `o`, that is, `(o . m)` is equivalent to `(send o m)`. Another nice feature of the "dot notation" is that it makes it easy to stack up a bunch of method calls, so for example `(o . m . n x y . p)` is shorthand for

```
(send (send (send o m) n x y) p)
```

From here on out, the book will use the dot notation since it's so nice.

10.3 "Abstract" classes

At this point, it is worth considering the question: what does a `bt%` value represent? We have arrived at the `bt%` class as a means of abstracting identical methods in `leaf%`

and `node%`, but if we say `(new bt%)`, as we surely can, what does that *mean*? The answer is: *nothing*.

Going back to our data definition for BTs, it's clear that the value `(new bt%)` is **not** a BT since a BT is either a `(new leaf% Number)` or a `(new node% Number BT BT)`. In other words, a `(new bt%)` makes no more sense as a representation of a binary tree than does `(new node% "Hello Fred" 'y-is-not-a-number add1)`. With that in mind, it doesn't make sense for our program to ever construct `bt%` objects—they exist purely as an abstraction mechanism. Some languages, such as Java, allow you to enforce this property by declaring a class as "abstract"; a class that is declared abstract cannot be constructed. Our language will not enforce this property, much as it does not enforce contracts. Again we rely on data definitions to make sense of data, and `(new bt%)` doesn't make sense.

10.4   Data inheritance with binary trees

Inheritance allows us to share methods amongst classes, but it also possible to share data. Just as we observed that `double` was the same in both `leaf%` and `node%`, we can also observe that there are data similarities between `leaf%` and `node%`. In particular, both `leaf%` and `node%` contain a `number` field. This field can be abstracted just like `double` was—we can lift the field to the `bt%` super class and eliminate the duplicated field in the subclasses:

```
(define-class bt%
  (fields number)
  ;; -> BT
  ;; Double this tree and put the number on top.
  (define (double n)
    (new node% n this this)))

(define-class leaf%
  (super bt%)
  ;; -> Number
  ;; count the number of numbers in this leaf
  (define (count)
    1))

(define-class node%
  (super bt%)
  (fields left right)
  ;; -> Number
  ;; count the number of numbers in this node
  (define (count)
    (+ 1
       (this . left . count)
       (this . right . count))))
```

The `leaf%` and `node%` class now inherit both the `number` field and the `double` method from `bt%`. This has a consequence for constructing new instances. Previously it was straightforward to construct an object: you write down `new`, the class name, and as many expressions as there are fields in the class. But now that a class may inherit fields, you must write down as many expressions as there are fields in the class definition itself and in all of the super classes. What's more, the order of arguments is important. The fields defined in the class come first, followed by the fields in the immediate super class, followed by the super class's super classes, and so on. Hence, we still construct `leaf%`s as before, but the arguments to `new` for `node%` are changed: it takes the left subtree, the right subtree, and *then* the number at that node:

```
;; A BT is one of:
;; - (new leaf% Number)
;; - (new node% BT BT Number)
```

```
> (new leaf% 7)
(new leaf% 7)
> (new node% (new leaf% 7) (new leaf% 13) 8)
(new node% (new leaf% 7) (new leaf% 13) 8)
```

Although none of our method so far have needed to access the `number` field, it is possible to access `number` in `leaf%` and `node%` (and `bt%`) methods *as if* they had their own `number` field. Let's write a `sum` method to make it clear:

Notice that the `double` method swapped the order of arguments when constructing a new `node%` to reflect the fact that the node constructor now takes values for its fields first, then values for its inherited fields.

```
(define-class bt%
  (fields number)
  ;; -> BT
  ;; Double this tree and put the number on top.
  (define (double n)
    (new node% this this n)))

(define-class leaf%
  (super bt%)
  ;; -> Number
  ;; count the number of numbers in this leaf
  (define (count)
    1)

  ;; -> Number
  ;; sum all the numbers in this leaf
  (define (sum)
    (this . number)))

(define-class node%
  (super bt%)
  (fields left right)
```

```
;; -> Number
;; count the number of numbers in this node
(define (count)
  (+ 1
     (this . left . count)
     (this . right . count)))

;; -> Number
;; sum all the numbers in this node
(define (sum)
  (+ (this . number)
     (this . left . sum)
     (this . right . sum))))
```

As you can see, both of the `sum` methods refer to the `number` field, which is inherited from `bt%`.

10.5   Inheritance with shapes

Let's consider another example and see how data and method inheritance manifests. This example will raise some interesting issues for how super classes can invoke the methods of its subclasses. Suppose we are writing a program that deals with shapes that have position. To keep the example succinct, we'll consider two kinds of shapes: circles and rectangles. This leads us to a union data definition (and class definitions) of the following form:

We are using `+Real` to be really precise about the kinds of numbers that are allowable to make for a sensible notion of a shape. A circle with radius `-5` or `3+2i` doesn't make a whole lot of sense.

```
;; A Shape is one of:
;; - (new circ% +Real Real Real)
;; - (new rect% +Real +Real Real Real)
;; A +Real is a positive, real number.
(define-class circ%
  (fields radius x y))
(define-class rect%
  (fields width height x y))
```

Already we can see an opportunity for data abstraction since `circ%`s and `rect%`s both have `x` and `y` fields. Let's define a super class and inherit these fields:

```
;; A Shape is one of:
;; - (new circ% +Real Real Real)
;; - (new rect% +Real +Real Real Real)
;; A +Real is a positive, real number.
(define-class shape%
  (fields x y))
(define-class circ%
  (super shape%)
```

```
  (fields radius))
(define-class rect%
  (super shape%)
  (fields width height))
```

Now let's add a couple of methods: `area` will compute the area of the shape, and `draw-on` will take a scene and draw the shape on the scene at the appropriate position:

```
(define-class circ%
  (super shape%)
  (fields radius)

  ;; -> +Real
  (define (area)
    (* pi (sqr (this . radius))))

  ;; Scene -> Scene
  ;; Draw this circle on the scene.
  (define (draw-on scn)
    (place-image (circle (this . radius) "solid" "black")
                 (this . x)
                 (this . y)
                 scn)))

(define-class rect%
  (super shape%)
  (fields width height)

  ;; -> +Real
  ;; Compute the area of this rectangle.
  (define (area)
    (* (this . width)
       (this . height)))

  ;; Scene -> Scene
  ;; Draw this rectangle on the scene.
  (define (draw-on scn)
    (place-image (rectangle (this . width) (this . height) "solid" "black")
                 (this . x)
                 (this . y)
                 scn)))
```

Examples:

```
> (send (new circ% 30 75 75) area)
2827.4333882308138
> (send (new circ% 30 75 75) draw-on (empty-scene 150 150))
```
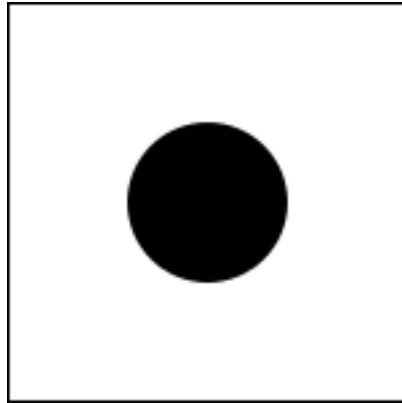
```
> (send (new rect% 30 50 75 75) area)
1500
> (send (new rect% 30 50 75 75) draw-on (empty-scene 150 150))
```
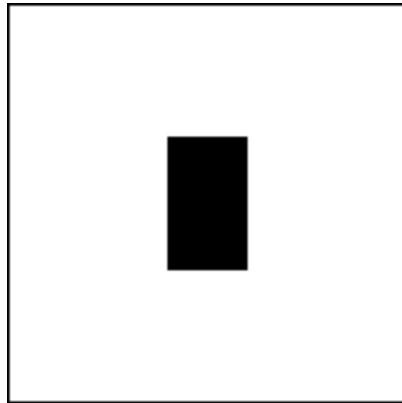

The `area` method is truly different in both variants of the shape union, so we shouldn't attempt to abstract it by moving it to the super class. However, the two definitions of the `draw-on` method are largely the same. If they were *identical*, it would be easy to abstract the method, but until the two methods are identical, we cannot lift the definition to the super class. One way forward is to rewrite the methods by pulling out the parts that differ and making them separate methods. What differs between these two methods is the expression constructing the image of the shape, which suggests defining a new method `img` that constructs the image. The `draw-on` method can now call `img` and rewriting it this way makes both `draw-on` methods identical; the method can now be lifted to the super class:

```
(define-class shape%
  (fields x y)

  ;; Scene -> Scene
  ;; Draw this shape on the scene.
  (define (draw-on scn)
```

```
      (place-image (img)
                   (send this x)
                   (send this y)
                   scn)))
```

But there is a problem with this code. While this code makes sense when it occurs inside of `rect%` and `circ%`, it doesn't make sense inside of `shape%`. In particular, what does `img` mean here? The `img` method is a method of `rect%` and `circ%`, but not of `shape%`, and therefore the name `img` is unbound in this context.

On the other hand, observe that all shapes are either `rect%`s or `circ%`s. We therefore *know* that the object invoking the `draw-on` method understands the `img` message, since both `rect%` and `circ%` implement the `img` method. Therefore we can use `send` to invoke the `img` method on `this` object and thanks to our data definitions for shapes, it's guaranteed to succeed. (The message send would fail if `this` referred to a `shape%`, but remember that `shape%`s don't make sense as objects in their own right and should never be constructed).

We arrive at the following final code:

```
#lang class/1
(require 2htdp/image)

;; A Shape is one of:
;; - (new circ% +Real Real Real)
;; - (new rect% +Real +Real Real Real)
;; A +Real is a positive, real number.
(define-class shape%
  (fields x y)

  ;; Scene -> Scene
  ;; Draw this shape on the scene.
  (define (draw-on scn)
    (place-image (send this img)
                 (send this x)
                 (send this y)
                 scn)))

(define-class circ%
  (super shape%)
  (fields radius)

  ;; -> +Real
  ;; Compute the area of this circle.
  (define (area)
    (* pi (sqr (send this radius))))

  ;; -> Image
```

```
  ;; Render this circle as an image.
  (define (img)
    (circle (send this radius) "solid" "black")))

(define-class rect%
  (super shape%)
  (fields width height)

  ;; -> +Real
  ;; Compute the area of this rectangle.
  (define (area)
    (* (send this width)
       (send this height)))

  ;; -> Image
  ;; Render this rectangle as an image.
  (define (img)
    (rectangle (send this width) (send this height) "solid" "black")))

(check-expect (send (new rect% 10 20 0 0) area)
              200)
(check-within (send (new circ% 10 0 0) area)
              (* pi 100)
              0.0001)
(check-expect (send (new rect% 5 10 10 20) draw-on
                    (empty-scene 40 40))
              (place-image (rectangle 5 10 "solid" "black")
                           10 20
                           (empty-scene 40 40)))
(check-expect (send (new circ% 4 10 20) draw-on
                    (empty-scene 40 40))
              (place-image (circle 4 "solid" "black")
                           10 20
                           (empty-scene 40 40)))
```

10.6   Revisiting the Rocket with Inheritance

At this point, you may recall that unsettling feeling you had in the discussion of section 3.4, in which we wrote duplicate, identical methods in both the `landing%` and `takeoff%` variants of Rocket objects:

landing% and takeoff%

```
;; render : -> Scene
;; Render this rocket as a scene.
(define (render)
```

```
    (send this draw-on MT-SCENE))

; draw-on : Scene -> Scene
; Draw this rocket on to scene.
(define (draw-on scn)
  (overlay/align/offset "center" "bottom"
                        ROCKET
                        0 (add1 (send this dist))
                        scn))
```

We now have the mechanism to eliminate this duplication. We can define a super class of `landing%` and `takeoff%` called `rocket%` and lift the methods to this class. Moreover, since there is duplication in the data of these classes, we can likewise lift the `dist` field to `rocket%`:

```
(define-class rocket%
  (fields dist)

  ;; render : -> Scene
  ;; Render this rocket as a scene.
  (define (render)
    (this . draw-on MT-SCENE))

  ;; draw-on : Scene -> Scene
  ;; Draw this rocket on to scene.
  (define (draw-on scn)
    (overlay/align/offset "center" "bottom"
                          ROCKET
                          0 (add1 (this . dist))
                          scn)))
```

To complete the revised program, the `landing%` and `takeoff%` classes should declare `rocket%` as a super class and remove the `dist` field and `render` and `draw-on` methods:

```
(define-class landing%
  (super rocket%)
  ...)
(define-class takeoff%
  (super rocket%)
  ...)
```

10.7 Exercises

10.7.1 Abstract Lists

section 33.7

Revisit your solution to the section 5.4.1 exercise.

Use inheritance to lift method definitions to a super class to the full extent possible. (*Hint*: it will help if you realize that many of these methods may be expressed in terms of a few "core" methods.) If possible, have both the recursive union representation and the wrapper representation share a common super class.

The `cons` and `empty` methods have been added to facilitate opportunities for abstraction. You might find them useful to use when you lift methods to a common super class so that the right kind of list (either a wrapped or a recursive union list) is constructed.

10.7.2 Shapes

Here is the signature for a method to compute the area of a shape's bounding box—the smallest rectangle that can contain the shape.

```
;; bba : -> Number     (short for "bounding-box-area")
;; Compute the area of the smallest bounding box for this shape.
```

Here are some examples of how `bba` should work:

```
(check-expect ((new rect% 3 4) . bba) 12)
(check-expect ((new circ% 1.5) . bba)  9)
```

1. Design the `bba` method for the `rect%` and `circ%` class.

2. Design a super class of `rect%` and `circ%` and lift the `bba` method to the super class. Extend the shape interface as needed, but implement any methods you add.

3. Design a new variant of a Shape, Square, which should support all of the methods of the interface.

Chapter 11

Abstraction via Functions

Functions can play the role of an abstraction mechanism. They can also play the role of a class of data definitions. In this chapter we explore both roles in the context of object-oriented programs by representing functions with objects.

11.1   Functions as objects: abstracting predicates

The Boston marathon is the world oldest annual marathon and is Boston's most widely viewed sporting event. Suppose the Boston Athletic Association hired you to write software that wrangled data about their nearly 40,000 finishers. A reasonable representation would be a list of runners, where the BAA wants to track the following about each runner:

- Name

- Age (in years)

- Bib number

- Gender

- Finish time (in seconds)

That leads to following data definitions:

```
;; A LoR is one of:
;; - (new mt%)
;; - (new cons% Runner)
(define-class mt%)
(define-class cons%
  (fields first rest))

;; A Runner is a
```

```
;;    (new runner% String Natural Natural Natural Gender).
(define-class runner%
  (fields name age bib time gender))

;; A Gender implements:
;; - is-male? : -> Boolean
;; - is-female? : -> Boolean
```

Here's a simple implementation of `Gender`:

```
(define-class male%    ; implements Gender
  (define (is-male?) true)
  (define (is-female?) false))
(define-class female% ; implements Gender
  (define (is-male?) false)
  (define (is-female?) true))

(define m (new male%))
(define f (new female%))
```

Here are a few examples of runners:

```
(define johnny (new runner% "Kelley" 97 1001 (* 351 60) m))
(define bobby  (new runner% "Cheruiyot" 33 8 (* 127 60) m))
(define roberta (new runner% "Gibb" 23 121 (* 200 60) f))

(define mt (new mt%))
(define rs
  (new cons% johnny (new cons% bobby (new cons% roberta mt)))))
```

Now let's consider a couple computations the BAA might ask you to support in their program. One task is to figure calculate a list of "fast runners," which we'll take as runners finishing in under three hours. That's a straightforward computation:

`mt%`

```
;; fast : -> LoR
;; Runners in this empty list with times under three hours.
(define (fast) this)
```

`mt%`

```
;; fast : -> LoR
;; Runners in this non-empty list with times under three hours.
(define (fast)
  (cond [(< (this . first . time) (* 180 60))
         (new cons%
              (this . first)
```

```
              (this . rest . fast))]
         [else
          (this . rest . fast)]]))
```

Examples:

```
> (mt . fast)
(new mt%)
> (rs . fast)
(new cons% (new runner% "Cheruiyot" 33 8 7620 (new male%)) (new
mt%))
```

A similar method the BAA might want is one to compute the list of all runners over 50 years of age:

mt%

```
;; old : -> LoR
;; Runners in this empty list with age over 50.
(define (old) this)
```

cons%

```
;; old : -> LoR
;; Runners in this non-empty list with age over 50.
(define (old)
  (cond [(> (this . first . age) 50)
         (new cons%
              (this . first)
              (this . rest . old))]
        [else
         (this . rest . old)]))
```

Examples:

```
> (mt . old)
(new mt%)
> (rs . old)
(new cons% (new runner% "Kelley" 97 1001 21060 (new male%)) (new
mt%))
```

These methods look so similar it's natural to wonder if we can't extract their differences and express them as the instantiation of some more general method. On closer inspection, the two method differ in predicate use to select whether a given runner is in the result list or not:

- For `fast`, the predicate is "is the runner's time less than 3 hours?"

- For `old`, the predicate is "is the runner's age more than 50?"

One natural design is to write a method that encapsulates the similarities of `fast` or `old`, say `filter`, and add a parameter that consumes the differences, that is, the predicate, represented as a function [`Runner -> Boolean`]. The original methods can be recreated by supplying the appropriate function to the `filter` method, respectively:

- `(λ (r) (< (this . time) (* 180 60)))`

- `(λ (r) (> (this . age) 50))`

But what if we wanted to represent these predicates with *objects*? Is such a thing possible?

Since both of these predicates are really asking questions of runners, it may be tempting to thinking of the predicate as something that lives in the `Runner` class, and certainly we could define `fast?` and `old?` methods in `runner%`. But that wouldn't help to define a single `filter` method since we cannot parameterize a method by a method `name`—the arguments of a method must be *values*. A more fruitful perspective is to see a *predicate on runners* as something worth representing in its own right. Once we have a representation of predicates on runners, we can recover the original functionality by passing the appropriate instance of a predicate to an abstraction of `fast` and `old`.

So how should a predicate be represented? Inspired by the functional representation where we might say:

```
;; A PredicateRunner is a [Runner -> Boolean].
```

we could represent predicates on runners with objects having a single `Runner -> Boolean` method. Since we want a uniform interface for predicates, we need to decide on the name of this method: `apply`. Thus,

```
;; A PredicateRunner implements
;; - apply : Runner -> Boolean
;;   Does this predicate apply to given runner?
```

To represent the "is fast?" predicate, we define a class:

```
(define-class is-fast%
  ;; apply : Runner -> Boolean
  ;; Is the given runner fast (time less than 250)?
  (check-expect ((new is-fast%) . apply johnny) false)
  (check-expect ((new is-fast%) . apply bobby) true)
  (define (apply r)
    (< (r . time) (* 180 60))))
```

Likewise, to define the "is old?" predicate, we define another class:

```
(define-class is-old%
  ;; apply : Runner -> Boolean
  ;; Is the given runner older than 50?
  (check-expect ((new is-old%) . apply johnny) true)
```

```
(check-expect ((new is-old%) . apply roberta) false)
(define (apply r)
  (> (r . age) 50)))
```

The `(new is-fast%)` and `(new is-old%)` values can be the arguments of the abstraction of `fast` and `old`:

`mt%`

```
;; filter : PredicateRunner -> LoR
;; Runners in this empty list satisfying given predicate.
(define (filter q) this)
```

`cons%`

```
;; filter : PredicateRunner -> LoR
;; Runners in this non-empty list satisfying given predicate.
(define (filter q)
  (cond [(q . apply (this . first))
          (new cons%
               (this . first)
               (this . rest . filter q))]
         [else
          (this . rest . filter q)]))
```

What's more, you'll notice when you recreate `fast` and `old` each have duplicated, identical definitions in `mt%` and `cons%`, which means they can be lifted to a super class:

`list%`

```
;; fast : -> LoR
;; Runners in this list with times less than 3 hours.
(define (fast) (this . filter (new is-fast%)))

;; old : -> LoR
;; Runners in this list with age more than 50 years.
(define (old) (this . filter (new is-old%)))
```

At this point, it's easy to create new predicates on runners and use them to filter a list of runners. For example, if we want to select the list of male or female runners:

```
;; A (new is-male%) implements PredicateRunner
(define-class is-male%
  ;; apply : Runner -> Boolean
  ;; Is the given runner male?
  (check-expect ((new is-male%) . apply johnny) true)
  (check-expect ((new is-male%) . apply roberta) false)
  (define (apply r)
    (r . gender . is-male?)))
```

```
;; A (new is-female%) implements PredicateRunner
(define-class is-female%
  ;; apply : Runner -> Boolean
  ;; Is the given runner female?
  (check-expect ((new is-female%) . apply johnny) false)
  (check-expect ((new is-female%) . apply roberta) true)
  (define (apply r)
    (r . gender . is-female?)))
```

Example:

```
> (rs . filter (new is-female%))
(new cons% (new runner% "Gibb" 23 121 12000 (new female%)) (new
mt%))
```

Of course, these are pretty simple predicates. Let's attack something a little more ambitious. The Boston Marathon doesn't let just anybody enter the race. You have to qualify. One way to qualify for the Boston Marahon is to run under a certain time in a previous Boston Marathon (the magic of recursion at work!). Let's write a predicate that will tell us the list of runners that will qualify for next year's marathon. The BAA qualification standards for 2013 are:

```
Age     Men            Women
-----   ----------     ----------
18-34   3hrs 5min      3 hrs 35min
35-39   3hrs 10min     3 hrs 40min
40-44   3hrs 15min     3 hrs 45min
45-49   3hrs 25min     3 hrs 55min
50-54   3hrs 30min     4 hrs 0min
55-59   3hrs 40min     4 hrs 10min
60-64   3hrs 55min     4 hrs 25min
65-69   4hrs 10min     4 hrs 40min
70-74   4hrs 25min     4 hrs 55min
75-79   4hrs 40min     5 hrs 10min
80+     4hrs 55min     5 hrs 25min
```

It's tempting to write a monolithic `qualify%` predicate that includes all of logic embodied in the above table, but let's instead break it down into smaller pieces.

If we just focus on a single row, we can see that each row could form its own predicate. For example, the first runner is "is the runner over 18 and younger than 35 and, if male, has a time less than 3hrs 5min, ...?" Even this predicate can be broken down into smaller peices: "is the runner older than 18 and younger than 35?", "is the runner male?", "is the runner's time less than 3hrs 5min?", etc. And among these simple questions, we can observe there are really related *families* of questions we might ask: "is the runner older than N and younger than M?", for example.

