Cards: Virtual Playing Cards Library

Version 9.4.0.3

November 1, 2026

`(require games/cards)` package: `games`

The `games/cards` module provides a toolbox for creating card games.

1 Creating Tables and Cards

```
(make-table [title w h #:mixin mixin]) → table<%>
  title : string? = "Cards"
  w : real? = 7
  h : real? = 3
  mixin : (make-mixin-contract table<%>) = values
```

Returns a table. The table is named by *title*, and it is *w* cards wide and *h* cards high (assuming a standard card of 71 by 96 pixels). If *mixin* is provided, it is applied to the class implementing `table<%>` before it is instantiated.

The table is not initially shown; `(send table show #t)` shows it.

```
(make-deck) → (listof card<%>)
```

Returns a list of 52 cards, one for each suit-value combination. The cards are all face-down, sorted lowest-suit then lowest-value. A card can only be on one table at a time.

```
(make-card front-bm back-bm suit-id value) → (is-a?/c card<%>)
  front-bm : (is-a?/c bitmap?)
  back-bm : (or/c (is-a?/c bitmap%) #f)
  suit-id : any/c
  value : any/c
```

Returns a single card given a bitmap for the front, an optional bitmap for the back, and arbitrary values for the card's suit and value (which are returned by the card's `get-value` and `get-suit-id` methods). If *back-bm* is `#f`, then the back defaults to a standard 71 by 96 bitmap. The front and back must be the same size.

```
(shuffle-list lst n) → list?
  lst : list?
  n : exact-nonnegative-integer?
```

Shuffles the given *lst* *n* times, returning the new list. Shuffling simulates an actual shuffle: the list is split into halves which are merged back together by repeatedly pulling the top card off one of the halves, randomly selecting one half or the other. According to [some mathematical theorem], 7 is a large enough *n* to get a perfect shuffle.

2 Regions and Buttons

```
(struct region (x
                y
                w
                h
                label
                [callback #:mutable]
                [interactive-callback #:auto #:mutable]
                [paint-callback #:auto #:mutable]
                [button? #:auto]
                [hilite? #:auto])
  #:extra-constructor-name make-region)
 x : real?
 y : real?
 w : (and/c real? (not/c negative?))
 h : (and/c real? (not/c negative?))
 label : (or/c string? (is-a?/c bitmap%) #f)
 callback : (or/c #f (if button?
                         (-> any)
                         (-> (listof (is-a?/c card<%>)) any)))
 interactive-callback : (or/c #f (-> any/c (listof (is-a?/c card<%>)) any))
 paint-callback : (or/c #f (-> (is-a?/c dc<%>) real? real? real? real? any))
 button? : any/c
 hilite? : any/c
```

The `x`, `y`, `w`, and `h` fields determine the region's location on the table.

When `label` is a string, it is drawn in the region in 12-pixel text, centered horizontally and 5 pixels down from the region's top outline. If label is `#f`, no label or box is drawn for the region.

The `callback` procedure takes a list of cards that were dragged to the region; if callback is `#f`, the region is not active (i.e., dragging cards to the region doesn't highlight the region box). The region remains hilited until the callback returns.

The `interactive-callback` procedure is invoked when a region is (un)hilited as the user drags a set of cards to the region. The callback is provided two arguments: a boolean indicating whether the region is hilited, and the list of cards being dragged. Like region-callback, the default is `#f`, which indicates that the region has no interactive callback (but does not affect whether the region is hilited as cards are dragged). The final unhilite (when cards are potentially delivered) does not trigger this callback.

The `paint-callback` function is called with a drawing context, x and y offsets, and the width and height (which are always `w` and `h`). The x and y offsets can be different than the supplied `x` and `y` when part of the table is drawn offscreen. Regions are painted in

the order that they are added to a table, and all regions are painted before any card. The paint-callback procedure should not assume a particular state for the drawing context (i.e.,current brush or pen), and it should restore any modified drawing context state before returning.

The only available mutators on the structure are set-region-callback!, set-region-interactive-callback!, and set-region-paint-callback!. The structure created by make-region actually has extra hidden fields.

```
(make-button-region x y w h label callback) → region?
  x : real?
  y : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
  label : (or/c string? (is-a?/c bitmap%) #f)
  callback : (or/c #f (-> any))
```

Returns a region like one made by make-region, but the is drawn slightly differently and it reacts differently to cards and the mouse. The label is drawn in the middle of the box instead of at the top, and the callback is called with no arguments when the user clicks the region (instead of dragging cards to the region).

```
(make-background-region x
                        y
                        w
                        h
                        paint-callback) → region?
  x : real?
  y : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
  paint-callback : (-> (is-a?/c dc<%>) real? real? real? real? any)
```

Returns a region that does not respond to mouse clicks, but which has a general paint callback.

