Teachpacks

Version 4.1.2

October 28, 2008

Teaching languages are small subsets of a full programming language. While such restrictions simplify error diagnosis and the construction of tools, they also make it impossible (or at least difficult) to write some interesting programs. To circumvent this restriction, it is possible to import teachpacks into programs written in a teaching language.

In principle, a teachpack is just a library written in the full language, not the teaching subset. Like any other library, it may export values, functions, etc. In contrast to an ordinary library, however, a teachpack must enforce the contracts of the ”lowest” teaching language into which it is imported and signal errors in a way with which students are familiar at that level.

This chapter covers the teachpacks for *How to Design Programs* and *How to Design Classes*.

Contents

1 HtDP Teachpacks

1.1 Manipulating Images: `"image.ss"`

The teachpack provides primitives for constructing and manipulating images. Basic, colored images are created as outlines or solid shapes. Additional primitives allow for the composition of images.

1.1.1 Images

```
(image? x) → boolean?
  x : any/c
```

Is *x* an image?

1.1.2 Modes and Colors

Mode `(one-of/c 'solid 'outline "solid" "outline")`

A Mode is used to specify whether painting a shape fills or outlines the form.

```
(struct color (red green blue))
  red : (and/c natural-number/c (<=/c 255))
  green : (and/c natural-number/c (<=/c 255))
  blue : (and/c natural-number/c (<=/c 255))
```

RGB `color?`

A RGB describes a color via a shade of red, blue, and green colors (e.g., `(make-color 100 200 30)`).

Color `(or/c symbol? string? color?)`

A Color is a color-symbol (e.g., `'blue`) or a color-string (e.g., `"blue"`) or an RGB structure.

```
(image-color? x) → boolean?
  x : any
```

Determines if the input is a valid image Color.

1.1.3 Creating Basic Shapes

In DrScheme, you can insert images from your file system. Use PNG images instead whenever possible for insertions. In addition, you can create basic shapes with the following functions.

`(rectangle w h m c) → image?`
  `w` : `(and/c number? (or/c zero? positive?))`
  `h` : `(and/c number? (or/c zero? positive?))`
  `m` : Mode
  `c` : Color

Creates a `w` by `h` rectangle, filled in according to `m` and painted in color `c`

`(circle r m c) → image?`
  `r` : `(and/c number? (or/c zero? positive?))`
  `m` : Mode
  `c` : Color

Creates a circle or disk of radius `r`, filled in according to `m` and painted in color `c`

`(ellipse w h m c) → image?`
  `w` : `(and/c number? (or/c zero? positive?))`
  `h` : `(and/c number? (or/c zero? positive?))`
  `m` : Mode
  `c` : Color

Creates a `w` by `h` ellipse, filled in according to `m` and painted in color `c`

`(triangle s m c) → image?`
  `s` : `number?`
  `m` : Mode
  `c` : Color

Creates an upward pointing equilateral triangle whose side is `s` pixels long, filled in according to `m` and painted in color `c`

`(star n outer inner m c) → image?`
  `n` : `(and/c number? (>=/c 2))`
  `outer` : `(and/c number? (>=/c 1))`
  `inner` : `(and/c number? (>=/c 1))`
  `m` : Mode

c : Color

Creates a multi-pointed star with *n* points, an *outer* radius for the max distance of the points to the center, and an *inner* radius for the min distance to the center.

(regular-polygon *s r m c* [*angle*]) → image?
  s : side
  r : number?
  m : Mode
  c : Color
  angle : real? = 0

Creates a regular polygon with *s* sides inscribed in a circle of radius *r*, using mode *m* and color *c*. If an angle is specified, the polygon is rotated by that angle.

(line *x y c*) → image?
  x : number?
  y : number?
  c : Color

Creates a line colored *c* from (0,0) to (*x* ,*y*). See add-line below.

(text *s f c*) → Image
  s : string?
  f : (and/c number? positive?)
  c : Color

Creates an image of the text *s* at point size *f* and painted in color *c*.

1.1.4 Basic Image Properties

To understand how images are manipulated, you need to understand the basic properties of images.

(image-width *i*) → integer?
  i : image?

Obtain *i*'s width in pixels

(image-height *i*) → integer?
  i : image?

Obtain `i`'s height in pixels

For the composition of images, you must know about *pinholes*. Each image, including primitive ones, come with a pinhole. For images created with the above primitives, the pinhole is at the center of the shape except for those created from `line` and `text`. The `text` function puts the pinhole at the upper left corner of the image, and `line` puts the pinhole at the beginning of the line (meaning that if the first two arguments to `line` are positive, the pinhole is also in the upper left corner). The pinhole can be moved, of course, and compositions locate pinholes according to their own rules. When in doubt you can always find out where the pinhole is and place it where convenient.

```
(pinhole-x i) → integer?
  i : image?
```

Determines the `x` coordinate of the pinhole, measuring from the left of the image.

```
(pinhole-y i) → integer?
  i : image?
```

Determines the `y` coordinate of the pinhole, measuring from the top (down) of the image.

```
(put-pinhole i x y) → image?
  i : image?
  x : number?
  y : number?
```

Creates a new image with the pinhole in the location specified by `x` and `y`, counting from the left and top (down), respectively.

```
(move-pinhole i delta-x delta-y) → image?
  i : image?
  delta-x : number?
  delta-y : number?
```

Creates a new image with the pinhole moved down and right by `delta-x` and `delta-y` with respect to its current location. Use negative numbers to move it up or left.

1.1.5 Composing Images

Images can be composed, and images can be found within compositions.

```
(add-line i x y z u c) → image?
  i : image?
  x : number?
  y : number?
  z : number?
  u : number?
  c : Color
```

Creates an image by adding a line (colored `c`) from `(x ,y)` to `(z ,u)` to image `i`.

```
(overlay img img2 img* ...) → image?
  img : image?
  img2 : image?
  img* : image?
```

Creates an image by overlaying all images on their pinholes. The pinhole of the resulting image is the same place as the pinhole in the first image.

```
(overlay/xy img delta-x delta-y other) → image?
  img : image?
  delta-x : number?
  delta-y : number?
  other : image?
```

Creates an image by adding the pixels of `other` to `img`.

Instead of lining the two images up on their pinholes, `other`'s pinhole is lined up on the point:

```
(make-posn (+ (pinhole-x img) delta-x)
           (+ (pinhole-y img) delta-y))
```

The pinhole of the resulting image is the same place as the pinhole in the first image.

The same effect can be had by combining `move-pinhole` and `overlay`,

```
(overlay img
         (move-pinhole other
                       (- delta-x)
                       (- delta-y)))
```

```
(image-inside? img other) → boolean?
  img : image?
  other : image?
```

Determines whether the pixels of the second image appear in the first.

Be careful when using this function with jpeg images. If you use an image-editing program to crop a jpeg image and then save it, `image-inside?` does not recognize the cropped image, due to standard compression applied to JPEG images.

```
(find-image img other) → posn?
  img : image?
  other : image?
```

Determines where the pixels of the second image appear in the first, with respect to the pinhole of the first image. If `(image-inside? img other)` isn't true, `find-image` signals an error.

1.1.6 Manipulating Images

Images can also be shrunk. These "shrink" functions trim an image by eliminating extraneous pixels.

```
(shrink-tl img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a *width* by *height* image, starting from the *top-left* corner. The pinhole of the resulting image is in the center of the image.

