GUI: Racket Graphics Toolkit

Version 5.0

Matthew Flatt,
Robert Bruce Findler,
and John Clements

June 6, 2010

This reference manual describes the GUI toolbox that is part of Racket and whose core is implemented by the GRacket executable.

```
(require racket/gui/base)
```

The `racket/gui/base` library provides all of the class, interface, and procedure bindings defined in this manual. At run time, this library needs primitive graphics support that the GRacket executable provides; this library cannot run inside the Racket executable.

```
#lang racket/gui
```

The `racket/gui` language combines all bindings of the `racket` language and the `racket/gui/base` modules.

Contents

1 Overview

For documentation purposes, the graphics toolbox is organized into three parts:

- The *windowing toolbox*, for implementing form-filling GUI programs (such as a database query window) using buttons, menus, text fields, and events. The windowing toolbox is described in §1.1 “Windowing”.
- The *drawing toolbox*, for drawing pictures or implementing dynamic GUI programs (such as a video game) using drawing canvases, pens, and brushes. The drawing toolbox is described in §1.2 “Drawing”.
- The *editor toolbox*, for developing traditional text editors, editors that mix text and graphics, or free-form layout editors (such as a word processor, HTML editor, or icon-based file browser). The editor toolbox is described in §1.3 “Editor”.

These three parts roughly represent layers of increasing sophistication. Simple GUI programs access only the windowing toolbox directly, more complex programs use both the windowing and drawing toolboxes, and large-scale applications rely on all three toolboxes.

1.1 Windowing

The Racket windowing toolbox provides the basic building blocks of GUI programs, including frames (top-level windows), modal dialogs, menus, buttons, check boxes, text fields, and radio buttons. The toolbox provides these building blocks via built-in classes, such as the `frame%` class:

```
; Make a frame by instantiating the frame% class
(define frame (new frame% [label "Example"]))

; Show the frame by calling its show method
(send frame show #t)
```

The built-in classes provide various mechanisms for handling GUI events. For example, when instantiating the `button%` class, the programmer supplies an event callback procedure to be invoked when the user clicks the button. The following example program creates a frame with a text message and a button; when the user clicks the button, the message changes:

```
; Make a frame by instantiating the frame% class
(define frame (new frame% [label "Example"]))

; Make a static text message in the frame
(define msg (new message% [parent frame]
```

```
                          [label "No events so far..."]))

; Make a button in the frame
(new button% [parent frame]
             [label "Click Me"]
             ; Callback procedure for a button click:
             (callback (lambda (button event)
                         (send msg set-label "Button click"))))

; Show the frame by calling its show method
(send frame show #t)
```

Programmers never implement the GUI event loop directly. Instead, the system automatically pulls each event from an internal queue and dispatches the event to an appropriate window. The dispatch invokes the window's callback procedure or calls one of the window's methods. In the above program, the system automatically invokes the button's callback procedure whenever the user clicks Click Me.

If a window receives multiple kinds of events, the events are dispatched to methods of the window's class instead of to a callback procedure. For example, a drawing canvas receives update events, mouse events, keyboard events, and sizing events; to handle them, a programmer must derive a new class from the built-in `canvas%` class and override the event-handling methods. The following expression extends the frame created above with a canvas that handles mouse and keyboard events:

```
; Derive a new canvas (a drawing window) class to handle events
(define my-canvas%
  (class canvas% ; The base class is canvas%
    ; Define overriding method to handle mouse events
    (define/override (on-event event)
      (send msg set-label "Canvas mouse"))
    ; Define overriding method to handle keyboard events
    (define/override (on-char event)
      (send msg set-label "Canvas keyboard"))
    ; Call the superclass init, passing on all init args
    (super-new)))

; Make a canvas that handles events in the frame
(new my-canvas% [parent frame])
```

After running the above code, manually resize the frame to see the new canvas. Moving the cursor over the canvas calls the canvas's `on-event` method with an object representing a motion event. Clicking on the canvas calls `on-event`. While the canvas has the keyboard focus, typing on the keyboard invokes the canvas's `on-char` method.

The system dispatches GUI events sequentially; that is, after invoking an event-handling callback or method, the system waits until the handler returns before dispatching the next event. To illustrate the sequential nature of events, we extend the frame again, adding a Pause button:

```
(new button% [parent frame]
             [label "Pause"]
             [callback (lambda (button event) (sleep 5))])
```

After the user clicks Pause, the entire frame becomes unresponsive for five seconds; the system cannot dispatch more events until the call to `sleep` returns. For more information about event dispatching, see §1.1.4 "Event Dispatching and Eventspaces".

In addition to dispatching events, the GUI classes also handle the graphical layout of windows. Our example frame demonstrates a simple layout; the frame's elements are lined up top-to-bottom. In general, a programmer specifies the layout of a window by assigning each GUI element to a parent container. A vertical container, such as a frame, arranges its children in a column, and a horizontal container arranges its children in a row. A container can be a child of another container; for example, to place two buttons side-by-side in our frame, we create a horizontal panel for the new buttons:

```
(define panel (new horizontal-panel% [parent frame]))
(new button% [parent panel]
             [label "Left"]
             [callback (lambda (button event)
                         (send msg set-label "Left click"))])
(new button% [parent panel]
             [label "Right"]
             [callback (lambda (button event)
                         (send msg set-label "Right click"))])
```

For more information about window layout and containers, see §1.1.2 "Geometry Management".

1.1.1 Core Windowing Classes

The fundamental graphical element in the windowing toolbox is an *area*. The following classes implement the different types of areas in the windowing toolbox:

- *Containers* — areas that can contain other areas:
  - `frame%` — a *frame* is a top-level window that the user can move and resize.
  - `dialog%` — a *dialog* is a modal top-level window; when a dialog is shown, other top-level windows are disabled until the dialog is dismissed.

  - panel% — a *panel* is a subcontainer within a container. The toolbox provides three subclasses of panel%: vertical-panel%, horizontal-panel%, and tab-panel%.
  - pane% — a *pane* is a lightweight panel. It has no graphical representation or event-handling capabilities. The pane% class has three subclasses: vertical-pane%, horizontal-pane%, and grow-box-spacer-pane%.

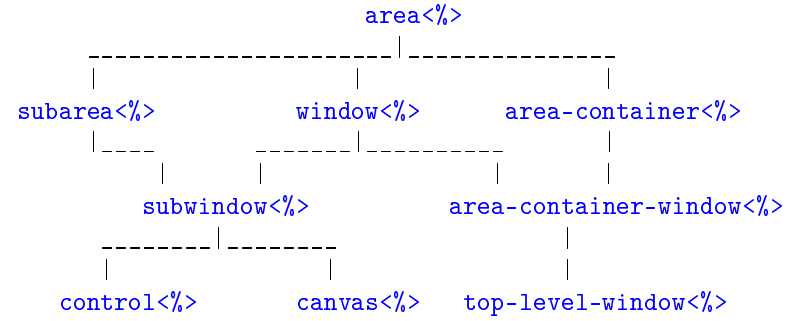
- *Containees* — areas that must be contained within other areas:
  - panel% — a panel is a containee as well as a container.
  - pane% — a pane is a containee as well as a container.
  - canvas% — a *canvas* is a subwindow for drawing on the screen.
  - editor-canvas% — an *editor canvas* is a subwindow for displaying a text editor or pasteboard editor. The editor-canvas% class is documented with the editor classes in §1.3 "Editor".
  - *Controls* — containees that the user can manipulate:
    * message% — a *message* is a static text field or bitmap with no user interaction.
    * button% — a *button* is a clickable control.
    * check-box% — a *check box* is a clickable control; the user clicks the control to set or remove its check mark.
    * radio-box% — a *radio box* is a collection of mutually exclusive *radio buttons*; when the user clicks a radio button, it is selected and the radio box's previously selected radio button is deselected.
    * choice% — a *choice item* is a pop-up menu of text choices; the user selects one item in the control.
    * list-box% — a *list box* is a scrollable lists of text choices; the user selects one or more items in the list (depending on the style of the list box).
    * text-field% — a *text field* is a box for simple text entry.
    * combo-field% — a *combo field* combines a text field with a pop-up menu of choices.
    * slider% — a *slider* is a dragable control that selects an integer value within a fixed range.
    * gauge% — a *gauge* is an output-only control (the user cannot change the value) for reporting an integer value within a fixed range.

As suggested by the above listing, certain areas, called containers, manage certain other areas, called containees. Some areas, such as panels, are both containers and containees.

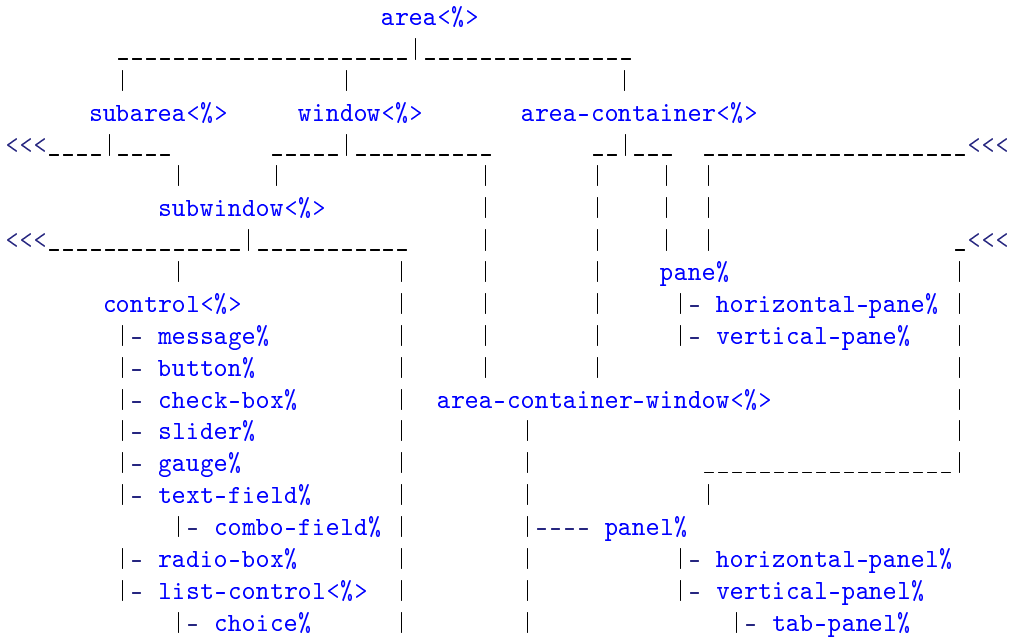
Most areas are *windows*, but some are *non-windows*. A window, such as a panel, has a graphical representation, receives keyboard and mouse events, and can be disabled or hidden. In contrast, a non-window, such as a pane, is useful only for geometry management; a non-window does not receive mouse events, and it cannot be disabled or hidden.

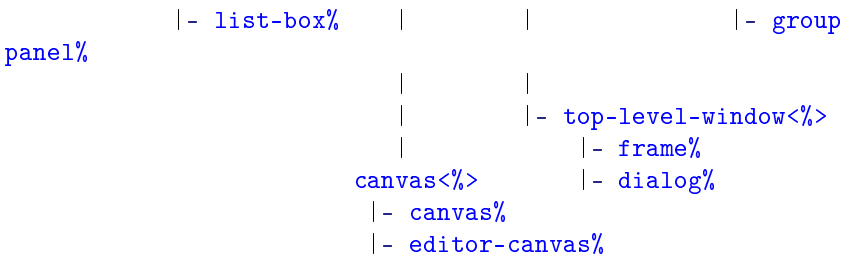
Every area is an instance of the area<%> interface. Each container is also an instance of the area-container<%> interface, whereas each containee is an instance of subarea<%>. Windows are instances of window<%>. The area-container<%>, subarea<%>, and window<%> interfaces are subinterfaces of area<%>.

The following diagram shows more of the type hierarchy under area<%>:



The diagram below extends the one above to show the complete type hierarchy under area<%>. (Some of the types are represented by interfaces, and some types are represented by classes. In principle, every area type should be represented by an interface, but whenever the windowing toolbox provides a concrete implementation, the corresponding interface is omitted from the toolbox.) To avoid intersecting lines, the hierarchy is drawn for a cylindrical surface; lines from subarea<%> and subwindow<%> wrap from the left edge of the diagram to the right edge.

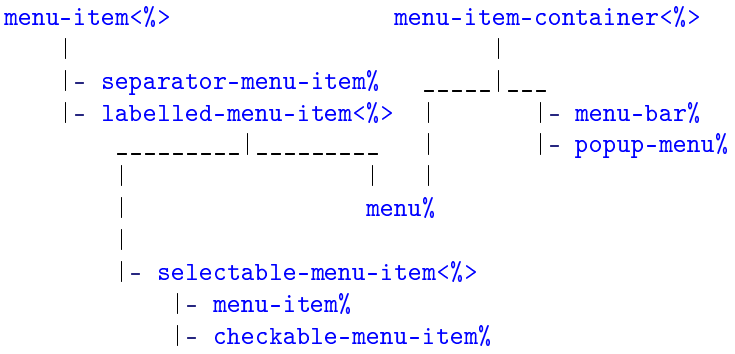




Menu bars, menus, and menu items are graphical elements, but not areas (i.e., they do not have all of the properties that are common to areas, such as an adjustable graphical size). Instead, the menu classes form a separate container–containee hierarchy:

- *Menu Item Containers*
  - menu-bar% — a *menu bar* is a top-level collection of menus that are associated with a frame.
  - menu% — a *menu* contains a set of menu items. The menu can appear in a menu bar, in a popup menu, or as a submenu in another menu.
  - popup-menu% — a *popup menu* is a top-level menu that is dynamically displayed in a canvas or editor canvas.
- *Menu Items*
  - separator-menu-item% — a *separator* is an unselectable line in a menu or popup menu.
  - menu-item% — a *plain menu item* is a selectable text item in a menu. When the item is selected, its callback procedure is invoked.
  - checkable-menu-item% — a *checkable menu item* is a text item in a menu; the user selects a checkable menu item to toggle a check mark next to the item.
  - menu% — a menu is a menu item as well as a menu item container.

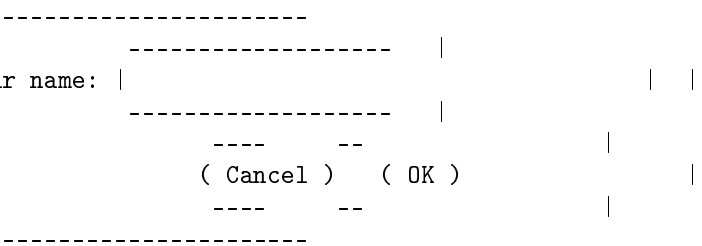
The following diagram shows the complete type hierarchy for the menu system:



1.1.2 Geometry Management

The windowing toolbox's geometry management makes it easy to design windows that look right on all platforms, despite different graphical representations of GUI elements. Geometry management is based on containers; each container arranges its children based on simple constraints, such as the current size of a frame and the natural size of a button.

The built-in container classes include horizontal panels (and panes), which align their children in a row, and vertical panels (and panes), which align their children in a column. By nesting horizontal and vertical containers, a programmer can achieve most any layout. For example, we can construct a dialog with the following shape:



with the following program:

```
; Create a dialog
(define dialog (instantiate dialog% ("Example")))

; Add a text field to the dialog
(new text-field% [parent dialog] [label "Your name"])

; Add a horizontal panel to the dialog, with centering for buttons
(define panel (new horizontal-panel% [parent dialog]
                                     [alignment '(center center)]))

; Add Cancel and Ok buttons to the horizontal panel
(new button% [parent panel] [label "Cancel"])
(new button% [parent panel] [label "Ok"])
(when (system-position-ok-before-cancel?)
  (send panel change-children reverse))

; Show the dialog
(send dialog show #t)
```

Each container arranges its children using the natural size of each child, which usually depends on instantiation parameters of the child, such as the label on a button or the number of choices in a radio box. In the above example, the dialog stretches horizontally to match the minimum width of the text field, and it stretches vertically to match the total height of the

field and the buttons. The dialog then stretches the horizontal panel to fill the bottom half of the dialog. Finally, the horizontal panel uses the sum of the buttons' minimum widths to center them horizontally.

As the example demonstrates, a stretchable container grows to fill its environment, and it distributes extra space among its stretchable children. By default, panels are stretchable in both directions, whereas buttons are not stretchable in either direction. The programmer can change whether an individual GUI element is stretchable.

The following subsections describe the container system in detail, first discussing the attributes of a containee in §1.1.2.1 "Containees", and then describing the attributes of a container in §1.1.2.2 "Containers". In addition to the built-in vertical and horizontal containers, programmers can define new types of containers as discussed in the final subsection, §1.1.2.3 "Defining New Types of Containers".

Containees

Each containee, or child, has the following properties:

- a *graphical minimum width* and a *graphical minimum height*;
- a *requested minimum width* and a *requested minimum height*;
- horizontal and vertical *stretchability* (on or off); and
- horizontal and vertical margins.

A container arranges its children based on these four properties of each containee. A containee's parent container is specified when the containee is created, and the parent cannot be changed. However, a containee can be hidden or deleted within its parent, as described in §1.1.2.2 "Containers".

The *graphical minimum size* of a particular containee, as reported by `get-graphical-min-size`, depends on the platform, the label of the containee (for a control), and style attributes specified when creating the containee. For example, a button's minimum graphical size ensures that the entire text of the label is visible. The graphical minimum size of a control (such as a button) cannot be changed; it is fixed at creation time. (A control's minimum size is *not* recalculated when its label is changed.) The graphical minimum size of a panel or pane depends on the total minimum size of its children and the way that they are arranged.

To select a size for a containee, its parent container considers the containee's *requested minimum size* rather than its graphical minimum size (assuming the requested minimum is larger than the graphical minimum). Unlike the graphical minimum, the requested minimum size of a containee can be changed by a programmer at any time using the `min-width` and `min-height` methods.

Unless a containee is stretchable (in a particular direction), it always shrinks to its minimum size (in the corresponding direction). Otherwise, containees are stretched to fill all available space in a container. Each containee begins with a default stretchability. For example, buttons are not initially stretchable, whereas a one-line text field is initially stretchable in the horizontal direction. A programmer can change the stretchability of a containee at any time using the `stretchable-width` and `stretchable-height` methods.

A *margin* is space surrounding a containee. Each containee's margin is independent of its minimum size, but from the container's point of view, a margin effectively increases the minimum size of the containee. For example, if a button has a vertical margin of `2`, then the container must allocate enough room to leave two pixels of space above and below the button, in addition to the space that is allocated for the button's minimum height. A programmer can adjust a containee's margin with `horiz-margin` and `vert-margin`. The default margin is `2` for a control, and `0` for any other type of containee.

In practice, the requested minimum size and margin of a control are rarely changed, although they are often changed for a canvas. Stretchability is commonly adjusted for any type of containee, depending on the visual effect desired by the programmer.

Containers

A container has the following properties:

- a list of (non-deleted) children containees;
- a requested minimum width and a requested minimum height;
- a spacing used between the children;
- a border margin used around the total set of children;
- horizontal and vertical stretchability (on or off); and
- an alignment setting for positioning leftover space.

These properties are factored into the container's calculation of its own size and the arrangement of its children. For a container that is also a containee (e.g., a panel), the container's requested minimum size and stretchability are the same as for its containee aspect.

A containee's parent container is specified when the containee is created, and the parent cannot be changed. However, a containee window can be hidden or deleted within its parent container (but a non-window containee cannot be hidden or deleted):

- A *hidden* child is invisible to the user, but space is still allocated for each hidden child within a container. To hide or show a child, call the child's `show` method.

- A *deleted* child is hidden *and* ignored by container as it arranges its other children, so no space is reserved in the container for a deleted child. To make a child deleted or non-deleted, call the container's `delete-child` or `add-child` method (which calls the child's `show` method).

When a child is created, it is initially shown and non-deleted. A deleted child is subject to garbage collection when no external reference to the child exists. A list of non-deleted children (hidden or not) is available from a container through its `get-children` method.

The order of the children in a container's non-deleted list is significant. For example, a vertical panel puts the first child in its list at the top of the panel, and so on. When a new child is created, it is put at the end of its container's list of children. The order of a container's list can be changed dynamically via the `change-children` method. (The `change-children` method can also be used to activate or deactivate children.)

The graphical minimum size of a container, as reported by `get-graphical-min-size`, is calculated by combining the minimum sizes of its children (summing them or taking the maximum, as appropriate to the layout strategy of the container) along with the spacing and border margins of the container. A larger minimum may be specified by the programmer using `min-width` and `min-height` methods; when the computed minimum for a container is larger than the programmer-specified minimum, then the programmer-specified minimum is ignored.

A container's spacing determines the amount of space left between adjacent children in the container, in addition to any space required by the children's margins. A container's border margin determines the amount of space to add around the collection of children; it effectively decreases the area within the container where children can be placed. A programmer can adjust a container's border and spacing dynamically via the `border` and `spacing` methods. The default border and spacing are `0` for all container types.

Because a panel or pane is a containee as well as a container, it has a containee margin in addition to its border margin. For a panel, these margins are not redundant because the panel can have a graphical border; the border is drawn inside the panel's containee margin, but outside the panel's border margin.

For a top-level-window container, such as a frame or dialog, the container's stretchability determines whether the user can resize the window to something larger than its minimum size. Thus, the user cannot resize a frame that is not stretchable. For other types of containers (i.e., panels and panes), the container's stretchability is its stretchability as a containee in some other container. All types of containers are initially stretchable in both directions—except instances of `grow-box-spacer-pane%`, which is intended as a lightweight spacer class rather than a useful container class—but a programmer can change the stretchability of an area at any time via the `stretchable-width` and `stretchable-height` methods.

The alignment specification for a container determines how it positions its children when the container has leftover space. (A container can only have leftover space in a particular

direction when none of its children are stretchable in that direction.) For example, when the container's horizontal alignment is `'left`, the children are left-aligned in the container and leftover space is accumulated to the right. When the container's horizontal alignment is `'center`, each child is horizontally centered in the container. A container's alignment is changed with the `set-alignment` method.

Defining New Types of Containers

Although nested horizontal and vertical containers can express most layout patterns, a programmer can define a new type of container with an explicit layout procedure. A programmer defines a new type of container by deriving a class from `panel%` or `pane%` and overriding the `container-size` and `place-children` methods. The `container-size` method takes a list of size specifications for each child and returns two values: the minimum width and height of the container. The `place-children` method takes the container's size and a list of size specifications for each child, and returns a list of sizes and placements (in parallel to the original list).

An input size specification is a list of four values:

- the child's minimum width;
- the child's minimum height;
- the child's horizontal stretchability (`#t` means stretchable, `#f` means not stretchable); and
- the child's vertical stretchability.

For `place-children`, an output position and size specification is a list of four values:

- the child's new horizontal position (relative to the parent);
- the child's new vertical position;
- the child's new actual width;
- the child's new actual height.

The widths and heights for both the input and output include the children's margins. The returned position for each child is automatically incremented to account for the child's margin in placing the control.

1.1.3 Mouse and Keyboard Events

Whenever the user moves the mouse, clicks or releases a mouse button, or presses a key on the keyboard, an event is generated for some window. The window that receives the event depends on the current state of the graphic display:

- The receiving window of a mouse event is usually the window under the cursor when the mouse is moved or clicked. If the mouse is over a child window, the child window receives the event rather than its parent.

  When the user clicks in a window, the window "grabs" the mouse, so that *all* mouse events go to that window until the mouse button is released (regardless of the location of the cursor). As a result, a user can click on a scrollbar thumb and drag it without keeping the cursor strictly inside the scrollbar control.

  A mouse button-release event is normally generated for each mouse button-down event, but a button-release event might get dropped. For example, a modal dialog might appear and take over the mouse. More generally, any kind of mouse event can get dropped in principle, so avoid algorithms that depend on precise mouse-event sequences. For example, a mouse tracking handler should reset the tracking state when it receives an event other than a dragging event.

- The receiving window of a keyboard event is the window that owns the *keyboard focus* at the time of the event. Only one window owns the focus at any time, and focus ownership is typically displayed by a window in some manner. For example, a text field control shows focus ownership by displaying a blinking caret.

  Within a top-level window, only certain kinds of subwindows can have the focus, depending on the conventions of the platform. Furthermore, the subwindow that initially owns the focus is platform-specific. A user can moves the focus in various ways, usually by clicking the target window. A program can use the `focus` method to move the focus to a subwindow or to set the initial focus.

  Under X, a `'wheel-up` or `'wheel-down` event may be sent to a window other than the one with the keyboard focus, because X generates wheel events based on the location of the mouse pointer.

  A key-press event may correspond to either an actual key press or an auto-key repeat. Multiple key-press events without intervening key-release events normally indicate an auto-key. Like any input event, however, key-release events sometimes get dropped (e.g., due to the appearance of a modal dialog).

Controls, such as buttons and list boxes, handle keyboard and mouse events automatically, eventually invoking the callback procedure that was provided when the control was created. A canvas propagates mouse and keyboard events to its `on-event` and `on-char` methods, respectively.

A mouse and keyboard event is delivered in a special way to its window. Each ancestor of the receiving window gets a chance to intercept the event through the `on-subwindow-event`

and on-subwindow-char methods. See the method descriptions for more information.

The default on-subwindow-char method for a top-level window intercepts keyboard events to detect menu-shortcut events and focus-navigation events. See on-subwindow-char in frame% and on-subwindow-char in dialog% for details. Certain OS-specific key combinations are captured at a low level, and cannot be overridden. For example, under Windows and X, pressing and releasing Alt always moves the keyboard focus to the menu bar. Similarly, Alt-Tab switches to a different application under Windows. (Alt-Space invokes the system menu under Windows, but this shortcut is implemented by on-system-menu-char, which is called by on-subwindow-char in frame% and on-subwindow-char in dialog%.)

1.1.4 Event Dispatching and Eventspaces

A graphical user interface is an inherently multi-threaded system: one thread is the program managing windows on the screen, and the other thread is the user moving the mouse and typing at the keyboard. GUI programs typically use an *event queue* to translate this multi-threaded system into a sequential one, at least from the programmer's point of view. Each user action is handled one at a time, ignoring further user actions until the previous one is completely handled. The conversion from a multi-threaded process to a single-threaded one greatly simplifies the implementation of GUI programs.

Despite the programming convenience provided by a purely sequential event queue, certain situations require a less rigid dialog with the user:

- *Nested event handling:* In the process of handling an event, it may be necessary to obtain further information from the user. Usually, such information is obtained via a modal dialog; in whatever fashion the input is obtained, more user events must be received and handled before the original event is completely handled. To allow the further processing of events, the handler for the original event must explicitly *yield* to the system. Yielding causes events to be handled in a nested manner, rather than in a purely sequential manner.
- *Asynchronous event handling:* An application may consist of windows that represent independent dialogs with the user. For example, a drawing program might support multiple drawing windows, and a particularly time-consuming task in one window (e.g., a special filter effect on an image) should not prevent the user from working in a different window. Such an application needs sequential event handling for each individual window, but asynchronous (potentially parallel) event handling across windows. In other words, the application needs a separate event queue for each window, and a separate event-handling thread for each event queue.

An *eventspace* is a context for processing GUI events. Each eventspace maintains its own queue of events, and events in a single eventspace are dispatched sequentially by a designated *handler thread*. An event-handling procedure running in this handler thread can yield to the

system by calling `yield`, in which case other event-handling procedures may be called in a nested (but single-threaded) manner within the same handler thread. Events from different eventspaces are dispatched asynchronously by separate handler threads.

When a frame or dialog is created without a parent, it is associated with the current eventspace as described in §1.1.4.3 "Creating and Setting the Eventspace". Events for a top-level window and its descendants are always dispatched in the window's eventspace. Every dialog is modal; a dialog's `show` method implicitly calls `yield` to handle events while the dialog is shown. (See also §1.1.4.2 "Eventspaces and Threads" for information about threads and modal dialogs.) Furthermore, when a modal dialog is shown, the system disables all other top-level windows in the dialog's eventspace, but windows in other eventspaces are unaffected by the modal dialog. (Disabling a window prevents mouse and keyboard events from reaching the window, but other kinds of events, such as update events, are still delivered.)

Event Types and Priorities

In addition to events corresponding to user and windowing actions, such as button clicks, key presses, and updates, the system dispatches two kinds of internal events: timer events and explicitly queued events.

Timer events are created by instances of `timer%`. When a timer is started and then expires, the timer queues an event to call the timer's `notify` method. Like a top-level window, each timer is associated with a particular eventspace (the current eventspace as described in §1.1.4.3 "Creating and Setting the Eventspace") when it is created, and the timer queues the event in its eventspace.

Explicitly queued events are created with `queue-callback`, which accepts a callback procedure to handle the event. The event is enqueued in the current eventspace at the time of the call to `queue-callback`, with either a high or low priority as specified by the (optional) second argument to `queue-callback`.

An eventspace's event queue is actually a priority queue with events sorted according to their kind, from highest-priority (dispatched first) to lowest-priority (dispatched last):

- The highest-priority events are high-priority events installed with `queue-callback`.
- Timer events have the second-highest priority.
- Graphical events, such as mouse clicks or window updates, have the second-lowest priority.
- The lowest-priority events are low-priority events installed with `queue-callback`.

Although a programmer has no direct control over the order in which events are dispatched, a programmer can control the timing of dispatches by setting the event dispatch handler via

the event-dispatch-handler parameter. This parameter and other eventspace procedures are described in more detail in §2.2.2 “Eventspaces”.

Eventspaces and Threads

When a new eventspace is created, a corresponding handler thread is created for the eventspace. When the system dispatches an event for an eventspace, it always does so in the eventspace’s handler thread. A handler procedure can create new threads that run indefinitely, but as long as the handler thread is running a handler procedure, no new events can be dispatched for the corresponding eventspace.

When a handler thread shows a dialog, the dialog’s show method implicitly calls yield for as long as the dialog is shown. When a non-handler thread shows a dialog, the non-handler thread simply blocks until the dialog is dismissed. Calling yield with no arguments from a non-handler thread has no effect. Calling yield with a semaphore from a non-handler thread is equivalent to calling semaphore-wait.

Creating and Setting the Eventspace

Whenever a frame, dialog, or timer is created, it is associated with the *current eventspace* as determined by the current-eventspace parameter (see §10.3.2 “Parameters”).

The make-eventspace procedure creates a new eventspace. The following example creates a new eventspace and a new frame in the eventspace (the parameterize syntactic form temporary sets a parameter value):

```
(let ([new-es (make-eventspace)])
  (parameterize ([current-eventspace new-es])
    (new frame% [label "Example"])))
```

When an eventspace is created, it is placed under the management of the current custodian. When a custodian shuts down an eventspace, all frames and dialogs associated with the eventspace are destroyed (without calling can-close? or on-close in top-level-window<%>), all timers in the eventspace are stopped, and all enqueued callbacks are removed. Attempting to create a new window, timer, or explicitly queued event in a shut-down eventspace raises the exn:misc exception.

An eventspace is a synchronizable event (not to be confused with a GUI event), so it can be used with sync. As a synchronizable event, an eventspace is in a blocking state when a frame is visible, a timer is active, a callback is queued, or a menu-bar% is created with a 'root parent. (Note that the blocking state of an eventspace is unrelated to whether an event is ready for dispatching.)

Exceptions and Continuation Jumps

Whenever the system dispatches an event, the call to the handler procedure is wrapped so that full continuation jumps are not allowed to escape from the dispatch, and escape continuation jumps are blocked at the dispatch site. The following `block` procedure illustrates how the system blocks escape continuation jumps:

```
(define (block f)
  ; calls f and returns void if f tries to escape
  (let ([done? #f])
    (let/ec k
      (dynamic-wind
        void
        (lambda () (begin0 (f) (set! done? #t)))
        (lambda () (unless done? (k (void))))))))

> (block (lambda () 5))
5
> (let/ec k (block (lambda () (k 10))))
> (let/ec k ((lambda () (k 10))) 11)
10
> (let/ec k (block (lambda () (k 10))) 11)
11
```

Calls to the event dispatch handler are also protected with `block`.

This blocking of continuation jumps complicates the interaction between `with-handlers` and `yield` (or the default event dispatch handler). For example, in evaluating the expression

```
(with-handlers ([(lambda (x) #t)
                 (lambda (x) (error "error during yield"))])
   (yield))
```

the `"error during yield"` handler is *never* called, even if a callback procedure invoked by `yield` raises an exception. The `with-handlers` expression installs an exception handler that tries to jump back to the context of the `with-handlers` expression before invoking a handler procedure; this jump is blocked by the dispatch within `yield`, so `"error during yield"` is never printed. Exceptions during `yield` are "handled" in the sense that control jumps out of the event handler, but `yield` may dispatch another event rather than escaping or returning.

The following expression demonstrates a more useful way to handle exceptions within `yield`, for the rare cases where such handling is useful:

```
(let/ec k
  (call-with-exception-handler
```

```
(lambda (x)
  (error "error during yield")
  (k))
(lambda ()
  (yield))))
```

This expression installs an exception handler that prints an error message *before* trying to escape. Like the continuation escape associated with `with-handlers`, the escape to `k` never succeeds. Nevertheless, if an exception is raised by an event handler during the call to `yield`, an error message is printed before control returns to the event dispatcher within `yield`.

1.2 Drawing

Drawing in Racket requires a *device context* (*DC*), which is an instance of the `dc<%>` interface. For example, the `get-dc` method of a canvas returns a `dc<%>` instance for drawing into the canvas window. Other kinds of DCs draw to different kinds of devices:

- `bitmap-dc%` — a *bitmap DC* draws to an offscreen bitmap.
- `post-script-dc%` — a *PostScript DC* records drawing commands to a PostScript file.
- `printer-dc%` — a *printer DC* draws to a platform-specific printer device (Windows, Mac OS X).

Tools that are used for drawing include the following: `pen%` objects for drawing lines and shape outlines, `brush%` objects for filling shapes, `bitmap%` objects for storing bitmaps, and `dc-path%` objects for describing paths to draw and fill.

The following example creates a frame with a drawing canvas, and then draws a round, blue face with square, yellow eyes and a smiling, red mouth:

```
; Make a 300 x 300 frame
(define frame (new frame% [label "Drawing Example"]
                          [width 300]
                          [height 300]))
; Make the drawing area
(define canvas (new canvas% [parent frame]))
; Get the canvas's drawing context
(define dc (send canvas get-dc))

; Make some pens and brushes
(define no-pen (make-object pen% "BLACK" 1 'transparent))
(define no-brush (make-object brush% "BLACK" 'transparent))
```

```
(define blue-brush (make-object brush% "BLUE" 'solid))
(define yellow-brush (make-object brush% "YELLOW" 'solid))
(define red-pen (make-object pen% "RED" 2 'solid))

; Define a procedure to draw a face
(define (draw-face dc)
  (send dc set-pen no-pen)
  (send dc set-brush blue-brush)
  (send dc draw-ellipse 50 50 200 200)

  (send dc set-brush yellow-brush)
  (send dc draw-rectangle 100 100 10 10)
  (send dc draw-rectangle 200 100 10 10)

  (send dc set-brush no-brush)
  (send dc set-pen red-pen)
  (let ([-pi (atan 0 -1)])
    (send dc draw-arc 75 75 150 150 (* 5/4 -pi) (* 7/4 -pi))))

; Show the frame
(send frame show #t)
; Wait a second to let the window get ready
(sleep/yield 1)
; Draw the face
(draw-face dc)
```

The `sleep/yield` call is necessary under X because drawing to the canvas has no effect when the canvas is not shown. Although the `(send frame show #t)` expression queues a show request for the frame, the actual display of the frame and its canvas requires handling several events. The `sleep/yield` procedure pauses for a specified number of seconds, handling events while it pauses.

One second is plenty of time for the frame to show itself, but a better solution is to create a canvas with a paint callback function (or overriding `on-paint`). Using a paint callback function is better for all platforms; when the canvas in the above example is resized or temporarily covered by another window, the face disappears. To ensure that the face is redrawn whenever the canvas itself is repainted, we provide a paint callback when creating the canvas:

```
; Make a 300 x 300 frame
(define frame (new frame% [label "Drawing Example"]
                          [width 300]
                          [height 300]))

; Make the drawing area with a paint callback
(define canvas
```

```
  (new canvas% [parent frame]
               [paint-callback
                (lambda (canvas dc) (draw-face dc))]))

; ... pens, brushes, and draw-face are the same as above ...

; Show the frame
(send frame show #t)
```

Suppose that `draw-face` creates a particularly complex face that takes a long time to draw. We might want to draw the face once into an offscreen bitmap, and then have the paint callback copy the cached bitmap image onto the canvas whenever the canvas is updated. To draw into a bitmap, we first create a `bitmap%` object, and then we create a `bitmap-dc%` to direct drawing commands into the bitmap:

```
; ... pens, brushes, and draw-face are the same as above ...

; Create a 300 x 300 bitmap
(define face-bitmap (make-object bitmap% 300 300))
; Create a drawing context for the bitmap
(define bm-dc (make-object bitmap-dc% face-bitmap))
; A bitmap's initial content is undefined; clear it before drawing
(send bm-dc clear)

; Draw the face into the bitmap
(draw-face bm-dc)

; Make a 300 x 300 frame
(define frame (new frame% [label "Drawing Example"]
                          [width 300]
                          [height 300]))

; Make a drawing area whose paint callback copies the bitmap
(define canvas
  (new canvas% [parent frame]
               [paint-callback
                (lambda (canvas dc)
                  (send dc draw-bitmap face-bitmap 0 0))]))

; Show the frame
(send frame show #t)
```

For all types of DCs, the drawing origin is the top-left corner of the DC. When drawing to a window or bitmap, DC units initially correspond to pixels, but the `set-scale` method changes the scale. When drawing to a PostScript or printer device, DC units initially correspond to points (1/72 of an inch).

More complex shapes are typically best implemented with *paths*. The following example uses paths to draw the Racket logo. It also enables smoothing, so that the logo's curves are anti-aliased when smoothing is available. (Smoothing is always available under Mac OS X, smoothing is available under Windows XP or when `"gdiplus.dll"` is installed, and smoothing is available under X when Cairo is installed before GRacket is compiled.)

```
(require mzlib/math) ; for pi

; Construct paths for a 630 x 630 logo

(define left-lambda-path ; left side of the lambda
  (let ([p (new dc-path%)])
    (send p move-to 153 44)
    (send p line-to 161.5 60)
    (send p curve-to 202.5 49 230 42 245 61)
    (send p curve-to 280.06 105.41 287.5 141 296.5 186)
    (send p curve-to 301.12 209.08 299.11 223.38 293.96 244)
    (send p curve-to 281.34 294.54 259.18 331.61 233.5 375)
    (send p curve-to 198.21 434.63 164.68 505.6 125.5 564)
    (send p line-to 135 572)
    p))

(define left-logo-path ; left side of the lambda and circle
  (let ([p (new dc-path%)])
    (send p append left-lambda-path)
    (send p arc 0 0 630 630 (* 47/72 2 pi) (* 121/360 2 pi) #f)
    p))

(define bottom-lambda-path
  (let ([p (new dc-path%)])
    (send p move-to 135 572)
    (send p line-to 188.5 564)
    (send p curve-to 208.5 517 230.91 465.21 251 420)
    (send p curve-to 267 384 278.5 348 296.5 312)
    (send p curve-to 301.01 302.98 318 258 329 274)
    (send p curve-to 338.89 288.39 351 314 358 332)
    (send p curve-to 377.28 381.58 395.57 429.61 414 477)
    (send p curve-to 428 513 436.5 540 449.5 573)
    (send p line-to 465 580)
    (send p line-to 529 545)
    p))

(define bottom-logo-path
  (let ([p (new dc-path%)])
    (send p append bottom-lambda-path)
    (send p arc 0 0 630 630 (* 157/180 2 pi) (* 47/72 2 pi) #f)
```

```
    p))

(define right-lambda-path
  (let ([p (new dc-path%)])
    (send p move-to 153 44)
    (send p curve-to 192.21 30.69 233.21 14.23 275 20)
    (send p curve-to 328.6 27.4 350.23 103.08 364 151)
    (send p curve-to 378.75 202.32 400.5 244 418 294)
    (send p curve-to 446.56 375.6 494.5 456 530.5 537)
    (send p line-to 529 545)
    p))

(define right-logo-path
  (let ([p (new dc-path%)])
    (send p append right-lambda-path)
    (send p arc 0 0 630 630 (* 157/180 2 pi) (* 121/360 2 pi) #t)
    p))

(define lambda-path ; the lambda by itself (no circle)
  (let ([p (new dc-path%)])
    (send p append left-lambda-path)
    (send p append bottom-lambda-path)
    (let ([t (make-object dc-path%)])
        (send t append right-lambda-path)
        (send t reverse)
        (send p append t))
    (send p close)
    p))

; This function draws the paths with suitable colors:
(define (paint-plt dc)
  ; Paint white lambda, no outline:
  (send dc set-pen "BLACK" 0 'transparent)
  (send dc set-brush "WHITE" 'solid)
  (send dc draw-path lambda-path)
  ; Paint outline and colors...
  (send dc set-pen "BLACK" 0 'solid)
  ; Draw red regions
  (send dc set-brush "RED" 'solid)
  (send dc draw-path left-logo-path)
  (send dc draw-path bottom-logo-path)
  ; Draw blue region
  (send dc set-brush "BLUE" 'solid)
  (send dc draw-path right-logo-path))

; Create a frame to display the logo on a light-purple background:
```

```
(define f (new frame% [label "Racket Logo"]))
(define c
  (new canvas%
       [parent f]
       [paint-callback
        (lambda (c dc)
          (send dc set-background (make-object color% 220 200 255))
          (send dc clear)
          (send dc set-smoothing 'smoothed)
          (send dc set-origin 5 5)
          (send dc set-scale 0.5 0.5)
          (paint-plt dc))]))
(send c min-client-width (/ 650 2))
(send c min-client-height (/ 650 2))
(send f show #t)
```

Drawing effects are not completely portable across platforms or across types of DC. Drawing in smoothed mode tends to produce more reliable and portable results than in unsmoothed mode, and drawing with paths tends to produce more reliable results even in unsmoothed mode. Drawing with a pen of width 0 or 1 in unsmoothed mode in an unscaled DC produces relatively consistent results for all platforms, but a pen width of 2 or drawing to a scaled DC looks significantly different in unsmoothed mode on different platforms and destinations.

1.3 Editor

The editor toolbox provides a foundation for two common kinds of applications:

- *Programs that need a sophisticated text editor* — The simple text field control is inadequate for text-intensive applications. Many programs need editors that can handle multiple fonts and non-text items.
- *Programs that need a canvas with dragable objects* — The drawing toolbox provides a generic drawing surface for plotting lines and boxes, but many applications need an interactive canvas, where the user can drag and resize individual objects.

Both kinds of applications need an extensible editor that can handle text, images, programmer-defined items, and even embedded editors. The difference between them is the layout of items. The editor toolbox therefore provides two kinds of editors via two classes:

- `text%` — in a *text editor*, items are automatically positioned in a paragraph flow.
- `pasteboard%` — in a *pasteboard editor*, items are explicitly positioned and dragable.

This editor architecture addresses the full range of real-world issues for an editor—including cut-and-paste, extensible file formats, and layered text styles—while supporting a high level of extensibility. Unfortunately, the system is fairly complex as a result, and using the editor classes effectively requires a solid understanding of the structure and terminology of the editor toolbox. Nevertheless, enough applications fit one (or both) of the descriptions above to justify the depth and complexity of the toolbox and the learning investment required to use it.

A brief example illustrates how editors work. To start, an editor needs an `editor-canvas%` to display its contents. Then, we can create a text editor and install it into the canvas:

```
(define f (new frame% [label "Simple Edit"]
                      [width 200]
                      [height 200]))
(define c (new editor-canvas% [parent f]))
(define t (new text%))
(send c set-editor t)
(send f show #t)
```

At this point, the editor is fully functional: the user can type text into the editor, but no cut-and-paste operations are available. We can support all of the standard operations on an editor via the menu bar:

```
(define mb (new menu-bar% [parent f]))
(define m-edit (new menu% [label "Edit"] [parent mb]))
(define m-font (new menu% [label "Font"] [parent mb]))
(append-editor-operation-menu-items m-edit #f)
(append-editor-font-menu-items m-font)
```

Now, the standard cut and paste operations work, and the user can even set font styles. The user can also insert an embedded editor by selecting Insert Text from the Edit menu; after selecting the menu item, a box appears in the editor with the caret inside. Typing with the caret in the box stretches the box as text is added, and font operations apply wherever the caret is active. Text on the outside of the box is rearranged as the box changes sizes. Note that the box itself can be copied and pasted.

The content of an editor is made up of *snips*. An embedded editor is a single snip from the embedding editor's point-of-view. To encode immediate text, a snip can be a single character, but more often a snip is a sequence of adjacent characters on the same line. The `find-snip` method extracts a snip from a text editor:

```
(send t find-snip 0 'after)
```

The above expression returns the first snip in the editor, which may be a string snip (for immediate text) or an editor snip (for an embedded editor).

An editor is not permanently attached to any display. We can take the text editor out of our canvas and put a pasteboard editor in the canvas, instead:

```
(define pb (new pasteboard%))
(send c set-editor pb)
```

With the pasteboard editor installed, the user can no longer type characters directly into the editor (because a pasteboard does not support directly entered text). However, the user can cut text from elsewhere and paste it into pasteboard, or select one of the Insert menu items in the Edit menu. Snips are clearly identifiable in a pasteboard editor (unlike a text editor) because each snip is separately dragable.

We can insert the old text editor (which we recently removed from the canvas) as an embedded editor in the pasteboard by explicitly creating an editor snip:

```
(define s (make-object editor-snip% t)) ; t is the old text editor
(send pb insert s)
```

An individual snip cannot be inserted into different editors at the same time, or inserted multiple times in the same editor:

```
(send pb insert s) ; no effect
```

However, we can make a deep copy of the snip and insert the copy into the pasteboard:

```
(send pb insert (send s copy))
```

Applications that use the editor classes typically derive new versions of the `text%` and `pasteboard%` classes. For example, to implement an append-only editor (which allows insertions only at the end and never allows deletions), derive a new class from `text%` and override the `can-insert?` and `can-delete?` methods:

```
(define append-only-text%
  (class text%
    (inherit last-position)
    (define/augment (can-insert? s l) (= s (last-position)))
    (define/augment (can-delete? s l) #f)
    (super-new)))
```

1.3.1 Editor Structure and Terminology

The editor toolbox supports extensible and nestable editors by decomposing an editor assembly into three functional parts:

- The *editor* itself stores the state of the text or pasteboard and handles most events and editing operations. The `editor<%>` interface defines the core editor functionality, but

editors are created as instances of `text%` or `pasteboard%`.

- A *snip* is a segment of information within the editor. Each snip can contain a sequence of characters, a picture, or an interactive object (such as an embedded editor). In a text editor, snips are constrained to fit on a single line and generally contain data of a single type. The `snip%` class implements a basic snip. Other snip classes include `string-snip%` for managing text, `image-snip%` for managing pictures, and `editor-snip%` for managing embedded editors.
- A *display* presents the editor on the screen. The display lets the user scroll around an editor or change editors. Most displays are instances of the `editor-canvas%` class, but the `editor-snip%` class also acts as a display for embedded editors.

These three parts are illustrated by a simple word processor. The editor corresponds to the text document. The editor object receives keyboard and mouse commands for editing the text. The text itself is distributed among snips. Each character could be a separate snip, or multiple characters on a single line could be grouped together into a snip. The display roughly corresponds to the window in which the text is displayed. While the editor manages the arrangement of the text as it is displayed into a window, the display determines which window to draw into and which part of the editor to display.

Each selectable entity in an editor is an *item*. In a pasteboard, all selection and dragging operations work on snips, so there is a one-to-one correspondence between snips and items. In an editor, one snip contains one or more consecutive items, and every item belongs to some snip. For example, in a simple text editor, each character is an item, but multiple adjacent characters may be grouped into a single snip. The number of items in a snip is the snip's *count*.

Each place where the insertion point can appear in a text editor is a *position*. A text editor with n items contains $n+1$ positions: one position before each item, and one position after the last item.

The order of snips within a pasteboard determines each snip's drawing plane. When two snips overlap within the pasteboard, the snip that is earlier in the order is in front of the other snip (i.e., the former is drawn after the latter, such that the former snip may cover part of the latter snip).

When an editor is drawn into a display, each snip and position has a *location*. The location of a position or snip is specified in coordinates relative to the top-left corner of the editor. Locations in an editor are only meaningful when the editor is displayed.

Administrators

Two extra layers of administration manage the display-editor and editor-snip connections. An editor never communicates directly with a display; instead, it always communicates with

an *editor administrator*, an instance of the `editor-admin%` class, which relays information to the display. Similarly, a snip communicates with a *snip administrator*, an instance of the `snip-admin%` class.

The administrative layers make the editor hierarchy flexible without forcing every part of an editor assembly to contain the functionality of several parts. For example, a text editor can be a single item within another editor; without administrators, the `text%` class would also have to contain all the functionality of a display (for the containing editor) and a snip (for the embedded editor). Using administrators, an editor class can serve as both a containing and an embedded editor without directly implementing the display and snip functionality.

A snip belongs to at most one editor via a single administrator. An editor also has only one administrator at a time. However, the administrator that connects the an editor to the standard display (i.e., an editor canvas) can work with other such administrators. In particular, the administrator of an `editor-canvas%` (each one has its own administrator) can work with other `editor-canvas%` administrators, allowing an editor to be displayed in multiple `editor-canvas%` windows at the same time.

When an editor is displayed by multiple canvases, one of the canvases' administrators is used as the editor's primary administrator. To handle user and update events for other canvases, the editor's administrator is temporarily changed and then restored through the editor's `set-admin` method. The return value of the editor's `get-admin` method thus depends on the context of the call.

Styles

A *style*, an instance of the `style<%>` interface, parameterizes high-level display information that is common to all snip classes. This includes the font, color, and alignment for drawing the item. A single style is attached to each snip.

Styles are hierarchical: each style is defined in terms of another style. There is a single *root style*, named `"Basic"`, from which all other styles in an editor are derived. The difference between a base style and each of its derived style is encoded in a *style delta* (or simply *delta*). A delta encodes changes such as

- change the font family to *X*;
- enlarge the font by adding *Y* to the point size;
- toggle the boldness of the font; or
- change everything to match the style description *Z*.

Style objects are never created separately; rather, they are always created through a *style list*, an instance of the `style-list%` class. A style list manages the styles, servicing external

requests to find a particular style, and it manages the hierarchical relationship between styles. A global style list is available, `the-style-list`, but new style lists can be created for managing separate style hierarchies. For example, each editor will typically have its own style list.

Each new style is defined in one of two ways:

- A *derived style* is defined in terms of a base style and a delta. Every style (except for the root style) has a base style, even if it does not depend on the base style in any way (i.e., the delta describes a fixed style rather than extensions to an existing style). (This is the usual kind of style inheritance, as found in word processors such as Microsoft Word.)
- A *join style* is defined in terms of two other styles: a base style and a *shift style*. The meaning of a join style is determined by reinterpreting the shift style; in the reinterpretation, the base style is used as the *root* style for the shift style. (This is analogous to multi-level styles, like the paragraph and character styles in FrameMaker. In this analogy, the paragraph style is the base style, and the character style is the shift style. However, FrameMaker allows only those two levels; with join styles support any number of levels.)

Usually, when text is inserted into a text editor, it inherits the style of the preceding snip. If text is inserted into an empty editor, the text is usually assigned a style called `"Standard"`. By default, the `"Standard"` style is unmodified from the root style. The default style name can be changed by overriding `default-style-name`.

The exception to the above is when `change-style` in `text%` is called with the current selection position (when the selection is a position and not a range). In that case, the style is remembered, and if the next editor-modifying action is a text insertion, the inserted text gets the remembered style.

See `get-styles-sticky` in `text%` for more information about the style of inserted text.

1.3.2 File Format

To allow editor content to be saved to a file, the editor classes implement a special file format called *WXME*. (The format is used when cutting and pasting between applications or eventspaces, too). The file format is not documented, except that it begins `WXME01`⟨*digit*⟩⟨*digit*⟩ `##` . Otherwise, the `load-file` and `save-file` methods define the format internally. The file format is the same for text and pasteboard editors. When a pasteboard saves its content to a file, it saves the snips from front to back, and also includes extra location information. The `wxme` library provides utilities for manipulating WXME files.

Editor data is read and written using `editor-stream-in%` and `editor-stream-out%` objects. Editor information can only be read from or written to one stream at a time. To write

one or more editors to a stream, first call the function write-editor-global-header to write initialization data into an output stream. When all editors are written to the stream, call write-editor-global-footer. Similarly, reading editors from a stream is initialized with read-editor-global-header and finalized with read-editor-global-footer. Optionally, to support streams that span versions of Racket, use write-editor-version and read-editor-version before the header operations.

The editor file data format can be embedded within another file, and it can be extended with new kinds of data. The editor file format can be extended in two ways: with snip- or content-specific data, and with editor-specific global data. These are described in the remainder of this section.

Encoding Snips

The generalized notion of a snip allows new snip types to be defined and immediately used in any editor class. Also, when two applications support the same kinds of snips, snip data can easily be cut and pasted between them, and the same data files will be readable by each program. This interoperability is due to a consistent encoding mechanism that is built into the snip system.

Graceful and extensible encoding of snips requires that two issues are addressed:

- The encoding function for a snip can be associated with the snip itself. To convert a snip from an encoded representation (e.g., as bytes in a file) to a memory object, a decoding function must be provided for each type of snip. Furthermore, a list of such decoders must be available to the high-level decoding process. This decoding mapping is defined by associating a *snip class* object to every snip. A snip class is an instance of the snip-class% class.
- Some editors may require additional information to be stored about a snip; this information is orthogonal to the type-specific information stored by the snip itself. For example, a pasteboard needs to remember a snip's location, while a text editor does not need this information. If data is being cut and pasted from one pasteboard to another, then information about relative locations needs to be maintained, but this information should not inhibit pasting into an editor. Extra data is associated with a snip through *editor data* objects, which are instances of the editor-data% class; decoding requires that each editor data object has an *editor data class*, which is an instance of the editor-data-class% class.

Snip classes, snip data, and snip data classes solve problems related to encoding and decoding snips. In an application that has no need for saving files or cut-and-paste, these issues can be safely ignored.

Snip Classes

Each snip can be associated to a snip class. This "class" is not a class description in the programmer's language; it is an object which provides a way to create new snips of the appropriate type from an encoded snip specification.

Snip class objects can be added to the eventspace-specific *snip class list*, which is returned by `get-the-snip-class-list`. When a snip is encoded, the snip's class name is associated with the encoding; when the snip needs to be decoded, then the snip class list is searched by name to find the snip's class. The snip class will then provide a decoding function that can create a new snip from the encoding.

If a snip class's name is of the form `"((lib ...) (lib ...))"`, then the snip class implementation can be loaded on demand. The name is parsed using `read`; if the result has the form `((lib` *string* `...) (lib` *string* `...))`, then the first element used with `dynamic-require` along with `'snip-class`. If the `dynamic-require` result is a `snip-class%` object, then it is inserted into the current eventspace's snip class list, and loading or saving continues using the new class.

The second `lib` form in `"((lib ...) (lib ...))"` supplies a reader for a text-only version of the snip. See `wxme` for more information.

A snip class's name can also be just `"(lib ...)"`, which is used like the first part of the two-`lib` form. However, this form provides no information for the text-only `wxme` reader.

Editor Data

While a snip belongs to an editor, the editor may store extra information about a snip in some specialized way. When the snip is to be encoded, this extra information needs to be put into an editor data object so that the extra information can be encoded as well. In a text editor, extra information can be associated with ranges of items, as well as snips.

Just as a snip must be associated with a snip class to be decoded (see §1.3.2.1.1 "Snip Classes"), an editor data object needs an editor data class for decoding. Every editor data class object can be added to the eventspace-specific *editor data class list*, returned by `get-the-editor-data-class-list`. Alternatively, like snip classes (see §1.3.2.1.1 "Snip Classes"), editor data class names can use the form `"((lib ...) (lib ...))"` to enable on-demand loading. The corresponding module should export an `editor-data-class%` object named `'editor-data-class`.

To store and load information about a snip or region in an editor:

- derive new classes from `editor-data%` and `editor-data-class%`.
- derive a new class from the `text%` or `pasteboard%` class, and override the `get-`

snip-data and set-snip-data methods and/or the get-region-data and set-region-data methods.

Note: the get-region-data and set-region-data methods are called for cut-and-paste encoding, but not for file-saving encoding; see §1.3.2.2 "Global Data: Headers and Footers" for information on extending the file format.

Global Data: Headers and Footers

The editor file format provides for adding extra global data in special header and footer sections. To save and load special header and/or footer records:

- Pick a name for each header/footer record. This name should not conflict with any other header/footer record name in use, and no one else should use these names. All names beginning with "wx" are reserved for internal use. By tagging extra header and footer records with a unique name, the file can be safely loaded under a system that does not support the records.
- Derive a new class from the text% or pasteboard% class, and override the write-headers-to-file, write-footers-to-file, read-header-from-file and/or read-footer-from-file methods.

When an editor is saved, the methods write-headers-to-file and write-footers-to-file are invoked; at this time, the derived text% or pasteboard% object has a chance to save records. To write a header/footer record, first invoke the begin-write-header-footer-to-file method, at which point the record name is provided. Once the record is written, call end-write-header-footer-to-file.

When an editor is loaded and a header/footer record is encountered, the read-header-from-file or read-footer-from-file method is invoked, with the record name as the argument. If the name matches a known record type, then the data can be loaded.

See also write-headers-to-file and read-header-from-file.

1.3.3 End of Line Ambiguity

Because an editor can force a line break even when there is no carriage return item, a position alone does not always specify a location for the caret. Consider the last position of a line that is soft-broken (i.e., no carriage return is present): there is no item between the last item of the line and the first item of the next line, so two locations (one end-of-line and one start-of-line) map to the same position.

For this reason, position-setting and position-getting methods often have an extra argument. In the case of a position-setting method, the argument specifies whether the caret should be

drawn at the left or right side of the page (in the event that the location is doubly defined); `#t` means that the caret should be drawn on the right side. Similarly, methods which calculate a position from a location will take an extra boxed boolean; the box is filled with `#t` if the position is ambiguous and it came from a right-side location, or `#f` otherwise.

1.3.4 Flattened Text

In plain text editors, there is a simple correlation between positions and characters. In an `editor<%>` object, this is not true much of the time, but it is still sometimes useful to just "get the text" of an editor.

Text can be extracted from an editor in either of two forms:

- *Simple text*, where there is one character per item. Items that are characters are mapped to themselves, and all other items are mapped to a period. Line breaks are represented by carriage-return characters (ASCII 13).
- *Flattened text*, where each item can map to an arbitrary string. Items that are characters are still mapped to themselves, but more complicated items can be represented with a useful string determined by the item's snip. Newlines are mapped to platform-specific character sequences (linefeed under X, carriage return under Mac OS X, and linefeed-carriage return under Windows). This form is called "flattened" because the editor's items have been reduced to a linear sequence of characters.

1.3.5 Caret Ownership

Within a frame, only one object can contain the keyboard focus. This property must be maintained when a frame contains multiple editors in multiple displays, and when a single editor contains other editors as items.

When an editor has the keyboard focus, it will usually display the current selection or a line indicating the insertion point; the line is called the *caret*.

When an editor contains other editors, it keeps track of caret ownership among the contained sub-editors. When the caret is taken away from the main editor, it will revoke caret ownership from the appropriate sub-editor.

When an editor or snip is drawn, an argument to the drawing method specifies whether the caret should be drawn with the data. This argument can be any of (in increasing order):

- `'no-caret` — The caret should not be drawn at all.
- `'show-inactive-caret` — The caret should be drawn as inactive; items may be identified as the local current selection, but the keyboard focus is elsewhere.

- 'show-caret — The caret should be drawn to show keyboard focus ownership.

The 'show-inactive-caret display mode is useful for showing selection ranges in text editors that do not have the focus. This 'show-inactive-caret mode is distinct from 'no-caret mode; when editors are embedded, only the locally-active editor shows its selection.

1.3.6 Cut and Paste Time Stamps

Methods of editor<%> that use the clipboard — including copy, cut, paste, and do-edit-operation — consume a time stamp argument. This time stamp is generally extracted from the mouse-event% or key-event% object that triggered the clipboard action. X uses the time stamp to synchronize clipboard operations among the clipboard clients.

All instances of event% include a time stamp, which can be obtained using get-time-stamp.

If the time stamp is 0, it defaults to the current time. Using 0 as the time stamp almost always works fine, but it is considered bad manners under X.

1.3.7 Clickbacks

Clickbacks in a text% editor facilitate the creation of simple interactive objects, such as hypertext. A clickback is defined by associating a callback function with a range of items in the editor. When a user clicks on the items in that range, the callback function is invoked. For example, a hypertext clickback would associate a range to a callback function that changes the selection range in the editor.

By default, the callback function is invoked when the user releases the mouse button. The set-clickback method accepts an optional argument that causes the callback function to be invoked on the button press, instead. This behavior is useful, for example, for a clickback that creates a popup menu.

Note that there is no attempt to save clickback information when a file is saved, since a clickback will have an arbitrary procedure associated with it.

1.3.8 Internal Editor Locks

Instances of editor<%> have three levels of internal locking:

- write locking — When an editor is internally locked for writing, the abstract content of the editor cannot be changed (e.g., insertion attempts fail silently). However, snips

in a text editor can still be split and merged, and the text editor can be changed in ways that affect the flow of lines. The `locked-for-write?` method reports whether an editor is currently locked for writing.

- flow locking — When a text editor is internally locked for reflowing, it is locked for writing, the snip content of the editor cannot change, the location of a snip cannot be computed if it is not already known (see `locations-computed?` in `editor<%>`), and the editor cannot be drawn to a display. A request for uncomputed location information during a flow lock produces undefined results. The `locked-for-flow?` method reports whether an editor is currently locked for flowing.
- read locking — When an editor is internally locked for reading, no operations can be performed on the editor (e.g., a request for the current selection position returns an undefined value). This extreme state is used only during callbacks to its snips for setting the snip's administrator, splitting the snip, or merging snips. The `locked-for-read?` method reports whether an editor is currently locked for reading.

The internal lock for an editor is *not* affected by calls to `lock`.

Methods that report location-independent information about an editor never trigger a lock. A method that reports location information may trigger a flow lock or write lock if the relevant information has not been computed since the last modification to the editor (see `locations-computed?` in `editor<%>`). A method that modifies the editor in any way, even setting the selection position, can trigger a read lock, flow lock, or write lock.

1.3.9 Editors and Threads

An editor is not tied to any particular thread or eventspace, except to the degree that it is displayed in a canvas (which has an eventspace). Concurrent access of an editor is always safe, in the sense that the editor will not become corrupted. However, because editor access can trigger locks, and because lock-rejected operations tend to fail silently, concurrent access can produce unexpected results.

Nevertheless, the editor supports certain concurrent patterns reliably. One relevant pattern is updating an editor in one thread while the editor is displayed in a canvas that is managed by a different (handler) thread. To ensure that canvas refreshes are not performed while the editor is locked for flowing, and to ensure that refreshes do not prevent editor modifications, the following are guaranteed:

- When an editor's `refresh` method is called during an edit sequence (which is started by `begin-edit-sequence` and ended with `end-edit-sequence`), the requested refresh region is recorded, but the refresh is not performed. Instead, the refresh is delayed until the end of the edit sequence.
- Attempting to start an edit sequence while a refresh is in progress blocks until the refresh is complete.

- The `on-display-size-when-ready` method calls `on-display-size` only when the editor is not being refreshed and only when an edit sequence is not in progress. In the first case, the `on-display-size` call is delegated to the refreshing thread to be called after the refresh completes. In the second case, the `on-display-size` call is delegated to the edit-sequence thread, to be called when the edit sequence is complete.

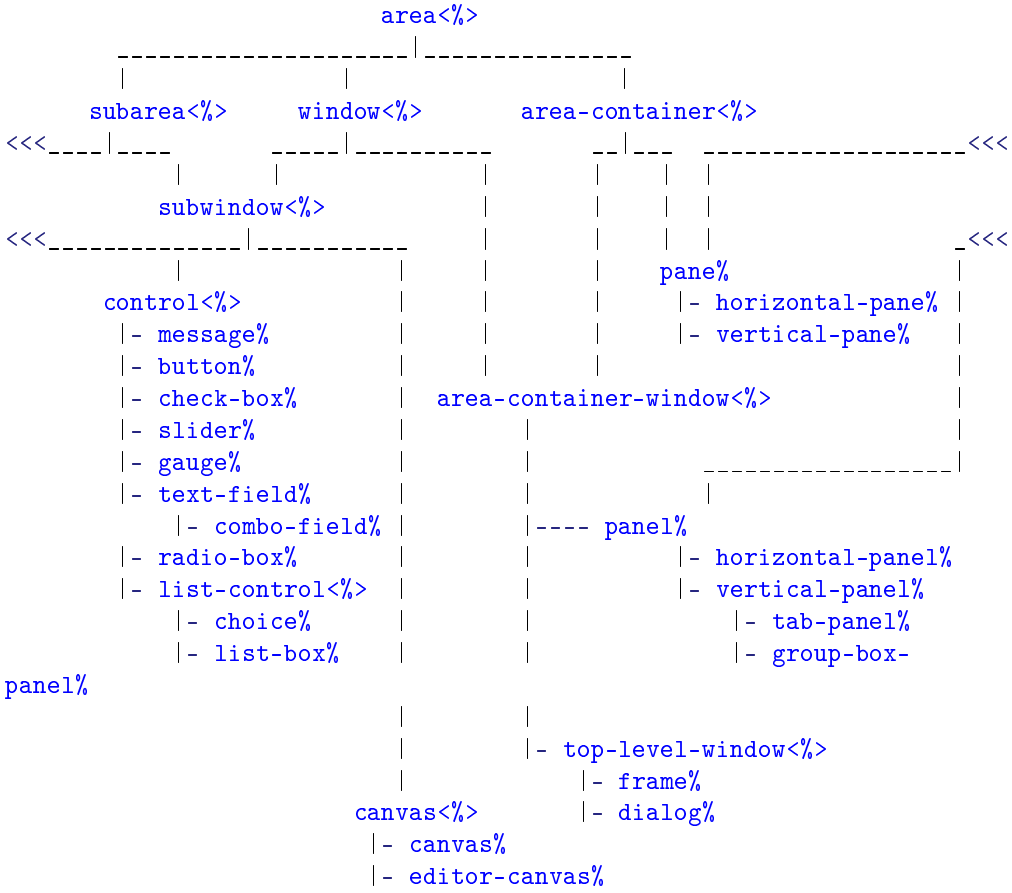
Thus, disabling an `editor-canvas%` object (using `enable`) is sufficient to ensure that a background thread can modify an editor displayed by the canvas, as long as all modifications are in edit sequences. The background modifications will impair canvas refreshes minimally and temporarily, and refreshes will not impair modifications in the background thread.

A second supported pattern is reading an editor in a background thread while the editor may be manipulated in other threads. Since no location-independent reads introduce locks, the such reads in the background thread will not impair other threads. However, other threads may interfere with the background thread, causing it to receive erroneous or out-of-date content information. This one-sided guarantee is useful if the background thread's work can be discarded when the editor is modified.

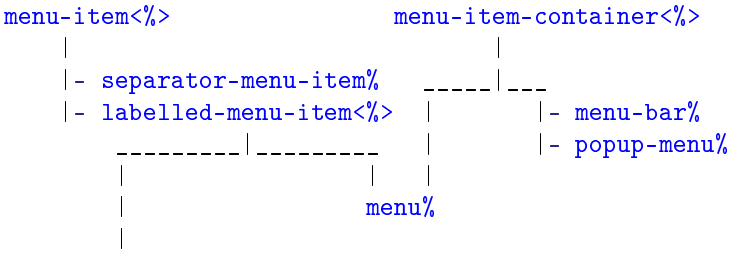
2 Reference

2.1 Windowing Classes

Windows and controls:



Menus:



```
    |- selectable-menu-item<%>
        |- menu-item%
        |- checkable-menu-item%
```

Events and other:

```
event%                                  timer%
 |- key-event%                         cursor%
 |- mouse-event%
 |- scroll-event%                      clipboard<%>
 |- control-event%                     clipboard-client%
```

Alphabetical:

2.1.1 area<%>

---

area<%> : interface?

An area<%> object is either a window or a windowless container for managing the position and size of other areas. An area<%> can be a container, a containee, or both. The only areas without a parent are top-level windows.

All area<%> classes accept the following named instantiation arguments:

- min-width — default is the initial graphical minimum width; passed to min-width
- min-height — default is the initial graphical minimum height; passed to min-height
- stretchable-width — default is class-specific; passed to stretchable-width
- stretchable-height — default is class-specific; passed to stretchable-height

---

```
(send an-area get-graphical-min-size)
 → (integer-in 0 10000) (integer-in 0 10000)
```

Returns the area's graphical minimum size as two values: the minimum width and the minimum height (in pixels).

See §1.1.2 "Geometry Management" for more information. Note that the return value *does not* depend on the area's min-width and min-height settings.

```
(send an-area get-parent)
```

```
 → (or/c (is-a?/c area-container<%>) false/c)
```

Returns the area's parent. A top-level window may have no parent (in which case `#f` is returned), or it may have another top-level window as its parent.

```
(send an-area get-top-level-window)
 → (or/c (is-a?/c frame%) (is-a?/c dialog))
```

Returns the area's closest frame or dialog ancestor. For a frame or dialog area, the frame or dialog itself is returned.

```
(send an-area min-width) → (integer-in 0 10000)
(send an-area min-width w) → void?
  w : (integer-in 0 10000)
```

Gets or sets the area's minimum width (in pixels) for geometry management.

The minimum width is ignored when it is smaller than the area's graphical minimum width, or when it is smaller than the width reported by `container-size` if the area is a container. See §1.1.2 "Geometry Management" for more information.

An area's initial minimum width is its graphical minimum width. See also `get-graphical-min-size` .

When setting the minimum width, if *w* is smaller than the internal hard minimum, an `exn:fail:contract` exception is raised.

```
(send an-area min-height) → (integer-in 0 10000)
(send an-area min-height h) → void?
  h : (integer-in 0 10000)
```

Gets or sets the area's minimum height for geometry management.

The minimum height is ignored when it is smaller than the area's graphical minimum height, or when it is smaller than the height reported by `container-size` if the area is a container. See §1.1.2 "Geometry Management" for more information.

An area's initial minimum height is its graphical minimum height. See also `get-graphical-min-size` .

When setting the minimum height (in pixels); if *h* is smaller than the internal hard minimum, an `exn:fail:contract` exception is raised.

```
(send an-area stretchable-height) → boolean?
```

```
(send an-area stretchable-height stretch?) → void?
  stretch? : any/c
```

Gets or sets the area's vertical stretchability for geometry management. See §1.1.2 "Geometry Management" for more information.

```
(send an-area stretchable-width) → boolean?
(send an-area stretchable-width stretch?) → void?
  stretch? : any/c
```

Gets or sets the area's horizontal stretchability for geometry management. See §1.1.2 "Geometry Management" for more information.

2.1.2 area-container<%>

```
area-container<%> : interface?
  implements: area<%>
```

An area-container<%> is a container area<%>.

All area-container<%> classes accept the following named instantiation arguments:

- border — default is 0; passed to border
- spacing — default is 0; passed to spacing
- alignment — default is class-specific, such as '(center top) for vertical-panel%; the list elements are passed to set-alignment

---

```
(send an-area-container add-child child) → void?
  child : (is-a?/c subwindow<%>)
```

Add the given subwindow to the set of non-deleted children. See also change-children.

```
(send an-area-container after-new-child child) → void?
  child : (is-a?/c subarea<%>)
```

Specification: This method is called after a new containee area is created with this area as its container. The new child is provided as an argument to the method.

Default implementation: Does nothing.

```
(send an-area-container begin-container-sequence) → void?
```

Suspends geometry management in the container's top-level window until `end-container-sequence` is called. The `begin-container-sequence` and `end-container-sequence` methods are used to bracket a set of container modifications so that the resulting geometry is computed only once. A container sequence also delays show and hide actions by `change-children`, as well as the on-screen part of showing via `show` until the sequence is complete. Sequence begin and end commands may be nested arbitrarily deep.

```
(send an-area-container border) → (integer-in 0 1000)
(send an-area-container border margin) → void?
  margin : (integer-in 0 1000)
```

Gets or sets the border margin for the container in pixels. This margin is used as an inset into the panel's client area before the locations and sizes of the subareas are computed.

```
(send an-area-container change-children filter) → void?
  filter : ((listof (is-a?/c subarea<%>))
            . -> . (listof (is-a?/c subarea<%>)))
```

Takes a filter procedure and changes the container's list of non-deleted children. The filter procedure takes a list of children areas and returns a new list of children areas. The new list must consist of children that were created as subareas of this area (i.e., `change-children` cannot be used to change the parent of a subarea).

After the set of non-deleted children is changed, the container computes the sets of newly deleted and newly non-deleted children. Newly deleted windows are hidden. Newly non-deleted windows are shown.

Since non-window areas cannot be hidden, non-window areas cannot be deleted. If the filter procedure removes non-window subareas, an exception is raised and the set of non-deleted children is not changed.

```
(send an-area-container container-flow-modified) → void?
```

Call this method when the result changes for an overridden flow-defining method, such as `place-children`. The call notifies the geometry manager that the placement of the container's children needs to be recomputed.

The `reflow-container`method only recomputes child positions when the geometry manager thinks that the placement has changed since the last computation.

```
(send an-area-container container-size info)
```

```
→ (integer-in 0 10000) (integer-in 0 10000)
 info : (listof (list/c (integer-in 0 10000)
                        (integer-in 0 10000)
                        any/c
                        any/c))
```

Called to determine the minimum size of a container. See §1.1.2 “Geometry Management” for more information.

```
(send an-area-container delete-child child) → void?
  child : (is-a?/c subwindow<%>)
```

Removes the given subwindow from the list of non-deleted children. See also change-children.

```
(send an-area-container end-container-sequence) → void?
```

See begin-container-sequence.

```
(send an-area-container get-alignment)
 → (symbols/c right center left)
   (symbols/c bottom center top)
```

Returns the container’s current alignment specification. See set-alignment for more information.

```
(send an-area-container get-children)
 → (listof (is-a?/c subarea<%>))
```

Returns a list of the container’s non-deleted children. (The non-deleted children are the ones currently managed by the container; deleted children are generally hidden.) The order of the children in the list is significant. For example, in a vertical panel, the first child in the list is placed at the top of the panel.

```
(send an-area-container place-children info
                                       width
                                       height)
 → (listof (list/c (integer-in 0 10000)
                   (integer-in 0 10000)
                   (integer-in 0 10000)
                   (integer-in 0 10000)))
```

```
  info : (listof (list/c (integer-in 0 10000)
                         (integer-in 0 10000)
                         any/c
                         any/c))
  width : (integer-in 0 10000)
  height : (integer-in 0 10000)
```

Called to place the children of a container. See §1.1.2 “Geometry Management” for more information.

```
(send an-area-container reflow-container) → void?
```

When a container window is not shown, changes to the container’s set of children do not necessarily trigger the immediate re-computation of the container’s size and its children’s sizes and positions. Instead, the recalculation is delayed until the container is shown, which avoids redundant computations between a series of changes. The `reflow-container` method forces the immediate recalculation of the container’s and its children’s sizes and locations.

Immediately after calling the `reflow-container` method, `get-size`, `get-client-size`, `get-width`, `get-height`, `get-x`, and `get-y` report the manager-applied sizes and locations for the container and its children, even when the container is hidden. A container implementation can call functions such as `get-size` at any time to obtain the current state of a window (because the functions do not trigger geometry management).

See also `container-flow-modified`.

```
(send an-area-container set-alignment horiz-align
                                    vert-align) → void?
  horiz-align : (symbols/c right center left)
  vert-align : (symbols/c bottom center top)
```

Sets the alignment specification for a container, which determines how it positions its children when the container has leftover space (when a child was not stretchable in a particular dimension).

When the container’s horizontal alignment is `'left`, the children are left-aligned in the container and whitespace is inserted to the right. When the container’s horizontal alignment is `'center`, each child is horizontally centered in the container. When the container’s horizontal alignment is `'right`, leftover whitespace is inserted to the left.

Similarly, a container’s vertical alignment can be `'top`, `'center`, or `'bottom`.

```
(send an-area-container spacing) → (integer-in 0 1000)
(send an-area-container spacing spacing) → void?
  spacing : (integer-in 0 1000)
```

Gets or sets the spacing, in pixels, used between subareas in the container. For example, a vertical panel inserts this spacing between each pair of vertically aligned subareas (with no extra space at the top or bottom).

2.1.3 area-container-window<%>

```
area-container-window<%> : interface?
  implements: area-container<%>
              window<%>
```

Combines two interfaces.

2.1.4 button%

```
button% : class?
  superclass: object%
  extends: control<%>
```

Whenever a button is clicked by the user, the button's callback procedure is invoked. A callback procedure is provided as an initialization argument when each button is created.

```
(new button%  [label label]
              [parent parent]
             [[callback callback]
              [style style]
              [font font]
              [enabled enabled]
              [vert-margin vert-margin]
              [horiz-margin horiz-margin]
              [min-width min-width]
              [min-height min-height]
              [stretchable-width stretchable-width]
              [stretchable-height stretchable-height]])
 → (is-a?/c button%)
  label : (or/c label-string? (is-a?/c bitmap%))
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c button%) (is-a?/c control-event%) . -> . any)
           = (lambda (b e) (void))
  style : (one-of/c 'border 'deleted) = null
  font : (is-a?/c font%) = normal-control-font
```

```
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #f
  stretchable-height : any/c = #f
```

Creates a button with a string or bitmap label. If *label* is a bitmap, then the bitmap must be valid (see ok? in bitmap%) and not installed in a bitmap-dc% object; otherwise, an exn:fail:contract exception is raised. If the bitmap has a mask (see get-loaded-mask in bitmap%) that is the same size as the bitmap, then the mask is used for the label; furthermore, in contrast to the limitations of draw-bitmap in dc<%>, non-monochrome label masks work consistently on all platforms.

If & occurs in *label* (when *label* is a string), it is specially parsed; under Windows and X, the character following & is underlined in the displayed control to indicate a keyboard mnemonic. (Under Mac OS X, mnemonic underlines are not shown.) The underlined mnemonic character must be a letter or a digit. The user can effectively click the button by typing the mnemonic when the control's top-level-window contains the keyboard focus. The user must also hold down the Meta or Alt key if the keyboard focus is currently in a control that handles normal alphanumeric input. The & itself is removed from *label* before it is displayed for the control; a && in *label* is converted to & (with no mnemonic underlining). Under Mac OS X, a parenthesized mnemonic character is removed (along with any surrounding space) before the label is displayed, since a parenthesized mnemonic is often used for non-Roman languages. Finally, any text after a tab character is removed on all platforms. Mnemonic keyboard events are handled by on-traverse-char (but not under Mac OS X).

The *callback* procedure is called (with the event type 'button) whenever the user clicks the button.

If *style* includes 'border, the button is drawn with a special border that indicates to the user that it is the default action button (see on-traverse-char). If *style* includes 'deleted, then the button is created as hidden, and it does not affect its parent's geometry; the button can be made active later by calling *parent*'s add-child method.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-button set-label label) → void?
  label : (or/c label-string? (is-a?/c bitmap%))
```

Overrides set-label in window<%>.

The same as set-label in window<%> when label is a string.

Otherwise, sets the bitmap label for a bitmap button. Since label is a bitmap, the bitmap must be valid (see ok? in bitmap%) and not installed in a bitmap-dc% object; otherwise, an exn:fail:contract exception is raised. If the bitmap has a mask (see get-loaded-mask in bitmap%) that is the same size as the bitmap, then the mask is used for the label; furthermore, in contrast to the limitations of draw-bitmap in dc<%>, non-monochrome label masks work consistently on all platforms. The bitmap label is installed only if the control was originally created with a bitmap label.

2.1.5 canvas<%>

---

canvas<%> : interface?
  implements: subwindow<%>

A canvas is a subwindow onto which graphics and text can be drawn. Canvases also receive mouse and keyboard events.

To draw onto a canvas, get its device context (see get-dc).

The canvas<%> interface is implemented by two classes:

- canvas% — a canvas for arbitrary drawing and event handling
- editor-canvas% — a canvas for displaying editor<%> objects

(send a-canvas accept-tab-focus) → boolean?
(send a-canvas accept-tab-focus on?) → void?
  on? : any/c

Gets or sets whether tab-focus is enabled for the canvas (assuming that the canvas is not created with the 'no-focus style for canvas%). When tab-focus is enabled, the canvas can receive the keyboard focus when the user navigates among a frame or dialog's controls with the Tab and arrow keys. By default, tab-focus is disabled.

When tab-focus is enabled for a canvas% object, Tab, arrow, Enter, and Escape keyboard events are consumed by a frame's default on-traverse-char method. (In addition, a dialog's default method consumes Escape key events.) Otherwise, on-traverse-char allows the keyboard events to be propagated to the canvas.

For an editor-canvas% object, handling of Tab, arrow, Enter, and Escape keyboard events is determined by the allow-tab-exit method.

```
(send a-canvas get-canvas-background)
 → (or/c (is-a?/c color%) false/c)
```

Returns the color currently used to "erase" the canvas content before `on-paint` is called. See also `set-canvas-background`.

The result is `#f` if the canvas was created with the `'transparent` style, otherwise it is always a `color%` object.

```
(send a-canvas get-dc) → (is-a?/c dc<%>)
```

Gets the canvas's device context. See `dc<%>` for more information about drawing.

```
(send a-canvas min-client-height) → (integer-in 0 10000)
(send a-canvas min-client-height h) → void?
  h : (integer-in 0 10000)
```

Gets or sets the canvas's minimum height for geometry management, based on the client size rather than the full size. The client height is obtained or changed via `min-height` in `area<%>`, adding or subtracting border and scrollbar sizes as appropriate.

The minimum height is ignored when it is smaller than the canvas's graphical minimum height. See §1.1.2 "Geometry Management" for more information.

```
(send a-canvas min-client-width) → (integer-in 0 10000)
(send a-canvas min-client-width w) → void?
  w : (integer-in 0 10000)
```

Gets or sets the canvas's minimum width for geometry management, based on the canvas's client size rather than its full size. The client width is obtained or changed via `min-width` in `area<%>`, adding or subtracting border and scrollbar sizes as appropriate.

The minimum width is ignored when it is smaller than the canvas's graphical minimum width. See §1.1.2 "Geometry Management" for more information.

```
(send a-canvas on-char ch) → void?
  ch : (is-a?/c key-event%)
```

Specification: Called when the canvas receives a keyboard event. See also §1.1.3 "Mouse and Keyboard Events".

Default implementation: Does nothing.

```
(send a-canvas on-event event) → void?
  event : (is-a?/c mouse-event%)
```

Specification: Called when the canvas receives a mouse event. See also §1.1.3 “Mouse and Keyboard Events”, noting in particular that certain mouse events can get dropped.

Default implementation: Does nothing.

```
(send a-canvas on-paint) → void?
```

Specification: Called when the canvas is exposed or resized so that the image in the canvas can be repainted.

When `on-paint` is called in response to a system expose event and only a portion of the canvas is newly exposed, any drawing operations performed by `on-paint` are clipped to the newly-exposed region; however, the clipping region as reported by `get-clipping-region` does not change.

Default implementation: Does nothing.

```
(send a-canvas on-tab-in) → void?
```

Specification: Called when the keyboard focus enters the canvas via keyboard navigation events. The `on-focus` method is also called, as usual for a focus change. When the keyboard focus leaves a canvas due to a navigation event, only `on-focus` is called.

See also `accept-tab-focus` and `on-traverse-char` in `top-level-window<%>` .

Default implementation: Does nothing.

```
(send a-canvas set-canvas-background color) → void?
  color : (is-a?/c color%)
```

Sets the color used to “erase” the canvas content before `on-paint` is called. (This color is typically associated with the canvas at a low level, so that it is used even when a complete refresh of the canvas is delayed by other activity.)

If the canvas was created with the `'transparent` style, an `exn:fail:contract` exception is raised.