Let's start with the age. We can represent the age group predicates with a functional object that contains data:

```
;; A (new in-age% Natural Natural) implements PredicateRunner
(define-class in-age%
  (fields lo hi)
  ;; Is the given runner within [lo,hi] age?
  (check-expect ((new in-age% 18 34) . apply roberta) true)
  (check-expect ((new in-age% 18 34) . apply johnny) false)
  (define (apply r)
    (<= (this . lo) (r . age) (this . hi))))
```

Likewise we can represent the "faster than *N* time?" predicate as a functional object with data:

```
;; A (new is-faster% Natural Natural) implements PredicateRunner
(define-class faster%
  (fields hrs mins)
  ;; apply : Runner -> Boolean
  ;; Is the given runner's time faster than hrs:mins?
  (check-expect ((new faster% 3 35) . apply roberta) true)
  (check-expect ((new faster% 3 35) . apply johnny) false)
  (define (apply r)
    (< (r . time) (* 60 (+ (* 60 (this . hrs)) (this . mins))))))
```

The "is the runner male?" and "female?" predicates are straightforward:

```
;; A (new is-male%) implements PredicateRunner
(define-class is-male%
  ;; apply : Runner -> Boolean
  ;; Is the given runner male?
  (check-expect ((new is-male%) . apply johnny) true)
  (check-expect ((new is-male%) . apply roberta) false)
  (define (apply r)
    (r . gender . is-male?)))

;; A (new is-female%) implements PredicateRunner
(define-class is-female%
  ;; apply : Runner -> Boolean
  ;; Is the given runner female?
  (check-expect ((new is-female%) . apply johnny) false)
  (check-expect ((new is-female%) . apply roberta) true)
  (define (apply r)
    (r . gender . is-female?)))
```

Now we have the small peices to say things like "the given runner qualifies if they are 18-34 years-old, a woman, and have a running time less than 3 hours and 35 minutes", but we lack the ability to put them together to form the overall predicate. In particular, we are lacking the ability to combine predicates with "and", "or", "implies", etc. But what's an "and"? It's really just a predicate built out of predicates:

```
;; A (new and% PredicateRunner PredicateRunner)
;; implements PredicateRunner
(define-class and%
  (fields q1 q2)
  ;; apply : Runner -> Boolean
  ;; Does the given runner satisfy q1 *and* q2?
  (check-expect ((new and% f? young?) . apply roberta) true)
  (check-expect ((new and% f? young?) . apply bobby) false)
  (check-expect ((new and% f? young?) . apply johnny) false)
  (define (apply r)
    (and (this . q1 . apply r)
         (this . q2 . apply r))))
```

The test cases make use of the following definitions for succinctness:

```
(define f? (new is-female%))
(define young? (new in-age% 18 34))
```

Now we can construct the predicate "is the given runner 18-34, a woman, and did she finish in under 3 hours and 35 minutes?":

```
(new and%
     (new in-age% 18 34)
     (new and% (new is-female%) (new faster% 3 35)))
```

And from here, it's easy to express the predicate of a row (assuming you can define or%):

```
;; A (new qrow% Natural^6) implements PredicateRunner
(define-class qrow%
  (fields lo hi m-hrs m-mins f-hrs f-mins)
  ;; apply : Runner -> Boolean
  ;; Does given runner satifisy conditions of this row?
  (check-expect ((new qrow% 18 34 3 05 3 35) . apply roberta) true)
  (check-expect ((new qrow% 18 34 3 05 3 35) . apply bobby) true)
  (check-expect ((new qrow% 18 34 3 05 3 35) . apply johnny) false)
  (define (apply r)
    ((new and%
          (new in-age% (this . lo) (this . hi))
          (new or%
               (new and%
                    (new is-male%)
                    (new faster% (this . m-hrs) (this . m-mins)))
               (new and%
                    (new is-female%)
                    (new faster% (this . f-hrs) (this . f-mins)))))
     . apply r)))
```

Only the last row, which asks about being 80+ is a different (here, we assume you can define `older%` with the obvious meaning):

```
;; A (new qrow% Natural^5) implements PredicateRunner
(define-class lastrow%
  (fields lo m-hrs m-mins f-hrs f-mins)
  ;; apply : Runner -> Boolean
  ;; Does given runner satifisy conditions of this last row?
  (check-expect ((new lastrow% 80 4 55 5 10) . apply roberta) false)
  (check-expect ((new lastrow% 80 4 55 5 10) . apply bobby) false)
  (check-expect ((new lastrow% 80 4 55 5 10) . apply johnny) false)
  (define (apply r)
    ((new and%
          (new older% (this . lo))
          (new or%
               (new and%
                    (new is-male%)
                    (new faster% (this . m-hrs) (this . m-mins)))
               (new and%
                    (new is-female%)
                    (new faster% (this . f-hrs) (this . f-mins)))))
     . apply r)))
```

And now the `qualify%` class is easy and exactly matches the structure of the BAA's table:

```
;; A (new qualify%) implements PredicateRunner
(define-class qualify%
  ;; apply : Runner -> Boolean
  ;; Does given runner qualify for next Boston Marathon?
  (check-expect ((new qualify%) . apply roberta) true)
  (check-expect ((new qualify%) . apply bobby) true)
  (check-expect ((new quality%) . apply johnny) true)
  (define (apply r)
    (or ((new qrow% 18 34 3 05 3 35) . apply r)
        ((new qrow% 35 39 3 10 3 40) . apply r)
        ((new qrow% 40 44 3 15 3 45) . apply r)
        ((new qrow% 45 49 3 25 3 55) . apply r)
        ((new qrow% 50 54 3 30 4 00) . apply r)
        ((new qrow% 55 59 3 40 4 10) . apply r)
        ((new qrow% 60 64 3 55 4 25) . apply r)
        ((new qrow% 65 69 4 10 4 40) . apply r)
        ((new qrow% 70 74 4 25 4 55) . apply r)
        ((new qrow% 75 79 4 40 5 10) . apply r)
        ((new lastrow% 80 4 55 5 10) . apply r))))
```

Now it's easy to filter the list of runners for those qualifying for next year's race. Unfortunately, Johnny doesn't make the cut, but considering he's 97 and won the 1935

and 1945 Boston Marathon, perhaps the BAA will cut him some slack.
Examples:

```
> (rs . filter (new qualify%) . filter (new is-male%))
(new cons% (new runner% "Cheruiyot" 33 8 7620 (new male%)) (new
mt%))
> (rs . filter (new qualify%) . filter (new is-female%))
(new cons% (new runner% "Gibb" 23 121 12000 (new female%)) (new
mt%))
```

11.2   Parametric data and separation of concerns

One thing that is sure to stand out from the last section is we define a class of objects
`LoR` rather than `[List Runner]`, despite having previously honed a nice parametric
list library. Why?

Consider the methods we were developing, such as

```
;; fast : -> LoR
;; Runners in this list with times less than 3 hours.

;; old : -> LoR
;; Runners in this list with age more than 50 years.
```

These methods require that this list be a list of runners and will probably not work if
this list is a list of anything else. There's no mechanism stopping us from defining `fast`
and `old` within a generic `[List X]` library. We could define these runner-specific
methods within the general library and state the assumption that this list is a `[List
Runner]`. But this is ultimately a losing strategy as we must pollute our list library
with element-specific methods, which seem out of place in a general-purpose library,
may have conflicts, and whose requirements can certainly grow beyond any anticipated
limits.

Instead, we should interpret `[List X]` as requiring all code to be totally inde-
pendent of X, in which case methods specific to `[List Runner]` are definitely not
acceptable. And at first glance, it may appear we've painted ourselves into a corner:
how can we perform runner-specific computations if we can't be runner-specific in the
definition of list methods?

The answer is actually enabled by our abstraction of the runner-specific computa-
tion (the predicate) from the list-specific computation (the filter). Once these computa-
tions are disentangled, it's easy to design re-usable libraries for lists and runners.

First, let's focus on the `PredicateRunner` interface:

```
;; A PredicateRunner implements
;; - apply : Runner -> Boolean
;;   Does this predicate apply to given runner?
```

This interface definition can be parameterized by the class of objects in the domain
of the predicate:

```
;; A [Predicate X] implements
;; - apply : X -> Boolean
;;    Does this predicate apply to given X?
```

And obviously plugging `Runner` in for X in `[Predicate Runner]` results in the original interface definition.

Next, let's focus on the `filter` method in the LoR interface:

```
;; filter : PredicateRunner -> LoR
;; Runners in this list satisfying given predicate.
```

By abstracting over the class of elements, we obtain a more general contract for `filter` that only requires the domain of the predicate match the class of elements:

```
;; filter : [Predicate X] -> [List X]
;; Elements in this list satisfying given predicate.
```

On further inspection, we can also see many of our runner-specific utilities are actually not runner-specific at all. For example:

```
;; A (new and% PredicateRunner PredicateRunner)
;; implements PredicateRunner
```

can be given the more general contract:

```
;; A (new and% [Predicate X] [Predicate X])
;; implements [Predicate X]
```

11.3 Functions as objects: abstracting comparisons

Continuing with the Boston Marathon example, let's develop another series of methods.

For sporting events like the Boston Marathon, we're often interesting in the ranking of the participants. With that in mind, let's develop a couple sorting methods. The BAA would like to list the finishers in ascending order of time. They'd also like to like to list runners in alphabetic order by name. They may also have some other orders in the future, but let's start with `sort-time` and `sort-name`:

LoR interface

```
;; sort-time : -> LoR
;; Sort this list of runners in ascending order of finish times.

;; sort-name : -> LoR
;; Sort this list of runners in lexicographic order of names.
```

Let's start by defining some expected results examples:

```
(define rs-sort-time
  (new cons% bobby (new cons% roberta (new cons% johnny mt))))

(define rs-sort-name ; Coincidentally the same as above.
  (new cons% bobby (new cons% roberta (new cons% johnny mt))))
```

A straightforward structural recursive design leads us to:

LoR interface

```
;; insert-time : Runner -> LoR
;; Insert given runner into this sorted list of runners by time.

;; insert-name : Runner -> LoR
;; Insert given runner into this sorted list of runners by name.
```

mt%

```
(check-expect (mt . sort-time) mt)
(define (sort-time) this)

(check-expect (mt . sort-name) mt)
(define (sort-name) this)

(check-expect (mt . insert-time bobby) (new cons% bobby mt))
(define (insert-time r) (new cons% r this))

(check-expect (mt . insert-name bobby) (new cons% bobby mt))
(define (insert-name r) (new cons% r this))
```

cons%

```
(check-expect (rs . sort-time) rs-sort-time)
(define (sort-time)
  (this . rest . sort-time . insert-time (this . first)))

(check-expect (rs . sort-name) rs-sort-name)
(define (sort-name)
  (this . rest sort-name . insert-name (this . first)))

(check-expect ((new cons% bobby mt) . insert-time johnny)
              (new cons% johnny (new cons% bobby mt)))
(define (insert-time r)
  (cond [(< (r . time) (this . first . time))
         (new cons% r this)]
        [else
         (new cons%
              (this . first)
```

```
                  (this . rest . insert-time r))])))

(check-expect ((new cons% bobby mt) . insert-name johnny)
              (new cons% johnny (new cons% bobby mt)))
(define (insert-name r)
  (cond [(string-<? (r . name) (this . first . name))
         (new cons% r this)]
        [else
         (new cons%
              (this . first)
              (this . rest . insert-name r))])))
```

At this point, it's clear much of the code between `sort-time` and `sort-name` has been duplicated. By following the same approach as the last section, we can abstract these two methods by parameterizing the common code by functional objects that represent the differences. Here though, the crucial difference is not a predicate of a single runner, but a comparison between two runners. For `sort-time`, the comparison is "is the first runner's finish time less than the second's?" and for `sort-name`, the comparison is "does the first runner's name come before the second's, alphabetically?". We can codify both using a comparison function object:

```
;; A RunnerComparison implements:
;; - compare : Runner Runner -> Runner
;;   Does the first runner compare "better" than the second?
```

And the two comparisons are:

```
(define-class faster% ; implements RunnerComparison
  ;; Is the first runner faster than the second?
  (define (compare r1 r2)
    (< (r1 . time) (r2 . time))))

(define-class alpha% ; implements RunnerComparison
  ;; Does the first runner's name come before the second?
  (define (compare r1 r2)
    (string-<? (r1 . name) (r2 . name))))
```

The abstraction of `sort-time` and `sort-name` is:

LoR interface

```
;; sort : RunnerComparison -> LoR
;; Sort this list of runners in ascending order by comparison.

;; insert : Runner RunnerComparison -> LoR
;; Insert given runner into this list of runners sorted by comparison.
```

mt%

```
(define (sort c) this)
(define (insert r c) (new cons% r this))
```

cons%

```
(define (sort c)
  (this . rest . sort c . insert (this . first) c))

(define (insert r c)
  (cond [(c . compare r (this . first))
         (new cons% r this)]
        [else
         (new cons%
              (this . first)
              (this . rest . insert r c))]))
```

To recreate the original methods, we can lift the application of `sort` to the appropriate function object into the `list%` abstract class:

list%

```
(define (sort-time) (sort (new faster%)))
(define (sort-name) (sort (new alpha%)))
```

And, of course, we're not done until we verify that our original tests pass.

11.4   Functions as data as objects: infinite sequences

Functions are a useful abstraction mechanism, but they are also a useful kind of data definition. In this section, we explore the use of functional objects to represent infinite data.

```
;; An ISequence implements
;; - term : Natural -> Number
;;   compute the i^th element of this sequence

(define-class even%
  ;; term : Natural -> Natural
  ;; compute the i^th even number
  (check-expect ((new even%) . term 7) 14)
  (define (term i)
    (* i 2)))

(define-class odd%
  ;; term : Natural -> Natural
  ;; compute the i^th odd number
  (check-expect ((new odd%) . term 7) 15)
```

```
    (define (term i)
      (add1 (* i 2)))))

(define-class even-series%
  ;; term : Natural -> Natural
  ;; sum up the first n even numbers
  (check-expect ((new even-series%) . term 10) 90)
  (define (term n)
    (cond [(zero? n) 0]
          [else (+ ((new even%) . term (sub1 n))
                   (term (sub1 n)))])))

(define-class odd-series%
  ;; term : Natural -> Natural
  ;; sum up the first n odd numbers
  (check-expect ((new odd-series%) . term 10) 100)
  (define (term n)
    (cond [(zero? n) 0]
          [else (+ ((new odd%) . term (sub1 n))
                   (term (sub1 n)))])))
```

We can now apply the process for designing abstractions with functions-as-values from *HtDP* Section 22.2 adapted for functions-as-objects.

```
(define-class series%
  ;; Σ : ISequence -> ISequence
  ;; Make sequence summing up first n numbers of sequence
  (check-expect ((new series%) . Σ (new even%) . term 10) 90)
  (check-expect ((new series%) . Σ (new odd%) . term 10) 100)
  (define (Σ seq)
    (local [(define-class s%
              (define (term n)
                (cond [(zero? n) 0]
                      [else (+ (seq . term (sub1 n))
                               (term (sub1 n)))])))]
      (new s%))))
```

11.5 Exercises

11.5.1 Functional programming with objects

One perspective that unifies the paradigms of programming with functions and programming with objects is to view a "function" as an object that understands a method called `apply`. With that in mind, we can define an interface for functions-as-objects:

```
;; A [Fun X Y] implements:
```

```
;; apply : X -> Y
;; Apply this function to the given input.
```

Here we are representing a X -> Y function as an object that has an apply method that consumes an X and produces a Y.

- Design a class that wraps a real function with contract (X -> Y) to implement [Fun X Y].

- Using your wrapper class, construct the objects representing the "add 1" function and "sub 1" function.

- Another useful operation on functions is composition. Here is the interface for a ∘ method that composes two functions represented as objects:

  ```
  ;; [Fun X Y] also implements:
  ;; ∘ : [Fun Y Z] -> [Fun X Z]
  ;; Produce a function that applies this function to its input,
  ;; then applies the given function to that result.
  ```

  For example, if addone and subone refer to the objects you constructed in part 2, the following check should succeed:

  ```
  (check-expect ((addone . ∘ subone) . apply 5) 5)
  (check-expect ((addone . ∘ addone) . apply 5) 7)
  ```

  Implement the ∘ method for your wrapper class.

11.5.2   Lists and functional objects

Revisit your solution to the section 13.7.1 exercise. Revise the interface to replace any uses of function inputs with function object inputs and carry out the necessary changes to your design of the implementation.

11.5.3   Searching JSON with String predicates

Suppose you have a large library of string predicates, which implement the [Predicate String] interface. You can put these predicates to use in searching large collections of JSON data (section 6.6.2). To accomodate this, design the following method for JSON objects:

```
;; Find the first string in this JSON value satisfying
;; given predicate, or #f if there's no such string.
;; find : [Predicate String] -> String or #f
```

Chapter 12

Overriding

Is the test really on the 22nd? -- yes.

Inheritance and overriding

  Why do we use inheritance?
    1. Abstracting common code
    2. Managing the type system (in other languages)
    3. Extend existing system -- often overused, but very useful

  Example: points
    color-point% extends point%
      Use overriding to implement mv correctly for color-point%
    color-point% contains point%
      1. Class relationships aren't explicit
      2. Must manually delegate every inherited behavior

  Use mutation?
  Use update methods?
  Use copying + mutation?
    Performance analysis is hard and is best done empirically

  Dispatch
    Call the method most specific to the _object_
    e.g. double-move calls mv which dispatches to the appropriate method
body

  Example: worlds
    big-bang expects many behaviors that we usually don't want to spec-
ify

  super to dispatch one level up

159

Jumping >1 level would very often be a bad idea and break modularity

With inheritance, data definitions are always _open_
   e.g. A Point is one of
          - (point% Num Num)
          - (color-point% Color Num Num)
          - ...

Overriding lets you parametrize behavior of methods by supplying new behavior
   Powerful, just like higher-order functions in FP
   But power comes with responsibility: must respect behavioral subtyping!
   Overriding in Java:

```java
class Point {
    Integer x;
    Integer y;

    public Point(Integer x, Integer y) {
        this.x = x;
        this.y = y;
    }

    public Point mv(Integer dx, Integer dy) {
        return new Point(this.x + dx, this.y + dy);
    }

    public Point double_move(Integer dxy) {
        return this.mv(dxy,dxy);
    }

}

class CPoint extends Point {
    String color;

    public CPoint(String color, Integer x, Integer y) {
        super(x,y);
        this.color = color;
    }

    public CPoint mv(Integer dx, Integer dy) {
        return new CPoint(this.color, this.x-dx, this.y-dy);
    }

}
```

Overriding in `class/4`:

```
#lang class/4

;; A Point is (point% Number Number)
;; and implements
;; mv : Number Number -> Point
;; moves the point by this much in each direction
;; double-move : Number -> Point
;; move both x and y by the given amount
(define-class point%
  (fields x y)
  (define (mv dx dy)
    (point% (+ dx (send this x))
            (+ dy (send this y))))
  (define (double-move dxy)
    (mv dxy dxy))

  (check-expect (send (point% 1 2) mv 3 3) (point% 4 5))
  (check-expect (send (point% 1 2) double-move 3) (point% 4 5))
  )

(define-class color-point%
  (super point%)
  (fields color)
  (define (mv dx dy)
    (color-point% (send this color)
                  (+ dx (send this x))
                  (+ dy (send this y)))))

(check-expect (send (color-point% "red" 1 2) mv 2 2)
              (color-point% "red" 3 4))
```

Delegation in `class/4`:

```
#lang racket



;; A Point is (point% Number Number)
;; and implements
;; mv : Number Number -> Point
;; moves the point by this much in each direction
(define-class point%
  (fields x y)
  (define (mv dx dy)
```

```
    (point% (+ dx (send this x))
            (+ dy (send this y))))
  (check-expect (send (point% 1 2) mv 3 3) (point% 4 5))
  )

(define-class color-point
  (fields color pt)
  (define (mv dx dy)
    (color-point% (send this color)
                  (send (send this pt) mv dx dy))))
```

A default World:

```
#lang class/4
(require class/universe 2htdp/image)

;; A DWorld is (default-world%)
;; and implements
;; on-tick : DWorld -> DWorld
;; to-draw : DWorld -> Scene
;; ...
(define-class default-world%
  (define (on-tick)
    this)
  (define (to-draw)
    (empty-scene 300 300
                 )
    )
  )

(define-class circle-world%
  (super default-world%)
  (define (to-draw)
    (overlay (circle 20 "solid" "red")
             (empty-scene 300 300))))

(big-bang (circle-world%))
```

Chapter 13

Extensional Equality

One of the most useful notions of equality is that of *extensional equality*, which means that two values are equal if and only if one can be substituted for the other in all possible contexts without affecting the meaning of a program. So for example 5 is extensionally equal to 5, even though in some sense there are two different 5s: the one written first and one written second. Likewise, if we have the following data representation of Cartesian points:

```
;; A Posn is a (new posn% Number Number)
(define-class posn%
  (fields x y))
```

then (new posn% 3 4) is extensionally equal to (new posn% 3 4). If we wanted to implement a method for positions that determined if this position was equal to a given one, it would be straightforward:

posn%

```
;; Is this posn the same as that posn?
;; =? : Posn -> Boolean
(define (=? that)
  (and (= (this . x) (that . x))
       (= (this . y) (that . y))))
```

For some kinds of values such as numbers, strings, and positions, it's easy to determine if two values of that kind are equal. But it's not possible to compare all kinds of values for extensional equality. For example, two functions are extensionally equal when for all equal inputs they produce equal outputs. Suppose we have two Number -> Number functions, f and g; how can determine if f is substitutable for g? We'll have to apply f to *all* possible numbers n and then ask if (= (f n) (g n)). But there are an infinite number of numbers, so we'll have to apply f and g an infinite number of times before being sure that f and g are the same.

By exactly the same reasoning, it follows that some kinds of objects cannot be compared for extensional equality since comparing two [Fun Number Number] objects involves checking (= (f . apply n) (g . apply n)) for all numbers n.

13.1   Structural equality for recursive unions

It's fairly simple to define an equality method for simple compound structures such as
posn%, however matters are complicated when dealing with the equality of a recursive
union data representation. For example, suppose we define lists of positions:

```
;; A ListPosn is one of
;; - (new mt%)
;; - (new cons% Posn ListPosn)
;; and implements
;; - =? : ListPosn -> Boolean
;; - Is this list the same as given list?
(define-class mt%)
(define-class cons%
  (fields first rest))
```

To see the problem consider implementing =? in the mt% class:

<div align="right">`mt%`</div>

```
;; Is this empty list the same as given list?
(define (=? that) ...)
```

While it's clear that when evaluating the ellided ... code that this is a mt% object,
it not at all clear what kind of object that. All that is known is that is a ListPosn.
Unfortunately the answer the method should produce is entirely dependent on the *kind*
of object that is. If that is a mt%, the answer is true; otherwise it is false.