```
(shrink-tr img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a *width* by *height* image, starting from the *top-right* corner. The pinhole of the resulting image is in the center of the image.

```
(shrink-bl img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a *width* by *height* image, starting from the *bottom-left* corner. The

pinhole of the resulting image is in the center of the image.

```
(shrink-br img width height) → image?
  img : image?
  width : number?
  height : number?
```

Shrinks the image to a *width* by *height* image, starting from the *bottom-right* corner. The pinhole of the resulting image is in the center of the image.

```
(shrink img left above right below) → image?
  img : image?
  left : number?
  above : number?
  right : number?
  below : number?
```

Shrinks an image around its pinhole. The numbers are the pixels to save to left, above, to the right, and below the pinhole, respectively. The pixel directly on the pinhole is always saved.

1.1.7 Miscellaneous Image Manipulation and Creation

The last group of functions extracts the constituent colors from an image and converts a list of colors into an image.

```
List-of-color : list?
```

is one of:

```
  ; -- empty
  ; -- (cons Color List-of-color)
  ; Interpretation: represents a list of colors.
```

```
(image->color-list img) → List-of-color
  img : image?
```

Converts an image to a list of colors.

```
(color-list->image l width height x y) → image?
  l : List-of-color
  width : natural-number/c
```

```
  height : natural-number/c
  x : natural-number/c
  y : natural-number/c
```

Converts a list of colors *l* to an image with the given *width* and *height* and pinhole (*x*,*y*) coordinates, specified with respect to the top-left of the image.

The remaining functions provide alpha-channel information as well. Alpha channels are a measure of transparency; 0 indicates fully opaque and 255 indicates fully transparent.

```
(struct alpha-color (alpha red green blue))
  alpha : (and/c natural-number/c (<=/c 255))
  red : (and/c natural-number/c (<=/c 255))
  green : (and/c natural-number/c (<=/c 255))
  blue : (and/c natural-number/c (<=/c 255))
```

A structure representing an alpha color.

```
(image->alpha-color-list img) → (list-of alpha-color?)
  img : image?
```

to convert an image to a list of alpha colors

```
(alpha-color-list->image l width height x y) → image?
  l : (list-of alpha-color?)
  width : integer?
  height : integer?
  x : integer?
  y : integer?
```

Converts a list of alpha-colors *l* to an image with the given *width* and *height* and pinhole (*x*,*y*) coordinates, specified with respect to the top-left of the image.

1.2 Simulations and Animations: "world.ss"

Note: For a quick and educational introduction to the teachpack, see How to Design Programs, Second Edition: Prologue. As of August 2008, we also have a series of projects available as a small booklet on How to Design Worlds.

The purpose of this documentation is to give experienced Schemers a concise overview for using the library and for incorporating it elsewhere. The last section presents §1.2.5 "A First Example" for an extremely simple domain and is suited for a novice who knows how to

design conditional functions for symbols.

The teachpack provides two sets of tools. The first allows students to create and display a series of animated scenes, i.e., a simulation. The second one generalizes the first by adding interactive GUI features.

1.2.1 Basics

The teachpack assumes working knowledge of the basic image manipulation primitives and introduces a special kind of image: a scene.

Scene

```
(define (focus-at-0-0 i)
  (and (= (pinhole-x i) 0) (= (pinhole-y i) 0)))

(and/c image? focus-at-0-0)
```

The teachpack can display only Scenes, which are images whose pinholes are at position (0 ,0).

(empty-scene *width height*) → Scene
  width : natural-number/c
  height : natural-number/c

Creates a *width* x *height* Scene.

(place-image *img x y s*) → Scene
  img : image?
  x : number?
  y : number?
  s : Scene

Creates a scene by placing *img* at (*x* ,*y*) into *s*; (*x* ,*y*) are comp. graph. coordinates, i.e., they count right and down from the upper-left corner.

1.2.2 Simple Simulations

(run-simulation *w h r create-image* [*gifs?*]) → true
  w : natural-number/c
  h : natural-number/c

```
r : number?
create-image : (-> natural-number/c scene)
gifs? : boolean? = #f
```

creates and shows a canvas of width *w* and height *h* , starts a clock, making it tick every *r* (usually fractional) seconds. Every time the clock ticks, drscheme applies *create-image* to the number of ticks passed since this function call. The results of these applications are displayed in the canvas.

The fifth (and last) argument is optional. Providing `true` as the fifth argument causes drscheme to collect the scenes that the animation generates and to create an animated GIF from the results. Both the intermediate images as well as the final animated GIF are saved in a user-specified directory. This is useful for writing documentation and for describing students work.

Example:

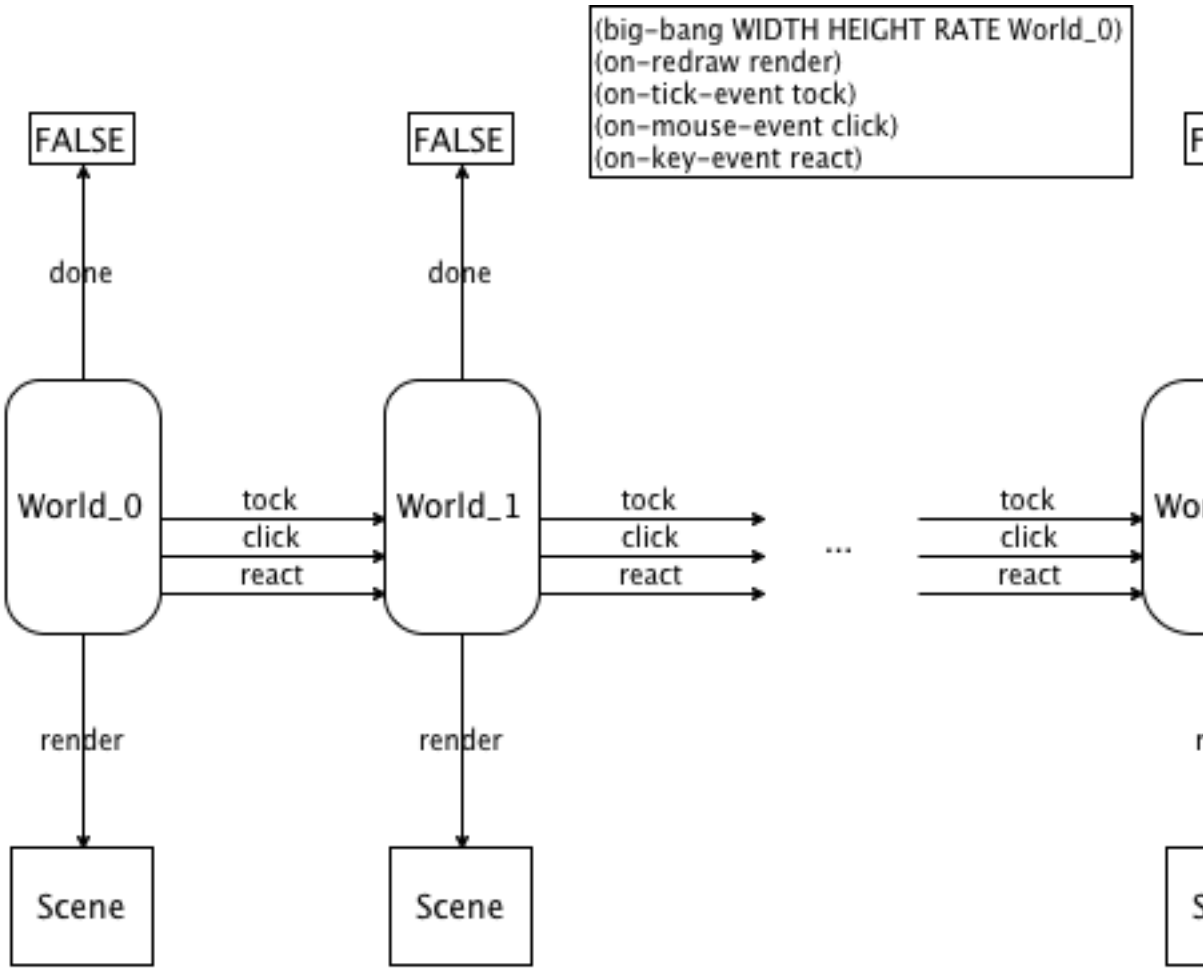
```
(define (create-UFO-scene height)
  (place-image UFO 50 height (empty-scene 100 100)))