```
(send a-canvas set-resize-corner on?) → void?
  on? : any/c
```

Under Mac OS X, enables or disables space for a resize tab at the canvas’s lower-right corner

when only one scrollbar is visible. This method has no effect under Windows or X, and it has no effect when both or no scrollbars are visible. The resize corner is disabled by default, but it can be enabled when a canvas is created with the 'resize-corner style.

```
(send a-canvas warp-pointer x y) → void?
  x : (integer-in 0 10000)
  y : (integer-in 0 10000)
```

Moves the cursor to the given location on the canvas.

2.1.6 canvas%

```
canvas% : class?
  superclass: object%
  extends: canvas<%>
```

A canvas% object is a general-purpose window for drawing and handling events.

```
(new canvas% [parent parent]
             [[style style]
              [paint-callback paint-callback]
              [label label]
              [gl-config gl-config]
              [enabled enabled]
              [vert-margin vert-margin]
              [horiz-margin horiz-margin]
              [min-width min-width]
              [min-height min-height]
              [stretchable-width stretchable-width]
              [stretchable-height stretchable-height]])
 → (is-a?/c canvas%)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  style : (listof (one-of/c 'border 'control-border 'combo
                            'vscroll 'hscroll 'resize-corner
                            'gl 'no-autoclear 'transparent
                            'no-focus 'deleted))
        = null
  paint-callback : ((is-a?/c canvas%) (is-a?/c dc<%>) . -> . any)
                 = void
  label : (or/c label-string? false/c) = #f
  gl-config : (or/c (is-a?/c gl-config%) false/c) = #f
```

```
enabled : any/c = #t
vert-margin : (integer-in 0 1000) = 0
horiz-margin : (integer-in 0 1000) = 0
min-width : (integer-in 0 10000) = graphical-minimum-width
min-height : (integer-in 0 10000) = graphical-minimum-height
stretchable-width : any/c = #t
stretchable-height : any/c = #t
```

The *style* argument indicates one or more of the following styles:

- 'border — gives the canvas a thin border
- 'control-border — gives the canvas a border that is like a text-field% control
- 'combo — gives the canvas a combo button that is like a combo-field% control; this style is intended for use with 'control-border and not with 'hscroll or 'vscroll
- 'hscroll — enables horizontal scrolling (initially visible but inactive)
- 'vscroll — enables vertical scrolling (initially visible but inactive)
- 'resize-corner — leaves room for a resize control at the canvas's bottom right when only one scrollbar is visible
- 'gl — *obsolete* (every canvas is an OpenGL context where supported)
- 'no-autoclear — prevents automatic erasing of the canvas before calls to on-paint
- 'transparent — the canvas is automatically "erased" before an update using it's parent window's background; the result is undefined if this flag is combined with 'no-autoclear
- 'no-focus — prevents the canvas from accepting the keyboard focus when the canvas is clicked, or when the focus method is called
- 'deleted — creates the canvas as initially hidden and without affecting *parent*'s geometry; the canvas can be made active later by calling *parent*'s add-child method

The 'hscroll and 'vscroll styles create a canvas with an initially inactive scrollbar. The scrollbars are activated with either init-manual-scrollbars or init-auto-scrollbars, and they can be hidden and re-shown with show-scrollbars.

The *paint-callback* argument is called by the default on-paint method, using the canvas and the DC returned by get-dc as the argument.

The *label* argument names the canvas for get-label, but it is not displayed with the canvas.

The *gl-config* argument determines properties of an OpenGL context for this canvas, as obtained through the canvas's drawing context. See also get-dc and get-gl-context in dc<%>.

For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-canvas get-scroll-page which)
 → (integer-in 1 1000000000)
  which : (one-of/c 'horizontal 'vertical)
```

Get the current page step size of a manual scrollbar. The result is 0 if the scrollbar is not active or it is automatic.

The *which* argument is either 'horizontal or 'vertical, indicating whether to get the page step size of the horizontal or vertical scrollbar, respectively.

See also init-manual-scrollbars.

```
(send a-canvas get-scroll-pos which)
 → (integer-in 0 1000000000)
  which : (one-of/c 'horizontal 'vertical)
```

Gets the current value of a manual scrollbar. The result is always 0 if the scrollbar is not active or it is automatic.

The *which* argument is either 'horizontal or 'vertical, indicating that the value of the horizontal or vertical scrollbar should be returned, respectively.

See also init-manual-scrollbars.

```
(send a-canvas get-scroll-range which)
 → (integer-in 0 1000000000)
  which : (one-of/c 'horizontal 'vertical)
```

Gets the current maximum value of a manual scrollbar. The result is always 0 if the scrollbar is not active or it is automatic.

The *which* argument is either 'horizontal or 'vertical, indicating whether to get the maximum value of the horizontal or vertical scrollbar, respectively.

See also init-manual-scrollbars.

```
(send a-canvas get-view-start)
 → (integer-in 0 10000) (integer-in 0 10000)
```

Get the location at which the visible portion of the canvas starts, based on the current values of the horizontal and vertical scrollbars if they are initialized as automatic (see `init-auto-scrollbars`). Combined with `get-client-size`, an application can efficiently redraw only the visible portion of the canvas. The values are in pixels.

If the scrollbars are disabled or initialized as manual (see `init-manual-scrollbars`), the result is `(values 0 0)`.

```
(send a-canvas get-virtual-size)
 → (value (integer-in 0 10000) (integer-in 0 10000))
```

Gets the size in device units of the scrollable canvas area (as opposed to the client size, which is the area of the canvas currently visible). This is the same size as the client size (as returned by `get-client-size`) unless scrollbars are initialized as automatic (see `init-auto-scrollbars`).

```
(send a-canvas init-auto-scrollbars horiz-pixels
                                    vert-pixels
                                    h-value
                                    v-value)     → void?
  horiz-pixels : (or/c (integer-in 1 1000000000) false/c)
  vert-pixels : (or/c (integer-in 1 1000000000) false/c)
  h-value : (real-in 0.0 1.0)
  v-value : (real-in 0.0 1.0)
```

Enables and initializes automatic scrollbars for the canvas. A horizontal or vertical scrollbar can be activated only in a canvas that was created with the `'hscroll` or `'vscroll` style flag, respectively.

With automatic scrollbars, the programmer specifies the desired virtual size of the canvas, and the scrollbars are automatically handled to allow the user to scroll around the virtual area. The scrollbars are not automatically hidden if they are unneeded; see `show-scrollbars`.

See also `init-manual-scrollbars` for information about manual scrollbars. The horizontal and vertical scrollbars are always either both manual or both automatic, but they are independently enabled. Automatic scrollbars can be re-initialized as manual, and vice versa.

If either *horiz-pixels* or *vert-pixels* is `#f`, the scrollbar is not enabled in the corresponding direction, and the canvas's virtual size in that direction is the same as its client size.

The *h-value* and *v-value* arguments specify the initial values of the scrollbars as a frac-

tion of the scrollbar's range. A 0.0 value initializes the scrollbar to its left/top, while a 1.0 value initializes the scrollbar to its right/bottom.

See also on-scroll and get-virtual-size.

```
(send a-canvas init-manual-scrollbars h-length
                                      v-length
                                      h-page
                                      v-page
                                      h-value
                                      v-value) → void?
  h-length : (or/c (integer-in 0 1000000000) false/c)
  v-length : (or/c (integer-in 0 1000000000) false/c)
  h-page : (integer-in 1 1000000000)
  v-page : (integer-in 1 1000000000)
  h-value : (integer-in 0 1000000000)
  v-value : (integer-in 0 1000000000)
```

Enables and initializes manual scrollbars for the canvas. A horizontal or vertical scrollbar can be activated only in a canvas that was created with the 'hscroll or 'vscroll style flag, respectively.

With manual scrollbars, the programmer is responsible for managing all details of the scrollbars, and the scrollbar state has no effect on the canvas's virtual size. Instead, the canvas's virtual size is the same as its client size.

See also init-auto-scrollbars for information about automatic scrollbars. The horizontal and vertical scrollbars are always either both manual or both automatic, but they are independently enabled. Automatic scrollbars can be re-initialized as manual, and vice versa.

The *h-length* and *v-length* arguments specify the length of each scrollbar in scroll steps (i.e., the maximum value of each scrollbar). If either is #f, the scrollbar is disabled in the corresponding direction.

The *h-page* and *v-page* arguments set the number of scrollbar steps in a page, i.e., the amount moved when pressing above or below the value indicator in the scrollbar control.

The *h-value* and *v-value* arguments specify the initial values of the scrollbars.

If *h-value* is greater than *h-length* or *v-value* is greater than *v-length*, an exn:fail:contract exception is raised. (The page step may be larger than the total size of a scrollbar.)

See also on-scroll and get-virtual-size.

```
(send a-canvas on-paint) → void?
```

Overrides on-paint in canvas<%>.

Calls the procedure supplied as the paint-callback argument when the canvas% was created.

```
(send a-canvas on-scroll event) → void?
  event : (is-a?/c scroll-event%)
```

Called when the user changes one of the canvas's scrollbars. A scroll-event% argument provides information about the scroll action.

This method is called only when manual scrollbars are changed, not automatic scrollbars; for automatic scrollbars, the on-paint method is called, instead.

```
(send a-canvas scroll h-value v-value) → void?
  h-value : (or/c (real-in 0.0 1.0) false/c)
  v-value : (or/c (real-in 0.0 1.0) false/c)
```

Sets the values of automatic scrollbars. (This method has no effect on manual scrollbars.)

If either argument is #f, the scrollbar value is not changed in the corresponding direction.

The *h-value* and *v-value* arguments each specify a fraction of the scrollbar's movement. A 0.0 value sets the scrollbar to its left/top, while a 1.0 value sets the scrollbar to its right/bottom. A 0.5 value sets the scrollbar to its middle. In general, if the canvas's virtual size is v, its client size is c, and (> v c), then scrolling to p sets the view start to (floor (* p (- v c))).

See also init-auto-scrollbars and get-view-start.

```
(send a-canvas set-scroll-page which value) → void?
  which : (one-of/c 'horizontal 'vertical)
  value : (integer-in 1 1000000000)
```

Set the current page step size of a manual scrollbar. (This method has no effect on automatic scrollbars.)

The *which* argument is either 'horizontal or 'vertical, indicating whether to set the page step size of the horizontal or vertical scrollbar, respectively.

See also init-manual-scrollbars.

```
(send a-canvas set-scroll-pos which value) → void?
  which : (one-of/c 'horizontal 'vertical)
  value : (integer-in 0 1000000000)
```

Sets the current value of a manual scrollbar. (This method has no effect on automatic scrollbars.)

The *which* argument is either 'horizontal or 'vertical, indicating whether to set the value of the horizontal or vertical scrollbar set, respectively.

The value of the canvas's scrollbar can be changed by the user scrolling, and such changes do not go through this method; use on-scroll to monitor scrollbar value changes.

See also init-manual-scrollbars and scroll.

```
(send a-canvas set-scroll-range which
                                value) → void?
  which : (one-of/c 'horizontal 'vertical)
  value : (integer-in 0 1000000000)
```

Sets the current maximum value of a manual scrollbar. (This method has no effect on automatic scrollbars.)

The *which* argument is either 'horizontal or 'vertical, indicating whether to set the maximum value of the horizontal or vertical scrollbar, respectively.

See also init-manual-scrollbars.

```
(send a-canvas show-scrollbars show-horiz?
                               show-vert?) → void?
  show-horiz? : any/c
  show-vert? : any/c
```

Shows or hides the scrollbars as indicated by *show-horiz?* and *show-vert?*. If *show-horiz?* is true and the canvas was not created with the 'hscroll style, an exn:fail:contract exception is raised. Similarly, if *show-vert?* is true and the canvas was not created with the 'vscroll style, an exn:fail:contract exception is raised.

The horizontal scrollbar can be shown only if the canvas was created with the 'hscroll style, and the vertical scrollbar can be shown only if the canvas was created with the 'vscroll style. See also init-auto-scrollbars and init-manual-scrollbars.

```
(send a-canvas swap-gl-buffers) → void?
```

Calls swap-buffers on the result of get-gl-context for this canvas's DC as returned by get-dc.

The swap-buffers method acquires a re-entrant lock, so nested calls to with-gl-context on different threads or OpenGL contexts can block or deadlock.

```
(send a-canvas with-gl-context thunk) → any
  thunk : (-> any)
```

Passes the given thunk to `call-as-current` of the result of `get-gl-context` for this canvas's DC as returned by `get-dc`.

The `call-as-current` method acquires a re-entrant lock, so nested calls to `with-gl-context` on different threads or OpenGL contexts can block or deadlock.

2.1.7 check-box%

```
check-box% : class?
  superclass: object%
  extends: control<%>
```

A check box is a labeled box which is either checked or unchecked.

Whenever a check box is clicked by the user, the check box's value is toggled and its callback procedure is invoked. A callback procedure is provided as an initialization argument when each check box is created.

```
(new check-box%  [label label]
                 [parent parent]
                [[callback callback]
                 [style style]
                 [value value]
                 [font font]
                 [enabled enabled]
                 [vert-margin vert-margin]
                 [horiz-margin horiz-margin]
                 [min-width min-width]
                 [min-height min-height]
                 [stretchable-width stretchable-width]
                 [stretchable-height stretchable-height]])
 → (is-a?/c check-box%)
  label : (or/c label-string? (is-a?/c bitmap%))
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c check-box%) (is-a?/c control-event%) . -> . any)
           = (lambda (c e) (void))
  style : (listof (one-of/c 'deleted)) = null
  value : any/c = #f
```

```
  font : (is-a?/c font%) = normal-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #f
  stretchable-height : any/c = #f
```

Creates a check box with a string or bitmap label. If *label* is a bitmap, then the bitmap must be valid (see ok? in bitmap%) and not installed in a bitmap-dc% object; otherwise, an exn:fail:contract exception is raised. If the bitmap has a mask (see get-loaded-mask in bitmap%) that is the same size as the bitmap, then the mask is used for the label; furthermore, in contrast to the limitations of draw-bitmap in dc<%>, non-monochrome label masks work consistently on all platforms.

If & occurs in *label* (when *label* is a string), it is specially parsed as for button%.

The *callback* procedure is called (with the event type 'check-box) whenever the user clicks the check box.

If *style* includes 'deleted, then the check box is created as hidden, and it does not affect its parent's geometry; the check box can be made active later by calling *parent*'s add-child method.

If *value* is true, it is passed to set-value so that the box is initially checked.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-check-box get-value) → boolean?
```

Gets the state of the check box: #t if it is checked, #f otherwise.

```
(send a-check-box set-label label) → void?
  label : (or/c label-string? (is-a?/c bitmap%))
```

Overrides set-label in window<%>.

The same as set-label in window<%> when *label* is a string.

Otherwise, sets the bitmap label for a bitmap check box. Since *label* is a bitmap, the bitmap must be valid (see ok? in bitmap%) and not installed in a bitmap-dc% object; otherwise, an exn:fail:contract exception is raised. If the bitmap has a mask (see get-loaded-

mask in bitmap%) that is the same size as the bitmap, then the mask is used for the label; furthermore, in contrast to the limitations of draw-bitmap in dc<%>, non-monochrome label masks work consistently on all platforms. The bitmap label is installed only if the control was originally created with a bitmap label.

```
(send a-check-box set-value state) → void?
  state : any/c
```

Sets the check box's state. (The control's callback procedure is *not* invoked.)

The check box's state can be changed by the user clicking the control, and such changes do not go through this method; use the control callback procedure (provided as an initialization argument) to monitor state changes.

If *state* is #f, the box is unchecked, otherwise it is checked.

2.1.8 checkable-menu-item%

```
checkable-menu-item% : class?
  superclass: object%
  extends: selectable-menu-item<%>
```

A checkable-menu-item% is a string-labelled menu item that maintains a check mark. Its parent must be a menu% or popup-menu%. When the user selects the menu item, the item's check mark is toggled and its callback procedure is called.

```
(new checkable-menu-item% [label label]
                          [parent parent]
                         [[callback callback]
                          [shortcut shortcut]
                          [help-string help-string]
                          [demand-callback demand-callback]
                          [checked checked]
                          [shortcut-prefix shortcut-prefix]])
 → (is-a?/c checkable-menu-item%)
  label : label-string?
  parent : (or/c (is-a?/c menu% popup-menu%))
  callback : ((is-a?/c checkable-menu-item%) (is-a?/c control-event%) . -> . any)
           = (lambda (i e) (void))
  shortcut : (or/c char? false/c) = #f
  help-string : (or/c label-string? false/c) = #f
  demand-callback : ((is-a?/c menu-item%) . -> . any)
                  = (lambda (i) (void))
```

```
  checked : any/c = #f
  shortcut-prefix : (listof (one-of/c 'alt 'cmd 'meta 'ctl
                                      'shift 'option))
                  = (get-default-shortcut-prefix)
```

Creates a new menu item in *parent*. The item is initially shown, appended to the end of its parent, and unchecked. The *callback* procedure is called (with the event type 'menu) when the menu item is selected (either via a menu bar, popup-menu in window<%>, or popup-menu in editor-admin%).

See set-label for information about mnemonic &s in *label*.

If *shortcut* is not #f, the item has a shortcut. See get-shortcut for more information. The *shortcut-prefix* argument determines the set of modifier keys for the shortcut; see get-shortcut-prefix.

If help is not #f, the item has a help string. See get-help-string for more information.

The *demand-callback* procedure is called by the default on-demand method with the object itself.

By default, the menu item is initially unchecked. If *checked* is true, then check is called so that the menu item is initially checked.

```
(send a-checkable-menu-item check check?) → void?
  check? : any/c
```

Checks or unchecks the menu item.

A menu item's check state can be changed by the user selecting the item, and such changes do not go through this method; use the menu item callback procedure (provided as an initialization argument) to monitor check state changes.

```
(send a-checkable-menu-item is-checked?) → boolean?
```

Returns #t if the item is checked, #f otherwise.

2.1.9 choice%

```
choice% : class?
  superclass: object%
  extends: list-control<%>
```

A choice item allows the user to select one string item from a pop-up list of items. Unlike a

list box, only the currently selection is visible until the user pops-up the menu of choices.

Whenever the selection of a choice item is changed by the user, the choice item's callback procedure is invoked. A callback procedure is provided as an initialization argument when each choice item is created.

See also list-box%.

```
(new choice%  [label label]
              [choices choices]
              [parent parent]
             [[callback callback]
              [style style]
              [selection selection]
              [font font]
              [enabled enabled]
              [vert-margin vert-margin]
              [horiz-margin horiz-margin]
              [min-width min-width]
              [min-height min-height]
              [stretchable-width stretchable-width]
              [stretchable-height stretchable-height]])
 → (is-a?/c choice%)
  label : (or/c label-string? false/c)
  choices : (listof label-string?)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c choice%) (is-a?/c control-event%) . -> . any)
           = (lambda (c e) (void))
  style : (listof (one-of/c 'horizontal-label 'vertical-label
                            'deleted))
        = null
  selection : exact-nonnegative-integer? = 0
  font : (is-a?/c font%) = normal-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #f
  stretchable-height : any/c = #f
```

Creates a choice item. If *label* is a string, it is used as the label for the choice item.

If & occurs in *label*, it is specially parsed as for button%.

The *choices* list specifies the initial list of user-selectable items for the control. The initial set of choices determines the control's minimum graphical width (see §1.1.2 "Geometry Management" for more information).

The *callback* procedure is called (with the event type 'choice) when the user selects a choice item (or re-selects the currently selected item).

If *style* includes 'vertical-label, then the choice item is created with a label above the control; if *style* does not include 'vertical-label (and optionally includes 'horizontal-label), then the label is created to the left of the choice item. If *style* includes 'deleted, then the choice item is created as hidden, and it does not affect its parent's geometry; the choice item can be made active later by calling *parent*'s add-child method.

By default, the first choice (if any) is initially selected. If *selection* is positive, it is passed to set-selection to set the initial choice selection. Although *selection* normally must be less than the length of *choices*, it can be 0 when *choices* is empty.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

2.1.10 clipboard-client%

---

clipboard-client% : class?
  superclass: object%

A clipboard-client% object allows a program to take over the clipboard and service requests for clipboard data. See clipboard<%> for more information.

A clipboard-client% object is associated to an eventspace when it becomes the current client; see set-clipboard-client for more information.

(new clipboard-client%) → (is-a?/c clipboard-client%)

Creates a clipboard client that supports no data formats.

(send a-*clipboard-client* add-type *format*) → void?
  format : string?

Adds a new data format name to the list supported by the clipboard client.

The *format* string is typically four capital letters. (Under Mac OS X, only four characters

for *format* are ever used.) For example, "TEXT" is the name of the UTF-8-encoded string format. New format names can be used to communicate application- and platform-specific data formats.

```
(send a-clipboard-client get-data format)
 → (or/c bytes? string? false/c)
  format : string?
```

Called when a process requests clipboard data while this client is the current one for the clipboard. The requested format is passed to the method, and the result should be a byte string matching the requested format, or #f if the request cannot be fulfilled.

Only data format names in the client's list will be passed to this method; see add-type.

When this method is called by the clipboard, the current eventspace is the same as the client's eventspace. If, at the point of the clipboard request, the current eventspace is not the client's eventspace, then current thread is guaranteed to be the handler thread of the client's eventspace.

```
(send a-clipboard-client get-types) → (listof string?)
```

Returns a list of names that are the data formats supported by the clipboard client.

```
(send a-clipboard-client on-replaced) → void?
```

Called when a clipboard client is dismissed as the clipboard owner (because the clipboard has be taken by another client or by an external application).

2.1.11 clipboard<%>

```
clipboard<%> : interface?
```

A single clipboard<%> object, the-clipboard, manages the content of the system-wide clipboard for cut and paste.

Under X, a second clipboard<%> object, the-x-selection-clipboard, manages the content of the system-wide X selection. If the 'GRacket:selectionAsClipboard preference preference (see §3.1 "Preferences") is set to a non-zero true value, however, then the-clipboard is always the same as the-x-selection-clipboard, and the system-wide X clipboard is not used.

Under Windows and Mac OS X, the-x-selection-clipboard is always the same as the-clipboard.

Data can be entered into a clipboard in one of two ways: by setting the current clipboard string or byte string, or by installing a clipboard-client% object. When a client is installed, requests for clipboard data are directed to the client.

Generic data is always retrieved from the clipboard as a byte string. When retrieving clipboard data, a data type string specifies the format of the data string. The availability of different clipboard formats is determined by the current clipboard owner.

```
(send a-clipboard get-clipboard-bitmap time)
 → (or/c (is-a?/c bitmap%) false/c)
  time : exact-integer?
```

Gets the current clipboard contents as a bitmap (Windows, Mac OS X), returning #f if the clipboard does not contain a bitmap.

See get-clipboard-data for information on eventspaces and the current clipboard client.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-clipboard get-clipboard-data format
                                     time)
 → (or/c bytes? string? false/c)
  format : string
  time : exact-integer?
```

Gets the current clipboard contents in a specific format, returning #f if the clipboard does not contain data in the requested format.

If the clipboard client is associated to an eventspace that is not the current one, the data is retrieved through a callback event in the client’s eventspace. If no result is available within one second, the request is abandoned and #f is returned.

See add-type in clipboard-client% for information on *format*.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-clipboard get-clipboard-string time)
 → (or/c string false/c)
  time : exact-integer?
```

Gets the current clipboard contents as simple text, returning #f if the clipboard does not contain any text.

See get-clipboard-data for information on eventspaces and the current clipboard client.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-clipboard same-clipboard-client? owner) → boolean?
  owner : (is-a?/c clipboard-client%)
```

Returns #t if *owner* currently owns the clipboard, #f otherwise.

```
(send a-clipboard set-clipboard-bitmap new-bitmap
                                       time)      → void?
  new-bitmap : (is-a?/c bitmap%)
  time : exact-integer?
```

Changes the current clipboard contents to *new-bitmap* (Windows, Mac OS X) and releases the current clipboard client (if any).

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-clipboard set-clipboard-client new-owner
                                       time)     → void?
  new-owner : (is-a?/c clipboard-client%)
  time : exact-integer?
```

Changes the clipboard-owning client: sets the client to *new-owner* and associates *new-owner* with the current eventspace (as determined by current-eventspace). The eventspace association is removed when the client is no longer the current one.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-clipboard set-clipboard-string new-text
                                       time)    → void?
  new-text : string
  time : exact-integer?
```

Changes the current clipboard contents to *new-text*, and releases the current clipboard client (if any).

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

2.1.12 combo-field%

```
combo-field% : class?
  superclass: text-field%
```

A combo-field% object is a text-field% object that also resembles a choice% object, because it has a small popup button to the right of the text field. By default, clicking the button pops up a menu, and selecting a menu item copies the item into the text field.

```
(new combo-field%  [label label]
                   [choices choices]
                   [parent parent]
                  [[callback callback]
                   [init-value init-value]
                   [style style]
                   [font font]
                   [enabled enabled]
                   [vert-margin vert-margin]
                   [horiz-margin horiz-margin]
                   [min-width min-width]
                   [min-height min-height]
                   [stretchable-width stretchable-width]
                   [stretchable-height stretchable-height]])
 → (is-a?/c combo-field%)
  label : (or/c label-string? false/c)
  choices : (listof label-string?)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c combo-field%) (is-a?/c control-event%) . -> . any)
           = (lambda (c e) (void))
  init-value : string = ""
  style : (listof (one-of/c 'horizontal-label 'vertical-label
                            'deleted))
        = null
  font : (is-a?/c font%) = normal-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
  stretchable-height : any/c = #f
```

If label is not #f, it is used as the combo label. Otherwise, the combo does not display its

label.

If & occurs in *label*, it is specially parsed as for button%.

The *choices* list specifies the initial list of items for the combo's popup menu. The append method adds a new item to the menu with a callback to install the appended item into the combo's text field. The get-menu method returns the combo's menu to allow arbitrary other operations. This menu might not be used at all if on-popup is overridden.

The *callback* procedure is called when the user changes the text in the combo or presses the Enter key (and Enter is not handled by the combo's frame or dialog; see on-traverse-char in top-level-window<%>). If the user presses Enter, the type of event passed to the callback is 'text-field-enter, otherwise it is 'text-field.

If *init-value* is not "", the minimum width of the text item is made wide enough to show *init-value*. Otherwise, a built-in default width is selected.

If *style* includes 'vertical-label, then the combo is created with a label above the control; if *style* does not include 'vertical-label (and optionally includes 'horizontal-label), then the label is created to the left of the combo. If *style* includes 'deleted, then the combo is created as hidden, and it does not affect its parent's geometry; the combo can be made active later by calling *parent*'s add-child method..

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-combo-field append l) → void?
  l : label-string?
```

Adds a new item to the combo's popup menu. The given label is used for the item's name, and the item's callback installs the label into the combo's text field.

```
(send a-combo-field get-menu) → (is-a?/c popup-menu%)
```

Returns the popup-menu% that is used by the default on-popup method. This menu is initialized with the labels argument when the combo-field% is created, and the append method adds a new item to the menu.

```
(send a-combo-field on-popup event) → void?
  event : (is-a?/c control-event%)
```

Specification: Called when the user clicks the combo's popup button.

Default implementation: Gets a menu from get-menu, sets its minimum width to match the combo control's width, and then pops up the menu.

2.1.13 control<%>

---

control<%> : interface?
  implements: subwindow<%>

The control<%> interface is implemented by the built-in control window classes:

- message%
- button%
- check-box%
- slider%
- gauge%
- text-field%
- radio-box%
- choice%
- list-box%

(send *a-control* command *event*) → void?
  event : (is-a?/c control-event%)

Calls the control's callback function, passing on the given control-event% object.

(send *a-control* get-font) → (is-a?/c font%)

Returns the font used for the control, which is optionally supplied when a control is created.

2.1.14 control-event%

---

control-event% : class?
  superclass: event%

A control-event% object contains information about a control event. An instance of control-event% is always provided to a control or menu item callback procedure.

```
(new control-event%  [event-type event-type]
                     [[time-stamp time-stamp]])
 → (is-a?/c control-event%)
  event-type : (one-of/c 'button 'check-box 'choice
                         'list-box 'list-box-dclick 'text-field
                         'text-field-enter 'slider 'radio-box
                         'menu-popdown 'menu-popdown-none 'tab-panel)
  time-stamp : exact-integer? = 0
```

The *event-type* argument is one of the following:

- 'button — for button% clicks
- 'check-box — for check-box% toggles
- 'choice — for choice% item selections
- 'list-box — for list-box% selections and deselections
- 'list-box-dclick — for list-box% double-clicks
- 'text-field — for text-field% changes
- 'text-field-enter — for single-line text-field% Enter event
- 'menu — for selectable-menu-item<%> callbacks
- 'slider — for slider% changes
- 'radio-box — for radio-box% selection changes
- 'menu-popdown — for popup-menu% callbacks (item selected)
- 'menu-popdown-none — for popup-menu% callbacks (no item selected)
- 'tab-panel — for tab-panel% tab changes

This value is extracted out of a control-event% object with the get-event-type method.

See get-time-stamp for information about *time-stamp*.

```
(send a-control-event get-event-type)
 → (one-of/c 'button 'check-box 'choice
             'list-box 'list-box-dclick 'text-field
             'text-field-enter 'slider 'radio-box
             'menu-popdown 'menu-popdown-none 'tab-panel)
```

Returns the type of the control event. See control-event% for information about each event type symbol.

```
(send a-control-event set-event-type type) → void?
  type : (one-of/c 'button 'check-box 'choice
                   'list-box 'list-box-dclick 'text-field
                   'text-field-enter 'slider 'radio-box
                   'menu-popdown 'menu-popdown-none 'tab-panel)
```

Sets the type of the event. See control-event% for information about each event type symbol.

2.1.15 cursor%

```
cursor% : class?
  superclass: object%
```

A cursor is a small icon that indicates the location of the mouse pointer. The bitmap image typically indicates the current mode or meaning of a mouse click at its current location.

A cursor is assigned to each window (or the window may use its parent's cursor; see set-cursor for more information), and the pointer image is changed to match the window's cursor when the pointer is moved over the window. Each cursor object may be assigned to many windows.

```
(make-object cursor% image
                     mask
                    [hot-spot-x
                     hot-spot-y]) → (is-a?/c cursor%)
  image : (is-a?/c bitmap%)
  mask : (is-a?/c bitmap%)
  hot-spot-x : (integer-in 0 15) = 0
  hot-spot-y : (integer-in 0 15) = 0
(make-object cursor% id) → (is-a?/c cursor%)
  id : (one-of/c 'arrow 'bullseye 'cross 'hand 'ibeam 'watch 'blank
                 'size-n/s 'size-e/w 'size-ne/sw 'size-nw/se)
```

The first case creates a cursor using an image bitmap and a mask bitmap. Both bitmaps must have depth 1 and size 16 by 16 pixels. The *hot-spot-x* and *hot-spot-y* arguments determine the focus point of the cursor within the cursor image, relative to its top-left corner.

The second case creates a cursor using a stock cursor, specified as one of the following:

- 'arrow — the default cursor
- 'bullseye — concentric circles
- 'cross — a crosshair
- 'hand — an open hand
- 'ibeam — a vertical line, indicating that clicks control a text-selection caret
- 'watch — a watch or hourglass, indicating that the user must wait for a computation to complete
- 'arrow+watch — the default cursor with a watch or hourglass, indicating that some computation is in progress, but the cursor can still be used
- 'blank — invisible
- 'size-e/w — arrows left and right
- 'size-n/s — arrows up and down
- 'size-ne/sw — arrows up-right and down-left
- 'size-nw/se — arrows up-left and down-right

If the cursor is created successfully, ok? returns #t, otherwise the cursor object cannot be assigned to a window.

(send *a-cursor* ok?) → boolean?

Returns #t if the cursor is can be assigned to a window, #f otherwise.

2.1.16 dialog%

---

dialog% : class?
  superclass: object%
  extends: top-level-window<%>

A dialog is a top-level window that is *modal*: while the dialog is shown, all other top-level windows in the dialog's eventspace are disabled.

```
(new dialog% [label label]
             [[parent parent]
              [width width]
              [height height]
              [x x]
              [y y]
              [style style]
              [enabled enabled]
              [border border]
              [spacing spacing]
              [alignment alignment]
              [min-width min-width]
              [min-height min-height]
              [stretchable-width stretchable-width]
              [stretchable-height stretchable-height]])
 → (is-a?/c dialog%)
  label : label-string?
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  width : (or/c (integer-in 0 10000) false/c) = #f
  height : (or/c (integer-in 0 10000) false/c) = #f
  x : (or/c (integer-in 0 10000) false/c) = #f
  y : (or/c (integer-in 0 10000) false/c) = #f
  style : (listof (one-of/c 'no-caption 'resize-border 'no-sheet))
        = null
  enabled : any/c = #t
  border : (integer-in 0 1000) = 0
  spacing : (integer-in 0 1000) = 0
  alignment : (list/c (one-of/c 'left 'center 'right)
                      (one-of/c 'top 'center 'bottom))
            = '(center top)
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
  stretchable-height : any/c = #t
```

The *label* string is used as the dialog's title in its title bar. If the dialog's label is changed (see set-label), the title bar is updated.

The *parent* argument can be #f or an existing frame. Under Windows, if *parent* is an existing frame, the new dialog is always on top of its parent. Under Windows and X, a dialog is iconized when its parent is iconized.

If *parent* is #f, then the eventspace for the new dialog is the current eventspace, as determined by current-eventspace. Otherwise, *parent*'s eventspace is the new dialog's

eventspace.

If the *width* or *height* argument is not #f, it specifies an initial size for the dialog (in pixels) assuming that it is larger than the minimum size, otherwise the minimum size is used. Under Windows and Mac OS X (and with some X window managers) dialogs are not resizeable.

If the *x* or *y* argument is not #f, it specifies an initial location for the dialog. Otherwise, if no location is set before the dialog is shown, it is centered (with respect *parent* if not #f, the screen otherwise).

The *style* flags adjust the appearance of the dialog on some platforms:

- 'no-caption — omits the title bar for the dialog (Windows)
- 'resize-border — adds a resizeable border around the window (Windows) or grow box in the bottom right corner (Mac OS X)
- 'no-sheet — uses a movable window for the dialog, even if a parent window is provided (Mac OS X)

Even if the dialog is not shown, a few notification events may be queued for the dialog on creation. Consequently, the new dialog's resources (e.g., memory) cannot be reclaimed until some events are handled, or the dialog's eventspace is shut down.

For information about the *enabled* argument, see window<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-dialog on-subwindow-char receiver
                                 event)    → boolean?
  receiver : (is-a?/c window<%>)
  event : (is-a?/c key-event%)
```

Overrides on-subwindow-char in window<%>.

Returns the result of

```
(or (send this on-system-menu-char event)
    (send this on-traverse-char event))
```

```
(send a-dialog show show?) → void?
  show? : any/c
```

Overrides show in top-level-window<%>.

If *show?* is true, the dialog is shown and all frames (and other dialogs) in the eventspace become disabled until the dialog is closed. If *show?* is false, the dialog is hidden and other frames and dialogs are re-enabled (unless a different, pre-existing dialog is still shown).

If *show?* is true, the method does not immediately return. Instead, it loops with `yield` until the dialog is found to be hidden between calls to `yield`. An internal semaphore is used with `yield` to avoid a busy-wait, and to ensure that the `show` method returns as soon as possible after the dialog is hidden.

2.1.17 event%

`event%` : `class?`
  superclass: `object%`

An `event%` object contains information about a control, keyboard, mouse, or scroll event. See also `control-event%`, `key-event%`, `mouse-event%`, and `scroll-event%`.

`(new event% [[time-stamp time-stamp]]) → (is-a?/c event%)`
  time-stamp : `exact-integer?` = `0`

See `get-time-stamp` for information about *time-stamp*.

`(send an-event get-time-stamp) → exact-integer?`

Returns the time, in milliseconds, when the event occurred. This time is compatible with times reported by Racket's `current-milliseconds` procedure.

`(send an-event set-time-stamp time) → void?`
  time : `exact-integer?`

Set the time, in milliseconds, when the event occurred. See also Racket's `current-milliseconds`.

If the supplied value is outside the platform-specific range of time values, an `exn:fail:contract` exception is raised.

2.1.18 frame%

`frame%` : `class?`
  superclass: `object%`

extends: top-level-window<%>

A frame is a top-level container window. It has a title bar (which displays the frame's label), an optional menu bar, and an optional status line.

Under Windows, both Multiple Document Interface (MDI) and Single Document Interface (SDI) frames are supported.

```
(new frame%  [label label]
             [[parent parent]
              [width width]
              [height height]
              [x x]
              [y y]
              [style style]
              [enabled enabled]
              [border border]
              [spacing spacing]
              [alignment alignment]
              [min-width min-width]
              [min-height min-height]
              [stretchable-width stretchable-width]
              [stretchable-height stretchable-height]])
 → (is-a?/c frame%)
  label : label-string?
  parent : (or/c (is-a?/c frame%) false/c) = #f
  width : (or/c (integer-in 0 10000) false/c) = #f
  height : (or/c (integer-in 0 10000) false/c) = #f
  x : (or/c (integer-in -10000 10000) false/c) = #f
  y : (or/c (integer-in -10000 10000) false/c) = #f
  style : (listof (one-of/c 'no-resize-border 'no-caption
                            'no-system-menu 'hide-menu-bar
                            'mdi-parent 'mdi-child
                            'toolbar-button 'float 'metal))
        = null
  enabled : any/c = #t
  border : (integer-in 0 1000) = 0
  spacing : (integer-in 0 1000) = 0
  alignment : (list/c (one-of/c 'left 'center 'right)
                      (one-of/c 'top 'center 'bottom))
            = '(center top)
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
  stretchable-height : any/c = #t
```

The *label* string is displayed in the frame's title bar. If the frame's label is changed (see set-label), the title bar is updated.

The *parent* argument can be #f or an existing frame. Under Windows, if *parent* is an existing frame, the new frame is always on top of its parent. Also, the *parent* frame may be an MDI parent frame from a new MDI child frame. Under Windows and X (for many window managers), a frame is iconized when its parent is iconized.

If *parent* is #f, then the eventspace for the new frame is the current eventspace, as determined by current-eventspace. Otherwise, *parent*'s eventspace is the new frame's eventspace.

If the *width* or *height* argument is not #f, it specifies an initial size for the frame (in pixels) assuming that it is larger than the minimum size, otherwise the minimum size is used.

If the *x* or *y* argument is not #f, it specifies an initial location for the frame. Otherwise, a location is selected automatically (tiling frames and dialogs as they are created).

The *style* flags adjust the appearance of the frame on some platforms:

- 'no-resize-border — omits the resizeable border around the window (Windows, X MWM) or grow box in the bottom right corner (Mac OS X)
- 'no-caption — omits the title bar for the frame (Windows, X MWM)

  (X Gnome, X KDE: the frame decoration is omitted completely when 'no-resize-border and 'no-caption are combined.)
- 'no-system-menu — omits the system menu (Windows)
- 'mdi-child — creates the frame as a MDI (multiple document interface) child frame, mutually exclusive with 'mdi-parent (Windows)
- 'mdi-parent — creates the frame as a MDI (multiple document interface) parent frame, mutually exclusive with 'mdi-child (Windows)
- 'toolbar-button — includes a toolbar button on the frame's title bar (Mac OS X); a click on the toolbar button triggers a call to on-toolbar-button-click
- 'hide-menu-bar — hides the menu bar and dock when the frame is active (Mac OS X)
- 'float — causes the frame to stay in front of all other non-floating windows (Windows and Mac OS X always, X when combined with 'no-caption); under Mac OS X, a floating frame shares the focus with an active non-floating frame; when this style is combined with 'no-caption, then showing the frame does not cause the keyboard focus to shift to the window, and under X, clicking the frame does not move the focus; under Windows, a floating frame has no taskbar button

- 'metal — draws the frame with a brushed-metal background (Mac OS X); this style is ignored when 'no-caption is specified

If the 'mdi-child style is specified, the *parent* must be a frame with the 'mdi-parent style, otherwise an exn:fail:contract exception is raised.

Even if the frame is not shown, a few notification events may be queued for the frame on creation. Consequently, the new frame's resources (e.g., memory) cannot be reclaimed until some events are handled, or the frame's eventspace is shut down.

For information about the *enabled* argument, see window<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

(send *a-frame* create-status-line) → void?

Creates a status line at the bottom of the frame. The width of the status line is the whole width of the frame (adjusted automatically when resizing), and the height and text size are platform-specific.

See also set-status-text.

(send *a-frame* get-menu-bar)
 → (or/c (is-a?/c menu-bar%) false/c)

Returns the frame's menu bar, or #f if none has been created for the frame.

(send *a-frame* has-status-line?) → boolean?

Returns #t if the frame's status line has been created, #f otherwise. See also create-status-line.

(send *a-frame* iconize *iconize?*) → void?
  iconize? : any/c

Iconizes (minimizes) or deiconizes (restores) the frame. Deiconizing brings the frame to the front.

A frame's iconization can be changed by the user, and such changes do not go through this method. A program cannot detect when a frame has been iconized except by polling is-iconized?.

(send *a-frame* is-iconized?) → boolean?

Returns `#t` if the frame is iconized (minimized), `#f` otherwise.

```
(send a-frame is-maximized?) → boolean?
```

Under Windows and Mac OS X, returns `#t` if the frame is maximized, `#f` otherwise. Under X, the result is always `#f`.

```
(send a-frame maximize maximize?) → void?
  maximize? : any/c
```

Specification: Maximizes or restores the frame under Windows and Mac OS X; the frame's show state is not affected. Under Windows, an iconized frame cannot be maximized or restored.

A window's maximization can be changed by the user, and such changes do not go through this method; use `on-size` to monitor size changes.

Default implementation: If *maximize?* is `#f`, the window is restored, otherwise it is maximized.

```
(send a-frame modified) → boolean?
(send a-frame modified modified?) → void?
  modified? : any/c
```

Gets or sets the frame's modification state as reflected to the user. Under Mac OS X, the modification state is reflected as a dot in the frame's close button. Under Windows and X, the modification state is reflected by an asterisk at the end of the frame's displayed title.

```
(send a-frame on-mdi-activate active?) → void?
  active? : any/c
```

Called under Windows when a MDI-child frame becomes the active frame within its parent (in which case the argument is `#t`), or when the child frame ceases to be the active frame (in which case the argument is `#f`).

MDI activation is different from keyboard-focus activation. If the parent frame is the frontmost top-level frame, so that the MDI child gets or loses the keyboard focus, then a separate `on-activate` notification is sent to the MDI-child frame.

```
(send a-frame on-menu-char event) → boolean?
  event : (is-a?/c key-event%)
```

If the frame has a menu bar with keyboard shortcuts, and if the key event includes a Control, Alt, Option, Meta, Command, Shift, or Function key, then `on-menu-char` attempts to match

the given event to a menu item. If a match is found, #t is returned, otherwise #f is returned.

When the match corresponds to a complete shortcut combination, the menu item's callback is called (before on-menu-char returns).

If the event does not correspond to a complete shortcut combination, the event may be handled anyway if it corresponds to a mnemonic in the menu bar (i.e., an underlined letter in a menu's title, which is installed by including an ampersand in the menu's label). If a mnemonic match is found, the keyboard focus is moved to the menu bar (selecting the menu with the mnemonic), and #t is returned.

```
(send a-frame on-subwindow-char receiver
                                event)   → boolean?
  receiver : (is-a?/c window<%>)
  event : (is-a?/c key-event%)
```

Overrides on-subwindow-char in window<%>.

Returns the result of

```
(or (send this on-menu-char event)
    (send this on-system-menu-char event)
    (send this on-traverse-char event))
```

```
(send a-frame on-toolbar-button-click) → void?
```

Under Mac OS X, called when the user clicks the toolbar button on a frame created with the 'toolbar-button style.

```
(send a-frame set-icon icon [mask which]) → void?
  icon : (is-a?/c bitmap%)
  mask : (is-a?/c bitmap%) = #f
  which : (one-of/c 'small 'large 'both) = 'both
```

Sets the large or small icon bitmap for this frame. Future changes to the bitmap do not affect the frame's icon.

The icon is used in a platform-specific way:

- Windows — the small icon is used for the frame's icon (in the top-left) and in the task bar, and the large icon is used for the Alt-Tab task switcher.
- Mac OS X — both icons are ignored.
- X — many window managers use the small icon in the same way as Windows, and others use the small icon when iconifying the frame; the large icon is ignored.

The bitmap for either icon can be any size, but most platforms scale the small bitmap to 16 by 16 pixels and the large bitmap to 32 by 32 pixels.

If a mask bitmap is not provided, then the entire (rectangular) bitmap is used as an icon.

If a mask bitmap is provided, the mask must be monochrome. In the mask bitmap, use black pixels to indicate the icon's region and use white pixels outside the icon's region. In the icon bitmap, use black pixels for the region outside the icon.

```
(send a-frame set-status-text text) → void?
  text : string
```

Sets the frame's status line text and redraws the status line. See also `create-status-line`.

2.1.19 gauge%

```
gauge% : class?
  superclass: object%
  extends: control<%>
```

A gauge is a horizontal or vertical bar for displaying the output value of a bounded integer quantity. Each gauge has an adjustable range, and the gauge's current value is always between 0 and its range, inclusive. Use `set-value` to set the value of the gauge.

```
(new gauge%  [label label]
             [range range]
             [parent parent]
            [[style style]
             [font font]
             [enabled enabled]
             [vert-margin vert-margin]
             [horiz-margin horiz-margin]
             [min-width min-width]
             [min-height min-height]
             [stretchable-width stretchable-width]
             [stretchable-height stretchable-height]])
 → (is-a?/c gauge%)
  label : (or/c label-string? false/c)
  range : (integer-in 1 10000)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
```

```
style : (listof (one-of/c 'horizontal 'vertical
                          'vertical-label 'horizontal-label
                          'deleted))
      = '(horizontal)
font : (is-a?/c font%) = normal-control-font
enabled : any/c = #t
vert-margin : (integer-in 0 1000) = 2
horiz-margin : (integer-in 0 1000) = 2
min-width : (integer-in 0 10000) = graphical-minimum-width
min-height : (integer-in 0 10000) = graphical-minimum-height
stretchable-width : any/c = (memq 'horizontal style)
stretchable-height : any/c = (memq 'vertical style)
```

If *label* is a string, it is used as the gauge label; otherwise the gauge does not display a label.

If & occurs in *label*, it is specially parsed; under Windows and X, the character following & is underlined in the displayed control to indicate a keyboard mnemonic. (Under Mac OS X, mnemonic underlines are not shown.) The mnemonic is meaningless for a gauge (as far as on-traverse-char in top-level-window<%> is concerned), but it is supported for consistency with other control types. A programmer may assign a meaning to the mnemonic (e.g., by overriding on-traverse-char).

The *range* argument is an integer specifying the maximum value of the gauge (inclusive). The minimum gauge value is always 0.

The *style* list must include either 'horizontal, specifying a horizontal gauge, or 'vertical, specifying a vertical gauge. If *style* includes 'vertical-label, then the gauge is created with a label above the control; if *style* does not include 'vertical-label (and optionally includes 'horizontal-label), then the label is created to the left of the gauge. If *style* includes 'deleted, then the gauge is created as hidden, and it does not affect its parent's geometry; the gauge can be made active later by calling *parent*'s add-child method.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-gauge get-range) → (integer-in 1 10000)
```

Returns the range (maximum value) of the gauge.

```
(send a-gauge get-value) → (integer-in 0 10000)
```

Returns the gauge's current value.

```
(send a-gauge set-range range) → void?
  range : (integer-in 1 10000)
```

Sets the range (maximum value) of the gauge.

```
(send a-gauge set-value pos) → void?
  pos : (integer-in 0 10000)
```

Sets the gauge's current value. If the specified value is larger than the gauge's range, an `exn:fail:contract` exception is raised.

2.1.20 group-box-panel%

```
group-box-panel% : class?
  superclass: vertical-panel%
```

A group-box panel arranges its subwindows in a single column, but also draws an optional label at the top of the panel and a border around the panel content.

Unlike most panel classes, a group-box panel's horizontal and vertical margins default to 2.

```
(new group-box-panel%
  [label label]
  [parent parent]
 [[style style]
  [font font]
  [enabled enabled]
  [vert-margin vert-margin]
  [horiz-margin horiz-margin]
  [border border]
  [spacing spacing]
  [alignment alignment]
  [min-width min-width]
  [min-height min-height]
  [stretchable-width stretchable-width]
  [stretchable-height stretchable-height]])
→ (is-a?/c group-box-panel%)
 label : label-string?
 parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                (is-a?/c panel%) (is-a?/c pane%))
```

```
  style : (listof (one-of/c 'deleted)) = null
  font : (is-a?/c font%) = small-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
  border : (integer-in 0 1000) = 0
  spacing : (integer-in 0 1000) = 0
  alignment : (list/c (one-of/c 'left 'center 'right)
                      (one-of/c 'top 'center 'bottom))
            = '(center top)
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
  stretchable-height : any/c = #t
```

Creates a group pane whose title is *label*.

If *style* includes 'deleted, then the group panel is created as hidden, and it does not affect its parent's geometry; the group panel can be made active later by calling *parent*'s add-child method.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

2.1.21 grow-box-spacer-pane%

---

grow-box-spacer-pane% : class?
  superclass: pane%

A grow-box-spacer-pane% object is intended for use as a lightweight spacer in the bottom-right corner of a frame, rather than as a container. Under Mac OS X, a grow-box-spacer-pane% has the same width and height as the grow box that is inset into the bottom-right corner of a frame. Under Windows and X, a grow-box-spacer-pane% has zero width and height. Unlike all other container types, a grow-box-spacer-pane% is unstretchable by default.

```
(new grow-box-spacer-pane% ...superclass-args...)
 → (is-a?/c grow-box-spacer-pane%)
```

See pane% for information on initialization arguments.

2.1.22 horizontal-pane%

---

horizontal-pane% : class?
  superclass: pane%

A horizontal pane arranges its subwindows in a single row. See also pane%.

```
(new horizontal-pane%
  [parent parent]
 [[vert-margin vert-margin]
  [horiz-margin horiz-margin]
  [border border]
  [spacing spacing]
  [alignment alignment]
  [min-width min-width]
  [min-height min-height]
  [stretchable-width stretchable-width]
  [stretchable-height stretchable-height]])
→ (is-a?/c horizontal-pane%)
 parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                (is-a?/c panel%) (is-a?/c pane%))
 vert-margin : (integer-in 0 1000) = 0
 horiz-margin : (integer-in 0 1000) = 0
 border : (integer-in 0 1000) = 0
 spacing : (integer-in 0 1000) = 0
 alignment : (list/c (one-of/c 'left 'center 'right)
                     (one-of/c 'top 'center 'bottom))
           = '(left center)
 min-width : (integer-in 0 10000) = graphical-minimum-width
 min-height : (integer-in 0 10000) = graphical-minimum-height
 stretchable-width : any/c = #t
 stretchable-height : any/c = #t
```

For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

2.1.23 horizontal-panel%

---

horizontal-panel% : class?
  superclass: panel%

A horizontal panel arranges its subwindows in a single row. See also panel%.

```
(new horizontal-panel%
  [parent parent]
 [[style style]
  [enabled enabled]
  [vert-margin vert-margin]
  [horiz-margin horiz-margin]
  [border border]
  [spacing spacing]
  [alignment alignment]
  [min-width min-width]
  [min-height min-height]
  [stretchable-width stretchable-width]
  [stretchable-height stretchable-height]])
→ (is-a?/c horizontal-panel%)
 parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                (is-a?/c panel%) (is-a?/c pane%))
 style : (listof (one-of/c 'border 'deleted)) = null
 enabled : any/c = #t
 vert-margin : (integer-in 0 1000) = 0
 horiz-margin : (integer-in 0 1000) = 0
 border : (integer-in 0 1000) = 0
 spacing : (integer-in 0 1000) = 0
 alignment : (list/c (one-of/c 'left 'center 'right)
                     (one-of/c 'top 'center 'bottom))
           = '(left center)
 min-width : (integer-in 0 10000) = graphical-minimum-width
 min-height : (integer-in 0 10000) = graphical-minimum-height
 stretchable-width : any/c = #t
 stretchable-height : any/c = #t
```

If the 'border style is specified, the window is created with a thin border (only in this case, the client size of the panel may be less than its total size). If *style* includes 'deleted, then the panel is created as hidden, and it does not affect its parent's geometry; the panel can be made active later by calling *parent*'s add-child method.

For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-horizontal-panel set-orientation horizontal?) → void?
```

horizontal? : boolean?

Sets the orientation of the panel, switching it between the behavior of the vertical-panel% and that of the horizontal-panel%.

(send *a-horizontal-panel* get-orientation) → boolean?

Initially returns #t, but if set-orientation is called, this method returns whatever the last value passed to it was.

2.1.24 key-event%

---

key-event% : class?
  superclass: event%

A key-event% object contains information about a key press or release event. Key events are primarily processed by on-subwindow-char in window<%> and on-char in canvas<%>.

For a key-press event, a virtual key code is provided by get-key-code. For a key-release event, get-key-code reports 'release, and a virtual key code is provided by get-key-release-code.

See also §1.1.3 "Mouse and Keyboard Events".

```
(new key-event% [[key-code key-code]
                 [shift-down shift-down]
                 [control-down control-down]
                 [meta-down meta-down]
                 [alt-down alt-down]
                 [x x]
                 [y y]
                 [time-stamp time-stamp]
                 [caps-down caps-down]])
 → (is-a?/c key-event%)
  key-code : (or/c char? key-code-symbol?) = #\nul
  shift-down : any/c = #f
  control-down : any/c = #f
  meta-down : any/c = #f
  alt-down : any/c = #f
  x : exact-integer? = 0
  y : exact-integer? = 0
  time-stamp : exact-integer? = 0
```

caps-down : any/c = #f

See the corresponding get- and set- methods for information about *key-code*, *shift-down*, *control-down*, *meta-down*, *alt-down*, *x*, *y*, *time-stamp*, *caps-down*.

The release key code, as returned by get-key-release-code, is initialized to 'press.

(send *a-key-event* get-alt-down) → boolean?

Returns #t if the Option (Mac OS X) key was down for the event. When the Alt key is pressed in Windows, it is reported as a Meta press (see get-meta-down).

(send *a-key-event* get-caps-down) → boolean?

Returns #t if the Caps Lock key was on for the event.

(send *a-key-event* get-control-down) → boolean?

Returns #t if the Control key was down for the event.

Under Mac OS X, if a control-key press is combined with a mouse button click, the event is reported as a right-button click and get-control-down for the event reports #f.

(send *a-key-event* get-key-code)
 → (or/c char? key-code-symbol?)

Gets the virtual key code for the key event. The virtual key code is either a character or a special key symbol, one of the following:

- 'start
- 'cancel
- 'clear
- 'shift
- 'control
- 'menu
- 'pause
- 'capital
- 'prior

- 'next
- 'end
- 'home
- 'left
- 'up
- 'right
- 'down
- 'escape
- 'select
- 'print
- 'execute
- 'snapshot
- 'insert
- 'help
- 'numpad0
- 'numpad1
- 'numpad2
- 'numpad3
- 'numpad4
- 'numpad5
- 'numpad6
- 'numpad7
- 'numpad8
- 'numpad9
- 'numpad-enter
- 'multiply
- 'add
- 'separator

- 'subtract
- 'decimal
- 'divide
- 'f1
- 'f2
- 'f3
- 'f4
- 'f5
- 'f6
- 'f7
- 'f8
- 'f9
- 'f10
- 'f11
- 'f12
- 'f13
- 'f14
- 'f15
- 'f16
- 'f17
- 'f18
- 'f19
- 'f20
- 'f21
- 'f22
- 'f23
- 'f24
- 'numlock

- 'scroll
- 'wheel-up — mouse wheel up one notch
- 'wheel-down — mouse wheel down one notch
- 'release — indicates a key-release event
- 'press — indicates a key-press event; usually only from get-key-release-code

The special key symbols attempt to capture useful keys that have no standard ASCII representation. A few keys have standard representations that are not obvious:

- #\space — the space bar
- #\return — the Enter or Return key (on all platforms), but not necessarily the Enter key near the numpad (which is reported as 'numpad-enter if the platform distinguishes the two Enter keys)
- #\tab — the tab key
- #\backspace — the backspace key
- #\rubout — the delete key

If a suitable special key symbol or ASCII representation is not available, #\nul (the NUL character) is reported.

Under X, a 'wheel-up or 'wheel-down event may be sent to a window other than the one with the keyboard focus, because X generates wheel events based on the location of the mouse pointer.

Under Windows, when the Control key is pressed without Alt, the key code for ASCII characters is downcased, roughly cancelling the effect of the Shift key. Under Mac OS X, the key code is computed without Caps Lock effects when the Control or Command key is pressed; in the case of Control, Caps Lock is used normally if special handling is disabled for the Control key via special-control-key. Under X, the key code is computed with Caps Lock effects when the Control key is pressed without Alt.

See also get-other-shift-key-code.

```
(send a-key-event get-key-release-code)
 → (or/c char? key-code-symbol?)
```

Gets the virtual key code for a key-release event; the result is 'press for a key-press event. See get-key-code for the list of virtual key codes.

```
(send a-key-event get-meta-down) → boolean?
```

Returns `#t` if the Meta (X), Alt (Windows), or Command (Mac OS X) key was down for the event.

```
(send a-key-event get-other-altgr-key-code)
 → (or/c char? key-code-symbol? false/c)
```

See `get-other-shift-key-code`.

```
(send a-key-event get-other-caps-key-code)
 → (or/c char? key-code-symbol? false/c)
```

See `get-other-shift-key-code`.

```
(send a-key-event get-other-shift-altgr-key-code)
 → (or/c char? key-code-symbol? false/c)
```

See `get-other-shift-key-code`.

```
(send a-key-event get-other-shift-key-code)
 → (or/c char? key-code-symbol? false/c)
```

Since keyboard mappings vary, it is sometimes useful in key mappings for a program to know the result that the keyboard would have produced for an event if the Shift key had been toggled differently. The `get-other-shift-key-code` produces that other mapping, returning `#f` if the alternate mapping is unavailable, otherwise returning the same kind of result as `get-key-code`.

The `get-other-altgr-key-code` method provides the same information with respect to the AltGr key (i.e., Alt combined with Control) under Windows and X, or the Option key under Mac OS X. The `get-other-shift-altgr-key-code` method reports a mapping for in tha case that both Shift and AltGr/Option were different from the actual event.

The `get-other-shift-key-code`, `get-other-altgr-key-code`, and `get-other-shift-altgr-key-code` results all report key mappings where Caps Lock is off, independent of whether Caps Lock was on for the actual event. The `get-other-caps-key-code` method reports a mapping for in that case that the Caps Lock state was treated opposite as for the `get-key-code` result. (Caps Lock normally has either no effect or the same effect as Shift, so further combinations involving Caps Lock and other modifier keys would not normally produce further alternatives.)

Alternate mappings are not available for all events. Under Windows, alternate mappings are

reported when they produce ASCII letters, ASCII digits, and ASCII symbols. Under Mac OS X, alternate mappings are available only when the Command key is pressed. Under X, alternate mappings are usually available.

```
(send a-key-event get-shift-down) → boolean?
```

Returns `#t` if the Shift key was down for the event.

```
(send a-key-event get-x) → exact-integer?
```

Returns the x-position of the mouse at the time of the event, in the target's window's (client-area) coordinate system.

```
(send a-key-event get-y) → exact-integer?
```

Returns the y-position of the mouse at the time of the event in the target's window's (client-area) coordinate system.

```
(send a-key-event set-alt-down down?) → void?
  down? : any/c
```

Sets whether the Option (Mac OS X) key was down for the event. When the Alt key is pressed in Windows, it is reported as a Meta press (see `set-meta-down`).

```
(send a-key-event set-caps-down down?) → void?
  down? : any/c
```

Sets whether the Caps Lock key was on for the event.

```
(send a-key-event set-control-down down?) → void?
  down? : any/c
```

Sets whether the Control key was down for the event.

Under Mac OS X, if a control-key press is combined with a mouse button click, the event is reported as a right-button click and `get-control-down` for the event reports `#f`.

```
(send a-key-event set-key-code code) → void?
  code : (or/c char? key-code-symbol?)
```

Sets the virtual key code for the event, either a character or one of the special symbols listed with `get-key-code`.

```
(send a-key-event set-key-release-code code) → void?
  code : (or/c char? key-code-symbol?)
```

Sets the virtual key code for a release event, either a character or one of the special symbols listed with get-key-code. See also get-key-release-code.

```
(send a-key-event set-meta-down down?) → void?
  down? : any/c
```

Sets whether the Meta (X), Alt (Windows), or Command (Mac OS X) key was down for the event.

```
(send a-key-event set-other-altgr-key-code code) → void?
  code : (or/c char? key-code-symbol? false/c)
```

Sets the key code produced by get-other-altgr-key-code.

```
(send a-key-event set-other-caps-key-code code) → void?
  code : (or/c char? key-code-symbol? false/c)
```

Sets the key code produced by get-other-caps-key-code.

```
(send a-key-event set-other-shift-altgr-key-code code) → void?
  code : (or/c char? key-code-symbol? false/c)
```

Sets the key code produced by get-other-shift-altgr-key-code.

```
(send a-key-event set-other-shift-key-code code) → void?
  code : (or/c char? key-code-symbol? false/c)
```

Sets the key code produced by get-other-shift-key-code.

```
(send a-key-event set-shift-down down?) → void?
  down? : any/c
```

Sets whether the Shift key was down for the event.

```
(send a-key-event set-x pos) → void?
  pos : exact-integer?
```

Sets the x-position of the mouse at the time of the event in the target's window's (client-area) coordinate system.

```
(send a-key-event set-y pos) → void?
  pos : exact-integer?
```

Sets the y-position of the mouse at the time of the event in the target's window's (client-area) coordinate system.

2.1.25 labelled-menu-item<%>

```
labelled-menu-item<%> : interface?
  implements: menu-item<%>
```

A labelled-menu-item<%> object is a menu-item<%> with a string label (i.e., any menu item other than a separator). More specifically, it is an instance of either menu-item% (a plain menu item), checkable-menu-item% (a checkable menu item), or menu% (a submenu).

```
(send a-labelled-menu-item enable enabled?) → void?
  enabled? : any/c
```

Enables or disables the menu item. If the item is a submenu (or menu in a menu bar), the entire menu is disabled, but each submenu item's is-enabled? method returns #f only if the item is specifically disabled (in addition to the submenu).

```
(send a-labelled-menu-item get-help-string)
 → (or/c label-string? false/c)
```

Returns the help string for the menu item, or #f if the item has no help string.

When an item has a help, the string may be used to display help information to the user.

```
(send a-labelled-menu-item get-label) → label-string?
```

Returns the item's label.

See also set-label and get-plain-label.

```
(send a-labelled-menu-item get-plain-label) → label-string?
```

Like get-label, except that &s in the label are removed as described in set-label.

(send a-labelled-menu-item is-enabled?) → boolean?

Returns #t if the menu item is enabled, #f otherwise.

See also enable.

(send a-labelled-menu-item on-demand) → void?

Specification: Normally called when the user clicks on the menu bar containing the item (before the user sees any menu items), just before the popup menu containing the item is popped up, or just before inspecting the menu bar containing the item for a shortcut key binding.

A on-demand in menu-item-container<%> method can be overridden in such a way that the container does not call the on-demand method of its items.

Default implementation: Calls the demand-callback procedure that was provided when the object was created.

(send a-labelled-menu-item set-help-string help) → void?
  help : (or/c label-string? false/c)

Sets the help string for the menu item. Use #f to remove the help string for an item.

(send a-labelled-menu-item set-label label) → void?
  label : label-string?

Sets the menu item's label. If the item has a shortcut, the shortcut is not affected.

If the label contains & and the window is a control, the label is parsed specially; under Windows and X, the character following a & is underlined in the displayed menu to indicate a keyboard mnemonic. Pressing the Alt key with an underlined character from a menu's name in the menu bar causes the menu to be selected (via on-menu-char). When a menu has the focus, the mnemonic characters are used for navigation without Alt. A && in the label is replaced by a literal (non-navigation) &. Under Mac OS X, &s in the label are parsed in the same way as for X and Windows, but no mnemonic underline is displayed.

A & is always preserved in the label returned by get-label, but never preserved in the label returned by get-plain-label.

For historical reasons, if a label contains a tab character, then the tab and all remaining characters are hidden in the displayed menu.

2.1.26 list-box%

```
list-box% : class?
  superclass: object%
  extends: list-control<%>
```

A list box allows the user to select one or more string items from a scrolling list. A list box is either a single-selection control (if an item is selected, the previous selection is removed) or a multiple-selection control (clicking an item toggles the item on or off independently of other selections).

Whenever the user changes the selection in a list box, the list box's callback procedure is called. A callback procedure is provided as an initialization argument when each list box is created.

List box items are indexed from 0.

See also choice%.

```
(new list-box%  [label label]
                [choices choices]
                [parent parent]
               [[callback callback]
                [style style]
                [selection selection]
                [font font]
                [label-font label-font]
                [enabled enabled]
                [vert-margin vert-margin]
                [horiz-margin horiz-margin]
                [min-width min-width]
                [min-height min-height]
                [stretchable-width stretchable-width]
                [stretchable-height stretchable-height]])
 → (is-a?/c list-box%)
  label : (or/c label-string? false/c)
  choices : (listof label-string?)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c list-box%) (is-a?/c control-event%) . -> . any)
           = (lambda (c e) (void))
  style : (listof (one-of/c 'single 'multiple 'extended
                            'vertical-label 'horizontal-label
                            'deleted))
        = '(single)
```

```
selection : (or/c exact-nonnegative-integer? false/c) = #f
font : (is-a?/c font%) = view-control-font
label-font : (is-a?/c font%) = normal-control-font
enabled : any/c = #t
vert-margin : (integer-in 0 1000) = 2
horiz-margin : (integer-in 0 1000) = 2
min-width : (integer-in 0 10000) = graphical-minimum-width
min-height : (integer-in 0 10000) = graphical-minimum-height
stretchable-width : any/c = #t
stretchable-height : any/c = #t
```

If *label* is not #f, it is used as the list box label. Otherwise, the list box will not display its label.

If & occurs in *label*, it is specially parsed as for button%.

The *choices* list specifies the initial list of items to appear in the list box.

The *callback* procedure is called when the user changes the list box selection, by either selecting, re-selecting, deselecting, or double-clicking an item. The type of the event provided to the callback is 'list-box-dclick when the user double-clicks on an item, or 'list-box otherwise.

The *style* specification must include exactly one of the following:

- 'single — Creates a single-selection list.
- 'multiple — Creates a multiple-selection list where a single click deselects other items and selects a new item. Use this style for a list when single-selection is common, but multiple selections are allowed.
- 'extended — Creates a multiple-selection list where a single click extends or contracts the selection by toggling the clicked item. Use this style for a list when multiple selections are the rule rather than the exception.

The 'multiple and 'extended styles determine a platform-independent interpretation of unmodified mouse clicks, but dragging, shift-clicking, control-clicking, etc. have platform-standard interpretations. Whatever the platform-specific interface, the user can always select disjoint sets of items or deselect items (and leave no items selected). On some platforms, the user can deselect the (sole) selected item in a 'single list box.

If *style* includes 'vertical-label, then the list box is created with a label above the control; if *style* does not include 'vertical-label (and optionally includes 'horizontal-label), then the label is created to the left of the list box. If *style* includes 'deleted, then the list box is created as hidden, and it does not affect its parent's geometry; the list box can be made active later by calling *parent*'s add-child method.

If *selection* is an integer, it is passed to set-selection to set the initial selection. The *selection* must be less than the length of *choices*.

The *font* argument determines the font for the control content, and *label-font* determines the font for the control label. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-list-box append item [data]) → void?
  item : string
  data : any/c = #f
```

Overrides append in list-control<%>.

Adds a new item to the list box with an associated "data" object. The *data* object is not displayed in the list box; it is provided merely as a convenience for use with get-data, possibly allowing a programmer to avoid managing a separate item-to-data mapping in addition to the list box control.

See also append in list-control<%>.

```
(send a-list-box delete n) → void?
  n : exact-nonnegative-integer?
```

Deletes the item indexed by *n*. List box items are indexed from 0. If *n* is equal to or larger than the number of items in the control, an exn:fail:contract exception is raised.

Selected items that are not deleted remain selected, and no other items are selected.

```
(send a-list-box get-data n) → any/c
  n : exact-nonnegative-integer?
```

Returns the data for the item indexed by *n*, or #f if there is no associated data. List box items are indexed from 0. If *n* is equal to or larger than the number of choices, an exn:fail:contract exception is raised.

See also append and set-data.

```
(send a-list-box get-first-visible-item)
 → exact-nonnegative-integer?
```

Reports the index of the item currently scrolled to the top of the list box. List box items are indexed from 0.

```
(send a-list-box get-label-font) → (is-a?/c font%)
```

Returns the font used for the control's label, which is optionally supplied when a list box is created.

```
(send a-list-box get-selections)
 → (listof exact-nonnegative-integer?)
```

Returns a list of indices for all currently selected items. List box items are indexed from `0`.

For single-selection lists, the result is always either `null` or a list containing one number.

```
(send a-list-box is-selected? n) → boolean?
  n : exact-nonnegative-integer?
```

Returns `#t` if the item index by *n* is selected, `#f` otherwise. List box items are indexed from `0`. If *n* is equal to or larger than the number of choices, an `exn:fail:contract` exception is raised.

A list box's selection can be changed by the user clicking the control, and such changes do not go through this method; use the control callback procedure (provided as an initialization argument) to monitor selection changes.

```
(send a-list-box number-of-visible-items)
 → exact-positive-integer?
```

Returns the maximum number of items in the list box that are visible to the user with the control's current size (rounding down if the exact answer is fractional, but returning at least `1`).

```
(send a-list-box select n [select?]) → void?
  n : exact-nonnegative-integer?
  select? : any/c = #t
```

Selects or deselects an item. For selection in a single-selection list box, if a different choice is currently selected, it is automatically deselected. For selection in a multiple-selection list box, other selections are preserved, unlike `set-selection`.

If *select?* is `#f`, the item indexed by *n* is deselected; otherwise it is selected. List box items are indexed from `0`. If *n* is equal to or larger than the number of choices, an `exn:fail:contract` exception is raised.

A list box's selection can be changed by the user clicking the control, and such changes do

not go through this method; use the control callback procedure (provided as an initialization argument) to monitor selection changes.

The control's callback procedure is *not* invoked.

```
(send a-list-box set choices) → void?
  choices : (listof label-string?)
```

Clears the list box and installs a new list of items.

```
(send a-list-box set-data n data) → void?
  n : exact-nonnegative-integer?
  data : any/c
```

Sets the associated data for item indexed by *n*. List box items are indexed from 0. If *n* is equal to or larger than the number of choices, an `exn:fail:contract` exception is raised.

See also `append`.

```
(send a-list-box set-first-visible-item n) → void?
  n : exact-nonnegative-integer?
```

Scrolls the list box so that the item indexed by *n* is at the top of the list box display. List box items are indexed from 0. If *n* is equal to or larger than the number of choices, an `exn:fail:contract` exception is raised.

A list box's scroll position can be changed by the user clicking the control, and such changes do not go through this method. A program cannot detect when the scroll position changes except by polling `get-first-visible-item`.

```
(send a-list-box set-string n label) → void?
  n : exact-nonnegative-integer?
  label : label-string?
```

Sets the item indexed by *n*. List box items are indexed from 0. If *n* is equal to or larger than the number of choices, an `exn:fail:contract` exception is raised.

2.1.27 `list-control<%>`

```
list-control<%> : interface?
  implements: control<%>
```

A list control gives the user a list of string items to choose from. There are two built-in classes that implement `list-control<%>`:

- `choice%` — presents the list in a popup menu (so the user can choose only one item at a time)
- `list-box%` — presents the list in a scrolling box, allowing the use to choose one item (if the style includes `'single`) or any number of items

In either case, the set of user-selectable items can be changed dynamically.

```
(send a-list-control append item) → void?
  item : string
```

Adds a new item to the list of user-selectable items. The current selection is unchanged (unless the list control is an empty choice control, in which case the new item is selected).

```
(send a-list-control clear) → void?
```

Removes all user-selectable items from the control.

```
(send a-list-control find-string s)
 → (or/c exact-nonnegative-integer? false/c)
  s : string
```

Finds a user-selectable item matching the given string. If no matching choice is found, `#f` is returned, otherwise the index of the matching choice is returned (items are indexed from `0`).

```
(send a-list-control get-number) → exact-nonnegative-integer?
```

Returns the number of user-selectable items in the control (which is also one more than the greatest index in the list control).

```
(send a-list-control get-selection)
 → (or/c exact-nonnegative-integer? false/c)
```

Returns the index of the currently selected item (items are indexed from `0`). If the choice item currently contains no choices or no selections, `#f` is returned. If multiple selections are allowed and multiple items are selected, the index of the first selection is returned.

```
(send a-list-control get-string n)
 → (and/c immutable? label-string?)
```

n : exact-nonnegative-integer?

Returns the item for the given index (items are indexed from 0). If the provided index is larger than the greatest index in the list control, an exn:fail:contract exception is raised.

(send *a-list-control* get-string-selection)
 → (or/c (and/c immutable? label-string?) false/c)

Returns the currently selected item. If the control currently contains no choices, #f is returned. If multiple selections are allowed and multiple items are selected, the first selection is returned.

(send *a-list-control* set-selection *n*) → void?
  n : exact-nonnegative-integer?

Selects the item specified by the given index (items are indexed from 0). If the given index larger than the greatest index in the list control, an exn:fail:contract exception is raised.

In a list box control, all other items are deselected, even if multiple selections are allowed in the control. See also select in list-box%.

The control's callback procedure is *not* invoked when this method is called.

The list control's selection can be changed by the user clicking the control, and such changes do not go through this method; use the control callback procedure (provided as an initialization argument) to monitor selection changes.

(send *a-list-control* set-string-selection *s*) → void?
  s : string

Selects the item that matches the given string. If no match is found in the list control, an exn:fail:contract exception is raised.

In a list box control, all other items are deselected, even if multiple selections are allowed in the control. See also select in list-box%.

The control's callback procedure is *not* invoked when this method is called.

The list control's selection can be changed by the user clicking the control, and such changes do not go through this method; use the control callback procedure (provided as an initialization argument) to monitor selection changes.

2.1.28 menu%

```
menu% : class?
  superclass: object%
  extends: menu-item-container<%>
           labelled-menu-item<%>
```

A menu% object is a submenu within a menu% or popup-menu%, or as a top-level menu in a menu-bar%.

```
(new menu% [label label]
           [parent parent]
          [[help-string help-string]
           [demand-callback demand-callback]])
 → (is-a?/c menu%)
  label : label-string?
  parent : (or/c (is-a?/c menu%) (is-a?/c popup-menu%)
                 (is-a?/c menu-bar%))
  help-string : (or/c label-string? false/c) = #f
  demand-callback : ((is-a?/c menu%) . -> . any)
                  = (lambda (m) (void))
```

Creates a new menu with the given label.

If *label* contains a &, it is handled specially; under Windows, the character following a & is underlined in the displayed menu title to indicate a keyboard mnemonic. Pressing and releasing the Alt key switches to menu-selection mode in the menu bar where mnemonic characters are used for navigation. An Alt combination might select a specific menu via on-menu-char. A && in *label* is replaced by a literal (non-navigation) &ampersand. Under X and Mac OS X, &s in the label are parsed in the same way as for Windows, but no mnemonic underline is displayed.

If *help-string* is not #f, the menu has a help string. See get-help-string for more information.

The *demand-callback* procedure is called by the default on-demand method with the object itself.

2.1.29 menu-bar%

```
menu-bar% : class?
  superclass: object%
  extends: menu-item-container<%>
```

A menu-bar% object is created for a particular frame% object. A frame can have at most one menu bar; an exn:fail:contract exception is raised when a new menu bar is created for a frame that already has a menu bar.

```
(new menu-bar% [parent parent]
               [[demand-callback demand-callback]])
 → (is-a?/c menu-bar%)
  parent : (or/c (is-a?/c frame%) (one-of/c 'root))
  demand-callback : ((is-a?/c menu-bar%) . -> . any)
                  = (lambda (m) (void))
```

Creates a menu bar in the specified frame. The menu bar is initially empty. If 'root is supplied as *parent*, the menu bar becomes active only when no other frames are shown. A 'root *parent* is allowed only when current-eventspace-has-menu-root? returns #t, and only if no such menu bar has been created before, otherwise an exn:fail:contract exception is raised.

The *demand-callback* procedure is called by the default on-demand method with the object itself.

```
(send a-menu-bar enable enable?) → void?
  enable? : any/c
```

Enables or disables the menu bar (i.e., all of its menus). Each menu's is-enabled? method returns #f only if the menu is specifically disabled (in addition to the menu bar).

```
(send a-menu-bar get-frame) → (is-a?/c frame%)
```

Returns the menu bar's frame.

```
(send a-menu-bar is-enabled?) → boolean?
```

Returns #t if the menu bar is enabled, #f otherwise.

2.1.30 menu-item<%>

```
menu-item<%> : interface?
```

A menu-item<%> object is an element within a menu%, popup-menu%, or menu-bar%. Operations that affect the parent — such as renaming the item, deleting the item, or adding a check beside the item — are accomplished via the menu-item<%> object.

A menu item is either a separator-menu-item% object (merely a separator), of a labelled-menu-item<%> object; the latter is more specifically an instance of either menu-item% (a plain menu item), checkable-menu-item% (a checkable menu item), or menu% (a submenu).

(send a-menu-item delete) → void?

Removes the item from its parent. If the menu item is already deleted, delete has no effect.

See also restore.

(send a-menu-item get-parent)
 → (or/c (is-a?/c menu%) (is-a?/c popup-menu%) (is-a?/c menu-bar%))

Returns the menu, popup menu, or menu bar containing the item. The parent for a menu item is specified when the menu item is created, and it cannot be changed.

(send a-menu-item is-deleted?) → boolean?

Returns #t if the menu item is deleted from its parent, #f otherwise.

(send a-menu-item restore) → void?

Adds a deleted item back into its parent. The item is always restored to the end of the parent, regardless of its original position. If the item is not currently deleted, restore has no effect.

2.1.31 menu-item%

---

menu-item% : class?
  superclass: object%
  extends: selectable-menu-item<%>

A menu-item% is a plain string-labelled menu item. Its parent must be a menu% or popup-menu%. When the user selects the menu item, its callback procedure is called.

```
(new menu-item%  [label label]
                 [parent parent]
                [[callback callback]
                 [shortcut shortcut]
                 [help-string help-string]
                 [demand-callback demand-callback]
                 [shortcut-prefix shortcut-prefix]])
 → (is-a?/c menu-item%)
  label : label-string?
  parent : (or/c (is-a?/c menu% popup-menu%))
  callback : ((is-a?/c menu-item%) (is-a?/c control-event%) . -> . any)
           = (lambda (i e) (void))
  shortcut : (or/c char? false/c) = #f
  help-string : (or/c label-string? false/c) = #f
  demand-callback : ((is-a?/c menu-item%) . -> . any)
                  = (lambda (i) (void))
  shortcut-prefix : (listof (one-of/c 'alt 'cmd 'meta 'ctl
                                      'shift 'option))
                  = (get-default-shortcut-prefix)
```

Creates a new menu item in *parent*. The item is initially shown, appended to the end of its parent. The *callback* procedure is called (with the event type 'menu) when the user selects the menu item (either via a menu bar, popup-menu in window<%>, or popup-menu in editor-admin%).

See set-label for information about mnemonic &s in *label*.

If *shortcut* is not #f, the item has a shortcut. See get-shortcut for more information. The *shortcut-prefix* argument determines the set of modifier keys for the shortcut; see get-shortcut-prefix.

If help is not #f, the item has a help string. See get-help-string for more information.

The *demand-callback* procedure is called by the default on-demand method with the object itself.

2.1.32 menu-item-container<%>

---

menu-item-container<%> : interface?

A menu-item-container<%> object is a menu%, popup-menu%, or menu-bar%.

```
(send a-menu-item-container get-items)
```

→ (listof (is-a?/c menu-item<%>))

Returns a list of the items in the menu, popup menu, or menu bar. The order of the items in the returned list corresponds to the order as the user sees them in the menu or menu bar.

(send *a-menu-item-container* on-demand) → void?

Specification: Called when the user clicks on the container as a menu bar (before the user sees any menu items), just before the container as a popup menu is popped up, or just before inspecting the menu bar containing the item for a shortcut key binding.

If the container is not a menu bar or a popup menu, this method is normally called via the on-demand method of the container's owning menu bar or popup menu, because the default implementation of the method chains to the on-demand method of its items. However, the method can be overridden in a container such that it does not call the on-demand method of its items.

Default implementation: Calls the demand-callback procedure that was provided when the object was created, then calls the on-demand method of the contained items.

2.1.33 message%

---

message% : class?
  superclass: object%
  extends: control<%>

A message control is a static line of text or a static bitmap. The text or bitmap corresponds to the message's label (see set-label).

```
(new message%  [label label]
               [parent parent]
              [[style style]
               [font font]
               [enabled enabled]
               [vert-margin vert-margin]
               [horiz-margin horiz-margin]
               [min-width min-width]
               [min-height min-height]
               [stretchable-width stretchable-width]
               [stretchable-height stretchable-height]
               [auto-resize auto-resize]])
 → (is-a?/c message%)
```

```
label : (or/c label-string? (is-a?/c bitmap%)
             (or-of/c 'app 'caution 'stop))
parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
               (is-a?/c panel%) (is-a?/c pane%))
style : (listof (one-of/c 'deleted)) = null
font : (is-a?/c font%) = normal-control-font
enabled : any/c = #t
vert-margin : (integer-in 0 1000) = 2
horiz-margin : (integer-in 0 1000) = 2
min-width : (integer-in 0 10000) = graphical-minimum-width
min-height : (integer-in 0 10000) = graphical-minimum-height
stretchable-width : any/c = #f
stretchable-height : any/c = #f
auto-resize : any/c = #f
```

Creates a string or bitmap message initially showing *label*. If *label* is a bitmap, then the bitmap must be valid (see ok? in bitmap%) and not installed in a bitmap-dc% object; otherwise, an exn:fail:contract exception is raised. If the bitmap has a mask (see get-loaded-mask in bitmap%) that is the same size as the bitmap, then the mask is used for the label; furthermore, in contrast to the limitations of draw-bitmap in dc<%>, non-monochrome label masks work consistently on all platforms. An 'app, 'caution, or 'stop symbol for *label* indicates an icon; 'app is the application icon (Windows and Mac OS X) or a generic "info" icon (X), 'caution is a caution-sign icon, and 'stop a stop-sign icon.

If & occurs in *label*, it is specially parsed; under Windows and X, the character following & is underlined in the displayed control to indicate a keyboard mnemonic. (Under Mac OS X, mnemonic underlines are not shown.) The mnemonic is meaningless for a message (as far as on-traverse-char in top-level-window<%> is concerned), but it is supported for consistency with other control types. A programmer may assign a meaning to the mnemonic (e.g., by overriding on-traverse-char).

If *style* includes 'deleted, then the message is created as hidden, and it does not affect its parent's geometry; the message can be made active later by calling *parent*'s add-child method.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

If *auto-resize* is not #f, then automatic resizing is initially enanbled (see auto-resize), and the message% object's graphical minimum size is as small as possible.

```
(send a-message auto-resize) → boolean?
(send a-message auto-resize on?) → void?
```

on? : any/c

Reports or sets whether the message%'s min-width and min-height are automatically set when the label is changed via set-label.

(send *a-message* set-label *label*) → void?
label : (or/c label-string? (is-a?/c bitmap%))

Overrides set-label in window<%>.

The same as set-label in window<%> when *label* is a string.

Otherwise, sets the bitmap label for a bitmap message. Since *label* is a bitmap, the bitmap must be valid (see ok? in bitmap%) and not installed in a bitmap-dc% object; otherwise, an exn:fail:contract exception is raised. If the bitmap has a mask (see get-loaded-mask in bitmap%) that is the same size as the bitmap, then the mask is used for the label; furthermore, in contrast to the limitations of draw-bitmap in dc<%>, non-monochrome label masks work consistently on all platforms. The bitmap label is installed only if the control was originally created with a bitmap label.

2.1.34 mouse-event%

---

mouse-event% : class?
superclass: event%

A mouse-event% object encapsulates a mouse event. Mouse events are primarily processed by on-subwindow-event in window<%> and on-event in canvas<%>.

See also §1.1.3 “Mouse and Keyboard Events”.