At this point we have a couple options for how to proceed. One is to add to the
ListPosn interface methods that tell compute whether an object is a mt% object. The
solution, then is simple:

<div align="right">`mt%`</div>

```
;; Is this empty list the same as given list?
;; ListPosn -> Boolean
(define (=? that)
  (cond [(that . mt?) true]
        [(that . cons?) false]))
```

We could do likewise for the case of a cons%:

<div align="right">`cons%`</div>

```
;; Is this empty list the same as given list?
;; ListPosn -> Boolean
(define (=? that)
  (cond [(that . mt?) false]
        [(that . cons?)
         (and (this . first . =? (that . first))
              (this . rest . =? (that . rest)))]))
```

Notice here that we first check that the given list is a cons, then and only then do we access the first and second part of the cons to determine if they are the same. In the case of the first, we use the `posn% =?` method to determine if the two positions are equal; for the rest, we use the `ListPosn =?` method to determine if the rest of the lists are equal recursively.

This approach is simple, but leaks some details about the representation of lists could cause difficulties if we tried to extend the set of implementations of `ListPosn`. That's because we are mixing our style of programming; while we are using the class dispatch mechanism to carry out case analysis on `this`, we are using a type-predicate and an explicit `cond` to analyze the cases of `that`. To employ a consistent object-oriented style, we should use the dispatch mechanism to analyze `that` and eliminate the need for the `cond` expressions in this program.

Here's an alternative that makes use of a common pattern for writing methods that follow the "cross product of inputs" recipe, such as `=?`. First, observe that it's easy to write object specific equality methods. In other words, writing a method to determine if two empty lists are equal is easy; so is writing a method to determine if two cons lists are equal is easy:

`mt%`

```
;; Is this empty list the same as given empty?
;; Empty -> Boolean
(define (=-empty? that) true)
```

`cons%`

```
;; Is this cons list the same as given cons?
;; Cons -> Boolean
(define (=-cons? that)
  (and (this . first . =? (that . first))
       (this . rest . =? (that . rest))))
```

Currently these methods are variant specific—we can't send the `=-empty?` to a `cons%` object, but that's an easy restriction to lift: we simply need to lift `=-empty?` and `=-cons?` to the `ListPosn` interface and then implement `=-cons?` in the `mt%` class and `=-empty?` in the `cons%` class:

```
;; A ListPosn ... and implements:
;; - =-empty? : Empty -> Boolean
;;   Is this list the same as given empty list?
;; - =-cons? : Cons -> Boolean
;;   Is this list the same as given cons list?
```

`mt%`

```
;; =-cons? : Cons -> Boolean
;; Is this empty list the same as given cons list?
(define (=-cons? that) false)
```

cons%

```
;; =-empty? : Empty -> Boolean
;; Is this cons list the same as given empty list?
(define (=-empty? that) false)
```

Since an empty lists and a cons lists are never the same, the implementation of both of these methods is just `false`.

We are now in a position to revisit our definitions of `=?` for `cons%` and `mt%`. Let's reconsider `=?` in the `mt%` class:

cons%

```
;; =? : ListPosn -> Boolean
;; Is this empty list the same as given list?
(define (=? that) ...)
```

If we take an inventory of what is available in `...`, we have

- `this` - a `mt%` (and therefore, a `ListPosn`),

- `that` - a `ListPosn`,

- `(this . =-empty? Empty)` - a Boolean,

- `(this . =-cons? Cons)` - a Boolean (which is always `false`),

- `(that . =-empty? Empty)` - a Boolean,

- `(that . =-cons? Cons)` - a Boolean.

Now, let's consider how we could put each of these methods to use.

- `(this . =-empty? Empty)`

  We need to give an `Empty` argument, but there is only one thing that is definitely an `Empty` and that is `this`. However, `(this . =-empty? this)` doesn't compute anything useful—it's always `true`.

- `(this . =-cons? Cons)`

  This doesn't compute anything useful, it's always `false`. (Not to mention we have nothing that is sure to be a `Cons` to supply as an argument.)

- `(that . =-empty? Empty)`

  We need to give an `Empty` argument and the only thing we know to be `Empty` is `this`. What does `(that . =-empty? this)` compute?

- `(that . =-cons? Cons)`

  We need to give a `Cons` argument and there's nothing we know to be a `Cons` so we can't call this method safely.

Of all these possibilities, only one is possible or computes anything interesting and that's `(that . =-empty? Empty)`. So what does it do? Since it's a method invoked on `that`, the object system will dispatch to the appropriate method based on what kind of object `that` is. If it's a `cons%`, we're going to invoke the `=-empty?` method that always returns false, but since `that` is a `cons%` and `this` is a `mt%`, that is the desired result. On the other hand, if `that` is a `mt%`, the `=-empty?` method that always returns `true` is invoked. But in that case, both `this` and `that` are `mt%`, so again we've produced the desired result. By similar reasoning, we can define the correct implementation of `=?` for `cons%`. Putting it all together, we get:

```
(define-class mt%
  (define (=? that) (that . =-empty? this))
  (define (=-empty? that) true)
  (define (=-cons? that) false))

(define-class cons%
  (fields first rest)
  (define (=? that) (that . =-cons? this))
  (define (=-empty? that) false)
  (define (=-cons? that)
    (and (this . first . =? (that . first))
         (this . rest . =? (that . rest)))))
```

Notice that this correctly defines equality for lists of positions and achieves our goal of programming in an interface-oriented style that has no explicit `cond` expressions performing case analysis on the kind of an object.

This technique which is generally applicable to methods of a recursive union that take arguments of that same recursive union and that needs to consider all possible scenarios for what `this` and `that` could be. The technique is often called *double-dispatch* since it does dispatch twice, once on the receiver and once on the argument, in order to achieve a complete case analysis.

This technique also gives rise to a completely mechanical, i.e. no thought involved, approach to implementing structural equality for recursive unions. If you have a union U defined as the union of variants `V1`, `V2`, ..., `Vn`, then the approach dictates

```
;; A U ... implements
;; =? : U -> Boolean
;; =-V1? : V1 -> Boolean
;; =-V2? : V2 -> Boolean
;; ...
;; =-Vn? : Vn -> Boolean
```

Where for each `Vi`, you write:

<div align="right">

`Vi`

</div>

```
;; =? : U -> Boolean
(define (=? that) (that . =-Vi? this))
```

```
;; =-Vi? : Vi -> Boolean
;; Is this Vi same as that Vi?
(define (=-Vi? that)
  ;; structural recur on each component,
  ;; equal if all parts are equal.
  ...)

;; One method for each Vj where j≠i.
;; Is this Vi same as that Vj?  No.
(define (=-Vj? that) false)
...
```

While this approach is mechanical, it's involves writing a lot of methods (can you construct a formulae for determining the number of method definitions you need if there are *i* variants in a union?) and can quickly become unwieldy. Luckily we can combat the problem throug the use of inheritence and overriding.

13.2 Abstracting equality with double dispatch

If you have a union with *i* variants, each variant will need to define *i-1* `=-Vj?` methods for each *j*≠*i*, and each of these methods will produce `false`. Instead of defining all of these methods in the variant, we can lift all *i* `=-Vi?` methods to an abstract class and in each case, return `false`. That behavior is *close* to the right thing, but not totally correct. In particular, for each of the `Vi`, the method specific to that variant, `=-Vi?` will be wrong since we want to structurally recur, not just return `false` blindly. To fix things up is easy though: simply override the `=-Vi?` method in the `Vi` implementation.

To make an example concrete, here is how to abstract the structural equality method of `ListPosn` using this approach:

```
(define-class alist%
  (define (=-empty? that) false)
  (define (=-cons? that) false))

(define-class mt%
  (super alist%)
  (define (=? that) (that . =-empty? this))
  (define (=-empty? that) true))

(define-class cons%
  (super alist%)
  (fields first rest)
  (define (=? that) (that . =-cons? this))
  (define (=-cons? that)
    (and (this . first . =? (that . first))
         (this . rest . =? (that . rest)))))
```

Because there are only two variants for a list, the overall size of the code didn't shrink (in fact it grew slightly longer), however the approach pays off when more variants are invovled. Each class now only defines behavior specific to that kind of object. If we had 100 variants, adding a 101th invaraint would only involving adding one method to `alist%` and two methods to the 101th variant. That's it. (Compare this to what's required if you can't use inheritance and overriding.) Importantly, such a design is extensible: to add new variants we have to modify the abstract class and the new variant; we don't have to edit the one hundred existing implementations.

13.3   Equality over interpretation

It is often the case that two peices of data are considered equal if and only if they are structurally equal, i.e. the data is comprised of the same thing. However, that is not the case when there are multiple representations of the same information. For example, consider the following data definition for a *set of numbers*, represented as a list of potentially non-unique elements:

```
;; A [SetNumber X] is one of:
;; - (new mt-set%)
;; - (new cons-set% Number SetNumber)
;; Interp: as the unordered, unique elements of this list.
```

So for example, the following two sets represents the same information but are structurally different:

```
(new cons-set% 1 (new cons-set% 2 (new mt-set%)))
(new cons-set% 2 (new cons-set% 1 (new mt-set%)))
```

So if we had instead used `cons%` and `mt%`, the `=?` method would produce `false` for these objects.

How can we define an equality method for sets that respects the interpretation of sets so that two sets are equal if and only if they represent the same information?

Let's start by defining set equality the way a mathematician would, which is that two sets are equal if one is a subset of the other and *vice versa*. This definition can be lifted to an abstract class for sets:

```
(define-class aset%
  (define (=? that)
    (and (this . ⊆ that)
         (that . ⊆ this))))
```

Now we need to define ⊆:

`mt-set%`

```
;; ⊆ : SetNumber -> Boolean
;; Is this empty set a subset of that?
;; The empty set is a subset of all sets.
(define (⊆ that) true)
```

cons-set%

```
;; ⊆ : SetNumber -> Boolean
;; Is this non-empty set a subset of that?
;; A non-empty set is a subset if that contains the first element
;; and the rest is a subset of that.
(define (⊆ that)
  (and (that . ∋ (this . first))
       (this . rest . ⊆ that)))
```

Finally, all that remains is ∋, pronounced "contains":

mt-set%

```
;; ∋ : Number -> Boolean
;; Does this empty set contain given number?
(define (∋ n) false)
```

cons-set%

```
;; ∋ : Number -> Boolean
;; Does this non-empty set contain given number?
(define (∋ n)
  (or (= n (this . first))
      (this . rest . ∋ n)))
```

13.4   Detatching objects from interpretation

In the previous section, we defined a =? method for sets that respected the set intpreta-
tion of objects. Since the set and list interpretation of equality is different, we needed
different classes of objects to represent lists and sets. But fundamentally, we chose the
same representation (namely lists) and our methods interpreted this representation dif-
ferently. Is it possible to instead have a single representation that could be interpreted
both as a list or a set? If we are to accomplish this, it cannot be through the use of =?
method, since there can only be one =? method per object. Instead what we would like
to do is define, external to the list classes, a notion of equivalence:

```
;; An [Equiv X] implements
;; - apply X X -> Boolean
;;   Are given Xs equal?
```

Moreover, an [Equiv X] needs to be reflexive, transitive, and symmetric, i.e. if ≡
is an [Equiv X], it must be the case that:

- (≡ . apply x x) is true.

- (≡ . apply x y) and (≡ . apply y z) implies (≡ . apply x z)

- (≡ . apply x y) implies (≡ . apply y x),

For example, here is the implementation of the usual equivalence relation on numbers:

```
;; A (new eqv-number%) implements [Equiv Number].
(define-class eqv-number%
  (define (apply n m) (= n m)))
```

Here is another equivalence relation, but it equates 5 with 10:

```
;; A (new eqv-number-mod-5%) implements [Equiv Number].
(define-class eqv-number-mod-5%
  (define (apply n m) (= (modulo n 5) (modulo m 5))))
```

13.5 Multiple Representations

```
;; A Posn implements IPosn

;; An IPosn implements
;; x : -> Number
;; y : -> Number
  ;; =? : IPosn -> Bool
  ;; is the given posn the same as this one?

;; A Posn2 is a (posn2% Number Number)
(define-class posn2%
  (fields ls)
  (constructor (x y)
               (fields (list x y)))
  (define (x) (first (field ls)))
  (define (y) (second (field ls)))
  (define (=? p)
    (and (= (x) (send p x))
         (= (y) (send p y)))))
```

Now we can compare the two different kinds of posns and everything works properly.

What about multiple representations for lists of posns?

```
#lang class/3

;; A Posn implements IPosn.

;; An IPosn implements:
;;
;; x : -> Number
```

```
;; y : -> Number
;; =? : IPosn -> Boolean

;; A Posn2 is a (posn2% Number Number).
;; implement IPosn.
(define-class posn2%
 (fields ls)
 (constructor (x0 y0)
   (fields (list x0 y0)))

 (define (x)
   (first (field ls)))

 (define (y)
   (second (field ls)))

 (check-expect (send (posn2% 3 4) =? (posn2% 3 4)) true)
 (check-expect (send (posn2% 3 4) =? (posn2% 3 5)) false)
 (define (=? p)
   (and (= (send this x) (send p x))
        (= (send this y) (send p y))))))


(check-expect (send (posn% 1 2) =? (posn2% 1 2)) true)
(check-expect (send (posn% 1 2) =? (posn2% 1 1)) false)


;; A Posn1 is a (posn% Number Number).
;; implements IPosn.
(define-class posn%
 (fields x y)


 ;; Posn -> Boolean
 ;; Is the given posn the same as this?
 (check-expect (send (posn% 3 4) =? (posn% 3 4)) true)
 (check-expect (send (posn% 3 4) =? (posn% 3 5)) false)
 (define (=? p)
   (and (equal? (field x) (send p x))
        (equal? (field y) (send p y))))))

;; A LoP is one of:
;; - (mt%)
;; - (cons% Posn LoP)
;; implements ILoP
```

```
;; An ILoP implements:
;;
;; first : -> Posn
;; rest : -> ILoP
;; empty? : -> Boolean

(define the-real-empty? empty?)
(define the-real-first first)
(define the-real-rest rest)

(define-class list%
 (fields ls)
 (define (empty?)
   (the-real-empty? (field ls)))

 (define (first)
   (the-real-first (field ls)))

 (define (rest)
   (list% (the-real-rest (field ls))))

 (define (=? lop)
   (cond [(send this empty?) (send lop empty?)]
         [else
          (and (not (send lop empty?))
               (send (send this first) =? (send lop first))
               (send (send this rest) =? (send lop rest)))])))



;; A [Listof X] is one of:
;; - (mt%)
;; - (cons% X [Listof X])
;; where X implements
;; =? : X -> Boolean

(define-class mt%

 ;; [Listof X] -> Boolean
 ;; Is the given list of posns the same as this?
 (define (=? lop)
```

```
      (send lop empty?))

  (define (empty?)
    true))

(define-class cons%
 (fields first rest)

 (define (=? lop)
   (and (not (send lop empty?))
        (send (field first) =? (send lop first))
        (send (field rest) =? (send lop rest))))

 (define (empty?)
   false))

(check-expect (send (mt%) =? (mt%)) true)
(check-expect (send (cons% (posn% 3 4) (mt%)) =? (cons% (posn% 3 4) (mt%)))
              true)
(check-expect (send (cons% (posn% 3 4) (mt%)) =?  (mt%))
              false)
(check-expect (send (cons% (posn% 3 4) (mt%)) =? (cons% (posn% 3 5) (mt%)))
              false)

(check-expect (send (list% empty) =? (list% empty)) true)
(check-expect (send (list% (list (posn% 3 4))) =? (list% (list (posn% 3 4))))
              true)
(check-expect (send (list% (list (posn% 3 4))) =? (list% (list)))
              false)
(check-expect (send (list% (list (posn% 3 4))) =? (list% (list (posn% 3 5))))
              false)

(check-expect (send (list% (list (posn% 3 4))) =? (cons% (posn2% 3 4) (mt%)))
              true)
```

Now we can make all of the appropriate combinations work together: different
kinds of lists with the same kind of posns, the same kind of lists with different kinds of
posns, and different kinds of lists with different kinds of posns.

13.6   Parameterized Data Defintions and Equality

Generalizing LoP to [Listof X].

```
;; A [Listof X] is one of:
;; - (mt%)
```

```
;; - (cons% X [Listof X])
```

But we have to change one more thing. Our `=?` method assumes that `X` implements an `=?` method themselves.

```
;; A [Listof X] is one of:
;; - (mt%)
;; - (cons% X [Listof X])
;; where X implements
;; =? : X -> Boolean
```

We could also change the signature of `=?` to take a comparison. But then we'd have to change all of our code. We've lifted a restriction, but only to things that can be compared for equality.

13.7 Exercises

13.7.1 Extensional equality JSON

Revisit your solution to the JSON representation problem (section 6.6.2) and develop a method for determining if a given JSON value is extensionally equal to another. Note that this method must respect the set interpretation of JSON objects.

Chapter 14

Visitors and Folds

14.1 The Visitor Pattern

The visitor pattern is a general design pattern that allows you to seperate data from functionality in an object-oriented style.

For instance, suppose you want to develop a library of ranges. Users of your library are going to want a bunch of different methods, and in principal, you can't possibly know or want to implement all of them. On the other hand, you may not want to expose the implementation details of *how* you chose to represent ranges.

The visitor pattern can help—it requires you to implement *one* method which accepts what we call a "visitor" that is then exposed to a view of the data. Any computation over shapes can be implemented as a visitor, so this one method is universal—no matter how people want to use your library, this one method is enough to ensure they can write whatever computation they want. What's better is that even if you change the representation of ranges, so long as you provide the same "view" of the data, everything will continue to work.

So here is our data definition for ranges:

```
; A Range is one of
;; - (new lo-range% Number Number)
;; Interp: represents the range between 'lo' and 'hi'
;;         including 'lo', but *not* including 'hi'

;; - (new hi-range% Number Number)
;; Interp: represents the range between 'lo' and 'hi'
;;         including 'hi', but *not* including 'lo'

;; - (new union-range% Range Range)
;; Interp: including all the numbers in both ranges

(define-class lo-range%
  (fields lo hi))
```

```
(define-class hi-range%
  (fields lo hi))

(define-class union-range%
  (fields left right))
```

We will add a single method to the interface for ranges:

```
;; The Range interface includes:
;; - visit : [RangeVisitor X] -> X
```

We haven't said what is in the [RangeVisitor X] interface, but the key idea is that something that implements a [RangeVisitor X] represents a computation over ranges that computes an X. So for example, if we wanted to compute where a number is included in a range, we would want to implement the [RangeVistitor Boolean] interface since that computation would produce a yes/no answer.

The idea, in general, of the visitor pattern is that the visitor will have a method for each variant of a union. And the method for a particular variant takes as many arguments as there are fields in that variant. In the case of a recursive union, the method takes the result of recursively visiting the data.

Under that guideline, the [RangeVisitor X] interface will contain 3 methods:

```
;; An [RangeVisitor X] implements:
;; lo-range : Number Number -> X
;; hi-range : Number Number -> X
;; union-range : X X -> X
```

Notice that the contracts for lo and hi-range match the contracts on the constructors for each variant, but rather than constructing a Range, we are computing an X. In the case of union-range, the method takes two inputs which are Xs, which represent the results of visiting the left and right ranges, respectively.

Now we need to implement the visit method in each of the Range classes, which will just invoke the appropriate method of the visitor on its data and recur where needed:

```
(define-class lo-range%
  (fields lo hi)

  (define (visit v)
    (v . lo-range (this . lo) (this . hi))))

(define-class hi-range%
  (fields lo hi)

  (define (visit v)
    (v . hi-range (this . lo) (this . hi))))
```

```
(define-class union-range%
  (fields left right)

  (define (visit v)
    (v . union-range (this . left . visit v)
                     (this . right . visit v))))
```

We've now established the visitor pattern. Let's actually construct a visitor that does something.

