LispWorks° for the Windows° Operating System $CAPI\,User\,Guide$

Version 4.1



Copyright and Trademarks

LispWorks for the Windows Operating System CAPI User Guide

Version 4.1

November 1998

Part number: 3LBDT2A15NF

Copyright © 1994–1998 by Harlequin Group plc.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Harlequin Group plc.

The information in this publication is provided for information only and is subject to change without notice. Harlequin Group plc and its affiliates assume no responsibility or liability for any loss or damage that may arise from the use of any information in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks is a registered trademark of Harlequin Group plc. Harlequin, Delivery, Transducer/PC, The Authoring Book, ClassWorks, and KnowledgeWorks are all trademarks of Harlequin Group plc.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.

The code for walker.lisp and compute-combination-points is excerpted with permission from PCL, Copyright © 1985, 1986, 1987, 1988 Xerox Corporation.

CLX and CLUE bear the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Texas Instruments Incorporated, P.O. Box 149149, MS 2151, Austin, Texas 78714-9149

Copyright © 1987, 1988, 1989, 1990, 1991 Texas Instruments Incorporated.

Permission is granted to any individual or institution to use, modify and distribute this software, provided that this complete copyright and permission notice is maintained, intact, in all copies and documentation. Texas Instruments Incorporated provides this software "as is" without express or implied warranty.

The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

US Government Use

The LispWorks Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: "Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin's customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be `unpublished' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge, Massachusetts 02142."

http://www.harlequin.com/

Europe:	North America:	Asia Pacific:
Harlequin Limited	Harlequin Incorporated	Harlequin Australia Pty.
Barrington Hall	One Cambridge Conter	Lovol 19

Level 12 12 Moore Street Canberra, ACT 2601 Australia Ltd.

telephone +61 2 6206 5522 fax +61 2 6206 5525

Barrington Hall Barrington Cambridge CB2 5RG UK

telephone +44 1223 873 800 fax +44 1223 873 873 One Cambridge Center Cambridge, MA 02142 USA

telephone +1 617 374 2400 fax +1 617 252 6505

Contents

Preface vii

1 Introduction to the CAPI 1

What is the CAPI? 1 The history of the CAPI 2 The CAPI model 2

2 Getting Started 5

Loading the CAPI 6 Creating a window 6 Linking code into CAPI elements 8

3 Creating Common Windows 11

Generic properties 11 Specifying titles 13 Displaying and entering text 15 Stream panes 16 Miscellaneous button elements 17

4 Choices 21

Choice classes 22 List panels 24 Graph panes 28 Option panes and drop-down list boxes 30 Menu components 31 General properties of choices 31

5 Laying Out CAPI Panes 35

Organizing panes in columns and rows 36 Other types of layout 38 Combining different layouts 39 Constraining the size of layouts 40 Advanced pane layouts 42

6 Creating Menus 45

Creating a menu 45 Grouping menu items together 46 Creating individual menu items 49 The CAPI menu hierarchy 50 Disabling menu items 51

7 Defining Interface Classes 53

The define-interface macro 53 An example interface 54 Adapting the example 56 Connecting an interface to an application 61

8 Prompting for Input 65

Some simple dialogs 65 Prompting for values 67 Creating your own dialogs 73

9 Creating Your Own Panes 79

Displaying graphics 79 Receiving input from the user 82 Creating graphical objects 83

10 Graphics Ports 89

Introduction 89 Features 90 Graphics state 90 Graphics state transforms 93 Pixmap graphics ports 95 Portable font descriptions 96

11 The Color System 99

Introduction 99 Reading the color database 100 Color specs 101 Color aliases 102 Color models 103 Loading the color database 105 Defining new color models 106

12 Printing from the CAPI—the Hardcopy API 109

Printers 109 Print jobs 109 Handling pages—page on demand printing 110 Handling pages—page sequential printing 110 Printing a page 110 Other printing functions 111

Index 113

Preface

This preface contains information you need when using the rest of the CAPI documentation. It discusses the purpose of this manual, the typographical conventions used, and gives a brief description of the rest of the contents.