(define UFO
  (overlay (circle 10 'solid 'green)
           (rectangle 40 4 'solid 'green)))

(run-simulation 100 100 (/ 1 28) create-UFO-scene)
```

1.2.3 Interactions

An animation starts from a given "world" and generates new ones in response to events on the computer. This teachpack keeps track of the "current world" and recognizes three kinds of events: clock ticks; keyboard presses and releases; and mouse movements, mouse clicks, etc.

Your program may deal with such events via the *installation* of *handlers*. The teachpack provides for the installation of three event handlers: `on-tick-event`, `on-key-event`, and `on-mouse-event`. In addition, it provides for the installation of a `draw` handler, which is called every time your program should visualize the current world.

The following picture provides an intuitive overview of the workings of "world".



The `big-bang` function installs *World_0* as the initial world; the callbacks *tock*, *react*, and *click* transform one world into another one; *done* checks each time whether the world is final; and *draw* renders each world as a scene.

World `any/c`

For animated worlds and games, using the teachpack requires that you provide a data definition for World. In principle, there are no constraints on this data definition. You can even keep it implicit, even if this violates the Design Recipe.

`(big-bang` *width height r world0*`)` → `true`

```
  width : natural-number/c
  height : natural-number/c
  r : number?
  world0 : World
(big-bang width height r world0 animated-gif?) → true
  width : natural-number/c
  height : natural-number/c
  r : number?
  world0 : World
  animated-gif? : boolean?
```

Creates and displays a *width* x *height* canvas, starts the clock, makes it tick every *r* seconds, and makes *world0* the current world. If it is called with five instead of four arguments and the last one (*animated-gif?*) is `true`, the teachpack allows the generation of images from the animation, including an animated GIF image.

```
(on-tick-event tock) → true
  tock : (-> World World)
```

Tell DrScheme to call *tock* on the current world every time the clock ticks. The result of the call becomes the current world.

KeyEvent `(or/c char? symbol?)`

A KeyEvent represents key board events, e.g., keys pressed or released, by the computer's user. A `char?` KeyEvent is used to signal that the user has hit an alphanumeric key. Symbols such as `'left`, `'right`, `'up`, `'down`, `'release` denote arrow keys or special events, such as releasing the key on the keypad.

```
(key-event? x) → boolean?
  x : any
```

is *x* a KeyEvent

```
(key=? x y) → boolean?
  x : key-event?
  y : key-event?
```

compares two KeyEvent for equality

```
(on-key-event change) → true
  change : (-> World key-event? World)
```

Tell DrScheme to call *change* on the current world and a KeyEvent for every keystroke the user of the computer makes. The result of the call becomes the current world.

Here is a typical key-event handler:

```
(define (change w a-key-event)
  (cond
    [(key=? a-key-event 'left)  (world-go w -DELTA)]
    [(key=? a-key-event 'right) (world-go w +DELTA)]
    [(char? a-key-event) w] ; to demonstrate order-free checking
    [(key=? a-key-event 'up)    (world-go w -DELTA)]
    [(key=? a-key-event 'down)  (world-go w +DELTA)]
    [else w]))
```

MouseEvent `(one-of/c 'button-down 'button-up 'drag 'move 'enter 'leave)`

A MouseEvent represents mouse events, e.g., mouse movements or mouse clicks, by the computer's user.

`(on-mouse-event clack)` → `true`
  clack : `(-> World natural-number/c natural-number/c MouseEvent World)`

Tell DrScheme to call *clack* on the current world, the current `x` and `y` coordinates of the mouse, and and a MouseEvent for every action of the mouse by the user of the computer. The result of the call becomes the current world.

`(on-redraw to-scene)` → `true`
  to-scene : `(-> World Scene)`

Tell DrScheme to call *to-scene* whenever the canvas must be redrawn. The canvas is usually re-drawn after a tick event, a keyboard event, or a mouse event has occurred. The generated scene is displayed in the world's canvas.

`(stop-when last-world?)` → `true`
  last-world? : `(-> World boolean?)`

Tell DrScheme to call *last-world?* whenever the canvas is drawn. If this call produces `true`, the clock is stopped; no more tick events, KeyEvents, or MouseEvents are forwarded to the respective handlers. As a result, the canvas isn't updated either.

Example: The following examples shows that `(run-simulation 100 100 (/ 1 28) create-UFO-scene)` is a short-hand for three lines of code:

```
(define (create-UFO-scene height)
  (place-image UFO 50 height (empty-scene 100 100)))
```

```
(define UFO
  (overlay (circle 10 'solid 'green)
           (rectangle 40 4 'solid 'green)))