We forgot to implement the `in-range?` method, but no worries – we dont' need to edit our class definitions, we can just write a visitor that does the `in-range?` computation, which is an implementation of [`RangeVisitor Boolean`]:

```
;; An InRange? is an (in-range?% Number)
;; implements [RangeVisitor Boolean].

(define-class in-range%?
  (fields n)

  (define (lo-range lo hi)
    (and (>= (this . n) lo)
         (<  (this . n) hi)))

  (define (hi-range lo hi)
    (and (>  (this . n) lo)
         (<= (this . n) hi)))

  (define (union-range left right)
    (or left right)))
```

Now if we have our hands on a range and want to find out if a number is in the range, we just invoke the `visit` method with an instance of the `in-range?%` class:

```
(some-range-value . visit (in-range?% 5))   ;; is 5 in some-range-
value ?
```

14.2   Folds

[FIXME]

14.3   Generators

`generator-bad.rkt`

```
#lang class/2
(require 2htdp/image class/universe)

;; A World is (world% Generator Number)
;; and implements IWorld
(define-class world%
  (fields generator num)
  ;; to-draw : -> Scene
  (define (to-draw)
    (overlay
      (text (number->string (this . num)) 20 "black")
      (empty-scene 500 500)))
  ;; on-key : Key -> World
  (define (on-key k)
    (world% (this . generator)
            ((this . generator) . pick))))

;; A Generator is a (generator% [Listof Number])
;; and implements
;; pick : -> Number
;; produce a number to show that isn't in bad
(define-class generator%
  (fields bad)
  (define (pick)
    (local [(define x (random 10))]
      (cond [(member x (this . bad)) (pick)]
            [else x]))))
(check-expect (<= 0 ((generator% empty) . pick) 10) true)
(check-expect (= ((generator% (list 4)) . pick) 4) false)

(big-bang (world% (generator% empty) 0))
```

 generator-register.rkt

```
#lang class/2
(require 2htdp/image class/universe)

;; A World is (world% Generator Number)
;; and implements IWorld
(define-class world%
  (fields generator num)
  ;; to-draw : -> Scene
  (define (to-draw)
    (overlay
      (text (number->string (this . num)) 20 "black")
      (empty-scene 500 500)))
  ;; on-key : Key -> World
```

```
    (define (on-key k)
      (cond [(key=? k "x")
               (local [(define g (this . generator . add-bad (this . num)))]
                 (world% g (g . pick)))]
             [else
              (world% (this . generator)
                      (this . generator . pick))]))))

;; A Generator is a (generator% [Listof Number])
;; and implements
;; pick : -> Number
;; produce a number to show that isn't in bad
;; add-bad : Number -> Generator
;; produce a generator like that with an additional bad number
(define-class generator%
  (fields bad)
  (define (add-bad n)
    (generator% (cons n (this . bad))))
  (define (pick)
    (local [(define x (random 10))]
      (cond [(member x (this . bad)) (pick)]
            [else x]))))
(check-expect (<= 0 ((generator% empty) . pick) 10) true)
(check-expect (= ((generator% (list 4)) . pick) 4) false)
(check-expect (= (((generator% empty) . add-bad 4) . pick) 4) false)

(big-bang (world% (generator% empty) 0))
```

 generator-initial.rkt

```
#lang class/3
(require 2htdp/image class/universe)

;; A World is (world% Generator Number)
;; and implements IWorld
(define-class world%
  (fields generator num)
  ;; to-draw : -> Scene
  (define (to-draw)
    (overlay
     (text (number->string (this . num)) 20 "black")
     (empty-scene 500 500)))
  ;; on-key : Key -> World
  (define (on-key k)
    (cond [(key=? "x" k)
             (begin (this . generator . tell-bad)
                    (world% (this . generator)
```

```
                                   (this . generator . pick))))]
              [else
               (world% (this . generator)
                       (this . generator . pick))]))))

;; A Generator is a (generator% [Listof Number] Number)
;; interp: the list of bad numbers, and the last number picked
;; and implements
;; pick : -> Number
;; produce a number to show not in the list
;; tell-bad : ->
;; produces nothing
;; effect : changes the generator to add the last number picked to the bad list
(define-class generator%
  (fields bad last)
  (define (tell-bad)
    (set-field! bad (cons (this . last) (this . bad))))
  (define (pick)
    (local [(define rnd (random 10))]
      (cond [(member rnd (this . bad)) (this . pick)]
            [else (begin
                    (set-field! last rnd)
                    rnd)]))))
(check-expect (<= 0 ((generator% (list 2 4 6) 0) . pick) 100) true)

(big-bang (world% (generator% (list 2 4 6) 0) 0))

(check-expect (member ((generator% (list 2 4 6) 0) . pick)
                      (list 2 4 6))
              false)

(define (tell-bad-prop g)
  (local [(define picked (g . pick))]
    (begin (g . tell-bad)
           (not (= picked (g . pick))))))

(check-expect (tell-bad-prop (generator% (list 1 2 3) 0)) true)

 generator-mutate.rkt

#lang class/3
(require 2htdp/image class/universe)

;; A World is (world% Generator Number)
;; and implements IWorld
(define-class world%
  (fields generator num)
```

```
;; to-draw : -> Scene
(define (to-draw)
  (overlay
   (text (number->string (this . num)) 20 "black")
   (empty-scene 500 500)))
;; on-key : Key -> World
(define (on-key k)
  (cond [(key=? k "x")
         (world% (this . generator)
                 (this . generator . pick-bad))]
        [else
         (world% (this . generator)
                 (this . generator . pick))]))))


;; A Generator is a (generator% [Listof Number] Number)
;; interp: numbers not to pick, last number picked
;; and implements
;; pick : -> Number
;; produce a number to show that isn't in bad
;; pick-bad : -> Number
;; pick a number to show, and remember that the last one was bad
(define-class generator%
  (fields bad last)
  (define (pick-bad)
    (begin (set-field! bad (cons (this . last) (this . bad)))
           (pick)))
  (define (pick)
    (local [(define x (random 10))]
      (cond [(member x (this . bad)) (this . pick)]
            [else (begin (set-field! last x)
                         x)]))))

(check-expect (<= 0 ((generator% empty 0) . pick) 10) true)
(check-expect (= ((generator% (list 4) 0) . pick) 4) false)

(big-bang (world% (generator% empty 0) 0))
```

14.4 Exercises

14.4.1 Quick visits

This problem builds on the *quick lists* problem.

Here was the interface you should have implemented for lists using the *quick list* data structure that supports a fast `list-ref` method:

```
;; A [List X] implements
;; - cons : X -> [List X]
;;   Cons given element on to this list.
;; - first : -> X
;;   Get the first element of this list
;;   (only defined on non-empty lists).
;; - rest : -> [List X]
;;   Get the rest of this
;;   (only defined on non-empty lists).
;; - list-ref : Natural -> X
;;   Get the ith element of this list
;;   (only defined for lists of i+1 or more elements).
;; - length : -> Natural
;;   Compute the number of elements in this list.

;; empty is a [List X] for any X.
```

Make sure your quick list implementation is working and place it into a file named
"quick-lists.rkt". That file should provide one name, empty, by including the
following at the top of the file:

```
(provide empty)
```

In a file named "slow-lists.rkt" *re*-develop an implementation of the list in-
terface, but in the usual way as a recursive union of mt% and cons% classes. That file
should also provide empty by including the same line above at the top.

Finally, start a third file called "use-lists.rkt" that will make use of both kinds
of lists by including the following at the top of the file:

```
(require (prefix-in q: "quick-lists.rkt"))
(require (prefix-in s: "slow-lists.rkt"))
```

You now have two lists: q:empty and s:empty; both are represented in very dif-
ferent ways, but so long as you use them accoring to the list interface, they should be
indistinguishable.

Let's now revise the [List X] interface to include support for visitors:

```
;; A [List X] implements ...
;; - accept : [ListVisitor X Y] -> Y
;;   Accept given visitor and visit this list's data.

;; A [ListVisitor X Y] implements
;; - visit-mt : -> Y
;;   Visit an empty list.
;; - visit-cons : X [Listof X] -> Y
;;   Visit a cons lists.
```

Implement the revised [`List X`] interface in both `"quick-lists.rkt"` and `"slow-lists.rkt"`.

In `"use-lists.rkt"` you should be able to define particular visitors and have it work on *both* representations of lists. As an example, here is a list visitor that computes the length of a list:

```
;; A (new length%) implements [ListVisitor X Natural].
;; List visitor for computing the length of a list.
(define-class length%
  (define (visit-mt) 0)
  (define (visit-cons x r)
    (add1 (r . accept this))))

(define len (new length%))

(check-expect (q:empty . accept len) 0)
(check-expect (s:empty . accept len) 0)
(check-expect (q:empty . cons 'c . cons 'b . cons 'a . accept len) 3)
(check-expect (s:empty . cons 'c . cons 'b . cons 'a . accept len) 3)
```

And here's one for the sum of a list of numbers:

```
;; A (new sum%) implements [ListVisitor Number Number].
;; List visitor for computing the sum of a list of numbers.
(define-class sum%
  (define (visit-mt) 0)
  (define (visit-cons n r)
    (+ n (r . accept this))))

(define sum (new sum%))

(check-expect (q:empty . accept sum) 0)
(check-expect (s:empty . accept sum) 0)
(check-expect (q:empty . cons 3 . cons 4 . cons 7 . accept sum) 14)
(check-expect (s:empty . cons 3 . cons 4 . cons 7 . accept sum) 14)
```

Implement a [`ListVisitor X X`] named `reverse%` that reverses a list (note: you may need to implement a "helper" visitor that corresponds to the helper function you'd write for the `reverse` function). Note that this visitor will have to commit to produces either a quick list or a slow list, but it really doesn't really matter which... well, except for testing. So for example, let's say the reverse visitor produces slow lists. Then we would expect the following test to pass, assuming `reverse%` works as specified:

```
(define rev (new reverse%))

(check-expect (q:empty . accept rev) s:empty)
(check-expect (q:empty . cons 'c . cons 'b . cons 'a . accept rev)
              (s:empty . cons 'a . cons 'b . cons 'c))
```

Of course, this isn't ideal since our *test* is testing more than is actually required of `reverse%`. In particular, it should be perfectly acceptable for `reverse%` to produce quick lists without tests failing.

What's happening here is that `check-expect` is checking too much because it is not treating the objects it compares solely according to their interface. We will see how to fix this problem by defining an interface-respecting equality computation, but for now, just test as shown above.

Now to build some larger pieces with visitors. First, here's an interface definition for functional objects that represent functions from Xs to Ys. Such an object has a single method called `apply` that consumes an X and produces a Y:

```
;; A [Fun X Y] implements
;; - apply : X -> Y
;;    Apply this function to given x.
```

Now implement the following two visitors:

```
;; A (new filter% [Predicate X]) implements [ListVisitor X [List X]].
;; Filters visited list to produce a list of elements satisfying predicate.

;; A (new map% [Fun X Y]) implements [ListVisitor X [List Y]].
;; Maps visited list to produce a list of results of applying the function.
```

Implement at least one [`Fun Natural String`] and one [`Predicate String`] to use for testing `filter%` and `map%`.

14.4.2   Folds vs Visitors

We can also implements *folds* over lists, in both for both kinds of lists. Extend your implementation of lists (both kinds) to support the `fold` method:

```
;; A [List X] implements ...
;; - fold : [ListFold X Y] -> Y
;;    Accept given fold and process this list's data.

;; A [ListFold X Y] implements
;; - fold-mt : -> Y
;;    Process an empty list.
;; - fold-cons : X Y -> Y
;;    Process a cons lists.
```

Now revise your implementations of the `filter%` and `map%` to implement folds as well as visitors. Be sure to specify what interfaces they implement.

Finally, implement the class `list-ref%`:

```
;; A (new list-ref% Number) implements [ListVisitor X X]
;; Retrieves the element at the specified index.
```

Could you implement this using the `ListFold` interface? Which was more elegant for `map%` and `filter%`?

14.4.3 JSON visitor

Develop the visitor pattern for JSON values.

Design an equality visitor for JSON values.

Part IV

Invariants

Chapter 15

Invariants, Testing, and Abstraction Barriers

15.1 Invariants of Data Structures

Here's an interface for a sorted list of numbers.

```
#lang class/1
;; An ISorted implements
;; insert : Number -> Sorted
;; contains? : Number -> Boolean
;; ->list : -> [Listof Number]
;; empty? : -> Boolean

;; Invariant: The list is sorted in ascending order.

;; Precondition: the list must not be empty
;; max : -> Number
;; min : -> Number
```

How would we implement this interface?

We can simply adopt the recursive union style that we've already seen for implementing lists. Here we see the basic defintion as well as the implementation of the `contains?` method.

```
#lang class/1
;; A Sorted is one of
;; - (new smt%)
;; - (new scons% Number Sorted)
(define-class smt%
  (check-expect ((new smt%) . contains? 5) false)
  (define (contains? n)
```

```
    false))

(define-class scons%
  (fields first rest)
  (check-expect ((new scons% 5 (new smt%)) . contains? 5) true)
  (check-expect ((new scons% 5 (new smt%)) . contains? 7) false)
  (check-expect ((new scons% 5 (new scons% 7 (new smt%))) . contains? 3)
                false)
  (check-expect ((new scons% 5 (new scons% 7 (new smt%))) . contains? 9)
                false)
  (define (contains? n)
    (or (= n (field first))
        ((field rest) . contains? n))))
```

However, we can write a new implementation that uses our invariant to avoid check-ing the rest of the list when it isn't necessary.

```
(define (contains? n)
  (cond [(= n (field first)) true]
        [(< n (field first)) false]
        [else ((field rest) . contains? n)]))
```

Because the list is always sorted in ascending order, if n is less than the first ele-ment, it must be less than every other element, and therefore can't possibly be equal to any element in the list.

Now we can implement the remaining methods from the interface. First, insert

smt%

```
(check-expect ((new smt%) . insert 5)
              (new scons% 5 (new smt%)))
(define (insert n)
  (new scons% n (new smt%)))
```

scons%

```
(check-expect ((new scons% 5 (new smt%)) . insert 7)
              (new scons% 5 (new scons% 7 (new smt%))))
(check-expect ((new scons% 7 (new smt%)) . insert 5)
              (new scons% 5 (new scons% 7 (new smt%))))
(define (insert n)
  (cond [(< n (field first))
         (new scons% n this)]
        [else
         (new scons%
              (field first)
              ((field rest) . insert n))]))
```

Note that we don't have to look at the whole list to insert the elements. This is again a benefit of programming using the invariant that we have a sorted list.

Next, the `max` method. We don't have to do anything for the empty list, because we have a precondition that we can only call `max` when the list is non-empty.

scons%

```
(define real-max max)
(check-expect ((new scons% 5 (new smt%)) . max) 5)
(check-expect ((new scons% 5 (new scons% 7 (new smt%))) . max) 7)
(define (max)
  (cond [((field rest) . empty?) (field first)]
        [else ((field rest) . max)]))
```

Again, this implementation relies on our data structure invariant. To make this work, though, we need to implement `empty?`.

smt%

```
(check-expect ((new smt%) . empty?) true)
(define (empty?) true)
```

scons%

```
(check-expect ((new scons% 1 (new smt%)) . empty?) false)
(define (empty?) false)
```

The final two methods are similar. Again, we don't implement `min` in `smt%`, because of the precondition in the interface.

smt%

```
;; no min method
(define (->list) empty)
```

scons%

```
(define (min) (field first))
(define (->list)
  (cons (field first) ((field rest) . ->list)))
```

15.2 Properties of Programs and Randomized Testing

A *property* is a claim about the behavior of a program. Unit tests check particular, very specific properties, but often there are more general properties that we can state and check about programs.

Here's a property about our sorted list library, which we would like to be true:

∀ sls : ISorted . ∀ n : Number . ((sls . insert n) . contains? n)

How would we check this? We can check a few instances with unit tests, but this property makes a very strong claim. If we were working in ACL2, as in the Logic

and Computation class, we could provide a machine-checked proof of the property, verifying that it is true for every single `Sorted` and `Number`.

For something in between these two extremes, we can use *randomized testing*. This allows us to gain confidence that our property is true, with just a bit of programming effort.

First, we want to write a program that asks the question associated with this property.

```
;; Property: forall sorted lists and numbers, this predicate holds
;; insert-contains? : ISorted Number -> Boolean
(define (insert-contains? sls n)
  ((sls . insert n) . contains? n))
```

Now we make lots of randomly generated tests, and see if the predicate holds. First, let's build a random sorted list generator.

```
;; build-sorted : Nat (Nat -> Number) -> Sorted
(define (build-sorted i f)
  (cond [(zero? i) (new smt%)]
        [else
         (new scons%
              (f i)
              (build-sorted (sub1 i) f))]))
(build-sorted 5 (lambda (x) x))
```

Oh no! We broke the invariant. The `scons%` constructor allows you to break the invariant, and now all of our methods don't work. Fortunately, we can implement a fixed version that uses the `insert` method to maintain the sorted list invariant:

```
;; build-sorted : Nat (Nat -> Number) -> Sorted
(define (build-sorted i f)
  (cond [(zero? i) (new smt%)]
        [else
         ((build-sorted (sub1 i) f) . insert (f i))]))
(check-expect (build-sorted 3 (lambda (x) x))
              (new scons% 1 (new scons% 2 (new scons% 3 (new smt%)))))
```

Now `build-sorted` produces the correct answer, which we can easily verify at the Interactions window.

Using `build-sorted`, we can develop `random-sorted`, which generates a sorted list of random numbers.:

```
;; Nat -> Sorted
(define (random-sorted i)
  (build-sorted i (lambda (_) (random 100))))
```

Given these building blocks, we can write a test that checks our property.

```
(check-expect (insert-contains? (random-sorted 30) (random 50))
              true)
```

Every time we hit the Run button, we generate a random sorted list of numbers, and check if a particular random integer behaves appropriately when inserted into it. But if we could repeatedly check this property hundreds or thousands of times, it would be even more unlikely that our program violates the property. After all, we could have just gotten lucky.

First, we write a function to perform some action many times:

```
;; Nat (Any -> Any) -> 'done
;; run the function f i times
(define (do i f)
  (cond [(zero? i) 'done]
        [else (f (do (sub1 i) f))]))
```

Then we can run our test many times:

```
(do 1000
    (lambda (_)
      (check-expect (insert-contains? (random-sorted 30) (random 50))
                    true)))
```

When this says that we've passed 1000 tests, we'll be more sure that we've gotten our function right.

What if we change our property to this untrue statement?

```
;; Property: forall sorted lists and numbers, this predicate holds
;; insert-contains? : ISorted Number -> Boolean
(define (insert-contains? sls n)
  (sls . contains? n))
```

Now we get lots of test failures, but the problem is *not* in our implementation of sorted lists, it's in our property definition. If we had instead had a bug in our implementation, we would have similarly seen many failures. Thus, it isn't always possible to tell from a test failure, or even many failures, whether it's the code or the specification is wrong—you have to look at the test failure to check.

This is why it's extremely important to get your specifications (like contracts, data definitions, and interface definitions) correct. Your program can only be correct if they are.

15.3 Abstraction Barriers and Modules

Recall that in our original version of build-sorted, we saw that the scons% constructor allowed us to violate the invariant—it didn't check that the value provided for first was at least as small as the elements of rest. We would like to prevent clients

of our sorted list implementation from having access to this capability, so that we can be sure that our invariant is maintained.

To address this, we set up an *abstraction barrier*, preventing other people from seeing the `scons%` constructor. To create these barriers, we use a *module system*. We will consider our implementation of sorted lists to be one module, and we can add a simple specification to allow other modules to see parts of the implementation (but not all of it).

Modules in our languages are very simple—you've already written them. They start with `#lang class/N` and cover a whole file.

Here's the module implementing our sorted list, which we save in a file called `"sorted-list.rkt"`.

sorted-list.rkt

```
#lang class/1

;; ... all of the rest of the code ...

(define smt (new smt%))
(provide smt)
```

We've added two new pieces to this file. First, we define `smt` to be an instance of the empty sorted list. Then, we use `provide` to make `smt`, but not *any* other definition from our module, available to other modules.

Therefore, the *only* part of our code that the rest of the world can see is the `smt` value. To add new elements, the rest of the world has to use the `insert` method.

```
#lang class/1
(require "sorted-list.rkt")

(smt . insert 4)
```

Here, we've used `require`, which we've used to refer to libraries that come with DrRacket. However, we can specify the name of a file, and we get everything that the module in that file `provide`s, which here is just the `smt` definition. Everything else, such as the dangerous `scons%` constructor, is hidden, and our implementation of sorted lists can rely on its invariant.

15.4   Exercises

15.4.1   Quick Lists

Van Horn has always been underwhelmed by the fact that `list-ref` is such a slow operation when you're accessing elements deep down in a big list. Why should it take a million `rest`s just to get the millionth element?

To combat this drawback of an otherwise lovely data structure, the list, Van Horn has devised an idea for a new implementation of lists that would let you get the millionth element in about 20 operations. If his idea works, the `list-ref` operation will

take roughly *log(i)* steps to get the *i*th element. The other list operations, on the other hand, would remain more or less just as efficient as before; taking the rest of a list, for example, might take a few more steps to compute, but it would be some small constant number of extra steps. In the end, we'd have something that behaves just like a list, but with a much better `list-ref` operation.

Your task is to take Van Horn's idea and implement it. Since you'll be building a new kind of list data structure, let's first agree on the list interface we want:

```
;; A [List X] implements
;; - cons : X -> [List X]
;;   Cons given element on to this list
;; - first : -> X
;;   Get the first element of this list
;;   (only defined on non-empty lists)
;; - rest : -> [List X]
;;   Get the rest of this
;;   (only defined on non-empty lists)
;; - list-ref : Natural -> X
;;   Get the ith element of this list
;;   (only defined for lists of i+1 or more elements)
;; - length : -> Natural
;;   Compute the number of elements in this list

;; empty is a [List X] for any X.
```

In other words, you have to make an object named `empty` that implements the list interface above. Lists should work just like we're used to, so for example, these tests should all pass if `empty` is appropriately defined:

```
#lang class/1
(require "your-implementation-of-lists.rkt") ; provides empty

(define ls (empty . cons 'a . cons 'b . cons 'c . cons 'd . cons 'e))

(check-expect (empty . length) 0)
(check-expect (ls . length) 5)
(check-expect (ls . first) 'e)
(check-expect (ls . rest . first) 'd)
(check-expect (ls . rest . rest . first) 'c)
(check-expect (ls . rest . rest . rest . first) 'b)
(check-expect (ls . rest . rest . rest . rest . first) 'a)

(check-expect (ls . list-ref 0) 'e)
(check-expect (ls . list-ref 1) 'd)
(check-expect (ls . list-ref 2) 'c)
(check-expect (ls . list-ref 3) 'b)
(check-expect (ls . list-ref 4) 'a)
```

So now let's talk about Van Horn's idea.

Van Horn thinks if instead of representing a list as a "list of elements" you could do better by representing a list as a "forest of trees of elements". (A *forest* is just an arbitrarily long sequence of trees.) Moreover, the trees will get bigger and bigger as you go deeper into the forest, and every tree is full. (A *full tree* is a binary in which every node has a left and right subtree that are full and of the same.) For the moment, don't worry about *why* this makes `list-ref` fast—think about that after you've implemented Van Horn's idea.

So here are the key invariants of a *quick list*:

- A quick list is a forest of increasingly large full binary trees.

- With the possible exception of the first two trees, every successive tree is *strictly* larger.

Now that we have the invariant, let's talk about the operations and how they both can use and maintain the invariant.

First, `first`. Since the list must be non-empty, we know the forest has at least one tree, so we can get the first element of the list by getting "the first" element of the tree, which for quick lists, will be the top element.

Now, `length`. If the forest is empty, the list has length `0`. If a forest has a tree, the length of the list is the size of the tree plus the size of the rest of the forest. (It's useful to store the size of a tree separately from a tree so that you don't have to compute it every time you need it.)

The `list-ref` method works as follows: if the index is `0`, the list must be non-empty, so take the first element, i.e. the top element of the first tree in the forest. If the index is non-zero, there are two case: if it's less than the size of the first tree, the element is in that tree, so fetch it from the first tree. If it's larger, adjust the index, and look in the remaining trees of the forest.

To fetch an element from a tree: if the index is zero, the element is the top element. Otherwise, if the index is less than half the size, it's on the left side; if the index is greater than half, it's on the right. (You might do yourself a favor a develop `tree-ref` for full binary trees and get it working and thoroughly tested before attempting `list-ref`.)

These element-producing operations considered so far have used the invariant. Now let's turn to the list-producing operations which must maintain it.