Assumptions

The CAPI documentation assumes that you are familiar with:

- LispWorks
- Common Lisp and CLOS, the Common Lisp Object System
- The Windows environment

Conventions used in the manual

Throughout this manual, certain typographical conventions have been adopted to aid readability.

1. Whenever an instruction is given, is numbered and printed like this.

Text which you should type explicitly is printed like this.

A Description of the Contents

This guide forms an introductory course in developing applications using the CAPI. Please note that, like the rest of the LispWorks documentation, it does assume knowledge of Common Lisp.

Chapter 1, *Introduction to the CAPI*, introduces the principles behind the CAPI, some of its fundamental concepts, and what it sets out to achieve.

Chapter 2, *Getting Started*, presents a series of simple examples whose aim is to familiarize you with some of the most important elements and functions.

Chapter 3, *Creating Common Windows*, introduces more of the fundamental CAPI elements. These elements are explained in greater detail in the remainder of the manual.

Chapter 4, *Choices*, explains the key CAPI concept of the *choice*. A choice groups CLOS objects together and provides the notion of there being a selected object amongst that group of objects. Button panels and list panels are examples of choices.

Chapter 5, *Laying Out CAPI Panes* introduces the idea of *layouts*. These let you combine different CAPI elements inside a single window.

Chapter 6, Creating Menus, shows you how to add menus to a window.

Chapter 7, *Defining Interface Classes*, introduces the macro define-interface. This macro can be used to define interface classes composed of CAPI elements — either the predefined elements explained elsewhere in this manual or your own.

Chapter 8, *Prompting for Input*, discusses the ways in which dialog boxes may be used to prompt a user for input.

Chapter 9, *Creating Your Own Panes*, shows you how you can define your own classes when those provided by the CAPI are not sufficient for your needs.

Chapter 10, *Graphics Ports*, provides information on the Graphics Ports package, which provides a selection of drawing and image tranformation functions. Although not part of the CAPI package, and therefore not strictly part of the CAPI, the Graphics Ports functions are used in conjunction with CAPI panes, and are therefore documented in this manual and the *CAPI Reference Manual*.

Chapter 11, *The Color System*, allows applications to use keyword symbols as aliases for colors in Graphics Ports drawing functions. They can also be used for backgrounds and foregrounds of windows and CAPI objects.

The Reference Manual

The second part of the CAPI documentation is the *CAPI Reference Manual*. This provides a complete description of every CAPI class and function, and also provides a reference chapter on the Graphics Port functions. Entries are listed alphabetically, and the typographical conventions used are similar to those used in *Common Lisp: the Language* (2nd Edition) (Steele, 1990).

Introduction to the CAPI

1.1 What is the CAPI?

The CAPI (Common Application Programmer's Interface) is a library for implementing portable window-based application interfaces. It is a conceptually simple, CLOS-based model of interface elements and their interaction. It provides a standard set of these elements and their behaviors, as well as giving you the opportunity to define elements of your own.

The CAPI's model of window-based user interfaces is an abstraction of the concepts that are shared between all contemporary window systems, such that you do not need to consider the details of a particular system. These hidden details are taken care of by a back end library written for that system alone.

An advantage of making this abstraction is that each of the system-specific libraries can be highly specialized, concentrating on getting things right for that particular window system. Furthermore, because the implementation libraries and the CAPI model are completely separate, libraries can be written for new window systems without affecting either the CAPI model or the applications you have written with it.

The CAPI currently runs under X Window System with Motif, and Microsoft Windows.

1.2 The history of the CAPI

Until recently, window-based applications written with LispWorks used CLX, CLUE, and the LispWorks Toolkit. Such applications are restricted to running under X Windows. Because we and our customers wanted a way to write portable window code, we decided to develop a new system for this purpose: the CAPI.

Part of this portability exercise was undertaken before the development of the CAPI, for graphics ports, the generic graphics library. This includes the portable color, font, and image systems in LispWorks. The CAPI is built on top of this technology.

All Lisp-based environment and application development in Harlequin now uses the CAPI. We recommend that you use the CAPI for window-based application development in preference to the systems mentioned earlier.