```
(new mouse-event%  [event-type event-type]
                   [[left-down left-down]
                    [middle-down middle-down]
                    [right-down right-down]
                    [x x]
                    [y y]
                    [shift-down shift-down]
                    [control-down control-down]
                    [meta-down meta-down]
                    [alt-down alt-down]
                    [time-stamp time-stamp]
                    [caps-down caps-down]])
 → (is-a?/c mouse-event%)
```

```
event-type : (or/c 'enter 'leave 'left-down 'left-up
                   'middle-down 'middle-up
                   'right-down 'right-up 'motion)
left-down : any/c = #f
middle-down : any/c = #f
right-down : any/c = #f
x : exact-integer? = 0
y : exact-integer? = 0
shift-down : any/c = #f
control-down : any/c = #f
meta-down : any/c = #f
alt-down : any/c = #f
time-stamp : exact-integer? = 0
caps-down : any/c = #f
```

Creates a mouse event for a particular type of event. The event types are:

- `'enter` — mouse pointer entered the window
- `'leave` — mouse pointer left the window
- `'left-down` — left mouse button pressed
- `'left-up` — left mouse button released
- `'middle-down` — middle mouse button pressed
- `'middle-up` — middle mouse button released
- `'right-down` — right mouse button pressed (Mac OS X: click with control key pressed)
- `'right-up` — right mouse button released (Mac OS X: release with control key pressed)
- `'motion` — mouse moved, with or without button(s) pressed

See the corresponding `get-` and `set-` methods for information about *left-down*, *middle-down*, *right-down*, *x*, *y*, *shift-down*, *control-down*, *meta-down*, *alt-down*, *time-stamp*, and *caps-down*.

```
(send a-mouse-event button-changed? [button]) → boolean?
  button : (or/c 'left 'middle 'right 'any) = 'any
```

Returns `#t` if this was a mouse button press or release event, `#f` otherwise. See also `button-up?` and `button-down?`.

If *button* is not `'any`, then `#t` is only returned if it is a release event for a specific button.

```
(send a-mouse-event button-down? [button]) → boolean?
  button : (or/c 'left 'middle 'right 'any) = 'any
```

Returns #t if the event is for a button press, #f otherwise.

If *button* is not 'any, then #t is only returned if it is a press event for a specific button.

```
(send a-mouse-event button-up? [button]) → boolean?
  button : (or/c 'left 'middle 'right 'any) = 'any
```

Returns #t if the event is for a button release, #f otherwise. (As noted in §1.1.3 "Mouse and Keyboard Events", button release events are sometimes dropped.)

If *button* is not 'any, then #t is only returned if it is a release event for a specific button.

```
(send a-mouse-event dragging?) → boolean?
```

Returns #t if this was a dragging event (motion while a button is pressed), #f otherwise.

```
(send a-mouse-event entering?) → boolean?
```

Returns #t if this event is for the mouse entering a window, #f otherwise.

When the mouse button is up, an enter/leave event notifies a window that it will start/stop receiving mouse events. When the mouse button is down, however, the window receiving the mouse-down event receives all mouse events until the button is released; enter/leave events are not sent to other windows, and are not reliably delivered to the click-handling window (since the window can detect movement out of its region via get-x and get-y). See also §1.1.3 "Mouse and Keyboard Events".

```
(send a-mouse-event get-alt-down) → boolean?
```

Returns #t if the Option (Mac OS X) key was down for the event. When the Alt key is pressed in Windows, it is reported as a Meta press (see get-meta-down).

```
(send a-mouse-event get-caps-down) → boolean?
```

Returns #t if the Caps Lock key was on for the event.

```
(send a-mouse-event get-control-down) → boolean?
```

Returns #t if the Control key was down for the event.

Under Mac OS X, if a control-key press is combined with a mouse button click, the event is reported as a right-button click and get-control-down for the event reports #f.

```
(send a-mouse-event get-event-type)
 → (or/c 'enter 'leave 'left-down 'left-up
         'middle-down 'middle-up
         'right-down 'right-up 'motion)
```

Returns the type of the event; see mouse-event% for information about each event type. See also set-event-type .

```
(send a-mouse-event get-left-down) → boolean?
```

Returns #t if the left mouse button was down (but not pressed) during the event.

```
(send a-mouse-event get-meta-down) → boolean?
```

Returns #t if the Meta (X), Alt (Windows), or Command (Mac OS X) key was down for the event.

```
(send a-mouse-event get-middle-down) → boolean?
```

Returns #t if the middle mouse button was down (but not pressed) for the event. Under Mac OS X, a middle-button click is impossible.

```
(send a-mouse-event get-right-down) → boolean?
```

Returns #t if the right mouse button was down (but not pressed) for the event. Under Mac OS X, a control-click combination is treated as a right-button click.

```
(send a-mouse-event get-shift-down) → boolean?
```

Returns #t if the Shift key was down for the event.

```
(send a-mouse-event get-x) → exact-integer?
```

Returns the x-position of the mouse at the time of the event, in the target's window's (client-area) coordinate system.

```
(send a-mouse-event get-y) → exact-integer?
```

Returns the y-position of the mouse at the time of the event in the target's window's (client-area) coordinate system.

```
(send a-mouse-event leaving?) → boolean?
```

Returns `#t` if this event is for the mouse leaving a window, `#f` otherwise.

See `entering?` for information about enter and leave events while the mouse button is clicked.

```
(send a-mouse-event moving?) → boolean?
```

Returns `#t` if this was a moving event (whether a button is pressed is not), `#f` otherwise.

```
(send a-mouse-event set-alt-down down?) → void?
  down? : any/c
```

Sets whether the Option (Mac OS X) key was down for the event. When the Alt key is pressed in Windows, it is reported as a Meta press (see `set-meta-down`).

```
(send a-mouse-event set-caps-down down?) → void?
  down? : any/c
```

Sets whether the Caps Lock key was on for the event.

```
(send a-mouse-event set-control-down down?) → void?
  down? : any/c
```

Sets whether the Control key was down for the event.

Under Mac OS X, if a control-key press is combined with a mouse button click, the event is reported as a right-button click and `get-control-down` for the event reports `#f`.

```
(send a-mouse-event set-event-type event-type) → void?
  event-type : (or/c 'enter 'leave 'left-down 'left-up
                     'middle-down 'middle-up
                     'right-down 'right-up 'motion)
```

Sets the type of the event; see `mouse-event%` for information about each event type. See also `get-event-type` .

```
(send a-mouse-event set-left-down down?) → void?
  down? : any/c
```

Sets whether the left mouse button was down (but not pressed) during the event.

```
(send a-mouse-event set-meta-down down?) → void?
  down? : any/c
```

Sets whether the Meta (X), Alt (Windows), or Command (Mac OS X) key was down for the event.

```
(send a-mouse-event set-middle-down down?) → void?
  down? : any/c
```

Sets whether the middle mouse button was down (but not pressed) for the event. Under Mac OS X, a middle-button click is impossible.

```
(send a-mouse-event set-right-down down?) → void?
  down? : any/c
```

Sets whether the right mouse button was down (but not pressed) for the event. Under Mac OS X, a control-click combination by the user is treated as a right-button click.

```
(send a-mouse-event set-shift-down down?) → void?
  down? : any/c
```

Sets whether the Shift key was down for the event.

```
(send a-mouse-event set-x pos) → void?
  pos : exact-integer?
```

Sets the x-position of the mouse at the time of the event in the target's window's (client-area) coordinate system.

```
(send a-mouse-event set-y pos) → void?
  pos : exact-integer?
```

Sets the y-position of the mouse at the time of the event in the target's window's (client-area) coordinate system.

2.1.35 pane%

```
pane% : class?
  superclass: object%
  extends: area-container<%>
           subarea<%>
```

A pane is a both a container and a containee area. It serves only as a geometry management device. A pane% cannot be hidden or disabled like a panel% object.

A pane% object has a degenerate placement strategy for managing its children; it places them all in the upper left corner and does not stretch any of them. The horizontal-pane% and vertical-pane% classes provide useful geometry management.

See also grow-box-spacer-pane%.

```
(new pane%  [parent parent]
            [[vert-margin vert-margin]
             [horiz-margin horiz-margin]
             [border border]
             [spacing spacing]
             [alignment alignment]
             [min-width min-width]
             [min-height min-height]
             [stretchable-width stretchable-width]
             [stretchable-height stretchable-height]])
 → (is-a?/c pane%)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  vert-margin : (integer-in 0 1000) = 0
  horiz-margin : (integer-in 0 1000) = 0
  border : (integer-in 0 1000) = 0
  spacing : (integer-in 0 1000) = 0
  alignment : (list/c (one-of/c 'left 'center 'right)
                      (one-of/c 'top 'center 'bottom))
            = '(center top)
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
  stretchable-height : any/c = #t
```

For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-*

width, and *stretchable-height* arguments, see area<%>.

2.1.36 panel%

```
panel% : class?
  superclass: object%
  extends: area-container-window<%>
           subwindow<%>
```

A panel is a both a container and a containee window. It serves mainly as a geometry management device, but the 'border creates a container with a border. Unlike a pane% object, a panel% object can be hidden or disabled.

A panel% object has a degenerate placement strategy for managing its children; it places them all in the upper left corner and does not stretch any of them. The horizontal-panel% and vertical-panel% classes provide useful geometry management.

```
(new panel%  [parent parent]
             [[style style]
              [enabled enabled]
              [vert-margin vert-margin]
              [horiz-margin horiz-margin]
              [border border]
              [spacing spacing]
              [alignment alignment]
              [min-width min-width]
              [min-height min-height]
              [stretchable-width stretchable-width]
              [stretchable-height stretchable-height]])
 → (is-a?/c panel%)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  style : (listof (one-of/c 'border 'deleted)) = null
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 0
  horiz-margin : (integer-in 0 1000) = 0
  border : (integer-in 0 1000) = 0
  spacing : (integer-in 0 1000) = 0
  alignment : (list/c (one-of/c 'left 'center 'right)
                      (one-of/c 'top 'center 'bottom))
            = '(center center)
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
```

```
  stretchable-width : any/c = #t
  stretchable-height : any/c = #t
```

If the 'border style is specified, the window is created with a thin border (only in this case, the client size of the panel may be less than its total size). If *style* includes 'deleted, then the panel is created as hidden, and it does not affect its parent's geometry; the panel can be made active later by calling *parent*'s add-child method.

For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

2.1.37 popup-menu%

```
popup-menu% : class?
  superclass: object%
  extends: menu-item-container<%>
```

A popup-menu% object is created without a parent. Dynamically display a popup-menu% with popup-menu in window<%> or popup-menu in editor-admin%.

A popup menu is *not* a control. A choice% control, however, displays a single value that the user selects from a popup menu. A choice% control's popup menu is built into the control, and it is not accessible to the programmer.

```
(new popup-menu% [[title title]
                  [popdown-callback popdown-callback]
                  [demand-callback demand-callback]
                  [font font]])
 → (is-a?/c popup-menu%)
  title : (or/c label-string? false/c) = #f
  popdown-callback : ((is-a?/c popup-menu%) (is-a?/c control-event%)
                      . -> . any)
                     = (lambda (p e) (void))
  demand-callback : ((is-a?/c popup-menu%) . -> . any)
                    = (lambda (p) (void))
  font : (is-a?/c font%) = normal-control-font
```

If *title* is not #f, it is used as a displayed title at the top of the popup menu.

If *title* contains &, it is handled specially, the same as for menu% titles. A popup menu mnemonic is not useful, but it is supported for consistency with other menu labels.

The *popdown-callback* procedure is invoked when a popup menu is dismissed. If the popup menu is dismissed without an item being selected, *popdown-callback* is given a control-event% object with the event type 'menu-popdown-none. If the popup menu is dismissed via an item selection, the item's callback is invoked first, and then *popdown-callback* is given a control-event% object with the event type 'menu-popdown.

The *demand-callback* procedure is called by the default on-demand method with the object itself.

The *font* argument determines the font for the popup menu's items.

(send a-popup-menu get-font) → (is-a?/c font%)

Returns the font used for the popup menu's items, which is optionally supplied when a popup menu is created.

(send a-popup-menu get-popup-target)
 → (or/c (is-a?/c window<%>) (is-a?/c editor<%>) false/c)

Returns the context in which the popup menu is currently displayed, or #f if it is not popped up in any window.

The context is set before the on-demand method is called, and it is not removed until after the popup-menu's callback is invoked. (Consequently, it is also set while an item callback is invoked, if the user selected an item.)

(send a-popup-menu set-min-width *width*) → void?
  width : (integer-in 0 10000)

Sets the popup menu's minimum width in pixels.

2.1.38 radio-box%

---

radio-box% : class?
  superclass: object%
  extends: control<%>

A radio-box% control allows the user to select one of number of mutually exclusive items. The items are displayed as a vertical column or horizontal row of labelled *radio buttons*. Unlike a list-control<%>, the set of items in a radio-box% cannot be changed dynamically.

Whenever the user changes the selected radio button, the radio box's callback procedure is

invoked. A callback procedure is provided as an initialization argument when each radio box is created.

```
(new radio-box%  [label label]
                 [choices choices]
                 [parent parent]
                [[callback callback]
                 [style style]
                 [selection selection]
                 [font font]
                 [enabled enabled]
                 [vert-margin vert-margin]
                 [horiz-margin horiz-margin]
                 [min-width min-width]
                 [min-height min-height]
                 [stretchable-width stretchable-width]
                 [stretchable-height stretchable-height]])
 → (is-a?/c radio-box%)
  label : (or/c label-string? false/c)
  choices : (or/c (listof label-string?) (listof (is-a?/c bitmap%)))
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c radio-box%) (is-a?/c control-event%)
              . -> . any)
           = (lambda (r e) (void))
  style : (listof (one-of/c 'horizontal 'vertical
                            'vertical-label 'horizontal-label
                            'deleted))
        = '(vertical)
  selection : (or/c exact-nonnegative-integer? #f) = 0
  font : (is-a?/c font%) = normal-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #f
  stretchable-height : any/c = #f
```

Creates a radio button set with string or bitmap labels. The *choices* list specifies the radio button labels; the list of choices must be homogeneous, either all strings or all bitmaps.

If & occurs in *label*, it is specially parsed as for button%.

Each string in *choices* can also contain a &, which creates a mnemonic for clicking the

corresponding radio button. As for *label*, a && is converted to a &.

If *choices* is a list of bitmaps, then the bitmaps must be valid (see ok? in bitmap%) and not installed in a bitmap-dc% object; otherwise, an exn:fail:contract exception is raised. If the bitmap has a mask (see get-loaded-mask in bitmap%) that is the same size as the bitmap, then the mask is used for the label; furthermore, in contrast to the limitations of draw-bitmap in dc<%>, non-monochrome label masks work consistently on all platforms.

If *label* is a string, it is used as the label for the radio box. Otherwise, the radio box does not display its label.

The *callback* procedure is called (with the event type 'radio-box) when the user changes the radio button selection.

The *style* argument must include either 'vertical for a collection of radio buttons vertically arranged, or 'horizontal for a horizontal arrangement. If *style* includes 'vertical-label, then the radio box is created with a label above the control; if *style* does not include 'vertical-label (and optionally includes 'horizontal-label), then the label is created to the left of the radio box. If *style* includes 'deleted, then the radio box is created as hidden, and it does not affect its parent's geometry; the radio box can be made active later by calling *parent*'s add-child method.

By default, the first radio button is initially selected. If *selection* is positive or #f, it is passed to set-selection to set the initial radio button selection.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-radio-box enable enable?) → void?
  enable? : any/c
(send a-radio-box enable n enable?) → void?
  n : exact-nonnegative-integer?
  enable? : any/c
```

Overrides enable in window<%>.

If a single argument is provided, the entire radio box is enabled or disabled.

If two arguments are provided, then if *enable?* is #f, the *n*th radio button is disabled, otherwise it is enabled (assuming the entire radio box is enabled). Radio buttons are numbered from 0. If *n* is equal to or larger than the number of radio buttons in the radio box, an exn:fail:contract exception is raised.

```
(send a-radio-box get-item-label n) → string?
```

```
  n : exact-nonnegative-integer?
```

Gets the label of a radio button by position. Radio buttons are numbered from 0. If *n* is equal to or larger than the number of radio buttons in the radio box, an exn:fail:contract exception is raised.

```
(send a-radio-box get-item-plain-label n) → string?
  n : exact-nonnegative-integer?
```

Like get-item-label, except that the label must be a string and &s in the label are removed.

```
(send a-radio-box get-number) → exact-nonnegative-integer?
```

Returns the number of radio buttons in the radio box.

```
(send a-radio-box get-selection)
 → (or/c exact-nonnegative-integer? #f)
```

Gets the position of the selected radio button, returning #f if no button is selected. Radio buttons are numbered from 0.

```
(send a-radio-box is-enabled?) → boolean?
(send a-radio-box is-enabled? n) → boolean?
  n : exact-nonnegative-integer?
```

Overrides is-enabled? in window<%>.

If no arguments are provided, the enable state of the entire radio box is reported.

Otherwise, returns #f if *n*th radio button is disabled (independent of disabling the entire radio box), #t otherwise. Radio buttons are numbered from 0. If *n* is equal to or larger than the number of radio buttons in the radio box, an exn:fail:contract exception is raised.

```
(send a-radio-box set-selection n) → void?
  n : (or/c exact-nonnegative-integer? #f)
```

Sets the selected radio button by position, or deselects all radio buttons if *n* is #f. (The control's callback procedure is *not* invoked.) Radio buttons are numbered from 0. If *n* is equal to or larger than the number of radio buttons in the radio box, an exn:fail:contract exception is raised.

A radio box's selection can be changed by the user clicking the control, and such changes do not go through this method; use the control callback procedure (provided as an initialization argument) to monitor selection changes.

2.1.39 selectable-menu-item<%>

```
selectable-menu-item<%> : interface?
  implements: labelled-menu-item<%>
```

A selectable-menu-item<%> object is a labelled-menu-item<%> that the user can select. It may also have a keyboard shortcut; the shortcut is displayed in the menu, and the default on-subwindow-char method in the menu's frame dispatches to the menu item when the shortcut key combination is pressed.

```
(send a-selectable-menu-item command event) → void?
  event : (is-a?/c control-event%)
```

Invokes the menu item's callback procedure, which is supplied when an instance of menu-item% or checkable-menu-item% is created.

```
(send a-selectable-menu-item get-shortcut)
 → (or/c char? symbol? false/c)
```

Gets the keyboard shortcut character or virtual key for the menu item. This character or key is combined with the shortcut prefix, which is reported by get-shortcut-prefix.

If the menu item has no shortcut, #f is returned.

The shortcut part of a menu item name is not included in the label returned by get-label.

For a list of allowed key symbols, see get-key-code in key-event%.

```
(send a-selectable-menu-item get-shortcut-prefix)
 → (listof (one-of/c 'alt 'cmd 'meta 'ctl 'shift 'option))
```

Returns a list of symbols that indicates the keyboard prefix used for the menu item's keyboard shortcut. The allowed symbols for the list are the following:

- 'alt — Meta (Windows and X only)
- 'cmd — Command (Mac OS X only)
- 'meta — Meta (X only)
- 'ctl — Control
- 'shift — Shift
- 'option — Option (Mac OS X only)

Under X, at most one of 'alt and 'meta can be supplied; the only difference between 'alt and 'meta is the key combination's display in a menu.

The default shortcut prefix is available from get-default-shortcut-prefix.

The shortcut key, as determined by get-shortcut, matches a key event using either the normally reported key code or the other-Shift/AltGr key code (as produced by get-other-shift-key-code in key-event%, etc.). When the shortcut key is a key-code symbol or an ASCII letter or digit, then the shortcut matches only the exact combination of modifier keys listed in the prefix. For character shortcuts other than ASCII letters and digits, however, then the shortcut prefix merely determines a minimum set of modifier keys, because additional modifiers may be needed to access the character; an exception is that, under Windows or X, the Alt/Meta key press must match the prefix exactly (i.e., included or not). In all cases, the most precise match takes precedence; see map-function in keymap% for more information on match ranking.

An empty list can be used for a shortcut prefix. However, the default on-menu-char in frame% method checks for menu shortcuts only when the key event includes either a non-Shift modifier or a Function key. Thus, an empty shortcut prefix is normally useful only if the shortcut key is a Function key.

```
(send a-selectable-menu-item set-shortcut shortcut) → void?
  shortcut : (or/c char? symbol? false/c)
```

Sets the keyboard shortcut character for the menu item. See get-shortcut for more information.

If the shortcut character is set to #f, then menu item has no keyboard shortcut.

```
(send a-selectable-menu-item set-shortcut-prefix prefix)
 → void?
  prefix : (listof (one-of/c 'alt 'cmd 'meta 'ctl 'shift 'option))
```

Sets a list of symbols to indicates the keyboard prefix used for the menu item's keyboard shortcut.

See get-shortcut-prefix for more information.

2.1.40 separator-menu-item%

```
separator-menu-item% : class?
  superclass: object%
  extends: menu-item<%>
```

A separator is an unselectable line in a menu. Its parent must be a menu% or popup-menu%.

```
(new separator-menu-item% [parent parent])
 → (is-a?/c separator-menu-item%)
  parent : (or/c (is-a?/c menu%) (is-a?/c popup-menu%))
```

Creates a new separator in the menu.

2.1.41 scroll-event%

```
scroll-event% : class?
  superclass: event%
```

A scroll-event% object contains information about a scroll event. An instance of scroll-event% is always provided to on-scroll.

See get-event-type for a list of the scroll event types.

```
(new scroll-event% [[event-type event-type]
                    [direction direction]
                    [position position]
                    [time-stamp time-stamp]])
 → (is-a?/c scroll-event%)
  event-type : (one-of/c 'top 'bottom 'line-up 'line-down
                         'page-up 'page-down 'thumb)
             = 'thumb
  direction : (one-of/c 'horizontal 'vertical) = 'vertical
  position : (integer-in 0 10000) = 0
  time-stamp : exact-integer? = 0
```

See the corresponding get- and set- methods for information about *event-type*, *direction*, *position*, and *time-stamp*.

```
(send a-scroll-event get-direction)
 → (one-of/c 'horizontal 'vertical)
```

Gets the identity of the scrollbar that was modified by the event, either the horizontal scrollbar or the vertical scrollbar, as 'horizontal or 'vertical, respectively. See also set-direction.

```
(send a-scroll-event get-event-type)
```

→ (one-of/c 'top 'bottom 'line-up 'line-down 'page-up 'page-down 'thumb)

Returns the type of the event, one of the following:

- 'top — user clicked a scroll-to-top button
- 'bottom — user clicked a scroll-to-bottom button
- 'line-up — user clicked an arrow to scroll up or left one step
- 'line-down — user clicked an arrow to scroll down or right one step
- 'page-up — user clicked an arrow to scroll up or left one page
- 'page-down — user clicked an arrow to scroll down or right one page
- 'thumb — user dragged the scroll position indicator

(send *a-scroll-event* get-position) → (integer-in 0 10000)

Returns the position of the scrollbar after the action triggering the event. See also set-position.

(send *a-scroll-event* set-direction *direction*) → void?
  direction : (one-of/c 'horizontal 'vertical)

Sets the identity of the scrollbar that was modified by the event, either the horizontal scrollbar or the vertical scrollbar, as 'horizontal or 'vertical, respectively. See also get-direction.

(send *a-scroll-event* set-event-type *type*) → void?
  type : (one-of/c 'top 'bottom 'line-up 'line-down
                     'page-up 'page-down 'thumb)

Sets the type of the event. See get-event-type for information about each event type.

(send *a-scroll-event* set-position *position*) → void?
  position : (integer-in 0 10000)

Records the position of the scrollbar after the action triggering the event. (The scrollbar itself is unaffected). See also get-position.

2.1.42 slider%

---

slider% : class?
  superclass: object%
  extends: control<%>

A slider object is a panel item with a handle that the user can drag to change the control's value. Each slider has a fixed minimum and maximum value.

Whenever the user changes the value of a slider, its callback procedure is invoked. A callback procedure is provided as an initialization argument when each slider is created.

```
(new slider%  [label label]
              [min-value min-value]
              [max-value max-value]
              [parent parent]
             [[callback callback]
              [init-value init-value]
              [style style]
              [font font]
              [enabled enabled]
              [vert-margin vert-margin]
              [horiz-margin horiz-margin]
              [min-width min-width]
              [min-height min-height]
              [stretchable-width stretchable-width]
              [stretchable-height stretchable-height]])
 → (is-a?/c slider%)
  label : (or/c label-string? false/c)
  min-value : (integer-in -10000 10000)
  max-value : (integer-in -10000 10000)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c slider%) (is-a?/c control-event%) . -> . any)
           = (lambda (b e) (void))
  init-value : (integer-in -10000 10000) = min-value
  style : (listof (one-of/c 'horizontal 'vertical 'plain
                            'vertical-label 'horizontal-label
                            'deleted))
        = '(horizontal)
  font : (is-a?/c font%) = normal-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
```

```
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = (memq 'horizontal style)
  stretchable-height : any/c = (memq 'vertical style)
```

If *label* is a string, it is used as the label for the slider. Otherwise, the slider does not display its label.

If & occurs in *label*, it is specially parsed as for button%.

The *min-value* and *max-value* arguments specify the range of the slider, inclusive. The *init-value* argument optionally specifies the slider's initial value. If the sequence [*min-value*, initial-value, maximum-value] is not increasing, an exn:fail:contract exception is raised.

The *callback* procedure is called (with the event type 'slider) when the user changes the slider's value.

The *style* argument must include either 'vertical for a vertical slider, or 'horizontal for a horizontal slider. If *style* includes 'plain, the slider does not display numbers for its range and current value to the user. If *style* includes 'vertical-label, then the slider is created with a label above the control; if *style* does not include 'vertical-label (and optionally includes 'horizontal-label), then the label is created to the left of the slider. If *style* includes 'deleted, then the slider is created as hidden, and it does not affect its parent's geometry; the slider can be made active later by calling *parent*'s add-child method.

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-slider get-value) → (integer-in -10000 10000)
```

Gets the current slider value.

```
(send a-slider set-value value) → void?
  value : (integer-in -10000 10000)
```

Sets the value (and displayed position) of the slider. (The control's callback procedure is *not* invoked.) If *value* is outside the slider's minimum and maximum range, an exn:fail:contract exception is raised.

A slider's value can be changed by the user clicking the control, and such changes do not go through this method; use the control callback procedure (provided as an initialization argument) to monitor value changes.

2.1.43 subarea<%>

```
subarea<%> : interface?
  implements: area<%>
```

A subarea<%> is a containee area<%>.

All subarea<%> classes accept the following named instantiation arguments:

- horiz-margin — default is 2 for control<%> classes and group-box-panel%, 0 for others; passed to horiz-margin
- vert-margin — default is 2 for control<%> classes and group-box-panel%, 0 for others; passed to vert-margin

---

```
(send a-subarea horiz-margin) → (integer-in 0 1000)
(send a-subarea horiz-margin margin) → void?
  margin : (integer-in 0 1000)
```

Gets or sets the area's horizontal margin, which is added both to the right and left, for geometry management. See §1.1.2 "Geometry Management" for more information.

```
(send a-subarea vert-margin) → (integer-in 0 1000)
(send a-subarea vert-margin margin) → void?
  margin : (integer-in 0 1000)
```

Gets or sets the area's vertical margin, which is added both to the top and bottom, for geometry management. See §1.1.2 "Geometry Management" for more information.

2.1.44 subwindow<%>

```
subwindow<%> : interface?
  implements: subarea<%>
              window<%>
```

A subwindow<%> is a containee window.

2.1.45 tab-panel%

---

tab-panel% : class?
  superclass: vertical-panel%

A tab panel arranges its subwindows in a single column, but also includes a horizontal row of tabs at the top of the panel. See also panel%.

The tab-panel% class does not implement the virtual swapping of the panel content when a new tab is selected. Instead, it merely invokes a callback procedure to indicate that a user changed the tab selection.

```
(new tab-panel% [choices choices]
                [parent parent]
               [[callback callback]
                [style style]
                [font font]
                [enabled enabled]
                [vert-margin vert-margin]
                [horiz-margin horiz-margin]
                [border border]
                [spacing spacing]
                [alignment alignment]
                [min-width min-width]
                [min-height min-height]
                [stretchable-width stretchable-width]
                [stretchable-height stretchable-height]])
 → (is-a?/c tab-panel%)
  choices : (listof label-string?)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c tab-panel%) (is-a?/c control-event%)
              . -> . any)
           = (lambda (b e) (void))
  style : (listof (one-of/c 'no-border 'deleted)) = null
  font : (is-a?/c font%) = normal-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 0
  horiz-margin : (integer-in 0 1000) = 0
  border : (integer-in 0 1000) = 0
  spacing : (integer-in 0 1000) = 0
  alignment : (list/c (one-of/c 'left 'center 'right)
                      (one-of/c 'top 'center 'bottom))
            = '(center top)
```

```
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
  stretchable-height : any/c = #t
```

Creates a tab pane, where the *choices* list specifies the tab labels.

Each string in *choices* can contain an ampersand, which (in the future) may create a mnemonic for clicking the corresponding tab. A double ampersand is converted to a single ampersand.

The *callback* procedure is called (with the event type `'tab-panel`) when the user changes the tab selection.

If the *style* list includes `'no-border`, no border is drawn around the panel content. If *style* includes `'deleted`, then the tab panel is created as hidden, and it does not affect its parent's geometry; the tab panel can be made active later by calling *parent*'s `add-child` method.

The *font* argument determines the font for the control. For information about the *enabled* argument, see `window<%>`. For information about the *horiz-margin* and *vert-margin* arguments, see `subarea<%>`. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see `area<%>`.

```
(send a-tab-panel append choice) → void?
  choice : label-string?
```

Adds a tab to the right end of panel's top row of tabs.

The label string *choice* can contain `&`, which (in the future) may create a mnemonic for clicking the new tab. A `&&` is converted to `&`.

```
(send a-tab-panel delete n) → void?
  n : exact-nonnegative-integer?
```

Deletes an existing tab. If *n* is equal to or larger than the number of tabs on the panel, an `exn:fail:contract` exception is raised.

```
(send a-tab-panel get-item-label n) → string?
  n : exact-nonnegative-integer?
```

Gets the label of a tab by position. Tabs are numbered from `0`. If *n* is equal to or larger than the number of tabs in the panel, an `exn:fail:contract` exception is raised.

```
(send a-tab-panel get-number) → exact-nonnegative-integer?
```

Returns the number of tabs on the panel.

```
(send a-tab-panel get-selection)
 → (or/c exact-nonnegative-integer? false/c)
```

Returns the index (counting from 0) of the currently selected tab. If the panel has no tabs, the result is `#f`.

```
(send a-tab-panel set choices) → void?
  choices : (listof label-string?)
```

Removes all tabs from the panel and installs tabs with the given labels.

```
(send a-tab-panel set-item-label n label) → string?
  n : exact-nonnegative-integer?
  label : label-string?
```

Set the label for tab *n* to *label*. If *n* is equal to or larger than the number of tabs in the panel, an `exn:fail:contract` exception is raised.

```
(send a-tab-panel set-selection n) → void?
  n : exact-nonnegative-integer?
```

Sets the currently selected tab by index (counting from 0). If *n* is equal to or larger than the number of tabs in the panel, an `exn:fail:contract` exception is raised.

2.1.46 text-field%

```
text-field% : class?
  superclass: object%
  extends: control<%>
```

A `text-field%` object is an editable text field with an optional label displayed in front of it. There are two text field styles:

- A single line of text is visible, and a special control event is generated when the user presses Enter (when the text field has the focus) and the event is not handled by the text field's frame or dialog (see `on-traverse-char` in `top-level-window<%>`).

- Multiple lines of text are visible, and Enter is not handled specially.

Whenever the user changes the content of a text field, its callback procedure is invoked. A callback procedure is provided as an initialization argument when each text field is created.

The text field is implemented using a `text%` editor (with an inaccessible display). Thus, whereas `text-field%` provides only `get-value` and `set-value` to manipulate the text in a text field, the `get-editor` returns the field's editor, which provides a vast collection of methods for more sophisticated operations on the text.

The keymap for the text field's editor is initialized by calling the current keymap initializer procedure, which is determined by the `current-text-keymap-initializer` parameter.

```
(new text-field%  [label label]
                  [parent parent]
                 [[callback callback]
                  [init-value init-value]
                  [style style]
                  [font font]
                  [enabled enabled]
                  [vert-margin vert-margin]
                  [horiz-margin horiz-margin]
                  [min-width min-width]
                  [min-height min-height]
                  [stretchable-width stretchable-width]
                  [stretchable-height stretchable-height]])
 → (is-a?/c text-field%)
  label : (or/c label-string? false/c)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  callback : ((is-a?/c text-field%) (is-a?/c control-event%)
              . -> . any)
           = (lambda (t e) (void))
  init-value : string? = ""
  style : (listof (one-of/c 'single 'multiple 'hscroll 'password
                            'vertical-label 'horizontal-label
                            'deleted))
        = '(single)
  font : (is-a?/c font%) = normal-control-font
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 2
  horiz-margin : (integer-in 0 1000) = 2
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
```

stretchable-height : any/c = (memq 'multiple *style*)

If *label* is not #f, it is used as the text field label. Otherwise, the text field does not display its label.

If & occurs in *label*, it is specially parsed as for button%.

The *callback* procedure is called when the user changes the text in the text field or presses the Enter key (and Enter is not handled by the text field's frame or dialog; see on-traverse-char in top-level-window<%>). If the user presses Enter, the type of event passed to the callback is 'text-field-enter, otherwise it is 'text-field.

If *init-value* is not "", the minimum width of the text item is made wide enough to show *init-value*. Otherwise, a built-in default width is selected. For a text field in single-line mode, the minimum height is set to show one line and only the control's width is stretchable. For a multiple-line text field, the minimum height shows three lines of text and is stretchable in both directions.

The style must contain exactly one of 'single or 'multiple; the former specifies a single-line field and the latter specifies a multiple-line field. The 'hscroll style applies only to multiple-line fields; when 'hscroll is specified, the field has a horizontal scrollbar and autowrapping is disabled; otherwise, the field has no horizontal scrollbar and autowrapping is enabled. A multiple-line text field always has a vertical scrollbar. The 'password style indicates that the field should draw each character of its content using a generic symbol instead of the actual character. If *style* includes 'vertical-label, then the text field is created with a label above the control; if *style* does not include 'vertical-label (and optionally includes 'horizontal-label), then the label is created to the left of the text field. If *style* includes 'deleted, then the text field is created as hidden, and it does not affect its parent's geometry; the text field can be made active later by calling *parent*'s add-child method..

The *font* argument determines the font for the control. For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

(send *a-text-field* get-editor) → (is-a?/c text%)

Returns the editor used to implement the text field.

For a text field, the most useful methods of a text% object are the following:

- (send *a-text* get-text) returns the current text of the editor.
- (send *a-text* erase) deletes all text from the editor.
- (send *a-text* insert *str*) inserts *str* into the editor at the current caret posi-

tion.

```
(send a-text-field get-value) → string?
```

Returns the text currently in the text field.

```
(send a-text-field set-value val) → void?
  val : string?
```

Sets the text currently in the text field. (The control's callback procedure is *not* invoked.)

A text field's value can be changed by the user typing into the control, and such changes do not go through this method; use the control callback procedure (provided as an initialization argument) to monitor value changes.

2.1.47 timer%

```
timer% : class?
  superclass: object%
```

A `timer%` object encapsulates an event-based alarm. To use a timer, either instantiate it with a `timer-callback` thunk to perform the alarm-based action, to derive a new class and override the `notify` method to perform the alarm-based action. Start a timer with `start` and stop it with `stop`. Supplying an initial `interval` (in milliseconds) when creating a timer also starts the timer.

Timers have a relatively high priority in the event queue. Thus, if the timer delay is set low enough, repeated notification for a timer can preempt user activities (which might be directed at stopping the timer). For timers with relatively short delays, call `yield` within the `notify` procedure to allow guaranteed event processing.

See §1.1.4 "Event Dispatching and Eventspaces" for more information about event priorities.

```
(new timer% [[notify-callback notify-callback]
             [interval interval]
             [just-once? just-once?]])
 → (is-a?/c timer%)
  notify-callback : (-> any) = void
  interval : (or/c (integer-in 0 1000000000) false/c) = #f
  just-once? : any/c = #f
```

The *notify-callback* thunk is called by the default `notify` method when the timer ex-

pires.

If *interval* is #f (the default), the timer is not started; in that case, start must be called explicitly. If *interval* is a number (in milliseconds), then start is called with *interval* and *just-once?*.

(send *a-timer* interval) → (integer-in 0 1000000000)

Returns the number of milliseconds between each timer expiration (when the timer is running).

(send *a-timer* notify) → void?

Specification: Called (on an event boundary) when the timer's alarm expires.

Default implementation: Calls the notify-callback procedure that was provided when the object was created.

(send *a-timer* start *msec* [*just-once?*]) → void?
  msec : (integer-in 0 1000000000)
  just-once? : any/c = #f

Starts (or restarts) the timer. If the timer is already running, its alarm time is not changed.

The timer's alarm expires after *msec* milliseconds, at which point notify is called (on an event boundary). If *just-once?* is #f, the timer expires *every* *msec* milliseconds until the timer is explicitly stopped. (More precisely, the timer expires *msec* milliseconds after notify returns each time.) Otherwise, the timer expires only once.

(send *a-timer* stop) → void?

Stops the timer. A stopped timer never calls notify. If the timer has expired but the call to notify has not yet been dispatched, the call is removed from the event queue.

2.1.48 top-level-window<%>

---

top-level-window<%> : interface?
  implements: area-container-window<%>

A top-level window is either a frame% or dialog% object.

```
(send a-top-level-window can-close?) → boolean?
```

Refine this method with `augment`.

Called just before the window might be closed (e.g., by the window manager). If `#f` is returned, the window is not closed, otherwise `on-close` is called and the window is closed (i.e., the window is hidden, like calling `show` with `#f`).

This method is *not* called by `show`.

```
(send a-top-level-window can-exit?) → boolean?
```

Specification: Called before `on-exit` to check whether an exit is allowed. See `on-exit` for more information.

Default implementation: Calls `can-close?` and returns the result.

```
(send a-top-level-window center [direction]) → void?
  direction : (one-of/c 'horizontal 'vertical 'both) = 'both
```

Centers the window on the screen if it has no parent. If it has a parent, the window is centered with respect to its parent's location.

If *direction* is `'horizontal`, the window is centered horizontally. If *direction* is `'vertical`, the window is centered vertically. If *direction* is `'both`, the window is centered in both directions.

```
(send a-top-level-window get-edit-target-object)
 → (or/c (or/c (is-a?/c window<%>) (is-a?/c editor<%>)) false/c)
```

Like `get-edit-target-window`, but if an editor canvas had the focus and it also displays an editor, the editor is returned instead of the canvas. Further, if the editor's focus is delegated to an embedded editor, the embedded editor is returned.

See also `get-focus-object`.

```
(send a-top-level-window get-edit-target-window)
 → (or/c (is-a?/c window<%>) false/c)
```

Returns the window that most recently had the keyboard focus, either the top-level window or one of its currently-shown children. If neither the window nor any of its currently-shown children has even owned the keyboard focus, `#f` is returned.

See also `get-focus-window` and `get-edit-target-object`.

```
(send a-top-level-window get-eventspace) → eventspace
```

Returns the window's eventspace.

```
(send a-top-level-window get-focus-object)
 → (or/c (or/c (is-a?/c window<%>) (is-a?/c editor<%>)) false/c)
```

Like `get-focus-window`, but if an editor canvas has the focus and it also displays an editor, the editor is returned instead of the canvas. Further, if the editor's focus is delegated to an embedded editor, the embedded editor is returned.

See also `get-edit-target-object`.

```
(send a-top-level-window get-focus-window)
 → (or/c (is-a?/c window<%>) false/c)
```

Returns the window that has the keyboard focus, either the top-level window or one of its children. If neither the window nor any of its children has the focus, `#f` is returned.

See also `get-edit-target-window` and `get-focus-object`.

```
(send a-top-level-window move x y) → void?
  x : (integer-in -10000 10000)
  y : (integer-in -10000 10000)
```

Moves the window to the given position on the screen.

A window's position can be changed by the user dragging the window, and such changes do not go through this method; use `on-move` to monitor position changes.

```
(send a-top-level-window on-activate active?) → void?
  active? : any/c
```

Called when a window is *activated* or *deactivated*. A top-level window is activated when the keyboard focus moves from outside the window to the window or one of its children. It is deactivated when the focus moves back out of the window. Under Mac OS X, a child of a floating frames can have the focus instead of a child of the active non-floating frame; in other words, floating frames act as an extension of the active non-frame for keyboard focus.

The method's argument is `#t` when the window is activated, `#f` when it is deactivated.

```
(send a-top-level-window on-close) → void?
```

Refine this method with `augment`.

Called just before the window is closed (e.g., by the window manager). This method is *not* called by `show`.

See also `can-close?`.

`(send a-top-level-window on-exit) → void?`

Specification: Called by the default application quit handler (as determined by the `application-quit-handler` parameter) when the operating system requests that the application shut down (e.g., when the Quit menu item is selected in the main application menu under Mac OS X). In that case, this method is called for the most recently active top-level window in the initial eventspace, but only if the window's `can-exit?` method first returns true.

Default implementation: Calls `on-close` and then `show` to hide the window.

`(send a-top-level-window on-message message) → any/c`
`  message : any/c`

Specification: A generic message method, usually called by `send-message-to-window`.

If the method is invoked by `send-message-to-window`, then it is invoked in the thread where `send-message-to-window` was called (which is possibly *not* the handler thread of the window's eventspace).

Default implementation: Returns `#<void>`.

`(send a-top-level-window on-traverse-char event) → boolean?`
`  event : (is-a?/c key-event%)`

Specification: Attempts to handle the given keyboard event as a navigation event, such as a Tab key event that moves the keyboard focus. If the event is handled, `#t` is returned, otherwise `#f` is returned.

Default implementation: The following rules determine, in order, whether and how `event` is handled:

- If the window that currently owns the focus specifically handles the event, then `#f` is returned. The following describes window types and the keyboard events they specifically handle:
  - `editor-canvas%` — tab-exit is disabled (see `allow-tab-exit`): all keyboard events, except alphanumeric key events when the Meta (X) or Alt (Windows)

key is pressed; when tab-exit is enabled: all keyboard events except Tab, Enter, Escape, and alphanumeric Meta/Alt events.

  - `canvas%` — when tab-focus is disabled (see `accept-tab-focus`): all keyboard events, except alphanumeric key events when the Meta (X) or Alt (Windows) key is pressed; when tab-focus is enabled: no key events
  - `text-field%`, `'single` style — arrow key events and alphanumeric key events when the Meta (X) or Alt (Windows) key is not pressed (and all alphanumeric events under Mac OS X)
  - `text-field%`, `'multiple` style — all keyboard events, except alphanumeric key events when the Meta (X) or Alt (Windows) key is pressed
  - `choice%` — arrow key events and alphanumeric key events when the Meta (X) or Alt (Windows) key is not pressed
  - `list-box%` — arrow key events and alphanumeric key events when the Meta (X) or Alt (Windows) key is not pressed

- If *event* is a Tab or arrow key event, the keyboard focus is moved within the window and `#t` is returned. Across platforms, the types of windows that accept the keyboard focus via navigation may vary, but `text-field%` windows always accept the focus, and `message%`, `gauge%`, and `panel%` windows never accept the focus.

- If *event* is a Space key event and the window that currently owns the focus is a `button%`, `check-box%`, or `radio-box%` object, the event is handled in the same way as a click on the control and `#t` is returned.

- If *event* is an Enter key event and the current top-level window contains a border button, the button's callback is invoked and `#t` is returned. (The `'border` style for a `button%` object indicates to the user that pressing Enter is the same as clicking the button.) If the window does not contain a border button, `#t` is returned if the window with the current focus is not a text field or editor canvas.

- In a dialog, if *event* is an Escape key event, the event is handled the same as a click on the dialog's close box (i.e., the dialog's `can-close?` and `on-close` methods are called, and the dialog is hidden) and `#t` is returned.

- If *event* is an alphanumeric key event and the current top-level window contains a control with a mnemonic matching the key (which is installed via a label that contains `&`; see `get-label` for more information), then the keyboard focus is moved to the matching control. Furthermore, if the matching control is a `button%`, `check-box%`, or `radio-box%` button, the keyboard event is handled in the same way as a click on the control.

- Otherwise, `#f` is returned.

```
(send a-top-level-window on-system-menu-char event) → boolean?
  event : (is-a?/c key-event%)
```

Checks whether the given event pops open the system menu in the top-left corner of the window (Windows only). If the window's system menu is opened, `#t` is returned, otherwise `#f` is returned.

```
(send a-top-level-window resize width
                                height) → void?
  width : (integer-in 0 10000)
  height : (integer-in 0 10000)
```

Sets the size of the window (in pixels), but only if the given size is larger than the window's minimum size.

A window's size can be changed by the user, and such changes do not go through this method; use `on-size` to monitor size changes.

```
(send a-top-level-window show show) → void?
  show : any/c
```

If the window is already shown, it is moved front of other top-level windows. If the window is iconized (frames only), it is deiconized.

See also `show` in `window<%>`.

2.1.49 vertical-pane%

```
vertical-pane% : class?
  superclass: pane%
```

A vertical pane arranges its subwindows in a single column. See also `pane%`.

```
(new vertical-pane%  [parent parent]
                     [[vert-margin vert-margin]
                      [horiz-margin horiz-margin]
                      [border border]
                      [spacing spacing]
                      [alignment alignment]
                      [min-width min-width]
                      [min-height min-height]
                      [stretchable-width stretchable-width]
                      [stretchable-height stretchable-height]])
 → (is-a?/c vertical-pane%)
```

```
 parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                (is-a?/c panel%) (is-a?/c pane%))
 vert-margin : (integer-in 0 1000) = 0
 horiz-margin : (integer-in 0 1000) = 0
 border : (integer-in 0 1000) = 0
 spacing : (integer-in 0 1000) = 0
 alignment : (list/c (one-of/c 'left 'center 'right)
                     (one-of/c 'top 'center 'bottom))
           = '(center top)
 min-width : (integer-in 0 10000) = graphical-minimum-width
 min-height : (integer-in 0 10000) = graphical-minimum-height
 stretchable-width : any/c = #t
 stretchable-height : any/c = #t
```

For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

2.1.50 vertical-panel%

---

vertical-panel% : class?
  superclass: panel%

A vertical panel arranges its subwindows in a single column. See also panel%.

```
(new vertical-panel%
  [parent parent]
 [[style style]
  [enabled enabled]
  [vert-margin vert-margin]
  [horiz-margin horiz-margin]
  [border border]
  [spacing spacing]
  [alignment alignment]
  [min-width min-width]
  [min-height min-height]
  [stretchable-width stretchable-width]
  [stretchable-height stretchable-height]])
→ (is-a?/c vertical-panel%)
 parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                (is-a?/c panel%) (is-a?/c pane%))
 style : (listof (one-of/c 'border 'deleted)) = null
```

```
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 0
  horiz-margin : (integer-in 0 1000) = 0
  border : (integer-in 0 1000) = 0
  spacing : (integer-in 0 1000) = 0
  alignment : (list/c (one-of/c 'left 'center 'right)
                      (one-of/c 'top 'center 'bottom))
            = '(center top)
  min-width : (integer-in 0 10000) = graphical-minimum-width
  min-height : (integer-in 0 10000) = graphical-minimum-height
  stretchable-width : any/c = #t
  stretchable-height : any/c = #t
```

If the 'border style is specified, the window is created with a thin border (only in this case, the client size of the panel may be less than its total size). If *style* includes 'deleted, then the panel is created as hidden, and it does not affect its parent's geometry; the panel can be made active later by calling *parent*'s add-child method.

For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *border*, *spacing*, and *alignment* arguments, see area-container<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send a-vertical-panel set-orientation horizontal?) → void?
  horizontal? : boolean?
```

Sets the orientation of the panel, switching it between the behavior of the vertical-panel% and that of the horizontal-panel%.

```
(send a-vertical-panel get-orientation) → boolean?
```

Initially returns #f, but if set-orientation is called, this method returns whatever the last value passed to it was.

2.1.51 window<%>

```
window<%> : interface?
  implements: area<%>
```

A window<%> object is an area<%> with a graphical representation that can respond to events.

All window<%> classes accept the following named instantiation arguments:

- enabled — default is #t; passed to enable if #f

```
(send a-window accept-drop-files) → boolean?
(send a-window accept-drop-files accept-files?) → void?
  accept-files? : any/c
```

Enables or disables drag-and-drop dropping for the window, or gets the enable state. Dropping is initially disabled. See also on-drop-file.

```
(send a-window client->screen x y) → (integer-in -10000 10000)
                                     (integer-in -10000 10000)
  x : (integer-in -10000 10000)
  y : (integer-in -10000 10000)
```

Converts local window coordinates to screen coordinates.

Under Mac OS X, the screen coordinates start with (0, 0) at the upper left of the menu bar. In contrast, move in top-level-window<%> considers (0, 0) to be below the menu bar. See also get-display-left-top-inset.

```
(send a-window enable enable?) → void?
  enable? : any/c
```

Enables or disables a window so that input events are ignored. (Input events include mouse events, keyboard events, and close-box clicks, but not focus or update events.) When a window is disabled, input events to its children are also ignored.

The enable state of a window can be changed by enabling a parent window, and such changes do not go through this method; use on-superwindow-enable to monitor enable state changes.

If *enable?* is true, the window is enabled, otherwise it is disabled.

```
(send a-window focus) → void?
```

Moves the keyboard focus to the window, relative to its top-level window, if the window ever accepts the keyboard focus. If the focus is in the window's top-level window, then the focus is immediately moved to this window. Otherwise, the focus is not immediately moved, but when the window's top-level window gets the keyboard focus, the focus is delegated to this window.

See also on-focus.

Note that under X, keyboard focus can move to the menu bar when the user is selecting a menu item.

The current keyboard focus window can be changed by the user, and such changes do not go through this method; use on-focus to monitor focus changes.

```
(send a-window get-client-size)
 → (integer-in 0 10000) (integer-in 0 10000)
```

Gets the interior size of the window in pixels. For a container, the interior size is the size available for placing subwindows (including the border margin). For a canvas, this is the visible drawing area.

The client size is returned as two values: width and height (in pixels).

See also reflow-container.

```
(send a-window get-cursor) → (or/c (is-a?/c cursor%) false/c)
```

Returns the window's cursor, or #f if this window's cursor defaults to the parent's cursor. See set-cursor for more information.

```
(send a-window get-handle) → exact-integer?
```

Returns an exact integer representing a handle to the window in the current platform's GUI toolbox. Cast this number from a C `long` to a platform-specific C type:

- Windows: `HWND`
- Mac OS X: `WindowRef` for a top-level-window<%> object, `ControlRef` for other windows
- X: `Widget*`

Some windows may not have a representation in the platform's GUI level, in which case the result of this method is 0.

```
(send a-window get-height) → (integer-in 0 10000)
```

Returns the window's total height (in pixels).

See also reflow-container.

```
(send a-window get-label)
```

```
→ (or/c label-string? (is-a?/c bitmap%)
        (one-of/c 'app 'caution 'stop) false/c)
```

Gets a window's label, if any. Control windows generally display their label in some way. Frames and dialogs display their label as a window title. Panels do not display their label, but the label can be used for identification purposes. Messages, buttons, and check boxes can have bitmap labels (only when they are created with bitmap labels), but all other windows have string labels. In addition, a message label can be an icon symbol `'app`, `'caution`, or `'stop`.

The label string may contain `&`s, which serve as keyboard navigation annotations for controls under Windows and X. The ampersands are not part of the displayed label of a control; instead, ampersands are removed in the displayed label (under all platforms), and any character preceding an ampersand is underlined (Windows and X) indicating that the character is a mnemonic for the control. Double ampersands are converted into a single ampersand (with no displayed underline). See also `on-traverse-char`.

If the window does not have a label, `#f` is returned.

```
(send a-window get-plain-label) → (or/c string false/c)
```

Like `get-label`, except that ampersands in the label are removed. If the window has no label or the window's label is not a string, `#f` is returned.

```
(send a-window get-size)
 → (integer-in 0 10000) (integer-in 0 10000)
```

Gets the current size of the entire window in pixels, not counting horizontal and vertical margins. (Under X, this size does not include a title bar or borders for a frame/dialog.) See also `get-client-size`.

The geometry is returned as two values: width and height (in pixels).

See also `reflow-container`.

```
(send a-window get-width) → (integer-in 0 10000)
```

Returns the window's current total width (in pixels).

See also `reflow-container`.

```
(send a-window get-x) → (integer-in -10000 10000)
```

Returns the position of the window's left edge in its parent's coordinate system.

See also reflow-container.

(send a-window get-y) → (integer-in -10000 10000)

Returns the position of the window's top edge in its parent's coordinate system.

See also reflow-container.

(send a-window has-focus?) → boolean?

Indicates whether the window currently has the keyboard focus. See also on-focus.

(send a-window is-enabled?) → boolean?

Returns #t if the window is enabled when all of its ancestors are enabled, #f otherwise.

(send a-window is-shown?) → boolean?

Indicates whether the window is currently shown or not (when all of its ancestors are also shown).

The result is #t if this window is shown when its ancestors are shown, or #f if this window remains hidden when its ancestors are shown.

(send a-window on-drop-file pathname) → void?
  pathname : path

Called when the user drags a file onto the window. (Under X, drag-and-drop is supported via the XDND protocol.) Drag-and-drop must first be enabled for the window with accept-drop-files.

Under Mac OS X, when the application is running and user double-clicks an application-handled file or drags a file onto the application's icon, the main thread's application file handler is called (see application-file-handler). The default handler calls the on-drop-file method of the most-recently activated frame if drag-and-drop is enabled for that frame, independent of the frame's eventspace (but the method is called in the frame's eventspace's handler thread). When the application is not running, the filenames are provided as command-line arguments.

(send a-window on-focus on?) → void?
  on? : any/c

Specification: Called when a window receives or loses the keyboard focus. If the argument

is #t, the keyboard focus was received, otherwise it was lost.

Note that under X, keyboard focus can move to the menu bar when the user is selecting a menu item.

Default implementation: Does nothing.

```
(send a-window on-move x y) → void?
  x : (integer-in -10000 10000)
  y : (integer-in -10000 10000)
```

Specification: Called when the window is moved. (For windows that are not top-level windows, "moved" means moved relative to the parent's top-left corner.) The new position is provided to the method.

Default implementation: Does nothing.

```
(send a-window on-size width height) → void?
  width : (integer-in 0 10000)
  height : (integer-in 0 10000)
```

Specification: Called when the window is resized. The window's new size (in pixels) is provided to the method. The size values are for the entire window, not just the client area.

Default implementation: Does nothing.

```
(send a-window on-subwindow-char receiver
                                 event)    → boolean?
  receiver : (is-a?/c window<%>)
  event : (is-a?/c key-event%)
```

Specification: Called when this window or a child window receives a keyboard event. The on-subwindow-char method of the receiver's top-level window is called first (see get-top-level-window); if the return value is #f, then the on-subwindow-char method is called for the next child in the path to the receiver, and so on. Finally, if the receiver's on-subwindow-char method returns #f, the event is passed on to the receiver's normal key-handling mechanism.

BEWARE: The default on-subwindow-char in frame% and on-subwindow-char in dialog% methods consume certain keyboard events (e.g., arrow keys, Enter) used for navigating within the window. Because the top-level window gets the first chance to handle the keyboard event, some events never reach the "receiver" child unless the default frame or dialog method is overridden.

The *event* argument is the event that was generated for the *receiver* window.

Default implementation: Returns `#f`.

```
(send a-window on-subwindow-event receiver
                                  event)   → boolean?
  receiver : (is-a?/c window<%>)
  event : (is-a?/c mouse-event%)
```

Specification: Called when this window or a child window receives a mouse event. The `on-subwindow-event` method of the receiver's top-level window is called first (see `get-top-level-window`); if the return value is `#f`, the `on-subwindow-event` method is called for the next child in the path to the receiver, and so on. Finally, if the receiver's `on-subwindow-event` method returns `#f`, the event is passed on to the receiver's normal mouse-handling mechanism.

The *event* argument is the event that was generated for the *receiver* window.

Default implementation: Returns `#f`.

```
(send a-window on-superwindow-enable enabled?) → void?
  enabled? : any/c
```

Called via the event queue whenever the enable state of a window has changed, either through a call to the window's `enable` method, or through the enabling/disabling of one of the window's ancestors. The method's argument indicates whether the window is now enabled or not.

This method is not called when the window is initially created; it is called only after a change from the window's initial enable state. Furthermore, if an enable notification event is queued for the window and it reverts its enabled state before the event is dispatched, then the dispatch is canceled.

If the enable state of a window's ancestor changes while the window is deleted (e.g., because it was removed with `delete-child`), then no enable events are queued for the deleted window. But if the window is later re-activated into an enable state that is different from the window's state when it was de-activated, then an enable event is immediately queued.

```
(send a-window on-superwindow-show shown?) → void?
  shown? : any/c
```

Called via the event queue whenever the visibility of a window has changed, either through a call to the window's `show`, through the showing/hiding of one of the window's ancestors, or through the activating or deactivating of the window or its ancestor in a container (e.g., via `delete-child`). The method's argument indicates whether the window is now visible or not.

This method is not called when the window is initially created; it is called only after a change from the window's initial visibility. Furthermore, if a show notification event is queued for the window and it reverts its visibility before the event is dispatched, then the dispatch is canceled.

```
(send a-window popup-menu menu x y) → void?
  menu : (is-a?/c popup-menu%)
  x : (integer-in 0 10000)
  y : (integer-in 0 10000)
```

Pops up the given popup-menu% object at the specified coordinates (in this window's coordinates), and returns after handling an unspecified number of events; the menu may still be popped up when this method returns. If a menu item is selected from the popup-menu, the callback for the menu item is called. (The eventspace for menu item's callback is the window's eventspace.)

While the menu is popped up, its target is set to the window. See get-popup-target for more information.

The *menu* is popped up within the window at position (*x*, *y*).

```
(send a-window refresh) → void?
```

Enqueues an event to repaint the window.

```
(send a-window screen->client x y) → (integer-in -10000 10000)
                                     (integer-in -10000 10000)
  x : (integer-in -10000 10000)
  y : (integer-in -10000 10000)
```

Converts global coordinates to window local coordinates. See also client->screen for information on screen coordinates.

```
(send a-window set-cursor cursor) → void?
  cursor : (or/c (is-a?/c cursor%) false/c)
```

Sets the window's cursor. Providing #f instead of a cursor value removes the window's cursor.

If a window does not have a cursor, it uses the cursor of its parent. Frames and dialogs start with the standard arrow cursor, and text fields start with an I-beam cursor. All other windows are created without a cursor.

```
(send a-window set-label l) → void?
```

```
  l : label-string?
```

Sets a window's label. The window's natural minimum size might be different after the label is changed, but the window's minimum size is not recomputed.

If the window was not created with a label, or if the window was created with a non-string label, *l* is ignored.

See get-label for more information.

```
(send a-window show show?) → void?
  show? : any/c
```

Shows or hides a window.

The visibility of a window can be changed by the user clicking the window's close box, for example, and such changes do not go through this method; use on-superwindow-show or on-close to monitor visibility changes.

If *show?* is #f, the window is hidden. Otherwise, the window is shown.

2.2 Windowing Functions

2.2.1 Dialogs

These functions get input from the user and/or display messages.

```
(get-file [message
           parent
           directory
           filename
           extension
           style
           filters]) → (or/c path? false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  directory : (or/c path-string? false/c) = #f
  filename : (or/c path-string? false/c) = #f
  extension : (or/c string? false/c) = #f
  style : (listof (one-of/c 'packages 'enter-packages)) = null
  filters : (listof (list/c string? string?)) = '(("Any" "*.*"))
```

Obtains a file pathname from the user via the platform-specific standard (modal) dialog, using *parent* as the parent window if it is specified, and using *message* as a message at the top of the dialog if it is not #f.

The result is #f if the user cancels the dialog, the selected pathname otherwise. The returned pathname may or may not exist, although the style of the dialog is directed towards selecting existing files.

If *directory* is not #f, it is used as the starting directory for the file selector (otherwise the starting directory is chosen automatically in a platform-specific manner, usually based on the current directory and the user's interactions in previous calls to get-file, put-file, etc.). If *filename* is not #f, it is used as the default filename when appropriate, and it should *not* contain a directory path prefix.

Under Windows, if *extension* is not #f, the returned path will use the extension if the user does not supply one; the *extension* string should not contain a period. The extension is ignored on other platforms.

The *style* list can contain 'common, a platform-independent version of the dialog is used instead of a native dialog. Under Mac OS X, if the *style* list contains 'packages, a user is allowed to select a package directory, which is a directory with a special suffix (e.g., ".app") that the Finder normally displays like a file. If the list contains 'enter-packages, a user is allowed to select a file within a package directory. If the list contains both 'packages and 'enter-packages, the former is ignored.

Under Windows and X, *filters* determines a set of filters from which the user can choose in the dialog. Each element of the *filters* list contains two strings: a description of the filter as seen by the user, and a filter pattern matched against file names.

See also path-dialog%.

```
(get-file-list [message
                parent
                directory
                filename
                extension
                style
                filters]) → (or/c (listof path?) false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  directory : (or/c path-string? false/c) = #f
  filename : (or/c path-string? false/c) = #f
  extension : (or/c string? false/c) = #f
  style : null? = null
  filters : (listof (list/c string? string?)) = '(("Any" "*.*"))
```

Like get-file, except that the user can select multiple files, and the result is either a list of file paths of #f.

```
(put-file [message
           parent
           directory
           filename
           extension
           style
           filters]) → (or/c path? false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  directory : (or/c path-string? false/c) = #f
  filename : (or/c path-string? false/c) = #f
  extension : (or/c string? false/c) = #f
  style : (listof (one-of/c 'packages 'enter-packages)) = null
  filters : (listof (list/c string? string?)) = '(("Any" "*.*"))
```

Obtains a file pathname from the user via the platform-specific standard (modal) dialog, using *parent* as the parent window if it is specified, and using *message* as a message at the top of the dialog if it is not #f.

The result is #f if the user cancels the dialog, the selected pathname otherwise. The returned pathname may or may not exist, although the style of the dialog is directed towards creating a new file.

If *directory* is not #f, it is used as the starting directory for the file selector (otherwise the starting directory is chosen automatically in a platform-specific manner, usually based on the current directory and the user's interactions in previous calls to get-file, put-file, etc.). If *filename* is not #f, it is used as the default filename when appropriate, and it should *not* contain a directory path prefix.

Under Windows, if *extension* is not #f, the returned path will get a default extension if the user does not supply one. If *extension* is the empty string, then the extension is derived from the user's *filters* choice if the corresponding pattern is of the form (string-append "*." *extension*); if the pattern is "*.*", then no default extension is added. Finally, if *extension* is any string other than the empty string, *extension* is used as the default extension when the user's *filters* choice has the pattern "*.*". Meanwhile, the *filters* argument has the same format and auxiliary role as for get-file. In particular, if the only pattern in *filters* is (string-append "*." *extension*), then the result pathname is guaranteed to have an extension mapping *extension*.

Under Mac OS X 10.5 and later, if *extension* is not #f, the returned path will get a default extension if the user does not supply one. If *filters* contains as "*.*" pattern, then the user can supply any extension that is recognized by the system; otherwise, the extension

on the returned path will be either *extension* or *other-extension* for any (string-append "*." *other-extension*) pattern in *filters*. In particular, if the only pattern in *filters* is empty or contains only (string-append "*." *extension*), then the result pathname is guaranteed to have an extension mapping *extension*.

Under Mac OS X versions before 10.5, the returned path will get a default extension only if *extension* is not #f and *filters* contains only (string-append "*." *extension*).

The *extension* argument is ignored under X, and *filters* can be used to specify glob-patterns.

The *style* list is treated as for get-file.

See also path-dialog%.

```
(get-directory [message
               parent
               directory
               style])    → (or/c path false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  directory : (or/c path? false/c) = #f
  style : (listof (one-of/c 'enter-packages)) = null
```

Obtains a directory pathname from the user via the platform-specific standard (modal) dialog, using *parent* as the parent window if it is specified.

If *directory* is not #f, it is used on some platforms as the starting directory for the directory selector (otherwise the starting directory is chosen automatically in a platform-specific manner, usually based on the current directory and the user's interactions in previous calls to get-file, put-file, etc.).

The *style* argument is treated as for get-file, except that only 'common or 'enter-packages can be specified. The latter matters only under Mac OS X, where 'enter-packages enables the user to select package directory or a directory within a package. A package is a directory with a special suffix (e.g., ".app") that the Finder normally displays like a file.

See also path-dialog%.

```
(message-box title message [parent style])
 → (one-of/c 'ok 'cancel 'yes 'no)
  title : label-string?
  message : string
```

```
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  style : (listof (one-of/c 'ok 'ok-cancel 'yes-no 'caution 'stop))
        = '(ok)
```

See also message-box/custom.

Displays a message to the user in a (modal) dialog, using *parent* as the parent window if it is specified. The dialog's title is *title*. The *message* string can be arbitrarily long, and can contain explicit linefeeds or carriage returns for breaking lines.

The style must include exactly one of the following:

- 'ok — the dialog only has an OK button and always returns 'ok.
- 'ok-cancel — the message dialog has Cancel and OK buttons. If the user clicks Cancel, the result is 'cancel, otherwise the result is 'ok.
- 'yes-no — the message dialog has Yes and No buttons. If the user clicks Yes, the result is 'yes, otherwise the result is 'no. Note: instead of a Yes/No dialog, best-practice GUI design is to use message-box/custom and give the buttons meaningful labels, so that the user does not have to read the message text carefully to make a selection.

In addition, *style* can contain 'caution to make the dialog use a caution icon instead of the application (or generic "info") icon. Alternately, it can contain 'stop to make the dialog use a stop icon. If *style* contains both 'caution and 'stop, then 'caution is ignored.

The class that implements the dialog provides a get-message method that takes no arguments and returns the text of the message as a string. (The dialog is accessible through the get-top-level-windows function.)

```
(message-box/custom  title
                     message
                     button1-label
                     button2-label
                     button3-label
                    [parent
                     style
                     close-result])
 → (one-of/c 1 2 3 close-result)
  title : label-string?
  message : string
  button1-label : (or/c label-string? (is-a?/c bitmap%) false/c)
  button2-label : (or/c label-string? (is-a?/c bitmap%) false/c)
  button3-label : (or/c label-string? (is-a?/c bitmap%) false/c)
```

```
parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
       = #f
style : (listof (one-of/c 'stop 'caution 'number-order
                          'disallow-close 'no-default
                          'default=1 'default=2 'default=3))
      = '(no-default)
close-result : any/c = #f
```

Displays a message to the user in a (modal) dialog, using *parent* as the parent window if it is specified. The dialog's title is *title*. The *message* string can be arbitrarily long, and can contain explicit linefeeds or carriage returns for breaking lines.

The dialog contains up to three buttons for the user to click. The buttons have the labels *button1-label*, *button2-label*, and *button3-label*, where #f for a label indicates that the button should be hidden.

If the user clicks the button labelled *button1-label*, a 1 is returned, and so on for 2 and 3. If the user closes the dialog some other way—which is only allowed when *style* does not contain 'disallow-close—then the result is the value of *close-result*. For example, the user can usually close a dialog by typing an Escape. Often, 2 is an appropriate value for *close-result*, especially when Button 2 is a Cancel button.

If *style* does not include 'number-order, the order of the buttons is platform-specific, and labels should be assigned to the buttons based on their role:

- Button 1 is the normal action, and it is usually the default button. For example, if the dialog has an OK button, it is this one. Under Windows, this button is leftmost; under X and Mac OS X, it is rightmost. (See also system-position-ok-before-cancel?.) Use this button for dialogs that contain only one button.
- Button 2 is next to Button 1, and it often plays the role of Cancel (even when the default action is to cancel, such as when confirming a file replacement).
- Button 3 tends to be separated from the other two (under Mac OS X, it is left-aligned in the dialog). Use this button only for three-button dialogs.

Despite the above guidelines, any combination of visible buttons is allowed in the dialog.

If *style* includes 'number-order, then the buttons are displayed in the dialog left-to-right with equal spacing between all buttons, though aligned within the dialog (centered or right-aligned) in a platform-specific manner. Use 'number-order sparingly.

The *style* list must contain exactly one of 'default=1, 'default=2, 'default=3, and 'no-default to determine which button (if any) is the default. The default button is "clicked" when the user types Return. If 'default=n is supplied but button n has no label, then it is equivalent to 'no-default.

In addition, *style* can contain 'caution to make the dialog use a caution icon instead of the application (or generic "info") icon. Alternately, it can contain 'stop to make the dialog use a stop icon. If *style* contains both 'caution and 'stop, then 'caution is ignored.

The class that implements the dialog provides a get-message method that takes no arguments and returns the text of the message as a string. (The dialog is accessible through the get-top-level-windows function.)

```
(message+check-box  title
                    message
                    check-label
                   [parent
                    style])
 → (one-of/c 'ok 'cancel 'yes 'no)
  title : label-string?
  message : string
  check-label : label-string?
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  style : (listof (one-of/c 'ok 'ok-cancel 'yes-no    = '(ok)
                            'caution 'stop 'checked))
```

See also message+check-box/custom.

Like message-box, except that

- the dialog contains a check box whose label is *check-label*;
- the result is two values: the message-box result, and a boolean indicating whether the box was checked; and
- *style* can contain 'checked to indicate that the check box should be initially checked.

```
(message+check-box/custom  title
                           message
                           check-label
                           button1-label
                           button2-label
                           button3-label
                          [parent
                           style
                           close-result])
 → (one-of/c 1 2 3 close-result)
  title : label-string?
```

```
  message : string
  check-label : label-string?
  button1-label : (or/c label-string? (is-a?/c bitmap%) false/c)
  button2-label : (or/c label-string? (is-a?/c bitmap%) false/c)
  button3-label : (or/c label-string? (is-a?/c bitmap%) false/c)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  style : (listof (one-of/c 'stop 'caution 'number-order
                            'disallow-close 'no-default
                            'default=1 'default=2 'default=3))
        = '(no-default)
  close-result : any/c = #f
```

Like message-box/custom, except that

- the dialog contains a check box whose label is *check-label*;
- the result is two values: the message-box result, and a boolean indicating whether the box was checked; and
- *style* can contain 'checked to indicate that the check box should be initially checked.

```
(get-text-from-user  title
                     message
                    [parent
                     init-val
                     style])   → (or/c string? false/c)
  title : string
  message : (or/c string? false/c)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  init-val : string? = ""
  style : (listof (one-of/c 'password)) = null
```

Gets a text string from the user via a modal dialog, using *parent* as the parent window if it is specified. The dialog's title is *title*. The dialog's text field is labelled with *message* and initialized to *init-val* (but *init-val* does not determine the size of the dialog).

The result is #f if the user cancels the dialog, the user-provided string otherwise.

If *style* includes 'password, the dialog's text field draws each character of its content using a generic symbol, instead of the actual character.

```
(get-choices-from-user  title
                        message
                        choices
                       [parent
                        init-choices
                        style])
 → (or/c (listof exact-nonnegative-integer?) false/c)
  title : string
  message : (or/c string? false/c)
  choices : (listof string?)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  init-choices : (listof exact-nonnegative-integer?) = null
  style : (listof (one-of/c 'single 'multiple 'extended))
        = '(single)
```

Gets a list box selection from the user via a modal dialog, using *parent* as the parent window if it is specified. The dialog's title is *title*. The dialog's list box is labelled with *message* and initialized by selecting the items in *init-choices*.

The style must contain exactly one of `'single`, `'multiple`, or `'extended`. The styles have the same meaning as for creating a `list-box%` object. (For the single-selection style, only the last selection in *init-choices* matters.)

The result is `#f` if the user cancels the dialog, the list of selections otherwise.

```
(get-color-from-user [message
                      parent
                      init-color
                      style])
 → (or/c (is-a?/c color%) false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  init-color : (or/c (is-a?/c color%) false/c) = #f
  style : null? = null
```

Lets the user select a color though the platform-specific (modal) dialog, using *parent* as the parent window if it is specified. The *message* string is displayed as a prompt in the dialog if possible. If *init-color* is provided, the dialog is initialized to the given color.

The *style* argument is provided for future extensions. Currently, *style* must be the empty list.

The result is `#f` if the user cancels the dialog, the selected color otherwise.

```
(get-font-from-user [message
                     parent
                     init-font
                     style])    → (or/c (is-a?/c font%) false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  init-font : (or/c (is-a?/c font%) false/c) = #f
  style : null? = null
```

Lets the user select a font though the platform-specific (modal) dialog, using *parent* as the parent window if it is specified. The *message* string is displayed as a prompt in the dialog if possible. If *init-font* is provided, the dialog is initialized to the given font.

The *style* argument is provided for future extensions. Currently, *style* must be the empty list.

The result is `#f` if the user cancels the dialog, the selected font otherwise.

```
(get-ps-setup-from-user [message
                         parent
                         init-setup
                         style])
 → (or/c (is-a?/c ps-setup%) false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  init-setup : (or/c (is-a?/c ps-setup%) false/c) = #f
  style : null? = null
```

Lets the user select a PostScript configuration though a (modal) dialog, using *parent* as the parent window if it is specified. The *message* string is displayed as a prompt in the dialog. If *init-setup* is provided, the dialog is initialized to the given configuration, otherwise the current configuration from `current-ps-setup` is used.

The *style* argument is provided for future extensions. Currently, *style* must be the empty list.

The result is `#f` if the user cancels the dialog, , a `ps-setup%` object that encapsulates the selected PostScript configuration otherwise.

```
(get-page-setup-from-user [message
                           parent
                           init-setup
                           style])
 → (or/c (is-a?/c ps-setup%) false/c)
  message : (or/c string? false/c) = #f
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  init-setup : (or/c (is-a?/c ps-setup%) false/c) = #f
  style : null? = null
```

Like get-ps-setup-from-user, but the dialog configures page layout for native printing with printer-dc%. A dialog is shown only if can-get-page-setup-from-user? returns #t, otherwise no dialog is shown and the result is #f.

The *parent* argument is used as the parent window for a dialog if it is specified. The *message* string might be displayed as a prompt in the dialog. If *init-setup* is provided, the dialog is initialized to the given configuration, otherwise the current configuration from current-ps-setup is used.

The *style* argument is provided for future extensions. Currently, *style* must be the empty list.

The result is #f if the user cancels the dialog, a ps-setup% object that encapsulates the selected configuration otherwise.

```
(can-get-page-setup-from-user?) → boolean?
```

Returns #t if the current platform (Mac OS X) supports a page-layout dialog for use with printer-dc% printing, and if the page-layout dialog is different from the print-job dialog that is automatically shown when a printer-dc% is created. Returns #f if no separate page-layout dialog is needed (Windows and Unix).

2.2.2 Eventspaces

```
(make-eventspace) → eventspace
```

Creates and returns a new eventspace value. The new eventspace is created as a child of the current eventspace. The eventspace is used by making it the current eventspace with the current-eventspace parameter.

See §1.1.4 “Event Dispatching and Eventspaces” for more information about eventspaces.

```
(current-eventspace) → eventspace?
(current-eventspace e) → void?
  e : eventspace?
```

A parameter (see §10.3.2 “Parameters”) that determines the current eventspace.

See §1.1.4 “Event Dispatching and Eventspaces” for more information about eventspaces.

```
(eventspace? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an eventspace value or `#f` otherwise.

See §1.1.4 “Event Dispatching and Eventspaces” for more information about eventspaces.

```
(event-dispatch-handler) → (eventspace? . -> . any)
(event-dispatch-handler handler) → void?
  handler : (eventspace? . -> . any)
```

A parameter (see §10.3.2 “Parameters”) that determines the current event dispatch handler. The event dispatch handler is called by an eventspace’s handler thread for every queue-based event to be processed in the eventspace. The only argument to the handler is the eventspace in which an event should be dispatched. The event dispatch handler gives the programmer control over the timing of event dispatching, but not the order in which events are dispatched within a single eventspace.

An event dispatch handler must ultimately call the primitive event dispatch handler. If an event dispatch handler returns without calling the primitive handler, then the primitive handler is called directly by the eventspace handler thread.

```
(check-for-break) → boolean?
```

Inspects the event queue of the current eventspace, searching for a Shift-Ctl-C (X, Windows) or Cmd-. (Mac OS X) key combination. Returns `#t` if such an event was found (and the event is dequeued) or `#f` otherwise.

```
(get-top-level-windows)
 → (listof (or/c (is-a?/c frame%) (is-a?/c dialog%)))
```

Returns a list of visible top-level frames and dialogs in the current eventspace.

```
(get-top-level-focus-window)
```

```
 → (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
```

Returns the top level window in the current eventspace that has the keyboard focus (or contains the window with the keyboard focus), or `#f` if no window in the current eventspace has the focus.

```
(get-top-level-edit-target-window)
 → (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
```

Returns the top level window in the current eventspace that is visible and most recently had the keyboard focus (or contains the window that had the keyboard focus), or `#f` if there is no visible window in the current eventspace.

```
(special-control-key on?) → void?
  on? : any/c
(special-control-key) → boolean?
```

Enables or disables special Control key handling (Mac OS X). When Control is treated as a special key, the system's key-mapper is called without Control for keyboard translations. For some languages, Control key presses must be seen by the system translation, so this mode should be turned off, but the default is on.

If *on?* is provided and `#f`, Control is passed to the system translation as normal. This setting affects all windows and eventspaces.

If no argument is provided, the result is `#t` if Control is currently treated specially, `#f` otherwise.

```
(special-option-key on?) → void?
  on? : any/c
(special-option-key) → boolean?
```

Enables or disables special Option key handling (Mac OS X). When Option is treated as a special key, the system's key-mapper is called without Option for keyboard translations. By default, Option is not special.

If *on?* is provided `#f`, Option is passed to the system translation as normal. This setting affects all windows and eventspaces.

If no argument is provided, the result is `#t` if Option is currently treated specially, `#f` otherwise.

```
(queue-callback callback [high-priority?]) → void?
  callback : (-> any)
```

high-priority? : any/c = #t

Installs a procedure to be called via the current eventspace's event queue. The procedure is called once in the same way and under the same restrictions that a callback is invoked to handle a method.

A second (optional) boolean argument indicates whether the callback has a high or low priority in the event queue. See §1.1.4 "Event Dispatching and Eventspaces" for information about the priority of events.

(yield) → boolean?
(yield v) → any/c
  v : (or/c (one-of/c 'wait) evt?)

Yields control to event dispatching. See §1.1.4 "Event Dispatching and Eventspaces" for details.

A handler procedure invoked by the system during a call to yield can itself call yield, creating an additional level of nested (but single-threaded) event handling.

See also sleep/yield .

If no argument is provided, yield dispatches an unspecified number of events, but only if the current thread is the current eventspace's handler thread (otherwise, there is no effect). The result is #t if any events may have been handled, #f otherwise.

If v is 'wait, and yield is called in the handler thread of an eventspace, then yield starts processing events in that eventspace until

- no top-level windows in the eventspace are visible;
- no timers in the eventspace are running;
- no callbacks are queued in the eventspace; and
- no menu-bar% has been created for the eventspace with 'root (i.e., creating a 'root menu bar prevents an eventspace from ever unblocking).

When called in a non-handler thread, yield returns immediately. In either case, the result is #t.

Evaluating (yield 'wait) is thus similar to (yield (current-eventspace)), except that it is sensitive to whether the current thread is a handler thread, instead of the value of the current-eventspace parameter.

If v is an event in Racket's sense (not to be confused with a GUI event), yield blocks on v in the same way as sync, except that it may start a sync on v multiple times (but it will

complete a sync on v at most one time). If the current thread is the current eventspace's handler thread, events are dispatched until a v sync succeeds on an event boundary. For other threads, calling yield with a Racket event is equivalent to calling sync. In either case, the result is the same that of sync; however, if a wrapper procedure is associated with v via handle-evt, it is not called in tail position with respect to the yield.

Always use (yield v) instead of a busy-wait loop.

(sleep/yield *secs*) → void?
  secs : (and/c real? (not/c negative?))

Blocks for at least the specified number of seconds, handling events meanwhile if the current thread is the current eventspace's handler thread (otherwise, sleep/yield is equivalent to sleep).

(eventspace-shutdown? *e*) → boolean?
  e : eventspace

Returns #t if the given eventspace has been shut down by its custodian, #f otherwise. Attempting to create a new window, timer, or explicitly queued event in a shut-down eventspace raises the exn:misc exception.

Attempting to use certain methods of windows and timers in a shut-down eventspace also raises the exn:misc exception, but the get-top-level-window in area<%> and get-eventspace in top-level-window<%> methods work even after the area's eventspace is shut down.

(eventspace-handler-thread *e*) → (or/c thread false/c)
  e : eventspace

Returns the handler thread of the given eventspace. If the handler thread has terminated (e.g., because the eventspace was shut down), the result is #f.

2.2.3 System Menus

(current-eventspace-has-standard-menus?) → boolean?

Returns #t for Mac OS X when the current eventspace is the initial one, since that eventspace is the target for the standard application menus. For any other system or eventspace, the result is #f.

This procedure is intended for use in deciding whether to include a Quit, About, and

Preferences menu item in a frame's menu. Under Mac OS X, the application Quit menu triggers a call to a frame's `on-exit` method, the About menu item is controlled by `application-about-handler`, and the Preferences menu item is controlled by `application-preferences-handler`.

```
(current-eventspace-has-menu-root?) → boolean?
```

Returns `#t` for Mac OS X when the current eventspace is the initial one, since that eventspace can supply a menu bar to be active when no frame is visible. For any other system or eventspace, the result is `#f`.

This procedure is intended for use in deciding whether to create a `menu-bar%` instance with `'root` as its parent.

```
(application-about-handler) → (-> any)
(application-about-handler handler-thunk) → void?
  handler-thunk : (-> any)
```

When the current eventspace is the initial eventspace, this procedure retrieves or installs a thunk that is called when the user selects the application About menu item in Mac OS X. The thunk is always called in the initial eventspace's handler thread (as a callback).

The default handler displays a generic Racket dialog.

If the current eventspace is not the initial eventspace, this procedure returns `void` (when called with zero arguments) or has no effect (when called with a handler).

```
(application-preferences-handler) → (or/c (-> any) false/c)
(application-preferences-handler handler-thunk) → void?
  handler-thunk : (or/c (-> any) false/c)
```

When the current eventspace is the initial eventspace, this procedure retrieves or installs a thunk that is called when the user selects the application Preferences menu item in Mac OS X. The thunk is always called in the initial eventspace's handler thread (as a callback). If the handler is set to `#f`, the Preferences item is disabled.

The default handler is `#f`.

If the current eventspace is not the initial eventspace, this procedure returns `void` (when called with zero arguments) or has no effect (when called with a handler).

```
(application-quit-handler) → (-> any)
(application-quit-handler handler-thunk) → void?
  handler-thunk : (-> any)
```

When the current eventspace is the initial eventspace, this procedure retrieves or installs a thunk that is called when the user requests that the application quit (e.g., through the Quit menu item in Mac OS X, or when shutting down the machine in Windows). The thunk is always called in the initial eventspace's handler thread (as a callback). If the result of the thunk is `#f`, then the operating system is explicitly notified that the application does not intend to quit (under Windows).

The default handler queues a call to the `can-exit?` method of the most recently active frame in the initial eventspace (and then calls the frame's `on-exit` method if the result is true). The result is `#t` if the eventspace is left with no open frames after `on-exit` returns, `#f` otherwise.

If the current eventspace is not the initial eventspace, this procedure returns `void` (when called with zero arguments) or has no effect (when called with a handler).