(big-bang 100 100 (/1 28) 0)
(on-tick-event add1)
(on-redraw create-UFO-scene)
```

Exercise: Add a condition for stopping the flight of the UFO when it reaches the bottom.

1.2.4 Scenes and Images

For the creation of scenes from the world, use the functions from §1.1 "Manipulating Images: `"image.ss"`". The following two functions have turned out to be useful for the creation of scenes, too.

`(nw:rectangle` *width height solid-or-filled c*`)` → `image?`
width : `natural-number/c`
height : `natural-number/c`
solid-or-filled : Mode
c : Color

Creates a *width* x *height* rectangle, solid or outlined as specified by *solid-or-filled* and colored according to *c*, with a pinhole at the upper left corner.

`(scene+line` *s x0 y0 x1 y1 c*`)` → Scene
s : Scene
x0 : `number?`
y0 : `number?`
x1 : `number?`
y1 : `number?`
c : Color

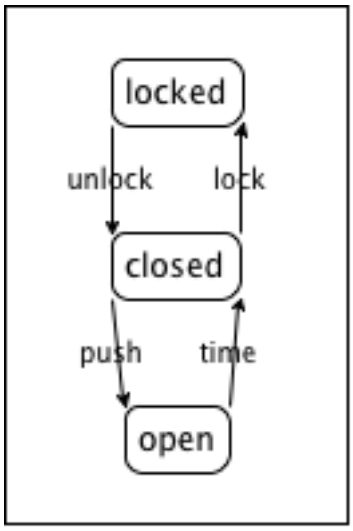
Creates a scene by placing a line of color *c* from `(`*x0* `,`*y0*`)` to `(`*x1* `,`*y1*`)` into `scene`; `(x ,y)` are comp. graph. coordinates; in contrast to the `add-line` function, this one cuts off those portions of the line that go beyond the boundaries of the given *s*.

1.2.5 A First Example

Understanding a Door

Say we want to represent a door with an automatic door closer. If this kind of door is locked, you can unlock it. While this doesn't open the door per se, it is now possible to do so. That is, an unlocked door is closed and pushing at the door opens it. Once you have passed through the door and you let go, the automatic door closer takes over and closes the door again. Of course, at this point you could lock it again.

Here is a picture that translates our words into a graphical representation:



The picture displays a so-called "state machine". The three circled words are the states that our informal description of the door identified: locked, closed (and unlocked), and open. The arrows specify how the door can go from one state into another. For example, when the door is open, the automatic door closer shuts the door as time passes. This transition is indicated by the arrow labeled "time passes." The other arrows represent transitions in a similar manner:

- "push" means a person pushes the door open (and let's go);
- "lock" refers to the act of inserting a key into the lock and turning it to the locked position; and
- "unlock" is the opposite of "lock".

Simulations of the World

Simulating any dynamic behavior via a program demands two different activities. First, we must tease out those portions of our "world" that change over time or in reaction to actions, and we must develop a data representation *D* for this information. Keep in mind that a good data definition makes it easy for readers to map data to information in the real world and vice versa. For all others aspects of the world, we use global constants, including graphical or visual constants that are used in conjunction with the rendering operations.

Second, we must translate the "world" actions—the arrows in the above diagram—into interactions with the computer that the world teachpack can deal with. Once we have decided to use the passing of time for one aspect and mouse movements for another, we must develop functions that map the current state of the world—represented as data—into the next state of the world. Since the data definition D describes the class of data that represents the world, these functions have the following general contract and purpose statements:

```
; tick : D -> D
; deal with the passing of time
(define (tick w) ...)

; click : D "Number" "Number" MouseEvent -> D
; deal with a mouse click at (x,y) of kind "me"
; in the current world "w"
(define (click w x y me) ...)

; control : D KeyEvent -> D
; deal with a key event (symbol, char) "ke"
; in the current world "w"
(define (control w ke) ...)
```

That is, the contracts of the various hooks dictate what the contracts of these functions are once we have defined how to represent the world in data.

A typical program does not use all three of these actions and functions but often just one or two. Furthermore, the design of these functions provides only the top-level, initial design goal. It often demands the design of many auxiliary functions.

Simulating a Door: Data

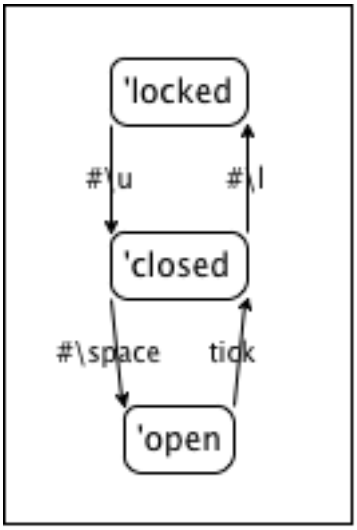
Our first and immediate goal is to represent the world as data. In this specific example, the world consists of our door and what changes about the door is whether it is locked, unlocked but closed, or open. We use three symbols to represent the three states:

SD

```
; DATA DEF.
; The state of the door (SD) is one of:
; -- 'locked
; -- 'closed
; -- 'open
```

Symbols are particularly well-suited here because they directly express the state of the door.

Now that we have a data definition, we must also decide which computer actions and interactions should model the various actions on the door. Our pictorial representation of the door's states and transitions, specifically the arrow from "open" to "closed" suggests the use of a function that simulates time. For the other three arrows, we could use either keyboard events or mouse clicks or both. Our solution uses three keystrokes: `"#\\u"` for unlocking the door, `"#\\l"` for locking it, and `"#\\space"` for pushing it open. We can express these choices graphically by translating the above "state machine" from the world of information into the world of data:



Simulating a Door: Functions

Our analysis and data definition leaves us with three functions to design:

- `"automatic-closer"`, which closes the time during one tick;
- `"door-actions"`, which manipulates the time in response to pressing a key; and
- `"render"`, which translates the current state of the door into a visible scene.

Let's start with "automatic-closer". We know its contract and it is easy to refine the purpose statement, too:

```
; automatic-closer : SD -> SD
; closes an open door over the period of one tick
(define (automatic-closer state-of-door) ...)
```

Making up examples is trivial when the world can only be in one of three states:

given state	desired state
'locked	'locked
'closed	'closed
'open	'closed

```
; automatic-closer : SD -> SD
; closes an open door over the period of one tick

(check-expect (automatic-closer 'locked) 'locked)
(check-expect (automatic-closer 'closed) 'closed)
(check-expect (automatic-closer 'open) 'closed)

(define (automatic-closer state-of-door) ...)
```

The template step demands a conditional with three clauses:

```
(define (automatic-closer state-of-door)
  (cond
    [(symbol=? 'locked state-of-door) ...]
    [(symbol=? 'closed state-of-door) ...]
    [(symbol=? 'open state-of-door) ...]))
```

The examples basically dictate what the outcomes of the three cases must be:

```
(define (automatic-closer state-of-door)
  (cond
    [(symbol=? 'locked state-of-door) 'locked]
    [(symbol=? 'closed state-of-door) 'closed]
    [(symbol=? 'open state-of-door) 'closed]))
```

Don't forget to run the example-tests.

For the remaining three arrows of the diagram, we design a function that reacts to the three chosen keyboard events. As mentioned, functions that deal with keyboard events consume both a world and a keyevent:

```
; door-actions : SD Keyevent -> SD
```

```
; key events simulate actions on the door
(define (door-actions s k) ...)
```

given state	given keyevent	desired state
'locked	#\u	'closed
'closed	#\l	'locked
'closed	#\space	'open
'open	—	'open

The examples combine what the above picture shows and the choices we made about mapping actions to keyboard events.

From here, it is straightforward to turn this into a complete design:

```
(define (door-actions s k)
  (cond
    [(and (symbol=? 'locked s) (key=? #\u k)) 'closed]
    [(and (symbol=? 'closed s) (key=? #\l k)) 'locked]
    [(and (symbol=? 'closed s) (key=? #\space k)) 'open]
    [else s]))

(check-expect (door-actions 'locked #\u) 'closed)
(check-expect (door-actions 'closed #\l) 'locked)
(check-expect (door-actions 'closed #\space) 'open)
(check-expect (door-actions 'open 'any) 'open)
(check-expect (door-actions 'closed 'any) 'closed)
```

Last but not least we need a function that renders the current state of the world as a scene. For simplicity, let's just use a large enough text for this purpose:

```
; render : SD -> "Scene"
; translate the current state of the door into a large text
(define (render s)
  (text (symbol->string s) 40 'red))

(check-expecy (render 'closed) (text "closed" 40 'red))
```

The function `"symbol->string"` translates a symbol into a string, which is needed because `"text"` can deal only with the latter, not the former. A look into the language documentation revealed that this conversion function exists, and so we use it.

Once everything is properly designed, it is time to *run* the program. In the case of the world teachpack, this means we must specify which function takes care of tick events, key events, and redraws:

```
(big-bang 100 100 1 'locked)
```

```
(on-tick-event automatic-closer)
(on-key-event door-actions)
(on-redraw render)
```

Now it's time for you to collect the pieces and run them in DrScheme to see whether it all works.

1.3 Converting Temperatures: "convert.ss"

The teachpack convert.ss provides three functions for converting Fahrenheit temperatures to Celsius. It is useful for a single exercise in HtDP. Its purpose is to demonstrate the independence of "form" (user interface) and "function" (also known as "model").