When an element is `cons`ed, there are two cases to consider:

- If there at least two trees in the forest and the first two trees are the same size, then make a new tree out of these two and with the given element on top. Here it is pictorially; we are given a forest of full binary trees where the first two trees have the same height:
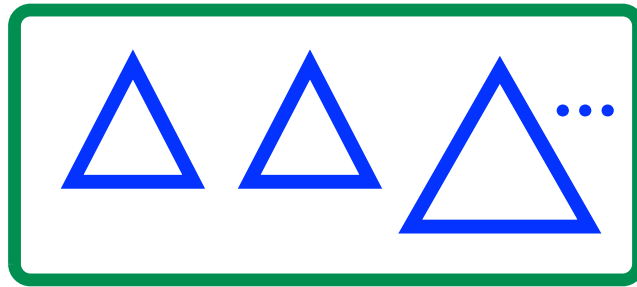
To cons on the new element, we make a node that contains the element and the first two trees as its left and right subtree:
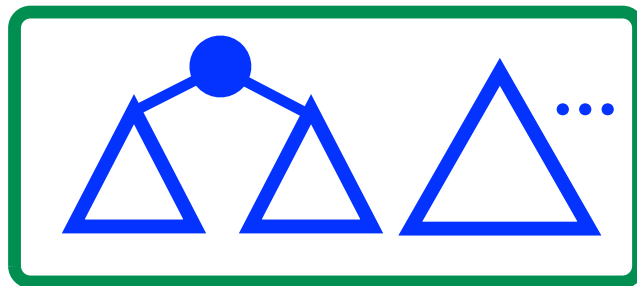
Notice how this first tree is necessarily full, since the first two trees were full and the same height; notice how this new first tree in the forest is at most as large as the second tree (previously the third tree). These two observations demonstrate that the invariant holds on the resulting forest, so `cons` really makes a quick list in this case.

- Otherwise, we know that the size of the trees in the forest is strictly increasing:
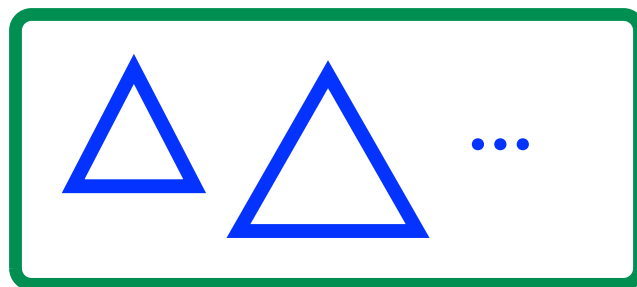
Therefore, we can just make a new tree with one element and make it the first tree in the forest:
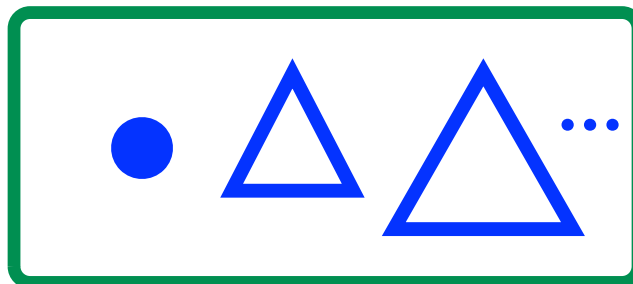
Notice how the one element tree is obviously full and that it is no larger than the (now) second tree in the forest, so the invariant holds in this case too.

To take the `rest` of a list, there must be at least one tree in the forest (since the list is non-empty):
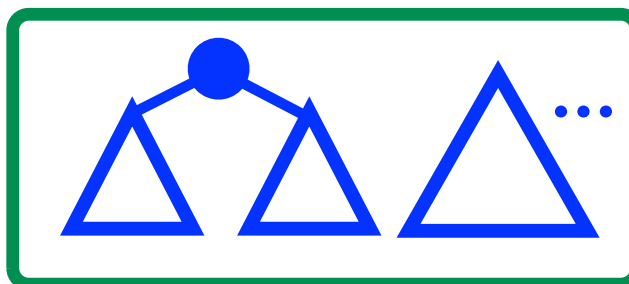
We want to split this tree into its left and right and make these the first two trees in the forest. The element that was on top is dropped on the floor and we're left with a representation of the rest of the list:
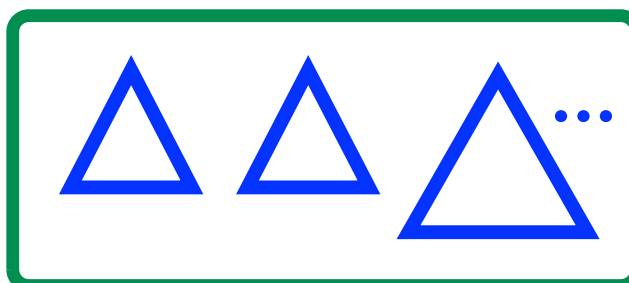
And that's that. When writing your code you want to make sure the invariants are always true. Good code should make this fact obvious; bad code, not so much.

This is a nice little exercise in data structure design and implementation, and although Van Horn *wishes* this were really his idea, he actually got it from reading a book by Chris Okasaki, who has designed a bunch of these kinds of data structures. Go forth, and may your `list-ref` never be slow again.

Chapter 16

Constructors

16.1 Canonical forms

Today we're going to look more at the concept of *invariants*. Invariants often let us write code that takes advantage of the fact that we know some property, the invariant, of our data. We saw this last class using sorted lists of numbers. Today we're going to examine a new example: fractions.

A fraction can be represented as a compound data that consists of two numbers representing the numerator and denominator:

```
;; A Fraction is a (new fraction% Integer Integer).
(define-class fraction%
  (fields numerator denominator))
```

The problem here is that we'd like to consider the fractions:

```
(new fraction% 1 2)
(new fraction% 2 4)
```

as representing the same number, namely 1/2, but these are different representations of the same information. The issue with fractions is a recurring issue we've seen with information that allows for multiple representations (sets are another example).

There are a couple approaches to solving this issue:

1. Represents information is some *canonical* way.

2. Codify the interpretation of data as a program.

The first approach basically eliminates the problem of multiple representations by picking a unique representation for any given piece of information. For example, with fractions, we might choose to represent all fractions in lowest terms. This means any fraction admits only a single representation and therefore any fractions which are interpreted as "the same" have exactly the same structure. (This approach is not always possible or feasible.)

The second approach requires us to write a program (a function, a method, etc.) that determines when two pieces of data are interpreted as representing the same information. For example, we could write a method that converts fractions to numbers and compares them for numerical equality; or we simplify the fraction to lowest terms and compare them structurally.

Along the lines of the second approach, let's consider adding the following method:

`fraction%`

```
;; to-number : -> Number
;; Convert this fraction to a number.
(define (to-number)
  (/ (this . numerator)
     (this . denominator)))
```

This method essentially embodies our interpretation of the `fraction%` class of data. It doesn't help with this issues:

```
(check-expect (new fraction% 1 2)
              (new fraction% 2 4))
```

But of course now we can write our tests to rely on this interpretation function:

```
(check-expect ((new fraction% 1 2) . to-number)
              ((new fraction% 2 4) . to-number))
```

But what if we wanted to go down the second route? We could define a method that computes a fraction in lowest terms:

`fraction%`

```
;; simplify : -> Fraction
;; Simplify a fraction to lowest terms.
(check-expect ((new fraction% 3 6) . simplify)
              (new fraction% 1 2))
```

We can use the `gcd` function to compute the greatest common denominator of the terms:

`fraction%`

```
(define (simplify)
  (new fraction%
       (/ (this . numerator)
          (gcd (this . numerator)
               (this . denominator)))
       (/ (this . denominator)
          (gcd (this . numerator)
               (this . denominator)))))
```

This allows us to structurally compare two fractions that have been simplified to lowest terms:

```
(check-expect ((new fraction% 3 6) . simplify)
              ((new fraction% 1 2) . simplify))
```

But it does not prevent us from constructing fractions that are not in lowest terms, which is what we were aiming for — we want it to be an invariant of `fraction%` objects that they are in their simplest form. One possibility is to define a *constructor function* that consumes a numerator and denominator and constructs a `fraction%` object in lowest terms:

```
;; fract-constructor : Number Number -> Fraction
;; construct a fraction in lowest terms.
(define (fract-constructor n d)
  (new fraction%
       (/ n (gcd n d))
       (/ d (gcd n d))))
```

So we are able to write a new function with the behavior we want and it establishes our invariant. That's good, but there are still some inconveniences:

- We have to write a function.

- We have to remember to use it everywhere in place of the constructor.

- We still have the `fraction%` constructor around, which allows users, including ourselves, to violate the invariant.

If we want to have a stronger guarantee that we maintain the lowest term invariant, we need a stronger mechanism for enforcing our discipline at construction-time. The idea is to allow arbitrary computation to occur between the call to a constructor and the initialization of an object. To enable this mechanism, we need to bump the language level up to `class/2`.

All `class/1` programs continue to work in `class/2`. The main difference is that we now the ability to write *constructors*.

`fraction%`

```
(constructor (n d)
  ;;...some expression that uses the fields form to return values
  ;;  for all of the fields...
  ...)
```

The `constructor` form can take any number of arguments and must use the `fields` to initialize each of the fields. If you leave off the constructor form, a default constructor is generated as:

```
(constructor (n d)
  (fields n d))
```

And in general if you have n fields, the defaults constructor looks like:

```
(constructor (field1 field2 ... fieldn)
  (fields field1 field2 ... fieldn))
```

But by writing our own constructor, we can insert computation to convert arguments in a canonical form. For our `fraction%` class, we can use the following code:

```
;; Number Number -> Fraction
(constructor (n d)
  (fields (/ n (gcd n d))
          (/ d (gcd n d))))
```

This code is used every time we have a `(new fraction% Number Number)` expression. Since this is the only way to construct a fraction, we know that *all fractions are represented in lowest terms*. It is simply impossible, through both error or malice, to construct an object that does not have this property.

Returning to our `simplify` method; we don't really need it any longer. (We could, if need be, re-write the code to take advantage of the invariant and give a correct implementation of `simplify` as `(define (simplify) this)`, since all fractions are already simplified.) Likewise, we no longer need the `fract-constructor` function.

Finally, we get to the point we wanted:

```
(check-expect (new fraction% 1 2)
              (new fraction% 2 4))
```

Q: Can you have multiple constructor?

A: No. We've been thinking about multiple constructors, but we don't have a strong story for them yet. Remember: you can always write functions and you can think of these as alternative constructors.

That brings up another feature in the `class/2` language — constructors and functions are treated more uniformly now: you may leave off the `new` keyword when constructing objects.
Examples:

```
> (define-class posn%
    (fields x y))

> (new posn% 2 3)
(object:posn% 2 3)
> (posn% 4 5)
(object:posn% 4 5)
```

Q: Can you have a different number of arguments to the constructor than to the number of fields?

A: Yes. There's no relation between the number of arguments to your constructor and the number of fields in the object being constructed.

One thing to note is that printing values has changed. You'll notice that `fraction%` values no longer print as `(new fraction% Number Number)`, but instead as

`(object:fraction% Number Number)`. This is because by adding arbitrary computation at construction-time, there's no longer a close relationship between a call to a constructor and the contents of an object. So in printing values we have a choice to make: either print the constructor call, which doesn't tell us about the contents of the object, *or* print the contents of the object, which doesn't tell us about the call to the constructor. We chose the latter.

Q: Can you call methods on the object being constructed?

A: No. What would they do? Suppose you invoked a method that referred to fields of `this` object — those things just don't exist yet.

Some languages allow this. Java for example, will let you invoke methods from within constructors and should those methods reference fields that are not initialized, bad things happen. (This is just poor language design, and descends from Sir Tony Hoare's "Billion Dollar Mistake": the null reference.)

16.2   Integrity checking

Beyond computing canonical forms, constructors are also useful for checking the integrity of data given to a constructor. For example, suppose we are writing a class to represent dates in terms of their year, month, and day of the month. Now, what if we're given the 67th day of March in the year -17? What should that data represent? Maybe it should be March 40 (because as we heard in class, `(= 40 (- 67 17))`; maybe it should be May 6th, 17 B.C., maybe it should May 6th, 17 years before the UNIX epoch of 1970; maybe it should be March 5, 17 A.D., which we arrive at by mod'ing 67 by the number of days in March and making the year positive; or maybe... this data is just bogus and we should raise an error and refuse to continue computing.

Let's see how we can implement a simple form of *integrity checking* in a constructor. We will implement a class to represent dates and raise an error in case of a situation like the above.

```
;; A Date is (date% Number Number Number).
;; Interp: Year Month Day.
;; Year must be positive.
;; Month must be in [1,12].
;; Day must be in [1,31].
(define-class date%
  (fields year month day))
```

We can still construct meaningless dates, so what we would like to do is check the inputs to a constructor make some sense. This let's us establish the integrity of all `date%` objects — if you have your hands on a `date%` object, you can safely assume it satisfies the specification we've given in the data definition.

The simplest way to satisfy the specification is with this constructor:

`date%`

```
(constructor (y m d)
  (error "I didn't like this date!"))
```

This is known as a "sound" solution in the program verification community. Notice: if you have your hands on a `date%` object, you can safely assume it satisfies the specification we've given in the data definition. Why? Because you *cannot* construct a `date%` object.

We'd like to do better by accepting more legitimate dates. Here is one that accepts all the things deemed acceptable in our specification (this is both "sound" and "complete"):

`date%`

```
(constructor (y m d)
  (cond [(<= y 0) (error "year was negative or zero")]
        [(or (> m 12) (< m 1)) (error "month too big or too small")]
        [(or (> d 31) (< d 1)) (error "day too big or too small")]
        [else (fields y m d)]))
```

Example:

```
> (date% 2011 3 67)
day too big or too small
```

It is still possible to construct meaningless dates, such as February 31, 2011. However, more stringent validation is just some more code away, and since we are more concerned with the *concept* of integrity checking than in a robust date library, we won't go into the details.

Thus we can *establish* invariants with computation, or we can *reject* inputs that don't have the invariant we want to maintain. And we can combine these approaches. (You may want to compute fractions in lowest terms *and* reject `0` as a denominator in `fraction%`, for example.)

16.3   Ordered binary trees

Now we want to look at a slightly larger program and how we use constructors to enforce important invariants. In this section, we want to develop a *representation of sorted lists of numbers*, which is what we did in section 19.1, but this time we're going to represent a sorted list of numbers as an *ordered binary tree*.

An ordered binary tree looks like this:

```
     *
    / \
   *   3
  / \
 1   2
```

Notice that there is data only at the leaves of the tree and that if you traverse the leaves in left-to-right order, you recover the sorted list of numbers. Thus there is an important invariant about this data structure: whenever we have an ordered binary tree node, the left sub-tree is sorted and the right sub-tree is sorted *and* and numbers in the left sub-tree are smaller than or equal to all the numbers in the right sub-tree.

Here is our data and class definition for ordered binary trees:

```
;; A OBT is one of:
;; - (node% OBT OBT)
```

```
;; - (leaf% Number)
(define-class leaf%
  (fields number))
(define-class node%
  (fields left right))
```

Some examples:

```
(leaf% 7)
(node% (leaf% 1) (leaf% 2))
```

Now, is this an example?

```
(node% (leaf% 7) (leaf% 2))
```

This example points out that we are currently missing the specification of our invariant in the data definition:

```
;; A OBT is one of:
;; - (node% OBT OBT)
;; - (leaf% Number)
;; Invariant: numbers are in ascending order from left to right.
```

What happens if we try to construct something that violates our invariant? Nothing – we just construct bogus things. Now how could enforce this ascending order invariant?

Well, let's first think about the `leaf%` case. We are given a number and we need to construct an ordered binary tree, meaning all the numbers in the tree are in ascending order. Since we are constructing a tree with only one number in it, it's trivial to enforce this invariant—it's always true!

Now consider the `node%` case. We are given two ordered binary trees. What does that mean? It means the numbers of each tree are in ascending order. But wait—isn't that the property we are trying to enforce? Yes. Notice that if we *assume* this of the inputs and *guarantee* this of the constructed value, then it *must be true of all OBTs*; i.e. the assumption was valid. If this reasoning seems circular to you, keep in mind this is not unlike "the magic of recursion", which is not magic at all, but seems to be since it lets you assume the very function you are writing works in recursive calls on structurally smaller inputs. If you do the right thing in the base case, and if on that assumption of the recursive call, you can construct the correct result, then that assumption about the recursive call was valid and your program is correct for all inputs.

OK, so the challenge at hand is not in verifying that the input `OBT`s posses the invariant, but in guaranteeing that the result of the constructor possesses it. If we can do that, than we know the given `OBT`s must have the property.

But now this assumption is not sufficient to guarantee that the default constructor works:

`node%`

```
;; OBT OBT -> OBT
(constructor (a b)
  (fields a b))
```

Why? Although we know that the left and right sub-tree are OBTs, we know nothing about the relationship *between* the left and right sub-tree, which was an important part of the invariant. Consider for example, the OBTs:

```
(node% (leaf% 4) (leaf% 5))
(node% (leaf% 2) (leaf% 3))
```

Independently considered, these are definitely OBTs. However, if we construct a node% out of these two trees, we get:

```
(node% (node% (leaf% 4) (leaf% 5))
       (node% (leaf% 2) (leaf% 3)))
```

which is definitely *not* an OBT. (Thus we have broken the stated contract on the constructor.)

We could correctly *compute* an OBT by determining that, in this example, the first given tree needs to be the right sub-tree and the second given tree needs to be the left sub-tree. We can make such a determination based on the maximum and minimum numbers in each of the given trees, and that suggest the following constructor:

node%

```
;; OBT OBT -> OBT
(constructor (a b)
  (cond [(<= (b . max) (a . min))
         (fields b a)]
        [(<= (a . max) (b . min))
         (fields a b)]
        [else
         ...]))
```

The max and min methods are easily dismissed from our wish list:

leaf%

```
(define (min)
  (this . n))
(define (max)
  (this . n))
```

node%

```
(define (min)
  (this . left . min))
(define (max)
  (this . right . max))
```

At this point, our constructor does the right thing when given two `OBT`s that do not overlap, as in the example we considered, but a troubling pair of examples to ponder over is:

```
(node% (leaf% 2) (leaf% 4))
(node% (leaf% 3) (leaf% 5))
```

Again, considered independently, these are definitely `OBT`s, but there's no way to construct an ordered binary tree with one of these as the left and the other as the right; either order you pick will be wrong. This case is the `else` clause of our constructor. What should we do? One solution is just to reject this case and raise and error:

`node%`

```
;; OBT OBT -> OBT
(constructor (a b)
  (cond [(<= (b . max) (a . min))
         (fields b a)]
        [(<= (a . max) (b . min))
         (fields a b)]
        [else
         (error "trees overlap")]))
```

But really this again fails to live up to the stated contract since we should be able to take any two `OBT`s and construct an `OBT` out of them. We know that if the trees overlap, we can't simple make a node with them as sub-trees; we have to do something more sophisticated. Here's an idea: insert all of the elements of one into the other. So long as we make this insertion do the right thing, our constructor will succeed in maintaining the invariant properly.

So if we indulge in some wishful thinking and suppose we have a `insert-tree` in our interface:

```
;; insert-tree : OBT -> OBT
;; Insert all elements of this tree into the given one.
```

then we can write the constructor as follows:

`node%`

```
;; OBT OBT -> OBT
(constructor (a b)
  (cond [(<= (b . max) (a . min))
         (fields b a)]
        [(<= (a . max) (b . min))
         (fields a b)]
        [else
         (local [(define t (a . insert-tree b))]
           (fields (t . left) (t . right)))]))
```

That leaves `insert-tree` to be written. First let's consider the case of inserting a `leaf%` into a tree. If we again rely on some wishful thinking and relegate the work to another method that inserts a number into a list, we can easily write `insert-tree` for the `leaf%` case:

`leaf%`

```
(define (insert-tree other)
  (send other insert (this . number)))
```

In the `node%` case, if we first consider the template (the inventory of what we have available to use), we have:

`node%`

```
(define (insert-tree other)
  (this . left . insert-tree other) ...
  (this . right . insert-tree other) ...)
```

But here we don't really want to insert the left tree into the other and the right into the other. We want to insert the right tree into the other, then insert the left tree into *that one* (other permutations of the order of insertions would work, too). That leads us to:

`node%`

```
(define (insert-tree other)
  (this . left . insert-tree (this . right . insert-tree other)))
```

We have only a single item remaining on our wish list—we need to implement the `insert` method for inserting a single number into a tree.

First let's consider the case of inserting a number into a `leaf%`. If we have a leaf and we insert a number into it, we know we get a node with two leaves. But where should the inserted number go? One solution is to compare the inserted number against the existing number to determine which side the number should go to:

`leaf%`

```
(define (insert m)
  (node% (leaf% (the-real-min n m))
         (leaf% (the-real-max n m))))
```

In the case of inserting a number into a node, we compare the number against the maximum of the left sub-tree to determine if the number should be inserted in the left or right:

`node%`

```
(define (insert n)
  (cond [(> n (this . left . max))
         (node% (this . left)
                (this . right . insert n))]
        [else
         (node% (this . left . insert n)
                (this . right))]))
```

16.4 Exercises

16.4.1 Queues

The University Registrar is instituting a new course registration system, in which each student will wait in a "Virtual Line" until every student ahead of them has registered. A simple way to represent a line (also known as a *queue*) is by using a list. But this representation makes it slow to add somebody to the end of the line (or to take somebody off the front of the line, depending on whether the front of the list represents the front or rear of the line).

In order to provide maximal waiting efficiency, you have been tasked with implementing a representation that uses *two* lists! The key idea of this fancy representation is that one list will represent some portion of the front of the line, while the other will represent the remainder of the line *in reverse order*. So if you're the first element of the first list, you are at the head of the line. On the other hand, if you're the first element of the second list, you are the very last person in line.

Here is the interface for queues:

```
;; A [IQ X] implements:

;; head : -> X
;; Produce the element at the head of this queue.
;; Assume: this queue is not empty.

;; deq : -> [IQ X]     (Short for "dequeue")
;; Produces a new queue like this queue, but without
;; this queue's first element.
;; Assume: this queue is not empty.

;; enq : X -> [IQ X]    (Short for "enqueue")
;; Produce new queue with given element added to the
;; END of this queue.

;; emp? : -> Boolean
;; Is this queue empty?
```

The `head` and `deq` operations require that the queue be non-empty when they are used, but this can be assumed and these operations do not need to check for an empty queue.

Further, the Registrar's office has just learned about *invariants*, and insists on maintaining the following invariant about all of their queues:

> *if the front of the queue is empty, the whole queue must also be empty.*

The Registrar's office has given you three tasks to prepare their Virtual Line for its launch later this semester:

- Design an implementation of the queue data structure to the Registrar's specifications. You must maintain the invariant stated above, and you should take advantage of the invariant when implementing the operations.

- Unfortunately, when testing the queue, the Registrar has discovered that some queues with the same elements in the same order can be represented in multiple ways. Give an example of two different representations of the same queue. Implement a `to-list` operation which produces a list of elements going in order from the front to the rear of the queue. In your tests, you should show how this addresses the problem.

- The Registrar has a problem with careless data entry. Design and implement a constructor for queues which, given two input lists of elements, ensures that the invariant is maintained.