```
(application-file-handler) → (path? . -> . any)
(application-file-handler handler-proc) → void?
  handler-proc : (path? . -> . any)
```

When the current eventspace is the initial eventspace, this procedure retrieves or installs a procedure that is called under Mac OS X and Windows when the application is running and user double-clicks an application-handled file or drags a file onto the application's icon. The procedure is always called in the initial eventspace's handler thread (as a callback), and the argument is a filename.

The default handler queues a callback to the `on-drop-file` method of the most-recently activated frame in the main eventspace (see `get-top-level-edit-target-window`), if drag-and-drop is enabled for that frame.

When the application is *not* running and user double-clicks an application-handled file or drags a file onto the application's icon, the filename is provided as a command-line argument to the application.

If the current eventspace is not the initial eventspace, this procedure returns `void` (when called with zero arguments) or has no effect (when called with a handler).

2.2.4 Miscellaneous

```
(begin-busy-cursor) → void?
```

Changes the cursor to a watch cursor for all windows in the current eventspace. Use `end-busy-cursor` to revert the cursor back to its previous state. Calls to `begin-busy-cursor` and `end-busy-cursor` can be nested arbitrarily.

The cursor installed by begin-busy-cursor overrides any window-specific cursors installed with set-cursor.

See also is-busy?.

```
(bell) → void?
```

Rings the system bell.

```
(end-busy-cursor) → void?
```

See begin-busy-cursor.

```
(file-creator-and-type filename
                       creator-string
                       type-bytes)    → void?
  filename : path
  creator-string : (lambda (s) (and (bytes? s)
                                    (= 4 (bytes-length s))))
  type-bytes : (lambda (s) (and (bytes? s)
                                (= 4 (bytes-length s))))
(file-creator-and-type filename)
 → (lambda (s) (and (bytes? s)
                    (= 4 (bytes-length s))))
   (lambda (s) (and (bytes? s)
                    (= 4 (bytes-length s))))
  filename : path
```

Gets or sets the creator and type of a file in Mac OS X.

The get operation always returns #"????" and #"????" for Unix or Windows. The set operation has no effect under Unix or Windows.

```
(find-graphical-system-path what) → (or/c path? false/c)
  what : (one-of/c 'init-file 'setup-file 'x-display)
```

Finds a platform-specific (and possibly user- or machine-specific) standard filename or directory. See also find-system-path.

The result depends on *what*, and a #f result is only possible when *what* is 'x-display:

- 'init-file returns the ,path to the user-specific initialization file (containing Racket code). The directory part of the path is the same path as returned for 'init-dir by Racket's find-system-path. The file name is platform-specific:

  - Unix and Mac OS X: `".gracketrc"`
  - Windows: `"racketrc.rktl"`

- `'setup-file` returns the path to the file containing resources used by `get-resource`; obsolete.

- `'x-display` returns a "path" whose string identifies the X display if specified by either the `-display` flag or the `DISPLAY` environment variable when GRacket starts under X. For other platforms, or when neither `-display` nor `DISPLAY` was specified, the result is `#f`.

```
(get-default-shortcut-prefix)
 → (listof (one-of/c 'alt 'cmd 'meta 'ctl 'shift 'option))
```

Returns an immutable list specifying the default prefix for menu shortcuts. See also `get-shortcut-prefix` in `selectable-menu-item<%>`.

Under Windows, the default is `'(ctl)`. Under Mac OS X, the default is `'(cmd)`. Under X, the default is normally `'(ctl)`, but the default can be changed through the `'GRacket:defaultMenuPrefix` preference low-level preference (see §3.1 "Preferences").

```
(get-panel-background) → (is-a?/c color%)
```

Returns the background color of a panel (usually some shade of gray) for the current platform.

```
(get-resource section entry value [file]) → boolean?
  section : string?
  entry : string?
  value : (box/c (or/c string? exact-integer?))
  file : (or/c path? false/c) = #f
```

Gets a resource value from the resource database. The resource value is keyed on the combination of *section* and *entry*. The return value is `#t` if a value is found, `#f` if it is not. The type of the value initially in the *value* box determines the way that the resource is interpreted, and *value* is filled with a new value of the same type if one is found.

If *file* is `#f`, platform-specific resource files are read, as determined by `find-graphical-system-path` with `'setup-file`. (Under X, when *file* is `#f`, the user's `".Xdefaults"` file is also read, or the file specified by the `"XENVIRONMENT"` environment variable.)

The format of a resource entry depends on the platform. Windows resources use the standard `".ini"` format. X and Mac OS X resources use the standard X resource format, where each entry consists of a *section.entry* resource name, a colon, and the resource value, terminated by a newline. Section and entry names are case-sensitive.

Under Windows, if *section* is one of the following strings, then *file* is ignored, and *entry* is used as a resource path:

- "HKEY_CLASSES_ROOT"
- "HKEY_CURRENT_CONFIG"
- "HKEY_CURRENT_USER"
- "HKEY_LOCAL_MACHINE"
- "HKEY_USERS"

In that case, the *entry* argument is parsed as a resource entry path, followed by a backslash, followed by a value name. To get the "default" value for an entry, use the empty name. For example, the following expression gets a command line for starting a browser:

```
(let ([b (box "")])
  (get-resource "HKEY_CLASSES_ROOT"
                "htmlfile\\shell\\open\\command\\" b)
  (unbox b))
```

See also write-resource.

```
(get-window-text-extent string
                        font
                       [combine?]) → exact-nonnegative-integer?
                                     exact-nonnegative-integer?
  string : string
  font : (is-a?/c font%)
  combine? : any/c = #f
```

Returns the pixel size of a string drawn as a window's label or value when drawn with the given font. The optional *combine?* argument is as for get-text-extent in dc<%>.

See also get-text-extent in dc<%>.

```
(graphical-read-eval-print-loop [eval-eventspace
                                 redirect-ports?]) → void?
  eval-eventspace : eventspace = #f
  redirect-ports? : any/c = (not eval-eventspace)
```

Similar to read-eval-print-loop, except that none of read-eval-print-loop's configuration parameters are used (such as current-read) and the interaction occurs in a GUI window instead of using the current input and output ports.

Expressions entered into the graphical read-eval-print loop can be evaluated in an eventspace (and thread) that is distinct from the one implementing the graphical-read-eval-print-loop window (i.e., the current eventspace when graphical-read-eval-print-loop is called).

If no eventspace is provided, or if #f is provided, an evaluation eventspace is created using (make-eventspace) with a new custodian; the eventspace and its threads are be shut down when the user closes the graphical-read-eval-print-loop window. If an eventspace is provided, closing the window performs no shut-down actions on eventspace.

When *redirect-ports?* is true, the following parameters are initialized in the created eventspace's handler thread:

- current-output-port — writes to the frame
- current-error-port — writes to the frame
- current-input-port — always returns eof

The keymap for the read-eval-print loop's editor is initialized by calling the current keymap initializer procedure, which is determined by the current-text-keymap-initializer parameter.

(textual-read-eval-print-loop) → void?

Similar to read-eval-print-loop, except that evaluation uses a newly created eventspace.

The current-prompt-read parameter is used in the current thread to read input. The result is queued for evaluation and printing in the created eventspace's handler thread, which uses current-eval and current-print. After printing completes for an interaction result, the next expression in read in the original thread, and so on.

If an exn:break exception is raised in the original thread during reading, it aborts the current call to (current-read) and a new one is started. If an exn:break exception is raised in the original thread while waiting for an interaction to complete, a break is sent (via break-thread) to the created eventspace's handler thread.

(hide-cursor-until-moved) → void?

Hides the cursor until the user moves the mouse or clicks the mouse button. (For some platforms, the cursor is not hidden if it is over a window in a different eventspace or application.)

(is-busy?) → boolean?

Returns #t if a busy cursor has been installed with begin-busy-cursor and not removed

with end-busy-cursor.

(label->plain-label *label*) → string?
  label : string

Strips shortcut ampersands from *label*, removes parenthesized ampersand–character combinations along with any surrounding space, and removes anything after a tab. Overall, it returns the label as it would appear on a button on a platform without support for mnemonics.

(make-gui-empty-namespace) → namespace?

Like make-base-empty-namespace, but with racket/class and racket/gui/base also attached to the result namespace.

(make-gui-namespace) → namespace?

Like make-base-namespace, but with racket/class and racket/gui/base also required into the top-level environment of the result namespace.

(play-sound *filename async?*) → boolean?
  filename : path-string?
  async? : any/c

Plays a sound file. If *async?* is false, the function does not return until the sound completes. Otherwise, it returns immediately. The result is #t if the sound plays successfully, #f otherwise.

Under Windows, only ".wav" files are supported.

Under X, the function invokes an external sound-playing program; looking for a few known programs (aplay, play, esdplay, sndfile-play, audioplay). In addition, a play command can be defined through the 'GRacket:playcmd preference preference (see §3.1 "Preferences"). The preference can hold a program name, or a format string containing a single ~a where the filename should be substituted—and used as a shell command. (Don't use ~s, since the string that is used with the format string will be properly quoted and wrapped in double quotes.) A plain command name is usually better since execution is faster. The command's output is discarded, unless it returns an error code—in this case the last part of the error output is shown.

Under Mac OS X, Quicktime is used to play sounds; most sound formats (.wav, .aiff, .mp3) are supported in recent versions of Quicktime. In order to play .wav files, Quicktime 3.0 (compatible with OS 7.5 and up) is required.

```
(send-event  receiver-bytes
             event-class-bytes
             event-id-bytes
            [direct-arg-v
             argument-list])    → any/c
  receiver-bytes : (lambda (s) (and (bytes? s)
                                    (= 4 (bytes-length s))))
  event-class-bytes : (lambda (s) (and (bytes? s)
                                       (= 4 (bytes-length s))))
  event-id-bytes : (lambda (s) (and (bytes? s)
                                    (= 4 (bytes-length s))))
  direct-arg-v : any/c = (void)
  argument-list : list? = null
```

Sends an AppleEvent or raises `exn:fail:unsupported`.

The *receiver-bytes*, *event-class-bytes*, and *event-id-bytes* arguments specify the signature of the receiving application, the class of the AppleEvent, and the ID of the AppleEvent.

The *direct-arg-v* value is converted (see below) and passed as the main argument of the event; if `direct-argument-v` is `#<void>`, no main argument is sent in the event. The *argument-list* argument is a list of two-element lists containing a typestring and value; each typestring is used ad the keyword name of an AppleEvent argument for the associated converted value.

The following types of Racket values can be converted to AppleEvent values passed to the receiver:

`#t` or `#f`	→	Boolean
small integer	→	Long Integer
inexact real number	→	Double
string	→	Characters
list of convertible values	→	List of converted values
`#(file pathname)`	→	Alias (file exists) or FSSpec (does not exist)
`#(record (typestring v) ...)`	→	Record of keyword-tagged values

If other types of values are passed to `send-event` for conversion, the `exn:fail:unsupported` exception is raised.

The `send-event` procedure does not return until the receiver of the AppleEvent replies. The result of `send-event` is the reverse-converted reply value (see below), or the `exn:fail` exception is raised if there is an error. If there is no error or return value, `send-event` returns `#<void>`.

The following types of AppleEvent values can be reverse-converted into a Racket value

returned by send-event:

Boolean	→	#t or #f
Signed Integer	→	integer
Float, Double, or Extended	→	inexact real number
Characters	→	string
List of reverse-convertible values	→	list of reverse-converted values
Alias or FSSpec	→	#(file pathname)
Record of keyword-tagged values	→	#(record (typestring v) ...)

If the AppleEvent reply contains a value that cannot be reverse-converted, the exn:fail exception is raised.

```
(send-message-to-window x y message) → any/c
  x : (integer-in -10000 10000)
  y : (integer-in -10000 10000)
  message : any/c
```

Finds the frontmost top-level window at (*x*, *y*) in global coordinates. If a window is there, this function calls the window's on-message method, providing *message* as the method's argument; the result of the function call is the result returned by the method. If no Racket window is at the given coordinates, or if it is covered by a non-Racket window at (*x*, *y*), #f is returned.

```
(system-position-ok-before-cancel?) → boolean?
```

Returns #t under Windows—indicating that a dialog with OK and Cancel buttons should place the OK button on to left of the Cancel button—and returns #f under Mac OS X and X.

```
the-clipboard : (is-a?/c clipboard<%>)
```

See clipboard<%>.

```
the-x-selection-clipboard : (is-a?/c clipboard<%>)
```

See clipboard<%>.

```
(write-resource section entry value [file]) → boolean?
  section : string?
  entry : string?
  value : (or/c string? exact-integer?)
  file : (or/c path-string? false/c) = #f
```

Writes a resource value to the specified resource database. The resource value is keyed on the combination of *section* and *entry*, with the same special handling of *entry* for under Windows as for get-resource.

If *file* is #f, the platform-specific resource database is read, as determined by find-graphical-system-path with 'setup-file.

The return value is #t if the write succeeds, #f otherwise. (A failure indicates that the resource file cannot be written.)

If *value* is an integer outside a platform-specific range, an exn:fail:contract exception is raised.

See also get-resource.

(label-string? *v*) → boolean?
  v : any/c

Returns #t if *v* is a string whose length is less than or equal to 200.

(key-code-symbol? *v*) → boolean?
  v : any/c

Returns #t if the argument is a symbol that can be returned by key-event%'s method get-key-code.

2.3 Drawing Classes

2.3.1 bitmap%

---

bitmap% : class?
  superclass: object%

A bitmap% object is a pixel-based image, either monochrome or color.

Sometimes, a bitmap object creation fails in a low-level manner. In that case, the ok? method returns #f, and the bitmap cannot be supplied to methods that consume or operate on bitmaps (otherwise, an exn:fail:contract exception is raised).

(make-object bitmap% *width*
                     height
                    [*monochrome?*]) → (is-a?/c bitmap%)

```
  width : (integer-in 1 10000)
  height : (integer-in 1 10000)
  monochrome? : any/c = #f
(make-object bitmap% filename [kind bg-color]) → (is-a?/c bitmap%)
  filename : path-string?
  kind : (one-of/c 'unknown 'unknown/mask
                   'gif 'gif/mask 'jpeg 'png 'png/mask
                   'xbm 'xpm 'bmp 'pict)
       = 'unknown
  bg-color : (or/c (is-a?/c color%) false/c) = #f
(make-object bitmap% bits width height) → (is-a?/c bitmap%)
  bits : bytes?
  width : (integer-in 1 10000)
  height : (integer-in 1 10000)
```

When *width* and *height* are provided: Creates a new bitmap. If *monochrome?* is `#f`, the bitmap matches the display depth of the screen. The initial content of the bitmap is undefined.

When *filename* is provided: Creates a bitmap from a file, where *kind* specifies the kind of image file. See `load-file` for details.

When a *bits* byte string is provided: Creates a monochrome bitmap from an array of bit values, where each byte in *bits* specifies eight bits, and padding bits are added so that each bitmap line starts on a character boundary. A `1` bit value indicates black, and `0` indicates white. If *width* times *height* is larger than 8 times the length of *bits*, an `exn:fail:contract` exception is raised.

```
(send a-bitmap get-argb-pixels  x
                                y
                                width
                                height
                                pixels
                                [alpha?]) → void?
  x : real?
  y : real?
  width : (integer-in 1 10000)
  height : (integer-in 1 10000)
  pixels : (and/c bytes? mutable?)
  alpha? : any/c = #f
```

Produces the same result as `get-argb-pixels` in `bitmap-dc%`, but the bitmap does not have to be selected into the DC (and this method works even if the bitmap is selected into another DC, attached as a button label, etc.).

```
(send a-bitmap get-depth) → exact-nonnegative-integer?
```

Gets the color depth of the bitmap. See also `is-color?`.

```
(send a-bitmap get-gl-config config) → void?
  config : (is-a?/c gl-config%)
```

Returns a copy of this bitmap's requested OpenGL configuration. See also `set-gl-config`.

```
(send a-bitmap get-height) → (integer-in 1 10000)
```

Gets the height of the bitmap in pixels.

```
(send a-bitmap get-loaded-mask)
 → (or/c (is-a?/c bitmap%) false/c)
```

Returns a mask bitmap that is stored with this bitmap.

When a GIF file is loaded with `'gif/mask` or `'unknown/mask` and the file contains a transparent "color," a mask bitmap is generated to identify the transparent pixels. The mask bitmap is monochrome, with white pixels where the loaded bitmap is transparent and black pixels everywhere else.

When a PNG file is loaded with `'png/mask` or `'unknown/mask` and the file contains a mask or alpha channel, a mask bitmap is generated to identify the mask or alpha channel. If the file contains a mask or an alpha channel with only extreme values, the mask bitmap is monochrome, otherwise it is grayscale (representing the alpha channel inverted).

The mask bitmap is *not* used automatically by drawing routines. The mask bitmap can be extracted and supplied explicitly as a mask (e.g., as the sixth argument to `draw-bitmap`). The mask bitmap is used by `save-file` when saving a bitmap as `'png` if the mask has the same dimensions as the saved bitmap. The mask bitmap is also used automatically when the bitmap is a control label.

```
(send a-bitmap get-width) → (integer-in 1 10000)
```

Gets the width of the bitmap in pixels.

```
(send a-bitmap is-color?) → boolean?
```

Returns `#f` if the bitmap is monochrome, `#t` otherwise.

```
(send a-bitmap load-file name [kind bg-color]) → boolean?
  name : path-string?
  kind : (one-of/c 'unknown 'unknown/mask
                   'gif 'gif/mask 'jpeg 'png 'png/mask
                   'xbm 'xpm 'bmp 'pict)
       = 'unknown
  bg-color : (or/c (is-a?/c color%) false/c) = #f
```

Loads a bitmap from a file. If the bitmap is in use by a `bitmap-dc%` object or a control, the bitmap file is not loaded. The bitmap changes its size and depth to match that of the loaded image.

The *kind* argument specifies the file's format:

- `'unknown` — examine the file to determine its format
- `'unknown/mask` — like `'unknown`, but see `get-loaded-mask`
- `'gif` — load a GIF bitmap file (X, Windows, Mac OS X)
- `'gif/mask` — like `'gif`, but see `get-loaded-mask` (X, Windows, Mac OS X)
- `'jpeg` — load a JPEG bitmap file (X, Windows, Mac OS X)
- `'png` — load a PNG bitmap file (X, Windows, Mac OS X)
- `'png/mask` — like `'png`, but see `get-loaded-mask` (X, Windows, Mac OS X)
- `'xbm` — load an X bitmap (XBM) file (X, Windows, Mac OS X); creates a monochrome bitmap
- `'xpm` — load an XPM bitmap file (X, Windows, Mac OS X)
- `'bmp` — load a Windows bitmap file (X, Windows, Mac OS X)
- `'pict` — load a PICT bitmap file (Mac OS X)

An XBM image is always loaded as a monochrome bitmap. A 1-bit grayscale PNG without a mask or alpha channel is also loaded as a monochrome bitmap. An image in any other format is always loaded as a bitmap that matches the depth of the screen.

For PNG loading, if *bg-color* is not `#f`, then it is combined with the file's alpha channel or mask (if any) while loading the image; in this case, no separate mask bitmap is generated, even if `'unknown/mask` or `'png/mask` is specified for the format. If the format is specified as `'unknown` or `'png` and *bg-color* is not specified, the PNG file is consulted for a background color to use for loading, and white is used if no background color is indicated in the file.

In all PNG-loading modes, gamma correction is applied when the file provides a gamma value, otherwise gamma correction is not applied. The current display's gamma factor is determined by the 'GRacket:gamma preference (see §3.1 "Preferences") if it is set, or else by the SCREEN_GAMMA environment variable if it is defined. If the preference and environment variable are both undefined, a platform-specific default is used.

(send *a-bitmap* ok?) → boolean?

Returns #t if the bitmap is usable (created or changed successfully). If #f is returned, the bitmap cannot be supplied to methods that consume or operate on bitmaps (otherwise, an exn:fail:contract exception is raised).

```
(send a-bitmap save-file name kind [quality]) → boolean?
  name : path-string?
  kind : (one-of/c 'png 'jpeg 'xbm 'xpm 'bmp)
  quality : (integer-in 0 100) = 75
```

Saves a bitmap in the named file.

The *kind* argument determined the type of file that is created, one of:

- 'png — save a PNG file (X, Windows, Mac OS X)
- 'jpeg — save a JPEG file (X, Windows, Mac OS X)
- 'xbm — save an X bitmap (XBM) file (X, Windows, Mac OS X)
- 'xpm — save an XPM bitmap file (X, Windows, Mac OS X)
- 'bmp — save a Windows bitmap file (Windows)

The *quality* argument is used only for saving as 'jpeg, in which case it specifies the trade-off between image precision (high quality matches the content of the bitmap% object more precisely) and size (low quality is smaller).

When saving as 'png, if get-loaded-mask returns a bitmap of the same size as this one, a grayscale version is included in the PNG file as the alpha channel.

A monochrome bitmap saved as 'png without a mask bitmap produces a 1-bit grayscale PNG file (which, when read with load-file, creates a monochrome bitmap% object.)

```
(send a-bitmap set-gl-config config) → void?
  config : (is-a?/c gl-config%)
```

Sets the requested OpenGL configuration for this bitmap. The configuration is used when

the bitmap selected into a drawing context, and then a GL context is created for the drawing context.

The given gl-config% object is copied, so that changes to the object do not affect the bitmap's configuration.

(send a-bitmap set-loaded-mask mask) → void?
  mask : (is-a?/c bitmap%)

See get-loaded-mask.

2.3.2 bitmap-dc%

---

bitmap-dc% : class?
  superclass: object%
  extends: dc<%>

A bitmap-dc% object allows drawing directly into a bitmap. A bitmap% object must be supplied at initialization or installed into a bitmap DC using set-bitmap before any other method of the DC is called, except get-text-extent, get-char-height, or get-char-width. If any other bitmap-dc% method is called before a bitmap is selected, the method call is ignored.

Drawing to a bitmap-dc% with a color bitmap is guaranteed to produce the same result as drawing into a canvas% instance (with appropriate clipping and offsets). Thus, a bitmap-dc% can be used for offscreen staging of canvas content.

(new bitmap-dc% [bitmap bitmap]) → (is-a?/c bitmap-dc%)
  bitmap : (or/c (is-a?/c bitmap%) false/c)

Creates a new memory DC. If *bitmap* is not #f, it is installed into the DC so that drawing commands on the DC draw to *bitmap*. Otherwise, no bitmap is installed into the DC and set-bitmap must be called before any other method of the DC is called.

```
(send a-bitmap-dc draw-bitmap-section-smooth source
                                             dest-x
                                             dest-y
                                             dest-width
                                             dest-height
                                             src-x
                                             src-y
                                             src-width
                                             src-height
                                             mask)
 → boolean?
  source : (is-a?/c bitmap%)
  dest-x : real?
  dest-y : real?
  dest-width : (and/c real? (not/c negative?))
  dest-height : (and/c real? (not/c negative?))
  src-x : real?
  src-y : real?
  src-width : (and/c real? (not/c negative?))
  src-height : (and/c real? (not/c negative?))
  mask : (or/c (is-a?/c bitmap%) false/c)
```

Display part of a bitmap with smooth scaling. For most platforms, this method produces better results than adjusting the scale of a drawing context before using `draw-bitmap` and `draw-bitmap-section`, but this method is much slower.

```
(send a-bitmap-dc get-argb-pixels  x
                                   y
                                   width
                                   height
                                   pixels
                                  [alpha?]) → void?
  x : real?
  y : real?
  width : (integer-in 1 10000)
  height : (integer-in 1 10000)
  pixels : (and/c bytes? (not/c immutable?))
  alpha? : any/c = #f
```

Gets a rectangle of pixels in the bitmap, subject to the same rules and performance characteristics of `get-pixel`, except that the block get is likely to be faster than the sequence of individual gets. Also, the `bitmap%` class also provides the same method directly, so it is not necessary to select a bitmap into a DC to extracts its pixel values.

The pixel RGB values are copied into *pixels*. The first byte represents an alpha value of the pixel at (*x*, *y*), the second byte represents a red value of the pixel at (*x*, *y*), the third byte is the blue value, etc. In this way, the first *width* * *height* *4 bytes of *pixels* are set to reflect the current pixel values in the DC. The pixels are in row-major order, left to right then top to bottom.

If *alpha?* is false, then the alpha value for each pixel is set to 255. If *alpha?* is true, then *only* the alpha value is set for each pixel, based on each pixel's inverted value. Thus, the same *pixels* byte string is in general filled from two bitmaps: one (the main image) for the pixel values and one (the mask) for the alpha values.

```
(send a-bitmap-dc get-bitmap)
 → (or/c (is-a?/c bitmap%) false/c)
```

Gets the bitmap currently installed in the DC, or `#f` if no bitmap is installed. See `set-bitmap` for more information.

```
(send a-bitmap-dc get-pixel x y color) → boolean?
  x : real?
  y : real?
  color : (is-a?/c color%)
```

Fills *color* with the color of the current pixel at position (*x*, *y*) in the drawing context. If the color is successfully obtained, the return value is `#t`, otherwise the result is `#f`.

Under X, interleaving drawing commands with `get-pixel` calls (for the same `bitmap-dc%` object) incurs a substantial performance penalty, except for interleaved calls to `set-pixel`, `set-argb-pixels`, and `get-argb-pixels`.

```
(send a-bitmap-dc set-argb-pixels  x
                                   y
                                   width
                                   height
                                   pixels
                                  [alpha?]) → void?
  x : real?
  y : real?
  width : (integer-in 1 10000)
  height : (integer-in 1 10000)
  pixels : bytes?
  alpha? : any/c = #f
```

Sets a rectangle of pixels in the bitmap, subject to the same rules and performance characteristics of `set-pixel`, except that the block set is likely to be faster than the sequence of

individual sets.

The pixel RGB values are taken from *pixels*. The first byte represents an alpha value, the second byte represents a red value to used for the pixel at (*x*, *y*), the third byte is a blue value, etc. In this way, the first *width* * *height* * 4 bytes of *pixels* determine the new pixel values in the DC. The pixels are in row-major order, left to right then top to bottom.

If *alpha?* is false, then the alpha value for each pixel is ignored. If *alpha?* is true, then each pixel is set based *only* on the alpha value, but inverted to serve as a mask. Thus, the same *pixels* byte string is in general used with two bitmaps, one (the main image) for the pixel values and one (the mask) for the alpha values.

(send *a-bitmap-dc* set-bitmap *bitmap*) → void?
  bitmap : (or/c (is-a?/c bitmap%) false/c)

Installs a bitmap into the DC, so that drawing operations on the bitmap DC draw to the bitmap. A bitmap is removed from a DC by setting the bitmap to #f.

A bitmap can be selected into at most one bitmap DC, and only when it is not used by a control (as a label) or in a pen% or brush% (as a stipple). If the argument to set-bitmap is already in use by another DC, a control, a pen%, or a brush%, an exn:fail:contract exception is raised.

(send *a-bitmap-dc* set-pixel *x y color*) → void?
  x : real?
  y : real?
  color : (is-a?/c color%)

Sets a pixel in the bitmap.

The current clipping region might not affect the pixel change. Under X, interleaving drawing commands with set-pixel calls (for the same bitmap-dc% object) incurs a substantial performance penalty, except for interleaved calls to get-pixel, get-argb-pixels, and set-argb-pixels.

2.3.3 brush%

---

brush% : class?
  superclass: object%

A brush is a drawing tool with a color and a style that is used for filling in areas, such as the interior of a rectangle or ellipse. On a monochrome display, all non-white brushes are drawn as black.

In addition to its color and style, a brush can have a stipple bitmap. This stipple is used only in unsmoothed mode (see set-smoothing) or in a PostScript drawing context. Painting with a stipple brush is similar to calling draw-bitmap with the stipple bitmap in the filled region, except that the bitmap may not be scaled in the same way (depending on the platform and device).

A brush's style is one of the following:

- 'transparent — Draws with no effect (on the interior of the drawn shape).
- 'solid — Draws using the brush's color. If a monochrome stipple is installed into the brush, black pixels from the stipple are transferred to the destination using the brush's color, and white pixels from the stipple are not transferred.
- 'opaque — Same as 'solid, except when a monochrome stipple is installed for unsmoothed or PostScript drawing; in that case, white pixels from the stipple are transferred to the destination using the destination's background color.
- 'xor — In a smoothing mode or if a color stipple is installed, 'xor is treated as 'solid. Otherwise, the brush's color or colored (monochrome) stipple is xor-ed with existing destination pixel values. The 'xor mapping is unspecified for arbitrary color combinations, but the mapping provides two guarantees:
  - Black-and-white drawing to a color or monochrome destination always works as expected: black xor white = black, white xor black = black, black xor black = white, and white xor white = white.
  - Performing the same drawing operation twice in a row with 'xor is equivalent to a no-op.
- 'hilite — In unsmoothed mode, existing destination pixels are "highlighted" in a platform-specific way when the brush color is black. Under Windows and X for a color drawing context, the inverted RGB components of destination pixel are combined with the RGB components of the system-wide highlight color using a bitwise "or", and the combination is used. (Under X, the color is specified by the 'GRacket:hiliteColor preference preference; see §3.1 "Preferences".) Under Mac OS X for a color drawing context, the inverted RGB components of the system-wide highlight color are subtracted from the RGB components of each destination pixel, and the difference (or 0 for a negative result) is used. For any monochrome drawing context, 'hilite is the same as 'xor. For PostScript output, 'hilite uses a stipple that is an array of small dots (essentially a halftone), otherwise 'hilite is treated like 'solid in a smoothing mode.
- 'panel — In unsmoothed mode, draws with the same color and pattern as a top-level panel background, if the brush's color is the same as the color returned by get-panel-background and if the brush has no stipple. To create a canvas% object that is drawn like a control, use the 'transparent canvas style instead, because certain kinds of nested panels have different background colors (e.g., a tab-panel% under Mac OS X). In a smoothing mode, 'panel is treated as 'solid.

- The following modes correspond to built-in stipples drawn in `'solid` mode:
  - `'bdiagonal-hatch` — diagonal lines, top-left to bottom-right
  - `'crossdiag-hatch` — crossed diagonal lines
  - `'fdiagonal-hatch` — diagonal lines, top-right to bottom-left
  - `'cross-hatch` — crossed horizontal and vertical lines
  - `'horizontal-hatch` — horizontal lines
  - `'vertical-hatch` — vertical lines

  However, when a specific stipple is installed into the brush for when drawing with a smoothing mode into a non-PostScript context, the above modes are ignored and `'solid` is used, instead.

To draw outline shapes (such as unfilled boxes and ellipses), use the `'transparent` brush style. See `set-style` for more information about styles.

To avoid creating multiple brushes with the same characteristics, use the global `brush-list%` object `the-brush-list`, or provide a color and style to `set-brush` in `dc<%>`.

```
(make-object brush%) → (is-a?/c brush%)
(make-object brush% color style) → (is-a?/c brush%)
  color : (is-a?/c color%)
  style : (one-of/c 'transparent 'solid 'opaque
                    'xor 'hilite 'panel
                    'bdiagonal-hatch 'crossdiag-hatch
                    'fdiagonal-hatch 'cross-hatch
                     'horizontal-hatch 'vertical-hatch)
(make-object brush% color-name style) → (is-a?/c brush%)
  color-name : string?
  style : (one-of/c 'transparent 'solid 'opaque
                    'xor 'hilite 'panel
                    'bdiagonal-hatch 'crossdiag-hatch
                    'fdiagonal-hatch 'cross-hatch
                     'horizontal-hatch 'vertical-hatch)
```

When no argument are provided, the result is a solid black brush. Otherwise, the result is a brush with the given color and style. For the case that the color is specified using a name, see `color-database<%>` for information about color names; if the name is not known, the brush's color is black.

```
(send a-brush get-color) → (is-a?/c color%)
```

Returns the brush's color.

```
(send a-brush get-stipple) → (or/c (is-a?/c bitmap%) false/c)
```

Gets the stipple bitmap, or `#f` if the brush has no stipple.

```
(send a-brush get-style)
 → (one-of/c 'transparent 'solid 'opaque
             'xor 'hilite 'panel
             'bdiagonal-hatch 'crossdiag-hatch
             'fdiagonal-hatch 'cross-hatch
             'horizontal-hatch 'vertical-hatch)
```

Returns the brush's style. See `brush%` for information about brush styles.

```
(send a-brush set-color color) → void?
  color : (is-a?/c color%)
(send a-brush set-color color-name) → void?
  color-name : string?
(send a-brush set-color red green blue) → void?
  red : (integer-in 0 255)
  green : (integer-in 0 255)
  blue : (integer-in 0 255)
```

Sets the brush's color. A brush cannot be modified if it was obtained from a `brush-list%` or while it is selected into a drawing context.

For the case that the color is specified using a string, see `color-database<%>` for information about color names.

```
(send a-brush set-stipple bitmap) → void?
  bitmap : (or/c (is-a?/c bitmap%) false/c)
```

Sets or removes the stipple bitmap, where `#f` removes the stipple. See `brush%` for information about drawing with stipples.

A bitmap cannot be used as a stipple if it is selected into a `bitmap-dc%` object; if the given bitmap is selected into a `bitmap-dc%` object, an `exn:fail:contract` exception is raised. A brush cannot be modified if it was obtained from a `brush-list%` or while it is selected into a drawing context.

A pen's stipple is not used in a smoothing mode, except for a `post-script-dc%` (which is always in a smoothing mode).

```
(send a-brush set-style style) → void?
```

```
  style : (one-of/c 'transparent 'solid 'opaque
                    'xor 'hilite 'panel
                    'bdiagonal-hatch 'crossdiag-hatch
                    'fdiagonal-hatch 'cross-hatch
                    'horizontal-hatch 'vertical-hatch)
```

Sets the brush's style. See brush% for information about the possible styles.

A brush cannot be modified if it was obtained from a brush-list% or while it is selected into a drawing context.

2.3.4 brush-list%

```
brush-list% : class?
  superclass: object%
```

A brush-list% object maintains a list of brush% objects to avoid creating brushes repeatedly. A brush% object in a brush list cannot be mutated.

A global brush list, the-brush-list, is created automatically.

```
(new brush-list%) → (is-a?/c brush-list%)
```

Creates an empty brush list.

```
(send a-brush-list find-or-create-brush color
                                        style)
 → (is-a?/c brush%)
  color : (is-a?/c color%)
  style : (one-of/c 'transparent 'solid 'opaque
                    'xor 'hilite 'panel
                    'bdiagonal-hatch 'crossdiag-hatch
                    'fdiagonal-hatch 'cross-hatch
                    'horizontal-hatch 'vertical-hatch)
(send a-brush-list find-or-create-brush color-name
                                        style)
 → (or/c (is-a?/c brush%) false/c)
  color-name : string?
  style : (one-of/c 'transparent 'solid 'opaque
                    'xor 'hilite 'panel
                    'bdiagonal-hatch 'crossdiag-hatch
                    'fdiagonal-hatch 'cross-hatch
                    'horizontal-hatch 'vertical-hatch)
```

Finds a brush of the given specification, or creates one and adds it to the list. See brush% for a further explanation of the arguments, which are the same as brush%'s initialization arguments.

2.3.5 color%

---

color% : class?
  superclass: object%

A color is an object representing a red-green-blue (RGB) combination of primary colors, and is used to determine drawing colors. Each red, green, or blue component of the color is in the range 0 to 255, inclusive. For example, (0, 0, 0) is black, (255, 255, 255) is white, and (255, 0, 0) is red.

See color-database<%> for information about obtaining a color object using a color name.

(make-object color%) → (is-a?/c color%)
(make-object color% *red green blue*) → (is-a?/c color%)
  red : (integer-in 0 255)
  green : (integer-in 0 255)
  blue : (integer-in 0 255)
(make-object color% *color-name*) → (is-a?/c color%)
  color-name : string?

Creates a new color with the given RGB values, or matching the given color name (using "black" if no color is given or if the name is not recognized). See color-database<%> for more information on color names.

(send *a-color* blue) → (integer-in 0 255)

Returns the blue component of the color.

(send *a-color* copy-from *src*) → (is-a?/c color%)
  src : (is-a?/c color%)

Copies the RGB values of another color object to this one, returning this object as the result.

(send *a-color* green) → (integer-in 0 255)

Returns the green component of the color.

```
(send a-color ok?) → boolean?
```

Returns `#t` if the color object is valid.

```
(send a-color red) → (integer-in 0 255)
```

Returns the red component of the color.

```
(send a-color set red green blue) → void?
  red : (integer-in 0 255)
  green : (integer-in 0 255)
  blue : (integer-in 0 255)
```

Sets the three (red, green, and blue) component values of the color.

2.3.6 color-database<%>

```
color-database<%> : interface?
```

The global `the-color-database` object is an instance of `color-database<%>`. It maintains a database of standard RGB colors for a predefined set of named colors (such as “black” and “light grey”).

The following colors are in the database:

Orange Red
OrangeRed
Tomato
DarkRed
Red
Firebrick
Crimson
DeepPink
Maroon
Indian Red
IndianRed
Medium Violet Red
MediumVioletRed
Violet Red
VioletRed
LightCoral
HotPink

PaleVioletRed
LightPink
RosyBrown
Pink
Orchid
LavenderBlush
Snow
Chocolate
SaddleBrown
Brown
DarkOrange
Coral
Sienna
Orange
Salmon
Peru
DarkGoldenrod
Goldenrod
SandyBrown
LightSalmon
DarkSalmon
Gold
Yellow
Olive
Burlywood
Tan
NavajoWhite
PeachPuff
Khaki
DarkKhaki
Moccasin
Wheat
Bisque
PaleGoldenrod
BlanchedAlmond
Medium Goldenrod
MediumGoldenrod
PapayaWhip
MistyRose
LemonChiffon
AntiqueWhite
Cornsilk
LightGoldenrodYellow
OldLace
Linen
LightYellow

SeaShell
Beige
FloralWhite
Ivory
Green
LawnGreen
Chartreuse
Green Yellow
GreenYellow
Yellow Green
YellowGreen
Medium Forest Green
OliveDrab
MediumForestGreen
Dark Olive Green
DarkOliveGreen
DarkSeaGreen
Lime
Dark Green
DarkGreen
Lime Green
LimeGreen
Forest Green
ForestGreen
Spring Green
SpringGreen
Medium Spring Green
MediumSpringGreen
Sea Green
SeaGreen
Medium Sea Green
MediumSeaGreen
Aquamarine
LightGreen
Pale Green
PaleGreen
Medium Aquamarine
MediumAquamarine
Turquoise
LightSeaGreen
Medium Turquoise
MediumTurquoise
Honeydew
MintCream
RoyalBlue
DodgerBlue

DeepSkyBlue
CornflowerBlue
Steel Blue
SteelBlue
LightSkyBlue
Dark Turquoise
DarkTurquoise
Cyan
Aqua
DarkCyan
Teal
Sky Blue
SkyBlue
Cadet Blue
CadetBlue
Dark Slate Gray
DarkSlateGray
LightSlateGray
SlateGray
Light Steel Blue
LightSteelBlue
Light Blue
LightBlue
PowderBlue
PaleTurquoise
LightCyan
AliceBlue
Azure
Medium Blue
MediumBlue
DarkBlue
Midnight Blue
MidnightBlue
Navy
Blue
Indigo
Blue Violet
BlueViolet
Medium Slate Blue
MediumSlateBlue
Slate Blue
SlateBlue
Purple
Dark Slate Blue
DarkSlateBlue
DarkViolet

Dark Orchid
DarkOrchid
MediumPurple
Cornflower Blue
Medium Orchid
MediumOrchid
Magenta
Fuchsia
DarkMagenta
Violet
Plum
Lavender
Thistle
GhostWhite
White
WhiteSmoke
Gainsboro
Light Gray
LightGray
Silver
Gray
Dark Gray
DarkGray
Dim Gray
DimGray
Black

The names are not case-sensitive.

See also color%.

```
(send a-color-database find-color color-name)
 → (or/c (is-a?/c color%) false/c)
  color-name : string?
```

Finds a color by name (character case is ignored). If no color is found for the name, #f is returned.

2.3.7 dc<%>

---

dc<%> : interface?

A dc<%> object is a drawing context for drawing graphics and text. It represents output

devices in a generic way; e.g., a canvas has a drawing context, as does a printer.

The drawing methods, such as draw-rectangle, accept real number values as arguments, but the results are only well-defined when the drawing coordinates are in the range -16383 to 16383. This restriction applies to the coordinates both before and after offsets and scaling factors are applied.

(send a-dc cache-font-metrics-key) → exact-integer?

Returns an integer that, if not 0, corresponds to a particular kind of device and scaling factor, such that text-extent information (from get-text-extent, get-char-height, etc.) is the same. The key is valid across all dc<%> instances, even among different classes.

A 0 result indicates that the current configuration of a-dc does not fit into a common category, and so no key is available for caching text-extent information.

(send a-dc clear) → void?

Clears the drawing region (fills it with the current background color, as determined by get-background).

```
(send a-dc draw-arc x
                    y
                    width
                    height
                    start-radians
                    end-radians)  → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  start-radians : real?
  end-radians : real?
```

Draws a counter-clockwise circular arc, a part of the ellipse inscribed in the rectangle specified by *x* (left), *y* (top), *width*, and *height*. The arc starts at the angle specified by *start-radians* (0 is three o'clock and half-pi is twelve o'clock) and continues counter-clockwise to *end-radians*. If *start-radians* and *end-radians* are the same, a full ellipse is drawn.

The current pen is used for the arc. If the current brush is not transparent, it is used to fill the wedge bounded by the arc plus lines (not drawn) extending to the center of the inscribed ellipse.

If both the pen and brush are non-transparent, the wedge is filled with the brush before the

arc is drawn with the pen. The wedge and arc meet so that no space is left between them, but the precise overlap between the wedge and arc is platform- and size-specific. Typically, the regions drawn by the brush and pen overlap. More generally, the pen is centered over the outline of the arc, rounding toward the center in unsmoothed mode.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-bitmap  source
                        dest-x
                        dest-y
                       [style
                        color
                        mask]) → boolean?
  source : (is-a?/c bitmap%)
  dest-x : real?
  dest-y : real?
  style : (one-of/c 'solid 'opaque 'xor) = 'solid
  color : (is-a?/c color%)
        = (send the-color-database find-color "black")
  mask : (or/c (is-a?/c bitmap%) false/c) = #f
```

Displays a bitmap. The *dest-x* and *dest-y* arguments are in DC coordinates.

For color bitmaps, the drawing style and color arguments are ignored. For monochrome bitmaps, draw-bitmap uses the style and color arguments in the same way that a brush uses its style and color settings to draw a monochrome stipple (see brush% for more information).

If a mask bitmap is supplied, it must have the same width and height as the bitmap to display, and its ok? must return true, otherwise an exn:fail:contract exception is raised. The bitmap to draw and the mask bitmap can be the same object, but if the drawing context is a bitmap-dc% object, both bitmaps must be distinct from the destination bitmap, otherwise an exn:fail:contract exception is raised.

If the mask bitmap is monochrome, drawing occurs in the target dc<%> only where the mask bitmap contains black pixels.

If the mask bitmap is grayscale and the bitmap to draw is not monochrome, then the blackness of each mask pixel controls the opacity of the drawn pixel (i.e., the mask acts as an inverted alpha channel), at least on most platforms. (Non-monochrome masks are collapsed to monochrome under X when the RENDER extension is not available, and under Windows 95 and NT when "msing32.dll" is not available.) Other combinations involving a non-monochrome mask (i.e., a non-grayscale mask or a monochrome bitmap to draw) produce platform-specific results.

The current brush, current pen, current text, and current alpha settings for the DC have no effect on how the bitmap is drawn, but the bitmap is scaled if the DC has a scale.

For post-script-dc% output, the mask bitmap is currently ignored, and the 'solid style is treated the same as 'opaque. (However, mask bitmaps and 'solid drawing may become supported for post-script-dc% in the future.)

The result is #t if the bitmap is successfully drawn, #f otherwise (possibly because the bitmap's ok? method returns #f).

See also draw-bitmap-section.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-bitmap-section  source
                                dest-x
                                dest-y
                                src-x
                                src-y
                                src-width
                                src-height
                               [style
                                color
                                mask])      → boolean?
  source : (is-a?/c bitmap%)
  dest-x : real?
  dest-y : real?
  src-x : real?
  src-y : real?
  src-width : (and/c real? (not/c negative?))
  src-height : (and/c real? (not/c negative?))
  style : (one-of/c 'solid 'opaque 'xor) = 'solid
  color : (is-a?/c color%)
        = (send the-color-database find-color "black")
  mask : (or/c (is-a?/c bitmap%) false/c) = #f
```

Displays part of a bitmap.

The *src-x*, *src-y*, *src-width*, and *src-height* arguments specify a rectangle in the source bitmap to copy into this drawing context.

See draw-bitmap for information about *dest-x*, *dest-y*, *style*, *color*, and *mask*.

```
(send a-dc draw-ellipse x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Draws an ellipse contained in a rectangle with the given top-left corner and size. The current pen is used for the outline, and the current brush is used for filling the shape.

If both the pen and brush are non-transparent, the ellipse is filled with the brush before the outline is drawn with the pen. The filling and outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and size-specific. Typically, the regions drawn by the brush and pen overlap. More generally, the pen is centered over the outline of the ellipse, rounding toward the center in unsmoothed mode.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-line x1 y1 x2 y2) → void?
  x1 : real?
  y1 : real?
  x2 : real?
  y2 : real?
```

Draws a line from one point to another. The current pen is used for drawing the line.

In unsmoothed mode, the points correspond to pixels, and the line covers both the start and end points. For a pen whose scaled width is larger than 1, the line is drawn centered over the start and end points.

See also set-smoothing for information on the 'aligned smoothing mode.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-lines  points
                      [xoffset
                       yoffset]) → void?
  points : (listof (is-a?/c point%))
  xoffset : real? = 0
  yoffset : real? = 0
```

Draws lines using a list of *points*, adding *xoffset* and *yoffset* to each point. The current pen is used for drawing the lines.

See also set-smoothing for information on the 'aligned smoothing mode.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-path  path
                     [xoffset
                      yoffset
                      fill-style]) → void?
```

```
  path : (is-a?/c dc-path%)
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (one-of/c 'odd-even 'winding) = 'odd-even
```

Draws the sub-paths of the given dc-path% object, adding *xoffset* and *yoffset* to each point. (See dc-path% for general information on paths and sub-paths.) The current pen is used for drawing the path as a line, and the current brush is used for filling the area bounded by the path.

If both the pen and brush are non-transparent, the path is filled with the brush before the outline is drawn with the pen. The filling and outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and size-specific. Thus, the regions drawn by the brush and pen may overlap. More generally, the pen is centered over the path, rounding left and down in unsmoothed mode.

The *fill-style* argument specifies the fill rule: 'odd-even or 'winding. In 'odd-even mode, a point is considered enclosed within the path if it is enclosed by an odd number of sub-path loops. In 'winding mode, a point is considered enclosed within the path if it is enclosed by more or less clockwise sub-path loops than counter-clockwise sub-path loops. In unsmoothed mode, the 'winding fill rule is not supported under Mac OS X and it is not supported when *path* contains multiple sub-paths; the 'winding fill rules is always supported when smoothing is enabled (see set-smoothing).

See also set-smoothing for information on the 'aligned smoothing mode.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-point x y) → void?
  x : real?
  y : real?
```

Plots a single point using the current pen.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-polygon  points
                        [xoffset
                         yoffset
                         fill-style]) → void?
  points : (listof (is-a?/c point%))
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (one-of/c 'odd-even 'winding) = 'odd-even
```

Draw a filled polygon using a list of *points*, adding *xoffset* and *yoffset* to each point. The polygon is automatically closed, so the first and last point can be different. The current pen is used for drawing the outline, and the current brush for filling the shape.

If both the pen and brush are non-transparent, the polygon is filled with the brush before the outline is drawn with the pen. The filling and outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and shape-specific. Thus, the regions drawn by the brush and pen may overlap. More generally, the pen is centered over the polygon lines, rounding left and down in unsmoothed mode.

The *fill-style* argument specifies the fill rule: `'odd-even` or `'winding`. In `'odd-even` mode, a point is considered enclosed within the polygon if it is enclosed by an odd number of loops. In `'winding` mode, a point is considered enclosed within the polygon if it is enclosed by more or less clockwise loops than counter-clockwise loops. The `'winding` fill rule is not supported under Mac OS X, except when smoothing is enabled (see `set-smoothing`).

See also `set-smoothing` for information on the `'aligned` smoothing mode.

Restrictions on the magnitude of drawing coordinates are described with `dc<%>`.

```
(send a-dc draw-rectangle x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Draws a rectangle with the given top-left corner and size. The current pen is used for the outline and the current brush for filling the shape.

If both the pen and brush are non-transparent, the rectangle is filled with the brush before the outline is drawn with the pen. In unsmoothed mode, when the pen is size 0 or 1, the filling precisely overlaps the entire outline. As a result, if a rectangle is drawn with a size-0 or size-1 `'xor` `pen%` and an `'xor` `brush%`, the outline is xored twice (first by the brush, then by the pen), leaving it unchanged. More generally, the pen is centered over the outline of the rectangle, rounding toward the center in unsmoothed mode.

See also `set-smoothing` for information on the `'aligned` smoothing mode.

Restrictions on the magnitude of drawing coordinates are described with `dc<%>`.

```
(send a-dc draw-rounded-rectangle  x
                                   y
                                   width
                                   height
                                  [radius]) → void?
```

```
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  radius : real? = -0.25
```

Draws a rectangle with the given top-left corner, and with the given size. The corners are quarter-circles using the given radius. The current pen is used for the outline and the current brush for filling the shape.

If *radius* is positive, the value is used as the radius of the rounded corner. If *radius* is negative, the absolute value is used as the *proportion* of the smallest dimension of the rectangle.

If *radius* is less than -0.5 or more than half of *width* or *height*, an exn:fail:contract exception is raised.

If both the pen and brush are non-transparent, the rectangle is filled with the brush before the outline is drawn with the pen. The filling and outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and size-specific. Thus, the regions drawn by the brush and pen may partially overlap. More generally, the pen is centered over the outline of the rounded rectangle, rounding toward the center in unsmoothed mode.

See also set-smoothing for information on the 'aligned smoothing mode.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-spline x1 y1 x2 y2 x3 y3) → void?
  x1 : real?
  y1 : real?
  x2 : real?
  y2 : real?
  x3 : real?
  y3 : real?
```

Draws a spline from (*x1*, *y1*) to (*x3*, *y3*) using (*x2*, *y2*) as the control point.

See also set-smoothing for information on the 'aligned smoothing mode. See also dc-path% and draw-path for drawing more complex curves.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc draw-text  text
                      x
                      y
                     [combine?
                      offset
                      angle])  → void?
  text : string?
  x : real?
  y : real?
  combine? : any/c = #f
  offset : exact-nonnegative-integer? = 0
  angle : real? = 0
```

Draws a text string at a specified point, using the current text font, and the current text foreground and background colors. For unrotated text, the specified point is used as the starting top-left point for drawing characters (e.g, if "W" is drawn, the point is roughly the location of the top-left pixel in the "W"). Rotated text is rotated around this point.

The *text* string is drawn starting from the *offset* character, and continuing until the end of *text* or the first null character.

If *combine?* is #t, then *text* may be measured with adjacent characters combined to ligature glyphs, with Unicode combining characters as a single glyph, with kerning, with right-to-left rendering of characters, etc. If *combine?* is #f, then the result is the same as if each character is measured separately, and Unicode control characters are ignored.

The string is rotated by *angle* radians counter-clockwise. If *angle* is not zero, then the text is always drawn in transparent mode (see set-text-mode).

The current brush and current pen settings for the DC have no effect on how the text is drawn.

See get-text-extent for information on the size of the drawn text.

See also set-text-foreground, set-text-background, and set-text-mode.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-dc end-doc) → void?
```

Ends a document, relevant only when drawing to a printer or PostScript device (including to a PostScript file).

For printer or PostScript output, an exception is raised if end-doc is called when the document is not started with start-doc, when a page is currently started by start-page and

not ended with end-page, or when the document has been ended already.

(send a-dc end-page) → void?

Ends a single page, relevant only when drawing to a printer or PostScript device (including to a PostScript file).

For printer or PostScript output, an exception is raised if end-page is called when a page is not currently started by start-page.

(send a-dc get-alpha) → (real-in 0 1)

Gets the current opacity for drawing; see set-alpha.

(send a-dc get-background) → (is-a?/c color%)

Gets the color used for painting the background. See also set-background.

(send a-dc get-brush) → (is-a?/c brush%)

Gets the current brush. See also set-brush.

(send a-dc get-char-height) → (and/c real? (not/c negative?))

Gets the height of a character using the current font.

Unlike most methods, this method can be called for a bitmap-dc% object without a bitmap installed.

(send a-dc get-char-width) → (and/c real? (not/c negative?))

Gets the average width of a character using the current font.

Unlike most methods, this method can be called for a bitmap-dc% object without a bitmap installed.

(send a-dc get-clipping-region)
 → (or/c (is-a?/c region%) false/c)

Gets the current clipping region, returning #f if the drawing context is not clipped (i.e., the clipping region is the entire drawing region).

```
(send a-dc get-font) → (is-a?/c font%)
```

Gets the current font. See also set-font.

```
(send a-dc get-gl-context)
 → (or/c (is-a?/c gl-context<%>) false/c)
```

Returns a gl-context<%> object for this drawing context if it supports OpenGL, #f otherwise.

See gl-context<%> for more information.

```
(send a-dc get-origin) → real? real?
```

Returns the device origin, i.e., the location in device coordinates of (0,0) in logical coordinates.

See also set-origin.

```
(send a-dc get-pen) → (is-a?/c pen%)
```

Gets the current pen. See also set-pen.

```
(send a-dc get-scale) → real? real?
```

Returns the scaling factor that maps logical coordinates to device coordinates.

See also set-scale.

```
(send a-dc get-size) → nonnegative-real? nonnegative-real?
```

Gets the size of the destination drawing area. For a dc<%> object obtained from a canvas<%>, this is the (virtual client) size of the destination window; for a bitmap-dc% object, this is the size of the selected bitmap (or 0 if no bitmap is selected); for a post-script-dc% or printer-dc% drawing context, this gets the horizontal and vertical size of the drawing area.

```
(send a-dc get-smoothing)
 → (one-of/c 'unsmoothed 'smoothed 'aligned)
```

Returns the current smoothing mode. See set-smoothing.

```
(send a-dc get-text-background) → (is-a?/c color%)
```

Gets the current text background color. See also set-text-background.

```
(send a-dc get-text-extent  string
                           [font
                            combine?
                            offset]) → nonnegative-real?
                                       nonnegative-real?
                                       nonnegative-real?
                                       nonnegative-real?
  string : string?
  font : (or/c (is-a?/c font%) false/c) = #f
  combine? : any/c = #f
  offset : exact-nonnegative-integer? = 0
```

Returns the size of str at it would be drawn in the drawing context, starting from the *offset* character of str, and continuing until the end of str or the first null character. The *font* argument specifies the font to use in measuring the text; if it is #f, the current font of the drawing area is used. (See also set-font.)

The result is four real numbers:

- the total width of the text (depends on both the font and the text);
- the total height of the font (depends only on the font);
- the distance from the baseline of the font to the bottom of the descender (included in the height, depends only on the font); and
- extra vertical space added to the font by the font designer (included in the height, and often zero; depends only on the font).

The returned width and height define a rectangle is that guaranteed to contain the text string when it is drawn, but the fit is not necessarily tight. Some undefined number of pixels on the left, right, top, and bottom of the drawn string may be "whitespace," depending on the whims of the font designer and the platform-specific font-scaling mechanism.

If *combine?* is #t, then text may be drawn with adjacent characters combined to ligature glyphs, with Unicode combining characters as a single glyph, with kerning, with right-to-left ordering of characters, etc. If *combine?* is #f, then the result is the same as if each character is drawn separately, and Unicode control characters are ignored.

Unlike most methods, this method can be called for a bitmap-dc% object without a bitmap installed.

```
(send a-dc get-text-foreground) → (is-a?/c color%)
```

Gets the current text foreground color. See also set-text-foreground.

```
(send a-dc get-text-mode) → (one-of/c 'solid 'transparent)
```

Reports how text is drawn; see set-text-mode .

```
(send a-dc glyph-exists? c [font]) → boolean?
  c : char
  font : (or/c (is-a?/c font%) false/c) = #f
```

Returns #t if the given character has a corresponding glyph for this drawing context, #f otherwise.

Due to automatic font substitution when drawing or measuring text, the result of this method does not depend on the given font, which merely provides a hint for the glyph search. If the font is #f, the drawing context's current font is used. The result depends on the type of the drawing context, but the result for canvas% dc<%> instances and bitmap-dc% instances is always the same for a given platform and a given set of installed fonts.

See also screen-glyph-exists? .

```
(send a-dc ok?) → boolean?
```

Returns #t if the drawing context is usable.

```
(send a-dc set-alpha opacity) → void?
  opacity : (real-in 0 1)
```

Determines the opacity of drawing, under certain conditions:

- pen- and brush-based drawing when get-smoothing produces 'smoothed or 'aligned, and when the drawing context is not an instance of post-script-dc%; and
- text drawing for most platforms (Mac OS X, X with Xft/fontconfig; transparency approximated under Windows by fading the drawing color), and when the drawing context is not an instance of post-script-dc.

A value of 0.0 corresponds to completely transparent (i.e., invisible) drawing, and 1.0 corresponds to completely opaque drawing. For intermediate values, drawing is blended

with the existing content of the drawing context.

```
(send a-dc set-background color) → void?
  color : (is-a?/c color%)
```

Sets the background color for drawing in this object (e.g., using `clear` or using a stippled `brush%` with the mode `'opaque`). For monochrome drawing, all non-black colors are treated as white.

```
(send a-dc set-brush brush) → void?
  brush : (is-a?/c brush%)
(send a-dc set-brush color style) → void?
  color : (is-a?/c color%)
  style : (one-of/c 'transparent 'solid 'opaque
                    'xor 'hilite 'panel
                    'bdiagonal-hatch 'crossdiag-hatch
                    'fdiagonal-hatch 'cross-hatch
                    'horizontal-hatch 'vertical-hatch)
(send a-dc set-brush color-name style) → void?
  color-name : string?
  style : (one-of/c 'transparent 'solid 'opaque
                    'xor 'hilite 'panel
                    'bdiagonal-hatch 'crossdiag-hatch
                    'fdiagonal-hatch 'cross-hatch
                    'horizontal-hatch 'vertical-hatch)
```

Sets the current brush for drawing in this object. While a brush is selected into a drawing context, it cannot be modified. When a color and style are given, the arguments are as for `find-or-create-brush` in `brush-list%`.

```
(send a-dc set-clipping-rect x
                             y
                             width
                             height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Sets the clipping region to a rectangular region.

See also `set-clipping-region` and `get-clipping-region`.

Restrictions on the magnitude of drawing coordinates are described with `dc<%>`.

```
(send a-dc set-clipping-region rgn) → void?
  rgn : (or/c (is-a?/c region%) false/c)
```

Sets the clipping region for the drawing area, turning off all clipping within the drawing region if `#f` is provided.

The clipping region must be reset after changing a `dc<%>` object's origin or scale (unless it is `#f`); see `region%` for more information.

See also `set-clipping-rect` and `get-clipping-region`.

```
(send a-dc set-font font) → void?
  font : (is-a?/c font%)
```

Sets the current font for drawing text in this object.

```
(send a-dc set-origin x y) → void?
  x : real?
  y : real?
```

Sets the device origin, i.e., the location in device coordinates of (0,0) in logical coordinates.

Changing a `dc<%>` object's origin or scale does not affect `region%` objects that were previously created. See `region%` for more information.

Restrictions on the magnitude of drawing coordinates are described with `dc<%>`.

```
(send a-dc set-pen pen) → void?
  pen : (is-a?/c pen%)
(send a-dc set-pen color width style) → void?
  color : (is-a?/c color%)
  width : (real-in 0 255)
  style : (one-of/c 'transparent 'solid 'xor 'hilite
                    'dot 'long-dash 'short-dash 'dot-dash
                    'xor-dot 'xor-long-dash 'xor-short-dash
                    'xor-dot-dash)
(send a-dc set-pen color-name width style) → void?
  color-name : string?
  width : (real-in 0 255)
  style : (one-of/c 'transparent 'solid 'xor 'hilite
                    'dot 'long-dash 'short-dash 'dot-dash
                    'xor-dot 'xor-long-dash 'xor-short-dash
                    'xor-dot-dash)
```

Sets the current pen for this object. When a color, width, and style are given, the arguments are as for `find-or-create-pen` in `pen-list%`.

The current pen does not affect text drawing; see also `set-text-foreground`.

While a pen is selected into a drawing context, it cannot be modified.

```
(send a-dc set-scale x-scale y-scale) → void?
  x-scale : (and/c real? (not/c negative?))
  y-scale : (and/c real? (not/c negative?))
```

Sets a scaling factor that maps logical coordinates to device coordinates.

Changing a `dc<%>` object's origin or scale does not affect `region%` objects that were previously created. See `region%` for more information.

Restrictions on the magnitude of drawing coordinates are described with `dc<%>`.

```
(send a-dc set-smoothing mode) → void?
  mode : (one-of/c 'unsmoothed 'smoothed 'aligned)
```

Enables or disables anti-aliased smoothing of lines, curves, rectangles, rounded rectangles, ellipses, polygons, paths, and clear operations. (Text smoothing is not affected by this method, and is instead controlled through the `font%` object.)

Smoothing is supported under Windows only when Microsoft's "gdiplus.dll" is installed (which is always the case for Windows XP). Smoothing is supported under Mac OS X always. Smoothing is supported under X only when Cairo is installed when GRacket is compiled. Smoothing is never supported for black-and-white contexts. Smoothing is always supported (and cannot be disabled) for PostScript output.

The smoothing mode is either `'unsmoothed`, `'smoothed`, or `'aligned`. Both `'aligned` and `'smoothed` are smoothing modes.

In `'smoothed` mode for a canvas or bitmap drawing context, integer drawing coordinates correspond to the boundary between pixels, and pen-based drawing is centered over a given line or curve. Thus, drawing with pen width `1` from (0, 10) to (10, 10) draws a 2-pixel wide line with 50% opacity.

The `'aligned` smoothing mode is like `'smoothed`, but it paints pixels more like `'unsmoothed` mode. Since it aligns shapes to pixel boundaries, `'aligned` mode often produces better results than `'smoothed`, but the results depend on the application. The `'aligned` mode is defined in terms of `'smoothed` mode, except that drawing coordinates are rounded down (via `floor`, after scaling and origin translation). For line drawing, coordinates are then shifted right and down by the `floor` of half a pen width. In addition, for pen drawing through `draw-rectangle`, `draw-ellipse`, `draw-rounded-rectangle`, and `draw-arc`,

the given width and height are each decreased by 1.0.

In either smoothing mode, brush and pen stipples are ignored (except for PostScript drawing), and 'hilite and 'xor drawing modes are treated as 'solid. If smoothing is not supported, then attempting to set the smoothing mode to 'smoothed or 'aligned will have no effect, and get-smoothing will always return 'unsmoothed. Similarly, get-smoothing for a post-script-dc% always returns 'smoothed.

(send a-dc set-text-background color) → void?
  color : (is-a?/c color%)

Sets the current text background color for this object. The text background color is painted behind text that is drawn with draw-text, but only for the 'solid text mode (see set-text-mode).

For monochrome drawing, all non-white colors are treated as black.

(send a-dc set-text-foreground color) → void?
  color : (is-a?/c color%)

Sets the current text foreground color for this object, used for drawing text with draw-text.

For monochrome drawing, all non-black colors are treated as white.

(send a-dc set-text-mode mode) → void?
  mode : (one-of/c 'solid 'transparent)

Determines how text is drawn:

- 'solid — Before text is drawn, the destination area is filled with the text background color (see set-text-background).
- 'transparent — Text is drawn directly over any existing image in the destination, as if overlaying text written on transparent film.

(send a-dc start-doc message) → boolean?
  message : string?

Starts a document, relevant only when drawing to a printer or PostScript device (including to a PostScript file). For some platforms, the *message* string is displayed in a dialog until end-doc is called.

For printer or PostScript output, an exception is raised if start-doc has been called already (even if end-doc has been called as well). Furthermore, drawing methods raise an exception

if not called while a page is active as determined by start-doc and start-page.

(send a-dc start-page) → void?

Starts a page, relevant only when drawing to a printer or PostScript device (including to a PostScript file).

For printer or PostScript output, an exception is raised if start-page is called when a page is already started, or when start-doc has not been called, or when end-doc has been called already. In addition, in the case of PostScript output, Encapsulated PostScript (EPS) cannot contain multiple pages, so calling start-page a second time for a post-script-dc% instance raises an exception; to create PostScript output with multiple pages, supply #f as the as-eps initialization argument for post-script-dc%.

(send a-dc try-color try result) → void?
  try : (is-a?/c color%)
  result : (is-a?/c color%)

Determines the actual color used for drawing requests with the given color. The *result* color is set to the RGB values that are actually produced for this drawing context to draw the color *try*.

2.3.8 dc-path%

---

dc-path% : class?
  superclass: object%

A path is a set of figures defined by curves. A path can be used with the draw-path method of a dc<%> object to draw the path's curves as lines, fill the region bounded by the path's curves, or both. A path can also be used with the set-path method of a region% object to generate a region bounded by the path's curves.

A path consists of zero or more *closed sub-paths*, and possibly one *open sub-path*. Some dc-path% methods extend the open sub-path, some dc-path% methods close the open sub-path, and some dc-path% methods add closed sub-paths. This approach to drawing formulation is inherited from PostScript [Adobe99].

When a path is drawn as a line, a closed sub-path is drawn as a closed figure, analogous to a polygon. An open sub-path is drawn with disjoint start and end points, analogous lines drawn with draw-lines in dc<%>.

When a path is filled or used as a region, the open sub-path (if any) is treated as if it were closed. The content of a path is determined either through the 'even-odd rule or the 'wind-

ing rule, as selected at the time when the path is filled or used to generate a region.

A path is not connected to any particular dc<%> object, so setting a dc<%> origin or scale does not affect path operations. Instead, a dc<%>'s origin and scale apply at the time that the path is drawn or used to set a region.

```
(new dc-path%) → (is-a?/c dc-path%)
```

Creates a new path that contains no sub-paths (and no open sub-path).

```
(send a-dc-path append path) → void?
  path : (is-a?/c dc-path%)
```

Adds the sub-paths of *path* to *a-dc-path*. Closed sub-paths of *path* are added as closed sub-paths to *a-dc-path*. If both paths have an open sub-path, then this path's sub-path is extended by the given path's open sub-path, adding a line from this path's current ending point to the given path's starting point. If only one of the paths has an open sub-path, then it becomes (or remains) this path's open sub-path.

```
(send a-dc-path arc  x
                     y
                     width
                     height
                     start-radians
                     end-radians
                    [counter-clockwise?]) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  start-radians : real?
  end-radians : real?
  counter-clockwise? : any/c = #t
```

Extends or starts the path's open sub-path with a curve that corresponds to a section of an ellipse. The ellipse is the one bounded by a rectangle whose top-left corner is (*x*, *y*) and whose dimensions are *width* by *height*. The ellipse section starts a the angle *start-radians* (0 is three o'clock and half-pi is twelve o'clock) and continues to the angle *end-radians*; if *counter-clockwise?* is true, then the arc runs counter-clockwise from *start-radians* to *end-radians*, otherwise it runs clockwise.

If the path has no open sub-path, a new one is started with the arc's starting point. Otherwise, the arc extends the existing sub-path, and the existing path is connected with a line to the arc's starting point.

(send a-dc-path close) → void?

Closes the path's open sub-path. If the path has no open sub-path, an exn:fail:contract exception is raised.

(send a-dc-path curve-to x1 y1 x2 y2 x3 y3) → void?
  x1 : real?
  y1 : real?
  x2 : real?
  y2 : real?
  x3 : real?
  y3 : real?

Extends the path's open sub-path with a Bezier curve to the given point (x3,y3), using the points (x1, y1) and (x2, y2) as control points. If the path has no open sub-path, an exn:fail:contract exception is raised.

(send a-dc-path ellipse x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))

Closes the open sub-path, if any, and adds a closed sub-path that represents an ellipse bounded by a rectangle whose top-left corner is (x, y) and whose dimensions are width by height. (This convenience method is implemented in terms of close and arc.)

(send a-dc-path get-bounding-box) → real? real? real? real?

Returns a rectangle that encloses the path's points. The return values are the left, top, width, and height of the rectangle.

For curves within the path, the bounding box enclosed the two control points as well as the start and end points. Thus, the bounding box does not always tightly bound the path.

(send a-dc-path line-to x y) → void?
  x : real?
  y : real?

Extends the path's open sub-path with a line to the given point. If the path has no open sub-path, an exn:fail:contract exception is raised.

```
(send a-dc-path lines points
                     [xoffset
                      yoffset]) → void?
  points : (listof (is-a?/c point%))
  xoffset : real? = 0
  yoffset : real? = 0
```

Extends the path's open sub-path with a sequences of lines to the given points. If the path has no open sub-path, an `exn:fail:contract` exception is raised. (This convenience method is implemented in terms of `line-to`.)

```
(send a-dc-path move-to x y) → void?
  x : real?
  y : real?
```

After closing the open sub-path, if any, starts a new open sub-path with the given initial point.

```
(send a-dc-path open?) → boolean?
```

Returns `#t` if the path has an open sub-path, `#f` otherwise.

```
(send a-dc-path rectangle x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Closes the open sub-path, if any, and adds a closed path that represents a rectangle whose top-left corner is (*x*, *y*) and whose dimensions are *width* by *height*. (This convenience method is implemented in terms of `close`, `move-to`, and `line-to`.)

```
(send a-dc-path reset) → void?
```

Removes all sub-paths of the path.

```
(send a-dc-path reverse) → void?
```

Reverses the order of all points in all sub-paths. If the path has an open sub-path, the starting point becomes the ending point, and extensions to the open sub-path build on this new ending point. Reversing a closed sub-path affects how it combines with other sub-paths when determining the content of a path in `'winding` mode.

```
(send a-dc-path rotate radians) → void?
  radians : real?
```

Adjusts all points within the path (including all sub-paths), rotating them *radians* counter-clockwise around (0, 0). Future additions to the path are not rotated by this call.

```
(send a-dc-path rounded-rectangle  x
                                   y
                                   width
                                   height
                                  [radius]) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  radius : real? = -0.25
```

Closes the open sub-path, if any, and adds a closed sub-path that represents a round-cornered rectangle whose top-left corner is (*x* *y*) and whose dimensions are *width* by *height*. (This convenience method is implemented in terms of `close`, `move-to`, `arc`, and `line-to`.)

If *radius* is positive, the value is used as the radius of the rounded corner. If *radius* is negative, the absolute value is used as the *proportion* of the smallest dimension of the rectangle.

If *radius* is less than `-0.5` or more than half of *width* or *height*, an `exn:fail:contract` exception is raised.

```
(send a-dc-path scale x y) → void?
  x : real?
  y : real?
```

Adjusts all points within the path (including all sub-paths), multiplying each x-coordinate by *x* and each y-coordinate by *y*. Scaling by a negative number flips the path over the corresponding axis. Future additions to the path are not scaled by this call.

```
(send a-dc-path translate x y) → void?
  x : real?
  y : real?
```

Adjusts all points within the path (including all sub-paths), shifting then *x* to the right and *y* down. Future additions to the path are not translated by this call.

2.3.9 font%

---

font% : class?
  superclass: object%

A *font* is an object which determines the appearance of text, primarily when drawing text to a device context. A font is determined by seven properties:

- size — The size of the text, either in points (the default) or logical drawing units, depending on the "size-in-pixels?" property (see below).
- family — A platform- and device-independent font designation. The families are:
  - 'default
  - 'decorative
  - 'roman
  - 'script
  - 'swiss
  - 'modern (fixed width)
  - 'symbol (Greek letters and more)
  - 'system (used to draw control labels)

  The 'symbol designation is special under Mac OS X and X with fontconfig/Xft; characters in the ASCII range 0-255 are converted to Unicode characters that match Adobe symbols. For example, a is converted to α.
- face — A string face name, such as "Courier" (under Windows and Mac OS X), "-*-courier" (under X), or " Luxi Sans" (under X with fontconfig/Xft; note the leading space). The format and meaning of a face name is platform- and device-specific. If a font's face name is #f, then the font's appearance depends only on the family. If a face is provided but no mapping is available for the face name (for a specific platform or device), then the face name is ignored and the family is used. See font-name-directory<%> for information about how face names are mapped for drawing text.
- style — The slant style of the font, one of:
  - 'normal
  - 'slant (Windows, Mac OS X: same as 'italic; X: tries 'italic if 'slant font does not exist)
  - 'italic (X: tries 'slant if 'italic font does not exist)
- weight — The weight of the font, one of:

  - 'normal
  - 'light
  - 'bold

- underline? — #t for underlined, #f for plain.
- smoothing — Amount of anti-alias smoothing, one of:
  - 'default (platform-specific, sometimes user-configurable)
  - 'partly-smoothed (Windows: TrueType when available; Mac OS X: 4-bit, pixel-aligned smoothing; X: fontconfig/Xft when available)
  - 'smoothed (Windows: ClearType when available, XP and up; Mac OS X: Quartz smoothing; X: fontconfig/Xft when available)
  - 'unsmoothed

  Special case: 'default corresponds to 'partly-smoothed when used with the 'modern family and a font size between 9 and 13 (inclusive).
- size-in-pixels? — #t if the size of the font is in logical drawing units (i.e., pixels for an unscaled screen or bitmap drawing context), #f if the size of the font is in points (which can depend on screen resolution).

To avoid creating multiple fonts with the same characteristics, use the global font-list% object the-font-list.

See also font-name-directory<%>.

```
(make-object font%) → (is-a?/c font%)
(make-object font%  size
                    family
                   [style
                    weight
                    underline?
                    smoothing
                    size-in-pixels?]) → (is-a?/c font%)
  size : (integer-in 1 255)
  family : (one-of/c 'default 'decorative 'roman 'script
                     'swiss 'modern 'symbol 'system)
  style : (one-of/c 'normal 'italic 'slant) = 'normal
  weight : (one-of/c 'normal 'bold 'light) = 'normal
  underline? : any/c = #f
  smoothing : (one-of/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
            = 'default
  size-in-pixels? : any/c = #f
```

```
(make-object font%  size
                    face
                    family
                   [style
                    weight
                    underline?
                    smoothing
                    size-in-pixels?]) → (is-a?/c font%)
  size : (integer-in 1 255)
  face : string?
  family : (one-of/c 'default 'decorative 'roman 'script
                     'swiss 'modern 'symbol 'system)
  style : (one-of/c 'normal 'italic 'slant) = 'normal
  weight : (one-of/c 'normal 'bold 'light) = 'normal
  underline? : any/c = #f
  smoothing : (one-of/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
            = 'default
  size-in-pixels? : any/c = #f
```

When no arguments are provided, creates an instance of the default font. If no face name is provided, the font is created without a face name. Otherwise, see font-name-directory<%> for information about the way *face* is interpreted for drawing text on various platforms and devices; when a platform- or device-specific interpretation of *face* is not available, the *family* is used to draw text.

See font% for information about *family*, *style*, and *weight*. font-name-directory<%>.

```
(send a-font get-face) → (or/c string? false/c)
```

Gets the font's face name, or #f if none is specified.

```
(send a-font get-family)
 → (one-of/c 'default 'decorative 'roman 'script
             'swiss 'modern 'symbol 'system)
```

Gets the font's family. See font% for information about families.

```
(send a-font get-font-id) → exact-integer?
```

Gets the font's ID, for use with a font-name-directory<%>. The ID is determined by the font's face and family specifications, only.

```
(send a-font get-point-size) → (integer-in 1 255)
```

Gets the font's size (roughly the height). Despite the method name, the size may be in logical units instead of points, depending on the result of get-size-in-pixels.

Due to space included in a font by a font designer, a font tends to generate text that is slightly taller than the nominal size.

(send a-font get-size-in-pixels) → boolean?

Returns #t if the size reported by get-point-size is in logical drawing units, #f if it is in points.

For a size in points and a screen or bitmap drawing context, the logical height depends on the resolution of the screen.

```
(send a-font get-smoothing)
 → (one-of/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
```

Gets the font's anti-alias smoothing mode. See font% for information about smoothing.

(send a-font get-style) → (one-of/c 'normal 'italic 'slant)

Gets the font's slant style. See font% for information about styles.

(send a-font get-underlined) → boolean?

Returns #t if the font is underlined or #f otherwise.

(send a-font get-weight) → (one-of/c 'normal 'bold 'light)

Gets the font's weight. See font% for information about weights.

```
(send a-font screen-glyph-exists? c
                                  [for-label?]) → boolean?
  c : char
  for-label? : any/c = #f
```

Returns #t if the given character has a corresponding glyph when drawing to the screen or a bitmap, #f otherwise.

If the second argument is true, the result indicates whether the glyph is available for control labels. Otherwise, it indicates whether the glyph is available for dc<%> drawing.

For dc<%> drawing, due to automatic font substitution when drawing or measuring text, the

result of this method does not depend on this font's attributes (size, face, etc.). The font's attributes merely provide a hint for the glyph search.

See also glyph-exists? .

2.3.10 font-list%

---

font-list% : class?
  superclass: object%

A font-list% object maintains a list of font% objects to avoid repeatedly creating fonts.

A global font list, the-font-list, is created automatically.

```
(new font-list%) → (is-a?/c font-list%)
```

Creates an empty font list.

```
(send a-font-list find-or-create-font  size
                                       family
                                       style
                                       weight
                                      [underline?
                                       smoothing
                                       size-in-pixels?])
 → (is-a?/c font%)
  size : (integer-in 1 255)
  family : (one-of/c 'default 'decorative 'roman 'script
                     'swiss 'modern 'symbol 'system)
  style : (one-of/c 'normal 'italic 'slant)
  weight : (one-of/c 'normal 'bold 'light)
  underline? : any/c = #f
  smoothing : (one-of/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
            = 'default
  size-in-pixels? : any/c = #f
(send a-font-list find-or-create-font  size
                                       face
                                       family
                                       style
                                       weight
                                      [underline
                                       smoothing
                                       size-in-pixels?])
```

```
→ (is-a?/c font%)
 size : (integer-in 1 255)
 face : string?
 family : (one-of/c 'default 'decorative 'roman 'script
                    'swiss 'modern 'symbol 'system)
 style : (one-of/c 'normal 'italic 'slant)
 weight : (one-of/c 'normal 'bold 'light)
 underline : any/c = #f
 smoothing : (one-of/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
           = 'default
 size-in-pixels? : any/c = #f
```

Finds an existing font in the list or creates a new one (that is automatically added to the list). The arguments are the same as for creating a font% instance.

2.3.11 font-name-directory<%>

---

font-name-directory<%> : interface?

There is one font-name-directory<%> object: the-font-name-directory. It implements the mapping from font specifications (face, family, style, and weight) to information for rendering text on a specific device. The mapping is different for each platform. For example, when drawing to a bitmap in Windows, the rendering information is simply the name of a Windows font. When drawing to a PostScript file, the rendering information is a PostScript font name, which encapsulates the style and weight. When drawing to a bitmap in X, the rendering information is an X font string, which encapsulates the style and weight, parameterized over the size (using a "%d" placeholder).

Programmers rarely need to directly invoke methods of the-font-name-directory. It is used automatically when drawing text to a dc<%> object. Nevertheless, the-font-name-directory is available so that programmers can query or modify the mapping manually. A programmer may also need to understand how the face-and-family mapping works.

To extract mapping information from the-font-name-directory, first obtain a *font ID*, which is an index based on a family and optional face string. Font IDs are returned by find-or-create-font-id and get-font-id . A Font ID can be combined with a weight and style to obtain a specific mapping value via get-screen-name or get-post-script-name.

For a family without a face string, the corresponding font ID has a useful built-in mapping for every platform and device. (The built-in mapping can be overridden through the user's preferences; see §3.2 "Font Configuration" for information.) For a family with a face string, the-font-name-directory interprets the string (in a platform-specific way) to generate a mapping for "screen" drawing (to a canvas's dc<%>, a bitmap-dc%, or a printer-dc%).

When drawing to a post-script-dc% object, the face-specific mapping defaults to the family's mapping.

Under Windows and Mac OS X, a face name is interpreted simply as a system font name for drawing to the screen, bitmap, or printer. The mapping succeeds if the system provides a font with the given name, and fails otherwise. For example, under Windows, "MS Sans\n Serif" maps to the font that is typically used for button labels. Under X, a face name has a more complex interpretation:

- If the string begins with a space, then the remainder of the string is interpreted as a fontconfig/Xft font name, but only if fontconfig/Xft support is enabled at compile time (which is the default when available), and only if the RENDER extension is available at run time. Multiple fontconfig/Xft font names can appear after the initial space, separated by commas; the first available font is used to draw text, and later fonts are substituted for missing characters in earlier fonts.
- If the string begins with +, then the remainder of the string is interpreted as an X font name. These names are usually long, such as +-b&h-lucidatypewriter-medium-r-normal-sans-24-240-75-75-m-140-iso8859-1. As usual for X font names, asterisks may appear in the string as wildcards. Furthermore, the size of the font can be parameterized by using %d in the place of a specific point size; if an asterisk appears in place of the pixel size, the asterisk and %d are swapped when the font size is specified in pixels (otherwise the size is always interpreted as points). For rotated text, %d will be replaced by a transformation matrix.
- A string of the form -⟨*provider*⟩-⟨*font*⟩ is equivalent to +-⟨*provider*⟩-⟨*font*⟩-⟨*weight*⟩-⟨*style*⟩-normal-*-*-%d-*-*-*-*-*-*-*, where ⟨*weight*⟩ is either medium, light, or bold (depending on the requested weight mapping) and ⟨*style*⟩ is either r, i, or i (depending on the requested style mapping).
- A string of the form -⟨*font*⟩ is equivalent to -*-⟨*font*⟩.
- A string of any other format is interpreted as an X font name, optionally parameterized with %d.

The mapping for face names can be overridden (on all platforms) through the user's preferences, as described in §3.2 "Font Configuration".

```
(send a-font-name-directory find-family-default-font-id family)
 → exact-integer?
  family : (one-of/c 'default 'decorative 'roman 'script
                     'swiss 'modern 'symbol 'system)
```

Gets the font ID representing the default font for a family. See font% for information about font families.

```
(send a-font-name-directory find-or-create-font-id name
                                                    family)
 → exact-integer?
  name : string?
  family : (one-of/c 'default 'decorative 'roman 'script
                     'swiss 'modern 'symbol 'system)
```

Gets the face name for a font ID, initializing the mapping for the face name if necessary.

Font ID are useful only as mapping indices for the-font-name-directory.

```
(send a-font-name-directory get-face-name font-id)
 → (or/c string? false/c)
  font-id : exact-integer?
```

Gets the face name for a font ID. If the font ID corresponds to the default font for a particular family, #f is returned.

```
(send a-font-name-directory get-family font-id)
 → (one-of/c 'default 'decorative 'roman 'script
             'swiss 'modern 'symbol 'system)
  font-id : exact-integer?
```

Gets the family for a font ID. See font% for information about font families.

```
(send a-font-name-directory get-font-id name
                                       family)
 → exact-integer?
  name : string?
  family : (one-of/c 'default 'decorative 'roman 'script
                     'swiss 'modern 'symbol 'system)
```

Gets the font ID for a face name paired with a default family. If the mapping for the given pair is not already initialized, 0 is returned. See also find-or-create-font-id.

Font ID are useful only as mapping indices for the-font-name-directory.

```
(send a-font-name-directory get-post-script-name font-id
                                                 weight
                                                 style)
 → (or/c string? false/c)
  font-id : exact-integer?
  weight : (one-of/c 'normal 'bold 'light)
```

```
  style : (one-of/c 'normal 'italic 'slant)
```

Gets a PostScript font name for a font ID, weight, and style combination. The PostScript font name is used both for the font name in PostScript output (sans character set) and as the Adobe Font Metrics file name; see also §3.2.6 “PostScript Fonts”.

See font% for information about *weight* and *style*.

```
(send a-font-name-directory get-screen-name font-id
                                            weight
                                            style)
 → (or/c string? false/c)
  font-id : exact-integer?
  weight : (one-of/c 'normal 'bold 'light)
  style : (one-of/c 'normal 'italic 'slant)
```

Gets a platform-dependent screen font name (used for drawing to a canvas’s dc<%>, a bitmap-dc%, or a printer-dc%) for a font ID, weight, and style combination.

See font% for information about *weight* and *style*.