```
(convert-gui convert) → true
  convert : (-> number? number?)
```

Consumes a conversion function from Fahrenheit to Celsius and creates a graphical user interface with two rulers, which users can use to convert temperatures according to the given temperature conversion function.

```
(convert-repl convert) → true
  convert : (-> number? number?)
```

Consumes a conversion function from Fahrenheit to Celsius and then starts a read-evaluate-print loop. The loop prompts users to enter a number and then converts the number according to the given temperature conversion function. A user can exit the loop by entering "x."

```
(convert-file in convert out) → true
  in : string?
  convert : (-> number? number?)
  out : string?
```

Consumes a file name *in*, a conversion function from Fahrenheit to Celsius, and a string *out*. The program then reads all the number from *in*, converts them according to *convert*, and prints the results to the newly created file *out*.

Warning: If *out* already exists, it is deleted.

Example: Create a file with name "in.dat" with some numbers in it, using your favorite text editor on your computer. Define a function f2c in the Definitions window and set teachpack to "convert.ss" and click RUN. Then evaluate

```
(convert-gui f2c)
```

```
; and
(convert-file "in.dat" f2c "out.dat")
; and
(convert-repl f2c)
```

Finally inspect the file `"out.dat"` and use the repl to check the answers.

1.4 Guessing Numbers: `"guess.ss"`

The teachpack provides operations to play a guess-the-number game. Each operation display a GUI in which a player can choose specific values for some number of digits and then check the guess. The more advanced operations ask students to implement more of the game.

```
(guess-with-gui check-guess) → true
  check-guess : (-> number? number? symbol?)
```

The *check-guess* function consumes two numbers: `guess`, which is the user's guess, and `target`, which is the randomly chosen number-to-be-guessed. The result is a symbol that reflects the relationship of the player's guess to the target.

```
(guess-with-gui-3 check-guess) → true
  check-guess : (-> digit? digit? digit? number? symbol?)
```

The *check-guess* function consumes three digits (`digit0`, `digit1`, `digit2`) and one number (`target`). The latter is the randomly chosen number-to-be-guessed; the three digits are the current guess. The result is a symbol that reflects the relationship of the player's guess (the digits converted to a number) to the target.

Note: `digit0` is the *least* significant digit that the user chose and `digit2` is the *most* significant one.

```
(guess-with-gui-list check-guess) → true
  check-guess : (-> (list-of digit?) number? symbol?)
```

The *check-guess* function consumes a list of digits (`digits`) and a number (`target`). The former is a list that makes up the user's guess, and the latter is the randomly chosen number-to-be-guessed. The result is a symbol that reflects the relationship of the player's guess (the digits converted to a number) to the target.

Note: the first item on `digits` is the *least* significant digit that the user chose, and the last one is the *most* significant digit.

1.5 MasterMinding: "master.ss"

The teachpack implements GUI for playing a simple master mind-like game, based on a function designed by a student. The player clicks on two colors and the program responds with an answer that indicates how many colors and places were correct.

```
(master check-guess) → symbol?
  check-guess : (-> symbol? symbol? symbol? symbol? boolean?)
```

Chooses two "secret" colors and then opens a graphical user interface for playing *Master-Mind*. The player is prompted to choose two colors, via a choice tablet and mouse clicks. Once chosen, `master` uses *check-guess* to compare them.

If the two guesses completely match the two secret colors, *check-guess* must return `'PerfectGuess`; otherwise it must return a different, informative symbol.

1.6 Simple Drawing: "draw.ss"

The teachpack provides two sets of functions: one for drawing into a canvas and one for reacting to canvas events.

Warning: *This teachpack is deprecated. Unless you're solving exercises taken from How To Design Programs, we strongly encourage you to use the world teachpack instead; see §1.2 "Simulations and Animations: "world.ss"".*

1.6.1 Drawing on a Canvas

DrawColor: `(and/c symbol? (one-of/c 'white 'yellow 'red 'blue 'green 'black))` These six colors are definitely provided. If you want other colors, guess! For example, `'orange` works, but `'mauve` doesn't. If you apply the function to a symbol that it doesn't recognize as a color, it raises an error.

```
(start width height) → true
  width : number?
  height : number?
```

Opens a *width* x *height* canvas.

```
(start/cartesian-plane width height) → true
  width : number?
  height : number?
```

Opens a *width* x *height* canvas and draws a Cartesian plane.

(stop) → true

Closes the canvas.

(draw-circle *p* *r* *c*) → true
  p : posn?
  r : number?
  c : DrawColor

Draws a *c* circle at *p* with radius *r*.

(draw-solid-disk *p* *r* *c*) → true
  p : posn?
  r : number?
  c : DrawColor

Draws a *c* disk at *p* with radius *r*.

(draw-solid-rect *ul* *width* *height* *c*) → true
  ul : posn?
  width : number?
  height : number?
  c : DrawColor

Draws a *width* x *height*, *c* rectangle with the upper-left corner at *ul*.

(draw-solid-line *strt* *end* *c*) → true
  strt : posn?
  end : posn?
  c : DrawColor

Draws a *c* line from *strt* to *end*.

(draw-solid-string *p* *s*) → true
  p : posn?
  s : string?

Draws *s* at *p*.

`(sleep-for-a-while s)` → `true`
  `s` : `number?`

Suspends evaluation for `s` seconds.

The teachpack also provides `clear-` operations for each `draw-` operation. The arguments are the same. Note: use `clear-rectangle` instead of `clear-string` for now. The color argument for all `clear-` functions are optional.

1.6.2 Interactions with Canvas

`(wait-for-mouse-click)` → `posn?`

Waits for the user to click on the mouse, within the canvas.

DrawKeyEvent: `(or/c char? symbol?)` A DrawKeyEvent represents keyboard events:

- `char?`, if the user pressed an alphanumeric key;
- `symbol?`, if the user pressed, for example, an arror key: `'up` `'down` `'left` `'right`

`(get-key-event)` → `(or/c false` DrawKeyEvent`)`

Checks whether the user has pressed a key within the window; `false` if not.

DrawWorld: For proper interactions, using the teachpack requires that you provide a data definition for DrawWorld . In principle, there are no constraints on this data definition. You can even keep it implicit, even if this violates the Design Recipe.

The following functions allow programs to react to events from the canvas.

`(big-bang n w)` → `true`
  `n` : `number?`
  `w` : DrawWorld

Starts the clock, one tick every `n` (fractal) seconds; `w` becomes the first "current" world.

`(on-key-event change)` → `true`
  `change` : `(->` DrawKeyEvent DrawWorld DrawWorld`)`

Adds *change* to the world. The function reacts to keyboard events and creates a new DrawWorld.

```
(on-tick-event tock) → true
  tock : (-> DrawWorld DrawWorld)
```

Adds *tock* to the world. The function reacts to clock tick events, creating a new current world.

