Chapter 18

Of Mice and Keys

18.1 Mouse handlers

Recall that the contract for a mouse-handling function must be

```
; model(old) number(x) number(y) event -> model(new)
```

but we didn't explain earlier what an "event" was. Guess what: it's a string. Specifically, it will always be one of the strings in Figure 18.1. And since you already know how to

Figure 18.1: Types of mouse events in DrRacket

`"button-down"`	The user pressed the mouse button.
`"button-up"`	The user released the mouse button.
`"move"`	The user moved the mouse, with the mouse button not pressed.
`"drag"`	The user dragged the mouse while holding the mouse button down.
`"enter"`	The user moved the mouse into the animation window.
`"leave"`	The user moved the mouse out of the animation window.

compare strings, you can write mouse handlers that respond differently to different user actions.

The obvious data analysis for a mouse-handling function would say that the fourth parameter is one of six choices. However, in practice we are usually only interested in one or two of the six, ignoring all the rest; this allows us to considerably simplify our functions. Here's an example:

Worked Exercise 18.1.1 ***Write an animation*** *that starts with a blank screen, and adds a small dot at the mouse location every time the mouse button is pressed.*

Solution: Our first question in writing an animation is always what handlers we'll need. There's no mention of time or keyboard, but we'll obviously need a mouse handler, and as usual we'll need a draw handler and a `check-with` handler.

The next question is what type the model should be. We're *adding* dots, and may eventually have dozens or hundreds of them; this problem may remind you of problems 8.5.3

and 17.2.3. As in those problems, the most reasonable model is an *image* showing all the dots added so far. (In Chapter 22, we'll see another way to handle this.)

Since the model is an image itself, and that image is all we want to show, we can use `show-it` as our draw handler rather than writing our own.

The mouse handler has the usual contract:

```
; add-dot-on-mouse-down :
; image(old) number(x) number(y) string(event-type) -> image(new)
```

An event can be any of six possible strings, which would suggest an six-way definition by choices. However, we're only interested in the `"button-down"` event, so the fourth parameter really falls into one of two categories: either `"button-down"` or "any other string"; the input template looks like

```
#|
(check-expect (function-on-mouse-press "button-down") ...)
(check-expect (function-on-mouse-press "button-up") ...)

(define (function-on-mouse-press event-type)
  (cond [(string=?  event-type "button-down") ...]
        [else                                 ...]
  ))
|#
```

We won't bother with an output template because we seldom need to produce a mouse event.

A simple example starts with an `empty-scene` as the old image. Since there are two categories for the fourth parameter (and no interesting "categories" for the other three), we'll need at least two examples:

```
(define WIDTH 300)
(define HEIGHT 200)
(define BACKGROUND (empty-scene WIDTH HEIGHT))
(define DOT (circle 3 "solid" "green"))
(check-expect (add-dot-on-mouse-down BACKGROUND 35 10 "button-down")
              (place-image DOT 35 10 BACKGROUND))
(check-expect (add-dot-on-mouse-down BACKGROUND 35 10 "move")
              BACKGROUND)
```

To make sure the program is actually *adding* a dot to the given image, we should also try it with a different first argument.

```
(define OTHER-BACKGROUND
  (ellipse 50 30 "solid" "red")
(check-expect (add-dot-on-mouse-down OTHER-BACKGROUND 35 10 "button-down")
              (place-image DOT 35 10 OTHER-BACKGROUND))
(check-expect (add-dot-on-mouse-down OTHER-BACKGROUND 35 10 "button-up")
              OTHER-BACKGROUND)
```

The skeleton and inventory are straightforward, if lengthy:

```
(define (add-dot-on-mouse-down old x y event-type)
  ; old           an image
  ; x             a number (the x coordinate)
  ; y             a number (the y coordinate)
  ; event-type    a string (either "button-down" or not)
  ; DOT           a fixed image we'll need
  ...)
```

Next, the template gives us

```
(define (add-dot-on-mouse-down old x y event-type)
  ; old           an image
  ; x             a number (the x coordinate)
  ; y             a number (the y coordinate)
  ; event-type    a string (either "button-down" or not)
  ; DOT           a fixed image we'll need
  (cond [ (string=?  event-type "button-down") ...]
        [ else                                 ...]
        )
  )
```