3 Table Methods

```
table<%> : interface?
  implements: frame%
```

Create an instance with `make-table`.

```
(send a-table add-card card x y) → void?
  card : (is-a?/c card<%>)
  x : real?
  y : real?
```

Adds *card* to the table with its top-left corner at (*x*, *y*) in table pixels.

```
(send a-table add-cards  cards
                         x
                         y
                        [offset-proc]) → void?
  cards : (listof (is-a?/c card<%>))
  x : real?
  y : real?
  offset-proc : (exact-nonnegative-integer?
                 . -> . (values real? real?))
              = (lambda (i) (values 0 0))
```

Adds a list of cards at (*x*, *y*). The optional *offset-proc* procedure is called with an index *i* (counting from 0) and should return two values: *dx* and *dy*; the *i*th card is the placed at `(+ x +dx)` and `(+ y dy)`. The cards are added in order on top of cards already one the table such that the first card in *cards* is topmost.

```
(send a-table add-cards-to-region cards
                                  region?) → void?
  cards : (listof (is-a?/c card<%>))
  region? : r
```

Adds *cards* to fill the region `r`, fanning them out bottom-right to top-left, assuming that all cards in *cards* have the same width and height. The region `r` does not have to be added to the table.

```
(send a-table remove-card card) → void?
  card : (is-a?/c card<%>)
```

Removes *card* from the table.

```
(send a-table remove-cards cards) → void?
  cards : (listof (is-a?/c card<%>))
```

Removes *cards* from the table.

```
(send a-table move-card card x y) → void?
  card : (is-a?/c card<%>)
  x : real?
  y : real?
```

Moves *card*, which must be on the same already. The movement of the cards is animated. If the cards are in `snap-back-after-move` mode and a drag is active, snapping back will use the new location.

```
(send a-table move-cards  cards
                          x
                          y
                         [offset-proc]) → void?
  cards : (listof (is-a?/c card<%>))
  x : real?
  y : real?
  offset-proc : (exact-nonnegative-integer?
                 . -> . (values real? real?))
              = (lambda (i) (values 0 0))
```

Like `add-cards`, but moves cards that are already on the table like `move-card`. All of the cards are moved at once.

```
(send a-table move-cards-to-region cards
                                   region?) → void?
  cards : (listof (is-a?/c card<%>))
  region? : r
```

Like `add-cards-to-region`, but moves cards that are already on the table like `move-card`. All of the cards are moved at once.

```
(send a-table flip-card card) → void?
  card : (is-a?/c card<%>)
(send a-table flip-cards cards) → void?
  cards : (listof (is-a?/c card<%>))
```

Flips *card* or all *cards* over (at once) with animation.

```
(send a-table card-face-up card) → void?
  card : (is-a?/c card<%>)
(send a-table cards-face-up cards) → void?
  cards : (listof (is-a?/c card<%>))
(send a-table card-face-down card) → void?
  card : (is-a?/c card<%>)
(send a-table cards-face-down cards) → void?
  cards : (listof (is-a?/c card<%>))
```

Like `flip-cards`, but only for *card* or elements of *cards* that are currently face down/up.

```
(send a-table rotate-card card mode) → void?
  card : (is-a?/c card<%>)
  mode : (or/c 'cw 'ccw 0 90 -90 180 -180 270 -270 360)
(send a-table rotate-cards cards mode) → void?
  cards : (listof (is-a?/c card<%>))
  mode : (or/c 'cw 'ccw 0 90 -90 180 -180 270 -270 360)
```

Rotates *card* or all *cards* (at once, currently without animation, but animation may be added in the future). The center of each card is kept in place, except that the card is moved as necessary to keep it on the table. See `rotate` in `card<%>` for information on *mode*.

```
(send a-table card-to-front card) → void?
  card : (is-a?/c card<%>)
(send a-table card-to-back card) → void?
  card : (is-a?/c card<%>)
```

Moves *card* before/behind of all other cards.

```
(send a-table stack-cards cards) → void?
  cards : (listof (is-a?/c card<%>))
```