```
(send a-font-name-directory set-post-script-name font-id
                                                 weight
                                                 style
                                                 name)
 → void?
  font-id : exact-integer?
  weight : (one-of/c 'normal 'bold 'light)
  style : (one-of/c 'normal 'italic 'slant)
  name : string?
```

Sets a PostScript font name for a font ID, weight, and style combination. See also get-post-script-name.

See font% for information about *weight* and *style*.

```
(send a-font-name-directory set-screen-name font-id
                                            weight
                                            style
                                            name)    → void?
  font-id : exact-integer?
  weight : (one-of/c 'normal 'bold 'light)
  style : (one-of/c 'normal 'italic 'slant)
  name : string?
```

Sets a platform-dependent screen font name (used for drawing to a canvas's dc<%>, a bitmap-dc%, or a printer-dc%) for a font ID, weight, and style combination.

Under X, if the screen name contains %d, it is replaced by the size of the font (point size times 10) to obtain the full screen font name.

See font% for information about *weight* and *style*.

2.3.12 gl-config%

---

gl-config% : class?
  superclass: object%

A gl-config% object encapsulates configuration information for an OpenGL drawing context. Use a gl-config% object as an initialization argument for canvas%, or provide it to set-gl-config in bitmap%.

(new gl-config%) → (is-a?/c gl-config%)

Creates a GL configuration that indicates double buffering, a depth buffer of size one, no stencil buffer, no accumulation buffer, no multisampling, and not stereo.

(send a-*gl-config* get-accum-size) → (integer-in 0 256)

Reports the accumulation-buffer size (for each of red, green, blue, and alpha) that the configuration requests, where zero means no accumulation buffer is requested.

(send a-*gl-config* get-depth-size) → (integer-in 0 256)

Reports the depth-buffer size that the configuration requests, where zero means no depth buffer is requested.

(send a-*gl-config* get-double-buffered) → boolean?

Reports whether the configuration requests double buffering or not.

(send a-*gl-config* get-multisample-size) → (integer-in 0 256)

Reports the multisampling size that the configuration requests, where zero means no multisampling is requested.

```
(send a-gl-config get-stencil-size) → (integer-in 0 256)
```

Reports the stencil-buffer size that the configuration requests, where zero means no stencil buffer is requested.

```
(send a-gl-config get-stereo) → boolean?
```

Reports whether the configuration requests stereo or not.

```
(send a-gl-config set-accum-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular accumulation-buffer size for every channel (red, green, blue, and alpha), where zero means no accumulation buffer is requested.

```
(send a-gl-config set-depth-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular depth-buffer size, where zero means no depth buffer is requested.

```
(send a-gl-config set-double-buffered on?) → void?
  on? : any/c
```

Adjusts the configuration to request double buffering or not.

```
(send a-gl-config set-multisample-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular multisample size, where zero means no multisampling is requested. If a multisampling context is not available, this request will be ignored.

```
(send a-gl-config set-stencil-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular stencil-buffer size, where zero means no stencil buffer is requested.

```
(send a-gl-config set-stereo on?) → void?
  on? : any/c
```

Adjusts the configuration to request stereo or not.

2.3.13 gl-context<%>

---

gl-context<%> : interface?

A gl-context<%> object represents a context for drawing with OpenGL to a specific dc<%> instance. To obtain a gl-context<%> object, call get-gl-context of the target drawing context.

Only canvas dc<%> and bitmap-dc% objects support OpenGL (always under Windows and Mac OS X, sometimes under X), and in the case of a bitmap-dc%, the context is usable only when the target bitmap is non-monochrome. When the target bitmap for a bitmap-dc% context is changed via set-bitmap, the associated OpenGL context is reset, but the gl-context<%> keeps its identity. Canvas contexts are double buffered, and bitmap contexts are single buffered.

The racket/gui/base library provides no OpenGL routines. Instead, they must be obtained from a separate library, such as sgl. The facilities in racket/gui/base merely manage the current OpenGL context, connecting it to windows and bitmaps.

Only one OpenGL context can be active at a time across all threads and eventspaces. Except under Mac OS X, OpenGL contexts are not protected against interference among threads; that is, if a thread selects one of its OpenGL contexts, then other threads can write into the context via OpenGL commands. However, if all threads issue OpenGL commands only within a thunk passed to call-as-current, then drawing from the separate threads will not interfere, because call-as-current uses a lock to serialize context selection across all threads in Racket.

```
(send a-gl-context call-as-current  thunk
                                    [alternate
                                     enable-breaks?]) → any/c
  thunk : (-> any)
  alternate : evt? = never-evt
  enable-breaks? : any/c = #f
```

Calls a thunk with this OpenGL context as the current context for OpenGL commands.

The method blocks to obtain a lock that protects the global OpenGL context, and it releases the lock when the thunk returns or escapes. The lock is re-entrant, so a nested use of the method in the same thread with the same OpenGL context does not obtain or release the lock.

The lock prevents interference among OpenGL-using threads. If a thread is terminated while

holding the context lock, the lock is released. Continuation jumps into the thunk do not grab the lock or set the OpenGL context. See gl-context<%> for more information on interference.

The method accepts an alternate synchronizable event for use while blocking for the context lock; see also sync.

The result of the method call is the result of the thunk if it is called, or the result of the alternate event if it is chosen instead of the context lock.

If ok? returns #f at the time that this method is called, then an exn:fail:contract exception is raised.

If *enable-breaks?* is true, then the method uses sync/enable-break while blocking for the context-setting lock instead of sync.

(send *a-gl-context* ok?) → boolean?

Returns #t if this context is available OpenGL drawing, #f otherwise.

A context is unavailable if OpenGL support is disabled at compile time or run time, if the context is associated with a bitmap-dc% with no selected bitmap or with a monochrome selected bitmap, if the context is for a canvas that no longer exists, or if there was a low-level error when preparing the context.

(send *a-gl-context* swap-buffers) → void?

Swaps the front (visible) and back (OpenGL-drawing) buffer for a context associated with a canvas, and has no effect on a bitmap context.

This method implicitly uses call-as-current to obtain the context lock. Since the lock is re-entrant, however, the swap-buffers method can be safely used within a call-as-current thunk.

2.3.14 pen%

---

pen% : class?
  superclass: object%

A pen is a drawing tool with a color, width, and style. A pen draws lines and outlines, such as the outline of a rectangle. On a monochrome display, all non-white pens are drawn as black.

In addition to its color, width, and style, a pen can have a stipple bitmap that is a 8 x 8 monochrome bitmap. This stipple is used only in unsmoothed mode (see `set-smoothing`) or in a PostScript drawing context. Painting with a stipple pen is similar to calling `draw-bitmap` with the stipple bitmap in region painted by the pen.

A pen's style is one of the following:

- `'transparent` — Draws with no effect (on the outline of the drawn shape).
- `'solid` — Draws using the pen's color. If a (monochrome) stipple is installed into the pen, black pixels from the stipple are transferred to the destination using the brush's color, and white pixels from the stipple are not transferred.
- `'xor` — In unsmoothed mode, the pen's color or colored stipple is xor-ed with existing destination pixel values. The `'xor` mapping is unspecified for arbitrary color combinations, but the mapping provides two guarantees:
  - Black-and-white drawing to a color or monochrome destination always works as expected: black xor white = black, white xor black = black, black xor black = white, and white xor white = white.
  - Performing the same drawing operation twice in a row with `'xor` is equivalent to a no-op.

  In a smoothing mode, `'xor` is equivalent to `'solid`.
- `'hilite` — In unsmoothed mode, existing destination pixels are "highlighted" in a platform-specific way when the pen color is black. Under Windows for a color drawing context, the inverted RGB components of destination pixel are combined with the RGB components of the system-wide highlight color using a bitwise "or", and the combination is used. Under Mac OS X for a color drawing context, the inverted RGB components of the system-wide highlight color are subtracted from the RGB components of each destination pixel, and the difference (or 0 for a negative result) is used. Under X or for any monochrome drawing context, `'hilite` is the same as `'xor`. In a smoothing mode, `'hilite` is treated like `'solid`.
- The following special pen modes use the pen's color, and they only apply when a stipple is not used:
  - `'dot`
  - `'long-dash`
  - `'short-dash`
  - `'dot-dash`
  - `'xor-dot`
  - `'xor-long-dash`
  - `'xor-short-dash`
  - `'xor-dot-dash`

To avoid creating multiple pens with the same characteristics, use the global pen-list% object the-pen-list, or provide a color, width, and style to set-pen in dc<%>.

A pen of size 0 uses the minimum line size for the destination drawing context. In (unscaled) canvases and bitmaps in unsmoothed mode, a zero-width pen behaves the nearly same as a pen of size 1. In a smoothing mode (including all post-script-dc% drawing), a pen of size 0 draws a line thinner than a pen of size 1. If the pen's width is not an integer, then the width is truncated to an integer (even before scaling) in unsmoothed mode.

```
(make-object pen%) → (is-a?/c pen%)
(make-object pen% color width style) → (is-a?/c pen%)
  color : (is-a?/c color%)
  width : (real-in 0 255)
  style : (one-of/c 'transparent 'solid 'xor 'hilite
                    'dot 'long-dash 'short-dash 'dot-dash
                    'xor-dot 'xor-long-dash 'xor-short-dash
                    'xor-dot-dash)
(make-object pen% color-name width style) → (is-a?/c pen%)
  color-name : string?
  width : (real-in 0 255)
  style : (one-of/c 'transparent 'solid 'xor 'dot 'hilite
                    'long-dash 'short-dash 'dot-dash
                    'xor-dot 'xor-long-dash 'xor-short-dash
                    'xor-dot-dash)
```

When no argument are provided, the result is a solid black pen of width 0. Otherwise, the result is a pen with the given color, width, and style. For the case that the color is specified using a name, see color-database<%> for information about color names; if the name is not known, the pen's color is black.

```
(send a-pen get-cap) → (one-of/c 'round 'projecting 'butt)
```

Returns the pen cap style (Windows unsmoothed, X unsmoothed, all smoothing). The default is 'round.

```
(send a-pen get-color) → (is-a?/c color%)
```

Returns the pen's color object.

```
(send a-pen get-join) → (one-of/c 'round 'bevel 'miter)
```

Returns the pen join style (Windows unsmoothed, X unsmoothed, all smoothing). The default is 'round.

```
(send a-pen get-stipple) → (or/c (is-a?/c bitmap%) false/c)
```

Gets the current stipple bitmap, or returns `#f` if no stipple bitmap is installed.

```
(send a-pen get-style)
 → (one-of/c 'transparent 'solid 'xor 'hilite
             'dot 'long-dash 'short-dash 'dot-dash
             'xor-dot 'xor-long-dash 'xor-short-dash
             'xor-dot-dash)
```

Returns the pen style. See `pen%` for information about possible styles.

```
(send a-pen get-width) → (real-in 0 255)
```

Returns the pen width.

```
(send a-pen set-cap cap-style) → void?
  cap-style : (one-of/c 'round 'projecting 'butt)
```

Sets the pen cap style (Windows unsmoothed, X unsmoothed, all smoothing). See `get-cap` for information about cap styles.

A pen cannot be modified if it was obtained from a `pen-list%` or while it is selected into a drawing context.

```
(send a-pen set-color color) → void?
  color : (is-a?/c color%)
(send a-pen set-color color-name) → void?
  color-name : string?
(send a-pen set-color red green blue) → void?
  red : (integer-in 0 255)
  green : (integer-in 0 255)
  blue : (integer-in 0 255)
```

Sets the pen color.

A pen cannot be modified if it was obtained from a `pen-list%` or while it is selected into a drawing context.

```
(send a-pen set-join join-style) → void?
  join-style : (one-of/c 'round 'bevel 'miter)
```

Sets the pen join style (Windows unsmoothed, X unsmoothed, all smoothing). See get-join for information about join styles.

A pen cannot be modified if it was obtained from a pen-list% or while it is selected into a drawing context.

```
(send a-pen set-stipple stipple) → void?
  stipple : (or/c (is-a?/c bitmap%) false/c)
```

Sets the pen stipple bitmap, which must be an 8 x 8 monochrome bitmap or #f, which turns off the stipple bitmap.

A bitmap cannot be used as a stipple if it is selected into a bitmap-dc% object; if the given bitmap is selected into a bitmap-dc% object, an exn:fail:contract exception is raised. A pen cannot be modified if it was obtained from a pen-list% or while it is selected into a drawing context.

A pen's stipple is not used in a smoothing mode, except for a post-script-dc% (which is always in smoothed mode).

```
(send a-pen set-style style) → void?
  style : (one-of/c 'transparent 'solid 'xor 'hilite
                    'dot 'long-dash 'short-dash 'dot-dash
                    'xor-dot 'xor-long-dash 'xor-short-dash
                    'xor-dot-dash)
```

Sets the pen style. See pen% for information about the possible styles.

A pen cannot be modified if it was obtained from a pen-list% or while it is selected into a drawing context.

```
(send a-pen set-width width) → void?
  width : (real-in 0 255)
```

Sets the pen width.

A pen cannot be modified if it was obtained from a pen-list% or while it is selected into a drawing context.

2.3.15 pen-list%

```
pen-list% : class?
```

superclass: object%

A pen-list% object maintains a list of pen% objects to avoid repeatedly creating pen objects. A pen% object in a pen list cannot be mutated.

A global pen list the-pen-list is created automatically.

```
(new pen-list%) → (is-a?/c pen-list%)
```

Creates an empty pen list.

```
(send a-pen-list find-or-create-pen color
                                    width
                                    style) → (is-a?/c pen%)
  color : (is-a?/c color%)
  width : (real-in 0 255)
  style : (one-of/c 'transparent 'solid 'xor 'hilite
                    'dot 'long-dash 'short-dash 'dot-dash
                    'xor-dot 'xor-long-dash 'xor-short-dash
                    'xor-dot-dash)
(send a-pen-list find-or-create-pen color-name
                                    width
                                    style)
 → (or/c (is-a?/c pen%) false/c)
  color-name : string?
  width : (real-in 0 255)
  style : (one-of/c 'transparent 'solid 'xor 'hilite
                    'dot 'long-dash 'short-dash 'dot-dash
                    'xor-dot 'xor-long-dash 'xor-short-dash
                    'xor-dot-dash)
```

Finds a pen of the given specification, or creates one and adds it to the list. The arguments are the same as for creating a pen% instance. When *color-name* is provided, however, the return value is #f when no color matching *color-name* can be found in the-color-database.

2.3.16 point%

```
point% : class?
  superclass: object%
```

A point% is used for certain drawing commands. It encapsulates two real numbers.

```
(make-object point%) → (is-a?/c point%)
(make-object point% x y) → (is-a?/c point%)
  x : real?
  y : real?
```

Creates a point. If *x* and *y* are not supplied, they are set to 0.

```
(send a-point get-x) → real?
```

Gets the point x-value.

```
(send a-point get-y) → real?
```

Gets the point y-value.

```
(send a-point set-x x) → void?
  x : real?
```

Sets the point x-value.

```
(send a-point set-y y) → void?
  y : real?
```

Sets the point y-value.

2.3.17 post-script-dc%

```
post-script-dc% : class?
  superclass: object%
  extends: dc<%>
```

A post-script-dc% object is a PostScript device context, that can write PostScript files on any platform. See also ps-setup%.

Be sure to use the following methods to start/end drawing:

- start-doc
- start-page
- end-page

- end-doc

Attempts to use a drawing method outside of an active page raises an exception.

See also printer-dc%.

```
(new post-script-dc% [[interactive interactive]
                      [parent parent]
                      [use-paper-bbox use-paper-bbox]
                      [as-eps as-eps]])
 → (is-a?/c post-script-dc%)
  interactive : any/c = #t
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
  use-paper-bbox : any/c = #f
  as-eps : any/c = #t
```

If *interactive* is true, the user is given a dialog for setting printing parameters (see get-ps-setup-from-user); the resulting configuration is installed as the current configuration). If the user chooses to print to a file (the only possibility under Windows and Mac OS X), another dialog is given to select the filename. If the user hits cancel in either of these dialogs, then ok? returns #f.

If *parent* is not #f, it is used as the parent window of the configuration dialog.

If *interactive* is #f, then the settings returned by current-ps-setup are used. A file dialog is still presented to the user if the get-file method returns #f, and the user may hit cancel in that case so that ok? returns #f.

If *use-paper-bbox* is #f, then the PostScript bounding box for the output is determined by drawing commands issued to the object; such a bounding box encloses all parts of the drawing *ignoring* clipping regions (so the bounding box may be approximate). If *use-paper-bbox* is not #f, then the bounding box is determined by the current paper size (as specified by current-ps-setup), and the bounding box does not include the margin (also specified by current-ps-setup).

If as-eps is #f, then the generated PostScript does not include an Encapsulated PostScript (EPS) header, and instead includes a generic PostScript header. Otherwise, the generated PostScript includes a header that identifiers it as EPS.

See also ps-setup% and current-ps-setup. The settings for a particular post-script-dc% object are fixed to the values in the current configuration when the object is created (after the user has interactively adjusted them when *interactive* is true).

2.3.18 printer-dc%

---

printer-dc% : class?
  superclass: object%
  extends: dc<%>

A printer-dc% object is a Windows or Mac OS X printer device context. The class cannot be instantiated under X (an exn:misc:unsupported exception is raised).

Under Mac OS X, a newly created printer-dc% object obtains orientation (portrait versus landscape) and scaling information from the current ps-setup% object, as determined by the current-ps-setup parameter. This information can be configured by the user through a dialog shown by get-page-setup-from-user.

Be sure to use the following methods to start/end drawing:

- start-doc
- start-page
- end-page
- end-doc

Attempts to use a drawing method outside of an active page raises an exception.

See also post-script-dc%.

When a printer-dc% object is created, the user gets platform-specific modal dialogs for configuring the output. If the user cancels the dialog, the ok? method of the object returns #f.

```
(new printer-dc% [[parent parent]]) → (is-a?/c printer-dc%)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) false/c)
         = #f
```

If *parent* is not #f, it is used as the parent window of the configuration dialog.

2.3.19 ps-setup%

---

ps-setup% : class?
  superclass: object%

A ps-setup% object contains configuration information for producing PostScript files using a post-script-dc% object. To a lesser extent, it contains information for printing with a printer-dc% object.

When a post-script-dc% object is created, its configuration is determined by the current-ps-setup parameter's ps-setup% value. After a post-script-dc% object is created, it is unaffected by changes to the current-ps-setup parameter or mutations to the ps-setup% object.

```
(new ps-setup%) → (is-a?/c ps-setup%)
```

Creates a new ps-setup% object with the (platform-specific) default configuration.

```
(send a-ps-setup copy-from source) → void?
  source : (is-a?/c ps-setup%)
```

Copies the settings copy-from to *a-ps-setup*.

```
(send a-ps-setup get-command) → string?
```

Gets the printer command used to print a file in X. The default is "lpr". This value is not used by other platforms.

```
(send a-ps-setup get-editor-margin h-margin
                                   v-margin) → void?
  h-margin : (box/c nonnegative-real?)
  v-margin : (box/c nonnegative-real?)
```

Returns the current settings for horizontal and vertical margins when printing an editor<%>. See also set-editor-margin.

```
(send a-ps-setup get-file) → (or/c path-string? false/c)
```

Gets the PostScript output filename. A #f value (the default) indicates that the user should be prompted for a filename when a post-script-dc% object is created.

```
(send a-ps-setup get-level-2) → boolean?
```

Reports whether Level 2 commands are output in PostScript files.

Currently, Level 2 commands are only needed to include color bitmap images in PostScript output (drawn with draw-bitmap), or bitmap pen and brush stipples. When Level 2 commands are disabled, bitmaps are converted to grayscale images and stipples are not sup-

ported.

```
(send a-ps-setup get-margin h-margin
                            v-margin) → void?
  h-margin : (box/c nonnegative-real?)
  v-margin : (box/c nonnegative-real?)
```

Returns the current settings for horizontal and vertical PostScript margins. See also set-margin.

```
(send a-ps-setup get-mode)
 → (one-of/c 'preview 'file 'printer)
```

Gets the printing mode that determines where output is sent: 'preview, 'file, or 'printer. The default for X is 'preview. The value in Windows and Mac OS X is always 'file.

```
(send a-ps-setup get-orientation)
 → (one-of/c 'portrait 'landscape)
```

Gets the orientation: 'portrait or 'landscape. The default is 'portrait. Unlike most other settings, this one affects native printing (via printer-dc%) as well as PostScript output.

Landscaped orientation affects the size of the drawing area as reported by get-size: the horizontal and vertical sizes determined by the selected paper type are transposed and then scaled.

```
(send a-ps-setup get-paper-name) → string?
```

Returns the name of the current paper type: "A4 210 x 297\n mm", "A3 297 x 420 mm", "Letter 8 1/2 x 11 in", or "Legal 8 1/2 x 14 in". The default is "Letter 8 1/2\n x 11 in".

The paper name determines the size of the drawing area as reported by get-size (along with landscape transformations from get-orientation and/or the scaling factors of get-scaling). It also determines the bounding box of PostScript output when a post-script-dc% context is created with a true value for the use-paper-bbox? initialization argument.

```
(send a-ps-setup get-preview-command) → string?
```

Gets the command used to view a PostScript file for X. The default is "gv". This value is not used by other platforms.

```
(send a-ps-setup get-scaling x y) → void?
  x : (box/c nonnegative-real?)
  y : (box/c nonnegative-real?)
```

Gets the scaling factor for PostScript output. The *x* box is filled with the horizontal scaling factor. The *y* box is filled with the vertical scaling factor. The default is `0.8` by `0.8`.

This scale is in addition to a scale that can be set by `set-scale` in a `post-script-dc%` context. The size reported by `get-size` is the size of the selected paper type (transposed for landscaped mode) divided by this scale.

```
(send a-ps-setup get-translation x y) → void?
  x : (box/c nonnegative-real?)
  y : (box/c nonnegative-real?)
```

Gets the translation (from the bottom left corner) for PostScript output. The *x* box is filled with the horizontal offset. The *y* box is filled with the vertical offset. The default is `0.0` and `0.0`.

The translation is not scaled by the numbers returned from `get-scaling` and the translation does not affect the size of the drawing area.

```
(send a-ps-setup set-command command) → void?
  command : string?
```

Sets the printer command used to print a file under X. See `get-command`.

```
(send a-ps-setup set-editor-margin h v) → void?
  h : exact-nonnegative-integer?
  v : exact-nonnegative-integer?
```

Sets the horizontal and vertical margins used when printing an editor with the `print` method. These margins are always used for printing, whether the drawing destination is a `post-script-dc%` or `printer-dc%`. The margins are in the units of the destination `printer-dc%` or `post-script-dc%`. In the case of `post-script-dc%` printing, the editor margin is in addition to the PostScript margin that is determined by `set-margin`.

```
(send a-ps-setup set-file filename) → void?
  filename : (or/c path-string? false/c)
```

Sets the PostScript output filename. See `get-file`.

```
(send a-ps-setup set-level-2 on?) → void?
  on? : any/c
```

Sets whether Level 2 commands are output in PostScript files. See get-level-2.

```
(send a-ps-setup set-margin h v) → void?
  h : (and/c real? (not/c negative?))
  v : (and/c real? (not/c negative?))
```

Sets the horizontal and vertical PostScript margins. When drawing to a post-script-dc%, the page size reported by get-size subtracts these margins from the normal page area (before taking into account scaling affects). In addition, drawing into the post-script-dc% produces PostScript output that is offset by the margins.

When using the output of a post-script-dc% as Encapsulated PostScript, the margin values are effectively irrelevant. Changing the margins moves the PostScript image in absolute coordinates, but it also moves the bounding box.

The margins are in unscaled post-script-dc% units, which are points. The default margins are 16 points.

```
(send a-ps-setup set-mode mode) → void?
  mode : (one-of/c 'preview 'file 'printer)
```

Sets the printing mode controlling where output is sent. See get-mode.

Under Windows and Mac OS X, if 'preview or 'printer is provided, an exn:fail:contract exception is raised.

```
(send a-ps-setup set-orientation orientation) → void?
  orientation : (one-of/c 'portrait 'landscape)
```

Sets the orientation. See get-orientation.

```
(send a-ps-setup set-paper-name type) → void?
  type : string?
```

Sets the name of the current paper type. See get-paper-name.

```
(send a-ps-setup set-preview-command command) → void?
  command : string?
```

Sets the command used to view a PostScript file under X. See get-preview-command.

```
(send a-ps-setup set-scaling x y) → void?
  x : (and/c real? (not/c negative?))
  y : (and/c real? (not/c negative?))
```

Sets the scaling factor for PostScript output. See get-scaling.

```
(send a-ps-setup set-translation x y) → void?
  x : real?
  y : real?
```

Sets the translation (from the bottom left corner) for PostScript output. See get-translation.

2.3.20 region%

```
region% : class?
  superclass: object%
```

A region% object specifies a portion of a drawing area (possibly discontinuous). It is normally used for clipping drawing operations.

Each region% object is associated to a particular dc<%> object, specified when the region is created. A region can only be used with its associated dc<%> object. The origin and scale of a drawing context determine the bounding box and drawing location of a region at the time that a region is created (or set); a region is independent of the current scale and origin when the region is used. For an auto-scrolled canvas, the canvas's current scrolling applies when the region is used (and it does not affect the region's bounding box).

See also set-clipping-region in dc<%> and get-clipping-region in dc<%>.

```
(new region% [dc dc]) → (is-a?/c region%)
  dc : (is-a?/c dc<%>)
```

Creates an empty region.

```
(send a-region get-bounding-box) → real? real? real? real?
```

Returns a rectangle that encloses the region. The return values are the left, top, width, and height of the rectangle. The bounding box is precisely correct for unsmoothed drawing, but it is only approximate for smoothed drawing.

(send *a-region* get-dc) → (is-a?/c dc<%>)

Returns the region's drawing context.

(send *a-region* in-region? *x* *y*) → boolean?
  x : real?
  y : real?

Returns #t if the given point is approximately within the region, #f otherwise. The given point is scaled and translated according to the region's dc<%>'s current scale and translation.

The approximate in-region test represents the true result for unsmoothed drawing, but it not necessarily for smoothed drawing.

(send *a-region* intersect *rgn*) → void?
  rgn : (is-a?/c region%)

Sets the region to the intersection of itself with the given region.

The DC of *rgn* and *a-region* must be the same.

The result is always reliable for unsmoothed and smoothed drawing. For smoothed drawing, an intersect corresponds to clipping with this region's path, and then clipping with the given region's path. Further combining sends to this region correspond to combination with the original path before initial clip, and further combination with this region as an argument correspond to a combination with the given path after the initial clip. Thus, an intersecting region is a poor input for union, subtract, or xor, but it intersects properly in further calls to intersect.

(send *a-region* is-empty?) → boolean?

Returns #t if the region is approximately empty, #f otherwise. An approximately empty region is truly empty for unsmoothed drawing, but it may contain points for smoothed drawing.

(send *a-region* set-arc *x*
                        y
                        width
                        height
                        start-radians
                        end-radians) → void?
  x : real?

```
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  start-radians : real?
  end-radians : real?
```

Sets the region to the interior of the specified wedge.

See also draw-ellipse in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The result is reliable for both unsmoothed and smoothed drawing. For smoothed drawing, the region corresponds to a clockwise path with a 'winding fill. The region is also *atomic* for the purposes of region combination.

```
(send a-region set-ellipse x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Sets the region to the interior of the specified ellipse.

See also draw-ellipse in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The result is reliable for both unsmoothed and smoothed drawing. For smoothed drawing, the region corresponds to a clockwise path with a 'winding fill. The region is also *atomic* for the purposes of region combination.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-region set-path path
                       [xoffset
                        yoffset
                        fill-style]) → void?
  path : (is-a?/c dc-path%)
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (one-of/c 'odd-even 'winding) = 'odd-even
```

Sets the region to the content of the given path.

See also draw-path in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The result is reliable for both unsmoothed and smoothed drawing. For smoothed drawing, the fill style affects how well the region reliably combines with other regions (via union, xor, and subtract). The region is also *atomic* for the purposes of region combination.

```
(send a-region set-polygon  points
                           [xoffset
                            yoffset
                            fill-style]) → void?
  points : (listof (is-a?/c point%))
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (one-of/c 'odd-even 'winding) = 'odd-even
```

Sets the region to the interior of the specified polygon.

See also draw-polygon in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The result is reliable for both unsmoothed and smoothed drawing. For smoothed drawing, the fill style affects how well the region reliably combines with other regions (via union, xor, and subtract). The region is also *atomic* for the purposes of region combination.

```
(send a-region set-rectangle x
                             y
                             width
                             height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Sets the region to the interior of the specified rectangle.

See also draw-rectangle in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The result is reliable for both unsmoothed and smoothed drawing. For smoothed drawing, the region corresponds to a clockwise path with a 'winding fill. The region is also *atomic* for the purposes of region combination.

Restrictions on the magnitude of drawing coordinates are described with dc<%>.

```
(send a-region set-rounded-rectangle  x
                                      y
                                      width
                                      height
                                     [radius]) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  radius : real? = -0.25
```

Sets the region to the interior of the specified rounded rectangle.

See also `draw-rounded-rectangle` in `dc<%>`, since the region content is determined the same way as brush-based filling in a `dc<%>`.

The result is reliable for both unsmoothed and smoothed drawing. For smoothed drawing, the region corresponds to a clockwise path with a `'winding` fill. The region is also *atomic* for the purposes of region combination.

Restrictions on the magnitude of drawing coordinates are described with `dc<%>`.

```
(send a-region subtract rgn) → void?
  rgn : (is-a?/c region%)
```

Sets the region to the subtraction of itself minus the given region. In other words, a point is removed from the region if it is included in the given region. (The given region may contain points that are not in the current region; such points are ignored.)

This region's DC and given region's DC must be the same.

The result is always reliable for unsmoothed drawing. For smoothed drawing, the result is consistent across platforms and devices, but it is never a true subtraction. A subtraction corresponds to combining the sub-paths of this region with the reversed sub-paths of the given region, then intersecting the result with this region. This fails as a true subtraction, because the boundary of loops (with either `'odd-even` or `'winding` filling) is ambiguous.

```
(send a-region union rgn) → void?
  rgn : (is-a?/c region%)
```

Sets the region to the union of itself with the given region.

This region's DC and given region's DC must be the same.

The result is always reliable for unsmoothed drawing. For smoothed drawing, a union cor-

responds to combining the sub-paths of each region into one path, using an 'odd-even fill if either of the region uses an 'odd-even fill (otherwise using a 'winding fill). Consequently, while the result is consistent across platforms and devices, it is a true union only for certain input regions. For example, it is a true union for non-overlapping atomic and union regions. It is also a true union for atomic and union regions (potentially overlapping) that are all clockwise and use 'winding fill.

(send *a-region* xor *rgn*) → void?
  rgn : (is-a?/c region%)

Sets the region to the xoring of itself with the given region (i.e., contains points that are enclosed by exactly one of the two regions).

This region's DC and given region's DC must be the same.

The result is always reliable for unsmoothed drawing. For smoothed drawing, the result is consistent across platforms and devices, but it is not necessarily a true xoring. An xoring corresponds to combining the sub-paths of this region with the reversed sub-paths of the given region. The result uses an 'odd-even fill if either of the region uses an 'odd-even fill (otherwise using a 'winding fill). Consequently, the result is a reliable xoring only for certain input regions. For example, it is reliable for atomic and xoring regions that all use 'even-odd fill.

2.4 Drawing Functions

2.4.1 Global Graphics

(flush-display) → void?

Under X and Mac OS X, flushes pending display messages such that the user's display reflects the actual state of the windows. Under Windows, the procedure has no effect.

(get-display-depth) → exact-nonnegative-integer?

Returns the depth of the main display (a value of 1 denotes a monochrome display).

(get-display-left-top-inset [*avoid-bars?*])
 → exact-nonnegative-integer?
   exact-nonnegative-integer?
  avoid-bars? : bool = #f

When the optional argument is #f (the default), this function returns the offset of the main

screen's origin from the top-left of the physical screen. Under X and Windows, the result is always 0 and 0; under Mac OS X, the result is 0 and the height of the menu bar.

When the optional argument is true, this function returns the amount space at the left and top of the main screen that is occupied by the task bar (Windows) or menu bar and dock (Mac OS X). Under X, the result is always 0 and 0.

```
(get-display-size [full-screen?]) → exact-nonnegative-integer?
                                    exact-nonnegative-integer?
  full-screen? : bool = #f
```

Gets the physical size of the display in pixels. Under Windows, this size does not include the task bar by default. Under Mac OS X, this size does not include the menu bar or dock area by default.

Under Windows and Mac OS X, if the optional argument is true, then the task bar, menu bar, and dock area are included in the result.

Returns the screen's width and height.

```
(is-color-display?) → boolean?
```

Returns #t if the main display has color, #f otherwise.

```
(register-collecting-blit  canvas
                           x
                           y
                           w
                           h
                           on
                           off
                          [on-x
                           on-y
                           off-x
                           off-y]) → void?
  canvas : (is-a?/c canvas%)
  x : real?
  y : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
  on : (is-a?/c bitmap%)
  off : (is-a?/c bitmap%)
  on-x : real? = 0
  on-y : real? = 0
```

```
  off-x : real? = 0
  off-y : real? = 0
```

Registers a blit to occur when garbage collection starts or ends.

When garbage collection starts, `(send (send canvas get-dc) draw-bitmap-section on on-x on-y x y w h)` is called. When garbage collection ends, `(send (send canvas get-dc) draw-bitmap-section off off-x off-y x y w h)` is called. If *canvas*'s device context has a scale, the scale may or may not be temporarily disabled during the bitmap drawing.

The *canvas* is registered weakly, so it will be automatically unregistered if the canvas becomes invisible and inaccessible. Multiple registrations can be installed for the same canvas.

See also `unregister-collecting-blit`.

```
(unregister-collecting-blit canvas) → void?
  canvas : (is-a?/c canvas%)
```

Unregisters a blit request installed with See also `register-collecting-blit`.

Unregisters all blits for *canvas*.

2.4.2 PostScript

```
(current-ps-afm-file-paths) → (listof path?)
(current-ps-afm-file-paths paths) → void?
  paths : (listof path?)
```

A parameter determines the list of paths that is used to find AFM files. See §3.2.6 "PostScript Fonts" for more information.

```
(current-ps-cmap-file-paths) → (listof path?)
(current-ps-cmap-file-paths paths) → void?
  paths : (listof path?)
```

A parameter that determines the list of paths that is used to find CMap files. See §3.2.6 "PostScript Fonts" for more information.

```
(current-ps-setup) → (is-a?/c ps-setup%)
(current-ps-setup pss) → void?
  pss : (is-a?/c ps-setup%)
```

A parameter that determines the current PostScript configuration settings. See post-script-dc% and printer-dc%.

2.4.3 Drawing Object Lists

---

the-brush-list : (is-a?/c brush-list%)

See brush-list%.

---

the-color-database : (is-a?/c color-database<%>)

See color-database<%>.

---

the-font-list : (is-a?/c font-list%)

See font-list%.

---

the-font-name-directory : (is-a?/c font-name-directory<%>)

See font-name-directory<%>.

---

the-pen-list : (is-a?/c pen-list%)

See pen-list%.

2.4.4 Fonts

(get-face-list [*family*]) → (listof string?)
  family : (one-of/c 'mono 'all) = 'all

Returns a list of font face names available on the current system. If 'mono is provided as the argument, then only faces that are known to correspond to monospace fonts are included in the list.

(get-family-builtin-face *family*) → string?
  family : (one-of/c 'default 'decorative 'roman 'script
                       'swiss 'modern 'symbol 'system)

Returns the built-in default face mapping for a particular font family. The built-in default

can be overridden via preferences, as described in §3.2 “Font Configuration”.

See font% for information about *family*.

---

menu-control-font : (is-a?/c font%)

This font is the default for popup-menu% objects.

Under Mac OS X, this font is slightly larger than normal-control-font. Under Windows and X, it is the same size as normal-control-font.

---

normal-control-font : (is-a?/c font%)

This font is the default for most controls, except list-box% and group-box-panel% objects.

---

small-control-font : (is-a?/c font%)

This font is the default for group-box-panel% objects, and it is a suitable for controls in a floating window and other contexts that need smaller controls.

Under Windows, this font is the same size as normal-control-font, since the Windows control font is already relatively small. Under X and Mac OS X, this font is slightly smaller than normal-control-font.

---

tiny-control-font : (is-a?/c font%)

This font is for tiny controls, and it is smaller than small-control-font on all platforms.

---

view-control-font : (is-a?/c font%)

This font is the default for list-box% objects (but not list box labels, which use normal-control-font).

Under Mac OS X, this font is slightly smaller than normal-control-font, and slightly larger than small-control-font. Under Windows and X, it is the same size as normal-control-font.

2.5 Editor Classes

Editors:

```
editor<%>
 |- text%
 |- pasteboard%
```

Snips:

```
snip%                    readable-snip<%>
 |- string-snip%
 |    |- tab-snip%
 |- image-snip%
 |- editor-snip%
```

Displays, Administrators, and Mappings:

```
editor-canvas%

editor-admin%                        snip-admin%
 |- editor-snip-editor-admin<%>

editor-wordbreak-map%    keymap%
```

Styles:

```
style<%>          style-delta%       add-color<%>
style-list%                          mult-color<%>
```

Streams for Saving and Cut-and-Paste:

```
editor-data%
editor-data-class%                     snip-class%
editor-data-class-list<%>              snip-class-list<%>

editor-stream-in%                  editor-stream-out%
editor-stream-in-base%             editor-stream-out-base%
 |- editor-stream-in-bytes-base%    |- editor-stream-out-bytes-base%
```

Alphabetical:

2.5.1 add-color<%>

`add-color<%> : interface?`

An `add-color<%>` object is used to additively change the RGB values of a `color%` object. An `add-color<%>` object only exists within a `style-delta%` object.

See also get-foreground-add and get-background-add.

```
(send an-add-color get r g b) → void?
  r : (box/c (integer-in -1000 1000))
  g : (box/c (integer-in -1000 1000))
  b : (box/c (integer-in -1000 1000))
```

Gets all of the additive values.

The r box is filled with the additive value for the red component of the color. The g box is filled with the additive value for the green component of the color. The b box is filled with the additive value for the blue component of the color.

```
(send an-add-color get-b) → (integer-in -1000 1000)
```

Gets the additive value for the blue component of the color.

```
(send an-add-color get-g) → (integer-in -1000 1000)
```

Gets the additive value for the green component of the color.

```
(send an-add-color get-r) → (integer-in -1000 1000)
```

Gets the additive value for the red component of the color.

```
(send an-add-color set r g b) → void?
  r : (integer-in -1000 1000)
  g : (integer-in -1000 1000)
  b : (integer-in -1000 1000)
```

Sets all of the additive values.

```
(send an-add-color set-b v) → void?
  v : (integer-in -1000 1000)
```

Sets the additive value for the blue component of the color.

```
(send an-add-color set-g v) → void?
  v : (integer-in -1000 1000)
```

Sets the additive value for the green component of the color.

```
(send an-add-color set-r v) → void?
  v : (integer-in -1000 1000)
```

Sets the additive value for the red component of the color.

2.5.2 editor<%>

```
editor<%> : interface?
```

The editor<%> interface is implemented by text% and pasteboard%.

```
(send an-editor add-canvas canvas) → void?
  canvas : (is-a?/c editor-canvas%)
```

Adds a canvas to this editor's list of displaying canvases. (See get-canvases.)

Normally, this method is called only by set-editor in editor-canvas%.

```
(send an-editor add-undo undoer) → void?
  undoer : (-> any)
```

Adds an undoer procedure to the editor's undo stack. If an undo is currently being performed, the undoer is added to the editor's redo stack. The undoer is called by the system when it is undoing (or redoing) changes to an editor, and when this undoer is the first item on the undo (or redo) stack.

The system automatically installs undo records to undo built-in editor operations, such as inserts, deletes, and font changes. Install an undoer only when it is necessary to maintain state or handle operations that are not built-in. For example, in a program where the user can assign labels to snips in a pasteboard, the program should install an undoer to revert a label change. Thus, when a user changes a snip's label and then selects Undo (from a standard menu bar), the snip's label will revert as expected. In contrast, there is no need to install an undoer when the user moves a snip by dragging it, because the system installs an appropriate undoer automatically.

After an undoer returns, the undoer is popped off the editor's undo (or redo) stack; if the return value is true, then the next undoer is also executed as part of the same undo (or redo) step. The undoer should return true if the action being undone was originally performed as part of a begin-edit-sequence and end-edit-sequence sequence. The return value should also be true if the undone action was implicitly part of a sequence. To extend the previous example, if a label change is paired with a move to realign the snip, then the label-change undoer should be added to the editor *after* the call to move, and it should return #t

when it is called. As a result, the move will be undone immediately after the label change is undone. (If the opposite order is needed, use `begin-edit-sequence` and `end-edit-sequence` to create an explicit sequence.)

The system adds undoers to an editor (in response to other method calls) without calling this method.

```
(send an-editor adjust-cursor event)
 → (or/c (is-a?/c cursor%) #f)
  event : (is-a?/c mouse-event%)
```

Specification: Gets a cursor to be used in the editor's display. If the return value is `#f`, a default cursor is used.

See also `set-cursor`.

Default implementation: If an overriding cursor has been installed with `set-cursor`, then the installed cursor is returned.

Otherwise, if the event is a dragging event, a snip in the editor has the focus, and the snip's `adjust-cursor` method returns a cursor, that cursor is returned.

Otherwise, if the cursor is over a snip and the snip's `adjust-cursor` method returns a cursor, that cursor is returned.

Otherwise, if a cursor has been installed with `set-cursor`, then the installed cursor is returned.

Otherwise, if the cursor is over a clickback region in an editor, an arrow cursor is returned.

Finally, if none of the above cases apply, a default cursor is returned. For a text editor, the default cursor is an I-beam. For a pasteboard editor, the default cursor is an arrow.

```
(send an-editor after-edit-sequence) → void?
```

Refine this method with `augment`.

Specification: Called after a top-level edit sequence completes (involving unnested `begin-edit-sequence` and `end-edit-sequence`).

See also `on-edit-sequence`.

Default implementation: Does nothing.

```
(send an-editor after-load-file success?) → void?
  success? : any/c
```

Refine this method with `augment`.

Specification: Called just after the editor is loaded from a file.

The argument to the method originally specified whether the save was successful, but failures now trigger exceptions such that the method is not even called. Consequently, the argument is always `#t`.

See also `can-load-file?` and `on-load-file`.

Default implementation: Does nothing.

```
(send an-editor after-save-file success?) → void?
  success? : any/c
```

Refine this method with `augment`.

Specification: Called just after the editor is saved to a file.

The argument to the method originally specified whether the save was successful, but failures now trigger exceptions such that the method is not even called. Consequently, the argument is always `#t`.

See also `can-save-file?` and `on-save-file`.

Default implementation: Does nothing.

```
(send an-editor auto-wrap) → boolean?
(send an-editor auto-wrap auto-wrap?) → void?
  auto-wrap? : any/c
```

Enables or disables automatically calling `set-max-width` in response to `on-display-size`, or gets the state of auto-wrapping. For text editors, this has the effect of wrapping the editor's contents to fit in a canvas displaying the editor (the widest one if multiple canvases display the editor). For pasteboard editors, "auto-wrapping" merely truncates the area of the pasteboard to match its canvas display.

When the wrapping mode is changed, the `on-display-size` method is called immediately to update the editor's maximum width.

Auto-wrapping is initially disabled.

```
(send an-editor begin-edit-sequence [undoable?
                                     interrupt-streak?]) → void?
  undoable? : any/c = #t
```

```
  interrupt-streak? : any/c = #t
```

Specification: The begin-edit-sequence and end-edit-sequence methods are used to bracket a set of editor modifications so that the results are all displayed at once. The commands may be nested arbitrarily deep. Using these functions can greatly speed up displaying the changes.

When an editor contains other editors, using begin-edit-sequence and end-edit-sequence on the main editor brackets some changes to the sub-editors as well, but it is not as effective when a sub-editor changes as calling begin-edit-sequence and end-edit-sequence for the sub-editor.

See also refresh-delayed? and in-edit-sequence?, and see §1.3.9 "Editors and Threads" for information about edit sequences and refresh requests.

If the *undoable?* flag is #f, then the changes made in the sequence cannot be reversed through the undo method. This flag is only effective for the outermost begin-edit-sequence when nested sequences are used. Note that, for a text% object, the character-inserting version of insert interferes with sequence-based undo groupings.

If the *interrupt-streak?* flag is #f and the sequence is outermost, then special actions before and after the sequence count as consecutive actions. For example, kills just before and after the sequence are appended in the copy buffer.

Default implementation: Starts a sequence.

```
(send an-editor begin-write-header-footer-to-file f
                                                  name
                                                  buffer)
 → void?
  f : (is-a?/c editor-stream-out%)
  name : string?
  buffer : (box/c exact-integer?)
```

This method must be called before writing any special header data to a stream. See §1.3.2 "File Format" and write-headers-to-file for more information.

The *name* string must be a unique name that can be used by a header reader to recognize the data. This method will store a value in *buffer* that should be passed on to end-write-header-footer-to-file.

```
(send an-editor blink-caret) → void?
```

Specification: Tells the editor to blink the selection caret. This method is called periodically when the editor's display has the keyboard focus.

Default implementation: Propagates the request to any snip with the editor-local focus.

```
(send an-editor can-do-edit-operation? op
                                       [recursive?]) → boolean?
  op : (or/c 'undo 'redo 'clear 'cut 'copy 'paste
             'kill 'select-all 'insert-text-box
             'insert-pasteboard-box 'insert-image)
  recursive? : any/c = #t
```

Specification: Checks whether a generic edit command would succeed for the editor. This check is especially useful for enabling and disabling menus on demand. See `do-edit-operation` for information about the *op* and *recursive?* arguments.

Default implementation: Allows the operation depending on the selection, whether the editor is locked, etc.

```
(send an-editor can-load-file? filename
                               format)  → boolean?
  filename : path?
  format : (or/c 'guess 'same 'copy 'standard
                 'text 'text-force-cr)
```

Refine this method with `augment`.

Specification: Called just before the editor is loaded from a file. If the return value is `#f`, the file is not loaded. See also `on-load-file` and `after-load-file`.

The *filename* argument is the name the file will be loaded from. See `load-file` for information about *format*.

Note that the *filename* argument cannot be a string; it must be a path value.

Default implementation: Returns `#t`.

```
(send an-editor can-save-file? filename
                               format)  → boolean?
  filename : path?
  format : (or/c 'guess 'same 'copy 'standard
                 'text 'text-force-cr)
```

Refine this method with `augment`.

Specification: Called just before the editor is saved to a file. If the return value is `#f`, the file is not saved. See also `on-save-file` and `after-save-file`.

The *filename* argument is the name the file will be saved to. See load-file for information about *format*.

Note that the *filename* argument cannot be a string; it must be a path value.

Default implementation: Returns #t.

```
(send an-editor change-style delta) → void?
  delta : (or/c (is-a?/c style-delta%) #f)
(send an-editor change-style style) → void?
  style : (or/c (is-a?/c style<%>) #f)
```

Changes the style for items in the editor, either by applying a style delta or using a specific style.

To change a large collection of snips from one style to another style, consider providing a style<%> instance rather than a style-delta% instance. Otherwise, change-style must convert the style-delta% instance to the style<%> instance for every snip; this conversion consumes both time and (temporary) memory.

The style within an editor can be changed by the system (in response to other method calls), and such changes do not go through this method; use on-change-style in text% to monitor style changes.

```
(send an-editor clear) → void?
```

Deletes the currently selected items.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use on-delete in text% or on-delete in pasteboard% to monitor content deletions changes.

```
(send an-editor clear-undos) → void?
```

Destroys the undo history of the editor.

```
(send an-editor copy [extend? time]) → void?
  extend? : any/c = #f
  time : exact-integer? = 0
```

Copies items into the clipboard. If *extend?* is not #f, the old clipboard contents are appended.

The system may execute a copy (in response to other method calls) without calling this

method. To extend or re-implement copying, override the do-copy in text% or do-copy in pasteboard% method of an editor.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send an-editor copy-self)
 → (or/c (is-a?/c text%) (is-a?/c pasteboard%))
```

Creates a new editor with the same properties as this one. After an editor is created (either a text% or pasteboard% instance, as appropriate), the new editor is passed to copy-self-to.

```
(send an-editor copy-self-to dest) → void?
  dest : (or/c (is-a?/c text%) (is-a?/c pasteboard%))
```

Copies the properties of *an-editor* to *dest*.

Each snip in *an-editor* is copied and inserted into *dest*. In addition, *an-editor*’s filename, maximum undo history setting, keymap, interactive caret threshold, and overwrite-styles-on-load settings are installed into *dest*. Finally, *an-editor*’s style list is copied and the copy is installed as the style list for *dest*.

```
(send an-editor cut [extend? time]) → void?
  extend? : any/c = #f
  time : exact-integer? = 0
```

Copies and then deletes the currently selected items. If *extend?* is not #f, the old clipboard contents are appended.

The system may execute a cut (in response to other method calls) without calling this method. To extend or re-implement the copying portion of the cut, override the do-copy in text% or do-copy in pasteboard% method of an editor. To monitor deletions in an editor, override on-delete in text% or on-delete in pasteboard%.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send an-editor dc-location-to-editor-location x
                                               y) → real? real?
  x : real?
  y : real?
```

Converts the given coordinates from top-level display coordinates (usually canvas coordi-

nates) to editor location coordinates. The same calculation is performed by global-to-local.

The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”).

See also editor-location-to-dc-location.

(send an-editor default-style-name) → string?

Returns the name of a style to be used for newly inserted text, etc. The default is "Standard".

(send an-editor do-copy) → void?

See do-copy in text% or do-copy in pasteboard%.

```
(send an-editor do-edit-operation  op
                                   [recursive?
                                    time])      → void?
  op : (or/c 'undo 'redo 'clear 'cut 'copy 'paste
             'kill 'select-all 'insert-text-box
             'insert-pasteboard-box 'insert-image)
  recursive? : any/c = #t
  time : exact-integer? = 0
```

Performs a generic edit command. The *op* argument must be a valid edit command, one of:

- 'undo — undoes the last operation
- 'redo — undoes the last undo
- 'clear — deletes the current selection
- 'cut — cuts
- 'copy — copies
- 'paste — pastes
- 'kill — cuts to the end of the current line, or cuts a newline if there is only whitespace between the selection and end of line
- 'select-all — selects everything in the editor
- 'insert-text-box — inserts a text editor as an item in this editor; see also on-new-box .

- 'insert-pasteboard-box — inserts a pasteboard editor as an item in this editor; see also on-new-box .
- 'insert-image — gets a filename from the user and inserts the image as an item in this editor; see also on-new-image-snip .

If *recursive?* is not #f, then the command is passed on to any active snips of this editor (i.e., snips which own the caret).

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

(send *an-editor* do-paste) → void?

See do-paste in text% or do-paste in pasteboard%.

(send *an-editor* do-paste-x-selection) → void?

See do-paste-x-selection in text% or do-paste-x-selection in pasteboard%.

(send *an-editor* editor-location-to-dc-location *x*
y) → real? real?
x : real?
y : real?

Converts the given coordinates from editor location coordinates to top-level display coordinates (usually canvas coordinates). The same calculation is performed by local-to-global.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology").

See also dc-location-to-editor-location.

(send *an-editor* end-edit-sequence) → void?

See begin-edit-sequence.

(send *an-editor* end-write-header-footer-to-file *f*
buffer-value)
→ void?
f : (is-a?/c editor-stream-out%)
buffer-value : exact-integer?

This method must be called after writing any special header data to a stream. The *buffer-value* argument must be the value put in the buffer argument box by begin-write-header-footer-to-file.

See §1.3.2 "File Format" and write-headers-to-file for more information.

(send *an-editor* find-first-snip) → (or/c (is-a?/c snip%) #f)

Returns the first snip in the editor, or #f if the editor is empty. To get all of the snips in the editor, use the next in snip% on the resulting snip.

The first snip in a text editor is the one at position 0. The first snip in a pasteboard is the frontmost snip. (See §1.3.1 "Editor Structure and Terminology" for information about snip order in pasteboards.)

(send *an-editor* find-scroll-line *location*)
 → exact-nonnegative-integer?
  location : real?

Maps a vertical location within the editor to a vertical scroll position.

For text% objects: Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?). The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology").

(send *an-editor* get-active-canvas)
 → (or/c (is-a?/c editor-canvas%) #f)

If the editor is displayed in a canvas, this method returns the canvas that most recently had the keyboard focus (while the editor was displayed). If no such canvas exists, #f is returned.

(send *an-editor* get-admin) → (or/c (is-a?/c editor-admin%) #f)

Returns the editor-admin% object currently managing this editor or #f if the editor is not displayed.

(send *an-editor* get-canvas)
 → (or/c (is-a?/c editor-canvas%) #f)

If get-active-canvas returns a canvas, that canvas is also returned by this method. Otherwise, if get-canvases returns a non-empty list, the first canvas in the list is returned, otherwise #f is returned.

```
(send an-editor get-canvases)
 → (listof (is-a?/c editor-canvas%))
```

Returns a list of canvases displaying the editor. An editor may be displayed in multiple canvases and no other kind of display, or one instance of another kind of display and no canvases. If the editor is not displayed or the editor's current display is not a canvas, `null` is returned.

```
(send an-editor get-dc) → (or/c (is-a?/c dc<%>) #f)
```

Typically used (indirectly) by snip objects belonging to the editor. Returns a destination drawing context which is suitable for determining display sizing information, or `#f` if the editor is not displayed.

```
(send an-editor get-descent) → (and/c real? (not/c negative?))
```

Returns the font descent for the editor. This method is primarily used when an editor is an item within another editor.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). For `text%` objects, calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see `refresh-delayed?`).

```
(send an-editor get-extent w h) → void?
  w : (or/c (box/c (and/c real? (not/c negative?))) #f)
  h : (or/c (box/c (and/c real? (not/c negative?))) #f)
```

Gets the current extent of the editor's graphical representation. The *w* box is filled with the editor's width, unless *w* is `#f`. The *h* box is filled with the editor's height, unless *h* is `#f`.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). For `text%` objects, calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see `refresh-delayed?`).

```
(send an-editor get-file directory) → (or/c path-string? #f)
  directory : (or/c path? #f)
```

Specification: Called when the user must be queried for a filename to load an editor. A starting-directory path is passed in, but is may be `#f` to indicate that any directory is fine.

Note that the *directory* argument cannot be a string; it must be a path value or `#f`.

Default implementation: Calls the global `get-file` procedure.

If the editor is displayed in a single canvas, then the canvas's top-level frame is used as the parent for the file dialog. Otherwise, the file dialog will have no parent.

```
(send an-editor get-filename [temp]) → (or/c path-string? #f)
  temp : (box/c (or/c any/c #f)) = (box #f)
```

Returns the path name of the last file saved from or loaded into this editor, `#f` if the editor has no filename.

The *temp* box is filled with `#t` if the filename is temporary or `#f` otherwise.

```
(send an-editor get-flattened-text) → string?
```

Returns the contents of the editor in text form. See §1.3.4 "Flattened Text" for a discussion of flattened vs. non-flattened text.

```
(send an-editor get-focus-snip) → (or/c (is-a?/c snip%) #f)
```

Returns the snip within the editor that gets the keyboard focus when the editor has the focus, or `#f` if the editor does not delegate the focus.

The returned snip might be an `editor-snip%` object. In that case, the embedded editor might delegate the focus to one of its own snips. However, the `get-focus-snip` method returns only the `editor-snip%` object, because it is the focus-owning snip within the immediate editor.

See also `set-caret-owner`.

```
(send an-editor get-inactive-caret-threshold)
 → (or/c 'no-caret 'show-inactive-caret 'show-caret)
```

Returns the threshold for painting an inactive selection. This threshold is compared with the `draw-caret` argument to `refresh` and if the argument is as least as large as the threshold (but larger than `'show-caret`), the selection is drawn as inactive.

See also `set-inactive-caret-threshold` and §1.3.5 "Caret Ownership".

```
(send an-editor get-keymap) → (or/c (is-a?/c keymap%) #f)
```

Returns the main keymap currently used by the editor.

```
(send an-editor get-load-overwrites-styles) → boolean?
```

Reports whether named styles in the current style list are replaced by `load-file` when the loaded file contains style specifications.

See also `set-load-overwrites-styles`.

```
(send an-editor get-max-height)
 → (or/c (and/c real? (not/c negative?)) 'none)
```

Gets the maximum display height for the contents of the editor; zero or `'none` indicates that there is no maximum.

```
(send an-editor get-max-undo-history)
 → (or/c (integer-in 0 100000) 'forever)
```

Returns the maximum number of undoables that will be remembered by the editor. Note that undoables are counted by insertion, deletion, etc. events, not by the number of times that `undo` can be called; a single `undo` call often reverses multiple events at a time (such as when the user types a stream of characters at once).

```
(send an-editor get-max-view-size) → real? real?
```

Returns the maximum visible area into which the editor is currently being displayed, according to the editor's administrators. If the editor has only one display, the result is the same as for `get-view-size`. Otherwise, the maximum width and height of all the editor's displaying canvases is returned.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology").

If the display is an editor canvas, see also `reflow-container`.

```
(send an-editor get-max-width)
 → (or/c (and/c real? (not/c negative?)) 'none)
```

Gets the maximum display width for the contents of the editor; zero or `'none` indicates that there is no maximum. In a text editor, zero of `'none` disables automatic line breaking.

```
(send an-editor get-min-height)
 → (or/c (and/c real? (not/c negative?)) 'none)
```

Gets the minimum display height for the contents of the editor; zero or `'none` indicates that

there is no minimum.

```
(send an-editor get-min-width)
 → (or/c (and/c real? (not/c negative?)) 'none)
```

Gets the minimum display width for the contents of the editor; zero or `'none` indicates that there is no minimum.

```
(send an-editor get-paste-text-only) → boolean?
```

If the result is `#t`, then the editor accepts only plain-text data from the clipboard. If the result is `#f`, the editor accepts both text and snip data from the clipboard.

```
(send an-editor get-snip-data thesnip)
 → (or/c (is-a?/c editor-data%) #f)
  thesnip : (is-a?/c snip%)
```

Specification: Gets extra data associated with a snip (e.g., location information in a pasteboard) or returns `#f` is there is no information. See §1.3.2.1.2 "Editor Data" for more information.

Default implementation: Returns `#f`.

```
(send an-editor get-snip-location  thesnip
                                   [x
                                    y
                                    bottom-right?]) → boolean?
  thesnip : (is-a?/c snip%)
  x : (or/c (box/c real?) #f) = #f
  y : (or/c (box/c real?) #f) = #f
  bottom-right? : any/c = #f
```

Gets the location of the given snip. If the snip is found in the editor, `#t` is returned; otherwise, `#f` is returned.

The *x* box is filled with the x-coordinate of the snip's location, unless *x* is `#f`. The *y* box is filled with the y-coordinate of the snip's location, unless *y* is `#f`.

If *bottom-right?* is not `#f`, the values in the *x* and *y* boxes are for the snip's bottom right corner instead of its top-left corner.

Obtaining the location if the bottom-right corner may trigger delayed size calculations (including snips other than the one whose location was requested).

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). As a special case, however, a pasteboard% object always reports valid answers when *bottom-right?* is #f. For text% objects, calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?).

(send *an-editor* get-space) → (and/c real? (not/c negative?))

Returns the maximum font space for the editor. This method is primarily used when an editor is an item within another editor.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). For text% objects, calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?).

(send *an-editor* get-style-list) → (is-a?/c style-list%)

Returns the style list currently in use by the editor.

(send *an-editor* get-view-size *w h*) → void?
  w : (or/c (box/c (and/c real? (not/c negative?))) #f)
  h : (or/c (box/c (and/c real? (not/c negative?))) #f)

Returns the visible area into which the editor is currently being displayed (according to the editor's administrator). See also get-view .

The *w* box is filled with the visible area width, unless *w* is #f. The *h* box is filled with the visible area height, unless *h* is #f.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology").

If the display is an editor canvas, see also reflow-container.

(send *an-editor* global-to-local *x y*) → void?
  x : (or/c (box/c real?) #f)
  y : (or/c (box/c real?) #f)

Converts the given coordinates from top-level display coordinates (usually canvas coordinates) to editor location coordinates. The same calculation is performed by dc-location-to-editor-location.

The *x* box is filled with the translated x-coordinate of the value initially in *x*, unless *x* is #f.

The *y* box is filled with the translated x-coordinate of the value initially in *y*, unless *y* is #f.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology").

See also local-to-global.

(send *an-editor* in-edit-sequence?) → boolean?

Returns #t if updating on this editor is currently delayed because begin-edit-sequence has been called for this editor.

See also refresh-delayed?.

(send *an-editor* insert *snip*) → void?
  snip : (is-a?/c snip%)

Inserts data into the editor. A snip cannot be inserted into multiple editors or multiple times within a single editor.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use on-insert in text% or on-insert in pasteboard% to monitor content additions changes.

(send *an-editor* insert-box [*type*]) → void?
  type : (or/c 'text 'pasteboard) = 'text

Inserts a box (a sub-editor) into the editor by calling on-new-box, then passing along *type* and inserts the resulting snip into the editor.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use on-insert in text% or on-insert in pasteboard% to monitor content additions changes.

(send *an-editor* insert-file *filename*
                             [*format*
                              show-errors?]) → boolean?
  filename : path-string?
  format : (or/c 'guess 'same 'copy 'standard = 'guess
                   'text 'text-force-cr)
  show-errors? : any/c = #t

Inserts the content of a file or port into the editor (at the current selection position in text% editors). The result is #t; if an error occurs, an exception is raised.

For information on *format*, see load-file. The *show-errors?* argument is no longer used.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use on-insert in text% or on-insert in pasteboard% to monitor content additions changes.

```
(send an-editor insert-image [filename
                              type
                              relative-path?
                              inline?])      → void?
  filename : (or/c path-string? #f) = #f
  type : (or/c 'unknown 'gif 'jpeg 'xbm 'xpm 'bmp 'pict)
       = 'unknown
  relative-path? : any/c = #f
  inline? : any/c = #t
```

Inserts an image into the editor.

If *filename* is #f, then the user is queried for a filename. The kind must one of the symbols that can be passed to load-file.

After the filename has been determined, an image is created by calling on-new-image-snip. See also image-snip%.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use on-insert in text% or on-insert in pasteboard% to monitor content additions changes.

```
(send an-editor insert-port port
                            [format
                             replace-styles?])
 → (or/c 'standard 'text 'text-force-cr)
  port : input-port?
  format : (or/c 'guess 'same 'copy 'standard = 'guess
                 'text 'text-force-cr)
  replace-styles? : any/c = #t
```

Inserts the content of a port into the editor (at the current selection position in text% editors) without wrapping the insert operations as an edit sequence. The result is the actual format of the loaded content (which is different from the given format type if the given format is 'guess, 'same, or 'copy).

The *port* must support position setting with file-position.

For information on *format*, see load-file.

if *replace-styles?* is true, then styles in the current style list are replaced by style specifications in *port*'s stream.

See also insert-file.

```
(send an-editor invalidate-bitmap-cache [x
                                         y
                                         width
                                         height]) → void?
  x : real? = 0.0
  y : real? = 0.0
  width : (or/c (and/c real? (not/c negative?)) 'end) = 'end
  height : (or/c (and/c real? (not/c negative?)) 'end) = 'end
```

When on-paint is overridden, call this method when the state of on-paint's drawing changes.

The *x*, *y*, *width*, and *height* arguments specify the area that needs repainting in editor coordinates. If *width*/*height* is 'end, then the total height/width of the editor (as reported by get-extent) is used. Note that the editor's size can be smaller than the visible region of its display.

The default implementation triggers a redraw of the editor, either immediately or at the end of the current edit sequence (if any) started by begin-edit-sequence.

See also size-cache-invalid.

```
(send an-editor is-locked?) → boolean?
```

Returns #t if the editor is currently locked, #f otherwise. See lock for more information.

```
(send an-editor is-modified?) → boolean?
```

Returns #t if the editor has been modified since the last save or load (or the last call to set-modified with #f), #f otherwise.

```
(send an-editor is-printing?) → boolean?
```

Returns #t if the editor is currently being printed through the print method, #f otherwise.

```
(send an-editor kill [time]) → void?
  time : exact-integer? = 0
```

In a text editor, cuts to the end of the current line, or cuts a newline if there is only whitespace between the selection and end of line. Multiple consecutive kills are appended. In a pasteboard editor, cuts the current selection.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an `exn:fail:contract` exception is raised.

See also `cut`.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use `on-delete` in `text%` or `on-delete` in `pasteboard%` to monitor content deletions changes.

```
(send an-editor load-file [filename
                           format
                           show-errors?]) → boolean?
  filename : (or/c path-string? #f) = #f
  format : (or/c 'guess 'same 'copy 'standard = 'guess
                 'text 'text-force-cr)
  show-errors? : any/c = #t
```

Loads a file into the editor and returns `#t`. If an error occurs, an exception is raised.

If *filename* is `#f`, then the internally stored filename will be used; if *filename* is `""` or if the internal name is unset or temporary, then the user will be prompted for a name.

The possible values for *format* are listed below. A single set of *format* values are used for loading and saving files:

- `'guess` — guess the format based on extension and/or contents; when saving a file, this is the same as `'standard`
- `'same` — read in whatever format was last loaded or saved
- `'standard` — read/write a standard file (binary format)
- `'copy` — write using whatever format was last loaded or saved, but do not change the modification flag or remember *filename* (saving only)
- `'text` — read/write a text file (`text%` only); file writing uses the platform’s text-mode conventions (e.g., newlines as return–linefeed combinations under Windows) when not specifically disabled via `use-file-text-mode`
- `'text-force-cr` — read/write a text file (`text%` only); when writing, change automatic newlines (from word-wrapping) into real carriage returns

In a `text%` instance, the format returned from `get-file-format` is always one of `'standard`, `'text`, or `'text-force-cr`.

The *show-errors?* argument is no longer used.

The filename used to load the file can be retrieved with get-filename. For a text% instance, the format can be retrieved with get-file-format. However, if an error occurs while loading the file, the filename is set to #f.

See also on-load-file, after-load-file, can-load-file?, and set-load-overwrites-styles.

```
(send an-editor local-to-global x y) → void?
  x : (or/c (box/c real?) #f)
  y : (or/c (box/c real?) #f)
```

Converts the given coordinates from editor location coordinates to top-level display coordinates (usually canvas coordinates). The same calculation is performed by editor-location-to-dc-location.

The *x* box is filled with the translated x-coordinate of the value initially in *x*, unless *x* is #f. The *y* box is filled with the translated x-coordinate of the value initially in *y*, unless *y* is #f.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology").

See also global-to-local.

```
(send an-editor locations-computed?) → boolean?
```

Returns #t if all location information has been computed after recent changes to the editor's content or to its snips, #f otherwise.

Location information is often computed on demand, and begin-edit-sequence tends to delay the computation.

When the editor is locked for reflowing, location information cannot be recomputed. See also §1.3.8 "Internal Editor Locks".

```
(send an-editor lock lock?) → void?
  lock? : any/c
```

Locks or unlocks the editor for modifications. If an editor is locked, *all* modifications are blocked, not just user modifications.

See also is-locked?.

This method does not affect internal locks, as discussed in §1.3.8 "Internal Editor Locks".

(send *an-editor* locked-for-flow?) → boolean?

Reports whether the editor is internally locked for flowing. See §1.3.8 “Internal Editor Locks” for more information.

(send *an-editor* locked-for-read?) → boolean?

Reports whether the editor is internally locked for reading. See §1.3.8 “Internal Editor Locks” for more information.

(send *an-editor* locked-for-write?) → boolean?

Reports whether the editor is internally locked for writing. See §1.3.8 “Internal Editor Locks” for more information.

```
(send an-editor needs-update snip
                             localx
                             localy
                             w
                             h)      → void?
  snip : (is-a?/c snip%)
  localx : real?
  localy : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Typically called (indirectly) by a snip within the editor to force the editor to be redrawn.

The *localx*, *localy*, width, and height arguments specify the area that needs repainting in the coordinate system of *snip*.

For text% objects, calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?).

(send *an-editor* num-scroll-lines) → exact-nonnegative-integer?

Reports the number of scroll positions available within the editor.

For text% objects: Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for line-start-position (which handles specially the case of no display when the editor has a

maximum width).

`(send` *an-editor* `on-change)` → `void?`

Refine this method with `augment`.

Specification: Called whenever any change is made to the editor that affects the way the editor is drawn or the values reported for the location/size of some snip in the editor. The `on-change` method is called just before the editor calls its administrator's `needs-update` method to refresh the editor's display, and it is also called just before and after printing an editor.

The editor is locked for writing and reflowing during the call to `on-change`.

Default implementation: Does nothing.

`(send` *an-editor* `on-char` *event*`)` → `void?`
  event : `(is-a?/c key-event%)`

Specification: Handles keyboard input to the editor.

Consider overriding `on-local-char` or `on-default-char` instead of this method.

Default implementation: Either passes this event on to a caret-owning snip or calls `on-local-char`. In the latter case, `text%` first calls `hide-cursor-until-moved`.

`(send` *an-editor* `on-default-char` *event*`)` → `void?`
  event : `(is-a?/c key-event%)`

Specification: Called by `on-local-char` when the event is *not* handled by a caret-owning snip or by the keymap.

Default implementation: Does nothing.

`(send` *an-editor* `on-default-event` *event*`)` → `void?`
  event : `(is-a?/c mouse-event%)`

Specification: Called by `on-local-event` when the event is *not* handled by a caret-owning snip or by the keymap.

Default implementation: Does nothing. See also `on-default-event` in `text%` and `on-default-event` in `pasteboard%`.

`(send` *an-editor* `on-display-size)` → `void?`

Refine this method with `augment`.

Specification: This method is called by the editor's display whenever the display's size (as reported by `get-view-size`) changes, but it is called indirectly through `on-display-size-when-ready`.

Default implementation: If automatic wrapping is enabled (see `auto-wrap`) then `set-max-width` is called with the maximum width of all of the editor's canvases (according to the administrators; `call-as-primary-owner` in `editor-canvas%` is used with each canvas to set the administrator and get the view size). If the editor is displayed but not in a canvas, the unique width is obtained from the editor's administrator (there is only one). If the editor is not displayed, the editor's maximum width is not changed.

`(send` *an-editor* `on-display-size-when-ready)` → `void?`

Calls `on-display-size` unless the editor is currently in an edit sequence or currently being refreshed. In the latter cases, the call to `on-display-size` is delegated to another thread; see §1.3.9 "Editors and Threads" for more information.

`(send` *an-editor* `on-edit-sequence)` → `void?`

Refine this method with `augment`.

Specification: Called just after a top-level (i.e., unnested) edit sequence starts.

During an edit sequence, all callbacks methods are invoked normally, but it may be appropriate for these callbacks to delay computation during an edit sequence. The callbacks must manage this delay manually. Thus, when overriding other callback methods, such as `on-insert` in `text%`, `on-insert` in `pasteboard%`, `after-insert` in `text%`, or `after-insert` in `pasteboard%`, consider overriding `on-edit-sequence` and `after-edit-sequence` as well.

"Top-level edit sequence" refers to an outermost pair of `begin-edit-sequence` and `end-edit-sequence` calls. The embedding of an editor within another editor does not affect the timing of calls to `on-edit-sequence`, even if the embedding editor is in an edit sequence.

Pairings of `on-edit-sequence` and `after-edit-sequence` can be nested if an `after-edit-sequence` starts a new edit sequence, since `after-edit-sequence` is called after an edit sequence ends. However, `on-edit-sequence` can never start a new top-level edit sequence (except through an unpaired `end-edit-sequence`), because it is called after a top-level edit sequence starts.

Default implementation: Does nothing.

`(send` *an-editor* `on-event` *event*`)` → `void?`

```
  event : (is-a?/c mouse-event%)
```

Specification: Handles mouse input to the editor. The event's x and y coordinates are in the display's co-ordinate system; use the administrator's `get-dc` method to obtain translation arguments (or use `dc-location-to-editor-location`).

Consider overriding `on-local-event` or `on-default-event` instead of this method.

Default implementation: Either passes this event on to a caret-owning snip, selects a new caret-owning snip (`text%` only) and passes the event on to the selected snip, or calls `on-local-event`. A new caret-owning snip is selected in a `text%` object when the click is on an event-handling snip, and not too close to the space between snips (see `get-between-threshold`).

```
(send an-editor on-focus on?) → void?
  on? : any/c
```

Called when the keyboard focus changes into or out of this editor (and not to/from a snip within the editor) with `#t` if the focus is being turned on, `#f` otherwise.

```
(send an-editor on-load-file filename
                             format)  → void?
  filename : path?
  format : (or/c 'guess 'same 'copy 'standard
                 'text 'text-force-cr)
```

Refine this method with `augment`.

Specification: Called just before the editor is loaded from a file, after calling `can-load-file?` to verify that the load is allowed. See also `after-load-file`.

The *`filename`* argument is the name the file will be loaded from. See `load-file` for information about *`format`*.

Note that the *`filename`* argument cannot be a string; it must be a path value.

Default implementation: Does nothing.

```
(send an-editor on-local-char event) → void?
  event : (is-a?/c key-event%)
```

Specification: Called by `on-char` when the event is *not* handled by a caret-owning snip.

Consider overriding `on-default-char` instead of this method.

Default implementation: Either lets the keymap handle the event or calls on-default-char.

```
(send an-editor on-local-event event) → void?
  event : (is-a?/c mouse-event%)
```

Specification: Called by on-event when the event is *not* handled by a caret-owning snip.

Consider overriding on-default-event instead of this method.

Default implementation: Either lets the keymap handle the event or calls on-default-event.

```
(send an-editor on-new-box type) → (is-a?/c snip%)
  type : (or/c 'text 'pasteboard)
```

Specification: Creates and returns a new snip for an embedded editor. This method is called by insert-box.

Default implementation: Creates a editor-snip% with either a sub-editor from text% or sub-pasteboard from pasteboard%, depending on whether *type* is 'text or 'pasteboard. The keymap (see keymap%) and style list (see style-list%) for of the new sub-editor are set to the keymap and style list of this editor.

```
(send an-editor on-new-image-snip filename
                                  kind
                                  relative-path?
                                  inline?)
 → (is-a?/c image-snip%)
  filename : path?
  kind : (or/c 'unknown 'gif 'jpeg 'xbm 'xpm 'bmp 'pict)
  relative-path? : any/c
  inline? : any/c
```

Specification: Creates and returns a new instance of image-snip% for insert-image.

Note that the *filename* argument cannot be a string; it must be a path value.

Default implementation: Returns (make-object image-snip% *filename kind relative-path? inline?*).

```
(send an-editor on-paint before?
                         dc
                         left
                         top
                         right
                         bottom
                         dx
                         dy
                         draw-caret) → void?
  before? : any/c
  dc : (is-a?/c dc<%>)
  left : real?
  top : real?
  right : real?
  bottom : real?
  dx : real?
  dy : real?
  draw-caret : (or/c 'no-caret 'show-inactive-caret 'show-caret)
```

Specification: Provides a way to add arbitrary graphics to an editor's display. This method is called just before and just after every painting of the editor.

The *before?* argument is `#t` when the method is called just before a painting the contents of the editor or `#f` when it is called after painting. The *left*, *top*, *right*, and *bottom* arguments specify which region of the editor is being repainted, in editor coordinates. To get the coordinates for *dc*, offset editor coordinates by adding (*dx*, *dy*). See §1.3.5 "Caret Ownership" for information about *draw-caret*.

The `on-paint` method, together with the snips' `draw` methods, must be able to draw the entire state of an editor. Never paint directly into an editor's display canvas except from within `on-paint` or `draw`. Instead, put all extra drawing code within `on-paint` and call `invalidate-bitmap-cache` when part of the display needs to be repainted.

If an `on-paint` method uses cached location information, then the cached information should be recomputed in response to a call of `invalidate-bitmap-cache`.

The `on-paint` method must not make any assumptions about the state of the drawing context (e.g., the current pen), except that the clipping region is already set to something appropriate. Before `on-paint` returns, it must restore any drawing context settings that it changes.

The editor is internally locked for writing and reflowing during a call to this method (see also §1.3.8 "Internal Editor Locks").

See also `invalidate-bitmap-cache`.

Default implementation: Does nothing.

```
(send an-editor on-save-file filename
                             format)  → void?
  filename : path?
  format : (or/c 'guess 'same 'copy 'standard
                 'text 'text-force-cr)
```

Refine this method with `augment`.

Specification: Called just before the editor is saved to a file, after calling `can-save-file?` to verify that the save is allowed. See also `after-save-file`.

The *`filename`* argument is the name the file will be saved to. See `load-file` for information about *`format`*.

Note that the *`filename`* argument cannot be a string; it must be a path value.

Default implementation: Does nothing.

```
(send an-editor on-snip-modified snip
                                 modified?) → void?
  snip : (is-a?/c snip%)
  modified? : any/c
```

Refine this method with `augment`.

Specification: This method is called whenever a snip within the editor reports that it has been modified (by calling its adminstrator's `modified` method). The method arguments are the snip that reported a modification-state change, and the snip's new modification state.

See also `set-modified`.

Default implementation: If *`modified?`* is true and the editor was not already modified (i.e., its `is-modified?` method reports `#f`), then the `set-modified` method is called with `#t`. If the editor was already modified, then the internal modify-counter is incremented.

If *`modified?`* is `#f`, and if the modify-counter is `1`, then the `set-modified` method is called with `#f` (on the assumption that the modify-counter was set to `1` by an earlier call to this method for the same snip).

```
(send an-editor own-caret own?) → void?
  own? : any/c
```

Specification: Tells the editor to display or not display the caret or selection.

The focus state of an editor can be changed by by the system, and such changes do not go through this method; use on-focus to monitor focus changes.

Default implementation: Propagates the flag to any snip with the editor-local focus. If no sub-editors are active, the editor assumes the caret ownership.

```
(send an-editor paste [time]) → void?
  time : exact-integer? = 0
```

Pastes the current contents of the clipboard into the editor.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

The system may execute a paste (in response to other method calls) without calling this method. To extend or re-implement copying, override the do-paste in text% or do-paste in pasteboard% method.

See also get-paste-text-only.

```
(send an-editor paste-x-selection [time]) → void?
  time : exact-integer? = 0
```

Like paste, but under X, uses the X selection instead of the X clipboard.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

To extend or re-implement copying, override the do-paste-x-selection in text% or do-paste-x-selection in pasteboard% method.

```
(send an-editor print [interactive?
                      fit-on-page?
                      output-mode
                      parent
                      force-ps-page-bbox?
                      as-eps?])            → void?
  interactive? : any/c = #t
  fit-on-page? : any/c = #t
  output-mode : (or/c 'standard 'postscript) = 'standard
  parent : (or/c (or/c (is-a?/c frame%) (is-a?/c dialog%)) #f)
         = #f
  force-ps-page-bbox? : any/c = #t
  as-eps? : any/c = #f
```

Prints the editor.

If *interactive?* is true and a PostScript file is created, the is given a dialog for adjusting printing parameters; see also get-ps-setup-from-user. Otherwise, if a PostScript file is created, the settings returned by current-ps-setup are used. (The user may still get a dialog to select an output file name; see post-script-dc% for more details.)

If *fit-on-page?* is a true value, then during printing for a text% editor, the editor's maximum width is set to the width of the page (less margins) and the autowrapping bitmap is removed.

The *output-mode* setting is used for Windows and Mac OS X. It determines whether the output is generated directly as a PostScript file (using Racket's built-in PostScript system) or generated using the platform-specific standard printing mechanism. The possible values are

- 'standard — print using the platform-standard mechanism (via a printer-dc%) under Windows and Mac OS X, PostScript for Unix (via a post-script-dc%)
- 'postscript — print to a PostScript file (via a post-script-dc%)

If *parent* is not #f, it is used as the parent window for configuration dialogs (for either PostScript or platform-standard printing). If *parent* is #f and if the editor is displayed in a single canvas, then the canvas's top-level frame is used as the parent for configuration dialogs. Otherwise, configuration dialogs will have no parent.

The *force-ps-page-bbox?* argument is used for PostScript printing, and is used as the third initialization argument when creating the post-script-dc% instance. Unless it is #f, the bounding-box of the resulting PostScript file is set to the current paper size.

The *as-eps?* argument is used for PostScript printing, and is used as the fourth initialization argument when creating the post-script-dc% instance. Unless it is #f, the resulting PostScript file is identified as Encapsulated PostScript (EPS).

The printing margins are determined by get-editor-margin in the current ps-setup% object (as determined by current-ps-setup).

```
(send an-editor print-to-dc dc [page-number]) → void?
  dc : (is-a?/c dc<%>)
  page-number : exact-integer? = -1
```

Prints the editor into the given drawing context. See also print.

If *page-number* is a non-negative integer, then just the indicated page is printed, where pages are numbered from 1. (So, supplying 0 as *page-number* produces no output.) When *page-number* is negative, the start-page and dc<%> end-page methods of *dc* are called for each page.

```
(send an-editor put-file directory
                         default-name) → (or/c path-string? #f)
  directory : (or/c path? #f)
  default-name : (or/c path? #f)
```

Specification: Called when the user must be queried for a filename to save an editor. Starting-directory and default-name paths are passed in, but either may be `#f` to indicate that any directory is fine or there is no default name.

Note that the *directory* and `filename` arguments cannot be strings; each must be a path value.

Default implementation: Calls the global `put-file` procedure.

If the editor is displayed in a single canvas, then the canvas's top-level frame is used as the parent for the file dialog. Otherwise, the file dialog will have no parent.

```
(send an-editor read-footer-from-file stream
                                      name)  → boolean?
  stream : (is-a?/c editor-stream-in%)
  name : string?
```

See `read-header-from-file`.

```
(send an-editor read-from-file  stream
                                [overwrite-styles?]) → boolean?
  stream : (is-a?/c editor-stream-in%)
  overwrite-styles? : any/c = #f
```

Reads new contents for the editor from a stream. The return value is `#t` if there are no errors, `#f` otherwise. See also §1.3.2 "File Format".

The stream provides either new mappings for names in the editor's style list, or it indicates that the editor should share a previously-read style list (depending on how style lists were shared when the editor was written to the stream; see also `write-to-file`).

- In the former case, if the *overwrite-styles?* argument is `#f`, then each style name in the loaded file that is already in the current style list keeps its current style. Otherwise, existing named styles are overwritten with specifications from the loaded file.
- In the latter case, the editor's style list will be changed to the previously-read list.

```
(send an-editor read-header-from-file stream
                                      name)  → boolean?
  stream : (is-a?/c editor-stream-in%)
  name : string?
```

Called to handle a named header that is found when reading editor data from a stream. The return value is `#t` if there are no errors, `#f` otherwise.

Override this method only to embellish the file format with new header information. Always call the inherited method if the derived reader does not recognize the header.

See also §1.3.2 “File Format”.

```
(send an-editor redo) → void?
```

Undoes the last undo, if no other changes have been made since. See `undo` for information about Emacs-style undo. If the editor is currently performing an undo or redo, the method call is ignored.

The system may perform a redo without calling this method in response to other method calls. Use methods such as `on-change` to monitor editor content changes.

See also `add-undo`.

```
(send an-editor refresh x
                        y
                        width
                        height
                        draw-caret
                        background) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  draw-caret : (or/c 'no-caret 'show-inactive-caret 'show-caret)
  background : (or/c (is-a?/c color%) #f)
```

Repaints a region of the editor, generally called by an editor administrator. The *x*, *y*, *width*, and *height* arguments specify the area that needs repainting in editor coordinates. The `get-dc` method of the editor's administrator (as returned by `get-admin`) supplies the target `dc<%>` object and offset for drawing.

See §1.3.5 “Caret Ownership” for information about *draw-caret*.

The *background* color corresponds to the background of the display; if it is `#f`, then the dis-

play is transparent. An editor should use the given background color as its own background (or not paint the background of *background* is #f).

See §1.3.9 "Editors and Threads" for information about edit sequences and refresh requests.