1.3 The CAPI model

The CAPI provides an abstract hierarchy of classes which represent different sorts of window interface elements, along with functions for interacting with them. Instances of these classes represent window objects in an application, with their slots representing different aspects of the object, such as the text on a button, or the items on a menu. These instances are not actual window objects but provide a convenient representation of them for you. When you ask the CAPI to display your object, it creates a real window system object to represent it. This means that if you display a CAPI button, a real Windows button is created for it when running on Windows, and a real Motif button when running on Motif.

A different approach would have been to simulate the window objects and their look and feel. This approach is problematic. Because the library makes itself entirely responsible for the application's look and feel, it may not simulate it correctly in obscure cases. Also, manufacturers occasionally change the look and feel of their window systems. Applications written with a library that simulates window objects will continue to have the old look and feel until the application is recompiled with an updated library.

The CAPI's approach makes the production of the screen objects the responsibility of the native window system, so it always produces the correct look and feel. Furthermore, the CAPI's use of the real interface to the window system means that it does not need to be upgraded to account for look and feel changes, and anything written with it is upwardly compatible, just like any well-written application.

1.3.1 CAPI Classes

There are four basic objects in the CAPI model: *interfaces, menus, panes* and *lay-outs*.

Everything that the CAPI displays is contained within an interface (an instance of the class interface). When an interface is displayed a window appears containing all the menus and panes you have specified for it.

An interface can contain a number of menus which are collected together on a menu bar. Each menu on the menu bar can contain menu items or other menus. Items can be grouped together visually and functionally inside *menu components*. Menus, menu items, and menu components are, respectively, instances of the classes menu, menu-item, and menu-component.

Panes are window objects such as buttons and lists. They can be positioned anywhere in an interface. The CAPI provides many different kinds of pane class, among them push-button, list-panel, editor-pane, and graph-pane.

The positions of panes are controlled by a layout, which allows objects to be collected together and positioned either regularly (with instances of the classes column-layout Or row-layout) or arbitrarily using a pinboard-layout. Layouts themselves can be laid out by other layouts — for example, a row of buttons can be laid out above a list by placing both the row-layout and the list in a column-layout. Introduction to the CAPI

Getting Started

This chapter introduces some of the most basic CAPI elements and functions. The intention is simply that you should become familiar with the most useful elements available, before learning how you can use them constructively. You should work through the examples in this chapter.

A CAPI application consists of a hierarchy of CAPI objects, in much the same way as an application written using the LispWorks Toolkit consists of Toolkit objects. CAPI objects are created using make-instance, and although they are standard CLOS objects, CAPI features should generally be accessed in the ways described in this manual, and not using the CLOS slot-value function. You should not rely on slot-value because the implementation of the CAPI classes may evolve.

Once an instance of a CAPI object has been created in an interface, it can be displayed on your screen using the function display.

2.1 Loading the CAPI

All symbols in this manual are exported from either the CAPI or COMMON-LISP packages unless explicitly stated otherwise. You should use the CAPI package in your code to access them:

```
(defpackage "MY-PACKAGE"
 (:add-use-defaults t)
 (:use "CAPI")
)
```

This creates and loads a package containing all the CAPI commands required to try out the examples in this guide.

2.2 Creating a window

This section shows how easy it is to create a simple window, and how to include CAPI elements, such as panes, in your window.

1. Type the following in a listener





Figure 2.1 Creating a simple window

A small window appears on your screen, called "My interface".

The usual way to displaying an instance of a CAPI window is display; however, another function, contain, is provided to help you during the course of development.

Only a top level CAPI element is shown by display — that is, an instance of an interface. To display other CAPI elements (for example, buttons, editor

panes, and so on), you must provide information about how they are to be arranged in the window. Such an arrangement is called a *layout* — you will learn more about layouts in Chapter 5.

On the other hand, contain automatically provides a default layout for any CAPI element you specify, and subsequently displays it. During development, it can be useful for displaying individual elements of interest on your screen, without having to create an interface for them explicitly. However, contain is only provided as a development tool, and should not be used for the final implementation of a CAPI element. See Chapter 7, "Defining Interface Classes" on how to display CAPI elements in an interface.

This is how you can create and display a button using contain.