Part V

Mutation

Chapter 17

Ch-Ch-Ch-Ch-Changes

We want to design a class, `counter%`, with the following interface

```
;; m : -> Number
;; Produce the number of times 'm' has been called
```

Now let's try to implement this class.

```
(define-class counter%
  (fields called)
  (define (m)
    hmmm))
```

Unfortunately, it's not immediately clear what to put in the body of `m`. We can understand our task better by writing examples.

```
(check-expect ((counter% 0) . m) 1)
(check-expect ((counter% 4) . m) 5)
```

This suggests the following implementation:

counter%

```
(define (m)
  (add1 (this . called)))
```

Now our all of our tests pass.
However, when we try our a few more examples, we see this:

```
> (define c (counter% 0))

> (send c m)
1
> (send c m)
1
```

Of course, this is the wrong answer. We shouldn't be surprised, since nothing has changed about c—in fact, nothing ever happens to c, and only one counter% instance is produced in this program. In order to give m the ability to remember things, we will need to do something to get a different counter%.

One possibility is to change m to produce both the desired result *and* a new counter.

```
> (define-struct r (n new-counter))

> (define-class counter%
    (fields called)
    (define (m)
      (make-r
       (add1 (send this called))
       (counter% (add1 (send this called))))))

> (define c (counter% 0))

> (send c m)
(make-r 1 (object:counter% 1))
> (define d (r-new-counter (send c m)))

> d
(object:counter% 1)
> (send d m)
(make-r 2 (object:counter% 2))
```

So far, so good—we can get a new counter%, and when we use that new value, we get the right answer. However, we haven't solved the problem yet:

```
> (send c m)
(make-r 1 (object:counter% 1))
```

This is the same answer that we had before, and not the desired one.

In fact, this behavior is the result of one of the important design principles of this class, and of Fundies 1, up until this point. If you call a function or method with the same inputs, you get the same result. Always!

Unfortunately, that make it impossible to implement m, because ms spec violates this assumption—if you call it, it is required to produce a *different* result from the last time it was called.

Previously, we've always been able to rely on this test passing, regardless of what you put in E

```
(check-expect E E)
```

Actually, it turns out that there have been a few exceptions to this rule:

- ```(random 5)```

- User input, such as in `big-bang`

Now, however, we are proposing a much more fundament violation of this principle.

Before we violate the principle, though, let's look at one more possible idea: accumulators.

We could add an acummulator to `m`, which is the previous number of times we've been called. We've used this solution before to create functions and methods that remember previous information. In this case, though, accumulators are a non-solution. If we add an accumulator to `m` to indicate what we're remembering, we get this method:

`counter%`

```
(define (m accum) (add1 accum))
```

But that's a pretty boring method—it's just a wrapper around `add1`. And it's not a solution to our problem: instead of the `counter%` class or the `m` method remembering the number of times we've called `m`, we have to remember it ourselves, and provide it as input every time we call `m`.

To truly solve our problem, and implement `m`, we need new language support. This support is provided in `class/3`.

The `class/3` language provides the new `set-field!` form, which is used like this:

```
(set-field! f new-value)
```

This *changes* the value of the field named `f` in `this` object to whatever `new-value` is.

We can now revise our defintion of `m` to

`counter%`

```
(define (m)
  (begin (set-field! called (add1 (send this called)))
         (add1 (send this called))))
```

Note that `set-field!` *doesn't produce* a new version of the field, instead it *changes* the field named `called` to something new.

We've also introduced one more language feature in `class/3`: `begin`. The `begin` form works by evaluating each expression in turn, throwing away the result of every expression except that last one. Then it does the last part, and produces that result.

Unlike `set-field!`, `begin` doesn't add any new capability to the language. For example, we can simulate `begin` using `local`. For example:

```
(local [(define dummy (set-field! called (add1 (send this called))))]
  (add1 (send this called)))
```

This is very verbose, and requires creating new variables like `dummy` that are never used. Therefore, `begin` is a useful addition to our language, now that we work with expressions like `set-field!` that don't produce any useful results.

Question: How would we do something like this in a purely functional language?
Answer: We would do something similar to the `make-a` approach
Question: Do we have `begin`?
Answer: No presented above. In Haskell, this approach is frequently used.

A brief discussion of `void`
What happens if we return the result of `set-field!`? It produces nothing—DrRacket doesn't print anything at all.
However, there's no way for DrRacket to truly have nothing at all, so it has an internal value called `void`. This value doesn't have any uses, though, and you shouldn't ever see it.

Now Expressions do two things: - produce a result (everything does this) - has some effect (some expressions do this)

Now we write effect statements. Have to write them for every method/function that has an effect.

counter%

```
;; m : -> Number
;; Produce the number of times m has been called
;; Effect : increment the called field
(define (m)
  (begin (set-field! called (add1 (send this called)))
         (add1 (send this called)))))
```

We've lost a lot of reasoning power but gained expressiveness.

What have I really gained, though?

Imagine that you're modeling bank financial systems. You want to deposit money into the account, and then the money should be there afterwards.

```
;; An Account is (account% Number)
(define-class account%
  (fields amt)

  ;; Number -> Account
  (define (deposit n)
    (account% (+ (this . amt) n))))
```

But this doesn't model bank accounts properly.

I deposit, my valentine deposits, I deposit – whoops!

New version:

```
;; An Account is (account% Number)
(define-class account%
  (fields amt)

  ;; Number -> Void
  ;; Effect: increases the field amt by n
  ;; Purpose: add money to this account
  (define (deposit n)
    (set-field! amt (+ (this . amt) n))))
```

Note that we don't need to produce any result at all.

```
;; A Person is (person% String Account Number)
(define-class person%
  (fields name bank paycheck)
  ;; -> Void
  ;; Deposit the appropriate amount
```

```
  ;; Effect: changes the the bank account amt
  (define (pay)
    (this . bank . deposit (this . paycheck))))

> (define dvh-acct (account% 0))

> (define dvh (person% "DVH" dvh-acct 150))

> (define sweetie (person% "Sweetie" dvh-acct 3000))

> (send dvh pay)

> dvh-acct
(object:account% 150)
> (send sweetie pay)

> dvh-acct
(object:account% 3150)
```

Note that we *cannot* replace dvh-acct with (account% 0) – we'd get totally different results.

Now equality is much more subtle – intensional equality vs extensional equality. Same fork example.

What if we do:

```
> (define new-acct dvh-acct)

> (define p (person% "Fred" new-acct 400))

> (send p pay)

; updated
> dvh-acct
(object:account% 3550)
```

What if we create new account% with 0? Then the effects are not shared.
What if we do:

```
> (define x (send dvh-acct amt))

> x
3550
> (send dvh pay)

; still the same
> x
3550
```

What if we do

```
> (define y (send dvh bank))

> y
(object:account% 3700)
> (send dvh pay)

; now different
> y
(object:account% 3850)
```

The differece is that x is the name of a number, and numbers don't change, but y is the name of an account, and accounts change over time.

Objects can change, but other things do not change. Structures and lists can contain objects that change, but the structures and lists themselves do not change, the object they point to are the same objects.

Testing is hard with mutation. Give an example in the notes.

Chapter 18

Circular Data

Books & Authors

Books have: title : String author : Author

Authors have: name : String books : [Listof Book]

As data def:

```
;; A Book is (book% String Author)
;; An Author is (author% String [Listof Book])
```

Can we make an `Author`?

```
(author% "Rose" empty)
(book% "Reign of Roquet" (author% "Rose" empty))
```

But this is wrong: the Rose has written a book, but the author object doesn't know about it.

Do we need books to know the author? Yes.

We've seen this before with graph structure. We represented graphs as association lists, using symbolic names.

```
;; A Book is (book% String Author)
(define-class book%
  (fields title author))

;; An Author is (author% String [Listof Book])
(define-class author%
  (fields name books))

(define rose  (author% "Rose" empty))
(define reign (book% "Reign of Roquet" rose))
```

But:

```
reign
rose
```

Question: Does `reign` contain a copy of `rose`, or are they all the same `rose`?
Answer: always the same, because we use the name `rose`, we didn't construct a new
one.

Let's add a new method for modifying the author after a book is written:

```
(define (add-book b)
  (set-field! books (cons b (this . books))))
```

Now we change our example:

```
(define rose  (author% "Rose" empty))
(define reign (book% "Reign of Roquet" rose))
(rose . add-book reign)
```

How does it print?

```
> rose
#0=(object:author% "Rose" (list (object:book% "Reign of Roquet"
#0#)))
> reign
#0=(object:book% "Reign of Roquet" (object:author% "Rose" '(#0#)))
```

See graph-style printing.

But every times we construct a book with an author, we want to use `add-book`. So,
let's use the constructor.

```
(constructor (t a)
  (fields t a)
  (send a add-book this))
```

In the first expression, we *cannot* use `this`, and we must produce the result using
`fields`. Later, we can use `this`, and we get the desired result.

Chapter 19

Back-channels

Up until this point, computations communicate by consuming arguments and producing values. Mutation enables more channels of communication between computations via *shared* mutable data.

As an example, recall our counter world program from section 9.3:

```
;; A Counter is a (counter-world% Natural)
(define-class counter-world%
  (fields n)
  ...
  ;; on-tick : -> Counter
  (define (on-tick)
    (new counter-world% (add1 (this . n)))))))

(big-bang (new counter-world% 0))
```

To illustrate how mutable data provides alternative channels of communication, let's develop a variant of the program that communicates the state of the world through an implicit stateful back-channel.

```
;; A Counter is a (counter-world% Natural)
(define-class counter-world%
  (fields n)
  ...
  ;; on-tick : -> Counter
  (define (on-tick)
    (begin (set-field! n (add1 (send this n)))
           this)))

(big-bang (new counter-world% 0))
```

Notice how the new `on-tick` method doesn't produce a new `counter-world%` object with an incremented counter; instead it *mutates* its own counter and returns its (mutilated) self.

You'll find that this program appears to behave just like the old version, but backchannels open up new forms of interaction (and interference) that may not be intended. For example, can you predict what thise program will do?

```
(launch-many-worlds (big-bang (new counter-world% 0))
                    (big-bang (new counter-world% 0)))
```

How about this (seemingly equivalent) one?

```
(define w (new counter-world% 0))
(launch-many-worlds (big-bang w)
                    (big-bang w))
```

Chapter 20

Intensional equality

Mutation exposes yet another sense in which two things may be consider *"the same"*: two things are the same if mutating one mutates the other.

How can we tell if two posns are two different names for the same thing, or if they're two different posns with the same contents?

For example:

```
(define p1 (posn% 3 4))
(define p2 (posn% 3 4))
```

  or

```
(define p1 (posn% 3 4))
(define p2 p1)
```

These are very different in the presence of mutation. We have a way of testing this: eq?. Similarly, the equal? function checks structural equality.

But that didn't shed any light on the question. Is there a way we can check this *in* our language?

Answer: yes. Do some operation to one of them that changes the state of the object, and see if it *also* happens to the other one.

Drawback: you can't necessarily undo this operation.

```
(define (really-the-same? p1 p2)
  ....)
```

Now we need some operation to perform on p1.

posn%

```
;; -> (posn% 'black-eye Number)
(define (punch!)
  (begin
    (set-field! x 'black-eye)
    this))
```

```
(define sam (posn% 3 4))
(send sam punch!)
```

"I punched him so hard, I punched him right out of the data defintion."
Now we can define `really-the-same?`.

```
(define (really-the-same? p1 p2)
  (begin
    (send p1 punch!)
    (symbol? (send p2 x))))
```

```
(really-the-same? p1 p2)
p1
```

Now `p1` is permanently broken, and can't be undone. So `really-the-same?` is a
very problematic, and you should use `eq?`, which uses DrRacket's internal knowledge
of where things came from to answer this question without changing the objects.
    Question:

```
(eq? p1 (send p1 punch!))
```

Produces true.

```
(send p2 =? (send p2 punch!))
```

Produces true (or crashes).

```
(send (send p3 punch!) =? p3)
```

Produces true (or crashes).
Question: Does intensional equality imply structural equality? Yes.

Part VI

Java

Chapter 21

Java

21.1 Two Ideas: Java and Types

Types are a *mechanism* for enforcing data definitions and contracts.

Java is a programming language like the one we've seen, with a different syntax and with types.

21.2 Programming in Java

21.2.1 Java Syntax

```
;; A C is (C Number String String)
(define-class C
  (fields x y z)
  ;; Number Number -> C
  (define (m p q)
    ...))
```

```
class C {
    Number x;
    String y;
    String z;
    public C m(Number p, Number q) {
        ...
    }
    public C(Number x, String y, String z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Let's create a simple pair of numbers:

```java
class Pair {
    Number left;
    Number right;

    public Pair(Number left, Number right) {
        this.left = left;
        this.right = right;
    }

    public Pair swap() {
        return new Pair(this.right, this.left);
    }
}
```

But really, this doesn't work, because Java doesn't have the type Number. So we'll choose Integer instead.

```java
class Pair {
    Integer left;
    Integer right;

    public Pair(Integer left, Integer right) {
        this.left = left;
        this.right = right;
    }

    public Pair swap() {
        return new Pair(this.right, this.left);
    }
}
```

To test this, we'll import a testing library:

```java
import tester.Tester;
```

and write some examples:

```java
class Examples {
    public Examples() {}
    public boolean testSwap(Tester t) {
        return t.checkExpect(new Pair(3,4).swap(),
                             new Pair(4,3));
    }
}
```

21.3   Running Java Programs

We don't have a Run button in Java, so we need a different way to run our program. To do this, we first need to install our test library. This requires installing a JAR file from the NU Tester web site.

The we have to compile the program.

- javac

- The classpath and the -cp option

- Now we have class files, which are binary and all mashed up

- To run this, we use the `java` command, which also has a `-cp` option

- If we change things, we have to recompile and then rerun.

21.4   A More Complex Example

What if we want to represent a union?
```
class Square {
    Integer size;
    public Square(Integer size) {
        this.size = size;
    }
}
class Circ {
    Integer radius;
    public Circ(Integer radius) {
        this.radius = radius;
    }
}
```
How do we declare that both of these are `Shapes`?
```
import tester.Tester;

interface IShape {}

class Square implements IShape {
    Integer size;
    public Square(Integer size) {
        this.size = size;
    }
    public IShape nothing(IShape i) {
        return i;
    }
}

class Circ implements IShape {
    Integer radius;
    public Circ(Integer radius) {
        this.radius = radius;
    }
```

```
}

class Examples {
    Examples () {}
    public boolean testNothing(Tester t) {
        Square s = new Square(5); // A local binding
        return t.checkExpect(s.nothing(new Circ(2)),
                             new Circ(2));
    }
}
```

21.5   Recursive Unions

```
import tester.Tester;
class Mt implements IList {
    public Mt() {}
}

class Cons implements IList {
    Integer first;
    IList rest;

    public Cons(Integer first, IList rest) {
        this.first = first;
        this.rest = rest;
    }
}

interface IList {}

class Examples {
    public Examples() {}

    public boolean testList(Tester t) {
        return t.checkExpect(new Mt(), new Mt())
            && t.checkExpect(new Cons(5, new Mt()), new Cons(5, new
Mt()));
    }
}
```

21.6   Enumerations

In Fundies I, we might have written:

```
;; A Title is one of
;; - 'dr
;; - 'mr
;; - 'ms
```

In Java, we write:
```
interface ITitle {}
class Dr implements ITitle {
    Dr() {}
}
class Mr implements ITitle {
    Mr() {}
}
class Ms implements ITitle {
    Ms() {}
}
```
Why write these silly constructors?
```
interface ITitle {}
class Dr implements ITitle {
    Dr() {}
}
class Mr implements ITitle {
    Mr() {}
}
class Ms implements ITitle {
    Integer x;
    Ms() {}

    public Integer m() {
        return x+1;
    }
}

class Main {
    public static void main(String[] args) {
        new Ms().m();
        return;
    }
}
```
Now we get a `NullPointerException`. But what is that?

A discussion of the evils of `null`.

For example, this compiles:

```
interface ITitle {}
class Dr implements ITitle {
    Dr() {}
}
class Mr implements ITitle {
    Mr() {}
}
class Ms implements ITitle {
    Integer x;
    Ms(Integer x) {
        this.x = x;
    }

    public Integer m() {
        return null;
    }
}

class Main {
    public static void main(String[] args) {
        new Ms().m();
        return;
    }
}
```
Never write `null` in your program!
A long sermon on `null`.

21.7 Parameterized Data Definitions

Consider our `Pair` class:
```
class Pair {
    Integer left;
    Integer right;

    public Pair(Integer left, Integer right) {
        this.left = left;
        this.right = right;
    }

    public Pair swap() {
        return new Pair(this.right, this.left);
    }
}
```
Now if we want a `Pair` of `Strings`:
```
class PairString {
```

```
    String left;
    String right;

    public Pair(String left, String right) {
        this.left = left;
        this.right = right;
    }

    public Pair swap() {
        return new Pair(this.right, this.left);
    }
}
```

This is obviously bad—we had to copy and paste. So let's abstract:

```
class Pair<T,V> {
    T left;
    V right;

    public Pair(T left, V right) {
        this.left = left;
        this.right = right;
    }

    public Pair<V,T> swap() {
        return new Pair<V,T>(this.right, this.left);
    }
}

class Examples {
    public Examples() {}
    public boolean testSwap(Tester t) {
        return t.checkExpect(new Pair<Integer,Integer>(3,4).swap(),
                             new Pair<Integer,Integer>(4,3));
    }
}
```

21.8   Abstraction

```
class C {
    Integer x;
    Integer y;
    C(Integer x, Integer y) {
        this.x = x;
        this.y = y;
```

```
    }

    public Integer sq() {
        return this.x * this.x;
    }
}


class D {
    Integer x;
    String z;
    D(Integer x, String z) {
        this.x = x;
        this.z = z;
    }

    public Integer sq() {
        return this.x * this.x;
    }
}
```

Now to abstract:

```
class S {
    Integer x;
    public Integer sq() {
        return this.x * this.x;
    }
    S(Integer x) {
        this.x = x;
    }
}

class C extends S {
    Integer y;
    C(Integer x, Integer y) {
        super(x);
        this.y = y;
    }
}


class D {
    Integer x;
    String z;
    D(Integer x, String z) {
```

```
        this.x = x;
        this.z = z;
    }

    public Integer sq() {
        return this.x * this.x;
    }
}
```

21.9 Types

What sorts of things count as *types* in Java?

- Class names

- Interface names

- Other stuff: `int`, `boolean`, ...

What should be a part of the contract and purpose in Java? Well, we don't need to write down things that are already part of the type. If the contract corresponds to the type, you don't have to repeat it. However, some contracts can't be checked by the type system—you should still write those down.

Chapter 22

Extensional Equality in Java

22.1   Posn

22.2   Equality in Java

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

   public Boolean isEqual(Posn p) {
       return this.x == p.x
           && this.y == p.y;
   }
}

interface LoP {
   Boolean isEmpty();
   Posn getFirst();
   LoP getRest();
}

class MT implements LoP {
   MT() {}

   public Posn getFirst() {
       return null;
```

```java
    }

    public LoP getRest() {
        return ????;
    }

    public Boolean isEmpty() {
        return true;
    }

    public Boolean isEqual(LoP lop) {
        return lop.isEmpty();
    }
}

class Cons implements LoP {
    Posn first;
    LoP rest;

    Cons(Posn first, LoP rest) {
        this.first = first;
        this.rest = rest;
    }

    public Boolean isEmpty() {
        return false;
    }

    public Boolean isEqual(LoP lop) {
        return (!lop.isEmpty())
            && this.first.isEqual(lop.getFirst())
            && this.rest.isEqual(lop.getRest());
    }

    public Posn getFirst() {
        return this.first;
    }

    public LoP getRest() {
        return this.rest;
    }


}
```

22.3   List of Posn

```
interface LoP {
   Boolean isEmpty();
   Posn getFirst();
   LoP getRest();
   Boolean isEqual();
}

class MT implements LoP {
   MT() {}

   public Posn getFirst() {
       return ????;
   }

   public LoP getRest() {
       return ????;
   }

   public Boolean isEmpty() {
       return true;
   }

   public Boolean isEqual(LoP lop) {
       return lop.isEmpty();
   }
}

class Cons implements LoP {
   Posn first;
   LoP rest;

   Cons(Posn first, LoP rest) {
       this.first = first;
       this.rest = rest;
   }

   public Boolean isEmpty() {
       return false;
   }

   public Boolean isEqual(LoP lop) {
       return (!lop.isEmpty())
           && this.first.isEqual(lop.getFirst())
           && this.rest.isEqual(lop.getRest());
```

```
    }

    public Posn getFirst() {
        return this.first;
    }

    public LoP getRest() {
        return this.rest;
    }


}
```

How did we get into this situation? Let's look at `isEqual` again:

```
public Boolean isEqual(LoP lop) {
    return (!lop.isEmpty())
        && this.first.isEqual(lop.getFirst())
        && this.rest.isEqual(lop.getRest());
}
```

After the first conjunct, we know that `lop` is a `Cons`, and so we want to get the first and rest. But *Java* doesn't know that, and so we created the `getFirst` and `getRest` methods.

If we return `null`, then we create problems for all possible clients of the `getFirst` method. So instead we raise an error:

```
public Posn getFirst() {
    throw new RuntimeException("Can't take the first of an empty list.")
}
```

```
public Posn getRest() {
    throw new RuntimeException("Can't take the rest of an empty list.")
}
```

But if we think about `isEqual` again, can we help Java do something smarter? Let's try taking `getFirst` and `getRest` out of the interface. If we do that, we get an error message about not finding those methods in the interface. But can we persuade Java that `lop` is really a `Cons`? Yes:

```
public Boolean isEqual(LoP lop) {
    return (!lop.isEmpty())
        && this.first.isEqual(((Cons)lop).getFirst())
        && this.rest.isEqual(((Cons)lop).getRest());
}
```

This is called *casting*, and it's crucial for (some kinds of) programming using the Java type system. However, it's important to remember that we're cheating the type system here. The cast is turned into a runtime check, and we could write anything down that we want. It's only when running the program that Java can check whether `lop` is really a `Cons`.

22.4 Equality and Parameterized Types

We try writing a parameterized version of Listof<X> with equality.

It doesn't work, because X doesn't have an `isEqual` method.

Fix: restrict X to implement the IEqual interface.

Question about structural vs nominal typing.

Discussion about polymorphism/bounds/extends/etc.

The class language version:

```
;; An [IEqual X] implements
;; =? : X -> Boolean

;; A [Listof X] implements
;; empty?: -> Boolean
;; and implements [IEqual [Listof X]]
;; where X implements [IEqual X]
```

If we unroll our definitions a little, we get:

```
;; An [IEqual X] implements
;; =? : X -> Boolean

;; A [Listof X] implements
;; empty? : -> Boolean
;; =? : [Listof X] -> Boolean
;; where X implements
;; =? : X -> Boolean
```

22.5 Comparing different kinds of things

This works:

```
(equal? 5 "fred")
```

This doesn't work:

```
new Posn(3,4).isEqual("fred")
```

The reason this doesn't work is that we're violating the contract on `isEqual`, which is enforced by the type system.

So let's create a different isEqual method:

```
public Boolean isEqualPosn(Posn p) {
    return this.x == p.x
        && this.y == p.y;
}

public Boolean isEqual(Object o) {
    return (o instanceof Posn)
```

```
        && this.x == p.x
        && this.y == p.y;
}
```
This doesn't typecheck, so we have to add casts again.
```
public Boolean isEqual(Object o) {
    return (o instanceof Posn)
        && this.x == ((Posn)p).x
        && this.y == ((Posn)p).y;
}
```
Or, we can use our way of comparing Posns.
```
public Boolean isEqual(Object o) {
    return (o instanceof Posn)
        && this.isEqualPosn((Posn)o);
}
```

But why would we want to compare things of different types? It turns out that Java has a built-in notion of equality, called the `equals` method. And this method expects its input to be an `Object`.

We can reuse this implementation:
```
public Boolean equals(Object o) {
    return (o instanceof Posn)
        && this.isEqualPosn((Posn)o);
}
```

There are two big problems with this:

- If we extend Posn, then equals will behave differently in different directions.

- We forgot to override `hashCode`.

22.5.1   equals and hashCode

The fundamental rules for overriding `equal`:

- If you override equals, you **must** override `hashCode`.

- `o.equals(p)` implies `o.hashCode() == p.hashCode()`

The rule of `hashCode`: if two objects have differnet hash codes, they are different. `hashCode` is a fast way to check if two objects are different.

Here's a quick and correct implementation of `hashCode`:
```
public int hashCode() {
    return 0;
}
```

Part VII

A Class of Your Own

Chapter 23

Under the Hood: Implementing OO

23.1 Pulling back the veil from object-oriented programming

We will implement OO in ISL+lambda.

In Fundies 1 we saw a way to implement the language we were writing programs in. Today we're going to do something similar. How can we do this?

We could write a Java compiler, which would help us program Java, but the JLS is like 900 pages long.

Structs with functions in them. That would be similar to having objects because we'd have data with functionality bundled together.

We could also implement dictionary and use those to associate values and fields and values and methods.

We could use built-in Racket objects. That would be similar to what we've done, and would be similar to structure and functions.

We could use functions to represents objects.

We could write an interpreter for `class/1` in ISL+.

We are going to start out by using functions to represent objects. Ask yourself: WHAT IS AN OBJECT?