```
(send an-editor refresh-delayed?) → boolean?
```

Returns #t if updating on this editor is currently delayed. Updating may be delayed because begin-edit-sequence has been called for this editor, or because the editor has no administrator, or because the editor's administrator returns #t from its refresh-delayed? method. (The administrator might return #t because an enclosing editor's refresh is delayed.)

See also in-edit-sequence?.

```
(send an-editor release-snip snip) → boolean?
  snip : (is-a?/c snip%)
```

Requests that the specified snip be deleted and released from the editor. If this editor is not the snip's owner or if the snip cannot be released, then #f is returned. Otherwise, #t is returned and the snip is no longer owned.

See also release-snip in snip-admin% .

```
(send an-editor remove-canvas canvas) → void?
  canvas : (is-a?/c editor-canvas%)
```

Removes a canvas from this editor's list of displaying canvases. (See get-canvases.)

Normally, this method is called only by set-editor in editor-canvas%.

```
(send an-editor resized snip redraw-now?) → void?
  snip : (is-a?/c snip%)
  redraw-now? : any/c
```

Called (indirectly) by snips within the editor: it forces a recalculation of the display information in which the specified snip has changed its size.

If *redraw-now?* is #f, the editor will require another message to repaint itself. (See also needs-update.)

```
(send an-editor save-file [filename
                           format
                           show-errors?]) → boolean?
```

```
  filename : (or/c path-string? #f) = #f
  format : (or/c 'guess 'same 'copy 'standard = 'same
                 'text 'text-force-cr)
  show-errors? : any/c = #t
```

Saves the editor into a file and returns `#t`. If an error occurs, an exception is raised.

If *filename* is `#f`, then the internally stored filename will be used; if *filename* is `""` or if the internal name is unset or temporary, then the user will be prompted for a name. The possible values for *format* are described at `load-file`.

The filename and format used to save the file can be retrieved with `get-filename`. In a `text%` instance, the format can be retrieved with `get-file-format`.

See also `on-save-file`, `after-save-file`, and `can-save-file?`.

Under Mac OS X, the file's type signature is set to `"TEXT"` for a text-format file or `"WXME"` for a standard-format (binary) file.

The *show-errors?* argument is no longer used.

```
(send an-editor save-port  port
                           [format
                            show-errors?]) → boolean?
  port : output-port?
  format : (or/c 'guess 'same 'copy 'standard = 'same
                 'text 'text-force-cr)
  show-errors? : any/c = #t
```

Saves the editor into a port and returns `#t`. If an error occurs, an exception is raised.

The possible values for *format* are described at `load-file`.

The *show-errors?* argument is no longer used.

```
(send an-editor scroll-editor-to localx
                                 localy
                                 width
                                 height
                                 refresh?
                                 bias)    → boolean?
  localx : real?
  localy : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  refresh? : any/c
```

```
  bias : (or/c 'start 'end 'none)
```

Causes the editor to be scrolled so that a given location is visible. If the editor is scrolled, `#t` is returned, otherwise `#f` is returned.

This method is normally called indirectly by `scroll-to` or `scroll-to-position` in `text%` to implement scrolling.

The default implementation forwards the request to the `scroll-to` method of the current administrator, if any (see `get-admin`). If the editor has no administrator, `#f` is returned.

```
(send an-editor scroll-line-location pos)
 → (and/c real? (not/c negative?))
  pos : exact-nonnegative-integer?
```

Maps a vertical scroll position to a vertical location within the editor.

For `text%` objects: Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see `refresh-delayed?`). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for `line-start-position` (which handles specially the case of no display when the editor has a maximum width).

```
(send an-editor scroll-to  snip
                           localx
                           localy
                           width
                           height
                           refresh?
                          [bias])    → boolean?
  snip : (is-a?/c snip%)
  localx : real?
  localy : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  refresh? : any/c
  bias : (or/c 'start 'end 'none) = 'none
```

Called (indirectly) by snips within the editor: it causes the editor to be scrolled so that a given location range within a given snip is visible. If the editor is scrolled immediately, `#t` is returned, otherwise `#f` is returned.

If refreshing is delayed (see `refresh-delayed?`), then the scroll request is saved until the delay has ended. The scroll is performed (immediately or later) by calling `scroll-editor-to`.

The *localx*, *localy*, *width*, and *height* arguments specify the area that needs to be visible in *snip*'s coordinate system.

When the specified region cannot fit in the visible area, *bias* indicates which end of the region to display. When *bias* is 'start, then the top-left of the region is displayed. When *bias* is 'end, then the bottom-right of the region is displayed. Otherwise, *bias* must be 'none.

```
(send an-editor select-all) → void?
```

Selects all data in the editor

```
(send an-editor set-active-canvas canvas) → void?
  canvas : (is-a?/c editor-canvas%)
```

Sets the active canvas for this editor. (See get-active-canvas.)

Normally, this method is called only by on-focus in editor-canvas% in an editor canvas that is displaying an editor.

```
(send an-editor set-admin admin) → void?
  admin : (or/c (is-a?/c editor-admin%) #f)
```

Sets the editor's administrator. This method is only called by an administrator.

The administrator of an editor can be changed by by the system, and such changes do not go through this method. A program cannot detect when the administrator changes except by polling get-admin.

```
(send an-editor set-caret-owner  snip
                                 [domain]) → void?
  snip : (or/c (is-a?/c snip%) #f)
  domain : (or/c 'immediate 'display 'global) = 'immediate
```

Attempts to give the keyboard focus to *snip*. If *snip* is #f, then the caret is taken away from any snip in the editor that currently has the caret and restored to this editor.

If the keyboard focus is moved to *snip* and the editor has the real keyboard focus, the own-caret method of the snip will be called.

If #f is provided as the new owner, then the local focus is moved to the editor itself. Otherwise, the local focus is moved to the specified snip.

The domain of focus-setting is one of:

- 'immediate — only set the focus owner within the editor
- 'display — make this editor or the new focus owner get the keyboard focus among the editors in this editor's display (if this is an embedded editor)
- 'global — make this editor or the new focus owner get the keyboard focus among all elements in the editor's frame

The focus state of an editor can be changed by by the system, and such changes do not go through this method; use on-focus to monitor focus changes.

See also get-focus-snip.

```
(send an-editor set-cursor cursor
                           [override?]) → void?
  cursor : (or/c (is-a?/c cursor%) #f)
  override? : any/c = #t
```

Sets the custom cursor for the editor to *cursor*. If *override?* is a true value and *cursor* is not #f, then this cursor overrides cursor settings in embedded editors.

If the custom cursor is #f, the current cursor is removed, and a cursor is selected automatically by the editor (depending on whether the cursor is pointing at a clickback). See adjust-cursor for more information about the default selection.

An embedding editor's custom cursor can override the cursor of an embedded editor—even if the embedded editor has the caret—if the cursor is specified as an overriding cursor.

```
(send an-editor set-filename filename
                            [temporary?]) → void?
  filename : (or/c path-string? #f)
  temporary? : any/c = #f
```

Sets the filename to *filename*. If *filename* is #f or *temporary?* is a true value, then the user will still be prompted for a name on future calls to save-file and load-file.

This method is also called when the filename changes through any method (such as load-file).

```
(send an-editor set-inactive-caret-threshold threshold) → void?
  threshold : (or/c 'no-caret 'show-inactive-caret 'show-caret)
```

Sets the threshold for painting an inactive selection. See get-inactive-caret-threshold for more information.

```
(send an-editor set-keymap [keymap]) → void?
  keymap : (or/c (is-a?/c keymap%) #f) = #f
```

Sets the current keymap for the editor. A `#f` argument removes all key mapping.

```
(send an-editor set-load-overwrites-styles overwrite?) → void?
  overwrite? : any/c
```

Determines whether named styles in the current style list are replaced by `load-file` when the loaded file contains style specifications.

See also `get-load-overwrites-styles` and `read-from-file`.

```
(send an-editor set-max-height width) → void?
  width : (or/c (and/c real? (not/c negative?)) 'none)
```

Sets the maximum display height for the contents of the editor. A value less or equal to `0` indicates that there is no maximum.

Setting the height is disallowed when the editor is internally locked for reflowing (see also §1.3.8 “Internal Editor Locks”).

```
(send an-editor set-max-undo-history count) → void?
  count : (or/c exact-nonnegative-integer? 'forever)
```

Sets the maximum number of undoables that will be remembered by the editor. The default is `0`, which disables undo. The symbol `'forever` is accepted as a synonym for a very large number.

```
(send an-editor set-max-width width) → void?
  width : (or/c (and/c real? (not/c negative?)) 'none)
```

Sets the maximum display width for the contents of the editor; zero or `'none` indicates that there is no maximum. In a text editor, having no maximum disables automatic line breaking, and the minimum (positive) maximum width depends on the width of the autowrap bitmap.

Setting the width is disallowed when the editor is internally locked for reflowing (see also §1.3.8 “Internal Editor Locks”).

See also `set-autowrap-bitmap`.

```
(send an-editor set-min-height width) → void?
  width : (or/c (and/c real? (not/c negative?)) 'none)
```

Sets the minimum display height for the contents of the editor; zero or 'none indicates that there is no minimum.

Setting the height is disallowed when the editor is internally locked for reflowing (see also §1.3.8 "Internal Editor Locks").

```
(send an-editor set-min-width width) → void?
  width : (or/c (and/c real? (not/c negative?)) 'none)
```

Sets the minimum display width for the contents of the editor; zero or 'none indicates that there is no minimum.

Setting the width is disallowed when the editor is internally locked for reflowing (see also §1.3.8 "Internal Editor Locks").

```
(send an-editor set-modified modified?) → void?
  modified? : any/c
```

Sets the modified state of the editor. Usually, the state is changed automatically after an insertion, deletion, or style change by calling this method. (This method is also called when the modification state changes through *any* method.) This method is usually not called when the state of the flag is not changing.

See also is-modified? and on-snip-modified.

When *modified?* is true, then an internal modify-counter is set to 1.

When *modified?* is #f and the editor's undo or redo stack contains a system-created undoer that resets the modified state (because the preceding undo or redo action puts the editor back to a state where the modification state was #f), the undoer is disabled.

Regardless of the value of *modified?*, the editor's adminstrator's modified method is called.

Finally, if *modified?* is #f and the internal modify-counter is set to 0, then the set-unmodified method is called on every snip within the editor.

```
(send an-editor set-paste-text-only text-only?) → void?
  text-only? : any/c
```

Sets whether the editor accepts only text from the clipboard, or both text and snips. By default, an editor accepts both text and snips.

See also get-paste-text-only.

```
(send an-editor set-snip-data thesnip data) → void?
  thesnip : (is-a?/c snip%)
  data : (is-a?/c editor-data%)
```

Sets extra data associated with the snip (e.g., location information in a pasteboard). See §1.3.2.1.2 “Editor Data” for more information.

```
(send an-editor set-style-list style-list) → void?
  style-list : (is-a?/c style-list%)
```

Sets the editor’s style list. Styles currently in use with the old style list will be “moved” to the new style list. In this “move,” if a named style already exists in the new style list, then the new style with the same name will be used in place of the old style.

Setting the style list is disallowed when the editor is internally locked for reflowing (see also §1.3.8 “Internal Editor Locks”).

```
(send an-editor size-cache-invalid) → void?
```

This method is called when the drawing context given to the editor by its administrator changes in a way that makes cached size information (such as the width of a string) invalid.

The default implementation eventually propagates the message to snips, and, more generally, causes location information to be recalculated on demand.

See also `invalidate-bitmap-cache`.

```
(send an-editor style-has-changed style) → void?
  style : (or/c (is-a?/c style<%>) #f)
```

Notifies the editor that a style in its style list has changed. This method is automatically registered with the editor’s style list using `notify-on-change` in `style-list%` and automatically deregistered when the style list is removed from the editor.

See `notify-on-change` in `style-list%` for more information.

```
(send an-editor undo) → void?
```

Undoes the last editor change, if undos have been enabled by calling `set-max-undo-history` with a non-zero integer.

If the editor is currently performing an undo or redo, the method call is ignored.

The user may enable Emacs-style undo for editors; see §3.1 “Preferences”. Normally, undo

operations add to the redo stack (see redo), and any undoable (non-undo) operation clears the redo stack. With Emacs-style undo, the redo stack is added back to the undo stack, along with the original undos, so that a complete history is kept in the undo stack.

The system may perform an undo without calling this method in response to other method calls. Use methods such as on-change to monitor editor content changes.

See also add-undo .

```
(send an-editor use-file-text-mode) → boolean?
(send an-editor use-file-text-mode on?) → void?
  on? : any/c
```

Gets or sets whether the current platform's text mode is used for writing files in 'text or 'text-force-cr mode, which affects the way that newlines are written. The setting is consulted by save-file after on-save-file is called. See also load-file for information on file modes.

```
(send an-editor write-footers-to-file stream) → boolean?
  stream : (is-a?/c editor-stream-out%)
```

See write-headers-to-file.

```
(send an-editor write-headers-to-file stream) → boolean?
  stream : (is-a?/c editor-stream-out%)
```

Specification: Called when the editor is being saved to a file. The return value is #t if there are no errors, #f otherwise. Override this method to add custom header data to a file, but always call the inherited method so that it can write its own extra headers.

To write a header item, call begin-write-header-footer-to-file, passing a box for an integer. Then write the header data and end by calling end-write-header-footer-to-file, passing back the integer that was put into the box. Follow this procedure correctly or the file will be corrupted.

Default implementation: Does nothing.

```
(send an-editor write-to-file stream) → boolean?
  stream : (is-a?/c editor-stream-out%)
```

Writes the current editor contents to the given stream. The return value is #t if there are no errors, #f otherwise. See also §1.3.2 "File Format".

If the editor's style list has already been written to the stream, it is not re-written. Instead, the editor content indicates that the editor shares a previously-written style list. This sharing

will be recreated when the stream is later read.

2.5.3 editor-admin%

```
editor-admin% : class?
  superclass: object%
```

See §1.3.1.1 "Administrators" for information about the role of administrators. The editor-admin% class is never instantiated directly. It is not even instantiated through derived classes by most programmers; each editor-canvas% and editor-snip% object creates its own administrator. However, it may be useful to derive a new instance of this class to display editors in a new context. Also, it may be useful to call the methods of an existing administrator from an owned editor.

To create a new editor-admin% class, all methods described here must be overridden. They are all invoked by the administrator's editor.

```
(new editor-admin%) → (is-a?/c editor-admin%)
```

Creates a (useless) editor administrator.

```
(send an-editor-admin get-dc [x y])
 → (or/c (is-a?/c dc<%>) false/c)
  x : (or/c (box/c real?) false/c) = #f
  y : (or/c (box/c real?) false/c) = #f
```

Specification: Returns either the drawing context into which the editor is displayed, or the context into which it is currently being drawn. When the editor is not embedded, the returned context is always the drawing content into which the editor is displayed. If the editor is not displayed, #f is returned.

The origin of the drawing context is also returned, translated into the local coordinates of the editor. For an embedded editor, the returned origin is reliable only while the editor is being drawn, or while it receives a mouse or keyboard event.

The *x* box is filled with the x-origin of the DC in editor coordinates, unless *x* is #f. The *y* box is filled with the y-origin of the DC in editor coordinates, unless *y* is #f.

See also editor-location-to-dc-location in editor<%> and dc-location-to-editor-location in editor<%>.

Default implementation: Fills all boxes with 0.0 and returns #f.

```
(send an-editor-admin get-max-view  x
                                    y
                                    w
                                    h
                                   [full?]) → void?
  x : (or/c (box/c real?) false/c)
  y : (or/c (box/c real?) false/c)
  w : (or/c (box/c (and/c real? (not/c negative?))) false/c)
  h : (or/c (box/c (and/c real? (not/c negative?))) false/c)
  full? : any/c = #f
```

Specification: Same as `get-view` unless the editor is visible in multiple standard displays. If the editor has multiple displays, a region is computed that includes the visible region in all displays.

See `get-view`.

Default implementation: Fills all boxes with `0.0`.

```
(send an-editor-admin get-view x y w h [full?]) → void?
  x : (or/c (box/c real?) false/c)
  y : (or/c (box/c real?) false/c)
  w : (or/c (box/c (and/c real? (not/c negative?))) false/c)
  h : (or/c (box/c (and/c real? (not/c negative?))) false/c)
  full? : any/c = #f
```

Specification: Gets the visible region of the editor within its display (in editor coordinates), or the overall size of the viewing region in the editor's top-level display (for an embedded editor).

If the display is an editor canvas, see also `reflow-container`. The viewing area within an editor canvas is not the full client area of the canvas, because an editor canvas installs a whitespace border around a displayed editor within the client area.

The calculation of the editor's visible region is based on the current size and scrollbar values of the top-level display. For an editor canvas display, the region reported by `get-view` does not depend on whether the canvas is hidden, obscured by other windows, or moved off the edge of the screen.

The `x` box is filled with the left edge of the visible region in editor coordinates, unless `x` is `#f`. The `y` box is filled with the top edge of the visible region in editor coordinates, unless `y` is `#f`. The `w` box is filled with the width of the visible region, which may be larger than the editor itself, unless `w` is `#f`. The `h` box is filled with the height of the visible region, which may be larger than the editor itself, unless `h` is `#f`.

If an editor is fully visible and *full?* is #f, then *x* and *y* will both be filled with 0.

If *full?* is a true value, then the returned area is the view area of the top-level display for the editor. This result is different only when the editor is embedded in another editor; in that case, the *x* and *y* values may be meaningless, because they are in the coordinate system of the immediate editor within the top-level display.

Default implementation: Fills all boxes with 0.0.

```
(send an-editor-admin grab-caret [domain]) → void?
  domain : (one-of/c 'immediate 'display 'global) = 'global
```

Specification: Called by the editor to request the keyboard focus. If the request is granted, then the administered editor's own-caret method will be called.

See set-caret-owner for information about the possible values of *domain*.

Default implementation: Does nothing.

```
(send an-editor-admin modified modified?) → void?
  modified? : any/c
```

Specification: Called by the editor to report that its modification state has changed to either modified or unmodified.

See also set-modified in editor<%>.

Default implementation: Does nothing.

```
(send an-editor-admin needs-update localx
                                   localy
                                   w
                                   h)      → void?
  localx : real?
  localy : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Specification: Called by the editor to request a refresh to its displayed representation. When the administrator decides that the displayed should be refreshed, it calls the editor's refresh method.

The *localx*, *localy*, *w*, and *h* arguments specify a region of the editor to be updated (in editor coordinates).

Default implementation: Does nothing.

```
(send an-editor-admin popup-menu menu x y) → boolean?
  menu : (is-a?/c popup-menu%)
  x : real?
  y : real?
```

Specification:

Pops up the given `popup-menu%` object at the specified coordinates (in this window's coordinates), and returns after handling an unspecified number of events; the menu may still be popped up when this method returns. If a menu item is selected from the popup-menu, the callback for the menu item is called. (The eventspace for menu item's callback is the administrator's display's eventspace.)

While the menu is popped up, its target is set to the top-level editor in this administrator's display. See `get-popup-target` for more information.

The result is `#t` if the popup succeeds, `#f` otherwise (independent of whether the user selects an item in the popup menu).

The menu is displayed at *x* and *y* in editor coordinates.

Default implementation: Returns `#f`.

```
(send an-editor-admin refresh-delayed?) → boolean?
```

Specification: Returns `#t` if updating on this administrator's display is currently delayed (usually by `begin-edit-sequence` in `editor<%>` in an enclosing editor).

Default implementation: Returns `#f`.

```
(send an-editor-admin resized refresh?) → void?
  refresh? : any/c
```

Specification: Called by the editor to notify its display that the editor's size or scroll count has changed, so the scrollbars need to be adjusted to reflect the new size. The editor generally needs to be updated after a resize, but the editor decides whether the update should occur immediately. If *refresh?* is not `#f`, then the editor is requesting to be updated immediately.

Default implementation: Does nothing.

```
(send an-editor-admin scroll-to  localx
                                 localy
                                 w
                                 h
                                [refresh?
                                 bias])    → boolean?
  localx : real?
  localy : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
  refresh? : any/c = #t
  bias : (one-of/c 'start 'end 'none) = 'none
```

Specification: Called by the editor to request scrolling so that the given region is visible. The editor generally needs to be updated after a scroll, but the editor decides whether the update should occur immediately.

The *localx*, *localy*, *w*, and *h* arguments specify a region of the editor to be made visible by the scroll (in editor coordinates).

If *refresh?* is not `#f`, then the editor is requesting to be updated immediately.

The *bias* argument is one of:

- `'start` — if the range doesn't fit in the visible area, show the top-left region
- `'none` — no special scrolling instructions
- `'end` — if the range doesn't fit in the visible area, show the bottom-right region

The return value is `#t` if the display is scrolled, `#f` if not (either because the requested region is already visible, because the display has zero size, or because the editor is currently printing.)

Default implementation: Return `#f`

```
(send an-editor-admin update-cursor) → void?
```

Specification: Queues an update for the cursor in the display for this editor. The actual cursor used will be determined by calling the editor's `adjust-cursor` method.

Default implementation: Does nothing.

2.5.4 editor-canvas%

---

editor-canvas% : class?
  superclass: object%
  extends: canvas<%>

An editor-canvas% object manages and displays a text% or pasteboard% object.

```
(new editor-canvas%  [parent parent]
                     [[editor editor]
                      [style style]
                      [scrolls-per-page scrolls-per-page]
                      [label label]
                      [wheel-step wheel-step]
                      [line-count line-count]
                      [horizontal-inset horizontal-inset]
                      [vertical-inset vertical-inset]
                      [enabled enabled]
                      [vert-margin vert-margin]
                      [horiz-margin horiz-margin]
                      [min-width min-width]
                      [min-height min-height]
                      [stretchable-width stretchable-width]
                      [stretchable-height stretchable-height]])
 → (is-a?/c editor-canvas%)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  editor : (or/c (or/c (is-a?/c text%) (is-a?/c pasteboard%)) false/c)
         = #f
  style : (listof (one-of/c 'no-border 'control-border 'combo
                            'no-hscroll 'no-vscroll
                            'hide-hscroll 'hide-vscroll
                            'auto-vscroll 'auto-hscroll
                            'resize-corner 'deleted 'transparent))
        = null
  scrolls-per-page : (integer-in 1 10000) = 100
  label : (or/c label-string? false/c) = #f
  wheel-step : (or/c (integer-in 1 10000) false/c) = 3
  line-count : (or/c (integer-in 1 1000) false/c) = #f
  horizontal-inset : (integer-in 0 1000) = 5
  vertical-inset : (integer-in 0 1000) = 5
  enabled : any/c = #t
  vert-margin : (integer-in 0 1000) = 0
  horiz-margin : (integer-in 0 1000) = 0
```

```
min-width : (integer-in 0 10000) = graphical-minimum-width
min-height : (integer-in 0 10000) = graphical-minimum-height
stretchable-width : any/c = #t
stretchable-height : any/c = #t
```

If a canvas is initialized with `#f` for *editor*, install an editor later with `set-editor`.

The *style* list can contain the following flags:

- `'no-border` — omits a border around the canvas
- `'control-border` — gives the canvas a border that is like a `text-field%` control
- `'combo` — gives the canvas a combo button that is like a `combo-field%` control; this style is intended for use with `'control-border`, `'hide-hscroll`, and `'hide-vscroll`
- `'no-hscroll` — disallows horizontal scrolling and hides the horizontal scrollbar
- `'no-vscroll` — disallows vertical scrolling and hides the vertical scrollbar
- `'hide-hscroll` — allows horizontal scrolling, but hides the horizontal scrollbar
- `'hide-vscroll` — allows vertical scrolling, but hides the vertical scrollbar
- `'auto-hscroll` — automatically hides the horizontal scrollbar when unneeded (unless `'no-hscroll` or `'hide-hscroll` is specified)
- `'auto-vscroll` — automatically hides the vertical scrollbar when unneeded (unless `'no-vscroll` or `'hide-vscroll` is specified)
- `'resize-corner` — leaves room for a resize control at the canvas's bottom right when only one scrollbar is visible
- `'deleted` — creates the canvas as initially hidden and without affecting *parent*'s geometry; the canvas can be made active later by calling *parent*'s `add-child` method
- `'transparent` — the canvas is "erased" before an update using it's parent window's background

While vertical scrolling of text editors is based on lines, horizontal scrolling and pasteboard vertical scrolling is based on a fixed number of steps per horizontal page. The *scrolls-per-page* argument sets this value.

If provided, the *wheel-step* argument is passed on to the `wheel-step` method. The default wheel step can be overridden globally though the `'GRacket:wheelStep` preference; see §3.1 "Preferences".

If *line-count* is not `#f`, it is passed on to the `set-line-count` method.

If *horizontal-inset* is not 5, it is passed on to the horizontal-inset method. Similarly, if *vertical-inset* is not 5, it is passed on to the vertical-inset method.

For information about the *enabled* argument, see window<%>. For information about the *horiz-margin* and *vert-margin* arguments, see subarea<%>. For information about the *min-width*, *min-height*, *stretchable-width*, and *stretchable-height* arguments, see area<%>.

```
(send an-editor-canvas allow-scroll-to-last) → boolean?
(send an-editor-canvas allow-scroll-to-last on?) → void?
  on? : any/c
```

Enables or disables last-line scrolling, or gets the current enable state. If last-line scrolling is enabled, then an editor displayed in this canvas can be scrolled so that the last line of text is at the top of the canvas (or bottom of the canvas when bottom-based scrolling is enabled; see scroll-with-bottom-base). By default, an editor can only be scrolled until the last line is at the bottom (or top) of the canvas.

```
(send an-editor-canvas allow-tab-exit) → boolean?
(send an-editor-canvas allow-tab-exit on?) → void?
  on? : any/c
```

Gets or sets whether tab-exit is enabled for the editor canvas. When tab-exit is enabled, the user can move the keyboard focus out of the editor using the Tab and arrow keys, invoke the default button using the Enter/Return key, or invoke a dialog's close action with Escape. By default, tab-exit is disabled.

When tab-exit is enabled for an editor canvas, Tab and Enter keyboard events are consumed by a frame's default on-traverse-char method; in addition, a dialog's default method consumes Escape key events. Otherwise, on-traverse-char allows the keyboard events to be propagated to the canvas.

```
(send an-editor-canvas call-as-primary-owner f) → any
  f : (-> any)
```

Calls a thunk and returns the value. While the thunk is being called, if the canvas has an editor, the editor's get-admin method returns the administrator for this canvas. This method is only useful when an editor is displayed in multiple canvases.

```
(send an-editor-canvas force-display-focus) → boolean?
(send an-editor-canvas force-display-focus on?) → void?
  on? : any/c
```

Enables or disables force-focus mode. In force-focus mode, the caret or selection of the

editor displayed in this canvas is drawn even when the canvas does not have the keyboard focus.

```
(send an-editor-canvas get-editor)
 → (or/c (or/c (is-a?/c text%) (is-a?/c pasteboard%)) false/c)
```

Returns the editor currently displayed by this canvas, or `#f` if the canvas does not have an editor.

```
(send an-editor-canvas get-line-count)
 → (or/c (integer-in 1 1000) false/c)
```

Returns a line count installed with `set-line-count`, or `#f` if no minimum line count is set.

```
(send an-editor-canvas horizontal-inset)
 → (integer-in 1 10000)
(send an-editor-canvas horizontal-inset step) → void?
  step : (integer-in 1 10000)
```

Gets or sets the number of pixels within the canvas reserved to the left and right of editor content. The default is `5`.

```
(send an-editor-canvas lazy-refresh) → boolean?
(send an-editor-canvas lazy-refresh on?) → void?
  on? : any/c
```

Enables or disables lazy-refresh mode, or gets the current enable state. In lazy-refresh mode, the canvas's `refresh` method is called when the window needs to be updated, rather than `on-paint`. By default, an `editor-canvas%` object is *not* in lazy-refresh mode.

```
(send an-editor-canvas on-char event) → void?
  event : (is-a?/c key-event%)
```

Overrides `on-char` in `canvas<%>`.

Handles `'wheel-up` and `'wheel-down` events by scrolling vertically. Otherwise, passes the event to the canvas's editor, if any, by calling its `on-char` method.

See also `get-editor`.

```
(send an-editor-canvas on-event event) → void?
  event : (is-a?/c mouse-event%)
```

Overrides on-event in canvas<%>.

Passes the event to the canvas's editor, if any, by calling its on-event method.

See also get-editor.

```
(send an-editor-canvas on-focus on?) → void?
  on? : any/c
```

Overrides on-focus in window<%>.

Enables or disables the caret in the display's editor, if there is one.

```
(send an-editor-canvas on-paint) → void?
```

Overrides on-paint in canvas<%>.

Repaints the editor, or clears the canvas if no editor is being displayed.

This method is called after clearing the margin around the editor, unless the canvas is created with the 'transparent style, but the editor area is not automatically cleared. In other words, editor-canvas% update by default is like canvas% update with the 'no-autoclear style, except that the margin around the editor area is always cleared.

```
(send an-editor-canvas on-size width
                                height) → void?
  width : (integer-in 0 10000)
  height : (integer-in 0 10000)
```

Overrides on-size in window<%>.

If the canvas is displaying an editor, its on-display-size method is called.

```
(send an-editor-canvas scroll-to  localx
                                  localy
                                  w
                                  h
                                  refresh?
                                 [bias])    → boolean?
  localx : real?
  localy : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
  refresh? : any/c
```

```
  bias : (one-of/c 'start 'end 'none) = 'none
```

Requests scrolling so that the given region in the currently displayed editor is made visible.

The *localx*, *localy*, *w*, and *h* arguments specify a region of the editor to be made visible by the scroll (in editor coordinates).

If *refresh?* is not `#f`, then the editor is updated immediately after a successful scroll.

The *bias* argument is one of:

- `'start` — if the range doesn't fit in the visible area, show the top-left region
- `'none` — no special scrolling instructions
- `'end` — if the range doesn't fit in the visible area, show the bottom-right region

The return value is `#t` if the display is scrolled, `#f` if not (either because the requested region is already visible, because the display has zero size, or because the editor is currently printing).

```
(send an-editor-canvas scroll-with-bottom-base) → boolean?
(send an-editor-canvas scroll-with-bottom-base on?) → void?
  on? : any/c
```

Enables or disables bottom-base scrolling, or gets the current enable state. If bottom-base scrolling is on, then scroll positions are determined by line boundaries aligned with the bottom of the viewable area (rather than with the top of the viewable area). If last-line scrolling is also enabled (see `allow-scroll-to-last`), then the editor is bottom-aligned in the display area even when the editor does not fill the viewable area.

```
(send an-editor-canvas set-editor  edit
                                   [redraw?]) → void?
  edit : (or/c (or/c (is-a?/c text%) (is-a?/c pasteboard%)) false/c)
  redraw? : any/c = #t
```

Sets the editor that is displayed by the canvas, releasing the current editor (if any). If the new editor already has an administrator that is not associated with an `editor-canvas%`, then the new editor is *not* installed into the canvas.

If *redraw?* is `#f`, then the editor is not immediately drawn; in this case, something must force a redraw later (e.g., a call to the `on-paint` method).

If the canvas has a line count installed with `set-line-count`, the canvas's minimum height is adjusted.

```
(send an-editor-canvas set-line-count count) → void?
  count : (or/c (integer-in 1 1000) false/c)
```

Sets the canvas's graphical minimum height to display a particular number of lines of text. The line height is determined by measuring the difference between the top and bottom of a displayed editor's first line. The minimum height is not changed until the canvas gets an editor. When the canvas's editor is changed, the minimum height is recalculated.

If the line count is set to `#f`, then the canvas's graphical minimum height is restored to its original value.

```
(send an-editor-canvas vertical-inset) → (integer-in 1 10000)
(send an-editor-canvas vertical-inset step) → void?
  step : (integer-in 1 10000)
```

Gets or sets the number of pixels within the canvas reserved above and below editor content. The default is `5`.

```
(send an-editor-canvas wheel-step)
 → (or/c (integer-in 1 10000) false/c)
(send an-editor-canvas wheel-step step) → void?
  step : (or/c (integer-in 1 10000) false/c)
```

Gets or sets the number of vertical scroll steps taken for one click of the mouse wheel via a `'wheel-up` or `'wheel-down` `key-event%`. A `#f` value disables special handling for wheel events (i.e., wheel events are passed on to the canvas's editor).

2.5.5 editor-data%

```
editor-data% : class?
  superclass: object%
```

An `editor-data%` object contains extra data associated to a snip or region in an editor. See also §1.3.2.1.2 "Editor Data".

```
(new editor-data%) → (is-a?/c editor-data%)
```

The element returned by `get-next` is initialized to `#f`.

```
(send an-editor-data get-dataclass)
 → (or/c (is-a?/c editor-data-class%) false/c)
```

Gets the class for this data.

```
(send an-editor-data get-next)
 → (or/c (is-a?/c editor-data%) false/c)
```

Gets the next editor data element in a list of editor data elements. A `#f` terminates the list.

```
(send an-editor-data set-dataclass v) → void?
  v : (is-a?/c editor-data-class%)
```

Sets the class for this data.

```
(send an-editor-data set-next v) → void?
  v : (or/c (is-a?/c editor-data%) false/c)
```

Sets the next editor data element in a list of editor data elements. A `#f` terminates the list.

```
(send an-editor-data write f) → boolean?
  f : (is-a?/c editor-stream-out%)
```

Specification: Writes the data to the specified stream, returning `#t` if data is written successfully or `#f` otherwise.

Default implementation: Returns `#f`.

2.5.6 editor-data-class%

```
editor-data-class% : class?
  superclass: object%
```

An `editor-data-class%` object defines a type for `editor-data%` objects. See also §1.3.2.1.2 “Editor Data”.

```
(new editor-data-class%) → (is-a?/c editor-data-class%)
```

Creates a (useless) instance.

```
(send an-editor-data-class get-classname) → string?
```

Gets the name of the class. Names starting with `wx` are reserved for internal use.

```
(send an-editor-data-class read f)
 → (or/c (is-a?/c editor-data%) false/c)
  f : (is-a?/c editor-stream-in%)
```

Reads a new data object from the given stream, returning `#f` if there is an error.

```
(send an-editor-data-class set-classname v) → void?
  v : string?
```

Sets the name of the class. Names starting with `wx` are reserved for internal use.

An editor data class name should usually have the form `"(lib\n ...)"` to enable on-demand loading of the class; see §1.3.2.1.2 “Editor Data” for details.

2.5.7 editor-data-class-list<%>

```
editor-data-class-list<%> : interface?
```

Each eventspace has an instance of `editor-data-class-list<%>`, obtained with `(get-the-editor-data-class-list)`. New instances cannot be created directly. This list keeps a list of editor data classes; this list is needed for loading snips from a file. See also §1.3.2.1.2 “Editor Data”.

```
(send an-editor-data-class-list add snipclass) → void?
  snipclass : (is-a?/c editor-data-class%)
```

Adds a snip data class to the list. If a class with the same name already exists in the list, this one will not be added.

```
(send an-editor-data-class-list find name)
 → (or/c (is-a?/c snip-class%) false/c)
  name : string?
```

Finds a snip data class from the list with the given name, returning `#f` if none can be found.

```
(send an-editor-data-class-list find-position class)
 → exact-nonnegative-integer?
  class : (is-a?/c editor-data-class%)
```

Returns an index into the list for the specified class.

```
(send an-editor-data-class-list nth n)
 → (or/c (is-a?/c editor-data-class%) false/c)
  n : exact-nonnegative-integer?
```

Returns the nth class in the list (counting from 0), returning `#f` if the list has n or less classes.

```
(send an-editor-data-class-list number)
 → exact-nonnegative-integer?
```

Returns the number of editor data classes in the list.

2.5.8 editor-snip-editor-admin<%>

```
editor-snip-editor-admin<%> : interface?
```

An instance of this administrator interface is created with each `editor-snip%` object; new instances cannot be created directly.

```
(send an-editor-snip-editor-admin get-snip)
 → (is-a?/c editor-snip%)
```

Returns the snip that owns this administrator (and displays the editor controlled by the administrator, if any).

2.5.9 editor-snip%

```
editor-snip% : class?
  superclass: snip%
```

An `editor-snip%` object is a `snip%` object that contains and displays an `editor<%>` object. This snip class is used to insert an editor as a single item within another editor.

```
(new editor-snip% [[editor editor]
                   [with-border? with-border?]
                   [left-margin left-margin]
                   [top-margin top-margin]
                   [right-margin right-margin]
                   [bottom-margin bottom-margin]
                   [left-inset left-inset]
                   [top-inset top-inset]
                   [right-inset right-inset]
                   [bottom-inset bottom-inset]
                   [min-width min-width]
                   [max-width max-width]
                   [min-height min-height]
                   [max-height max-height]])
 → (is-a?/c editor-snip%)
  editor : (or/c (is-a?/c text%) false/c) = #f
  with-border? : any/c = #t
  left-margin : exact-nonnegative-integer? = 5
  top-margin : exact-nonnegative-integer? = 5
  right-margin : exact-nonnegative-integer? = 5
  bottom-margin : exact-nonnegative-integer? = 5
  left-inset : exact-nonnegative-integer? = 1
  top-inset : exact-nonnegative-integer? = 1
  right-inset : exact-nonnegative-integer? = 1
  bottom-inset : exact-nonnegative-integer? = 1
  min-width : (or/c (and/c real? (not/c negative?)) (one/of 'none))
            = 'none
  max-width : (or/c (and/c real? (not/c negative?)) (one/of 'none))
            = 'none
  min-height : (or/c (and/c real? (not/c negative?)) (one/of 'none))
             = 'none
  max-height : (or/c (and/c real? (not/c negative?)) (one/of 'none))
             = 'none
```

If *editor* is non-#f, then it will be used as the editor contained by the snip. See also set-editor.

If *with-border?* is not #f, then a border will be drawn around the snip. The editor display will be inset in the snip area by the amounts specified in the -margin arguments. The border will be drawn with an inset specified by the -inset arguments.

See get-inset and get-margin for information about the inset and margin arguments.

```
(send an-editor-snip adjust-cursor dc
                                   x
                                   y
                                   editorx
                                   editory
                                   event)
 → (or/c (is-a?/c cursor%) false/c)
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  editorx : real?
  editory : real?
  event : (is-a?/c mouse-event%)
```

Overrides `adjust-cursor` in `snip%`.

Gets a cursor from the embedded editor by calling its `adjust-cursor` method.

```
(send an-editor-snip border-visible?) → boolean?
```

Returns `#t` if the snip has a border draw around it, `#f` otherwise.

See also `show-border`.

```
(send an-editor-snip get-align-top-line) → boolean?
```

Reports whether the snip is in align-top-line mode. See `get-extent` for more information.

See also `set-align-top-line`.

```
(send an-editor-snip get-editor)
 → (or/c (or/c (is-a?/c text%) (is-a?/c pasteboard%)) false/c)
```

Returns the editor contained by the snip, or `#f` is there is no editor.

```
(send an-editor-snip get-extent  dc
                                 x
                                 y
                                [w
                                 h
                                 descent
                                 space
                                 lspace
                                 rspace]) → void?
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  w : (or/c (box/c (and/c real? (not/c negative?))) false/c)
    = #f
  h : (or/c (box/c (and/c real? (not/c negative?))) false/c)
    = #f
  descent : (or/c (box/c (and/c real? (not/c negative?))) false/c)
          = #f
  space : (or/c (box/c (and/c real? (not/c negative?))) false/c)
        = #f
  lspace : (or/c (box/c (and/c real? (not/c negative?))) false/c)
         = #f
  rspace : (or/c (box/c (and/c real? (not/c negative?))) false/c)
         = #f
```

Overrides get-extent in snip%.

Calls its editor's get-extent method, then adds the editor snip's margins.

The top space always corresponds to the space of the editor's top line, plus the snip's top margin. Normally, the descent corresponds to the descent of the editor's last line plus the snip's bottom margin. However, if the snip is in align-top-line mode (see set-align-top-line), the descent corresponds to the descent of the top line, plus the height rest of the editor's lines, plus the snip's bottom margin.

If the editor is a text editor, then 1 is normally subtracted from the editor's width as returned by get-extent, because the result looks better for editing. If the snip is in tight-text-fit mode (see set-tight-text-fit) then 2 is subtracted from a text editor's width, eliminating the two pixels that the text editor reserves for the blinking caret. In addition, tight-text-fit mode subtracts an amount equal to the line spacing from the editor's height. By default, tight-text-fit mode is disabled.

```
(send an-editor-snip get-inset l t r b) → void?
  l : (box/c exact-nonnegative-integer?)
```

```
  t : (box/c exact-nonnegative-integer?)
  r : (box/c exact-nonnegative-integer?)
  b : (box/c exact-nonnegative-integer?)
```

Gets the current border insets for the snip. The inset sets how much space is left between the edge of the snip and the border.

The `l` box is filled with left inset. The `t` box is filled with top inset. The `r` box is filled with right inset. The `b` box is filled with bottom inset.

```
(send an-editor-snip get-margin l t r b) → void?
  l : (box/c exact-nonnegative-integer?)
  t : (box/c exact-nonnegative-integer?)
  r : (box/c exact-nonnegative-integer?)
  b : (box/c exact-nonnegative-integer?)
```

Gets the current margins for the snip. The margin sets how much space is left between the edge of the editor's contents and the edge of the snip.

The `l` box is filled with left margin. The `t` box is filled with top margin. The `r` box is filled with right margin. The `b` box is filled with bottom margin.

```
(send an-editor-snip get-max-height)
 → (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

Gets the maximum display height of the snip; zero or `'none` indicates that there is no maximum.

```
(send an-editor-snip get-max-width)
 → (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

Gets the maximum display width of the snip; zero or `'none` indicates that there is no maximum.

```
(send an-editor-snip get-min-height)
 → (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

Gets the minimum display height of the snip; zero or `'none` indicates that there is no minimum.

```
(send an-editor-snip get-min-width)
 → (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

Gets the minimum display width of the snip; zero or `'none` indicates that there is no mini-

mum.

(send *an-editor-snip* get-tight-text-fit) → boolean?

Reports whether the snip is in tight-text-fit mode. See get-extent for more information.

See also set-tight-text-fit.

(send *an-editor-snip* resize *w* *h*) → boolean?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))

Overrides resize in snip%.

Sets the snip's minimum and maximum width and height to the specified values minus the snip border space. See also set-min-width set-max-width set-max-height set-min-height.

Also sets the minimum and maximum width of the editor owned by the snip to the given width (minus the snip border space) via set-max-width and set-min-width.

(send *an-editor-snip* set-align-top-line *tight?*) → void?
  tight? : any/c

Enables or disables align-top-line mode. See get-extent for more information.

See also get-align-top-line.

(send *an-editor-snip* set-editor *editor*) → void?
  editor : (or/c (or/c (is-a?/c text%) (is-a?/c pasteboard%)) false/c)

Sets the editor contained by the snip, releasing the old editor in the snip (if any). If the new editor already has an administrator, then the new editor is *not* installed into the snip.

When an editor-snip% object is not inserted in an editor, it does not have an administrator. During this time, it does not give its contained editor an administrator, either. The administratorless contained editor can therefore "defect" to some other display with an administrator. When a contained editor defects and the snip is eventually inserted into a different editor, the snip drops the traitor contained editor, setting its contained editor to #f.

(send *an-editor-snip* set-inset *l* *t* *r* *b*) → void?
  l : exact-nonnegative-integer?
  t : exact-nonnegative-integer?
  r : exact-nonnegative-integer?

```
  b : exact-nonnegative-integer?
```

Sets the current border insets for the snip. The inset sets how much space is left between the edge of the snip and the border.

```
(send an-editor-snip set-margin l t r b) → void?
  l : exact-nonnegative-integer?
  t : exact-nonnegative-integer?
  r : exact-nonnegative-integer?
  b : exact-nonnegative-integer?
```

Sets the current margins for the snip. The margin sets how much space is left between the edge of the editor's contents and the edge of the snip.

```
(send an-editor-snip set-max-height h) → void?
  h : (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

An editor-snip% normally stretches to wrap around the size of the editor it contains. This method limits the height of the snip (and if the editor is larger, only part of the editor is displayed).

Zero or 'none disables the limit.

```
(send an-editor-snip set-max-width w) → void?
  w : (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

An editor-snip% normally stretches to wrap around the size of the editor it contains. This method limits the width of the snip (and if the editor is larger, only part of the editor is displayed). The contained editor's width limits are not changed by this method.

Zero or 'none disables the limit.

```
(send an-editor-snip set-min-height h) → void?
  h : (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

An editor-snip% normally stretches to wrap around the size of the editor it contains. This method sets the minimum height of the snip (and if the editor is smaller, the editor is top-aligned in the snip).

Zero or 'none disables the limit.

```
(send an-editor-snip set-min-width w) → void?
  w : (or/c (and/c real? (not/c negative?)) (one/of 'none))
```

An editor-snip% normally stretches to wrap around the size of the editor it contains. This method sets the minimum width of the snip (and if the editor is smaller, the editor is left-aligned in the snip). The contained editor's width limits are not changed by this method.

Zero or 'none disables the limit.

(send an-editor-snip set-tight-text-fit tight?) → void?
  tight? : any/c

Enables or disables tight-text-fit mode. See get-extent for more information.

See also get-tight-text-fit.

(send an-editor-snip show-border show?) → void?
  show? : any/c

Shows or hides the snip's border.

(send an-editor-snip style-background-used?) → boolean?

Returns #t if the snip uses its style's background and transparency information when drawing, #f otherwise.

See also use-style-background.

(send an-editor-snip use-style-background use?) → void?
  use? : any/c

Causes the snip to use or not used (the default) its style's background and transparency information for drawing the background within the snip's border.

If use? is #f, the style background and transparency information is ignored, otherwise is it used.

2.5.10 editor-stream-in%

---

editor-stream-in% : class?
  superclass: object%

An editor-stream-in% object is used to read editor information from a file or other input stream (such as the clipboard).

```
(make-object editor-stream-in% base)
 → (is-a?/c editor-stream-in%)
  base : (is-a?/c editor-stream-in-base%)
```

An in-stream base—possibly an editor-stream-in-bytes-base% object—must be supplied in base.

```
(send an-editor-stream-in get v) → (is-a?/c editor-stream-in%)
  v : (box/c exact-integer?)
(send an-editor-stream-in get v) → (is-a?/c editor-stream-in%)
  v : (box/c real?)
```

Reads data from the stream, returning itself. Reading from a bad stream always gives 0.

The v box is filled with the next integer or floating-point value in the stream.

```
(send an-editor-stream-in get-bytes [len])
 → (or/c bytes? false/c)
  len : (or/c (box/c exact-nonnegative-integer?) false/c) = #f
```

Like get-unterminated-bytes, but the last read byte is assumed to be a nul terminator and discarded. Use this method when data is written by a call to put without an explicit byte count, and use get-unterminated-bytes when data is written with an explicit byte count.

The len box is filled with the length of the byte string plus one (to indicate the terminator), unless len is #f.

```
(send an-editor-stream-in get-exact) → exact-integer?
```

Returns the next integer value in the stream.

```
(send an-editor-stream-in get-fixed v)
 → (is-a?/c editor-stream-in%)
  v : (box/c exact-integer?)
```

The v box is filled with a fixed-size integer from the stream obtained through get-fixed-exact.

```
(send an-editor-stream-in get-fixed-exact) → exact-integer?
```

Gets a fixed-sized integer from the stream. See put-fixed for more information. Reading from a bad stream always gives 0.

```
(send an-editor-stream-in get-inexact) → real?
```

Returns the next floating-point value in the stream.

```
(send an-editor-stream-in get-unterminated-bytes [len])
 → (or/c bytes? false/c)
  len : (or/c (box/c exact-nonnegative-integer?) false/c) = #f
```

Returns the next byte string from the stream. This is the recommended way to read bytes back in from a stream; use `put` with two arguments (passing along the length of the bytes) to write out the bytes to match this method.

Reading from a bad stream returns `#f` or `#""`.

Note that when `put` is not given a byte length, it includes an extra byte for a nul terminator; use `get-bytes` to read such byte strings.

The *len* box is filled with the length of the byte string, unless *len* is `#f`.

```
(send an-editor-stream-in jump-to pos) → void?
  pos : exact-nonnegative-integer?
```

Jumps to a given position in the stream.

```
(send an-editor-stream-in ok?) → boolean?
```

Returns `#t` if the stream is ready for reading, `#f` otherwise. Reading from a bad stream always returns `0` or `""`.

```
(send an-editor-stream-in remove-boundary) → void?
```

See `set-boundary`.

```
(send an-editor-stream-in set-boundary n) → void?
  n : exact-nonnegative-integer?
```

Sets a file-reading boundary at *n* bytes past the current stream location. If there is an attempt to read past this boundary, an error is signaled. The boundary is removed with a call to `remove-boundary`. Every call to `set-boundary` must be balanced by a call to `remove-boundary`.

Boundaries help keep a subroutine from reading too much data leading to confusing errors. However, a malicious subroutine can call `remove-boundary` on its own.

```
(send an-editor-stream-in skip n) → void?
  n : exact-nonnegative-integer?
```

Skips past the next *n* bytes in the stream.

```
(send an-editor-stream-in tell) → exact-nonnegative-integer?
```

Returns the current stream position.

2.5.11 editor-stream-in-base%

```
editor-stream-in-base% : class?
  superclass: object%
```

An editor-stream-in-base% object is used by an editor-stream-in% object to perform low-level reading of data.

The editor-stream-in-base% class is never instantiated directly, but the derived class editor-stream-in-bytes-base% can be instantiated. New derived classes must override all of the methods described in this section.

```
(send an-editor-stream-in-base bad?) → boolean?
```

Returns #t if there has been an error reading from the stream, #f otherwise.

```
(send an-editor-stream-in-base read data)
 → exact-nonnegative-integer?
  data : (and/c vector? (not immutable?))
```

Like read-bytes, but fills a supplied vector with Latin-1 characters instead of filling a byte string. This method is implemented by default via read-bytes.

```
(send an-editor-stream-in-base read-bytes bstr)
 → exact-nonnegative-integer?
  bstr : (and/c bytes? (not immutable?))
```

Reads bytes to fill the supplied byte string. The return value is the number of bytes read, which may be less than the number requested if the stream is emptied. If the stream is emptied, the next call to bad? must return #t.

(send *an-editor-stream-in-base* read-byte) → (or/c byte? #f)

Reads a single byte and return it, or returns #f if no more bytes are available. The default implementation of this method uses read-bytes.

(send *an-editor-stream-in-base* seek *pos*) → void?
  pos : exact-nonnegative-integer?

Moves to the specified absolute position in the stream.

(send *an-editor-stream-in-base* skip *n*) → void?
  n : exact-nonnegative-integer?

Skips past the next *n* characters in the stream.

(send *an-editor-stream-in-base* tell)
 → exact-nonnegative-integer?

Returns the current stream position.

2.5.12 editor-stream-in-bytes-base%

---

editor-stream-in-bytes-base% : class?
  superclass: editor-stream-in-base%

An editor-stream-in-bytes-base% object can be used to read editor data from a byte string.

(make-object editor-stream-in-bytes-base% *s*)
 → (is-a?/c editor-stream-in-bytes-base%)
  s : bytes?

Creates a stream base that reads from *s*.

2.5.13 editor-stream-out%

---

editor-stream-out% : class?
  superclass: object%

An editor-stream-out% object is used to write editor information to a file or other output stream (such as the clipboard).

```
(make-object editor-stream-out% base)
 → (is-a?/c editor-stream-out%)
  base : (is-a?/c editor-stream-out-base%)
```

An out-stream base—possibly an editor-stream-out-bytes-base% object—must be supplied in *base*.

```
(send an-editor-stream-out jump-to pos) → void?
  pos : exact-nonnegative-integer?
```

Jumps to a given position in the stream.

```
(send an-editor-stream-out ok?) → boolean?
```

Returns #t if the stream is ready for writing, #f otherwise. Writing to a bad stream has no effect.

```
(send an-editor-stream-out pretty-finish) → void?
```

Ensures that the stream ends with a newline. This method is called by write-editor-global-footer.

```
(send an-editor-stream-out pretty-start) → void?
```

Writes a "comment" into the stream that identifies the file format. This method is called by write-editor-global-header.

```
(send an-editor-stream-out put n v)
 → (is-a?/c editor-stream-out%)
  n : exact-nonnegative-integer?
  v : bytes?
(send an-editor-stream-out put v)
 → (is-a?/c editor-stream-out%)
  v : bytes?
(send an-editor-stream-out put v)
 → (is-a?/c editor-stream-out%)
  v : exact-integer?
(send an-editor-stream-out put v)
 → (is-a?/c editor-stream-out%)
```

v : real?

Writes *v*, or *n* bytes of *v*.

When *n* is supplied with a byte-string *v*, use get-unterminated-bytes to read the bytes later. This is the recommended way to write out bytes to be easily read in later; use get-unterminated-bytes to read the bytes back in.

If *n* is not supplied and *v* is a byte string, then for historical reasons, the actual number of bytes written includes a #\nul terminator, so use get-bytes instead of get-unterminated-bytes to read the bytes later.

(send *an-editor-stream-out* put-fixed *v*)
→ (is-a?/c editor-stream-out%)
v : exact-integer?

Puts a fixed-sized integer into the stream. This method is needed because numbers are usually written in a way that takes varying numbers of bytes. In some cases it is useful to temporary write a 0 to a stream, write more data, and then go back and change the 0 to another number; such a process requires a fixed-size number.

Numbers written to a stream with put-fixed must be read with get-fixed-exact or get-fixed.

(send *an-editor-stream-out* put-unterminated *v*)
→ (is-a?/c editor-stream-out%)
v : bytes?

The same as calling put with (bytes-length *v*) and *v*.

(send *an-editor-stream-out* tell) → exact-nonnegative-integer?

Returns the current stream position.

2.5.14 editor-stream-out-base%

---

editor-stream-out-base% : class?
superclass: object%

An editor-stream-out-base% object is used by an editor-stream-out% object to perform low-level writing of data.

The editor-stream-out-base% class is never instantiated directly, but the derived class

editor-stream-out-bytes-base% can be instantiated. New derived classes must override all of the methods described in this section.

(send an-editor-stream-out-base bad?) → boolean?

Returns #t if there has been an error writing to the stream, #f otherwise.

(send an-editor-stream-out-base seek pos) → void?
  pos : exact-nonnegative-integer?

Moves to the specified absolute position in the stream.

(send an-editor-stream-out-base tell)
 → exact-nonnegative-integer?

Returns the current stream position.

(send an-editor-stream-out-base write data) → void?
  data : (listof char?)

Writes data (encoded as Latin-1 characters) to the stream. This method is implemented by default via write-bytes.

(send an-editor-stream-out-base write-bytes bstr) → void?
  bstr : bytes?

Writes data to the stream.

2.5.15 editor-stream-out-bytes-base%

---

editor-stream-out-bytes-base% : class?
  superclass: editor-stream-out-base%

An editor-stream-out-bytes-base% object can be used to write editor data into a byte string.

(new editor-stream-out-bytes-base%)
 → (is-a?/c editor-stream-out-bytes-base%)

Creates an empty stream.

(send *an-editor-stream-out-bytes-base* get-bytes) → bytes?

Returns the current contents of the stream.

2.5.16 editor-wordbreak-map%

---

editor-wordbreak-map% : class?
  superclass: object%

An editor-wordbreak-map% objects is used with a text% objects to specify word-breaking criteria for the default wordbreaking function. See also set-wordbreak-map, get-wordbreak-map, find-wordbreak, and set-wordbreak-func.

A global object the-editor-wordbreak-map is created automatically and used as the default map for all text% objects.

A wordbreak objects implements a mapping from each character to a list of symbols. The following symbols are legal elements of the list:

- 'caret
- 'line
- 'selection
- 'user1
- 'user2

The presence of a flag in a character's value indicates that the character does not break a word when searching for breaks using the corresponding reason. For example, if 'caret is present, then the character is a non-breaking character for caret-movement words. (Each stream of non-breaking characters is a single word.)

(new editor-wordbreak-map%) → (is-a?/c editor-wordbreak-map%)

All ASCII alpha-numeric characters are initialized with '(caret line selection). All other ASCII non-whitespace characters except - are initialized with '(line). All ASCII whitespace characters and - are initialized with null.

(send *an-editor-wordbreak-map* get-map *char*)
 → (listof (one-of/c 'caret 'line 'selection 'user1 'user2))

```
  char : char?
```

Gets the mapping value for *char*. See editor-wordbreak-map% for more information.

```
(send an-editor-wordbreak-map set-map char
                                    value) → void?
  char : char?
  value : (listof (one-of/c 'caret 'line 'selection 'user1 'user2))
```

Sets the mapping value for *char* to *value*. See editor-wordbreak-map% for more information.

2.5.17 image-snip%

```
image-snip% : class?
  superclass: snip%
  extends: equal<%>
```

An image-snip% is a snip that can display bitmap images (usually loaded from a file). When the image file cannot be found, a box containing an "X" is drawn.

```
(make-object image-snip% [filename
                          kind
                          relative-path?
                          inline?])
 → (is-a?/c image-snip%)
  filename : (or/c path-string? false/c) = #f
  kind : (one-of/c 'unknown 'unknown/mask = 'unknown
                   'gif 'gif/mask
                   'jpeg 'png 'png/mask
                   'xbm 'xpm 'bmp 'pict)
  relative-path? : any/c = #f
  inline? : any/c = #t
(make-object image-snip% bitmap [mask]) → (is-a?/c image-snip%)
  bitmap : (is-a?/c bitmap%)
  mask : (or/c (is-a?/c bitmap%) false/c) = #f
```

Creates an image snip, loading the image *filename* if specified (see also load-file), or using the given *bitmap*.

```
(send an-image-snip equal-hash-code hash-code) → exact-integer?
  hash-code : (any/c . -> . exact-integer?)
```

Returns an integer that can be used as a `equal?`-based hash code for *an-image-snip* (using the same notion of `equal?` as `other-equal-to?`).

See also `equal<%>`.

```
(send an-image-snip equal-secondary-hash-code hash-code)
 → exact-integer?
  hash-code : (any/c . -> . exact-integer?)
```

Returns an integer that can be used as a `equal?`-based secondary hash code for *an-image-snip* (using the same notion of `equal?` as `other-equal-to?`).

See also `equal<%>`.

```
(send an-image-snip equal-to? snip equal?) → boolean?
  snip : (is-a?/c image-snip%)
  equal? : (any/c any/c . -> . boolean?)
```

Calls the `other-equal-to?` method of *snip* (to simulate multi-method dispatch) in case *snip* provides a more specific equivalence comparison.

See also `equal<%>`.

```
(send an-image-snip get-bitmap)
 → (or/c (is-a?/c bitmap%) false/c)
```

Returns the bitmap that is displayed by the snip, whether set through `set-bitmap` or `load-file`. If no bitmap is displayed, the result is `#f`.

The returned bitmap cannot be selected into a `bitmap-dc%` as long as it belongs to the snip, but it can be used as a pen or brush stipple.

```
(send an-image-snip get-bitmap-mask)
 → (or/c (is-a?/c bitmap%) false/c)
```

Returns the mask bitmap that is used for displaying by the snip, if one was installed with `set-bitmap`. If no mask is used, the result is `#f`.

The returned bitmap cannot be selected into a `bitmap-dc%` as long as it belongs to the snip, but it can be used as a pen or brush stipple.

```
(send an-image-snip get-filename [relative-path])
 → (or/c path-string? false/c)
```

```
  relative-path : (or/c (box/c any/c) false/c) = #f
```

Returns the name of the currently loaded, non-inlined file, or `#f` if a file is not loaded or if a file was loaded with inlining (the default).

The *relative-path* box is filled with `#t` if the loaded file's path is relative to the owning editor's path, unless *relative-path* is `#f`.

```
(send an-image-snip get-filetype)
 → (one-of/c 'unknown 'unknwon/mask
             'gif 'gif/mask
             'jpeg 'png 'png/mask 'xbm 'xpm 'bmp 'pict)
```

Returns the kind used to load the currently loaded, non-inlined file, or `'unknown` if a file is not loaded or if a file was loaded with inlining (the default).

```
(send an-image-snip load-file  filename
                               [kind
                                relative-path?
                                inline?])       → void?
  filename : (or/c path-string? false/c)
  kind : (one-of/c 'unknown 'unknown/mask = 'unknown
                   'gif 'gif/mask
                   'jpeg 'png 'png/mask
                   'xbm 'xpm 'bmp 'pict)
  relative-path? : any/c = #f
  inline? : any/c = #t
```

Loads the file by passing *filename* and *kind* to `load-file` If a bitmap had previously been specified with `set-bitmap`, that bitmap (and mask) will no longer be used. If *filename* is `#f`, then the current image is cleared.

When `'unknown/mask`, `'gif/mask`, or `'png/mask` is specified and the loaded bitmap object includes a mask (see `get-loaded-mask`), the mask is used for drawing the bitmap (see `draw-bitmap`).

If *relative-path?* is not `#f` and *filename* is a relative path, then the file will be read using the path of the owning editor's filename. If the image is not inlined, it will be saved as a relative pathname.

If *inline?* is not `#f`, the image data will be saved directly to the file or clipboard when the image is saved or copied (preserving the bitmap's mask, if any). The source filename and kind is no longer relevant.

```
(send an-image-snip other-equal-to? snip
                                    equal?) → boolean?
  snip : (is-a?/c image-snip%)
  equal? : (any/c any/c . -> . boolean?)
```

Returns #t if *an-image-snip* and *snip* both have bitmaps and the bitmaps are the same dimensions. If either has a mask bitmap with the same dimensions as the main bitmap, then the masks must be the same (or if only one mask is present, it must correspond to a solid mask).

The given *equal?* function (for recursive comparisons) is not used.

```
(send an-image-snip resize w h) → boolean?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Overrides resize in snip%.

The bitmap will be cropped to fit in the given dimensions.

```
(send an-image-snip set-bitmap bm [mask]) → void?
  bm : (is-a?/c bitmap%)
  mask : (or/c (is-a?/c bitmap%) false/c) = #f
```

Sets the bitmap that is displayed by the snip. This method also accepts an optional mask to be used when drawing the bitmap (see draw-bitmap), but supplying the mask directly is now deprecated. Instead, if no mask is supplied but the bitmap's get-loaded-mask method produces a bitmap of the same dimensions, it is used as the mask. Furthermore, such a mask is saved with the snip when it is saved to a file or copied (whereas a directly supplied mask is not saved).

The supplied bitmap must not be selected into a bitmap-dc% object, otherwise an exn:fail:contract exception is raised, and it cannot be selected into a bitmap-dc% as long as it belongs to the snip, but it can be used as a pen or brush stipple.

```
(send an-image-snip set-offset dx dy) → void?
  dx : real?
  dy : real?
```

Sets a graphical offset for the bitmap within the image snip.

2.5.18 keymap%

---

keymap% : class?
  superclass: object%

A keymap% object is used by editor<%> objects to map keyboard and mouse sequences to arbitrary functions in an extensible way. Keymaps can be used without editors, as well. A keymap% object contains

- a mapping from function names to event-handling procedures; and
- a mapping from key and mouse sequences to function names.

A handler procedure in a keymap is invoked with a key-event% object or a mouse-event% object. It is also given another value that depends on the context in which the keymap is used (or, more specifically, the arguments to handle-key-event or handle-mouse-event). For keymaps associated with editor<%> objects, the extra parameter is generally the editor<%> object that received the keyboard or mouse event.

(new keymap%) → (is-a?/c keymap%)

Creates an empty keymap.

(send a-keymap add-function name func) → void?
  name : string?
  func : (any/c (is-a?/c event%) . -> . any/c)

Names a new function to handle events, called in response to handle-key-event, handle-mouse-event, or call-function. The return value is of the procedure is ignored.

If there was already a function mapped to this name, it will be replaced with the given function.

When the function is called, it gets the arguments that were passed to handle-key-event, handle-mouse-event, or call-function. For keymaps associated with an editor, this is normally the target editor.

(send a-keymap break-sequence) → void?

Clears the state of the keymap if it is in the middle of a key sequence. For example, the user may have hit escape, and then changed to another window; if escape is part of a keyboard sequence, the keymap state needs to be cleared because the user is not going to complete the sequence.

A break callback function can be installed with set-break-sequence-callback.

```
(send a-keymap call-function name
                             in
                             event
                            [try-chain?]) → boolean?
  name : string?
  in : any/c
  event : (is-a?/c event%)
  try-chain? : any/c = #f
```

Calls a named event handler directly. If the function cannot be found or the found handler did not want to handle the event, #f is returned. Otherwise, the return value is the boolean return value of the event handler.

The *in* and *event* arguments are passed on to the keymap handler procedure if one is found.

If *try-chain?* is not #f, keymaps chained to this one are searched for the function name. If the function is not found and *try-chain?* is #f; an exception is also raised, but the exception handler cannot escape (see §1.1.4.4 "Exceptions and Continuation Jumps").

```
(send a-keymap chain-to-keymap next
                               prefix?) → void?
  next : (is-a?/c keymap%)
  prefix? : any/c
```

Chains *next* off *a-keymap* The *next* keymap will be used to handle events which are not handled by *a-keymap*. If *prefix?* is a true value, then *next* will take precedence over other keymaps already chained to *a-keymap*.

Multiple keymaps can be chained off one keymap using chain-to-keymap. When keymaps are chained off a main keymap, events not handled by the main keymap are passed to the chained keymaps until some chained keymap handles the events. Keymaps can be chained together in an arbitrary acyclic graph.

Keymap chaining is useful because multiple-event sequences are handled correctly for chained groups. Without chaining, a sequence of events can produce state in a keymap that must be reset when a callback is invoked in one of the keymaps. This state can be manually cleared with break-sequence, though calling the break-sequence method also invokes the handler installed by set-break-sequence-callback.

```
(send a-keymap get-double-click-interval)
 → (integer-in 0 1000000)
```

Returns the maximum number of milliseconds that can separate the clicks of a double-click.

The default interval is determined in a platform-specific way, but it can be overridden globally though the 'GRacket:doubleClickTime preference; see §3.1 “Preferences”.

```
(send a-keymap handle-key-event in event) → boolean?
  in : any/c
  event : (is-a?/c key-event%)
```

Attempts to handle a keyboard event, returning #t if the event was handled (i.e., a handler was found and it returned a true value), #f otherwise.

See also call-function.

```
(send a-keymap handle-mouse-event in event) → boolean?
  in : any/c
  event : (is-a?/c mouse-event%)
```

Attempts to handle a mouse event, returning #t if the event was handled (i.e., a handler was found and it returned a true value), #f otherwise.

See also call-function.

```
(send a-keymap map-function keyname fname) → void?
  keyname : string?
  fname : string?
```

Maps an input state sequence to a function name using a string-encoded sequence in *keyname*. The format of *keyname* is a sequence of semicolon-delimited input states; each state is made up of a sequence of modifier identifiers followed by a key identifier.

The modifier identifiers are:

- s: — All platforms: Shift
- c: — All platforms: Control
- a: — Mac OS X: Option
- m: — Windows: Alt; X: Meta; Mac OS X: Command, when map-command-as-meta-key produces #t
- d: — Mac OS X: Command
- l: — All platforms: Caps Lock
- ?: — All platforms: allow match to character produced by opposite use of Shift, AltGr/Option, and/or Caps Lock, when available; see get-other-shift-key-code in key-event%

If a particular modifier is not mentioned in a state string, it matches states whether that modifier is pressed or not pressed. A ~ preceding a modifier makes the string match only states where the corresponding modifier is not pressed. If the state string begins with `:`, then the string matches a state only if modifiers (other than Caps Lock) not mentioned in the string are not pressed.

A key identifier can be either a character on the keyboard (e.g., `a`, `2`, `?`) or a special name. The special names are as follows:

- `leftbutton` (button down)
- `rightbutton`
- `middlebutton`
- `leftbuttondouble` (button down for double-click)
- `rightbuttondouble`
- `middlebuttondouble`
- `leftbuttontriple` (button down for triple-click)
- `rightbuttontriple`
- `middlebuttontriple`
- `leftbuttonseq` (all events from button down through button up)
- `rightbuttonseq`
- `middlebuttonseq`
- `wheelup`
- `wheeldown`
- `esc`
- `delete`
- `del` (same as `delete`)
- `insert`
- `ins` (same as `insert`)
- `add`
- `subtract`
- `multiply`

- `divide`
- `backspace`
- `back`
- `return`
- `enter` (same as `return`)
- `tab`
- `space`
- `right`
- `left`
- `up`
- `down`
- `home`
- `end`
- `pageup`
- `pagedown`
- `semicolon` (since `;` separates sequence steps)
- `colon` (since `:` separates modifiers)
- `numpad0`
- `numpad1`
- `numpad2`
- `numpad3`
- `numpad4`
- `numpad5`
- `numpad6`
- `numpad7`
- `numpad8`
- `numpad9`
- `numpadenter`

- `f1`
- `f2`
- `f3`
- `f4`
- `f5`
- `f6`
- `f7`
- `f8`
- `f9`
- `f10`
- `f11`
- `f12`
- `f13`
- `f14`
- `f15`
- `f16`
- `f17`
- `f18`
- `f19`
- `f20`
- `f21`
- `f22`
- `f23`
- `f24`

For a special keyword, the capitalization does not matter. However, capitalization is important for single-letter keynames. Furthermore, single-letter ASCII keynames are treated specially: `A` and `s:a` are both treated as `s:A`. However, when `c:` is included under Windows without `m:`, or when `d:` is included under Mac OS X, then ASCII letters are not upcased

with `s:`, since the upcasing behavior of the Shift key is cancelled by Control without Alt (under Windows) or by Command (under Mac OS X).

A state can match multiple state strings mapped in a keymap (or keymap chain); when a state matches multiple state strings, a mapping is selected by ranking the strings according to specificity. A state string that mentions more pressed modifiers ranks higher than other state strings, and if two strings mention the same number of pressed modifiers, the one that mentions more unpressed modifiers ranks higher. Finally, a state string that includes `?:` and matches only with the opposite use of Shift, AltGr/Option, and/or Caps Lock ranks below all matches that do not depend on `?:`, and one that requires the opposite use of both Shift and AltGr/Option ranks even lower. In the case that multiple matching strings have the same rank, a match is selected arbitrarily.

Examples:

- `"space"` — matches whenever the space bar is pressed, regardless of the state of modifiers keys.
- `"~c:space"` — matches whenever the space bar is pressed and the Control key is not pressed.
- `"a"` — matches whenever `a` is typed, regardless of the state of modifiers keys (other than Shift).
- `":a"` — matches only when `a` is typed with no modifier keys pressed.
- `"~c:a"` — matches whenever `a` is typed and neither the Shift key nor the Control key is pressed.
- `":esc;:c:c"` — matches an Escape key press (no modifiers) followed by a Control-C press (no modifiers other than Control).
- `"?:d:+"` — matches when Command is pressed with key that produces `+`, even if producing `+` normally requires pressing Shift.

A call to `map-function` that would map a particular key sequence both as a prefix and as a complete sequence raises an exception, but the exception handler cannot escape (see §1.1.4.4 “Exceptions and Continuation Jumps”).

A function name does not have to be mapped to a handler before input states are mapped to the name; the handler is dispatched by name at the time of invocation. The event handler mapped to a function name can be changed without affecting the map from input states to function names.

```
(send a-keymap remove-chained-keymap keymap) → void?
  keymap : (is-a?/c keymap%)
```

If *keymap* was previously chained from this keymap (through chain-to-keymap), then it is removed from the chain-to list.

```
(send a-keymap remove-grab-key-function) → void?
```

Removes a callback installed with set-grab-key-function.

```
(send a-keymap remove-grab-mouse-function) → void?
```

Removes a callback installed with set-grab-mouse-function.

```
(send a-keymap set-break-sequence-callback f) → void?
  f : (-> any)
```

Installs a callback procedure that is invoked when break-sequence is called. After it is invoked once, the callback is removed from the keymap. If another callback is installed before break-sequence is called, the old callback is invoked immediately before the new one is installed.

```
(send a-keymap set-double-click-interval n) → void?
  n : (integer-in 0 1000000)
```

Sets the maximum number of milliseconds that can separate the clicks of a double-click.

```
(send a-keymap set-grab-key-function f) → void?
  f : ((or/c string? false?)
       (is-a?/c keymap%)
       any/c
       (is-a?/c key-event%)
       . -> . any)
```

Installs a callback procedure that is invoked after the keymap matches input to a function name or fails to match an input. Only one keyboard grab function can be installed at a time. When keymaps are chained to a keymap with a grab callback, the callback is invoked for matches in the chained keymap (when the chained keymap does not have its own grab callback).

If a grab callback returns a true value for a matching or non-matching callback, the event is considered handled. If the callback returns a true value for a matching callback, then the matching keymap function is not called by the keymap.

The callback procedure *f* will be invoked as:

```
(f str keymap editor event)
```

The *str* argument is the name of a function for a matching callback, or #f for a non-matching callback. The *keymap* argument is the keymap that matched (possibly a keymap chained to the one in which the callback was installed) or the keymap in which the callback was installed. The *editor* and *event* arguments are the same as passed on to the matching keymap function.

Key grab callback functions are de-installed with remove-grab-key-function.

```
(send a-keymap set-grab-mouse-function f) → void?
  f : ((or/c string? false?)
       (is-a?/c keymap%)
       any/c
       (is-a?/c mouse-event%)
       . -> . any)
```

Like set-grab-key-function, but for mouse events.

2.5.19 mult-color<%>

---

mult-color<%> : interface?

A mult-color<%> object is used to scale the RGB values of a color% object. A mult-color<%> object exist only within a style-delta% object.

See also get-foreground-mult and get-background-mult.

```
(send a-mult-color get r g b) → void?
  r : (box/c real?)
  g : (box/c real?)
  b : (box/c real?)
```

Gets all of the scaling values.

The *r* box is filled with the scaling value for the red component of the color. The *g* box is filled with the scaling value for the green component of the color. The *b* box is filled with the scaling value for the blue component of the color.

```
(send a-mult-color get-b) → real?
```

Gets the multiplicative scaling value for the blue component of the color.

```
(send a-mult-color get-g) → real?
```

Gets the multiplicative scaling value for the green component of the color.

```
(send a-mult-color get-r) → real?
```

Gets the multiplicative scaling value for the red component of the color.

```
(send a-mult-color set r g b) → void?
  r : real?
  g : real?
  b : real?
```

Sets all of the scaling values.

```
(send a-mult-color set-b v) → void?
  v : real?
```

Sets the multiplicative scaling value for the blue component of the color.

```
(send a-mult-color set-g v) → void?
  v : real?
```

Sets the multiplicative scaling value for the green component of the color.

```
(send a-mult-color set-r v) → void?
  v : real?
```

Sets the additive value for the red component of the color.

2.5.20 pasteboard%

```
pasteboard% : class?
  superclass: object%
  extends: editor<%>
```

A pasteboard% object is an editor for displaying snips with arbitrary locations.

```
(new pasteboard%) → (is-a?/c pasteboard%)
```

The editor will not be displayed until it is attached to an editor-canvas% object or some other display.

A new keymap% object is created for the new editor. See also get-keymap and set-keymap.

A new style-list% object is created for the new editor. See also get-style-list and set-style-list.

```
(send a-pasteboard add-selected snip) → void?
  snip : (is-a?/c snip%)
(send a-pasteboard add-selected x y w h) → void?
  x : real?
  y : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Selects snips without deselecting other snips. When coordinates are given, this method selects all snips that intersect with the given rectangle (in editor coordinates).

The selection in a pasteboard can be changed by the system in response to other method calls, and such changes do not go through this method; use on-select to monitor selection changes.

```
(send a-pasteboard after-delete snip) → void?
  snip : (is-a?/c snip%)
```

Refine this method with augment.

Specification: Called after a snip is deleted from the editor (and after the display is refreshed; use on-delete and begin-edit-sequence to avoid extra refreshes when after-delete modifies the editor).

See also can-delete? and on-edit-sequence.

No internals locks are set when this method is called.

Default implementation: Does nothing.

```
(send a-pasteboard after-insert snip
                                before
                                x
                                y)      → void?
  snip : (is-a?/c snip%)
  before : (or/c (is-a?/c snip%) false/c)
  x : real?
  y : real?
```

Refine this method with augment.

Specification: Called after a snip is inserted into the editor (and after the display is refreshed; use `on-insert` and `begin-edit-sequence` to avoid extra refreshes when `after-insert` modifies the editor).

See also `can-insert?` and `on-edit-sequence`.

No internals locks are set when this method is called.

Default implementation: Does nothing.

```
(send a-pasteboard after-interactive-move event) → void?
  event : (is-a?/c mouse-event%)
```

Refine this method with `augment`.

Specification: Called after the user stops interactively dragging snips (the ones that are selected; see `find-next-selected-snip`). The mouse event that terminated the move (usually a button-up event) is provided.

See also `can-interactive-move?` and `on-interactive-move`.

Default implementation: Does nothing.

```
(send a-pasteboard after-interactive-resize snip) → void?
  snip : (is-a?/c snip%)
```

Refine this method with `augment`.

Specification: Called after the user stops interactively resizing a snip (the one that is currently selected; see `find-next-selected-snip`). The *snip* argument is the snip that was resized.

See also `can-interactive-resize?` and `on-interactive-resize`.

Default implementation: Does nothing.

```
(send a-pasteboard after-move-to snip
                                 x
                                 y
                                 dragging?) → void?
  snip : (is-a?/c snip%)
  x : real?
  y : real?
  dragging? : any/c
```

Refine this method with `augment`.

Specification: Called after a given snip is moved within the editor (and after the display is refreshed; use `on-move-to` and `begin-edit-sequence` to avoid extra refreshes when `after-move-to` modifies the editor).

If *`dragging?`* is not `#f`, then this move was a temporary move for dragging.

See also `can-move-to?` and `on-edit-sequence`.

No internals locks are set when this method is called.

Default implementation: Does nothing.

```
(send a-pasteboard after-reorder snip
                                 to-snip
                                 before?) → boolean?
  snip : (is-a?/c snip%)
  to-snip : (is-a?/c snip%)
  before? : any/c
```

Refine this method with `augment`.

Specification: Called before a snip is moved in the pasteboard's front-to-back snip order (and after the display is refreshed; use `on-reorder` and `begin-edit-sequence` to avoid extra refreshes when `after-reorder` modifies the editor).

If *`before?`* is `#t`, then *`snip`* was moved before *`to-snip`*, otherwise *`snip`* was moved after *`to-snip`*.

See also `can-reorder?` and `on-edit-sequence`.

No internals locks are set when this method is called.

Default implementation: Does nothing.

```
(send a-pasteboard after-resize snip
                                w
                                h
                                resized?) → void?
  snip : (is-a?/c snip%)
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
  resized? : any/c
```

Refine this method with `augment`.

Specification: Called after a given snip is resized (and after the display is refreshed; use on-resize and begin-edit-sequence to avoid extra refreshes when after-resize modifies the editor), or after an unsuccessful resize attempt was made.

If *resized?* is not #f, the snip was successfully resized.

See also can-resize? and on-edit-sequence.

No internals locks are set when this method is called.

Default implementation: Does nothing.

```
(send a-pasteboard after-select snip on?) → void?
  snip : (is-a?/c snip%)
  on? : any/c
```

Refine this method with augment.

Specification: Called after a snip in the pasteboard is selected or deselected. See also on-select. This method is not called after selected snip is deleted (and thus de-selected indirectly); see also after-delete.

If *on?* is #t, then *snip* was just selected, otherwise *snip* was just deselected.

See also can-select? and on-edit-sequence.

No internals locks are set when this method is called.

Default implementation: Does nothing.

```
(send a-pasteboard can-delete? snip) → boolean?
  snip : (is-a?/c snip%)
```

Refine this method with augment.

Specification: Called before a snip is deleted from the editor. If the return value is #f, then the delete will be aborted.

See also on-delete and after-delete.

The editor is internally locked for writing when this method is called (see also §1.3.8 “Internal Editor Locks”).

Default implementation: Returns #t.

```
(send a-pasteboard can-insert? snip
                               before
                               x
                               y)      → boolean?
  snip : (is-a?/c snip%)
  before : (or/c (is-a?/c snip%) false/c)
  x : real?
  y : real?
```

Refine this method with `augment`.

Specification: Called before a snip is inserted from the editor. If the return value is `#f`, then the insert will be aborted.

See also `on-insert` and `after-insert`.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks").

Default implementation: Returns `#t`.

```
(send a-pasteboard can-interactive-move? event) → boolean?
  event : (is-a?/c mouse-event%)
```

Refine this method with `augment`.

Specification: Called when the user starts interactively dragging snips (the ones that are selected; see `find-next-selected-snip`). All of the selected snips will be moved. If `#f` is returned, the interactive move is disallowed. The mouse event that started the move (usually a button-down event) is provided.

See also `on-interactive-move`, `after-interactive-move`, and `interactive-adjust-move`.

Default implementation: Returns `#t`.

```
(send a-pasteboard can-interactive-resize? snip) → boolean?
  snip : (is-a?/c snip%)
```

Refine this method with `augment`.

Specification: Called when the user starts interactively resizing a snip (the one that is selected; see `find-next-selected-snip`). If `#f` is returned, the interactive resize is disallowed.

The *snip* argument is the snip that will be resized.

See also after-interactive-resize, after-interactive-resize, and interactive-adjust-resize.

Default implementation: Returns #t.

```
(send a-pasteboard can-move-to? snip
                                x
                                y
                                dragging?) → boolean?
  snip : (is-a?/c snip%)
  x : real?
  y : real?
  dragging? : any/c
```

Refine this method with augment.

Specification: Called before a snip is moved in the editor. If the return value is #f, then the move will be aborted.

If *dragging?* is not #f, then this move is a temporary move for dragging.

See also on-move-to and after-move-to.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks").

Default implementation: Returns #t.

```
(send a-pasteboard can-reorder? snip
                                to-snip
                                before?) → boolean?
  snip : (is-a?/c snip%)
  to-snip : (is-a?/c snip%)
  before? : any/c
```

Refine this method with augment.

Specification: Called before a snip is moved in the pasteboard's front-to-back snip order. If the return value is #f, then the reordering will be aborted.

If *before?* is #t, then *snip* is to be moved before *to-snip*, otherwise *snip* is to be moved after *to-snip*.

See also on-reorder and after-reorder.

The editor is internally locked for writing when this method is called (see also §1.3.8 “Internal Editor Locks”).

Default implementation: Returns `#t`.

```
(send a-pasteboard can-resize? snip w h) → boolean?
  snip : (is-a?/c snip%)
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Refine this method with `augment`.

Specification: Called before a snip is resized in the editor. If the return value is `#f`, then the resize will be aborted.

See also `on-resize` and `after-resize`.

The editor is internally locked for writing when this method is called (see also §1.3.8 “Internal Editor Locks”).

Default implementation: Returns `#t`.

```
(send a-pasteboard can-select? snip on?) → boolean?
  snip : (is-a?/c snip%)
  on? : any/c
```

Refine this method with `augment`.

Specification: This method is called before a snip in the pasteboard is selected or deselected. If `#f` is returned, the selection change is disallowed. This method is not called when a selected snip is to be deleted (and thus de-selected indirectly); see also `can-delete?`.

If *on?* is `#t`, then *snip* will be selected, otherwise *snip* will be deselected.

See also `on-select` and `after-select`.

The editor is internally locked for writing when this method is called (see also §1.3.8 “Internal Editor Locks”).

Default implementation: Returns `#t`.

```
(send a-pasteboard change-style [style snip]) → void?
  style : (or/c (is-a?/c style-delta%) (is-a?/c style<%>) #f)
        = #f
  snip : (or/c (is-a?/c snip%) #f) = #f
```

Overrides change-style in editor<%>.

Changes the style of *snip* to a specific style or by applying a style delta. If *snip* is #f, then all currently selected snips are changed. If *style* is #f, then the default style is used, according to default-style-name.

See also change-style in editor<%>.

When a *style* is provided: The editor's style list must contain *style*, otherwise the style is not changed. See also convert in style-list%.

```
(send a-pasteboard copy-self-to dest) → void?
  dest : (or/c (is-a?/c text%) (is-a?/c pasteboard%))
```

Overrides copy-self-to in editor<%>.

In addition to the default copy-self-to in editor<%> work, the dragability, selection visibility state, and scroll step of *a-pasteboard* are installed into *dest*.

```
(send a-pasteboard delete) → void?
(send a-pasteboard delete snip) → void?
  snip : (is-a?/c snip%)
```

Deletes *snip* when provided, or deletes the currently selected snips from the editor when *snip* is not provided.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use on-delete to monitor content deletion changes.

```
(send a-pasteboard do-copy time extend?) → void?
  time : exact-integer?
  extend? : any/c
```

Overrides do-copy in editor<%>.