```
(end-of-time) → DrawWorld
```

Stops the world; returns the current world.

1.7 Hangman: `"hangman.ss"`

The teachpack implements the callback functions for playing a *Hangman* game, based on a function designed by a student. The player guesses a letter and the program responds with an answer that indicates how many times, if at all, the letter occurs in the secret word.

The teachpack provides all the drawing operations from §1.6 "Simple Drawing: `"draw.ss"`" for managing a canvas into which the "hangman" is drawn.

```
(hangman make-word reveal draw-next-part) → true
  make-word : (-> symbol? symbol? symbol? word?)
  reveal : (-> word? word? word?)
  draw-next-part : (-> symbol? true)
```

Chooses a "secret" three-letter word and uses the given functions to manage the *Hangman* game.

```
(hangman-list reveal-for-list
              draw-next-part) → true
  reveal-for-list : (-> symbol? (list-of symbol?) (list-of symbol?)
                        (list-of symbol?))
  draw-next-part : (-> symbol? true)
```

Chooses a "secret" word—a list of symbolic letters—and uses the given functions to manage the *Hangman* game: *reveal-for-list* determines how many times the chosen letter occurs in the secret word; *draw-next-part* is given the symbolic name of a body part and draws it on a separately managed canvas.

1.8 Managing Control Arrows: "arrow.ss"

The teachpack implements a controller for moving shapes across a canvass. A shape is a class of data for which move and draw operations can be drawn.

```
(control-left-right shape n move draw) → true
  shape : Shape
  n : number?
  move : (-> number? Shape Shape)
  draw : (-> Shape true)
```

Moves shape *n* pixels left (negative) or right (positive).

```
(control-up-down shape n move draw) → true
  shape : Shape
  n : number?
  move : (-> number? Shape Shape)
  draw : (-> Shape true)
```

Moves shape *n* pixels up (negative) or down (positive).

```
(control shape n move-lr move-ud draw) → true
  shape : Shape
  n : number?
  move-lr : (-> number? Shape Shape)
  move-ud : (-> number? Shape Shape)
  draw : (-> Shape true)
```

Moves shape N pixels left or right and up or down, respectively.

Example:

```
; A shape is a structure:
; (make-posn num num)

; RAD : the radius of the simple disk moving across a canvas
(define RAD 10)

; move : number shape -> shape or false
; to move a shape by delta according to translate
; effect: to redraw it
(define (move delta sh)
  (cond
    [(and (clear-solid-disk sh RAD)
```

```
          (draw-solid-disk (translate sh delta) RAD))
     (translate sh delta)]
    [else false]))

; translate : shape number -> shape
; to translate a shape by delta in the x direction
(define (translate sh delta)
  (make-posn (+ (posn-x sh) delta) (posn-y sh)))

; draw-it : shape -> true
; to draw a shape on the canvas: a disk with radius
(define (draw-it sh)
  (draw-solid-disk sh RAD))

; RUN:

; this creates the canvas
(start 100 50)

; this creates the controller GUI
(control-left-right (make-posn 10 20) 10 move draw-it)
```

1.9 Manipulating Simple HTML Documents: `"docs.ss"`

The teachpack provides three operations for creating simple "HTML" documents:

Annotation An Annotation is a symbol that starts with "<" and ends in ">". An end annotation is one that starts with "</".

`(atom? x)` → `boolean?`
  `x` : `any/c`

Determines whether or not a Scheme value is a number, a symbol, or a string.

`(annotation? x)` → `boolean?`
  `x` : `any/c`

Determines whether or not a Scheme symbol is a document annotation.

`(end-annotation x)` → Annotation
  `x` : Annotation

Consumes an annotation and produces a matching ending annotation.

```
(write-file l) → true
  l : (list-of atom)
```

Consumes a list of symbols and annotations and prints them out as a ”file”.

Sample session: set teachpack to “docs.ss”> and click RUN:

```
> (annotation? 0)
false
> (annotation? '<bold>)
true
> (end-annotation 0)
end-annotation: not an annotation: 0
> (write-file (list 'a 'b))
a b
```

1.10 Working with Files and Directories: `"dir.ss"`

The teachpack provides structures and operations for working with files and directories:

```
(struct dir (name dirs files))
  name : string?
  dirs : (list-of dir?)
  files : (list-of file?)
```

```
(struct file (name content))
  name : string?
  content : (list-of char?)
```

```
(create-dir path) → dir?
  path : string?
```

Turns the directory found at *path* on your computer into an instance of `dir?`.

Sample: Set teachpack to <code>dir.ss</code> and click RUN:

```
> (create-dir ".")
(make-dir
  '|.|
```

```
empty
(cons (make-file 'ball1.gif 1289 empty)
      (cons (make-file 'blueball.gif 205 empty)
            (cons (make-file 'greenbal.gif 204 empty)
                  (cons (make-file 'redball.gif 203 empty)
                        (cons (make-file 'ufo.gif 1044 empty)
                              (cons (make-file 'gif-test.ss 5811 empty)
                                    empty)))))))
```

Using "." usually means the directory in which your program is located. In this case, the directory contains no sub-directories and six files.

Note: Softlinks are always treated as if they were empty files.

1.11 Graphing Functions: "graphing.ss"

The teachpack provides two operations for graphing functions in the regular (upper right) quadrant of the Cartesian plane (between 0 and 10 in both directions):

```
(graph-fun f color) → true
  f : (-> number?  number?)
  color : symbol?
```

Draws the graph of *f* with the given *color*.

```
(graph-line line color) → true
  line : (-> number? number?)
  color : symbol?
```

Draws *line*, a function representing a straight line, with a given color.

For color symbols, see §1.6 "Simple Drawing: "draw.ss"".

1.12 Simple Graphical User Interfaces: "gui.ss"

The teachpack provides operations for creating and manipulating graphical user interfaces. We recommend using the world teachpack instead.

Window A Window is a data representation of a visible window on your computer screen.

GUI-ITEM A GUI-Item is a data representation of an active component of a window on your computer screen.

```
(create-window g) → Window
  g : (listof (listof GUI-ITEM))
```

Creates a window from the "matrix" of gui items *g*.

```
(window? x) → boolean?
  x : any/c
```

Is the given value a window?

```
(show-window w) → true
  w : Window
```

Shows *w*.

```
(hide-window w) → true
  w : window
```

Hides *w*.

```
(make-button label callback) → GUI-ITEM
  label : string>
  callback : (-> event% boolean)
```

Creates a button with *label* and *callback* function. The latter receives an argument that it may safely ignore.

```
(make-message msg) → GUI-ITEM
  msg : string?
```

Creates a message item from *msg*.

```
(draw-message g m) → true
  g : GUI-ITEM
  m : string?
```

Displays *m* in message item *g* and erases the current message.