```
- Data + functions - this is the "how they are made" or "what they
are
  contructed out of".
```

23.2 Objects as Functions

Another view is: what do objects do?

```
- Objects respond to messages.
```

What are messages? Messages are names.

So let's write something that responds to things that are names. Let's make an example of something that responds to messages. As an example, we'll make a square object that responds to the messages:

- `side`
- `area`

If we make a square-10 object, what's its contract:

```
;; square-10 : Message -> Number
```

How should we represent messages? Symbols.

```
;; A Message is a Symbol
```

```
;; square-10 : Message -> Number
(define (square-10 msg)
  (cond [(symbol=? msg 'side) 10]
        [(symbol=? msg 'area) 100]
        [else "message not understood"]))
```

```
(check-expect (square-10 'side) 10)
(check-expect (square-10 'area) 100)
(check-error  (square-10 'bad))
```

How would we write a square-5 object?

```
;; square-5 : Message -> Number
(define (square-5 msg)
  (cond [(symbol=? msg 'side) 5]
        [(symbol=? msg 'area) 25]
        [else "message not understood"]))
```

```
(check-expect (square-5 'side) 5)
(check-expect (square-5 'area) 25)
(check-error  (square-5 'bad))
```

Now we have two simple objects that look very similar. Let's abstract.

```
;; A Square is a Message -> Number
;; square% : Number -> Square
(define (square% side)
  (local [(define (the-square msg)
            (cond [(symbol=? msg 'side) side]
                  [(symbol=? msg 'area) (sqr side)]
                  [else (error "message not understood")]))]
    the-square))
```

```
(define square-10 (square% 10))
(define square-5  (square% 5))
```

Why is what've done a little weird considering we said that an object is data plus functions.

Depending on your perspective:

- `We only have data.`

```
- We only have functions.
```
Our messages are always just a symbol. But what about arguments?

But there's something else. Where did the data go?

The trick is that when we produce the square function, it remembers the values it can see, like side, when it was created. So the data is remember in the function.

Functions are really code plus data.

Basically a list of fields that map to values, plus code.

That's why we are able to implement objects so easily.

But how do we do inheritance?

Let's create another kind of shape – let's create circles.

```
;; A Circle is a Message -> Number
;; Number -> Circle
(define (circle% radius)
  (local [(define (the-circle msg)
            (cond [(symbol=? msg 'radius) radius]
                  [(symbol=? msg 'area) (* radius radius pi)]
                  [else (error "message not understood")]))]
    the-circle))

(define circle-2 (circle% 2))
(check-expect (circle-2 'radius) 2)
(check-within (circle-2 'area) (* 2 2 pi) 0.0001)
```

There's a lot of repeated code here. How can we abstract something common to both of these definitions.

We have different code for handling message, but all of the objects have the same code for the message that is not understood.

```
(define (dumb-object msg)
  (error "message not understood"))
```

This is an object – not a constructor for an object.

Let's write a constructor for this:

```
(define (dumb%)
  (local [(define (the-dumb-object msg)
            (error "message not understood"))]
    the-dumb-object))
```

Now how could we use this?

```
(define (square% side super)
  (local [(define (the-square msg)
            (cond [(symbol=? msg 'side) side]
                  [(symbol=? msg 'area) (sqr side)]
                  [else ((super) msg)]))]
    the-square))
```

This is odd – we can make squares with different super classes. Let's fix that.

```
(define (square% side)
  (local [(define (the-square msg)
            (cond [(symbol=? msg 'side) side]
                  [(symbol=? msg 'area) (sqr side)]
```

```
                              [else ((dumb%) msg)])))]
       the-square))
```
dumb% is actually object%.
```
(define (object%)
  (local [(define (the-dumb-object msg)
             (error "message not understood"))]
    the-dumb-object))
```
How many times are we going to create an object% object? How do we have it happen only once?
```
(define (square% side)
  (local [(define super (object%))
          (define (the-square msg)
             (cond [(symbol=? msg 'side) side]
                   [(symbol=? msg 'area) (sqr side)]
                   [else (super msg)]))]
    the-square))
```
And likewise for circle:
```
(define (circle% radius)
  (local [(define super (object%))
          (define (the-circle msg)
             (cond [(symbol=? msg 'radius) radius]
                   [(symbol=? msg 'area) (* radius radius pi)]
                   [else (super msg)]))]
    the-circle))
```
We've now abstracted out the behavior of the error message.

But, let's come back to the observation that we can add methods to object% that every object will now understand.

For example, we could add =?, but that sounds hard. Let's do something really simple:
```
(define (object%)
  (local [(define (the-dumb-object msg)
             (cond [(symbol=? msg 'hi) "Howdy"]
                   [else (error "message not understood")]))]
    the-dumb-object))
```
But we've broken the contract. So we'll say instead that an object can respond to any message and produce anything.

OK, let's see how it works:
```
(check-expect ((square% 10) 'hi) "Howdy")
```
Look! We have inheritance! All istances of subclasses of object% understand the hi message!

This is almost everything that is going on under the hood in classN, Java, Ruby, etc.

But does this do overriding?
```
(define (square% side)
  (local [(define super (object%))
          (define (the-square msg)
             (cond [(symbol=? msg 'side) side]
```

```
                    [(symbol=? msg 'area) (sqr side)]
                    ;; Overriding the hi method.
                    [(symbol=? msg 'hi) "Good day, sir."]
                    [else (super msg)]))]
    the-square))
```

Great, but none of our messages take arguments. How can we do that? We could change our representation of a messages to include arguments.

```
;; A Message is a (make-msg Symbol [Listof Anything]).
(define-struct msg (name args))

;; Symbol Message -> Boolean
(define (msg-is? sym msg)
  (symbol=? sym (msg-name msg)))

(define (object%)
  (local [(define (the-dumb-object msg)
            (cond [(msg-is? 'hi msg) "Howdy"]
                  [else (error "message not understood")]))]
    the-dumb-object))

(check-expect ((object%) (make-msg 'hi empty)) "Howdy")
```

Another approach, revert back to Message = Symbol.

```
;; An Object is a Message [Listof Symbol] -> Anything

(define (object%)
  (local [(define (the-dumb-object msg args)
            (cond [(symbol=? 'hi msg) "Howdy"]
                  [else (error "message not understood")]))]
    the-dumb-object))
```

Another approach, return a function that takes the arguments.

Suppose we want to add a multiply method to circles.

```
(define (circle% radius)
  (local [(define super (object%))
          (define (the-circle msg)
            (cond [(symbol=? msg 'radius) radius]
                  [(symbol=? msg 'area) (* radius radius pi)]
                  [(symbol=? msg 'multiply)
                   (lambda (factor)
                     (circle% (* factor radius)))]
                  [else (super msg)]))]
    the-circle))
```

These contracts suck. We really want to talk about the contract of each method that is supported by an object.

```
;; A Circle is a Object that implements
;; 'radius -> Number
;; 'area -> Number
```

```
;; 'multiply -> (Number -> Circle)

(((circle% 10) 'multiply) 4) => circle with radius 40

(check-expect (((((circle% 10) 'multiply) 4) 'radius) 40)
```

So we've got classes, objects, inheritance, overriding, and basically everything you'd want in a class system.

We might like to have a nice notation to make it more convenient to write programs in this style, but this is really all that is going on.

Suppose we add a field to square%, called area, which is computed at construction time and stored away in the field.

```
(define (square% side)
  (local [(define super (object%))
          (define area (sqr side))
          (define (the-square msg)
            (cond [(symbol=? msg 'side) side]
                  [(symbol=? msg 'area) area]
                  ;; Overriding the hi method.
                  [(symbol=? msg 'hi) "Good day, sir."]
                  [else (super msg)]))]
    the-square))
```

So we can write constructors that do computation.

What about this?

```
(define (square% side)
  (local [(define super (object%))
          (define area (sqr side))
          (define (this msg)
            (cond [(symbol=? msg 'side) side]
                  [(symbol=? msg 'area) area]
                  ;; Overriding the hi method.
                  [(symbol=? msg 'hi) "Good day, sir."]
                  [else (super msg)]))]
    this))
```

Let's redefine area to use. So in class/1 we might write (sqr (this . side)). We'll if we pick our names better, it should become obvious.

```
(define (square% side)
  (local [(define super (object%))
          (define (this msg)
            (cond [(symbol=? msg 'side) side]
                  [(symbol=? msg 'area) (sqr (this 'side))]
                  ;; Overriding the hi method.
                  [(symbol=? msg 'hi) "Good day, sir."]
                  [else (super msg)]))]
    this))
```

23.3   Objects as Structures

Now I want to step back and look at a different approach for doing this.

Guiding principle: Data + Functions

```
;; A Method is a Function.
;; An Object is a (make-obj [Listof Any] [Listof Method]).

(define-struct obj (fields methods))
```

But the "is a Function" contract is not very useful, but really we don't know what the contract on a method is until later.

Let's create our simple square-10 object:

```
(define square-10 (make-obj (list ...) (list ...)))
```

What should go in these lists?

```
(define square-10
  (make-obj (list 10)
            (list
              ;; side : -> Number
              (lambda () 10)
              ;; area : -> Number
              (lambda () 100))))
```

```
(check-expect ((first (obj-methods square-10))) 10)
(check-expect ((second (obj-methods square-100))) 100)
```

What's wrong with this? Nothing is called by name.

Methods can't access the fields! What the hell is going on here?

Here's an idea: pass the object itself to the methods.

```
(define square-10
  (make-obj (list 10)
            (list
              ;; side : -> Number
              (lambda (itself)
                (first (object-fields itself)))
              ;; area : -> Number
              (lambda (itself)
                (* (first (obj-fields itself))
                   (first (obj-fields itself)))))))
```

```
(check-expect ((first (obj-methods square-10)) square-10) 10)
(check-expect ((second (obj-methods square-100)) square-10) 100)
```

```
(define square-10
  (make-obj (list 10)
            (list
              ;; side : -> Number
              (lambda (itself)
                (first (object-fields itself)))
```

```
                    ;; area : -> Number
                    (lambda (itself)
                      (* ((first (obj-methods itself)) itself)
                         ((first (obj-methods itself)) itself))))))
```

A better name for itself: this!

It's annoying to program like this, but we can abstract this

```
;; Object Name -> Anything
(define (send obj meth)
  ...)
```

Why would you ever do this? Every single object-oriented language you've programmed in works like this: it has a table of data and functions and those functions take as its first object the object itself.

Python makes you write self as the first argument, which is just exposing this implementation detail.

Why would you do one or the other?

The functional style is slow, but easy.

The structural style is fast, but hard.

Here's a question: where in the methods do we need to refer to square-10? Nowhere. Thus we can easily lift the methods out of the definition for square-10.

```
(define square-methods
  (list
    ;; side : -> Number
    (lambda (itself)
      (first (object-fields itself)))
    ;; area : -> Number
    (lambda (itself)
      (* ((first (obj-methods itself)) itself)
         ((first (obj-methods itself)) itself)))))

(define square-10
  (make-obj (list 10) square-methods))

(define square-5
  (make-obj (list 5) square-methods))

(define (square% side)
  (make-obj (list side) square-methods))
```

Part VIII

Solutions

Chapter 24

Solutions

This appendix contains solutions to selected exercises.

24.1   Solution: section 2.4.1

This is a solution for the section 2.4.1 exercise.

```
#lang class/0
; ============================================================
; Complex Structure

; A Complex is a (make-complex Real Real).
; Interp: real and imaginary parts.
(define-struct complex (real imag))

; =? : Complex Complex -> Boolean
; Are the complexes equal?
(check-expect (=? (make-complex 0 0) (make-complex 0 0)) true)
(check-expect (=? (make-complex 0 0) (make-complex 2 3)) false)
(define (=? n m)
  (and (= (complex-real n)
          (complex-real m))
       (= (complex-imag n)
          (complex-imag m))))

; plus : Complex Complex -> Complex
; Add the complexes.
(check-expect (plus (make-complex 2 3) (make-complex 4 5))
              (make-complex 6 8))
(define (plus n m)
  (make-complex (+ (complex-real n) (complex-real m))
                (+ (complex-imag n) (complex-imag m))))
```

```
; minus : Complex Complex -> Complex
; Subtract the complexes.
(check-expect (minus (make-complex 2 3) (make-complex 4 5))
              (make-complex -2 -2))
(define (minus n m)
  (make-complex (- (complex-real n) (complex-real m))
                (- (complex-imag n) (complex-imag m))))

; times : Complex Complex -> Complex
; Multiply the complexes.
(check-expect (times (make-complex 2 3) (make-complex 4 5))
              (make-complex -7 22))
(define (times n m)
  (make-complex (- (* (complex-real n) (complex-real m))
                   (* (complex-imag n) (complex-imag m)))
                (+ (* (complex-imag n) (complex-real m))
                   (* (complex-real n) (complex-imag m)))))

; div : Complex Complex -> Complex
; Divide the complexes.
(check-expect (div (make-complex 2 3) (make-complex 4 5))
              (make-complex 23/41 2/41))
(define (div n m)
  (make-complex (/ (+ (* (complex-real n) (complex-real m))
                      (* (complex-imag n) (complex-imag m)))
                   (+ (sqr (complex-real m))
                      (sqr (complex-imag m))))
                (/ (- (* (complex-imag n) (complex-real m))
                      (* (complex-real n) (complex-imag m)))
                   (+ (sqr (complex-real m))
                      (sqr (complex-imag m))))))

; sq : Complex -> Complex
; Multiply the complex by itself.
(check-expect (sq (make-complex 0 1))
              (make-complex -1 0))
(define (sq n)
  (times n n))

; mag : Complex -> Number
; Compute the magnitude of the complex.
(check-expect (mag (make-complex -1 0)) 1)
(check-expect (mag (make-complex 3 4)) 5)
(define (mag n)
  (sqrt (+ (sqr (complex-real n))
```

```
            (sqr (complex-imag n)))))

; sqroot : Complex -> Complex
; Compute the square root of the complex.
(check-expect (sqroot (make-complex -1 0))
              (make-complex 0 1))
(define (sqroot n)
  (make-complex (sqrt (/ (+ (mag n) (complex-real n)) 2))
                (* (sqrt (/ (- (mag n) (complex-real n)) 2))
                   (if (negative? (complex-imag n))
                       -1
                       1)))))

; to-number : Complex -> Number
; Convert the complex to a Racket complex number.
(check-expect (to-number (make-complex 2 3)) 2+3i)
(define (to-number n)
  (+ (complex-real n)
     (* 0+1i (complex-imag n))))

; Alternative:
; OK, this relies on knowing about 'make-rectangular'.
; (define (to-number n)
;   (make-rectangular (complex-real n)
;                     (complex-imag n)))


; ========================================================
; Complex Class

; A Complex is a (new complex% Real Real).
; Interp: real and imaginary parts.

(define-class complex%
  (fields real imag)

  ; =? : Complex -> Boolean
  ; Is the given complex equal to this one?
  (check-expect (send (new complex% 0 0) =? (new complex% 0 0)) true)
  (check-expect (send (new complex% 0 0) =? (new complex% 2 3)) false)
  (define (=? n)
    (and (= (send this real)
            (send n real))
         (= (send this imag)
            (send n imag))))
```

```
; plus : Complex -> Complex
; Add the given complex to this one.
(check-expect (send (new complex% 2 3) plus (new complex% 4 5))
              (new complex% 6 8))
(define (plus n)
  (new complex%
       (+ (send this real) (send n real))
       (+ (send this imag) (send n imag))))

; minus : Complex -> Complex
; Subtract the given complex from this one.
(check-expect (send (new complex% 2 3) minus (new complex% 4 5))
              (new complex% -2 -2))
(define (minus n)
  (new complex%
       (- (send this real) (send n real))
       (- (send this imag) (send n imag))))

; times : Complex -> Complex
; Multiply the given complex by this one.
(check-expect (send (new complex% 2 3) times (new complex% 4 5))
              (new complex% -7 22))
(define (times n)
  (new complex%
       (- (* (send this real) (send n real))
          (* (send this imag) (send n imag)))
       (+ (* (send this imag) (send n real))
          (* (send this real) (send n imag)))))

; div : Complex -> Complex
; Divide this complex by the given one.
(check-expect (send (new complex% 2 3) div (new complex% 4 5))
              (new complex% 23/41 2/41))
(define (div n)
  (new complex%
       (/ (+ (* (send this real) (send n real))
             (* (send this imag) (send n imag)))
          (+ (sqr (send n real))
             (sqr (send n imag))))
       (/ (- (* (send this imag) (send n real))
             (* (send this real) (send n imag)))
          (+ (sqr (send n real))
             (sqr (send n imag))))))

; sq : -> Complex
; Multiply this complex by itself.
```

```
(check-expect (send (new complex% 0 1) sq)
              (new complex% -1 0))
(define (sq)
  (send this times this))

; Alternative:
; OK, but 'this' solution is preferred.
; (define (sq)
;   (times (new complex%
;               (send this real)
;               (send this imag))))

; Alternative:
; Not OK: no code re-use.
; (define (times n)
;   (new complex%
;        (- (* (send this real) (send this real))
;           (* (send this imag) (send this imag)))
;        (+ (* (send this imag) (send this real))
;           (* (send this real) (send this imag)))))

; mag : -> Number
; Compute the magnitude of this complex.
(check-expect (send (new complex% -1 0) mag) 1)
(check-expect (send (new complex% 3 4) mag) 5)
(define (mag)
  (sqrt (+ (sqr (send this real))
           (sqr (send this imag)))))

; sqroot : -> Complex
; Compute the square root of this complex.
(check-expect (send (new complex% -1 0) sqroot)
              (new complex% 0 1))
(define (sqroot)
  (new complex%
       (sqrt (/ (+ (send this mag) (send this real)) 2))
       (* (sqrt (/ (- (send this mag) (send this real)) 2))
          (if (negative? (send this imag))
              -1
              1))))

; to-number : -> Number
; Convert this complex to a Racket complex number.
(check-expect (send (new complex% 2 3) to-number) 2+3i)
(define (to-number)
  (+ (send this real)
```

```
      (* 0+1i (send this imag)))))
```

24.2   Solution: section 3.5.1

This is a solution for the section 3.5.1 exercise.

```
#lang class/0
;; =========================================================
;; Lists of Numbers

;; INTERFACE
;; ---------
;; A ListofNumber implements:
;;
;; - length : -> Natural
;;   Count the number of elements in this list.
;;
;; - append : ListofNumber -> ListofNumber
;;   Append the elements of this list with the given list.
;;
;; - sum : -> Number
;;   Add all the elements of this list.
;;
;; - prod : -> Number
;;   Multiply all the elements of this list.
;;
;; - contains? : Number -> Boolean
;;   Is the given number an element in this list?
;;
;; - reverse : -> ListofNumber
;;   Reverse the elements of this list.
;;
;; - map : [Number -> Number] -> ListofNumber
;;   Apply given function to each element of this list and
;;   construct a list of the results
;;
;; - max-acc : Number -> Number
;;   Find largest number between given and this list of numbers.
;;
;; A NeListofNumber implements ListofNumber and:
;;
;; - max : -> Number
;;   Find largest number in this non-empty list.