Specification: Called to copy the editor's current selection into the clipboard. This method is provided so that it can be overridden by subclasses. Do not call this method directly; instead, call copy.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

Default implementation: Copies the current selection, extending the current clipboard contexts if *extend?* is true.

```
(send a-pasteboard do-paste time) → void?
  time : exact-integer?
```

Overrides do-paste in editor<%>.

Specification: Called to paste the current contents of the clipboard into the editor. This method is provided so that it can be overridden by subclasses. Do not call this method directly; instead, call paste.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

Default implementation: Pastes.

```
(send a-pasteboard do-paste-x-selection time) → void?
  time : exact-integer?
```

Overrides do-paste-x-selection in editor<%>.

Specification: Called to paste the current contents of the X selection under X (or the clipboard under Windows and Mac OS X) into the editor. This method is provided so that it can be overridden by subclasses. Do not call this method directly; instead, call paste-x-selection.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

Default implementation: Pastes.

```
(send a-pasteboard erase) → void?
```

Deletes all snips from the editor.

See also delete.

```
(send a-pasteboard find-next-selected-snip start)
 → (or/c (is-a?/c snip%) false/c)
  start : (or/c (is-a?/c snip%) false/c)
```

Returns the next selected snip in the editor, starting the search after *start*. (See §1.3.1 "Editor Structure and Terminology" for information about snip order in pasteboards.) If *start* is #f, then the search starts with the first snip in the editor (and thus returns the first selected snip, if any are selected). If no more selected snips are available, or if *start* is not in the pasteboard, #f is returned.

```
(send a-pasteboard find-snip x y [after])
 → (or/c (is-a?/c snip%) false/c)
  x : real?
  y : real?
  after : (or/c (is-a?/c snip%) false/c) = #f
```

Finds the frontmost snip (after a given snip) that intersects a given location. See §1.3.1 “Editor Structure and Terminology” for information about snip order in pasteboards.

The *x* and *y* arguments are in editor coordinates. If *after* is not supplied, the frontmost snip at *x* and *y* is returned, otherwise the frontmost snip behind *after* is returned. If *after* is a snip that is not in the pasteboard, `#f` is returned.

The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”).

```
(send a-pasteboard get-center) → real? real?
```

Returns the center of the pasteboard in pasteboard coordinates.

The first result is the x-coordinate of the center and the second result is the y-coordinate of the center.

```
(send a-pasteboard get-dragable) → boolean?
```

Returns whether snips in the editor can be interactively dragged by event handling in `on-default-event`: `#t` if dragging is allowed, `#f` otherwise. By default, dragging is allowed. See also `set-dragable`.

```
(send a-pasteboard get-scroll-step)
 → (and/c real? (not/c negative?))
```

Gets the editor location offset for each vertical scroll position. See also `set-scroll-step`.

```
(send a-pasteboard get-selection-visible) → boolean?
```

Returns whether selection dots are drawn around the edge of selected snips in the pasteboard. By default, selection dots are on. See also `set-selection-visible`.

```
(send a-pasteboard insert snip) → void?
  snip : (is-a?/c snip%)
(send a-pasteboard insert snip before x y) → void?
```

```
  snip : (is-a?/c snip%)
  before : (or/c (is-a?/c snip%) false/c)
  x : real?
  y : real?
(send a-pasteboard insert snip x y) → void?
  snip : (is-a?/c snip%)
  x : real?
  y : real?
(send a-pasteboard insert snip before) → void?
  snip : (is-a?/c snip%)
  before : (or/c (is-a?/c snip%) false/c)
```

Extends `insert` in `editor<%>`.

Inserts *snip* at location (*x*, *y*) just in front of *before*. (See §1.3.1 "Editor Structure and Terminology" for information about snip order in pasteboards.) If *before* is not provided or is `#f`, then *snip* is inserted behind all other snips. If *x* and *y* are not provided, the snip is added at (0, 0).

```
(send a-pasteboard interactive-adjust-mouse x
                                            y) → void?
  x : (box/c real?)
  y : (box/c real?)
```

Specification: This method is called during interactive dragging and resizing (of the currently selected snips; see `find-next-selected-snip`) to preprocess the current mouse location (in editor coordinates). The snip and actual x and y coordinates are passed into the method (boxed); the resulting coordinates are used instead of the actual mouse location.

See also `interactive-adjust-resize`.

Default implementation: A negative value for either *x* or *y* is replaced with `0`.

```
(send a-pasteboard interactive-adjust-move snip
                                           x
                                           y)   → void?
  snip : (is-a?/c snip%)
  x : (box/c real?)
  y : (box/c real?)
```

Specification: This method is called during an interactive move (for each selected snip) to preprocess the user-determined snip location for each selected snip. The snip and mouse-determined locations (in editor coordinates) are passed into the method (boxed); the resulting locations are used for graphical feedback to the user during moving.

The actual mouse coordinates are first sent through `interactive-adjust-mouse` before determining the locations passed into this method.

Default implementation: Does nothing.

```
(send a-pasteboard interactive-adjust-resize snip
                                             width
                                             height) → void?
  snip : (is-a?/c snip%)
  width : (box/c (and/c real? (not/c negative?)))
  height : (box/c (and/c real? (not/c negative?)))
```

Specification: This method is called during interactive resizing of a snip to preprocess the user-determined snip size. The snip and mouse-determined height and width are passed into the method (boxed); the resulting height and width are used for graphical feedback to the user during resizing.

The actual mouse coordinates are first sent through `interactive-adjust-mouse` before determining the sizes passed into this method.

Default implementation: Does nothing.

```
(send a-pasteboard is-selected? snip) → boolean?
  snip : (is-a?/c snip%)
```

Returns `#t` if a specified snip is currently selected or `#f` otherwise.

```
(send a-pasteboard lower snip) → void?
  snip : (is-a?/c snip%)
```

Moves the snip one level deeper (i.e., behind one more other snip) in the pasteboard's snip order. See §1.3.1 "Editor Structure and Terminology" for information about snip order in pasteboards.

See also `raise`, `set-before`, and `set-after`.

```
(send a-pasteboard move snip x y) → void?
  snip : (is-a?/c snip%)
  x : real?
  y : real?
(send a-pasteboard move x y) → void?
  x : real?
  y : real?
```

Moves *snip* right *x* pixels and down *y* pixels. If *snip* is not provided, then all selected snips are moved.

Snip locations in a pasteboard can be changed by the system in response to other method calls, and such changes do not go through this method; use on-move-to to monitor snip position changes.

```
(send a-pasteboard move-to snip x y) → void?
  snip : (is-a?/c snip%)
  x : real?
  y : real?
```

Moves *snip* to a given location in the editor.

Snip locations in a pasteboard can be changed by the system in response to other method calls, and such changes do not go through this method; use on-move-to to monitor snip position changes.

```
(send a-pasteboard no-selected) → void?
```

Deselects all selected snips in the editor.

The selection in a pasteboard can be changed by the system in response to other method calls, and such changes do not go through this method; use on-select to monitor selection changes.

```
(send a-pasteboard on-default-event event) → void?
  event : (is-a?/c mouse-event%)
```

Overrides on-default-event in editor<%>.

Selects, drags, and resizes snips:

- Clicking on a snip selects the snip. Shift-clicking extends the current selection with the snip.
- Clicking in the space between snips drags a selection box; once the mouse button is released, all snips touching the box are selected. Shift-clicking extends the current selection with the new snips.
- Double-clicking on a snip calls on-double-click.
- Clicking on a selected snip drags the selected snip(s) to a new location.
- Clicking on a hiliting tab for a selected object resizes the object.

```
(send a-pasteboard on-delete snip) → void?
  snip : (is-a?/c snip%)
```

Refine this method with `augment`.

Called before a snip is deleted from the editor, after `can-delete?` is called to verify that the deletion is allowed. The `after-delete` method is guaranteed to be called after the delete has completed.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks"). Use `after-delete` to modify the editor, if necessary.

```
(send a-pasteboard on-double-click snip
                                   event) → void?
  snip : (is-a?/c snip%)
  event : (is-a?/c mouse-event%)
```

Specification: This method is called when the user double-clicks on a snip in the editor. The clicked-on snip and event records are passed to the method.

Default implementation: If *snip* accepts events, it is designated as the caret owner and all snips in the editor are unselected.

```
(send a-pasteboard on-insert snip before x y) → void?
  snip : (is-a?/c snip%)
  before : (or/c (is-a?/c snip%) false/c)
  x : real?
  y : real?
```

Refine this method with `augment`.

Called before a snip is inserted from the editor, after `can-insert?` is called to verify that the insertion is allowed. The `after-insert` method is guaranteed to be called after the insert has completed.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks"). Use `after-insert` to modify the editor, if necessary.

```
(send a-pasteboard on-interactive-move event) → void?
  event : (is-a?/c mouse-event%)
```

Refine this method with `augment`.

Specification: Called when the user starts interactively dragging snips (the ones that are selected; see `find-next-selected-snip`), after `can-interactive-move?` is called to verify that the move is allowed. The `after-interactive-move` method is guaranteed to be called after the move has completed. All of the selected snips will be moved. The mouse event that started the move (usually a button-down event) is provided.

See also `interactive-adjust-move`.

Default implementation: Does nothing.

```
(send a-pasteboard on-interactive-resize snip) → void?
  snip : (is-a?/c snip%)
```

Refine this method with `augment`.

Specification: Called when the user starts interactively resizing a snip (the one that is selected; see `find-next-selected-snip`), after `can-interactive-resize?` is called to verify that the resize is allowed. The `after-interactive-resize` method is guaranteed to be called after the resize has completed.

The *snip* argument is the snip that will be resized.

Default implementation: Does nothing.

```
(send a-pasteboard on-move-to snip
                              x
                              y
                              dragging?) → void?
  snip : (is-a?/c snip%)
  x : real?
  y : real?
  dragging? : any/c
```

Refine this method with `augment`.

Specification: Called before a snip is moved in the editor, after `can-move-to?` is called to verify that the move is allowed. The `after-move-to` method is guaranteed to be called after the move has completed.

If *dragging?* is not `#f`, then this move is a temporary move for dragging.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks"). Use `after-move-to` to modify the editor, if necessary. See also `on-interactive-move` and `interactive-adjust-move`.

Default implementation: Does nothing.

```
(send a-pasteboard on-reorder snip
                              to-snip
                              before?) → void?
  snip : (is-a?/c snip%)
  to-snip : (is-a?/c snip%)
  before? : any/c
```

Refine this method with `augment`.

Specification: Called before a snip is moved in the pasteboard's front-to-back snip order, after `can-reorder?` is called to verify that the reorder is allowed. The `after-reorder` method is guaranteed to be called after the reorder has completed.

If *before?* is `#t`, then *snip* is to be moved before *to-snip*, otherwise *snip* is to be moved after *to-snip*.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks"). Use `after-reorder` to modify the editor, if necessary.

Default implementation: Does nothing.

```
(send a-pasteboard on-resize snip w h) → void?
  snip : (is-a?/c snip%)
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Refine this method with `augment`.

Specification: Called before a snip is resized by the editor, after `can-resize?` is called to verify that the resize is allowed. The `after-resize` method is guaranteed to be called after the resize has completed.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks"). Use `after-resize` to modify the editor, if necessary.

Note that a snip calls `resized`, not this method, to notify the pasteboard that the snip resized itself.

Default implementation: Does nothing.

```
(send a-pasteboard on-select snip on?) → void?
  snip : (is-a?/c snip%)
  on? : any/c
```

Refine this method with `augment`.

Specification: Called before a snip in the pasteboard is selected or deselected, after `can-select?` is called to verify that the selection is allowed. The `after-select` method is guaranteed to be called after the selection has completed. This method is not called when a selected snip is to be deleted (and thus de-selected indirectly); see also `on-delete` .

If *on?* is `#t`, then *snip* will be selected, otherwise *snip* will be deselected.

The editor is internally locked for writing when this method is called (see also §1.3.8 "Internal Editor Locks"). Use `after-select` to modify the editor, if necessary.

Default implementation: Does nothing.

```
(send a-pasteboard raise snip) → void?
  snip : (is-a?/c snip%)
```

Moves a snip one level shallower (i.e., in front of one more other snip) in the pasteboard's snip order. See §1.3.1 "Editor Structure and Terminology" for information about snip order in pasteboards.

See also `lower`, `set-before`, and `set-after`.

```
(send a-pasteboard remove snip) → void?
  snip : (is-a?/c snip%)
```

Removes the specified snip from the editor in a non-undoable manner (so the snip is completely free of the pasteboard can be used in other editors).

See also `delete`.

```
(send a-pasteboard remove-selected snip) → void?
  snip : (is-a?/c snip%)
```

Deselects *snip* (if it is currently selected) without deselecting any other snips.

The selection in a pasteboard can be changed by the system in response to other method calls, and such changes do not go through this method; use `on-select` to monitor selection changes.

```
(send a-pasteboard resize snip w h) → boolean?
  snip : (is-a?/c snip%)
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Attempts to resize a given snip. If the snip allows resizing, `#t` is returned, otherwise `#f` is returned. Using this method instead of calling the snip's `resize` method directly will make the resize undo-able.

```
(send a-pasteboard set-after snip after) → void?
  snip : (is-a?/c snip%)
  after : (or/c (is-a?/c snip%) false/c)
```

Changes the depth of *snip* moving it just behind *after*. If *after* is `#f`, *snip* is moved to the back. See §1.3.1 "Editor Structure and Terminology" for information about snip order in pasteboards.

See also `raise`, `lower`, and `set-before`.

```
(send a-pasteboard set-before snip before) → void?
  snip : (is-a?/c snip%)
  before : (or/c (is-a?/c snip%) false/c)
```

Changes the depth of *snip* moving it just in front of *before*. If *before* is `#f`, *snip* is moved to the front. See §1.3.1 "Editor Structure and Terminology" for information about snip order in pasteboards.

See also `raise`, `lower`, and `set-after`.

```
(send a-pasteboard set-dragable allow-drag?) → void?
  allow-drag? : any/c
```

Sets whether snips in the editor can be interactively dragged by event handling in `on-default-event`: a true value allows dragging, `#f` disallows dragging. See also `get-dragable`.

```
(send a-pasteboard set-scroll-step stepsize) → void?
  stepsize : (and/c real? (not/c negative?))
```

Sets the editor location offset for each vertical scroll position. See also `get-scroll-step`.

```
(send a-pasteboard set-selected snip) → void?
  snip : (is-a?/c snip%)
```

Selects a specified snip (deselecting all others).

The selection in a pasteboard can be changed by the system in response to other method calls, and such changes do not go through this method; use `on-select` to monitor selection changes.

```
(send a-pasteboard set-selection-visible visible?) → void?
  visible? : any/c
```

Sets whether selection dots are drawn around the edge of selected snips in the pasteboard. See also get-selection-visible.

2.5.21 readable-snip<%>

---

readable-snip<%> : interface?

A readable-snip<%> object is treated specially by the port generated by open-input-text-editor: When a readable-snip<%> object is encountered for the input stream, its read-special method is called to generate the read result for the snip, which is returned from the port as a "special" value in the sense of read-char-or-special.

Since read and read-syntax build on read-char-or-special, a snip can implement readable-snip<%> so that it produces a whole S-expression or some other kind of value when read is used on a stream containing the snip.

```
(send a-readable-snip read-special source
                                  line
                                  column
                                  position) → any/c
  source : any/c
  line : (or/c exact-nonnegative-integer? false/c)
  column : (or/c exact-nonnegative-integer? false/c)
  position : (or/c exact-nonnegative-integer? false/c)
```

The arguments are the same as the arguments to a procedure returned by a custom input port's *read-in*; see §12.1.9 "Custom Ports" for details. The result is also the same as the result from a *read-in*-produced procedure.

2.5.22 snip%

---

snip% : class?
  superclass: object%

A direct instance of snip% is uninteresting. Useful snips are defined by instantiating derived subclasses, but this class defines the basic functionality.

In deriving a new snip class, these methods must be overridden to create a useful snip:

- `get-extent`
- `draw`
- `resize` if the snip can be resized by the user
- `partial-offset` if the snip can contain more than one item
- `split` if the snip can contain more than one item
- `size-cache-invalid` if the snip caches the result to`get-extent`
- `get-text` (not required)
- `find-scroll-step`, `get-num-scroll-steps`, and `get-scroll-step-offset` if the snip can contain more than one scroll position
- `set-unmodified` if the snip's internal state can be modified by the user, and call `modified` in the snip's administrator when the state changes the first time

If a snip can contain more than one item, then the snip's count must be maintained as well.

To define a class of snips that can be saved or cut-and-pasted:

- Create an instance of `snip-class%`, implementing the `read` method. Export the `snip-class%` instance as `snip-class` from a module, and use a classname of the form `"(lib ...)"` as described in §1.3.2.1.1 "Snip Classes".
- For each instance of the snip class, set the snip's class object with `set-snipclass`.
- Override the `copy` method.
- Override the `write` method.

To define a class of snips that read specially with `open-input-text-editor`:

- Make your `snip%` class implement `readable-snip<%>`.
- Implement the `read-special` method.

`(new snip%)` → `(is-a?/c snip%)`

Creates a plain snip of length 1 with the `"Basic"` style of `the-style-list`.

```
(send a-snip adjust-cursor dc
                           x
                           y
                           editorx
                           editory
                           event)
 → (or/c (is-a?/c cursor%) false/c)
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  editorx : real?
  editory : real?
  event : (is-a?/c mouse-event%)
```

Specification: Called to determine the cursor image used when the cursor is moved over the snip in an editor. If `#f` is returned, a default cursor is selected by the editor. (See `adjust-cursor` in `editor<%>` for more information.)

Default implementation: Returns `#f`.

```
(send a-snip blink-caret dc x y) → void?
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
```

Tells the snip to blink the selection caret. This method is called periodically when the snip's editor's display has the keyboard focus, and the snip has the editor-local focus.

The drawing context and snip's locations in drawing context coordinates are provided.

```
(send a-snip can-do-edit-operation? op
                                    [recursive?]) → boolean?
  op : (one-of/c 'undo 'redo 'clear 'cut 'copy
                 'paste 'kill 'select-all
                 'insert-text-box 'insert-pasteboard-box
                 'insert-image)
  recursive? : any/c = #t
```

See `can-do-edit-operation?` in `editor<%>`.

Called when the snip's editor's method is called, *recursive?* is not `#f`, and this snip owns the caret.

```
(send a-snip copy) → (is-a?/c snip%)
```

Creates and returns a copy of this snip. The copy method is responsible for copying this snip's style (as returned by get-style) to the new snip.

```
(send a-snip do-edit-operation  op
                                [recursive?
                                 time])     → void?
  op : (one-of/c 'undo 'redo 'clear 'cut 'copy
                 'paste 'kill 'select-all
                 'insert-text-box 'insert-pasteboard-box
                 'insert-image)
  recursive? : any/c = #t
  time : exact-integer? = 0
```

See do-edit-operation in editor<%>.

Called when the snip's editor's method is called, *recursive?* is not #f, and this snip owns the caret.

```
(send a-snip draw dc
                  x
                  y
                  left
                  top
                  right
                  bottom
                  dx
                  dy
                  draw-caret) → void?
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  left : real?
  top : real?
  right : real?
  bottom : real?
  dx : real?
  dy : real?
  draw-caret : (one-of/c 'no-caret 'show-inactive-caret 'show-caret)
```

Specification: Called (by an editor) to draw the snip into the given drawing context with the snip's top left corner at location (*x*, *y*) in DC coordinates.

The arguments *left*, *top*, *right*, and *bottom* define a clipping region (in DC coordinates) that the snip can use to optimize drawing, but it can also ignore these arguments.

The *dx* and *dy* argument provide numbers that can be subtracted from *x* and *y* to obtain the snip's location in editor coordinates (as opposed to DC coordinates, which are used for drawing).

See §1.3.5 "Caret Ownership" for information about *draw-caret*.

Before this method is called, the correct font, text color, and pen color for the snip's style will have been set in the drawing context already. (This is *not* true for get-extent or partial-offset.) The draw method must not make any other assumptions about the state of the drawing context, except that the clipping region is already set to something appropriate. Before draw returns, it must restore any drawing context settings that it changes.

See also on-paint in editor<%>.

The snip's editor is usually internally locked for writing and reflowing when this method is called (see also §1.3.8 "Internal Editor Locks").

Default implementation: Draws nothing.

```
(send a-snip find-scroll-step y) → exact-nonnegative-integer?
  y : real?
```

Specification: If a snip contains more than one vertical scroll step (see get-num-scroll-steps) then this method is called to find a scroll step offset for a given y-offset into the snip.

Default implementation: Returns 0.

```
(send a-snip get-admin) → (or/c (is-a?/c snip-admin%) false/c)
```

Returns the administrator for this snip. (The administrator can be #f even if the snip is owned but not visible in the editor.)

```
(send a-snip get-count) → (integer-in 0 100000)
```

Returns the snip's count (i.e., number of items within the snip).

```
(send a-snip get-extent  dc
                         x
                         y
                        [w
                         h
                         descent
                         space
                         lspace
                         rspace]) → void?
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  w : (or/c (box/c (and/c real? (not/c negative?))) false/c)
    = #f
  h : (or/c (box/c (and/c real? (not/c negative?))) false/c)
    = #f
  descent : (or/c (box/c (and/c real? (not/c negative?))) false/c)
          = #f
  space : (or/c (box/c (and/c real? (not/c negative?))) false/c)
        = #f
  lspace : (or/c (box/c (and/c real? (not/c negative?))) false/c)
         = #f
  rspace : (or/c (box/c (and/c real? (not/c negative?))) false/c)
         = #f
```

Specification: Calculates the snip's width, height, descent (amount of height which is drawn below the baseline), space (amount of height which is "filler" space at the top), and horizontal spaces (amount of width which is "filler" space at the left and right). Those values are returned by filling the `w`, `h`, `descent`, `space`, `lspace`, and `rspace` boxes.

This method is called by the snip's administrator; it is not normally called directly by others. To get the extent of a snip, use `get-snip-location` in `editor<%>` .

A drawing context is provided for the purpose of finding font sizes, but no drawing should occur. The `get-extent` and `partial-offset` methods must not make any assumptions about the state of the drawing context, except that it is scaled properly. In particular, the font for the snip's style is not automatically set in the drawing context before the method is called. (Many snips cache their size information, so automatically setting the font would be wasteful.) If `get-extent` or `partial-offset` changes the drawing context's setting, it must restore them before returning. However, the methods should not need to change the drawing context; only font settings can affect measurement results from a device context, and `get-text-extent` in `dc<%>` accepts a `font%` argument for sizing that overrides that device context's current font.

The snip's left and top locations are provided as `x` and `y` in editor coordinates, in case the

snip's size depends on its location; the *x* and *y* arguments are usually ignored. In a text editor, the *y*-coordinate is the *line's* top location; the snip's actual top location is potentially undetermined until its height is known.

If a snip caches the result size for future replies, it should invalidate its cached size when `size-cache-invalid` is called (especially if the snip's size depends on any device context properties).

If a snip's size changes after receiving a call to `get-extent` and before receiving a call to `size-cache-invalid`, then the snip must notify its administrator of the size change, so that the administrator can recompute its derived size information. Notify the administrator of a size change by call its `resized` method.

The snip's editor is usually internally locked for writing and reflowing when this method is called (see also §1.3.8 "Internal Editor Locks").

Default implementation: Fills in all boxes with `0.0`.

`(send a-snip get-flags)` → `(listof symbol?)`

Returns flags defining the behavior of the snip, a list of the following symbols:

- `'is-text` — this is a text snip derived from `string-snip%`; do not set this flag
- `'can-append` — this snip can be merged with another snip of the same type
- `'invisible` — the user doesn't "see" this snip; e.g.: a carriage return
- `'hard-newline` — a newline must follow the snip
- `'newline` — a newline currently follows the snip; only an owning editor should set this flag
- `'handles-events` — this snip can handle keyboard and mouse events
- `'width-depends-on-x` — this snip's display width depends on the snip's x-location within the editor; e.g.: tab
- `'height-depends-on-y` — this snip's display height depends on the snip's y-location within the editor
- `'width-depends-on-y` — this snip's display width depends on the snip's y-location within the editor
- `'height-depends-on-x` — this snip's display height depends on the snip's x-location within the editor
- `'uses-editor-path` — this snip uses its editor's pathname and should be notified when the name changes; notification is given as a redundant call to `set-admin`

```
(send a-snip get-num-scroll-steps)
 → exact-nonnegative-integer?
```

Specification: Returns the number of horizontal scroll steps within the snip. For most snips, this is `1`. Embedded editor snips use this method so that scrolling in the owning editor will step through the lines in the embedded editor.

Default implementation: Returns `1`.

```
(send a-snip get-scroll-step-offset offset)
 → (and/c real? (not/c negative?))
  offset : exact-nonnegative-integer?
```

Specification: If a snip contains more than one vertical scroll step (see `get-num-scroll-steps`) then this method is called to find the y-offset into the snip for a given scroll offset.

Default implementation: Returns `0.0`.

```
(send a-snip get-snipclass) → (or/c #f (is-a?/c snip-class%))
```

Returns the snip's class, which is used for file saving and cut-and-paste.

Since this method returns the snip class stored by `set-snipclass`, it is not meant to be overridden.

```
(send a-snip get-style) → (is-a?/c style<%>)
```

Returns the snip's style. See also `set-style`.

```
(send a-snip get-text offset num [flattened?]) → string?
  offset : exact-nonnegative-integer?
  num : exact-nonnegative-integer?
  flattened? : any/c = #f
```

Specification: Returns the text for this snip starting with the position *offset* within the snip, and continuing for a total length of *num* items. If *offset* is greater than the snip's count, then `""` is returned. If *num* is greater than the snip's count minus the offset, then text from the offset to the end of the snip is returned.

If *flattened?* is not `#f`, then flattened text is returned. See §1.3.4 "Flattened Text" for a discussion of flattened vs. non-flattened text.

Default implementation: Returns `""`.

```
(send a-snip get-text! buffer
                       offset
                       num
                       buffer-offset) → void?
  buffer : (and/c string? (not/c immutable?))
  offset : exact-nonnegative-integer?
  num : exact-nonnegative-integer?
  buffer-offset : exact-nonnegative-integer?
```

Specification: Like `get-text` in non-flattened mode, except that the characters are put into the given mutable string, instead of returned in a newly allocated string.

The *buffer* string is filled starting at position *buffer-offset*. The *buffer* string must be at least *num*+*buffer-offset* characters long.

Default implementation: Calls `get-text`, except in the case of a `string-snip%`, in which case *buffer* is filled directly.

```
(send a-snip is-owned?) → boolean?
```

Returns `#t` if this snip has an owner, `#f` otherwise. Note that a snip may be owned by an editor if it was inserted and then deleted from the editor, if it's still in the editor's undo history.

```
(send a-snip match? snip) → boolean?
  snip : (is-a?/c snip%)
```

Specification: Return `#t` if *a-snip* "matches" *snip*, `#f` otherwise.

Default implementation: Returns `#t` if the *snip* and *a-snip* are from the same class and have the same length.

```
(send a-snip merge-with prev) → (or/c (is-a?/c snip%) false/c)
  prev : (is-a?/c snip%)
```

Specification: Merges *a-snip* with *prev*, returning `#f` if the snips cannot be merged or a new merged snip otherwise. This method will only be called if both snips are from the same class and both have the `'can-append` flag.

If the returned snip does not have the expected count, its count is forcibly modified. If the returned snip is already owned by another administrator, a surrogate snip is created.

The snip's editor is usually internally locked for reading when this method is called (see also

§1.3.8 “Internal Editor Locks”).

Default implementation: Returns `#f`.

`(send a-snip next) → (or/c (is-a?/c snip%) false/c)`

Returns the next snip in the editor owning this snip, or `#f` if this is the last snip.

In a text editor, the next snip is the snip at the position following this snip’s (last) position. In a pasteboard, the next snip is the one immediately behind this snip. (See §1.3.1 “Editor Structure and Terminology” for information about snip order in pasteboards.)

```
(send a-snip on-char dc
                     x
                     y
                     editorx
                     editory
                     event)  → void?
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  editorx : real?
  editory : real?
  event : (is-a?/c key-event%)
```

Specification: Called to handle keyboard events when this snip has the keyboard focus and can handle events. The drawing context is provided, as well as the snip’s location in display coordinates (the event uses display coordinates), and the snip’s location in editor coordinates.

The *x* and *y* arguments are the snip’s location in display coordinates. The *editorx* and *editory* arguments are the snip’s location in editor coordinates. To get *event*’s x location in snip coordinates, subtract *x* from `(send event get-x)`.

See also `'handles-events` in `get-flags`.

Default implementation: Does nothing.

```
(send a-snip on-event dc
                      x
                      y
                      editorx
                      editory
                      event)  → void?
  dc : (is-a?/c dc<%>)
  x : real?
```

```
  y : real?
  editorx : real?
  editory : real?
  event : (is-a?/c mouse-event%)
```

Specification: Called to handle mouse events on the snip when this snip can handle events and when the snip has the keyboard focus. See `on-char` for information about the arguments.

The *x* and *y* arguments are the snip's location in display coordinates. The *editorx* and *editory* arguments are the snip's location in editor coordinates. To get *event*'s x location in snip coordinates, subtract *x* from `(send event get-x)`.

See also `'handles-events` in `get-flags`.

Default implementation: Does nothing.

```
(send a-snip own-caret own-it?) → void?
  own-it? : any/c
```

Specification: Notifies the snip that it is or is not allowed to display the caret (indicating ownership of keyboard focus) in some display. This method is *not* called to request that the caret is actually shown or hidden; the `draw` method is called for all display requests.

The *own-it?* argument is `#t` if the snip owns the keyboard focus or `#f` otherwise.

Default implementation: Does nothing.

```
(send a-snip partial-offset dc x y len) → real?
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  len : exact-nonnegative-integer?
```

Specification: Calculates a partial width for the snip, starting from the first snip item and continuing for *len* items. The drawing context and snip's locations in editor coordinates are provided. See also `get-extent`.

The snip's editor is usually internally locked for writing and reflowing when this method is called (see also §1.3.8 "Internal Editor Locks").

Default implementation: Returns `0.0`.

```
(send a-snip previous) → (or/c (is-a?/c snip%) false/c)
```

Returns the previous snip in the editor owning this snip, or #f if this is the first snip.

```
(send a-snip release-from-owner) → boolean?
```

Specification: Asks the snip to try to release itself from its owner. If the snip is not owned or the release is successful, then #t is returned. Otherwise, #f is returned and the snip remains owned. See also is-owned?.

Use this method for moving a snip from one editor to another. This method notifies the snip's owning editor that someone else really wants control of the snip. It is not necessary to use this method for "cleaning up" a snip when it is deleted from an editor.

Default implementation: Requests a low-level release from the snip's owning administrator.

```
(send a-snip resize w h) → boolean?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Specification: Resizes the snip. The snip can refuse to be resized by returning #f. Otherwise, the snip will resize (it must call its administrator's resized method) and return #t.

See also on-interactive-resize in pasteboard%.

Default implementation: Returns #f.

```
(send a-snip set-admin admin) → void?
  admin : (or/c (is-a?/c snip-admin%) false/c)
```

Sets the snip's administrator. Only an administrator should call this method.

The default method sets the internal state of a snip to record its administrator. It will not modify this state if the snip is already owned by an administrator and the administrator has not blessed the transition. If the administrator state of a snip is not modified as expected during a sensitive call to this method by an instance of text% or pasteboard%, the internal state may be forcibly modified (if the new administrator was #f) or a surrogate snip may be created (if the snip was expected to receive a new administrator).

The snip's (new) editor is usually internally locked for reading when this method is called (see also §1.3.8 "Internal Editor Locks").

```
(send a-snip set-count c) → void?
  c : (integer-in 1 100000)
```

Specification: Sets the snip's count (i.e., the number of items within the snip).

The snip's count may be changed by the system (in extreme cases to maintain consistency) without calling this method.

Default implementation: Sets the snip's count and notifies the snip's administrator that the snip's size has changed.

```
(send a-snip set-flags flags) → void?
  flags : (listof symbol?)
```

Specification: Sets the snip's flags. See `get-flags`.

Default implementation: Sets the snip flags and notifies the snip's editor that its flags have changed.

```
(send a-snip set-snipclass class) → void?
  class : (is-a?/c snip-class%)
```

Sets the snip's class, used for file saving and cut-and-paste.

This method stores the snip class internally; other editor objects may access the snip class directly, instead of through the `get-snipclass` method.

```
(send a-snip set-style style) → void?
  style : (is-a?/c style<%>)
```

Sets the snip's style if it is not owned by any editor. See also `get-style` and `is-owned?`.

The snip's style may be changed by the system without calling this method.

```
(send a-snip set-unmodified) → void?
```

Specification: Called by the snip's administrator to notify the snip that its changed have been saved. The next time snip's internal state is modified by the user, it should call `modified` to report the state change (but only on the first change after this method is called, or the first change after the snip acquires a new administrator).

Default implementation: Does nothing.

```
(send a-snip size-cache-invalid) → void?
```

Specification: Called to notify the snip that it may need to recalculate its display arguments (width, height, etc.) when it is next asked, because the style or location of the snip has changed.

The snip's (new) editor is usually internally locked for reflowing when this method is called (see also §1.3.8 "Internal Editor Locks").

Default implementation: Does nothing.

```
(send a-snip split position first second) → void?
  position : exact-nonnegative-integer?
  first : (box/c (is-a?/c snip%))
  second : (box/c (is-a?/c snip%))
```

Specification: Splits the snip into two snips. This is called when a snip has more than one item and something is inserted between two items.

The arguments are a relative position integer and two boxes. The position integer specifies how many items should be given to the new first snip; the rest go to the new second snip. The two boxes must be filled with two new snips. (The old snip is no longer used, so it can be recycled as a new snip.)

If the returned snips do not have the expected counts, their counts are forcibly modified. If either returned snip is already owned by another administrator, a surrogate snip is created.

The snip's editor is usually internally locked for reading when this method is called (see also §1.3.8 "Internal Editor Locks").

Default implementation: Creates a new `snip%` instance with *`position`* elements, and modifies `a-snip` to decrement its count by *`position`*. The nest snip is installed into *`first`* and `a-snip` is installed into *`second`*.

```
(send a-snip write f) → void?
  f : (is-a?/c editor-stream-out%)
```

Writes the snip to the given stream. (Snip reading is handled by the snip class.) Style information about the snip (i.e., the content of `get-style`) will be saved and restored automatically.

2.5.23 snip-admin%

```
snip-admin% : class?
  superclass: object%
```

See §1.3.1.1 "Administrators" for information about the role of administrators. The `snip-admin%` class is never instantiated directly. It is not even instantiated through derived classes by most programmers; each `text%` or `pasteboard%` object creates its own administrator.

However, it may be useful to derive a new instance of this class to display snips in a new context. Also, it may be useful to call the methods of an existing administrator from an owned snip.

To create a new snip-admin% class, all methods described here must be overridden. They are all invoked by the administrator's snip.

Because a snip-admin% object typically owns more than one snip, many methods require a snip% object as an argument.

```
(new snip-admin%) → (is-a?/c snip-admin%)
```

Creates a (useless) editor administrator.

```
(send a-snip-admin get-dc) → (or/c (is-a?/c dc<%>) false/c)
```

Gets a drawing context suitable for determining display size information. If the snip is not displayed, #f is returned.

```
(send a-snip-admin get-editor)
 → (or/c (is-a?/c text%) (is-a?/c pasteboard%))
```

Returns the editor that this administrator reports to (directly or indirectly).

```
(send a-snip-admin get-view x y w h [snip]) → void?
  x : (or/c (box/c real?) false/c)
  y : (or/c (box/c real?) false/c)
  w : (or/c (box/c (and/c real? (not/c negative?))) false/c)
  h : (or/c (box/c (and/c real? (not/c negative?))) false/c)
  snip : (or/c (is-a?/c snip%) false/c) = #f
```

Specification: Gets the location and size of the visible region of a snip in snip coordinates. The result is undefined if the given snip is not managed by this administrator.

If *snip* is not #f, the current visible region of the snip is installed in the boxes *x*, *y*, *w*, and *h*. The *x* and *y* values are relative to the snip's top-left corner. The *w* and *h* values may be larger than the snip itself.

If *snip* is #f, the total visible region of the snip's top-level display is returned in editor coordinates. Using #f for *snip* is analogous to using #t for full? in get-view in editor-admin%.

If no snip is specified, then the location and size of the snip's editor are returned, instead, in editor coordinates.

See also get-view in editor-admin%.

Default implementation: Fills all boxes with 0.0.

```
(send a-snip-admin get-view-size h w) → void?
  h : (or/c (box/c (and/c real? (not/c negative?))) false/c)
  w : (or/c (box/c (and/c real? (not/c negative?))) false/c)
```

Specification: Gets the visible size of the administrator's display region.

If the display is an editor canvas, see also reflow-container.

Default implementation: Fills all boxes with 0.0.

```
(send a-snip-admin modified snip modified?) → void?
  snip : (is-a?/c snip%)
  modified? : any/c
```

Specification: Called by a snip to report that its modification state has changed to either modified or unmodified.

Default implementation: Does nothing.

```
(send a-snip-admin needs-update snip
                                localx
                                localy
                                w
                                h)      → void?
  snip : (is-a?/c snip%)
  localx : real?
  localy : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
```

Specification: Called by the snip to request that the snip's display needs to be updated. The administrator determines when to actually update the snip; the snip's draw method is eventually called.

The *localx*, *localy*, *w*, and *h* arguments specify a region of the snip to be refreshed (in snip coordinates).

No update occurs if the given snip is not managed by this administrator.

Default implementation: Does nothing.

```
(send a-snip-admin popup-menu menu snip x y) → boolean?
  menu : (is-a?/c popup-menu%)
  snip : (is-a?/c snip%)
  x : real?
  y : real?
```

Specification: Opens a popup menu in the display for this snip's editor. The result is `#t` if the popup succeeds, `#f` otherwise (independent of whether the user selects an item in the popup menu).

The menu is placed at *x* and *y* in *snip* coordinates.

While the menu is popped up, its target is set to the top-level editor in the display for this snip's editor. See `get-popup-target` for more information.

Default implementation: Returns `#f`.

```
(send a-snip-admin recounted snip refresh?) → void?
  snip : (is-a?/c snip%)
  refresh? : any/c
```

Specification: Called by a snip to notify the administrator that the specified snip has changed its count. The snip generally needs to be updated after changing its count, but the snip decides whether the update should occur immediately.

If *refresh?* is not `#f`, then the snip is requesting to be updated immediately. Otherwise, `needs-update` must eventually be called as well.

The method call is ignored if the given snip is not managed by this administrator.

Default implementation: Does nothing.

```
(send a-snip-admin release-snip snip) → boolean?
  snip : (is-a?/c snip%)
```

Specification: Requests that the specified snip be released. If this administrator is not the snip's owner or if the snip cannot be released, then `#f` is returned. Otherwise, `#t` is returned and the snip is no longer owned.

See also `release-snip` in `editor<%>` .

The result is `#f` if the given snip is not managed by this administrator.

Default implementation: Returns `#f`.

```
(send a-snip-admin resized snip refresh?) → void?
  snip : (is-a?/c snip%)
  refresh? : any/c
```

Specification: Called by a snip to notify the administrator that the specified snip has changed its display size. The snip generally needs to be updated after a resize, but the snip decides whether the update should occur immediately.

If *refresh?* is not `#f`, then the snip is requesting to be updated immediately, as if calling `needs-update`. Otherwise, `needs-update` must eventually be called as well.

The method call is ignored if the given snip is not managed by this administrator.

Default implementation: Does nothing.

```
(send a-snip-admin scroll-to  snip
                              localx
                              localy
                              w
                              h
                              refresh?
                             [bias])    → boolean?
  snip : (is-a?/c snip%)
  localx : real?
  localy : real?
  w : (and/c real? (not/c negative?))
  h : (and/c real? (not/c negative?))
  refresh? : any/c
  bias : (one-of/c 'start 'end 'none) = 'none
```

Specification: Called by the snip to request scrolling so that the given region is visible. The snip generally needs to be updated after a scroll, but the snip decides whether the update should occur immediately.

The *localx*, *localy*, *w*, and *h* arguments specify a region of the snip to be made visible by the scroll (in snip coordinates).

If *refresh?* is not `#f`, then the editor is requesting to be updated immediately.

The *bias* argument is one of:

- `'start` — if the range doesn't fit in the visible area, show the top-left region
- `'none` — no special scrolling instructions
- `'end` — if the range doesn't fit in the visible area, show the bottom-right region

The result is `#t` if the editor is scrolled, `#f` otherwise.

The method call is ignored (and the result is `#f`) if the given snip is not managed by this administrator.

Default implementation: Returns `#f`.

```
(send a-snip-admin set-caret-owner snip
                                   domain) → void?
  snip : (is-a?/c snip%)
  domain : (one-of/c 'immediate 'display 'global)
```

Specification: Requests that the keyboard focus is assigned to the specified snip. If the request is granted, the `own-caret` method of the snip is called.

See `set-caret-owner` for information about the possible values of *domain*.

The method call is ignored if the given snip is not managed by this administrator.

Default implementation: Does nothing.

```
(send a-snip-admin update-cursor) → void?
```

Specification: Queues an update for the cursor in the display for this snip's editor. The actual cursor used will be determined by calling the snip's `adjust-cursor` method as appropriate.

Default implementation: Does nothing.

2.5.24 snip-class%

```
snip-class% : class?
  superclass: object%
```

Useful snip classes are defined by instantiating derived subclasses of `snip-class%`. A class derived from `snip-class%` serves as a kind of "meta-class" for snips; each snip is associated with an instance of `snip-class%` as its snip class.

In deriving a new `snip-class%` class, override the `read` method. Then, for each instance of the derived class (where each instance corresponds to a single snip class):

- Set the classname using `set-classname`.
- Set the version using `set-version`.

- Install the class into the list returned by `get-the-snip-class-list` using the `add` method. Note that if the same name is inserted into the same class list multiple times, all but the first insertion is ignored.

See also §1.3.2.1.1 “Snip Classes”.

`(new snip-class%) → (is-a?/c snip-class%)`

Creates a (useless) snip class.

`(send a-snip-class get-classname) → string?`

Returns the class’s name, a string uniquely designating this snip class. For example, the standard text snip classname is `"wxtext"`. Names beginning with `wx` are reserved.

A snip class name should usually have the form `"((lib ...)\n(lib ...))"` to enable on-demand loading of the class. See §1.3.2.1.1 “Snip Classes” for details.

`(send a-snip-class get-version) → exact-integer?`

Returns the version of this snip class. When attempting to load a file containing a snip with the same class name but a different version, the user is warned.

```
(send a-snip-class read f) → (or/c (is-a?/c snip%) false/c)
  f : (is-a?/c editor-stream-in%)
```

Specification: Reads a snip from a given stream, returning a newly created snip as the result or `#f` if there is an error.

Default implementation: Returns `#f`.

```
(send a-snip-class read-header f) → boolean?
  f : (is-a?/c editor-stream-in%)
```

Specification: Called to read header information that may be useful for every snip read in this class. This method is only called once per editor read session, and only if the stream contains header information for this class.

The return value is `#f` if a read error occurs or anything else otherwise.

See also `write-header`.

Default implementation: Returns `#t`.

```
(send a-snip-class reading-version stream) → exact-integer?
  stream : (is-a?/c editor-stream-in%)
```

Returns the version number specified for this snip class for snips currently being read from the given stream.

```
(send a-snip-class set-classname name) → void?
  name : string?
```

Sets the class's name. See also `get-classname`.

```
(send a-snip-class set-version v) → void?
  v : exact-integer?
```

Sets the version of this class. See `get-version`.

```
(send a-snip-class write-header stream) → boolean?
  stream : (is-a?/c editor-stream-out%)
```

Specification: Called to write header information that may be useful for every snip written for this class. This method is only called once per editor write session, and only if the editor contains snips in this class.

When reading the snips back in, `read-header` will only be called if `write-header` writes some data to the stream.

The return value is `#f` if a write error occurs or anything else otherwise.

Default implementation: Returns `#t`.

2.5.25 snip-class-list<%>

```
snip-class-list<%> : interface?
```

Each eventspace has its own instance of `snip-class-list<%>`, obtained with `(get-the-snip-class-list)`. New instances cannot be created directly. Each instance keeps a list of snip classes. This list is needed for loading snips from a file. See also §1.3.2.1.1 "Snip Classes".

```
(send a-snip-class-list add snipclass) → void?
```

```
  snipclass : (is-a?/c snip-class%)
```

Adds a snip class to the list. If a class with the same name already exists in the list, this one will not be added.

```
(send a-snip-class-list find name)
 → (or/c (is-a?/c snip-class%) false/c)
  name : string?
```

Finds a snip class from the list with the given name, returning `#f` if none is found.

```
(send a-snip-class-list find-position class)
 → exact-nonnegative-integer?
  class : (is-a?/c snip-class%)
```

Returns an index into the list for the specified class.

```
(send a-snip-class-list nth n)
 → (or/c (is-a?/c snip-class%) false/c)
  n : exact-nonnegative-integer?
```

Returns the *n*th class in the list, or `#f` if the list has *n* classes or less.

```
(send a-snip-class-list number) → exact-nonnegative-integer?
```

Returns the number of snip classes in the list.

2.5.26 string-snip%

```
string-snip% : class?
  superclass: snip%
```

An instance of `string-snip%` is created automatically when text is inserted into a text editor. See also `on-new-string-snip` in `text%`.

```
(make-object string-snip% [allocsize]) → (is-a?/c string-snip%)
  allocsize : exact-nonnegative-integer? = 0
(make-object string-snip% s) → (is-a?/c string-snip%)
  s : string?
```

Creates a string snip whose initial content is *s*, if supplied, empty otherwise. In the latter

case, the optional *allocsize* argument is a hint about how much storage space for text should be initially allocated by the snip.

```
(send a-string-snip insert s len [pos]) → void?
  s : string?
  len : exact-nonnegative-integer?
  pos : exact-nonnegative-integer? = 0
```

Inserts *s* (with length *len*) into the snip at relative position *pos* within the snip.

```
(send a-string-snip read len f) → void?
  len : exact-nonnegative-integer?
  f : (is-a?/c editor-stream-in%)
```

Reads the snip's data from the given stream.

The *len* argument specifies the maximum length of the text to be read. (When a text snip is written to a file, the very first field is the length of the text contained in the snip.) This method is usually invoked by the text snip class's `read` method.

2.5.27 style<%>

`style<%> : interface?`

A `style<%>` object encapsulates drawing information (font, color, alignment, etc.) in a hierarchical manner. A `style<%>` object always exists within the context of a `style-list%` object and is never created except by a `style-list%` object.

See also §1.3.1.2 "Styles".

```
(send a-style get-alignment) → (one-of/c 'top 'center 'bottom)
```

Returns the style's alignment: `'top`, `'center`, or `'bottom`.

```
(send a-style get-background) → (is-a?/c color%)
```

Returns the style's background color.

```
(send a-style get-base-style)
 → (or/c (is-a?/c style<%>) false/c)
```

Returns the style's base style. See §1.3.1.2 "Styles" for more information. The return value is `#f` only for the basic style in the list.

```
(send a-style get-delta delta) → void?
  delta : (is-a?/c style-delta%)
```

Returns the style's delta information if the style is not a join style. See §1.3.1.2 "Styles" for more information.

```
(send a-style get-face) → (or/c string? false/c)
```

Returns the style's face name. See `font%`.

```
(send a-style get-family)
 → (one-of/c 'default 'decorative 'roman 'script
             'swiss 'modern 'symbol 'system)
```

Returns the style's font family. See `font%`.

```
(send a-style get-font) → (is-a?/c font%)
```

Returns the style's font information.

```
(send a-style get-foreground) → (is-a?/c color%)
```

Returns the style's foreground color.

```
(send a-style get-name) → (or/c string? false/c)
```

Returns the style's name, or `#f` if it is unnamed. Style names are only set through the style's `style-list%` object.

```
(send a-style get-shift-style) → (is-a?/c style<%>)
```

Returns the style's shift style if it is a join style. Otherwise, the root style is returned. See §1.3.1.2 "Styles" for more information.

```
(send a-style get-size) → (integer-in 0 255)
```

Returns the style's font size.

```
(send a-style get-size-in-pixels) → boolean?
```

Returns `#t` if the style size is in pixels, instead of points, or `#f` otherwise.

```
(send a-style get-smoothing)
 → (one-of/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
```

Returns the style's font smoothing. See `font%`.

```
(send a-style get-style) → (one-of/c 'normal 'italic 'slant)
```

Returns the style's font style. See `font%`.

```
(send a-style get-text-descent dc)
 → (and/c real? (not/c negative?))
  dc : (is-a?/c dc<%>)
```

Returns the descent of text using this style in a given DC.

```
(send a-style get-text-height dc)
 → (and/c real? (not/c negative?))
  dc : (is-a?/c dc<%>)
```

Returns the height of text using this style in a given DC.

```
(send a-style get-text-space dc)
 → (and/c real? (not/c negative?))
  dc : (is-a?/c dc<%>)
```

Returns the vertical spacing for text using this style in a given DC.

```
(send a-style get-text-width dc)
 → (and/c real? (not/c negative?))
  dc : (is-a?/c dc<%>)
```

Returns the width of a space character using this style in a given DC.

```
(send a-style get-transparent-text-backing) → boolean?
```

Returns `#t` if text is drawn without erasing the text background or `#f` otherwise.

```
(send a-style get-underlined) → boolean?
```

Returns `#t` if the style is underlined or `#f` otherwise.

```
(send a-style get-weight) → (one-of/c 'normal 'bold 'light)
```

Returns the style's font weight. See `font%`.

```
(send a-style is-join?) → boolean?
```

Returns `#t` if the style is a join style or `#f` otherwise. See §1.3.1.2 "Styles" for more information.

```
(send a-style set-base-style base-style) → void?
  base-style : (is-a?/c style<%>)
```

Sets the style's base style and recomputes the style's font, etc. See §1.3.1.2 "Styles" for more information.

```
(send a-style set-delta delta) → void?
  delta : (is-a?/c style-delta%)
```

Sets the style's delta (if it is not a join style) and recomputes the style's font, etc. See §1.3.1.2 "Styles" for more information.

```
(send a-style set-shift-style style) → void?
  style : (is-a?/c style<%>)
```

Sets the style's shift style (if it is a join style) and recomputes the style's font, etc. See §1.3.1.2 "Styles" for more information.

```
(send a-style switch-to dc old-style) → void?
  dc : (is-a?/c dc<%>)
  old-style : (or/c (is-a?/c style<%>) false/c)
```

Sets the font, pen color, etc. of the given drawing context. If `oldstyle` is not `#f`, only differences between the given style and this one are applied to the drawing context.

2.5.28 style-delta%

---

style-delta% : class?
  superclass: object%

A style-delta% object encapsulates a style change. The changes expressible by a delta include:

- changing the font family
- changing the font face
- changing the font size to a new value
- enlarging the font by an additive amount
- enlarging the font by a multiplicative amount, etc.
- changing the font style (normal, *italic*, or *slant*)
- toggling the font style
- changing the font to *italic* if it is currently *slant*, etc.
- changing the font weight, etc.
- changing the underline, etc.
- changing the vertical alignment, etc.
- changing the foreground color
- dimming or brightening the foreground color, etc.
- changing the background color, etc.
- changing text backing transparency

The set-delta method is convenient for most style delta settings; it takes a high-level delta specification and sets the internal delta information.

To take full advantage of a style delta, it is necessary to understand the internal on/off settings that can be manipulated through methods such as set-weight-on. For example, the font weight change is specified through the weight-on and weight-off internal settings. Roughly, weight-on turns on a weight setting when it is not present and weight-off turns off a weight setting when it is present. These two interact precisely in the following way:

- If both weight-on and weight-off are set to 'base, then the font weight is not changed.

- If `weight-on` is not `'base`, then the weight is set to `weight-on`.
- If `weight-off` is not `'base`, then the weight will be set back to `'normal` when the base style has the weight `weight-off`.
- If both `weight-on` and `weight-off` are set to the same value, then the weight is toggled with respect to that value: if the base style has the weight `weight-on`, then weight is changed to `'normal`; if the base style has a different weight, it is changed to `weight-on`.
- If both `weight-on` and `weight-off` are set, but to different values, then the weight is changed to `weight-on` only when the base style has the weight `weight-off`.

Font styles, smoothing, underlining, and alignment work in an analogous manner.

The possible values for `alignment-on` and `alignment-off` are:

- `'base`
- `'top`
- `'center`
- `'bottom`

The possible values for `style-on` and `style-off` are:

- `'base`
- `'normal`
- `'italic`
- `'slant`

The possible values for `smoothing-on` and `smoothing-off` are:

- `'base`
- `'default`
- `'partly-smoothed`
- `'smoothed`
- `'unsmoothed`

The possible values for `underlined-on` and `underlined-off` are:

- `#f` (acts like `'base`)
- `#t`

The possible values for `size-in-pixels-on` and `size-in-pixels-off` are:

- `#f` (acts like `'base`)
- `#t`

The possible values for `transparent-text-backing-on` and `transparent-text-backing-off` are:

- `#f` (acts like `'base`)
- `#t`

The possible values for `weight-on` and `weight-off` are:

- `'base`
- `'normal`
- `'bold`
- `'light`

The family and face settings in a style delta are interdependent:

- When a delta's face is `#f` and its family is `'base`, then neither the face nor family are modified by the delta.
- When a delta's face is a string and its family is `'base`, then only face is modified by the delta.
- When a delta's family is not `'base`, then both the face and family are modified by the delta. If the delta's face is `#f`, then applying the delta sets a style's face to `#f`, so that the family setting prevails in choosing a font.

```
(make-object style-delta% [change-command])
 → (is-a?/c style-delta%)
```

```
  change-command : (or/c 'change-nothing
                         'change-normal
                         'change-toggle-underline
                         'change-toggle-size-in-pixels
                         'change-normal-color
                         'change-bold)
                 = 'change-nothing
(make-object style-delta% change-command v)
 → (is-a?/c style-delta%)
  change-command : (or/c 'change-family
                         'change-style
                         'change-toggle-style
                         'change-weight
                         'change-toggle-weight
                         'change-smoothing
                         'change-toggle-smoothing
                         'change-alignment)
  v : symbol
(make-object style-delta% change-command v)
 → (is-a?/c style-delta%)
  change-command : (or/c 'change-size
                         'change-bigger
                         'change-smaller)
  v : (integer-in 0 255)
(make-object style-delta% change-command v)
 → (is-a?/c style-delta%)
  change-command : (or/c 'change-underline
                         'change-size-in-pixels)
  v : any/c
```

The initialization arguments are passed on to `set-delta`.

```
(send a-style-delta collapse delta) → boolean?
  delta : (is-a?/c style-delta%)
```

Tries to collapse into a single delta the changes that would be made by applying this delta after a given delta. If the return value is `#f`, then it is impossible to perform the collapse. Otherwise, the return value is `#t` and this delta will contain the collapsed change specification.

```
(send a-style-delta copy delta) → void?
  delta : (is-a?/c style-delta%)
```

Copies the given style delta's settings into this one.

```
(send a-style-delta equal? delta) → boolean?
  delta : (is-a?/c style-delta%)
```

Returns #t if the given delta is equivalent to this one in all contexts or #f otherwise.

```
(send a-style-delta get-alignment-off)
 → (or/c 'base 'top 'center 'bottom)
```

See style-delta%.

```
(send a-style-delta get-alignment-on)
 → (or/c 'base 'top 'center 'bottom)
```

See style-delta%.

```
(send a-style-delta get-background-add)
 → (is-a?/c add-color<%>)
```

Gets the object additive color shift for the background (applied after the multiplicative factor). Call this add-color<%> object's methods to change the style delta's additive background color shift.

```
(send a-style-delta get-background-mult)
 → (is-a?/c mult-color<%>)
```

Gets the multiplicative color shift for the background (applied before the additive factor). Call this mult-color<%> object's methods to change the style delta's multiplicative background color shift.

```
(send a-style-delta get-face) → (or/c string? false/c)
```

Gets the delta's font face string. If this string is #f and the family is 'base when the delta is applied to a style, the style's face and family are not changed. However, if the face string is #f and the family is not 'base, then the style's face is changed to #f.

See also get-family.

```
(send a-style-delta get-family)
 → (or/c 'base 'default 'decorative 'roman 'script
         'swiss 'modern 'symbol 'system)
```

Returns the delta's font family. The possible values are

- 'base — no change to family
- 'default
- 'decorative
- 'roman
- 'script
- 'swiss
- 'modern (fixed width)
- 'symbol (Greek letters)
- 'system (used to draw control labels)

See also get-face.

```
(send a-style-delta get-foreground-add)
 → (is-a?/c add-color<%>)
```

Gets the additive color shift for the foreground (applied after the multiplicative factor). Call this add-color<%> object's methods to change the style delta's additive foreground color shift.

```
(send a-style-delta get-foreground-mult)
 → (is-a?/c mult-color<%>)
```

Gets the multiplicative color shift for the foreground (applied before the additive factor). Call this mult-color<%> object's methods to change the style delta's multiplicative foreground color shift.

```
(send a-style-delta get-size-add) → (integer-in 0 255)
```

Gets the additive font size shift (applied after the multiplicative factor).

```
(send a-style-delta get-size-in-pixels-off) → boolean?
```

See style-delta%.

```
(send a-style-delta get-size-in-pixels-on) → boolean?
```

See style-delta%.

```
(send a-style-delta get-size-mult) → real?
```

Gets the multiplicative font size shift (applied before the additive factor).

```
(send a-style-delta get-smoothing-off)
 → (or/c 'base 'default 'partly-smoothed 'smoothed 'unsmoothed)
```

See style-delta%.

```
(send a-style-delta get-smoothing-on)
 → (or/c 'base 'default 'partly-smoothed 'smoothed 'unsmoothed)
```

See style-delta%.

```
(send a-style-delta get-style-off)
 → (or/c 'base 'normal 'italic 'slant)
```

See style-delta%.

```
(send a-style-delta get-style-on)
 → (or/c 'base 'normal 'italic 'slant)
```

See style-delta%.

```
(send a-style-delta get-transparent-text-backing-off)
 → boolean?
```

See style-delta%.

```
(send a-style-delta get-transparent-text-backing-on)
 → boolean?
```

See style-delta%.

```
(send a-style-delta get-underlined-off) → boolean?
```

See style-delta%.

```
(send a-style-delta get-underlined-on) → boolean?
```

See style-delta%.

```
(send a-style-delta get-weight-off)
 → (or/c 'base 'normal 'bold 'light)
```

See style-delta%.

```
(send a-style-delta get-weight-on)
 → (or/c 'base 'normal 'bold 'light)
```

See style-delta%.

```
(send a-style-delta set-alignment-off v) → void?
  v : (or/c 'base 'top 'center 'bottom)
```

See style-delta%.

```
(send a-style-delta set-alignment-on v) → void?
  v : (or/c 'base 'top 'center 'bottom)
```

See style-delta%.

```
(send a-style-delta set-delta [change-command])
 → (is-a?/c style-delta%)
  change-command : (or/c 'change-nothing
                         'change-normal
                         'change-toggle-underline
                         'change-toggle-size-in-pixels
                         'change-normal-color
                         'change-bold)
                 = 'change-nothing
(send a-style-delta set-delta change-command
                              param)
 → (is-a?/c style-delta%)
  change-command : (or/c 'change-family
                         'change-style
                         'change-toggle-style
                         'change-weight
                         'change-toggle-weight
                         'change-smoothing
                         'change-toggle-smoothing
                         'change-alignment)
  param : symbol
```

```
(send a-style-delta set-delta change-command
                                param)
 → (is-a?/c style-delta%)
  change-command : (or/c 'change-size
                         'change-bigger
                         'change-smaller)
  param : (integer-in 0 255)
(send a-style-delta set-delta change-command
                                on?)
 → (is-a?/c style-delta%)
  change-command : (or/c 'change-underline
                         'change-size-in-pixels)
  on? : any/c
```

Configures the delta with high-level specifications. The return value is the delta itself.

Except for `'change-nothing` and `'change-normal`, the command only changes part of the delta. Thus, applying `'change-bold` and then `'change-italic` sets the delta for both the style and weight change.

The *change-command* argument specifies how the delta is changed; the possible values are:

- `'change-nothing` — reset all changes
- `'change-normal` — turn off all styles and resizings
- `'change-toggle-underline` — underline regions that are currently not underlined, and vice versa
- `'change-toggle-size-in-pixels` — interpret sizes in pixels for regions that are currently interpreted in points, and vice versa
- `'change-normal-color` — change the foreground and background to black and white, respectively
- `'change-italic` — change the style of the font to *italic*
- `'change-bold` — change the weight of the font to **bold**
- `'change-family` — change the font family (*param* is a family; see `font%`); see also `get-family`
- `'change-style` — change the style of the font (*param* is a style; see `font%`)
- `'change-toggle-style` — toggle the style of the font (*param* is a style; see `font%`)
- `'change-weight` — change the weight of the font (*param* is a weight; see `font%`)
- `'change-toggle-weight` — toggle the weight of the font (*param* is a weight; see `font%`)

- 'change-smoothing — change the smoothing of the font (*param* is a smoothing; see font%)
- 'change-toggle-smoothing — toggle the smoothing of the font (*param* is a smoothing; see font%)
- 'change-alignment — change the alignment (*param* is an alignment; see style-delta%)
- 'change-size — change the size to an absolute value (*param* is a size)
- 'change-bigger — make the text larger (*param* is an additive amount)
- 'change-smaller — make the text smaller (*param* is an additive amount)
- 'change-underline — set the underline status to either underlined or plain
- 'change-size-in-pixels — set the size interpretation to pixels or points

---

```
(send a-style-delta set-delta-background name)
 → (is-a?/c style-delta%)
  name : string?
(send a-style-delta set-delta-background color)
 → (is-a?/c style-delta%)
  color : (is-a?/c color%)
```

Makes the delta encode a background color change to match the absolute color given; that is, it sets the multiplicative factors to 0.0 in the result of get-background-mult, and it sets the additive values in the result of get-background-add to the specified color's values. In addition, it also disables transparent text backing by setting transparent-text-backing-on to #f and transparent-text-backing-off to #t. The return value of the method is the delta itself.

For the case that a string color name is supplied, see color-database<%>.

```
(send a-style-delta set-delta-face  name
                                    [family])
 → (is-a?/c style-delta%)
  name : string?
  family : (or/c 'base 'default 'decorative 'roman
                 'script 'swiss 'modern 'symbol 'system)
         = 'default
```

Like set-face, but sets the family at the same time.

The return value is a-style-delta.

```
(send a-style-delta set-delta-foreground name)
 → (is-a?/c style-delta%)
  name : string?
(send a-style-delta set-delta-foreground color)
 → (is-a?/c style-delta%)
  color : (is-a?/c color%)
```

Makes the delta encode a foreground color change to match the absolute color given; that is, it sets the multiplicative factors to `0.0` in the result of `get-foreground-mult`, and it sets the additive values in the result of `get-foreground-add` to the specified color's values. The return value of the method is the delta itself.

For the case that a string color name is supplied, see `color-database<%>`.

```
(send a-style-delta set-face v) → void?
  v : (or/c string? false/c)
```

See `get-face`. See also `set-delta-face`.

```
(send a-style-delta set-family v) → void?
  v : (or/c 'base 'default 'decorative 'roman 'script
            'swiss 'modern 'symbol 'system)
```

Sets the delta's font family. See `get-family`.

```
(send a-style-delta set-size-add v) → void?
  v : (integer-in 0 255)
```

Sets the additive font size shift (applied after the multiplicative factor).

```
(send a-style-delta set-size-in-pixels-off v) → void?
  v : any/c
```

See `style-delta%`.

```
(send a-style-delta set-size-in-pixels-on v) → void?
  v : any/c
```

See `style-delta%`.

```
(send a-style-delta set-size-mult v) → void?
```

```
  v : real?
```

Sets the multiplicative font size shift (applied before the additive factor).

```
(send a-style-delta set-smoothing-off v) → void?
  v : (or/c 'base 'default 'partly-smoothed 'smoothed 'unsmoothed)
```

See style-delta%.

```
(send a-style-delta set-smoothing-on v) → void?
  v : (or/c 'base 'default 'partly-smoothed 'smoothed 'unsmoothed)
```

See style-delta%.

```
(send a-style-delta set-style-off v) → void?
  v : (or/c 'base 'normal 'italic 'slant)
```

See style-delta%.

```
(send a-style-delta set-style-on v) → void?
  v : (or/c 'base 'normal 'italic 'slant)
```

See style-delta%.

```
(send a-style-delta set-transparent-text-backing-off v) → void?
  v : any/c
```

See style-delta%.

```
(send a-style-delta set-transparent-text-backing-on v) → void?
  v : any/c
```

See style-delta%.

```
(send a-style-delta set-underlined-off v) → void?
  v : any/c
```

See style-delta%.

```
(send a-style-delta set-underlined-on v) → void?
  v : any/c
```

See style-delta%.

(send a-style-delta set-weight-off v) → void?
  v : (or/c 'base 'normal 'bold 'light)

See style-delta%.

(send a-style-delta set-weight-on v) → void?
  v : (or/c 'base 'normal 'bold 'light)

See style-delta%.

2.5.29 style-list%

---

style-list% : class?
  superclass: object%

A style-list% object contains a set of style<%> objects and maintains the hierarchical relationships between them. A style<%> object can only be created through the methods of a style-list% object. There is a global style list object, the-style-list, but any number of independent lists can be created for separate style hierarchies. Each editor creates its own private style list.

See §1.3.1.2 “Styles” for more information.

(new style-list%) → (is-a?/c style-list%)

The root style, named "Basic", is automatically created.

(send a-style-list basic-style) → (is-a?/c style<%>)

Returns the root style. Each style list has its own root style.

See also §3.1 “Preferences” for information about the 'GRacket:default-font-size preference.

(send a-style-list convert style) → (is-a?/c style<%>)
  style : (is-a?/c style<%>)

Converts style, which can be from another style list, to a style in this list. If style is already in this list, then style is returned. If style is named and a style by that name is

already in this list, then the existing named style is returned. Otherwise, the style is converted by converting its base style (and shift style if *style* is a join style) and then creating a new style in this list.

```
(send a-style-list find-named-style name)
 → (or/c (is-a?/c style<%>) false/c)
  name : string?
```

Finds a style by name. If no such style can be found, `#f` is returned.

```
(send a-style-list find-or-create-join-style base-style
                                             shift-style)
 → (is-a?/c style<%>)
  base-style : (is-a?/c style<%>)
  shift-style : (is-a?/c style<%>)
```

Creates a new join style, or finds an appropriate existing one. The returned style is always unnamed. See §1.3.1.2 “Styles” for more information.

The *base-style* argument must be a style within this style list.

```
(send a-style-list find-or-create-style base-style
                                        delta)
 → (is-a?/c style<%>)
  base-style : (is-a?/c style<%>)
  delta : (is-a?/c style-delta%)
```

Creates a new derived style, or finds an appropriate existing one. The returned style is always unnamed. See §1.3.1.2 “Styles” for more information.

The *base-style* argument must be a style within this style list. If *base-style* is not a join style, if it has no name, and if its delta can be collapsed with *delta* (see `collapse` in `style-delta%`), then the collapsed delta is used in place of *delta*, and the base style of *base-style* is used in place of *base-style*; this collapsing and substitution of base styles is performed recursively.

```
(send a-style-list forget-notification key) → void?
  key : any/c
```

See `notify-on-change`.

The *key* argument is the value returned by `notify-on-change`.

```
(send a-style-list index-to-style i)
 → (or/c (is-a?/c style<%>) false/c)
  i : exact-nonnegative-integer?
```

Returns the style associated with the given index, or `#f` for a bad index. See also `style-to-index`.

```
(send a-style-list new-named-style name
                                    like-style)
 → (is-a?/c style<%>)
  name : string?
  like-style : (is-a?/c style<%>)
```

Creates a new named style, unless the name is already being used.

If *name* is already being used, then *like-style* is ignored and the old style associated to the name is returned. Otherwise, a new style is created for *name* with the same characteristics (i.e., the same base style and same style delta or shift style) as *like-style*.

The *like-style* style must be in this style list, otherwise the named style is derived from the basic style with an empty style delta.

```
(send a-style-list notify-on-change f) → any/c
  f : ((or/c (is-a?/c style<%>) #f) . -> . any)
```

Attaches a callback to the style list. The callback is invoked whenever a style is modified.

Often, a change in one style will trigger a change in several other derived styles; to allow clients to handle all the changes in a batch, `#f` is passed in as the changing style after a set of styles has been processed.

The return value from `notify-on-change` is an opaque key to be used with `forget-notification`.

```
(send a-style-list number) → exact-nonnegative-integer?
```

Returns the number of styles in the list.

```
(send a-style-list replace-named-style name
                                        like-style)
 → (is-a?/c style<%>)
  name : string?
  like-style : (is-a?/c style<%>)
```

Like new-named-style, except that if the name is already mapped to a style, the existing mapping is replaced.

```
(send a-style-list style-to-index style)
 → (or/c exact-nonnegative-integer? false/c)
  style : (is-a?/c style<%>)
```

Returns the index for a particular style. The index for a style's base style (and shift style, if it is a join style) is guaranteed to be lower than the style's own index. (As a result, the root style's index is always 0.) A style's index can change whenever a new style is added to the list, or the base style or shift style of another style is changed.

If the given style is not in this list, #f is returned.

2.5.30 tab-snip%

```
tab-snip% : class?
  superclass: string-snip%
```

An instance of tab-snip% is created automatically when a tab is inserted into an editor.

```
(new tab-snip%) → (is-a?/c tab-snip%)
```

Creates a snip for a single tab, though the tab is initially empty.

Normally, a single tab is inserted into a tab-snip% object using the insert method.

The tab's content is not drawn, through it is used when determining the size of a single character in editors where tabbing is determined by the character width (see set-tabs); if the content is a single tab character (the normal case), then the average character width of snip's font is used as the tab's width.

2.5.31 text%

```
text% : class?
  superclass: object%
  extends: editor<%>
```

A text% object is a standard text editor. A text editor is displayed on the screen through an editor-canvas% object or some other display.

```
(new text% [[line-spacing line-spacing]
            [tab-stops tab-stops]
            [auto-wrap auto-wrap]])      → (is-a?/c text%)
  line-spacing : (and/c real? (not/c negative?)) = 1.0
  tab-stops : (listof real?) = null
  auto-wrap : any/c = #f
```

The *line-spacing* argument sets the additional amount of space (in DC units) inserted between each line in the editor when the editor is displayed. This spacing is included in the reported height of each line.

See set-tabs for information about tabstops.

If *auto-wrap* is true, then auto-wrapping is enabled via auto-wrap.

A new keymap% object is created for the new editor. See also get-keymap and set-keymap.

A new style-list% object is created for the new editor. See also get-style-list and set-style-list.

```
(send a-text after-change-style start len) → void?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?
```

Refine this method with augment.

Specification: Called after the style is changed for a given range (and after the display is refreshed; use on-change-style and begin-edit-sequence to avoid extra refreshes when after-change-style modifies the editor).

See also can-change-style? and on-edit-sequence.

No internals locks are set when this method is called.

Default implementation: Does nothing.

```
(send a-text after-delete start len) → void?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?
```

Refine this method with augment.

Specification: Called after a given range is deleted from the editor (and after the display is refreshed; use on-delete and begin-edit-sequence to avoid extra refreshes when after-delete modifies the editor).

The *start* argument specifies the starting position of the deleted range. The *len* argument specifies number of deleted items (so *start*+*len* is the ending position of the deleted range).

See also can-delete? and on-edit-sequence.

No internals locks are set when this method is called.

Default implementation: Does nothing.

(send a-text after-insert start len) → void?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?

Refine this method with augment.

Specification: Called after items are inserted into the editor (and after the display is refreshed; use on-insert and begin-edit-sequence to avoid extra refreshes when after-insert modifies the editor).

The *start* argument specifies the position of the insert. The *len* argument specifies the total length (in positions) of the inserted items.

See also can-insert? and on-edit-sequence.

No internals locks are set when this method is called.

Default implementation: Does nothing.

(send a-text after-merge-snips pos) → void?
  pos : exact-nonnegative-integer?

Refine this method with augment.

Specification: Called after adjacent snips in the editor are combined into one.

The *pos* argument specifies the position within the editor where the snips were merged (i.e., one old snip was just before *pos*, one old was just after *pos*, and the new snip spans *pos*).

See also merge-with.

Default implementation: Does nothing.

(send a-text after-set-position) → void?

Refine this method with augment.

Specification: Called after the start and end position have been moved (but not when the position is moved due to inserts or deletes).

See also on-edit-sequence.

Default implementation: Does nothing.

(send a-text after-set-size-constraint) → void?

Refine this method with augment.

Specification: Called after the editor's maximum or minimum height or width is changed (and after the display is refreshed; use on-set-size-constraint and begin-edit-sequence to avoid extra refreshes when after-set-size-constraint modifies the editor).

(This callback method is provided because setting an editor's maximum width may cause lines to be re-flowed with soft carriage returns.)

See also can-set-size-constraint? and on-edit-sequence.

Default implementation: Does nothing.

(send a-text after-split-snip pos) → void?
  pos : exact-nonnegative-integer?

Refine this method with augment.

Specification: Called after a snip in the editor is split into two, either through a call to split-snip or during some other action, such as inserting.

The pos argument specifies the position within the editor where a snip was split.

Default implementation: Does nothing.

(send a-text call-clickback start end) → void?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?

Simulates a user click that invokes a clickback, if the given range of positions is within a clickback's region. See also §1.3.7 "Clickbacks".

(send a-text can-change-style? start len) → boolean?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?

Refine this method with `augment`.

Specification: Called before the style is changed in a given range of the editor. If the return value is `#f`, then the style change will be aborted.

The editor is internally locked for writing during a call to this method (see also §1.3.8 "Internal Editor Locks"). Use `after-change-style` to modify the editor, if necessary.

See also `on-change-style`, `after-change-style`, and `on-edit-sequence`.

Default implementation: Returns `#t`.

```
(send a-text can-delete? start len) → boolean?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?
```

Refine this method with `augment`.

Specification: Called before a range is deleted from the editor. If the return value is `#f`, then the delete will be aborted.

The *start* argument specifies the starting position of the range to delete. The *len* argument specifies number of items to delete (so *start*+*len* is the ending position of the range to delete).

The editor is internally locked for writing during a call to this method (see also §1.3.8 "Internal Editor Locks"). Use `after-delete` to modify the editor, if necessary.

See also `on-delete`, `after-delete`, and `on-edit-sequence`.

Default implementation: Returns `#t`.

```
(send a-text can-insert? start len) → boolean?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?
```

Refine this method with `augment`.

Specification: Called before items are inserted into the editor. If the return value is `#f`, then the insert will be aborted.

The *start* argument specifies the position of the potential insert. The *len* argument specifies the total length (in positions) of the items to be inserted.

The editor is internally locked for writing during a call to this method (see also §1.3.8 "Internal Editor Locks"). Use `after-insert` to modify the editor, if necessary.

See also on-insert, after-insert, and on-edit-sequence.

Default implementation: Returns #t.

(send a-text can-set-size-constraint?) → boolean?

Refine this method with augment.

Specification: Called before the editor's maximum or minimum height or width is changed. If the return value is #f, then the change will be aborted.

(This callback method is provided because setting an editor's maximum width may cause lines to be re-flowed with soft carriage returns.)

See also on-set-size-constraint, after-set-size-constraint, and on-edit-sequence.

Default implementation: Returns #t.

(send a-text caret-hidden?) → boolean?

Returns #t if the caret is hidden for this editor or #f otherwise.

See also hide-caret.

```
(send a-text change-style  delta
                           [start
                            end
                            counts-as-mod?]) → void?
  delta : (or/c (is-a?/c style-delta%) #f)
  start : (or/c exact-nonnegative-integer? 'start) = 'start
  end : (or/c exact-nonnegative-integer? 'end) = 'end
  counts-as-mod? : any/c = #t
(send a-text change-style  style
                           [start
                            end
                            counts-as-mod?]) → void?
  style : (or/c (is-a?/c style<%>) #f)
  start : (or/c exact-nonnegative-integer? 'start) = 'start
  end : (or/c exact-nonnegative-integer? 'end) = 'end
  counts-as-mod? : any/c = #t
```

Extends change-style in editor<%>.

Changes the style for a region in the editor by applying a style delta or installing a specific

style. If *start* is 'start and *end* is 'end, then the currently selected items are changed. Otherwise, if *end* is 'end, then the style is changed from *start* until the end of the selection. If *counts-as-mod?* is #f, then set-modified is not called after applying the style change.

When *style* is provided: The editor's style list must contain *style*, otherwise the style is not changed. See also convert in style-list%.

```
(send a-text copy [extend? time start end]) → void?
  extend? : any/c = #f
  time : exact-integer? = 0
  start : (or/c exact-nonnegative-integer? 'start) = 'start
  end : (or/c exact-nonnegative-integer? 'end) = 'end
```

Extends copy in editor<%>.

Copies specified range of text into the clipboard. If *extend?* is not #f, the old clipboard contents are appended. If *start* is 'start or *end* is 'end, then the current selection start/end is used.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-text copy-self-to dest) → void?
  dest : (or/c (is-a?/c text%) (is-a?/c pasteboard%))
```

Overrides copy-self-to in editor<%>.

In addition to the default copy-self-to in editor<%> work, this editor's file format, wordbreak function, wordbreak map, click-between-threshold, caret visibility state, overwrite mode state, and autowrap bitmap are installed into *dest*.

```
(send a-text cut [extend? time start end]) → void?
  extend? : any/c = #f
  time : exact-integer? = 0
  start : (or/c exact-nonnegative-integer? 'start) = 'start
  end : (or/c exact-nonnegative-integer? 'end) = 'end
```

Overrides cut in editor<%>.

Copies and then deletes the specified range. If *extend?* is not #f, the old clipboard contents are appended. If *start* is 'start or *end* is 'end, then the current selection start/end is used.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is

outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-text delete start [end scroll-ok?]) → void?
  start : (or/c exact-nonnegative-integer? 'start)
  end : (or/c exact-nonnegative-integer? 'back) = 'back
  scroll-ok? : any/c = #t
(send a-text delete) → void?
```

Deletes the specified range or the currently selected text (when no range is provided) in the editor. If *start* is 'start, then the starting selection position is used; if *end* is 'back, then only the character preceding *start* is deleted. If *scroll-ok?* is not #f and *start* is the same as the current caret position, then the editor's display may be scrolled to show the new selection position.

The content of an editor can be changed by the system in response to other method calls, and such changes do not go through this method; use on-delete to monitor content deletion changes.

```
(send a-text do-copy start end time extend?) → void?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?
  time : exact-integer?
  extend? : any/c
```

Overrides do-copy in editor<%>.

Specification: Called to copy a region of the editor into the clipboard. This method is provided so that it can be overridden by subclasses. Do not call this method directly; instead, call copy.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

Default implementation: Copy the data from *start* to *end*, extending the current clipboard contexts if *extend?* is not #f.

```
(send a-text do-paste start time) → void?
  start : exact-nonnegative-integer?
  time : exact-integer?
```

Overrides do-paste in editor<%>.

Specification: Called to paste the current contents of the clipboard into the editor. This method is provided so that it can be overridden by subclasses. Do not call this method directly; instead, call paste.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

Default implementation: Pastes into the position *start*.

```
(send a-text do-paste-x-selection start
                                  time) → void?
  start : exact-nonnegative-integer?
  time : exact-integer?
```

Overrides do-paste-x-selection in editor<%>.

Specification: Called to paste the current contents of X selection under X (or the clipboard under Windows or Mac OS X) into the editor. This method is provided so that it can be overridden by subclasses. Do not call this method directly; instead, call paste-x-selection.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

Default implementation: Pastes into the position *start*.

```
(send a-text erase) → void?
```

Erases the contents of the editor.

See also delete.

```
(send a-text find-line y [on-it?]) → exact-nonnegative-integer?
  y : real?
  on-it? : (or/c (box/c any/c) #f) = #f
```

Given a location in the editor, returns the line at the location. Lines are numbered starting with 0.

The *on-it?* box is filled with #t if the line actually touches this position, or #f otherwise, unless *on-it?* is #f. (A large enough *y* will always return the last line number, but will set *on-it?* to #f.)

The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?).

```
(send a-text find-newline [direction
                           start
                           end])
```

```
 → (or/c exact-nonnegative-integer? #f)
  direction : (or/c 'forward 'backward) = 'forward
  start : (or/c exact-nonnegative-integer? 'start) = 'start
  end : (or/c exact-nonnegative-integer? 'eof) = 'eof
```

Like `find-string`, but specifically finds a paragraph break (possibly more efficiently than searching text).

```
(send a-text find-next-non-string-snip after)
 → (or/c (is-a?/c snip%) #f)
  after : (or/c (is-a?/c snip%) #f)
```

Given a snip, returns the next snip in the editor (after the given one) that is not an instance of `string-snip%`. If `#f` is given as the snip, the result is the first non-string snip in the editor (if any). If no non-string snip is found after the given snip, the result is `#f`.

```
(send a-text find-position  x
                            y
                           [at-eol?
                            on-it?
                            edge-close?])
 → exact-nonnegative-integer?
  x : real?
  y : real?
  at-eol? : (or/c (box/c any/c) #f) = #f
  on-it? : (or/c (box/c any/c) #f) = #f
  edge-close? : (or/c (box/c real?) #f) = #f
```

Given a location in the editor, returns the position at the location.

See §1.3.3 “End of Line Ambiguity” for a discussion of the *at-eol?* argument. The *on-it?* box is filled with `#t` if the line actually touches this position, or `#f` otherwise, unless *on-it?* is `#f`.

The *edge-close?* box is filled with it will be filled in with a value indicating how close the point is to the vertical edges of the item when the point falls on the item, unless *edge-close?* is `#f`. If the point is closest to the left edge of the item, the value will be negative; otherwise, the value will be positive. In either case, then absolute value of the returned result is the distance from the point to the edge of the item. The values 100 and -100 indicate infinity.

The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see `refresh-delayed?`).

```
(send a-text find-position-in-line  line
                                    x
                                   [at-eol?
                                    on-it?
                                    edge-close?])
 → exact-nonnegative-integer?
  line : exact-nonnegative-integer?
  x : real?
  at-eol? : (or/c (box/c any/c) #f) = #f
  on-it? : (or/c (box/c any/c) #f) = #f
  edge-close? : (or/c (box/c real?) #f) = #f
```

Given a location within a line of the editor, returns the position at the location. Lines are numbered starting with 0.

See §1.3.3 "End of Line Ambiguity" for a discussion of the *at-eol?* argument. The *on-it?* box is filled with #t if the line actually touches this position, or #f otherwise, unless *on-it?* is #f.

See find-position for a discussion of *edge-close?*.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?).

```
(send a-text find-snip pos direction [s-pos])
 → (or/c (is-a?/c snip%) #f)
  pos : exact-nonnegative-integer?
  direction : (or/c 'before-or-none 'before 'after 'after-or-none)
  s-pos : (or/c (box/c exact-nonnegative-integer?) #f) = #f
```

Returns the snip at a given position, or #f if an appropriate snip cannot be found.

If the position *pos* is between two snips, *direction* specifies which snip to return; *direction* can be any of the following:

- 'before-or-none — returns the snip before the position, or #f if *pos* is 0
- 'before — returns the snip before the position, or the first snip if *pos* is 0
- 'after — returns the snip after the position, or the last snip if *pos* is the last position
- 'after-or-none – returns the snip after the position, or #f if *pos* is the last position or larger

The *s-pos* box is filled with the position where the returned snip starts, unless *s-pos* is #f.