The first card in *cards* is not moved; the second card is moved to follow immediately behind the first one, then `stack-cards` is called on `(cdr cards)`. If *cards* is empty or contains only one card, no action is taken.

```
(send a-table card-location card) → real? real?
  card : (is-a?/c card<%>)
```

Returns the location of the given card; an exception is raised if the card is not on the table.

```
(send a-table all-cards) → (listof (is-a?/c card<%>))
```

Returns a list of all cards on the table in stacking order from front to back.

```
(send a-table table-width) → exact-nonnegative-integer?
(send a-table table-height) → exact-nonnegative-integer?
```

Returns the width/height of the table in pixels.

```
(send a-table begin-card-sequence) → void?
(send a-table end-card-sequence) → void?
```

Starts/ends a sequence of card or region changes that won't be animated or updated until the end of the sequence. Sequences can be nested via matching `begin-`/`end-` pairs.

```
(send a-table add-region r) → void
  r : region?
```

Adds the region *r* to the table; regions are drawn in the order that they are added to the table, and when a region added later is hilighted, it can obscure regions added earlier.

```
(send a-table remove-region r) → void
  r : region?
```

Removes the region *r* from the table.

```
(send a-table hilite-region r) → void?
  r : region?
(send a-table unhilite-region r) → void?
  r : region?
```

Manual (un)hilite, usually for animation.

```
(send a-table set-button-action which
                                action) → void?
  which : (one-of/c 'left 'middle 'right)
  action : symbol?
```

Sets the way that a mouse click is handled for a particular button indicated by *which*. The *action* argument must be one of the following:

- `'drag/one` — drag only the clicked-on card.
- `'drag-raise/one` — like drag/one, but raise the card to the top on a click.

- `'drag/above` — drag the card along with any card on top of the card (i.e., more towards the front and overlapping with the card). The on-top-of relation is closed transitively.
- `'drag-raise/above` — like `'drag/above`, but raises.
- `'drag-below` — drag the card along with any card underneath the card (i.e., more towards the back and overlapping with the card). The underneath relation is closed transitively.
- `'drag-raise/below` — like `'drag/below`, but raises.

The initial settings are: `'drag-raise/above` for `'left`, `'drag/one` for `'middle`, and `'drag/below` for `'right`.

```
(send a-table set-double-click-action proc) → void?
  proc : ((is-a?/c card<%>) . -> . any)
```

Sets the procedure to be called when a card is double-clicked. The procedure is called with the double-clicked card. The default procedure flips the cards along with its on-top-of cards, raises the cards, and reverses the front-to-back order of the cards

```
(send a-table set-single-click-action proc) → void?
  proc : ((is-a?/c card<%>) . -> . any)
```

Sets the procedure to be called when a card is single-clicked, after the button action is initiated. (If the card is double-clicked, this action is invoked for the first click, then the double-click action is invoked.) The default action does nothing.

```
(send a-table pause secs) → void?
  secs : real?
```

Pauses, allowing the table display to be updated (unless a sequence is active), but does not let the user click on the cards.

```
(send a-table animated) → boolean?
(send a-table animated on?) → void?
  on? : any/c
```

Gets/sets animation enabled/diabled.

```
(send a-table create-status-pane) → (is-a?/c pane%)
```

Creates a pane with a status message (initially empty) and returns the pane so that you can add additional controls.

```
(send a-table set-status str) → void?
  str : string?
```

Sets the text message in the status pane.

```
(send a-table add-help-button pane
                              coll-path
                              str
                              tt?)       → void?
  pane : (is-a?/c area-container<%>)
  coll-path : (listof string?)
  str : string?
  tt? : any/c
```

Adds a Help button to the given pane, where clicking the button opens a new window to display `"doc.txt"` from the given collection. The *str* argument is used for the help window title. If *tt?* is true, then `"doc.txt"` is displayed verbatim, otherwise it is formatted as for `show-help` from `games/show-help`.

```
(send a-table add-scribble-button pane
                                  mod-path
                                  tag)      → void?
  pane : (is-a?/c area-container<%>)
  mod-path : module-path?
  tag : string?
```

Adds a Help button to the given pane, where clicking the button opens Scribble-based documentation, as with `show-scribbling` from `games/show-scribbling`.

4 Card Methods

`card<%> : interface?`

Create instances with `make-deck` or `make-card`.