;; IMPLEMENTATION
```

```
;; --------------
;; A (new empty%) implements ListofNumber
;; A (new cons% Number ListofNumber) implements NeListofNumber

(define-class empty%
  (check-expect (send mt length) 0)
  (define (length) 0)

  (check-expect (send mt append ls) ls)
  (define (append ls) ls)

  (check-expect (send mt sum) 0)
  (define (sum) 0)

  (check-expect (send mt prod) 1)
  (define (prod) 1)

  (check-expect (send mt contains? 5) false)
  (define (contains? n) false)

  (check-expect (send mt reverse) mt)
  (define (reverse) this)

  (check-expect (send mt reverse-acc ls) ls)
  (define (reverse-acc ls) ls)

  (check-expect (send mt map add1) mt)
  (define (map f) this)

  (check-expect (send mt max-acc 5) 5)
  (define (max-acc a) a))

(define-class cons%
  (fields first rest)

  (check-expect (send ls length) 3)
  (define (length)
    (add1 (send (send this rest) length)))

  (check-expect (send ls append mt) ls)
  (define (append ls)
    (new cons%
         (send this first)
         (send (send this rest) append ls)))

  (check-expect (send ls sum) 9)
```

```
(define (sum)
  (+ (send this first)
     (send (send this rest) sum)))

(check-expect (send ls prod) 24)
(define (prod)
  (* (send this first)
     (send (send this rest) prod)))

(check-expect (send ls contains? 5) false)
(check-expect (send ls contains? 2) true)
(define (contains? n)
  (or (= n (send this first))
      (send (send this rest) contains? n)))

(check-expect (send ls reverse)
              (new cons% 4 (new cons% 2 (new cons% 3 mt))))
(define (reverse)
  (reverse-acc (new empty%)))

(check-expect (send ls reverse-acc ls)
              (new cons% 4 (new cons% 2 (new cons% 3 ls))))
;; ACCUM: elements seen in reverse order.
(define (reverse-acc ls)
  (send (send this rest) reverse-acc
        (new cons% (send this first) ls)))

(check-expect (send ls map add1)
              (new cons% 4 (new cons% 3 (new cons% 5 mt))))
(define (map f)
  (new cons% (f (send this first))
       (send (send this rest) map f)))


(check-expect (send ls max-acc 8) 8)
(check-expect (send ls max-acc 0) 4)
;; ACCUM: largest number seen so far.
(define (max-acc a)
  (send (send this rest) max-acc
        (cond [(> (send this first) a)
               (send this first)]
              [else a])))

(check-expect (send ls max) 4)
(define (max)
  (send (send this rest) max-acc (send this first))))
```

```
(define mt (new empty%))
(define ls (new cons% 3 (new cons% 2 (new cons% 4 mt))))
```

24.3   Solution: section 3.5.2

This is a solution for the section 3.5.2 exercise.

```
#lang class/0
; A Range is one of
; - (new range% Number Number)
; Interp: represents the range between 'lo' and 'hi'
;          including 'lo', but *not* including 'hi'
; and implements IRange.

; The IRange interface includes:
; - in-range? : Number -> Boolean
;   determine if the given number is in this range
; - union : Range -> Range
;   produce the range containing this and the given range

(define-class range%
(fields lo hi)
(check-expect ((range% 0 1) . in-range? 0) true)
(check-expect ((range% 0 1) . in-range? 1) false)
(define (in-range? n)
  (and (>= n (send this lo)) (< n (send this hi))))

(define (union r)
  (union-range% this r)))

; Part 2:

; A Range is one of
; - (new range% Number Number)
;   Interp: represents the range between 'lo' and 'hi'
;            including 'lo', but *not* including 'hi'
; - (new hi-range% Number Number)
;   Interp: represents the range between 'lo' and 'hi'
;            including 'hi', but *not* including 'lo'
; and implements IRange.
```

```
(define-class hi-range%
(fields lo hi)
(check-expect ((hi-range% 0 1) . in-range? 0) false)
(check-expect ((hi-range% 0 1) . in-range? 0.5) true)
(check-expect ((hi-range% 0 1) . in-range? 1) true)
(define (in-range? n)
  (and (> n (send this lo)) (<= n (send this hi))))

(define (union r)
  (union-range% this r)))

; Part 3:

; A Range is one of
; - (new range% Number Number)
;   Interp: represents the range between 'lo' and 'hi'
;           including 'lo', but *not* including 'hi'
; - (new hi-range% Number Number)
;   Interp: represents the range between 'lo' and 'hi'
;           including 'hi', but *not* including 'lo'
; - (new union-range% Range Range)
;   Interp: including all the numbers in both ranges
; and implements IRange.

(define-class union-range%
(fields left right)
(define (in-range? n)
  (or ((send this left) . in-range? n)
      ((send this right) . in-range? n)))

(define (union r)
  (union-range% this r)))

(define r1 (range% 0 1))
(define r2 (hi-range% 0 1))
(define r3 (range% 2 4))

; union + in-range? test
(check-expect (r1 . union r2 . union r3 . in-range? 3)
              true)

; testing all the union methods
(check-expect (r1 . union r2 . in-range? 1) true)
(check-expect (r2 . union r1 . in-range? 0) true)
```

24.4 Solution: section 8.9.2

This is a solution for the section 8.9.2 exercise.

```
#lang class/0
(require 2htdp/image)
(require class/universe)

; A Zombie is one of:
; - Undead
; - Dead
; implements Point and:
; - move-toward : Player -> Zombie
; - eat-zombie-brains? : Zombie -> Boolean
; - eat-player-brains? : Player -> Boolean
; - draw-on : Scene -> Scene

; A Undead is a (new undead% ...)
; A Dead is a (new dead% ...)

; A Zombies is one of:
; - (new emptyz%)
; - (new conz% Zombie Zombies)
; implements
; - draw-on : Scene -> Scene

; A Player is (new player% Posn)
; implements Point and
; - move-toward : Mouse -> Player
; - draw-on : Scene -> Scene

; A Point implements
; - x : -> Number
; - y : -> Number

; A Posn is a (new posn% Number Number)
; implements Point and
; - move-toward : Point -> Posn
; - draw-on/image : Image Scene -> Scene

; A Mouse is a (new mouse% Number Number)
; implements
; draw-on : Scene -> Scene
; Draws this mouse on the scene

(define PLAYER-VELOCITY 3)
```

```
(define ZOMBIE-VELOCITY 1)
(define PLAYER-RADIUS 10)
(define ZOMBIE-RADIUS 10)

(define-class emptyz%
  (define (draw-on scn) scn)
  (define (move-toward player) this)
  (define (any-eating-player-brains? player) false)
  (define (any-eating-zombie-brains? zombie) false)
  (define (kill-any-touching) this)
  (define (kill-any-touching/a zs) zs))

(define-class conz%
  (fields first rest)
  (define (draw-on scn)
    (send (send this first) draw-on
          (send (send this rest) draw-on scn)))

  (define (move-toward player)
    (new conz%
         (send (send this first) move-toward player)
         (send (send this rest) move-toward player)))

  (define (any-eating-player-brains? player)
    (or (send (send this first) eat-player-brains? player)
        (send (send this rest) any-eating-player-brains? player)))

  (define (any-eating-zombie-brains? zombie)
    (or (send (send this first) eat-zombie-brains? zombie)
        (send (send this rest) any-eating-zombie-brains? zombie)))

  (define (kill-any-touching)
    (send this kill-any-touching/a (new emptyz%)))

  (define (kill-any-touching/a zs)
    (local [(define first-z (send this first))
            (define rest-z (send this rest))]
      (cond [(or (send zs any-eating-zombie-brains? first-z)
                 (send rest-z any-eating-zombie-brains? first-z))
             (send rest-z kill-any-touching/a (new conz% (send first-
z kill) zs))]
            [else
             (send rest-z kill-any-touching/a (new conz% first-
z zs))]))))
```

```
; A Undead is a (new undead% Posn) implements Zombie
(define-class undead%
  (fields p)
  ; - move-toward : Player -> Zombie
  ; - eat-zombie-brains? : Zombie -> Boolean
  ; - eat-player-brains? : Player -> Boolean

  (define (x) (send (send this p) x))
  (define (y) (send (send this p) y))

  (check-expect (send (new undead% (new posn% 10 20)) draw-on (empty-
scene 100 100))
                (place-image (circle ZOMBIE-RADIUS "solid" "red")
                             10 20
                             (empty-scene 100 100)))
  (define (draw-on scn)
    (send (send this p) draw-on/image (circle ZOMBIE-RADIUS "solid" "red") scn))

  (check-expect (send (new undead% (new posn% 0 0)) move-toward
                      (new player% (new posn% 0 10)))
                (new undead% (new posn% 0 ZOMBIE-VELOCITY)))
  (check-expect (send (new undead% (new posn% 0 0)) move-toward
                      (new player% (new posn% 10 0)))
                (new undead% (new posn% ZOMBIE-VELOCITY 0)))

  ; move-toward : Player -> Zombie
  ; Move this player ZOMBIE-VELOCITY units toward the given player.
  (define (move-toward player)
    (new undead% (send (send this p) move-toward player ZOMBIE-
VELOCITY)))

  ; eat-player-brains? : Player -> Boolean
  (define (eat-player-brains? player)
    (< (send (send this p) dist player)
       (+ ZOMBIE-RADIUS PLAYER-RADIUS)))

  ; eat-zombie-brains? : Zombie -> Boolean
  (define (eat-zombie-brains? zombie)
    (< (send (send this p) dist zombie)
       (* 2 ZOMBIE-RADIUS)))

  ; kill : -> Dead
  (define (kill)
    (new dead% (send this p))))
```

```scheme
; A Dead is a (new dead% ...) implements Zombie
(define-class dead%
  (fields p)

  (define (x) (send (send this p) x))
  (define (y) (send (send this p) y))

  ; - move-toward : Player -> Zombie
  ; - eat-zombie-brains? : Zombie -> Boolean
  ; - eat-player-brains? : Player -> Boolean

  (check-expect (send (new dead% (new posn% 10 20)) draw-on (empty-
scene 100 100))
                (place-image (circle ZOMBIE-RADIUS "solid" "gray")
                             10 20
                             (empty-scene 100 100)))
  (define (draw-on scn)
    (send (send this p) draw-on/image (circle ZOMBIE-RADIUS "solid" "gray") scn))

  (check-expect (send (new dead% (new posn% 0 0)) move-toward
                      (new player% (new posn% 0 10)))
                (new dead% (new posn% 0 0)))
  (check-expect (send (new dead% (new posn% 0 0)) move-toward
                      (new player% (new posn% 10 0)))
                (new dead% (new posn% 0 0)))
  (define (move-toward player)
    this)

  ; eat-player-brains? : Player -> Boolean
  (define (eat-player-brains? player)
    (< (send (send this p) dist player)
       (+ ZOMBIE-RADIUS PLAYER-RADIUS)))

  ; eat-zombie-brains? : Zombie -> Boolean
  (define (eat-zombie-brains? zombie)
    (< (send (send this p) dist zombie)
       (* 2 ZOMBIE-RADIUS)))

  ; kill : -> Dead
  (define (kill)
    this))


(define-class posn%
  (fields x y)
  ; Move this position toward that one at given velocity.
```

```
; move-toward : Point Number -> Posn
(define (move-toward that velocity)
  (local [(define delta-x (- (send that x) (send this x)))
          (define delta-y (- (send that y) (send this y)))
          (define move-distance
            (min velocity
                 (max (abs delta-x)
                      (abs delta-y))))]

    (cond [(< (abs delta-x) (abs delta-y))
           ; move along y-axis
           (cond [(positive? delta-y)
                  (send this move 0 move-distance)]
                 [else
                  (send this move 0 (- move-distance))])]
          [else
           ; move along x-axis
           (cond [(positive? delta-x)
                  (send this move move-distance 0)]
                 [else
                  (send this move (- move-distance) 0)])])))

; move : Number Number -> Posn
(define (move delta-x delta-y)
  (new posn%
       (+ (send this x) delta-x)
       (+ (send this y) delta-y)))

; draw-on/image : Image Scene -> Scene
(define (draw-on/image img scn)
  (place-image img
               (send this x)
               (send this y)
               scn))

; dist : Point -> Number
; Compute the distance between this posn and that point.
(define (dist that)
  (sqrt (+ (sqr (- (send that y) (send this y)))
           (sqr (- (send that x) (send this x)))))))


(define-class player%
  (fields p)
  (define (x) (send (send this p) x))
  (define (y) (send (send this p) y))
```

```
; move-toward : Mouse -> Player
; Move this player PLAYER-VELOCITY units toward the given mouse.
(define (move-toward mouse)
  (new player%
       (send (send this p) move-toward mouse PLAYER-VELOCITY)))

(check-expect (send (new player% (new posn% 10 20)) draw-on (empty-
scene 100 100))
                (place-image (circle PLAYER-RADIUS "solid" "green")
                             10 20
                             (empty-scene 100 100)))
(define (draw-on scn)
  (send (send this p) draw-on/image (circle PLAYER-RADIUS "solid" "green") scn)))

(check-expect (send (new posn% 0 0) move-toward (new posn% 0 0) 100)
              (new posn% 0 0))
(check-expect (send (new posn% 0 0) move-toward (new posn% 100 0) 50)
              (new posn% 50 0))
(check-expect (send (new posn% 0 0) move-toward (new posn% 0 100) 50)
              (new posn% 0 50))
(check-expect (send (new posn% 0 0) move-toward (new posn% 100 100) 50)
              (new posn% 50 0))
(check-expect (send (new posn% 0 0) move-toward (new posn% 100 101) 50)
              (new posn% 0 50))
(check-expect (send (new posn% 0 0) move-toward (new posn% 101 100) 50)
              (new posn% 50 0))


(check-expect (send (new player% (new posn% 0 0)) move-toward (new mouse% 0 0))
              (new player% (new posn% 0 0)))
(check-expect (send (new player% (new posn% 0 0)) move-toward (new mouse% 100 0))
              (new player% (new posn% PLAYER-VELOCITY 0)))
(check-expect (send (new player% (new posn% 0 0)) move-toward (new mouse% 0 100))
              (new player% (new posn% 0 PLAYER-VELOCITY)))
(check-expect (send (new player% (new posn% 0 0)) move-toward (new mouse% 100 100))
              (new player% (new posn% PLAYER-VELOCITY 0)))
(check-expect (send (new player% (new posn% 0 0)) move-toward (new mouse% 100 101))
              (new player% (new posn% 0 PLAYER-VELOCITY)))
(check-expect (send (new player% (new posn% 0 0)) move-toward (new mouse% 101 100))
              (new player% (new posn% PLAYER-VELOCITY 0)))


(define-class mouse%
  (fields x y)
```

```
; draw-on : Scene -> Scene
; Draw this mouse on the scene
(check-expect (send (new mouse% 10 20) draw-on (empty-scene 100 100))
              (place-image (square 5 "solid" "red")
                           10
                           20
                           (empty-scene 100 100)))
(define (draw-on scn)
  (place-image (square 5 "solid" "red")
               (send this x)
               (send this y)
               scn)))

; A ZombieApocalypse (ZA) is a (new za% Zombies Player Mouse)
; implements
; - on-tick : -> ZA
; - to-draw : -> Scene
; - on-mouse : MouseEvent Number Number -> ZA
; - stop-when : -> Boolean

(define WIDTH 200)
(define HEIGHT 200)
(define-class za%
  (fields zombies player mouse)
  ; on-tick : -> ZA
  (define (on-tick)
    (new za%
         (send (send (send this zombies) kill-any-touching) move-
toward (send this player))
         (send (send this player) move-toward (send this mouse))
         (send this mouse)))

  ; to-draw : -> Scene
  (define (to-draw)
    (send (send this zombies) draw-on
          (send (send this player) draw-on
                (send (send this mouse) draw-on
                      (empty-scene WIDTH HEIGHT)))))

  ; on-mouse : Number Number MouseEvent -> ZA
  (define (on-mouse mx my me)
    (new za%
         (send this zombies)
         (send this player)
         (new mouse% mx my)))
```

```
; stop-when : -> Boolean
(define (stop-when)
  (send (send this zombies) any-eating-player-brains?
        (send this player))))


(define world0
  (new za%
       (new conz%
            (new undead% (new posn% 60 60))
            (new conz%
                 (new undead% (new posn% 100 100))
                 (new emptyz%)))
       (new player% (new posn% 0 0))
       (new mouse% 0 0)))
```

24.5   Solution: section 5.4.1

This is a solution for the section 5.4.1 exercise.

```
#lang class/0
;; ===========================================================
;; Lists of X

;; INTERFACE
;; ---------
;; A [List X] implements:
;;
;; - cons : X -> [List X]
;;   Cons given value on to this list.
;;
;; - length : -> Natural
;;   Count the number of elements in this list.
;;
;; - append : [List X] -> [List X]
;;   Append the elements of this list with the given list.
;;
;; - reverse : -> [List X]
;;   Reverse the elements of this list.
;;
;; - map [Y] : [X -> Y] -> [List Y]
;;   Apply given function to each element of this list and
;;   construct a list of the results
;;
;; - filter : [X -> Boolean] -> [List X]
```

```
;;   Select elements that satisfy the given predicate.
;;
;; - foldr [Y] : [X Y -> Y] Y -> Y
;;   Fold right over this list.
;;
;; - foldl [Y] : [X Y -> Y] Y -> Y
;;   Fold left over this list.

;; A [Cons X] implements [List X] and:
;;
;; - first : -> X
;;   Get first element of this non-empty list.
;;
;; - rest : -> [List X]
;;   Get the reset of this non-empty list.

;; IMPLEMENTATION
;; --------------
;; A (new empty%) implements [List X]
;; A (new cons% X [Listof X]) implements [Cons X]

(define-class empty%
  (check-expect (send mt cons 0) (new cons% 0 mt))
  (define (cons x) (new cons% x this))

  (check-expect (send mt length) 0)
  (define (length) 0)

  (check-expect (send mt append ls) ls)
  (define (append ls) ls)

  (check-expect (send mt reverse) mt)
  (define (reverse) this)

  (check-expect (send mt reverse-acc ls) ls)
  (define (reverse-acc ls) ls)

  (check-expect (send mt map add1) mt)
  (define (map f) this)

  (check-expect (send mt foldr + 0) 0)
  (define (foldr f b) b)

  (check-expect (send mt foldl + 0) 0)
  (define (foldl f b) b))
```

```
(define-class cons%
  (fields first rest)

  (check-expect (send ls cons 0) (new cons% 0 ls))
  (define (cons x) (new cons% x this))

  (check-expect (send ls length) 3)
  (define (length)
    (add1 (send (send this rest) length)))

  (check-expect (send ls append mt) ls)
  (define (append ls)
    (new cons%
         (send this first)
         (send (send this rest) append ls)))

  (check-expect (send ls reverse)
                (new cons% 4 (new cons% 2 (new cons% 3 mt))))
  (define (reverse)
    (reverse-acc (new empty%)))

  (check-expect (send ls reverse-acc ls)
                (new cons% 4 (new cons% 2 (new cons% 3 ls))))
  ;; ACCUM: elements seen in reverse order.
  (define (reverse-acc ls)
    (send (send this rest) reverse-acc
          (new cons% (send this first) ls)))

  (check-expect (send ls map add1)
                (new cons% 4 (new cons% 3 (new cons% 5 mt))))
  (define (map f)
    (new cons% (f (send this first))
         (send (send this rest) map f)))

  (check-expect (send ls foldr + 0) 9)
  (define (foldr f b)
    (f (this . first) (this . rest . foldr f b)))

  (check-expect (send ls foldl + 0) 9)
  (define (foldl f b)
    (this . rest . foldl f (f (this . first) b))))

(define mt (new empty%))
(define ls (new cons% 3 (new cons% 2 (new cons% 4 mt))))
```

24.6 Solution: Shapes

This is a solution for the Shapes exercise.

```
#lang class/1

;; Solution to part 1.
#;
(define-class rect%
  (fields width height)
  (define (bba)
    (* (this . width) (this . height))))

#;
(define-class circ%
  (fields radius)
  (define (bba)
    (sqr (* 2 (this . radius)))))

;; A Shape implements:
;; width : -> Number
;; height : -> Number
;; Compute the {width,height} of this shape.

;; Solution to part 2.
(define-class shape%
  (define (bba)
    (* (this . width) (this . height))))

(define-class rect%
  (super shape%)
  (fields width height))

(define-class circ%
  (super shape%)
  (fields radius)
  (define (width)
    (* 2 (this . radius)))
  (define (height)
    (this . width)))

;; Solution to part 3.
(define-class square%
```

```
  (super shape%)
  (fields width)
  (define (height)
    (this . width)))

(check-expect ((new rect% 3 4) . bba) 12)
(check-expect ((new circ% 1.5) . bba)  9)
(check-expect ((new square% 5) . bba) 25)
```

24.7   Solution: section 13.7.1

This is a solution for the section 13.7.1 exercise.

```
#lang class/1
;; ========================================================
;; Lists of X with inheritance

;; IMPLEMENTATION
;; --------------
;; A (new empty%) implements [List X], extends list%
;; A (new cons% X [Listof X]) implements [Cons X], extends list%

(define-class list%
  (check-expect (send mt empty) mt)
  (check-expect (send ls empty) mt)
  (define (empty) (new empty%))

  (check-expect (send mt cons 0) (new cons% 0 mt))
  (define (cons x) (new cons% x this))

  (check-expect (send mt length) 0)
  (check-expect (send ls length) 3)
  (define (length)
    (send this foldl (λ (x len) (add1 len)) 0))

  (check-expect (send mt append ls) ls)
  (check-expect (send ls append mt) ls)
  (define (append ls)
    (send this foldr (λ (x ls) (ls . cons x)) ls))

  (check-expect (send mt reverse) mt)
  (check-expect (send ls reverse)
                (new cons% 4 (new cons% 2 (new cons% 3 mt))))
  (define (reverse)
    (send this foldl (λ (x ls) (ls . cons x)) (this . empty)))
```

```
  (check-expect (send mt map add1) mt)
  (check-expect (send ls map add1)
                (new cons% 4 (new cons% 3 (new cons% 5 mt))))
  (define (map f)
    (send this foldr (λ (x ys) (ys . cons (f x))) (this . empty))))

(define-class empty%
  (super list%)

  (check-expect (send mt foldr + 0) 0)
  (define (foldr f b) b)

  (check-expect (send mt foldl + 0) 0)
  (define (foldl f b) b))

(define-class cons%
  (super list%)
  (fields first rest)

  (check-expect (send ls foldr + 0) 9)
  (define (foldr f b)
    (f (this . first) (this . rest . foldr f b)))

  (check-expect (send ls foldl + 0) 9)
  (define (foldl f b)
    (this . rest . foldl f (f (this . first) b))))

(define mt (new empty%))
(define ls (new cons% 3 (new cons% 2 (new cons% 4 mt))))
```

24.8 Solution: Functional programming with objects

This is a solution for the section 14.5.1 exercise.

```
#lang class/1

;; A [IFun X Y] implements:
;; - apply : X -> Y
;;   Apply this function to the given input.
;; - compose : [IFun Y Z] -> [IFun X Z]
;;   Produce a function that applies this function to its input,
```

```
;;   then applies the given function to that result.

;; A (new fun% [X -> Y]) implements [IFun X Y].
(define-class fun%
  (fields f)

  (define (apply x)
    ((this . f) x))

  (define (compose g)
    (new fun% (λ (x)
                (g . apply (this . apply x))))))

(define addone (new fun% add1))
(define subone (new fun% sub1))

(check-expect ((addone . compose subone) . apply 5) 5)
(check-expect ((addone . compose addone) . apply 5) 7)
```

Index