```
(send a-text find-string  str
                          [direction
                           start
                           end
                           get-start?
                           case-sensitive?])
 → (or/c exact-nonnegative-integer? #f)
  str : string?
  direction : (or/c 'forward 'backward) = 'forward
  start : (or/c exact-nonnegative-integer? 'start) = 'start
  end : (or/c exact-nonnegative-integer? 'eof) = 'eof
  get-start? : any/c = #t
  case-sensitive? : any/c = #t
```

Finds an exact-match string in the editor and returns its position. If the string is not found, `#f` is returned.

The *direction* argument can be `'forward` or `'backward`, indicating a forward search or backward search respectively. In the case of a forward search, the return value is the starting position of the string; for a backward search, the ending position is returned. However, if *get-start?* is `#f`, then the other end of the string position will be returned.

The *start* and *end* arguments set the starting and ending positions of a forward search (use *start* > *end* for a backward search). If *start* is `'start`, then the search starts at the start of the selection. If *end* is `'eof`, then the search continues to the end (for a forward search) or start (for a backward search) of the editor.

If *case-sensitive?* is `#f`, then an uppercase and lowercase of each alphabetic character are treated as equivalent.

```
(send a-text find-string-all  str
                              [direction
                               start
                               end
                               get-start?
                               case-sensitive])
 → (listof exact-nonnegative-integer?)
  str : string?
  direction : (or/c 'forward 'backward) = 'forward
  start : (or/c exact-nonnegative-integer? 'start) = 'start
  end : (or/c exact-nonnegative-integer? 'eof) = 'eof
  get-start? : any/c = #t
  case-sensitive : any/c = #t
```

Finds all occurrences of a string using `find-string`. If no occurrences are found, the empty list is returned. The arguments are the same as for `find-string`.

```
(send a-text find-wordbreak start
                            end
                            reason) → void?
  start : (or/c (box/c exact-nonnegative-integer?) #f)
  end : (or/c (box/c exact-nonnegative-integer?) #f)
  reason : (or/c 'caret 'line 'selection 'user1 'user2)
```

Finds wordbreaks in the editor using the current wordbreak procedure. See also `set-wordbreak-func`.

The contents of the *start* argument specifies an position to start searching backwards to the next word start; its will be filled with the starting position of the word that is found. If *start* is `#f`, no backward search is performed.

The contents of the *end* argument specifies an position to start searching forwards to the next word end; its will be filled with the ending position of the word that is found. If *end* is `#f`, no forward search is performed.

The *reason* argument specifies more information about what the wordbreak is used for. For example, the wordbreaks used to move the caret may be different from the wordbreaks used to break lines. The possible values of *reason* are:

- `'caret` — find a wordbreak suitable for moving the caret
- `'line` — find a wordbreak suitable for breaking lines
- `'selection` — find a wordbreak suitable for selecting the closest word
- `'user1` — for other (not built-in) uses
- `'user2` — for other (not built-in) uses

The actual handling of *reason* is controlled by the current wordbreak procedure; see `set-wordbreak-func`for details. The default handler and default wordbreak map treats alphanumeric characters the same for `'caret`, `'line`, and `'selection`. Non-alphanumeric, non-space, non-hyphen characters do not break lines, but do break caret and selection words. For example a comma should not be counted as part of the preceding word for moving the caret past the word or double-clicking the word, but the comma should stay on the same line as the word (and thus counts in the same "line word").

```
(send a-text flash-off) → void?
```

Turns off the hiliting and shows the normal selection range again; see `flash-on`. There is no effect if this method is called when flashing is already off.

```
(send a-text flash-on  start
                       end
                      [at-eol?
                       scroll?
                       timeout]) → void?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?
  at-eol? : any/c = #f
  scroll? : any/c = #t
  timeout : exact-nonnegative-integer? = 500
```

Temporarily hilites a region in the editor without changing the current selection.

See §1.3.3 "End of Line Ambiguity" for a discussion of the *at-eol?* argument. If *scroll?* is not `#f`, the editor's display will be scrolled if necessary to show the hilited region. If *timeout* is greater than 0, then the hiliting will be automatically turned off after the given number of milliseconds.

See also `flash-off`.

```
(send a-text get-anchor) → boolean?
```

Returns `#t` if the selection is currently auto-extending. See also `set-anchor`.

```
(send a-text get-between-threshold)
 → (and/c real? (not/c negative?))
```

Returns an amount used to determine the meaning of a user click. If the click falls within the threshold of a position between two items, then the click registers on the space between the items rather than on either item.

See also `set-between-threshold`.

```
(send a-text get-character start) → char?
  start : exact-nonnegative-integer?
```

Returns the character following the position *start*. The character corresponds to getting non-flattened text from the editor.

If *start* is greater than or equal to the last position, `#\nul` is returned.

```
(send a-text get-end-position) → exact-nonnegative-integer?
```

Returns the ending position of the current selection. See also get-position.

```
(send a-text get-file-format)
 → (or/c 'standard 'text 'text-force-cr)
```

Returns the format of the last file saved from or loaded into this editor. See also load-file.

```
(send a-text get-line-spacing)
 → (and/c real? (not/c negative?))
```

Returns the spacing inserted by the editor between each line. This spacing is included in the reported height of each line.

```
(send a-text get-overwrite-mode) → boolean?
```

Returns #t if the editor is in overwrite mode, #f otherwise. Overwrite mode only affects the way that on-default-char handles keyboard input for insertion characters. See also set-overwrite-mode.

```
(send a-text get-position start [end]) → void?
  start : (or/c (box/c exact-nonnegative-integer?) #f)
  end : (or/c (box/c exact-nonnegative-integer?) #f) = #f
```

Returns the current selection range in positions. If nothing is selected, the *start* and *end* will be the same number and that number will be where the insertion point is.

See also get-start-position and get-end-position.

The *start* box is filled with the starting position of the selection, unless *start* is #f. The *end* box is filled with the ending position of the selection, unless *end* is #f.

```
(send a-text get-region-data start end)
 → (or/c (is-a?/c editor-data%) #f)
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?
```

Gets extra data associated with a given region. See §1.3.2.1.2 “Editor Data” for more information.

This method is *not* called when the whole editor is saved to a file. In such cases, the information can be stored in the header or footer; see §1.3.2.2 “Global Data: Headers and Footers”.

This method is meant to be overridden; the default set-region-data method does not

store information to be retrieved by this method.

```
(send a-text get-revision-number)
 → (and/c real? (not/c negative?))
```

Returns an inexact number that increments every time the editor is changed in one of the following ways: a snip is inserted (see `after-insert`), a snip is deleted (see `after-delete`), a snip is split (see `after-split-snip`), snips are merged (see `after-merge-snips`), or a snip changes its count (which is rare; see `recounted`).

```
(send a-text get-snip-position snip)
 → (or/c exact-nonnegative-integer? #f)
  snip : (is-a?/c snip%)
```

Returns the starting position of a given snip or `#f` if the snip is not in this editor.

```
(send a-text get-snip-position-and-location  snip
                                             pos
                                            [x
                                             y])   → boolean?
  snip : (is-a?/c snip%)
  pos : (or/c (box/c exact-nonnegative-integer?) #f)
  x : (or/c (box/c real?) #f) = #f
  y : (or/c (box/c real?) #f) = #f
```

Gets a snip's position and top left location in editor coordinates. The return value is `#t` if the snip is found, `#f` otherwise.

The *pos* box is filled with starting position of *snip*, unless *pos* is `#f`. The *x* box is filled with left location of *snip* in editor coordinates, unless *x* is `#f`. The *y* box is filled with top location of *snip* in editor coordinates, unless *y* is `#f`.

When location information is requested: The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see `refresh-delayed?`).

```
(send a-text get-start-position) → exact-nonnegative-integer?
```

Returns the starting position of the current selection. See also `get-position`.

```
(send a-text get-styles-sticky) → boolean?
```

In the normal mode for a text editor, style settings are sticky. With sticky styles, when a string or character is inserted into an editor, it gets the style of the snip preceding the insertion point (or the snip that includes the insertion point if text is inserted into an exiting string snip). Alternatively, if `change-style` is called to set the style at the caret position (when it is not a range), then the style is remembered; if the editor is not changed before text is inserted at the caret, then the text gets the remembered style.

With non-sticky styles, text inserted into an editor always gets the style in the editor's style list named by `default-style-name`.

See also `set-styles-sticky`.

```
(send a-text get-tabs [length
                       tab-width
                       in-units]) → (listof real?)
  length : (or/c (box/c exact-nonnegative-integer?) #f) = #f
  tab-width : (or/c (box/c real?) #f) = #f
  in-units : (or/c (box/c any/c) #f) = #f
```

Returns the current tab-position array as a list.

The *length* box is filled with the length of the tab array (and therefore the returned list), unless *length* is `#f`. The *tab-width* box is filled with the width used for tabs past the end of the tab array, unless *tab-width* is `#f`. The *in-units* box is filled with `#t` if the tabs are specified in canvas units or `#f` if they are specified in space-widths, unless *in-units* is `#f`.

See also `set-tabs`.

```
(send a-text get-text [start
                       end
                       flattened?
                       force-cr?]) → string?
  start : exact-nonnegative-integer? = 0
  end : (or/c exact-nonnegative-integer? 'eof) = 'eof
  flattened? : any/c = #f
  force-cr? : any/c = #f
```

Gets the text from *start* to *end*. If *end* is `'eof`, then the contents are returned from *start* until the end of the editor.

If *flattened?* is not `#f`, then flattened text is returned. See §1.3.4 "Flattened Text" for a discussion of flattened vs. non-flattened text.

If *force-cr?* is not `#f` and *flattened?* is not `#f`, then automatic carriage returns (from word-wrapping) are written into the return string as real carriage returns.

```
(send a-text get-top-line-base)
 → (and/c real? (not/c negative?))
```

Returns the distance from the top of the editor to the alignment baseline of the top line. This method is primarily used when an editor is an item within another editor.

The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”). For `text%` objects, calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see `refresh-delayed?`).

```
(send a-text get-visible-line-range  start
                                     end
                                    [all?]) → void?
  start : (or/c (box/c exact-nonnegative-integer?) #f)
  end : (or/c (box/c exact-nonnegative-integer?) #f)
  all? : any/c = #t
```

Returns the range of lines which are currently visible (or partially visible) to the user. Lines are numbered starting with `0`.

The *start* box is filled with first line visible to the user, unless *start* is `#f`. The *end* box is filled with last line visible to the user, unless *end* is `#f`.

If the editor is displayed by multiple canvases and *all?* is `#t`, then the computed range includes all visible lines in all displays. Otherwise, the range includes only the visible lines in the current display.

The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see `refresh-delayed?`).

```
(send a-text get-visible-position-range  start
                                         end
                                        [all?]) → void?
  start : (or/c (box/c exact-nonnegative-integer?) #f)
  end : (or/c (box/c exact-nonnegative-integer?) #f)
  all? : any/c = #t
```

Returns the range of positions that are currently visible (or partially visible) to the user.

The *start* box is filled with first position visible to the user, unless *start* is `#f`. The *end* box is filled with last position visible to the user, unless *end* is `#f`.

If the editor is displayed by multiple canvases and *all?* is `#t`, then the computed range

includes all visible positions in all displays. Otherwise, the range includes only the visible positions in the current display.

The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?).

```
(send a-text get-wordbreak-map)
 → (or/c (is-a?/c editor-wordbreak-map%) #f)
```

Returns the wordbreaking map that is used by the standard wordbreaking function. See set-wordbreak-map and editor-wordbreak-map% for more information.

```
(send a-text hide-caret hide?) → void?
  hide? : any/c
```

Determines whether the caret is shown when the editor has the keyboard focus.

If *hide?* is not #f, then the caret or selection hiliting will not be drawn for the editor. The editor can still own the keyboard focus, but no caret will be drawn to indicate the focus.

See also caret-hidden? and lock.

```
(send a-text insert str start [end scroll-ok?]) → void?
  str : string?
  start : exact-nonnegative-integer?
  end : (or/c exact-nonnegative-integer? 'same) = 'same
  scroll-ok? : any/c = #t
(send a-text insert  n
                     str
                     start
                    [end
                     scroll-ok?]) → void?
  n : (and/c exact-nonnegative-integer?
             (<=/c (string-length str)))
  str : string?
  start : exact-nonnegative-integer?
  end : (or/c exact-nonnegative-integer? 'same) = 'same
  scroll-ok? : any/c = #t
(send a-text insert str) → void?
  str : string?
(send a-text insert n str) → void?
  n : (and/c exact-nonnegative-integer?
             (<=/c (string-length str)))
```

```
  str : string?
(send a-text insert  snip
                     start
                    [end
                     scroll-ok?]) → void?
  snip : (is-a?/c snip%)
  start : exact-nonnegative-integer?
  end : (or/c exact-nonnegative-integer? 'same) = 'same
  scroll-ok? : any/c = #t
(send a-text insert snip) → void?
  snip : (is-a?/c snip%)
(send a-text insert char) → void?
  char : char?
(send a-text insert char start [end]) → void?
  char : char?
  start : exact-nonnegative-integer?
  end : (or/c exact-nonnegative-integer? 'same) = 'same
```

Overrides `insert` in `editor<%>`.

Inserts text or a snip into *a-text* at position *start*. If *n* is provided, the only the first *n* characters of *str* are inserted.

When a *snip* is provided: The snip cannot be inserted into multiple editors or multiple times within a single editor. As the snip is inserted, its current style is converted to one in the editor's style list; see also `convert`.

When a *char* is provided: Multiple calls to the character-inserting method are grouped together for undo purposes, since this case of the method is typically used for handling user keystrokes. However, this undo-grouping feature interferes with the undo grouping performed by `begin-edit-sequence` and `end-edit-sequence`, so the string-inserting method should be used instead during undoable edit sequences.

When *start* is not provided, the current selection start is used. If the current selection covers a range of items, then *char* replaces the selected text. The selection's start and end positions are moved to the end of the inserted character.

For a case where *end* is not provided and has no default, the current selection end is used. Otherwise, if *end* is not `'same`, then the inserted value replaces the region from *start* to *end*, and the selection is left at the end of the inserted text. Otherwise, if the insertion position is before or equal to the selection's start/end position, then the selection's start/end position is incremented by the length of *str*.

If *scroll-ok?* is not `#f` and *start* is the same as the current selection's start position, then the editor's display is scrolled to show the new selection position.

See also `get-styles-sticky`.

```
(send a-text kill [time]) → void?
  time : exact-integer? = 0
(send a-text kill time start end) → void?
  time : exact-integer?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?
```

Overrides `kill` in `editor<%>`.

Cuts the text in the given region. If *start* and *end* are not supplied, then the selected region plus all whitespace to the end of line is cut; the newline is also cut if only whitespace exists between the selection and the end of line.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an `exn:fail:contract` exception is raised.

```
(send a-text last-line) → exact-nonnegative-integer?
```

Returns the number of the last line in the editor. Lines are numbered starting with `0`, so this is one less than the number of lines in the editor.

See also `paragraph-start-position`, which operates on paragraphs (determined by explicit newline characters) instead of lines (determined by both explicit newline characters and automatic line-wrapping).

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see `refresh-delayed?`). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for `line-start-position` (which handles specially the case of no display when the editor has a maximum width).

```
(send a-text last-paragraph) → exact-nonnegative-integer?
```

Returns the number of the last paragraph in the editor. Paragraphs are numbered starting with `0`, so this is one less than the number of paragraphs in the editor.

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see `refresh-delayed?`).

```
(send a-text last-position) → exact-nonnegative-integer?
```

Returns the last selection position in the editor. This is also the number of items in the editor.

```
(send a-text line-end-position  line
                                [visible?])
 → exact-nonnegative-integer?
  line : exact-nonnegative-integer?
  visible? : any/c = #t
```

Returns the last position of a given line. Lines are numbered starting with 0.

If there are fewer than *line*-1 lines, the end of the last line is returned. If *line* is less than 0, then the end of the first line is returned.

If the line ends with invisible items (such as a carriage return) and *visible?* is not #f, the first position before the invisible items is returned.

See also paragraph-start-position, which operates on paragraphs (determined by explicit newline characters) instead of lines (determined by both explicit newline characters and automatic line-wrapping).

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for line-start-position (which handles specially the case of no display when the editor has a maximum width).

```
(send a-text line-length i) → exact-nonnegative-integer?
  i : exact-nonnegative-integer?
```

Returns the number of items in a given line. Lines are numbered starting with 0.

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for line-start-position (which handles specially the case of no display when the editor has a maximum width).

```
(send a-text line-location line [top?]) → real?
  line : exact-nonnegative-integer?
  top? : any/c = #t
```

Given a line number, returns the location of the line. Lines are numbered starting with 0.

If *top?* is not #f, the location for the top of the line is returned; otherwise, the location for the bottom of the line is returned.

See also paragraph-start-position, which operates on paragraphs (determined by explicit newline characters) instead of lines (determined by both explicit newline characters and automatic line-wrapping).

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?).

```
(send a-text line-paragraph start) → exact-nonnegative-integer?
  start : exact-nonnegative-integer?
```

Returns the paragraph number of the paragraph containing the line. Lines are numbered starting with 0. Paragraphs are numbered starting with 0.

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for line-start-position (which handles specially the case of no display when the editor has a maximum width).

```
(send a-text line-start-position  line
                                  [visible?])
 → exact-nonnegative-integer?
  line : exact-nonnegative-integer?
  visible? : any/c = #t
```

Returns the first position of the given line. Lines are numbered starting with 0.

If there are fewer than *line*-1 lines, the start of the last line is returned. If *line* is less than 0, then the start of the first line is returned.

If the line starts with invisible items and *visible?* is not #f, the first position past the invisible items is returned.

See also paragraph-start-position, which operates on paragraphs (determined by explicit newline characters) instead of lines (determined by both explicit newline characters and automatic line-wrapping).

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?).

To calculate lines, if the following are true:

- the editor is not displayed (see §1.3.1 "Editor Structure and Terminology"),

- a maximum width is set for the editor, and
- the editor has never been viewed

then this method ignores the editor's maximum width and any automatic line breaks it might imply. If the first two of the above conditions are true and the editor was *formerly* displayed, this method uses the line breaks from the most recent display of the editor. (Insertions or deletions since the display shift line breaks within the editor in the same way as items.)

```
(send a-text move-position  code
                            [extend?
                             kind])  → void?
  code : (or/c 'home 'end 'right 'left 'up 'down)
  extend? : any/c = #f
  kind : (or/c 'simple 'word 'page 'line) = 'simple
```

Moves the current selection.

The possible values for *code* are:

- `'home` — go to start of file
- `'end` — go to end of file
- `'right` — move right
- `'left` — move left
- `'up` — move up
- `'down` — move down

If *extend?* is not `#f`, the selection range is extended instead of moved. If anchoring is on (see `get-anchor` and `set-anchor`), then *extend?* is effectively forced to `#t`.

The possible values for *kind* are:

- `'simple` — move one item or line
- `'word` — works with `'right` or `'left`
- `'page` — works with `'up` or `'down`
- `'line` — works with `'right` or `'left`; moves to the start or end of the line

See also set-position.

```
(send a-text on-change-style start len) → void?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?
```

Refine this method with augment.

Specification: Called before the style is changed in a given range of the editor, after can-change-style? is called to verify that the change is ok. The after-change-style method is guaranteed to be called after the change has completed.

The editor is internally locked for writing during a call to this method (see also §1.3.8 "Internal Editor Locks"). Use after-change-style to modify the editor, if necessary.

See also on-edit-sequence.

Default implementation: Does nothing.

```
(send a-text on-default-char event) → void?
  event : (is-a?/c key-event%)
```

Overrides on-default-char in editor<%>.

Handles the following:

- Delete and Backspace — calls delete.
- The arrow keys, Page Up, Page Down, Home, and End (including shifted versions) — moves the selection position with move-position.
- Any other character in the range (integer->char 32) to (integer->char 255) — inserts the character into the editor.

Note that an editor's editor-canvas% normally handles mouse wheel events (see also on-char).

```
(send a-text on-default-event event) → void?
  event : (is-a?/c mouse-event%)
```

Overrides on-default-event in editor<%>.

Tracks clicks on a clickback (see set-clickback) of changes the selection. Note that on-event dispatches to a caret-owning snip and detects a click on an event-handling snip before calling to this method.

- Clicking on a clickback region starts clickback tracking. See `set-clickback` for more information. Moving over a clickback changes the shape of the mouse cursor.
- Clicking anywhere else moves the caret to the closest position between items. Shift-clicking extends the current selection.
- Dragging extends the selection, scrolling if possible when the selection is dragged outside the editor's visible region.

```
(send a-text on-delete start len) → void?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?
```

Refine this method with `augment`.

Specification: Called before a range is deleted from the editor, after `can-delete?` is called to verify that the deletion is ok. The `after-delete` method is guaranteed to be called after the delete has completed.

The *start* argument specifies the starting position of the range to delete. The *len* argument specifies number of items to delete (so *start*+*len* is the ending position of the range to delete).

The editor is internally locked for writing during a call to this method (see also §1.3.8 "Internal Editor Locks"). Use `after-delete` to modify the editor, if necessary.

See also `on-edit-sequence`.

Default implementation: Does nothing.

```
(send a-text on-insert start len) → void?
  start : exact-nonnegative-integer?
  len : exact-nonnegative-integer?
```

Refine this method with `augment`.

Specification: Called before items are inserted into the editor, after `can-insert?` is called to verify that the insertion is ok. The `after-insert` method is guaranteed to be called after the insert has completed.

The *start* argument specifies the position of the insert. The *len* argument specifies the total length (in positions) of the items to be inserted.

The editor is internally locked for writing during a call to this method (see also §1.3.8 "Internal Editor Locks"). Use `after-insert` to modify the editor, if necessary.

See also `on-edit-sequence`.

Default implementation: Does nothing.

`(send a-text on-new-string-snip) → (is-a?/c string-snip%)`

Specification: Called by `insert` when a string or character is inserted into the editor, this method creates and returns a new instance of `string-snip%` to store inserted text. The returned string snip is empty (i.e., its count is zero).

Default implementation: Returns a `string-snip%` instance.

`(send a-text on-new-tab-snip) → (is-a?/c tab-snip%)`

Specification: Creates and returns a new instance of `tab-snip%` to store an inserted tab. The returned tab snip is empty (i.e., its count is zero).

Default implementation: Returns a `tab-snip%` instance.

`(send a-text on-reflow) → void?`

Refine this method with `augment`.

Specification: Called after locations have changed and are recomputed for the editor. *Default implementation:* Does nothing.

`(send a-text on-set-size-constraint) → void?`

Refine this method with `augment`.

Specification: Called before the editor's maximum or minimum height or width is changed, after `can-set-size-constraint?` is called to verify that the change is ok. The `after-set-size-constraint` method is guaranteed to be called after the change has completed.

(This callback method is provided because setting an editor's maximum width may cause lines to be re-flowed with soft carriage returns.)

See also `on-edit-sequence`.

Default implementation: Does nothing.

```
(send a-text paragraph-end-line paragraph)
 → exact-nonnegative-integer?
  paragraph : exact-nonnegative-integer?
```

Returns the ending line of a given paragraph. Paragraphs are numbered starting with `0`. Lines

are numbered starting with 0.

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for line-start-position (which handles specially the case of no display when the editor has a maximum width).

```
(send a-text paragraph-end-position  paragraph
                                     [visible?])
 → exact-nonnegative-integer?
  paragraph : exact-nonnegative-integer?
  visible? : any/c = #f
```

Returns the ending position of a given paragraph. Paragraphs are numbered starting with 0.

If there are fewer than *paragraph*-1 paragraphs, the end of the last paragraph is returned. If *paragraph* is less than 0, then the end of the first paragraph is returned.

If the paragraph ends with invisible items (such as a carriage return) and *visible?* is not #f, the first position before the invisible items is returned.

```
(send a-text paragraph-start-line paragraph)
 → exact-nonnegative-integer?
  paragraph : exact-nonnegative-integer?
```

Returns the starting line of a given paragraph. If *paragraph* is greater than the highest-numbered paragraph, then the editor's end position is returned. Paragraphs are numbered starting with 0. Lines are numbered starting with 0.

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see refresh-delayed?). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for line-start-position (which handles specially the case of no display when the editor has a maximum width).

```
(send a-text paragraph-start-position  paragraph
                                       [visible?])
 → exact-nonnegative-integer?
  paragraph : exact-nonnegative-integer?
  visible? : any/c = #f
```

Returns the starting position of a given paragraph. Paragraphs are numbered starting with 0.

If there are fewer than *paragraph*-1 paragraphs, the start of the last paragraph is returned.

If the paragraph starts with invisible items and *visible?* is not #f, the first position past the invisible items is returned.

```
(send a-text paste [time start end]) → void?
  time : exact-integer? = 0
  start : (or/c exact-nonnegative-integer? 'start 'end) = 'start
  end : (or/c exact-nonnegative-integer? 'same) = 'same
```

Overrides paste in editor<%>.

Pastes into the specified range. If *start* is 'start, then the current selection start position is used. If *start* is 'end, then the current selection end position is used. If *end* is 'same, then *start* is used for *end*, unless *start* is 'start, in which case the current selection end position is used.

See §1.3.6 "Cut and Paste Time Stamps" for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an exn:fail:contract exception is raised.

```
(send a-text paste-next) → void?
```

Editors collectively maintain a copy ring that holds up to 30 previous copies (and cuts) among the editors. When it is called as the next method on an editor after a paste, the paste-next method replaces the text from a previous paste with the next data in the copy ring, incrementing the ring pointer so that the next paste-next pastes in even older data.

It is a copy "ring" because the ring pointer wraps back to the most recent copied data after the oldest remembered data is pasted. Any cut, copy, or (regular) paste operation resets the copy ring pointer back to the beginning.

If the previous operation on the editor was not a paste, calling paste-next has no effect.

```
(send a-text paste-x-selection time
                              [start
                               end]) → void?
  time : exact-integer?
  start : (or/c exact-nonnegative-integer? 'start 'end) = 'start
  end : (or/c exact-nonnegative-integer? 'same) = 'same
```

Overrides paste-x-selection in editor<%>.

Pastes into the specified range. If *start* is 'start, then the current selection start position is used. If *start* is 'end, then the current selection end position is used. If *end* is 'same, then *start* is used for *end*, unless *start* is 'start, in which case the current selection end position is used.

See §1.3.6 “Cut and Paste Time Stamps” for a discussion of the *time* argument. If *time* is outside the platform-specific range of times, an `exn:fail:contract` exception is raised.

```
(send a-text position-line start [at-eol?])
 → exact-nonnegative-integer?
  start : exact-nonnegative-integer?
  at-eol? : any/c = #f
```

Returns the line number of the line containing a given position. Lines are numbered starting with `0`.

See also `paragraph-start-position`, which operates on paragraphs (determined by explicit newline characters) instead of lines (determined by both explicit newline characters and automatic line-wrapping).

See §1.3.3 “End of Line Ambiguity” for a discussion of *at-eol?*.

Calling this method may force the recalculation of location information if a maximum width is set for the editor, even if the editor currently has delayed refreshing (see `refresh-delayed?`). If the editor is not displayed and the editor has a maximum width, line breaks are calculated as for `line-start-position` (which handles specially the case of no display when the editor has a maximum width).

```
(send a-text position-location  start
                                [x
                                 y
                                 top?
                                 at-eol?
                                 whole-line?]) → void?
  start : exact-nonnegative-integer?
  x : (or/c (box/c real?) #f) = #f
  y : (or/c (box/c real?) #f) = #f
  top? : any/c = #t
  at-eol? : any/c = #f
  whole-line? : any/c = #f
```

Returns the location of a given position. See also `position-locations`.

The *x* box is filled with the x-location of the position *start* in editor coordinates, unless *x* is `#f`. The *y* box is filled with the y-location (top or bottom; see below) of the position *start* in editor coordinates, unless *y* is `#f`.

See §1.3.3 “End of Line Ambiguity” for a discussion of *at-eol?*.

If *top?* is not `#f`, the top coordinate of the location is returned, otherwise the bottom coor-

dinate of the location is returned.

The top *y* location may be different for different positions within a line when different-sized graphic objects are used. If *whole-line?* is not #f, the minimum top location or maximum bottom location for the whole line is returned in *y*.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?).

```
(send a-text position-locations  start
                                 [top-x
                                  top-y
                                  bottom-x
                                  bottom-y
                                  at-eol?
                                  whole-line?]) → void?
  start : exact-nonnegative-integer?
  top-x : (or/c (box/c real?) #f) = #f
  top-y : (or/c (box/c real?) #f) = #f
  bottom-x : (or/c (box/c real?) #f) = #f
  bottom-y : (or/c (box/c real?) #f) = #f
  at-eol? : any/c = #f
  whole-line? : any/c = #f
```

Like position-location, but returns both the "top" and "bottom" results at once.

The result is only valid when the editor is displayed (see §1.3.1 "Editor Structure and Terminology"). Calling this method may force the recalculation of location information, even if the editor currently has delayed refreshing (see refresh-delayed?).

```
(send a-text position-paragraph  start
                                 [at-eol?])
 → exact-nonnegative-integer?
  start : exact-nonnegative-integer?
  at-eol? : any/c = #f
```

See §1.3.3 "End of Line Ambiguity" for a discussion of *at-eol?*.

Returns the paragraph number of the paragraph containing a given position.

```
(send a-text read-from-file  stream
                             start
                            [overwrite-styles?]) → boolean?
```

```
  stream : (is-a?/c editor-stream-in%)
  start : (or/c exact-nonnegative-integer? 'start)
  overwrite-styles? : any/c = #f
```

Extends `read-from-file` in `editor<%>`.

New data is inserted at the position indicated by *start*, or at the current position if *start* is `'start`.

```
(send a-text remove-clickback start end) → void?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?
```

Removes all clickbacks installed for exactly the range *start* to *end*. See also §1.3.7 "Clickbacks".

```
(send a-text scroll-to-position  start
                                 [at-eol?
                                  end
                                  bias])  → boolean?
  start : exact-nonnegative-integer?
  at-eol? : any/c = #f
  end : (or/c exact-nonnegative-integer? 'same) = 'same
  bias : (or/c 'start 'end 'none) = 'none
```

Scrolls the editor so that a given position is visible.

If *end* is `'same` or equal to *start*, then position *start* is made visible. See §1.3.3 "End of Line Ambiguity" for a discussion of *at-eol?*.

If *end* is not `'same` and not the same as *start*, then the range *start* to *end* is made visible and *at-eol?* is ignored.

When the specified range cannot fit in the visible area, *bias* indicates which end of the range to display. When *bias* is `'start`, then the start of the range is displayed. When *bias* is `'end`, then the end of the range is displayed. Otherwise, *bias* must be `'none`.

If the editor is scrolled, then the editor is redrawn and the return value is `#t`; otherwise, the return value is `#f`. If refreshing is delayed (see `refresh-delayed?`), then the scroll request is saved until the delay has ended. The scroll is performed (immediately or later) by calling `scroll-editor-to`.

Scrolling is disallowed when the editor is internally locked for reflowing (see also §1.3.8 "Internal Editor Locks").

The system may scroll the editor without calling this method. For example, a canvas dis-

playing an editor might scroll the editor to handle a scrollbar event.

```
(send a-text set-anchor on?) → void?
  on? : any/c
```

Turns anchoring on or off. This method can be overridden to affect or detect changes in the anchor state. See also get-anchor.

If *on?* is not #f, then the selection will be automatically extended when cursor keys are used (or, more generally, when move-position is used to move the selection), otherwise anchoring is turned off. Anchoring is automatically turned off if the user does anything besides cursor movements.

```
(send a-text set-autowrap-bitmap bitmap)
 → (or/c (is-a?/c bitmap%) #f)
  bitmap : (or/c (is-a?/c bitmap%) #f)
```

Sets the bitmap that is drawn at the end of a line when it is automatically line-wrapped.

If *bitmap* is #f, no autowrap indicator is drawn (this is the default). The previously used bitmap (possibly #f) is returned.

The bitmap will not be modified. It may be selected into a bitmap-dc% object, but it will be selected out if this method is called again.

Setting the bitmap is disallowed when the editor is internally locked for reflowing (see also §1.3.8 “Internal Editor Locks”).

```
(send a-text set-between-threshold threshold) → void?
  threshold : (and/c real? (not/c negative?))
```

Sets the graphical distance used to determine the meaning of a user click. If a click falls within *threshold* of a position between two items, then the click registers on the space between the items rather than on either item.

See also get-between-threshold.

```
(send a-text set-clickback  start
                            end
                            f
                           [hilite-delta
                            call-on-down?]) → void?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?
```

```
f : (-> (is-a?/c text%)
        exact-nonnegative-integer?
        exact-nonnegative-integer?
        any)
hilite-delta : (or/c (is-a?/c style-delta%) #f) = #f
call-on-down? : any/c = #f
```

Installs a clickback for a given region. If a clickback is already installed for an overlapping region, this clickback takes precedence.

The callback procedure *f* is called when the user selects the clickback. The arguments to *f* are this editor and the starting and ending range of the clickback.

The *hilite-delta* style delta is applied to the clickback text when the user has clicked and is still holding the mouse over the clickback. If *hilite-delta* is `#f`, then the clickback region's style is not changed when it is being selected.

If *call-on-down?* is not `#f`, the clickback is called immediately when the user clicks the mouse button down, instead of after a mouse-up event. The *hilite-delta* argument is not used in this case.

See also §1.3.7 "Clickbacks".

```
(send a-text set-file-format format) → void?
  format : (or/c 'standard 'text 'text-force-cr)
```

Set the format of the file saved from this editor.

The legal formats are:

- `'standard` — a standard editor file
- `'text` — a text file
- `'text-force-cr` — a text file; when writing, change automatic newlines (from word-wrapping) into real carriage returns

The file format of an editor can be changed by the system in response to file loading and saving method calls, and such changes do not go through this method; use `on-load-file` and `on-save-file` to monitor such file format changes.

```
(send a-text set-line-spacing space) → void?
  space : (and/c real? (not/c negative?))
```

Sets the spacing inserted by the editor between each line. This spacing is included in the reported height of each line.

```
(send a-text set-overwrite-mode on?) → void?
  on? : any/c
```

Enables or disables overwrite mode. See get-overwrite-mode. This method can be overridden to affect or detect changes in the overwrite mode.

```
(send a-text set-paragraph-alignment paragraph
                                     alignment) → void?
  paragraph : exact-nonnegative-integer?
  alignment : (or/c 'left 'center 'right)
```

Sets a paragraph-specific horizontal alignment. The alignment is only used when the editor has a maximum width, as set with set-max-width. Paragraphs are numbered starting with 0.

This method is experimental. It works reliably only when the paragraph is not merged or split. Merging or splitting a paragraph with alignment settings causes the settings to be transferred unpredictably (although other paragraphs in the editor can be safely split or merged). If the last paragraph in an editor is empty, settings assigned to it are ignored.

```
(send a-text set-paragraph-margins paragraph
                                   first-left
                                   left
                                   right)     → void?
  paragraph : exact-nonnegative-integer?
  first-left : (and/c real? (not/c negative?))
  left : (and/c real? (not/c negative?))
  right : (and/c real? (not/c negative?))
```

Sets a paragraph-specific margin. Paragraphs are numbered starting with 0.

The first line of the paragraph is indented by *first-left* points within the editor. If the paragraph is line-wrapped (when the editor has a maximum width), subsequent lines are indented by *left* points. If the editor has a maximum width, the paragraph's maximum width for line-wrapping is *right* points smaller than the editor's maximum width.

This method is experimental. See set-paragraph-alignment for more information.

```
(send a-text set-position  start
                          [end
                           at-eol?
                           scroll?
                           seltype]) → void?
```

```
  start : exact-nonnegative-integer?
  end : (or/c exact-nonnegative-integer? 'same) = 'same
  at-eol? : any/c = #f
  scroll? : any/c = #t
  seltype : (or/c 'default 'x 'local) = 'default
```

Sets the current selection in the editor.

If *end* is `'same` or less than or equal to *start*, the current start and end positions are both set to *start*. Otherwise the given range is selected.

See §1.3.3 "End of Line Ambiguity" for a discussion of *at-eol?*. If *scroll?* is not `#f`, then the display is scrolled to show the selection if necessary.

The *seltype* argument is only used when the X Window System selection mechanism is enabled. The possible values are:

- `'default` — if this window has the keyboard focus and given selection is non-empty, make it the current X selection
- `'x` — if the given selection is non-empty, make it the current X selection
- `'local` — do not change the current X selection

Setting the position is disallowed when the editor is internally locked for reflowing (see also §1.3.8 "Internal Editor Locks").

The system may change the selection in an editor without calling this method (or any visible method).

See also `editor-set-x-selection-mode`.

```
(send a-text set-position-bias-scroll  bias
                                       start
                                      [end
                                       ateol?
                                       scroll?
                                       seltype]) → void?
  bias : (or/c 'start-only 'start 'none 'end 'end-only)
  start : exact-nonnegative-integer?
  end : (or/c exact-nonnegative-integer? 'same) = 'same
  ateol? : any/c = #f
  scroll? : any/c = #t
  seltype : (or/c 'default 'x 'local) = 'default
```

Like `set-position`, but a scrolling bias can be specified.

The possible values for *bias* are:

- 'start-only — only insure that the starting position is visible
- 'start — if the range doesn't fit in the visible area, show the starting position
- 'none — no special scrolling instructions
- 'end — if the range doesn't fit in the visible area, show the ending position
- 'end-only — only insure that the ending position is visible

See also scroll-to-position.

```
(send a-text set-region-data start end data) → void?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer?
  data : (is-a?/c editor-data%)
```

Specification: Sets extra data associated with a given region. See §1.3.2.1.2 "Editor Data" and get-region-data for more information.

This method is meant to be overridden in combination with get-region-data .

Default implementation: Does nothing.

```
(send a-text set-styles-sticky sticky?) → void?
  sticky? : any/c
```

See get-styles-sticky for information about sticky styles.

```
(send a-text set-tabs  tabs
                      [tab-width
                       in-units?]) → void?
  tabs : (listof real?)
  tab-width : real? = 20
  in-units? : any/c = #t
```

Sets the tabbing array for the editor.

The *tabs* list determines the tabbing array. The tabbing array specifies the x-locations where each tab occurs. Tabs beyond the last specified tab are separated by a fixed amount *tab-width*. If *in-units?* is not #f, then tabs are specified in canvas units; otherwise, they are specified as a number of spaces. (If tabs are specified in spaces, then the graphic tab positions will change with the font used for the tab.)

Setting tabs is disallowed when the editor is internally locked for reflowing (see also §1.3.8 “Internal Editor Locks”).

```
(send a-text set-wordbreak-func f) → void?
  f : ((is-a?/c text%) (or/c (box/c exact-nonnegative-integer?) #f)
                       (or/c (box/c exact-nonnegative-integer?) #f)
                       symbol?
       . -> . any)
```

Sets the word-breaking function for the editor. For information about the arguments to the word-breaking function, see `find-wordbreak`.

The standard wordbreaking function uses the editor’s `editor-wordbreak-map%` object to determine which characters break a word. See also `editor-wordbreak-map%` and `set-wordbreak-map`.

Since the wordbreak function will be called when line breaks are being determined (in an editor that has a maximum width), there is a constrained set of `text%` methods that the wordbreak function is allowed to invoke. It cannot invoke a member function that uses information about locations or lines (which are identified in this manual with “The result is only valid when the editor is displayed (see §1.3.1 “Editor Structure and Terminology”).”), but it can still invoke member functions that work with snips and items.

```
(send a-text set-wordbreak-map map) → void?
  map : (or/c (is-a?/c editor-wordbreak-map%) #f)
```

Sets the wordbreaking map that is used by the standard wordbreaking function. See `editor-wordbreak-map%` for more information.

If *map* is `#f`, then the standard map (`the-editor-wordbreak-map`) is used.

```
(send a-text split-snip pos) → void?
  pos : exact-nonnegative-integer?
```

Given a position, splits the snip that includes the position (if any) so that the position is between two snips. The snip may refuse to split, although none of the built-in snip classes will ever refuse.

Splitting a snip is disallowed when the editor is internally locked for reflowing (see also §1.3.8 “Internal Editor Locks”).

```
(send a-text write-to-file stream [start end]) → boolean?
  stream : (is-a?/c editor-stream-out%)
  start : exact-nonnegative-integer? = 0
```

end : (or/c exact-nonnegative-integer? 'eof) = 'eof

Extends write-to-file in editor<%>.

If *start* is 0 and *end* is 'eof negative, then the entire contents are written to the stream. If *end* is 'eof, then the contents are written from *start* until the end of the editor. Otherwise, the contents of the given range are written.

2.6 Editor Functions

(add-editor-keymap-functions *keymap*) → void?
keymap : (is-a?/c keymap%)

Given a keymap% object, the keymap is loaded with mappable functions that apply to all editor<%> objects:

- "copy-clipboard"
- "copy-append-clipboard"
- "cut-clipboard"
- "cut-append-clipboard"
- "paste-clipboard"
- "paste-x-selection"
- "delete-selection"
- "clear-selection"
- "undo"
- "redo"
- "select-all"

(add-pasteboard-keymap-functions *keymap*) → void?
keymap : (is-a?/c keymap%)

Given a keymap% object, the table is loaded with mappable functions that apply to pasteboard% objects. Currently, there are no such functions.

See also add-editor-keymap-functions.

```
(add-text-keymap-functions keymap) → void?
  keymap : (is-a?/c keymap%)
```

Given a keymap% object, the table is loaded with functions that apply to all text% objects:

- "forward-character"
- "backward-character"
- "previous-line"
- "next-line"
- "previous-page"
- "next-page"
- "forward-word"
- "backward-word"
- "forward-select"
- "backward-select"
- "select-down"
- "select-up"
- "select-page-up"
- "select-page-down"
- "forward-select-word"
- "backward-select-word"
- "beginning-of-file"
- "end-of-file"
- "beginning-of-line"
- "end-of-line"
- "select-to-beginning-of-file"
- "select-to-end-of-file"
- "select-to-beginning-of-line"
- "select-to-end-of-line"

- "copy-clipboard"
- "copy-append-clipboard"
- "cut-clipboard"
- "cut-append-clipboard"
- "paste-clipboard"
- "paste-x-selection"
- "delete-selection"
- "delete-previous-character"
- "delete-next-character"
- "clear-selection"
- "delete-to-end-of-line"
- "delete-next-word"
- "delete-previous-word"
- "delete-line"
- "undo"
- "redo"

See also add-editor-keymap-functions.

```
(append-editor-font-menu-items menu) → void?
  menu : (or/c (is-a?/c menu%) (is-a?/c popup-menu%))
```

Appends menu items to a given menu (not a popup menu) to implement a standard set of font-manipulation operations, such as changing the font face or style. The callback for each menu item uses get-edit-target-object in top-level-window<%> (finding the frame by following a chain of parents until a frame is reached); if the result is an editor<%> object, change-style in editor<%> is called on the editor.

```
(append-editor-operation-menu-items  menu
                                     [text-only?]) → void?
  menu : (or/c (is-a?/c menu%) (is-a?/c popup-menu%))
  text-only? : any/c = #t
```

Appends menu items to a given menu (not a popup menu) to implement the standard editor operations, such as cut and paste. The callback for each menu item uses `get-edit-target-object` in `top-level-window<%>` (finding the frame by following a chain of parents until a frame is reached); if the result is an `editor<%>` object, `do-edit-operation` in `editor<%>` is called on the editor.

If *text-only?* is `#f`, then menu items that insert non-text snips (such as Insert Image...) are appended to the menu.

```
(current-text-keymap-initializer)
 → ((is-a?/c keymap%) . -> . any/c)
(current-text-keymap-initializer proc) → void?
  proc : ((is-a?/c keymap%) . -> . any/c)
```

Parameter that specifies a keymap-initialization procedure. This procedure is called to initialize the keymap of a `text-field%` object or a `text%` object created by `graphical-read-eval-print-loop`.

The initializer takes a keymap object and returns nothing. The default initializer chains the given keymap to an internal keymap that implements standard text editor keyboard and mouse bindings for cut, copy, paste, undo, and select-all. The right mouse button is mapped to popup an edit menu when the button is released. Under X, start-of-line (Ctl-A) and end-of-line (Ctl-E) are also mapped.

```
(editor-set-x-selection-mode on) → void?
  on : any/c
```

Under X Windows, editor selections conform to the X Windows selection conventions instead of a clipboard-based convention. If *on* is `#f`, the behavior is switched to the clipboard-based convention (where copy must be explicitly requested before a paste).

```
(get-the-editor-data-class-list)
 → (is-a?/c editor-data-class-list<%>)
```

Gets the editor data class list instance for the current eventspace.

```
(get-the-snip-class-list) → (is-a?/c snip-class-list<%>)
```

Gets the snip class list instance for the current eventspace.

```
(map-command-as-meta-key on?) → void?
  on? : any/c
(map-command-as-meta-key) → boolean?
```

Determines the interpretation of `m:` for a `keymap%` mapping under Mac OS X. See also `map-function` in `keymap%`.

First case:

If *on?* is `#t`, `m:` corresponds to the Command key. If *on?* is `#f`, then `m:` corresponds to no key under Mac OS X.

Second case:

Returns `#t` if `m:` corresponds to Command, `#f` otherwise.

```
(open-input-graphical-file filename) → input-port
  filename : string?
```

Opens *filename* (in `'binary` mode) and checks whether it looks like a "graphical" file in editor format. If the file does not appear to be an editor file, the file port is returned with line counting enabled. Otherwise, the file is loaded into an editor, and the result port is created with `open-input-text-editor`.

```
(open-input-text-editor  text-editor
                        [start-position
                         end-position
                         snip-filter
                         port-name
                         expect-to-read-all?]) → input-port
  text-editor : (is-a?/c text%)
  start-position : exact-nonnegative-integer? = 0
  end-position : (or/c exact-nonnegative-integer? (one/of 'end))
               = 'end
  snip-filter : ((is-a?/c snip%) . -> . any/c) = (lambda (s) s)
  port-name : any/c = text-editor
  expect-to-read-all? : any/c = #f
```

Creates an input port that draws its content from *text-editor*. The editor content between positions *start-position* and *end-position* is the content of the port. If *end-position* is `'end`, the content runs until the end of the editor. If a snip that is not a `string-snip%` object spans *start-position* or *end-position*, the entire snip contributes to the port. If a `string-snip%` instance spans *start-position*, only the part of the snip after *start-position* contributes, and if a `string-snip%` object spans *end-position*, only the part before *end-position* contributes.

An instance of `string-snip%` in *text-editor* generates a character sequence in the resulting port. All other kinds of snips are passed to *snip-filter* to obtain a "special" value for the port. If a snip is returned as the first result from *snip-filter*, and if the snip is

an instance of readable-snip<%>, the snip generates a special value for the port through the read-special method. If *snip-filter* returns any other kind of snip, it is copied for the special result. Finally, a non-snip first result from *snip-filter* is used directly as the special result.

The *port-name* argument is used for the input port's name. The *expect-to-read-all?* argument is a performance hint; use #t if the entire port's stream will be read.

The result port must not be used if *text-editor* changes in any of the following ways: a snip is inserted (see after-insert), a snip is deleted (see after-delete), a snip is split (see after-split-snip), snips are merged (see after-merge-snips), or a snip changes its count (which is rare; see recounted). The get-revision-number method can be used to detect any of these changes.

```
(open-output-text-editor  text-editor
                          [start-position
                           special-filter
                           port-name])     → output-port
  text-editor : (is-a?/c text%)
  start-position : (or/c exact-nonnegative-integer? (one/of 'end))
                 = 'end
  special-filter : (any/c . -> . any/c) = (lambda (x) x)
  port-name : any/c = text-editor
```

Creates an output port that delivers its content to *text-editor*. The content is written to *text-editor* starting at the position *start-position*, where 'end indicates that output should start at the text editor's current end position.

If *special-filter* is provided, it is applied to any value written to the port with write-special, and the result is inserted in its place. If a special value is a snip% object, it is inserted into the editor. Otherwise, the special value is displayed into the editor.

If line counting is enabled for the resulting output port, then the port will report the line, offset from the line's start, and position within the editor at which the port writes data.

```
(read-editor-global-footer in) → boolean?
  in : (is-a?/c editor-stream-in%)
```

See read-editor-global-header. Call read-editor-global-footer even if read-editor-global-header returns #f.

```
(read-editor-global-header in) → boolean?
  in : (is-a?/c editor-stream-in%)
```

Reads data from *in* to initialize for reading editors from the stream. The return value is #t

if the read succeeds, or #f otherwise.

One or more editors can be read from the stream by calling the editor's read-from-file method. (The number of editors to be read must be known by the application beforehand.) When all editors are read, call read-editor-global-footer. Calls to read-editor-global-header and read-editor-global-footer must bracket any call to read-from-file, and only one stream at a time can be read using these methods or written using write-editor-global-header and write-editor-global-footer.

When reading from streams that span Racket versions, use read-editor-version before this procedure.

```
(read-editor-version  in
                      in-base
                      parse-format?
                     [raise-errors?]) → boolean?
  in : (is-a?/c editor-stream-in%)
  in-base : (is-a?/c editor-stream-in-base%)
  parse-format? : any/c
  raise-errors? : any/c = #t
```

Reads version information from *in-base*, where *in-base* is the base for *in*. The version information parsed from *in-base* is recorded in *in* for later version-sensitive parsing. The procedure result is true if the version information was read successfully and if the version is supported.

If *parse-format?* is true, then *in-base* is checked for an initial "WXME" format indicator. Use #f when "WXME" has been consumed already by format-dispatching code.

If *raise-errors?* is true, then an error in reading triggers an exception, instead of a #f result.

```
(text-editor-load-handler [filename]
                          expected-module-name) → any/c
  filename : path = string
  expected-module-name : (or/c symbol false/c)
```

This procedure is a load handler for use with current-load.

The handler recognizes Racket editor-format files (see §1.3.2 "File Format") and decodes them for loading. It is normally installed as GRacket starts (see §17.1 "Running Racket or GRacket").

The handler recognizes editor files by the first twelve characters of the file: WXME01⟨*digit*⟩⟨*digit*⟩ ## . Such a file is opened for loading by creating a text% object, loading the file into the object with insert-file, and then converting the editor content

into a port with open-input-text-editor. After obtaining a port in this way, the content is read in essentially the same way as by the default Racket load handler. The difference is that the editor may contain instances of readable-snip<%>, which are "read" though the snips' read-special method; see open-input-text-editor for details.

---

the-editor-wordbreak-map : (is-a?/c editor-wordbreak-map%)

See editor-wordbreak-map%.

---

the-style-list : (is-a?/c style-list%)

See style-list%.

(write-editor-global-footer *out*) → boolean?
  out : (is-a?/c editor-stream-out%)

See write-editor-global-header. Call write-editor-global-footer even if write-editor-global-header returns #f.

(write-editor-global-header *out*) → boolean?
  out : (is-a?/c editor-stream-out%)

Writes data to *out*, initializing it for writing editors to the stream. The return value is #t if the write succeeds, or #f otherwise.

One or more editors can be written to the stream by calling the editor's write-to-file method. When all editors are written, call write-editor-global-footer. Calls to write-editor-global-header and write-editor-global-footer must bracket any call to write-to-file, and only one stream at a time can be written using these methods or read using read-editor-global-header and read-editor-global-footer.

To support streams that span Racket versions, use write-editor-version before this procedure.

See also §1.3.2 "File Format".

(write-editor-version *out out-base*) → boolean?
  out : (is-a?/c editor-stream-out%)
  out-base : (is-a?/c editor-stream-out-base%)

Writes version information to *out-base* in preparation for writing editor information to the stream *out*.

The *out* argument is currently not used, but *out-base* should be the base for *out*. In the future, *out* may record information about the version for later version-sensitive output.

The result is `#t` if the write succeeded, `#f` otherwise.

2.7 WXME Decoding

```
(require wxme)
```

The `wxme` library provides tools for reading WXME `editor<%>`-format files (see §1.3.2 "File Format") without the `racket/gui` library (i.e., using `racket` instead of `gracket`).

```
(is-wxme-stream? in) → boolean?
  in : input-port?
```

Peeks from *in* and returns `#t` if it starts with the magic bytes indicating a WXME-format stream (see §1.3.2 "File Format"), `#f` otherwise.

```
(wxme-port->text-port in [close?]) → input-port?
  in : input-port?
  close? : any/c = #t
```

Takes an input port whose stream starts with WXME-format data and returns an input port that produces a text form of the WXME content, like the result of opening a WXME file in DrRacket and saving it as text.

If *close?* is true, then closing the result port close the original port.

See §2.7.1 "Snip Class Mapping" for information about the kinds of non-text content that can be read.

```
(wxme-port->port in [close? snip-filter]) → input-port?
  in : input-port?
  close? : any/c = #t
  snip-filter : (any/c . -> . any/c) = (lambda (x) x)
```

Takes an input port whose stream starts with WXME-format data and returns an input port that produces text content converted to bytes, and non-text content as "special" values (see `read-char-or-special`).

These special values produced by the new input port are different than the ones produced by reading a file into an `editor<%>` object. Instead of instances of the `snip%`, the special values are typically simple extensions of `object%`. See §2.7.1 "Snip Class Mapping" for

information about the kinds of non-text content that can be read.

If *close?* is true, then closing the result port close the original port.

The *snip-filter* procedure is applied to any special value generated for the stream, and its result is used as an alternate special value.

If a special value (possibly produced by the filter procedure) is an object implementing the readable<%> interface, then the object's read-special method is called to produce the special value.

```
(extract-used-classes in) → (listof string?) (listof string?)
  in : input-port?
```

Returns two values: a list of snip-class names used by the given stream, and a list of data-class names used by the stream. If the stream is not a WXME stream, the result is two empty lists. The given stream is not closed, and only data for a WXME stream (if any) is consumed.

```
(register-lib-mapping! str mod-path) → void?
  str : string?
  mod-path : (cons/c (one-of/c 'lib) (listof string?))
```

Maps a snip-class name to a quoted module path that provides a reader% implementation. The module path must have the form '(lib *string* ...), where each *string* contains only alpha-numeric ASCII characters, ., _, -, and spaces.

```
(string->lib-path str gui?)
 → (cons/c (one-of/c 'lib) (listof string?))
  str : string?
  gui? : any/c
```

Returns a quoted module path for *str* for either editor<%> mode when *gui?* is true, or wxme mode when *gui?* is #f. For the latter, built-in mappings and mapping registered via register-lib-mapping! are used. If *str* cannot be parsed as a library path, and if no mapping is available (either because the class is built-in or not known), the result is #f.

```
(unknown-extensions-skip-enabled) → boolean?
(unknown-extensions-skip-enabled skip?) → void?
  skip? : any/c
```

A parameter. When set to #f (the default), an exception is raised when an unrecognized snip class is encountered in a WXME stream. When set to a true value, instances of unrecognized snip classes are simply omitted from the transformed stream.

```
(broken-wxme-big-endian?) → boolean?
(broken-wxme-big-endian? big?) → void?
  big? : any/c
```

A parameter. Some old and short-lived WXME formats depended on the endian order of the machine where the file was saved. Set this parameter to pick the endian order to use when reading the file; the default is the current platform's endian order.

```
(wxme-read in) → any/c
  in : input-port?
```

Like `read`, but for a stream that starts with WXME-format data. If multiple S-expressions are in the WXME data, they are all read and combined with `'begin`.

If `racket/gui/base` is available (as determined by `gui-available?`), then `open-input-text-editor` is used. Otherwise, `wxme-port->port` is used.

```
(wxme-read-syntax source-v in) → (or/c syntax? eof-object?)
  source-v : any/c
  in : input-port?
```

Like `read-syntax`, but for a WXME-format input stream. If multiple S-expressions are in the WXME data, they are all read and combined with `'begin`.

If `racket/gui/base` is available (as determined by `gui-available?`), then `open-input-text-editor` is used. Otherwise, `wxme-port->port` is used.

```
snip-reader<%> : interface?
```

An interface to be implemented by a reader for a specific kind of data in a WXME stream. The interface has two methods: `read-header` and `read-snip`.

```
(send a-snip-reader read-header version
                                stream) → any
  version : exact-nonnegative-integer?
  stream : (is-a?/c stream<%>)
```

Called at most once per WXME stream to initialize the data type's stream-specific information. This method usually does nothing.

```
(send a-snip-reader read-snip text-only?
                              version
                              stream)    → any/c
```

text-only? : Boolean?
version : exact-nonnegative-integer?
stream : (is-a?/c stream<%>)

Called when an instance of the data type is encountered in the stream. This method reads the data and returns either bytes to be returned as part of the decoded stream or any other kind of value to be returned as a "special" value from the decoded stream. The result value can optionally be an object that implements readable<%>.

---

readable<%> : interface?

An interface to be implemented by values returned from a snip reader. The only method is read-special.

(send *a-readable* read-special *source*
line
column
position) → any/c
source : any/c
line : (or/c exact-nonnegative-integer? false/c)
column : (or/c exact-nonnegative-integer? false/c)
position : (or/c exact-nonnegative-integer? false/c)

Like read-special, but for non-graphical mode. When a value implements this interface, its read-special method is called with source-location information to obtain the "special" result from the WXME-decoding port.

---

stream<%> : interface?

Represents a WXME input stream for use by snip-reader<%> instances.

(send *a-stream* read-integer *what*) → exact-integer?
what : any/c

Reads an exact integer, analogous to get-exact.

The *what* field describes what is being read, for error-message purposes, in case the stream does not continue with an integer.

(send *a-stream* read-fixed-integer *what*) → exact-integer?
what : any/c

Reads an exact integer that has a fixed size in the stream, analogous to get-fixed.

The *what* argument is as for read-integer.

(send *a-stream* read-inexact *what*) → (and/c real? inexact?)
  what : any/c

Reads an inexact real number, analogous to get-inexact.

The *what* argument is as for read-integer.

(send *a-stream* read-raw-bytes *what*) → bytes?
  what : any/c

Reads raw bytes, analogous to get-unterminated-bytes.

The *what* argument is as for read-integer.

(send *a-stream* read-bytes *what*) → bytes?
  what : any/c

Reads raw bytes, analogous to get-bytes.

The *what* argument is as for read-integer.

(send *a-stream* read-editor *what*) → input-port?
  what : any/c

Reads a nested editor, producing a new input port to extract the editor's content.

The *what* argument is as for read-integer.

2.7.1 Snip Class Mapping

When graphical data is marshaled to the WXME format, it is associated with a snip-class name to be matched with an implementation at load time. See also §1.3.2.1.1 "Snip Classes".

Ideally, the snip-class name is generated as

```
(format "~s" (list '(lib string ...)
                   '(lib string ...)))
```

where each element of the formated list is a quoted module path (see module-path?). The *string*s must contain only alpha-numeric ASCII characters, plus ., _, -, and spaces, and they must not be "." or "..".

In that case, the first quoted module path is used for loading WXME files in graphical mode; the corresponding module must provide snip-class object that implements the snip-class% class. The second quoted module path is used by the wxme library for converting WXME streams without graphical support; the corresponding module must provide a

reader object that implements the reader<%> interface. Naturally, the snip-class% instance and reader<%> instance are expected to parse the same format, but generate different results suitable for the different contexts (i.e., graphical or not).

If a snip-class name is generated as

```
(format "~s" '(lib string ...))
```

then graphical mode uses the sole module path, and wxme needs a compatibility mapping. Install one with register-lib-mapping!.

If a snip-class name has neither of the above formats, then graphical mode can use the data only if a snip class is registered for the name, or if it the name of one of the built-in classes: "wxtext", "wxtab", "wximage", or "wxmedia" (for nested editors). The wxme library needs a compatibility mapping installed with register-lib-mapping! if it is not one of the built-in classes.

Several compatibility mappings are installed automatically for the wxme library. They correspond to popular graphical elements supported by various versions of DrRacket, including comment boxes, fractions, XML boxes, Racket boxes, text boxes, and images generated by the "world" and "image" teachpacks (or, more generally, from mrlib/cache-image-snip), and test-case boxes.

For a port created by wxme-port->port, nested editors are represented by instances of the editor% class provided by the wxme/editor library. This class provides a single method, get-content-port, which returns a port for the editor's content. Images are represented as instances of the image% class provided by the wxme/image library.

Comment boxes are represented as instances of a class that extends editor% to implement readable<%>; see wxme/comment. The read form produces a special comment (created by make-special-comment), so that the comment box disappears when read is used to read the stream; the special-comment content is the readable instance. XML, Racket, and text boxes similarly produce instances of editor% and readable<%> that expand in the usual way; see wxme/xml, wxme/scheme, and wxme/text. Images from the "world" and "image" teachpacks are packaged as instances of cache-image% from the wxme/cache-image library. Test-case boxes are packaged as instances of test-case% from the wxme/test-case library.

Nested Editors

```
(require wxme/editor)
```

---

editor% : class?
  superclass: object%

Instantiated for plain nested editors in a WXME stream in text mode.

(send *an-editor* get-content-port) → input-port?

Returns a port (like the one from wxme-port->port) for the editor's content.

Images

```
(require wxme/image)
```

---

image% : class?
  superclass: object%

Instantiated for images in a WXME stream in text mode.

(send *an-image* get-filename) → (or/c bytes? false/c)

Returns a filename as bytes, or #f if data is available instead.

(send *an-image* get-data) → (or/c bytes? false/c)

Returns bytes for a PNG, XBM,or XPM file for the image.

(send *an-image* get-w)
 → (or/c exact-nonnegative-integer? (one-of/c -1))

Returns the display width of the image, which may differ from the width of the actual image specified as data or by a filename; -1 means that the image data's width should be used.

(send *an-image* get-h)
 → (or/c exact-nonnegative-integer? (one-of/c -1))

Returns the display height of the image, which may differ from the height of the actual image specified as data or by a filename; -1 means that the image data's height should be used.

(send *an-image* get-dx) → exact-integer?

Returns an offset into the actual image to be used as the left of the display image.

(send *an-image* get-dy) → exact-integer?

Returns an offset into the actual image to be used as the top of the display image.

2.7.2 DrRacket Comment Boxes

```
(require wxme/comment)
```

`reader : (is-a?/c snip-reader<%>)`

A text-mode reader for comment boxes.

`comment-editor% : class?`
  superclass: `editor%`
  extends: `readable<%>`

Instantiated for DrRacket comment boxes in a WXME stream for text mode.

`(send a-comment-editor get-data) → false/c`

No data is available.

```
(send a-comment-editor read-special source
                                    line
                                    column
                                    position) → any/c
  source : any/c
  line : (or/c exact-nonnegative-integer? false/c)
  column : (or/c exact-nonnegative-integer? false/c)
  position : (or/c exact-nonnegative-integer? false/c)
```

Generates a special comment using `make-special-comment`. The special comment contains the comment text.

2.7.3 DrRacket XML Boxes

```
(require wxme/xml)
```

`reader : (is-a?/c snip-reader<%>)`

A text-mode reader for XML boxes.

`xml-editor% : class?`
  superclass: `editor%`
  extends: `readable<%>`

Instantiated for DrRacket XML boxes in a WXME stream for text mode.

(send *a-xml-editor* get-data) → any/c

Returns #t if whitespace is elimited from the contained XML literal, #f otherwise.

(send *a-xml-editor* read-special *source*
line
column
position) → any/c
source : any/c
line : (or/c exact-nonnegative-integer? false/c)
column : (or/c exact-nonnegative-integer? false/c)
position : (or/c exact-nonnegative-integer? false/c)

Generates a quasiquote S-expression that enclosed the XML, with unquote and unquote-splicing escapes for nested Racket boxes.

2.7.4 DrRacket Racket Boxes

```
(require wxme/scheme)
```

---

reader : (is-a?/c snip-reader<%>)

A text-mode reader for Racket boxes.

---

racket-editor% : class?
superclass: editor%
extends: readable<%>

Instantiated for DrRacket Racket boxes in a WXME stream for text mode.

(send *a-racket-editor* get-data) → any/c

Returns #t if the box corresponds to a splicing unquote, #f for a non-splicing unquote.

(send *a-racket-editor* read-special *source*
line
column
position) → any/c
source : any/c
line : (or/c exact-nonnegative-integer? false/c)

column : (or/c exact-nonnegative-integer? false/c)
position : (or/c exact-nonnegative-integer? false/c)

Generates an S-expression for the code in the box.

2.7.5 DrRacket Text Boxes

```
(require wxme/text)
```

---

reader : (is-a?/c snip-reader<%>)

A text-mode reader for text boxes.

---

text-editor% : class?
superclass: editor%
extends: readable<%>

Instantiated for DrRacket text boxes in a WXME stream for text mode.

(send *a-text-editor* get-data) → false/c

No data is available.

(send *a-text-editor* read-special *source*
line
column
position) → any/c
source : any/c
line : (or/c exact-nonnegative-integer? false/c)
column : (or/c exact-nonnegative-integer? false/c)
position : (or/c exact-nonnegative-integer? false/c)

Generates a string containing the text.

2.7.6 DrRacket Fractions

```
(require wxme/number)
```

---

reader : (is-a?/c snip-reader<%>)

A text-mode reader for DrRacket fractions that generates exact, rational numbers.

2.7.7 DrRacket Teachpack Images

```
(require wxme/cache-image)
```

`reader` : `(is-a?/c snip-reader<%>)`

A text-mode reader for images in a WXME stream generated by the “image” and “world” teachpacks—or, more generally, by `mrlib/cache-image-snip`.

`cache-image%` : `class?`
  superclass: `object%`

Instantiated for DrRacket teachpack boxes in a WXME stream for text mode.

`(send a-cache-image get-argb)` → `(vectorof byte?)`

Returns a vector of bytes representing the content of the image.

`(send a-cache-image get-width)` → `exact-nonnegative-integer?`

Returns the width of the image.

`(send a-cache-image get-height)` → `exact-nonnegative-integer?`

Returns the height of the image.

`(send a-cache-image get-pin-x)` → `exact-integer?`

Returns an offset across into the image for the pinhole.

`(send a-cache-image get-pin-y)` → `exact-integer?`

Returns an offset down into the image for the pinhole.

2.7.8 DrRacket Test-Case Boxes

```
(require wxme/test-case)
```

`reader` : `(is-a?/c snip-reader<%>)`

A text-mode reader for DrRacket test-case boxes in a WXME stream. It generates instances of `test-case%`.

`test-case%` : `class?`
  superclass: `object%`

Instantiated for old-style DrRacket test-case boxes in a WXME stream for text mode.

`(send a-test-case get-comment) → (or/c false/c input-port?)`

Returns a port for the comment field, if any.

`(send a-test-case get-test) → input-port?`

Returns a port for the “test” field.

`(send a-test-case get-expected) → input-port?`

Returns a port for the “expected” field.

`(send a-test-case get-should-raise)`
`→ (or/c false/c input-port?)`

Returns a port for the “should raise” field, if any.

`(send a-test-case get-error-message)`
`→ (or/c false/c input-port?)`

Returns a port for the “error msg” field, if any.

`(send a-test-case get-enabled?) → boolean?`

Returns `#t` if the test is enabled.

`(send a-test-case get-collapsed?) → boolean?`

Returns `#t` if the test is collapsed.

`(send a-test-case get-error-box?) → boolean?`

Returns `#t` if the test is for an exception.

3 Configuration

3.1 Preferences

GRacket supports a number of preferences for global configuration. The GRacket preferences are stored in the common file reported by `find-system-path` for `'pref-file`, and preference values can be retrieved and changed through `get-preference` and `put-preferences`. However, GRacket reads most preferences once at startup (all except the `'GRacket:playcmd` preference).

The following are the (case-sensitive) preference names used by GRacket:

- `'GRacket:default-font-size` preference — sets the default font size the basic style in a style list, and thus the default font size for an editor.
- `'GRacket:controlFontSize` preference — sets the font size for control and menu labels (Windows, X); the font is the `'system` font, which can be configured as described in §3.2 “Font Configuration”.
- `'GRacket:defaultMenuPrefix` preference — sets the prefix used by default for menu item shortcuts under X, one of `'ctl`, `'meta`, or `'alt`. The default is `'ctl`. When this preference is set to `'meta` or `'alt`, underlined mnemonics (introduced by `&` in menu labels) are suppressed.
- `'GRacket:altUpSelectsMenu` preference — a true value makes pressing and releasing the Alt key select the first menu in the menu bar under X.
- `'GRacket:emacsUndo` preference — a true value makes undo in editors work as in Emacs (i.e., undo operations are themselves kept in the undo stack).
- `'GRacket:hiliteColor` preference — a string to sets the color for highlighting text, menus, and other GUI elements under X; the preference string should contain six hexadecimal digits, two for each component of the color. For example, set `'GRacket:hiliteColor` preference to `"0000A0"` and set `'GRacket:hiliteMenuBorder` preference to `#t` for a Bluecurve-like look.
- `'GRacket:hiliteMenuBorder` preference — a true value causes a menu selection to be highlighted with a border (in addition to a color) under X.
- `'GRacket:wheelStep` preference — sets the default mouse-wheel step size of `editor-canvas%` objects.
- `'GRacket:outlineInactiveSelection` preference — a true value causes selections in text editors to be shown with an outline of the selected region when the editor does no have the keyboard focus.
- `'GRacket:playcmd` preference — used to format a sound-playing command; see `play-sound` for details.

- `'GRacket:forceFocus` preference — a true value enables extra effort in GRacket to move the focus to a top-level window that is shown or raised.
- `'GRacket:doubleClickTime` preference — overrides the platform-specific default interval (in milliseconds) for double-click events.
- `'GRacket:gamma` preference — sets the gamma value used in gamma-correcting PNG files.
- `'GRacket:selectionAsClipboard` preference — under X, a true value causes `the-clipboard` to be an alias to `the-x-selection-clipboard`, which means that cut and paste operations use the X selection instead of the X clipboard. See also `clipboard<%>`.

In addition, preference names built from font face names can provide or override default entries for the `font-name-directory<%>`; see §3.2 “Font Configuration” for information.

3.2 Font Configuration

This chapter describes how to set up face mappings for screen and PostScript fonts via preferences (see §3.1 “Preferences”). The font-configuration system is overkill; it was designed to handle especially complex X font mappings before fontconfig/Xft solved the problem.

An implementor for a GRacket-based program may find it easier to use the `set-screen-name` and `set-post-script-name` methods provided by `the-font-name-directory`. As a user of a GRacket-based program, preferences provide a mechanism for setting default mappings.

Whether a programmer or a user, see `font-name-directory<%>` for an overview of the font mapping system.

To find a font name for a family, GRacket looks for a preference name by concatenating `MrEd:`, a ⟨*dest*⟩, a ⟨*type*⟩, a ⟨*weight*⟩, and a ⟨*style*⟩, where

- ⟨*dest*⟩ is either `Screen` or `PostScript`.
- ⟨*type*⟩ is either `Default`, `Decorative`, `Roman`, `Script`, `Swiss`, `Modern`, `System`, or `Symbol` for a mapping defining the default font for a family. Otherwise, it is a face name prefixed with `@`.
- ⟨*weight*⟩ is either `Medium`, `Bold`, or `Light`.
- ⟨*style*⟩ is either `Straight`, `Italic`, or `Slant`.

Furthermore, any of the latter three parts can be wildcarded with `_`, as described below. The concatenated string is converted to a symbol (preserving case), and the associated preference value must be a string.

The value of the preference is parsed as described in font-name-directory<%> for parsing face names, except that the string can contain references and other tricks described below.

3.2.1 Wildcards

Building items names by concatenating ⟨*dest*⟩, ⟨*type*⟩, ⟨*weight*⟩, and ⟨*style*⟩ can create a large number of preference entries, and the ⟨*weight*⟩ and ⟨*style*⟩ parts are useful only for X screen fonts. To avoid an explosion of preferences, GRacket finds preferences via a wildcarding search.

The ⟨*type*⟩, ⟨*weight*⟩, and ⟨*style*⟩ parts of a preference name can be wildcarded by using _. Thus, to set the default font in X for all types, weights, and styles, use the following preference entry:

```
(MrEd:Screen___ "+-*-*-medium-r-normal-*-*-%d-*-*-*-*-*-*")
```

Wildcarded preference entries are used only when un-wildcarded values cannot be found. If two preference names both match for some search, then the one with the "earliest" (i.e., closest to the beginning of the preference name) non-wildcarded part will prevail.

The default GRacket preferences for Windows uses wildcarding to specify the basic font mapping, as if written as:

```
(MrEd:ScreenSystem__ "MS Sans Serif")
(MrEd:ScreenRoman__ "Times New Roman")
(MrEd:ScreenDecorative__ "Modern")
....
```

Wildcarding in the preference name naturally leads to references, variables, and wildcarding references in the preference value. These features are described in the following few sections.

3.2.2 References

Suppose we define the mapping for variants of "Default", and then we want "Roman" to use this setting, too. We could copy the preference entry, as in the following example:

```
(MrEd:ScreenDefault__ "+-*-*-medium-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenRoman__ "+-*-*-medium-r-normal-*-*-%d-*-*-*-*-*-*")
```

but the GRacket font-reading system provides a better syntax for referencing another preference entry. When a preference value contains ${x}, then the ${x} fragment is replaced by the preference value of x. Thus, the above can be re-written:

```
(MrEd:ScreenDefault__ "+-*-*-medium-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenRoman__ "${ScreenDefault__}")
```

A mini-language of `${x}` is used within the string (instead of an S-expression format) for historical reasons.

3.2.3 Variables

Variables can be used with referencing to configure default values based on the weight and style that is needed. When a preference value contains `$[weight]`, then `$[weight]` is replaced with a string for the desired font weight. Similarly, `$[style]` is replaced with the desired style. Variable expressions can be embedded within referencing expressions, as in the following example:

```
(MrEd:ScreenDefault__
 "+-*-*-${Def$[weight]}-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:DefMedium "medium")
(MrEd:DefBold "bold")
(MrEd:DefLight "medium")
```

Now, when the `'GRacket:ScreenDefault__` preference value is used for different weights, it will return different values; the `${Def$[weight]}` expression will turn into `${DefMedium}` for a medium-weight lookup, or `${DefBold}` for a bold-weight lookup. These references will in turn give either `medium` or `bold`.

3.2.4 Wildcarding References

Consider the following preference configuration:

```
(MrEd:ScreenDefault__ "+-*-*-medium-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenDefaultBold_ "+-*-*-bold-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenRoman__ "${ScreenDefault__}")
```

The effect of this statement is probably not what was intended; when a bold version of the `Roman` font is needed, the `'GRacket:ScreenRoman__` preference value references the `'GRacket:ScreenDefault__` preference value, which does not specify a bold font. We could try to remedy the situation as follows:

```
(MrEd:ScreenDefault__ "+-*-*-medium-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenDefaultBold_ "+-*-*-bold-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenRoman__ "${ScreenDefault$[weight]_}")
```

but this does not work either. It works fine for bold `Roman`, now, but medium `Roman` will cause a reference to the `'GRacket:ScreenDefaultMedium_` preference, which doesn't

exist. The problem is that our reference does not use wildcarding like the original medium `Roman` lookup did.

Wildcarding can be specified in a reference by separating each wildcardable field with a comma. The following preference specification does what we want:

```
(MrEd:ScreenDefault__ "+-*-*-medium-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenDefaultBold_ "+-*-*-bold-r-normal-*-*-%d-*-*-*-*-*-*")
(MrEd:ScreenRoman__ "${ScreenDefault,$[weight],_}")
```

Since `$[weight]` is between commas, it can be wildcarded if no name exactly matching `ScreenDefault$[weight]_` is found. In this case `ScreenDefault` and `_` can also be wildcarded, but this will have no effect.

The wildcarding used in references need not reflect the wildcarding GRacket initial uses for finding fonts. In other words, a number of comma-separated selects can appear between the curly braces.

3.2.5 Internal Preferences

The initial font setup is built into GRacket through a built-in preference table. The table is shown at the end of this section. When font information is computed, it is *almost* as if this table were installed into your preferences file; the difference is that preference specifications in your file override specifications in the built-in table, even when the wildcarding of your preference provides a weaker match.

When no information is available for mapping a face name to a font, GRacket falls back to the system described in `font-name-directory<%>`. (Since a mapping is built into GRacket for every family, information is always available for the default font of a family.)

Internal preferences for all platforms:

```
(MrEd:PostScriptMediumStraight "")
(MrEd:PostScriptMediumItalic "-Oblique")
(MrEd:PostScriptMediumSlant "-Oblique")
(MrEd:PostScriptLightStraight "")
(MrEd:PostScriptLightItalic "-Oblique")
(MrEd:PostScriptLightSlant "-Oblique")
(MrEd:PostScriptBoldStraight "-Bold")
(MrEd:PostScriptBoldItalic "-BoldOblique")
(MrEd:PostScriptBoldSlant "-BoldOblique")

(MrEd:PostScript___ "${PostScript$[family],$[weight],$[style]}")

(MrEd:PostScriptSystem__ "${PostScriptTimes,$[weight],$[style]}")
```

```
(MrEd:PostScriptRoman__ "${PostScriptTimes,$[weight],$[style]}")
(MrEd:PostScriptDecorative__ "${PostScriptTimes,$[weight],$[style]}")
(MrEd:PostScriptScript__ "ZapfChancery-MediumItalic")

(MrEd:PostScriptTimesMedium "")
(MrEd:PostScriptTimesLight "")
(MrEd:PostScriptTimesBold "Bold")

(MrEd:PostScriptTimes__ "Times${PostScript$[weight]$[style]}")
(MrEd:PostScriptTimesMediumStraight "Times-Roman")
(MrEd:PostScriptTimesLightStraight "Times-Roman")
(MrEd:PostScriptTimes_Slant
 "Times-${PostScriptTimes$[weight]}Italic")
(MrEd:PostScriptTimes_Italic
 "Times-${PostScriptTimes$[weight]}Italic")

(MrEd:PostScriptDefault__ "Helvetica${PostScript$[weight]$[style]}")
(MrEd:PostScriptSwiss__ "Helvetica${PostScript$[weight]$[style]}")
(MrEd:PostScriptModern__ "Courier${PostScript$[weight]$[style]}")
(MrEd:PostScriptSymbol__ "Symbol")
```

Internal preferences for X with fontconfig/Xft/RENDER only:

```
(MrEd:ScreenSystem__ " Sans")
(MrEd:ScreenDefault__ " Sans")
(MrEd:ScreenRoman__ " Serif")
(MrEd:ScreenDecorative__ " Nimbus Sans L")
(MrEd:ScreenModern__ " Monospace")
(MrEd:ScreenSwiss__ " Nimbus Sans L")
(MrEd:ScreenScript__ " URW Chancery L")
(MrEd:ScreenSymbolBase " Standard Symbols L,Nimbus Sans L")
```

Internal preferences for X only (except those overridden for fontconfig/Xft/RENDER):

```
(MrEd:ScreenMedium "medium")
(MrEd:ScreenBold "bold")
(MrEd:ScreenLight "light")
(MrEd:ScreenStraight "r")
(MrEd:ScreenItalic "i")
(MrEd:ScreenSlant "o")

(MrEd:ScreenSystemBase "*-lucida")
(MrEd:ScreenDefaultBase "*-lucida")
(MrEd:ScreenRomanBase "*-times")
(MrEd:ScreenDecorativeBase "*-helvetica")
(MrEd:ScreenModernBase "*-courier")
```

```
(MrEd:ScreenSwissBase "*-lucida")
(MrEd:ScreenScriptBase "*-zapfchancery")
(MrEd:ScreenSymbolBase "*-symbol")

(MrEd:ScreenStdSuffix
 "-${Screen$[weight]}-${Screen$[style]}-normal-*-*-%d-*-*-*-*-*-*")

(MrEd:ScreenSystem__ "+-${ScreenSystemBase}${ScreenStdSuffix}")
(MrEd:ScreenDefault__ "+-${ScreenDefaultBase}${ScreenStdSuffix}")
(MrEd:ScreenRoman__ "+-${ScreenRomanBase}${ScreenStdSuffix}")
(MrEd:ScreenDecorative__
 "+-${ScreenDecorativeBase}${ScreenStdSuffix}")
(MrEd:ScreenModern__ "+-${ScreenModernBase}${ScreenStdSuffix}")
(MrEd:ScreenSwiss__ "+-${ScreenSwissBase}${ScreenStdSuffix}")
(MrEd:ScreenScript__ "+-${ScreenScriptBase}${ScreenStdSuffix}")
(MrEd:ScreenSymbol__
 "+-${ScreenSymbolBase}-medium-r-normal-*-*-%d-*-*-*-*-*-*")
```

Internal preferences for Windows only:

```
(MrEd:ScreenSystem__ "MS Sans Serif")
(MrEd:ScreenDefault__ "MS Sans Serif")
(MrEd:ScreenRoman__ "Times New Roman")
(MrEd:ScreenDecorative__ "Arial")
(MrEd:ScreenModern__ "Courier New")
(MrEd:ScreenSwiss__ "Arial")
(MrEd:ScreenScript__ "Arial")
(MrEd:ScreenSymbol__ "Symbol")
```

Internal preferences for Mac OS X only:

```
(MrEd:ScreenDefault__ "Lucida Grande")
(MrEd:ScreenSystem__ "Lucida Grande")
(MrEd:ScreenRoman__ "Times")
(MrEd:ScreenDecorative__ "Arial")
(MrEd:ScreenModern__ "Courier New")
(MrEd:ScreenSwiss__ "Helvetica")
(MrEd:ScreenScript__ "Apple Chancery")
(MrEd:ScreenSymbol__ "Symbol")
```

3.2.6 PostScript Fonts

To generate PostScript output, GRacket must be able to find an Adobe Font Metrics (AFM) file corresponding to the PostScript font. An AFM file typically uses the suffix `".afm"`, and several AFM files are distributed with GRacket in the `"afm"` collection.

GRacket finds an AFM file by adding a ".afm" suffix to the PostScript name of the font, and checking all directories specified by the current-ps-afm-file-paths parameter. The initial value of this parameter is determined by the PLTAFMPATHS environment variable; the environment variable's setting is parsed with path-list-string->path-list using (list (collection-path "afm")) as the default list.

Depending on whether the font is CID-based (typically for the Chinese, Japanese, Korean, and Vietnamese language families, and as indicated in the AFM file), GRacket must find additional files:

- *Non-CID:* In addition to an AFM file "⟨*x*⟩.afm", GRacket looks for a "⟨*x*⟩-glyphlist.txt" file (in the same directory as the AFM file) to map glyph names in the AFM file to Unicode character values. In addition to this font-specific file, GRacket looks for a "glyphlist.txt" file to supply a mapping for Adobe's standard glyph names, and this mapping is used when a font-specific mapping is not supplied, or when the mapping does not cover a name found in the AFM file. GRacket looks for "glyphlist.txt" in the same place as AFM files. Since "glyphlist.txt" is large, if a "glyphshortlist.txt" file is available, it is read first, and then "glyphlist.txt" is read only if a character name must be resolved that is not in "glyphshortlist.txt".

- *CID:* In addition to an AFM file, GRacket must find and read CMap files to convert glyph IDs for the font to Unicode characters. The character set name is used as the name of the CMap file to load, and GRacket checks all directories specified by the current-ps-cmap-file-paths parameter. The initial value of this parameter is determined by the PLTCMAPPATHS environment variable; the environment variable's setting is parsed with path-list-string->path-list using (list (collection-path "afm" "CMap")) as the default list. In addition to a CMap file for the font's character set, GRacket must find a "UniCNS-UTF32-H" CMap file to complete the mapping to Unicode. GRacket automatically adds the font's character set to the font name when producing PostScript with a CID-based font.

When drawing or measuring text using a particular PostScript font, if the font does not contain a glyph for a character (or if a relevant AFM file cannot be found for the font), then GRacket attempts to substitute another PostScript font. A substitute font is selected by checking all ".afm" files in the directories specified by current-ps-afm-file-paths (in order), and choosing the first discovered match.

4 Dynamic Loading

```
(require racket/gui/dynamic)
```

The racket/gui/dynamic library provides functions for dynamically accessing the Racket GUI toolbox, instead of directly requiring racket/gui or racket/gui/base.

(gui-available?) → boolean?

Returns #t if dynamic access to the GUI bindings are available—that is, that the program is being run as a GRacket-based application, as opposed to a pure Racket-based application, and that GUI modules are attached to the namespace in which racket/gui/dynamic was instantiated.

This predicate can be used in code that optionally uses GUI elements when they are available.

(gui-dynamic-require *sym*) → any
  sym : symbol?

Like dynamic-require, but specifically to access exports of racket/gui/base.

Bibliography

[Adobe99] Adobe Systems Incorporated, *PostScript Language Reference, third edition*. 1999. `http://partners.adobe.com/public/developer/en/ps/PLRM.pdf`

Index