We still need to fill in the answers. The answer in the `else` case is simple: if the event type is *not* `"button-down"`, we shouldn't do anything, and should simply return the picture we were given. The answer in the `"button-down"` case is more complicated, but we know the `place-image` function is useful for adding things to an existing picture. It takes in the image to add (in our case, `DOT`), two numeric coordinates (obviously `x` and `y`), and the picture to add to (in our case, `old`). The final definition is

```
(define (add-dot-on-mouse-down old x y event-type)
  ; old           an image
  ; x             a number (the x coordinate)
  ; y             a number (the y coordinate)
  ; event-type    a string (either "button-down" or not)
  ; DOT           a fixed image we'll need
  (cond [ (string=?  event-type "button-down")
          (place-image DOT x y old) ]
        [ else  old ]
        )
  )
```

Once this works, we can try it in an animation as follows:

```
(big-bang BACKGROUND
          (check-with image?)
          (on-draw show-it)
          (on-mouse add-dot-on-mouse-down))
```

Does it work the way you expect? Does it work as it should? ∎

Exercise 18.1.2 ***Modify*** *this animation so it adds a dot whenever the mouse button is* released*, rather than whenever it is* pressed*. As a user, do you like this version better or worse?*

Exercise 18.1.3 ***Modify*** *this animation so it adds a dot whenever the mouse is* dragged *(*i.e. *moved while holding the mouse button down). The result should be a sort of "sketch-pad" program in which you can draw lines and curves with the mouse.*

Exercise 18.1.4 ***Modify*** *this animation so it adds a green dot whenever the mouse button is pressed, and a red dot whenever the mouse button is released.*

Hint: You're now interested in *two* of the event types, so there are now *three* interesting categories of input.

Exercise 18.1.5 ***Modify*** *the animation of Exercise 17.2.6 so that it stops* only *if it gets a* `"button-up"` *event inside the button.*

Exercise 18.1.6 ***Modify*** *exercise 18.1.1 so that rather than adding a pure-red dot, it adds a drop of red dye at the mouse location. The dye adds a certain amount to the red component of the picture, varying with distance from where it was dropped: for example, if it added 100 to the red component right at the mouse location, it might add 50 to the neighboring pixels, 33 to the pixels 2 units away, 25 to the pixels 3 units away, and so on. The green and blue components of the picture should be unchanged.*

Hint: Use `map3-image`.

18.2 Key handlers

Recall from chapter 6 that the contract for a key-handling function must be

```
; key-handler :  model key-event -> model
```

Chapter 6 was fuzzy on what this "key" parameter is. In fact, it's a string: if a user types "w", your key handler will be called with the string `"w"`, and so on. There are also some special keyboard keys, described in Figure 18.2. For convenience, DrRacket provides a built-in function named `key=?` which is just like `string=?` except that it works *only* on key-events (this can be useful because if you mistakenly call it on something that isn't a key-event at all, it'll produce an error message immediately rather than letting you go on with a mistake in your program).

Worked Exercise 18.2.1 ***Develop an animation*** *of a picture (say, a calendar) that moves left or right by 1 pixel when the user presses the left or right arrow key (respectively). It should ignore all other keys.*

Solution: What handlers do we need? We obviously need to respond to key presses, so we need a key handler. We might or might not need a draw handler, depending on the model type.

So what type should the model be? We've handled similar problems in the past in one of two ways: an image, moving left with `crop-left` and moving right with `beside` and `rectangle`, or a number, moving left with `sub1` and moving right with `add1`. The latter is more efficient, and has the advantage that we can move off the left-hand edge of the screen and come back. So let's use a number to represent the x-coordinate of the image.

This tells us that the key handler's contract is

```
; handle-key :  number(x) key-event -> number(new x)
```

Figure 18.2: Special keyboard keys

Key on keyboard	**key-event**
left arrow	`"left"`
right arrow	`"right"`
down arrow	`"down"`
up arrow	`"up"`
clear (on number pad)	`"clear"`
shift	`"shift"`
control	`"control"`
caps lock	`"capital"`
num lock	`"num lock"`
page up	`"prior"`
page down	`"next"`
end	`"end"`
home	`"home"`
help	`"help"`
esc	`"escape"`
F1, F2, *etc.*	`"f1"`, `"f2"`, *etc.*
+ (on number pad)	`"add"`
- (on number pad)	`"subtract"`
∗ (on number pad)	`"multiply"`
/ (on number pad)	`"divide"`
enter (on number pad)	`"numpad-enter"`

And we'll need a draw handler with contract

```
; calendar-at-x :  number(x) -> image
```

The `calendar-at-x` function is exactly the same as we've used in several previous exercises, so I'll leave it to the reader. Now, about that `handle-key` function...

Data analysis: the first parameter is a number, about which there's not much to say. The second parameter could be a *lot* of different things, but the only categories we're interested in are `"left"`, `"right"`, and anything else (in which case we ignore it). We could write explicit templates for this data type, but we don't expect to be writing lots of functions on it so we'll skip that step.