`(send a-card card-width) → exact-nonnegative-integer?`

Returns the width of the card in pixels. If the card is rotated 90 or 270 degrees, the result is the card's original height.

`(send a-card card-height) → exact-nonnegative-integer?`

Returns the height of the card in pixels. If the card is rotated 90 or 270 degrees, the result is the card's original width.

`(send a-card flip) → void?`

Flips the card without animation. This method is useful for flipping a card before it is added to a table.

`(send a-card face-up) → void?`

Makes the card face up without animation.

`(send a-card face-down) → void?`

Makes the card face down without animation.

`(send a-card face-down?) → boolean?`

Returns `#t` if the card is currently face down.

```
(send a-card rotate mode) → void?
  mode : (or/c 'cw 'ccw 0 90 -90 180 -180 270 -270 360)
```

Rotates the card. Unlike using the `rotate-card` in `table<%>` method, the card's top-left position is kept in place.

If *mode* is `'cw`, the card is rotated clockwise; if *mode* is `'ccw`, the card is rotated counter-clockwise; if *mode* is one of the allowed numbers, the card is rotated the corresponding amount in degrees counter-clockwise.

`(send a-card orientation) → (or/c 0 90 180 270)`

Returns the orientation of the card, where 0 corresponds to its initial state, 90 is rotated 90 degrees counter-clockwise, and so on.

```
(send a-card get-suit-id) → any/c
```

Normally returns 1, 2, 3, or 4 (see get-suit for corresponding suit names), but the result can be anything for a card created by make-card.

```
(send a-card get-suit) → symbol?
```

Returns 'clubs, 'diamonds, 'hearts, 'spades, or 'unknown, depending on whether get-suit-id returns 1, 2, 3, 4, or something else.

```
(send a-card get-value) → any/c
```

Normally returns 1 (Ace), 2, ... 10, 11 (Jack), 12 (Queen), or 13 (King), but the result can be anything for a card created by make-card.

```
(send a-card user-can-flip) → boolean?
(send a-card user-can-flip can?) → void?
  can? : any/c
```

Gets/sets whether the user can flip the card interactively, usually by double-clicking it. Initially #t.

```
(send a-card user-can-move) → boolean?
(send a-card user-can-move can?) → void?
  can? : any/c
```

Gets/sets whether the user can move the card interactively, usually by dragging it. Disabling moves has the side-effect of disabling raises and double-clicks. Initially #t.

```
(send a-card snap-back-after-move) → boolean?
(send a-card snap-back-after-move on?) → void?
  on? : any/c
```

Assuming user can move the card interactively, gets/sets whether the card stays where the user dragged it or snaps back to its original place. Initially #f.

A region callback can disable snap-back for a dragged card only if snap-back-after-regions mode is enabled for the card. Otherwise, a region's *interactive* callback can disable snap-back for a card (e.g., so that the card can be delivered to the region).

```
(send a-card snap-back-after-regions) → boolean?
(send a-card snap-back-after-regions on?) → void?
  on? : any/c
```

Determines whether `snap-back-after-move` and `home-region` constraints apply before or after region callbacks are invoked for dragged cards. Initially `#f` (i.e., constraints apply before callbacks).

Added in version 1.1 of package `games`.

```
(send a-card stay-in-region) → (or/c region? #f)
(send a-card stay-in-region r) → void?
  r : (or/c region? #f)
```

Gets/sets a constraining region *r*. If *r* is not `#f`, the user cannot move the card out of *r*. Initially `#f`.

```
(send a-card home-region) → (or/c region? #f)
(send a-card home-region r) → void?
  r : (or/c region? #f)
```

Gets/sets a home region *r*. If *r* is not `#f`, then the user can move the card freely, but the card snaps back if moved out of the region. (If the card is moved partly out of the region, the card is moved enough to get completely back in.) Initially `#f`.

A region callback can change the snap-back home for a dragged card only if `snap-back-after-regions` mode is enabled for the card. Otherwise, a region's *interactive* callback can adjust snap-back for a card.

```
(send a-card dim) → boolean?
(send a-card dim can?) → void?
  can? : any/c
```

Gets/sets a hilite on the card, which is rendered by drawing it dimmer than normal.

```
(send a-card copy) → (is-a?/c card<%>)
```

Makes a new card with the same suit and value.

Bibliography

[some mathematical theorem] Bayer, Dave; Diaconis, Persi, "Trailing the Dovetail Shuffle to its Lair." http://projecteuclid.org/euclid.aoap/1177005705