```
(make-text txt) → GUI-ITEM
  txt : string?
```

Creates an text editor (with label *txt*) that allows users to enter text.

(text-contents *g*) → string?
  g : GUI-ITEM

Determines the current contents of a text GUI-ITEM.

(make-choice *choices*) → GUI-ITEM
  choices : (listof string?)

Creates a choice menu from *choices* that permits users to choose from some alternatives.

(choice-index *g*) → natural-number/c
  g : GUI-ITEM

Determines the choice that is currently selected in a choice GUI-ITEM; the result is the 0-based index in the choice menu

Example 1:

```
> (define w
    (create-window
      (list (list (make-button "QUIT" (lambda (e) (hide-window w)))))))
; A button appears on the screen.
; Click on the button and it will disappear.
> (show-window w)
; The window disappears.
```

Example 2:

```
; text1 : GUI-ITEM
(define text1
  (make-text "Please enter your name"))

; msg1 : GUI-ITEM
(define msg1
  (make-message (string-append "Hello, World" (make-string 33 #\space))))

; Event -> true
; draws the current contents of text1 into msg1, prepended with "Hello, "
(define (respond e)
  (draw-message msg1 (string-append "Hello, " (text-contents text1))))

; set up window with three "lines":
```

```
; a text field, a message, and two buttons
; fill in text and click OKAY
(define w
 (create-window
  (list
   (list text1)
   (list msg1)
   (list (make-button "OKAY" respond)
         (make-button "QUIT" (lambda (e) (hide-window w)))))))
```

1.13 An Arrow GUI: `"arrow-gui.ss"`

The teachpack provides operations for creating and manipulating an arrow GUI. We recommend using the world teachpack instead.

modelT `(-> button% event% true)`

A modelT is a function that accepts and ignores two arguments.

`(control)` → `symbol?`

Reads out the current state of the message field.

`(view s)` → `true`
  `s` : `(or/c string? symbol?)`

Displays `s` in the message field.

`(connect l r u d)` → `true`
  `l` : modelT
  `r` : modelT
  `u` : modelT
  `d` : modelT

Connects four controllers with the four directions in the arrow window.

Example:

```
; Advanced
(define (make-model dir)
   (lambda (b e)
     (begin
       (view dir)
```

```
        (printf "~a ~n" (control)))))

  (connect
     (make-model "left")
     (make-model "right")
     (make-model "up")
     (make-model "down"))
```

Now click on the four arrows. The message field contains the current direction, the print-out the prior contents of the message field.

1.14 Controlling an Elevator: "elevator.ss"

The teachpack implements an elevator simulator.

It displays an eight-floor elevator and accepts mouse clicks from the user, which are translated into service demands for the elevator.

```
(run NextFloor) → any/c
  NextFloor : number?
```

Creates an elevator simulator that is controlled by *NextFloor*. This function consumes the current floor, the direction in which the elevator is moving, and the current demands. From that, it computes where to send the elevator next.

Example: Define a function that consumes the current state of the elevator (three arguments) and returns a number between 1 and 8. Here is a non-sensical definition:

```
  (define (controller x y z) 7)
```

It moves the elevator once, to the 7th floor.

Second, set the teachpack to <code>elevator.ss</code>, click RUN, and evaluate

```
  (run controller)
```

1.15 Queens: "show-queen.ss"

The teachpack provides the operation show-queen, which implements a GUI for exploring the n-queens problem.

```
(show-queen board) → true
  board : (list-of (list-of boolean?))
```

The function show-queen consumes a list of lists of booleans that describes a board. Each of the inner lists must have the same length as the outer list. The trues correspond to positions where queens are, and the falses correspond to empty squares. The function returns nothing.

In the GUI window that show-queen opens, the red and orange dots show where the queens are. The green dot shows where the mouse cursor is. Each queen that threatens the green spot is shown in red, and the queens that do not threaten the green spot are shown in orange.

1.16 Matrix Operations: "matrix.ss"

The experimental teachpack supports matrices and matrix operations. A matrix is just a rectangle of 'objects'. It is displayed as an image, just like the images from §1.1 "Manipulating Images: "image.ss"". Matrices are images and, indeed, scenes in the sense of the §1.2 "Simulations and Animations: "world.ss"".

No educational materials involving matrices exist.

The operations access a matrix in the usual (school-mathematics) manner: row first, column second.

The operations aren't tuned for efficiency so don't expect to build programs that process lots of data.

Rectangle A Rectangle (of X) is a non-empty list of lists containing X where all elements of the list are lists of equal (non-zero) length.

(matrix? o) → boolean?
  o : any/c

determines whether the given object is a matrix?

(matrix-rows m) → natural-number/c
  m : matrix?

determines how many rows this matrix m has

(matrix-cols m) → natural-number/c
  m : matrix?

determines ow many columns this matrix m has

```
(rectangle->matrix r) → matrix?
  r : Rectangle
```

creates a matrix from the given Rectangle

```
(matrix->rectangle m) → Rectangle
  m : matrix?
```

creates a rectangle from this matrix `m`

```
(make-matrix n m l) → matrix?
  n : natural-number/c
  m : natural-number/c
  l : (Listof X)
```

creates an `n` by `m` matrix from `l`

NOTE: `make-matrix` would consume an optional number of entries, if it were like `make-vector`

```
(build-matrix n m f) → matrix?
  n : natural-number/c
  m : natural-number/c
  f : (-> (and/c natural-number/c (</c m))
          (and/c natural-number/c (</c n))
          any/c)
```

creates an `n` by `m` matrix by applying `f` to `(0 ,0)`, `(0 ,1)`, ..., `((sub1 m) ,(sub1 n))`

```
(matrix-ref m i j) → any/c
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
```

retrieve the item at (`i`,`j`) in matrix `m`

```
(matrix-set m i j x) → matrix?
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
  x : any/c
```

creates a new matrix with `x` at (`i`,`j`) and all other places the same as in `m`

```
(matrix-where? m pred?) → (listof posn?)
  m : matrix?
  pred? : (-> any/c boolean?)
```

`(matrix-where? M P)` produces a list of `(make-posn i j)` such that `(P (matrix-ref M i j))` holds

```
(matrix-render m) → Rectangle
  m : matrix?
```

renders this matrix `m` as a rectangle of strings

```
(matrix-minor m i j) → matrix?
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
```

creates a matrix minor from `m` at (`i`,`j`)

```
(matrix-set! m i j x) → matrix?
  m : matrix?
  i : (and/c natural-number/c (</c (matrix-rows m)))
  j : (and/c natural-number/c (</c (matrix-rows m)))
  x : any/c
```

like `matrix-set` but uses a destructive update

2 HtDC Teachpacks

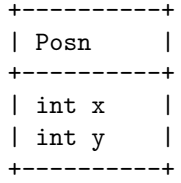
2.1 Geometry: geometry.*

Add

```
import geometry.*
```

at the top of your Definitions Window to import this library.

This package provides a class for representing positions in a Cartesian world:



Posn is a class with two fields, one per coordinate. The constructor consumes two integers.

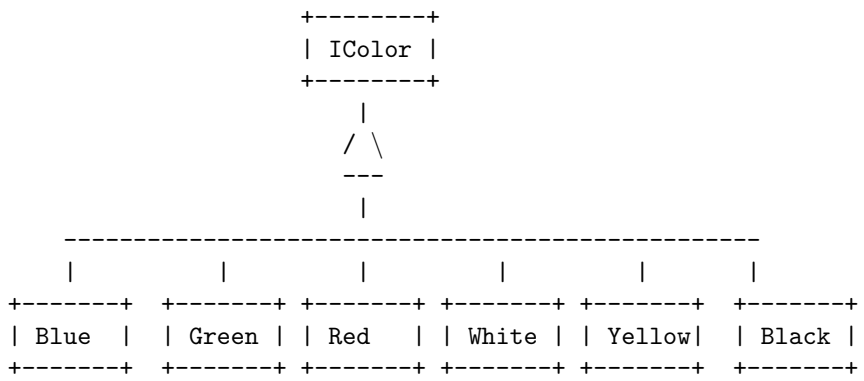
2.2 Colors: colors.*

Add