We'll need at least three examples: one with `"left"`, one with `"right"`, and one key-event that's not either of those: at least three examples in all. Note that if the key isn't `"left"` or `"right"`, we ignore it by returning a new model that's exactly the same as the old one.

```
(check-expect (handle-key 10 "D") 10)
(check-expect (handle-key 10 "left") 9)
(check-expect (handle-key 10 "right") 11)
```

The skeleton and inventory are straightforward:

```
(define (handle-key x key)
  ; x            number
  ; key          key-event (i.e.  string)
  ...)
```

Since there are three main categories (`"left"`, `"right"`, and anything else) of input, we'll need a `cond` with three clauses, with questions to check which one `key` is:

```
(define (handle-key x key)
  ; x            number
  ; key          key-event (i.e.  string)
  (cond [ (key=?  key "right") ...]
        [ (key=?  key "left")  ...]
        [ else                 ...]
        )
  )
```

Now that we've filled in all the questions, we need to fill in the answers. If `key` is anything other than `"left"` or `"right"`, this is easy: return the same x-coordinate we were given, unchanged. In the `"left"` case, we want to subtract 1 from it, and in the `"right"` case, we want to add 1 to it.

```
(define (handle-key x key)
  ; x            number
  ; key          key-event (i.e.  string)
  ; "left"       a fixed string we'll need
  ; "right"      another fixed string we'll need
  (cond [ (key=?  key "right") (+ x 1) ]
        [ (key=?  key "left")  (- x 1) ]
        [ else                 x ]
        )
  )
```

Once we've tested this and confirmed that the function works on its own, we can run the animation as follows:

```
(define WIDTH 400)
(define HEIGHT 100)
(define (calendar-at-x x)
  ...
  )
(big-bang (/ WIDTH 2)
          (check-with number?)
          (on-draw calendar-at-x)
          (on-key handle-key))
```

Exercise 18.2.2 ***Modify*** *the above animation so it also responds to some ordinary characters: the picture moves 5 pixels to the right in response to the ">" key, and 5 pixels to the left in response to the "<" key.*

Exercise 18.2.3 ***Modify*** *the above animation so it stops when the user types the letter "q".*

Exercise 18.2.4 ***Develop an animation*** *of a disk whose radius increases by 1 when the user presses the up-arrow key, and decreases by 1 when the user presses the down-arrow key.*

Exercise 18.2.5 ***Develop an animation*** *that allows the user to "type" into the animation window: every time the user types an ordinary character, that character is added to the right-hand end of the animation window. The program will ignore arrow keys, function keys,* etc.

Hint: See Exercise 10.2.1 for some ideas, and use `string-length` to check whether the key is an ordinary character (*i.e.* the *key-event* is a one-character string). Be sure to test your program with arrow keys as well as ordinary characters.

18.3 Key release

Key handlers are triggered whenever the user presses a key. Sometimes you want something to happen when the user *releases* a key instead. To handle this situation, you can install an `on-release` handler, which is just like an `on-key` handler except that it's called when the key is released rather than when it's pressed. It has contract

```
; handle-release :  model key-event -> model
```

The *key-event* tells you what key was just released.

Exercise 18.3.1 ***Modify*** *some of the key-based animations from this chapter so they trigger on key release rather than on key press.*

Exercise 18.3.2 ***Develop an animation*** *which shows the currently-pressed key for just as long as you hold it down; then it disappears when you release it.*

18.4 Review of important words and concepts

A mouse handler takes in, as its fourth argument, a *mouse-event*, a string which is one of six standard choices, indicating whether the mouse was pressed, released, moved, dragged, *etc.*. A mouse handler, therefore, has as its body a `cond` with up to six cases, handling each different kind of mouse action. More commonly, the handler will only test for one or two kinds of mouse action, then use an `else` clause to handle "all the rest".

A key handler takes in, as its second argument, a *key-event*, a string which is one of several dozen standard choices: single-character strings for ordinary keys, and special strings like `"left"`, `"help"`, *etc.*for special keys. As a convenience for writing key handlers, there's a built-in function `key=?` which works on *only* these strings, and produces an error message on anything else. If your key handler needs to respond only to a short list of specific keys, you can write it using a `cond` with a bunch of `key=?` questions. If you need to handle "all special keys" in one way and "all ordinary keys" in another way, you may need to do something cleverer, like use the fact that all ordinary keys are one-character strings and all the special ones have longer names.

18.5 Reference: Built-in functions for mouse and key handling

One new function was introduced in this chapter: `key=?`

We also introduced `big-bang`'s `on-release` clause, which works just like `on-key` except that it's triggered by releasing a key rather than pressing one.