```
import colors.*
```

at the top of your Definitions Window to import this library.

This package provides classes for representing colors:



IColor is an interface. Its variants are created with no arguments.

2.3 Draw: draw.*

Add

```
import draw.*
```

at the top of your Definitions Window to import this library.

This package provides classes and methods for a visual world. Here is its class diagram of public fields and methods:

```
import colors.*;
import geometry.*;

 +-----------------------------------+
 | abstract World                    |
 +-----------------------------------+
 | Canvas theCanvas                  |----+
 +-----------------------------------+    |
 | boolean bigBang(int,int,double)   |    |
 | boolean endOfTime(String)         |    |
 | World endOfWorld(String)          |    |
 |                                   |    |
 |                                   |    |
 | abstract World onTick()           |    |
 | abstract World onKeyEvent(String) |    |
 | abstract boolean draw()           |    |
 +-----------------------------------+    |
                                          |
                                          v
    +--------------------------------------+
    | Canvas                               |
    +--------------------------------------+
    +--------------------------------------+
    | boolean show()                       |
    | boolean close()                      |
    | boolean drawCircle(Posn,int,IColor)  |
    | boolean drawDisk(Posn,int,IColor)    |
    | boolean drawRect(Posn,int,int,IColor)|
    | boolean drawLine(Posn,Posn,IColor)   |
    | boolean drawString(Posn,String)      |
    +--------------------------------------+
```

Methods in these classes may fail due to the unavailability of the physical devices, inappropriate uses, etc. In those cases, they fail with an exception.

2.3.1 World

The abstract `World` class exports the following methods.

`bigBang : (int width,int height,double speed)`

Initializes the world, associates it with a `width` x `height` `Canvas`, displays this canvas, enables keyevents, and finally starts the clock at a rate of one tick per `speed` seconds. If it succeeds with all of its actions, the method produces `true`.

Note: `width`, `height` and `speed` must be a positive.

The canvas in `World` is called

`theCanvas`.

References to a "canvas" in conjunction with the `World` class denote this default canvas.

`endOfTime : ()`

Displays the given message, stops the clock and, if it succeeds, produces `true`. After the end of time, events no longer trigger calls to `onTick` or `onKeyEvent`. The canvas remains visible.

`endOfWorld : (String msg)`

Displays the given message, stops the clock and, if it succeeds, produces the last `World`. After the end of the world, events no longer trigger calls to `onTick` or `onKeyEvent` (see below). The canvas remains visible.

A derived concrete class must supply definitions for the following methods:

`onTick : ()`

Invoked for every tick of the clock. Its purpose is to create a `World` whose differences with `this` one represent what happened during the amount of time it takes the clock to tick.

`onKeyEvent : (String key)`

Invoked for every keyboard event associated with the canvas. Its purposes is to create a `World` whose differences with `this` one represent what happens due to the user's use of the keyboard. The latter is represented with the string-valued argument `key`.

`draw : ()`

Invoked *after* one of the two event handlers has been called. Its purpose is to present `this World` graphically on its canvas. If it succeeds, its result is `true`.

A program may, in principle, start several instances of (subclasses of) `World`. If it does, the event handlers are called in a unpredictable order.

2.3.2 Canvas

To create an instance of the `Canvas` class, a program must supply two `int` values: one for the width of the canvas and one for its height. The canvas is a rectangle, whose borders are parallel to the computer screen's borders. A program can use the following methods on instances of `Canvas`]

`show : ()`

Initializes the canvas to a white area, enables the drawing methods, and finally displays the canvas. If it succeeds, it produces `true`. Invoking the method a second time without calling `close` before has no effect.

`close : ()`

Hides the canvas and erases the current content. If it succeeds, it produces `true`.

Closing the Canvas using the display controls does not fully hide the canvas; it is still necessary to invoke `close` before `show` is re-enabled.

`drawCircle : (Posn p,int r,IColor c)`

Draws a circle on `this`Canvas] at `p` with radius `r` and color `c`. If it succeeds, it produces `true`.

`drawDisk : (Posn p,int r,IColor c)`

Draws a disk on `this`Canvas] at `p` with radius `r` and color `c`. If it succeeds, it produces `true`.

`drawRect : (Posn p,int w,int h,IColor c)`

Draws a solid rectangle on `thisCanvas]` at `p` with width `w`, height `h`, and color `c`. The rectangle's lines are parallel to the canvas's borders. If it succeeds, it produces `true`.

`drawLine : (Posn p0,Posn p1,IColor c)`

Draws a line on `thisCanvas]` from `p0` to `p1` using color `c`. If it succeeds, it produces `true`.

`drawString : (Posn p,String s)`

Draws the string `s` at `p` on `thisCanvas]`. If it succeeds, it produces `true`.

2.4 Draw: idraw.*

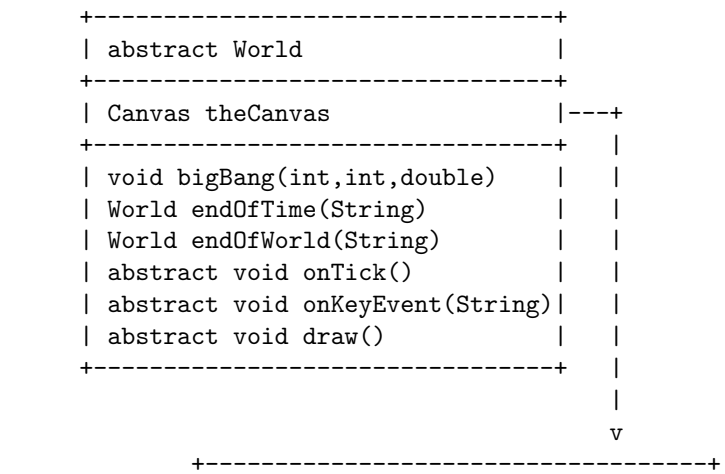
Add

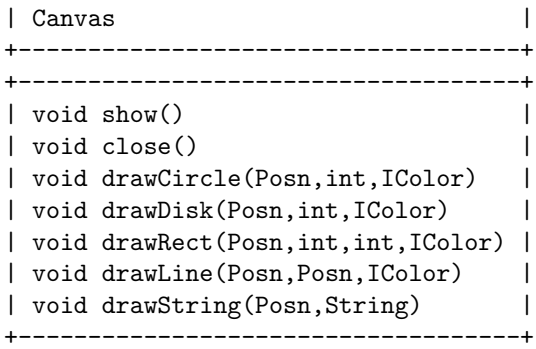
```
import idraw.*
```

at the top of your Definitions Window to import this library.

This package provides stateful classes and imperative methods for a visual world. Here is its class diagram of public fields and methods:

```
import colors.*;
import geometry.*;
```





The abstract `World` class in `idraw` provides the same methods as the `World` class in §2.3.1 "World" (`draw` package). Their return values are usually `void`, however, except for `endOfTime` and `endOfWorld`, which continue to return the last world.

In an analogous manner, the methods in the `Canvas` class export the same methods as the `Canvas` class in §2.3.2 "Canvas" (`draw` package). Again their return values are `void`.