1. Type the following into a listener:

```
(make-instance 'push-button
  :data "Button")
(contain *)
```



Figure 2.2 Creating a push-button interface

This creates an interface which contains a single push-button, with a label specified by the :data keyword. Notice that you could have performed the same example using display, but you would also have had to create a layout so that the button could have been placed in an interface and displayed.

You can click on the button, and it will respond in the way you would expect (it will depress). However, no code will be run which performs an action associated with the button. How to link code to window items is the topic of the next section.

2.3 Linking code into CAPI elements

Getting a CAPI element to perform an action is done by specifying a *callback*. This is a function which is performed whenever you change the state of a CAPI element. It calls a piece of code whenever a choice is made in a window.

Note that the result of the callback function is ignored, and that its usefulness is in its side-effects.

1. Try the following:

```
(make-instance 'push-button
  :data "Hello"
  :callback
  #'(lambda (&rest args)
        (display-message
        "Hello World")))
```

(contain *)

🛃 Co	-	×
Button		

Figure 2.3 Specifying a callback

2. Click on the Hello button.

A dialog appears containing the message "Hello World". The CAPI provides the function display-message to allow you to pop up a dialog box containing a message and a Confirm button. This is one of many pre-defined facilities that the CAPI offers.

Note: When developing applications in the CAPI, note that your CAPI application windows are run in the same Windows system event loop as the Common LispWorks environment itself. This - and the fact that in Common Lisp, user code exists in the same global namespace as the Common Lisp implementation itself - means that a CAPI application running under the Common LispWorks environment can modify the same values as you can concurrently modify from one of the environment's programming tools.

For example, your CAPI application might have a button that, when pressed, sets a slot in a particular object that you could also set by hand in the listener. This situation can lead to unexpected values and behavior in your CAPI application, which may seem to reveal bugs in the application or the Common Lisp product that do not in fact exist.

Getting Started

Creating Common Windows

So far you have only seen two types of CAPI element: the interface (which is the top level CAPI element, and is present in any CAPI window) and the push-button. This section shows how you can use the CAPI to create other common windowing elements you are likely to need.

Before trying out the examples in this chapter, define the functions testcallback and hello in your Listener. The first displays the list of arguments it is given, and returns nil. The second just displays a message.

We will use these callbacks in future examples.

3.1 Generic properties

Because CAPI elements are just like CLOS classes, many elements share a common set of properties. This section describes the properties that all the classes described in this chapter inherit.

3.1.1 Scroll bars

The CAPI lets you specify horizontal or vertical scroll bars for any subclass of the simple pane element (including all of the classes described in this chapter).

Horizontal and vertical scroll bars can be specified using the keywords :horizontal-scroll and :vertical-scroll. By default, :vertical-scroll is set to t, and :horizontal-scroll is set to nil.

3.1.2 Background and foreground colors

All subclasses of the simple pane element can have different foreground and background colors, using the *:background* and *:foreground* keywords. For example, including

```
:background :blue
:foreground :yellow
```

in the make-instance of a text pane would result in a pane with a blue background and yellow text.

3.1.3 Fonts

The CAPI interface supports the use of other fonts for text in title panes and other CAPI objects, such as buttons, through the use of the :font keyword. If the CAPI cannot find the specified font it reverts to the default font. The :font keyword applies to data following the :text keyword, and uses the graphics port package command gf. The weight of the lettering can be one of :bold, :medium, :demibold, and :light. The slant of the lettering can be :roman, :italic, :oblique, :reverse-oblique and :reverse-italic.

Here is an example of a title pane with an explicit font:

```
(contain (make-instance 'title-pane
    :text "A title pane"
    :font (gp:make-font-description
    :family "times" :size 12 :weight :medium
    :slant :roman)))
```

Here is an example of using :font to produce a title pane with larger lettering. Note that the CAPI automatically resized the pane to fit around the text.

```
(contain (make-instance 'title-pane
    :text "A large piece of text"
    :font (gp:make-font-description
    :family "times" :size 34 :weight :medium
    :slant :roman)))
```



Figure 3.1 An example of a generic font

3.2 Specifying titles

It is possible to specify a title for a window, or part of a window. Several of the examples that you have already seen have used titles. There are two ways that you can create titles: by using the title-pane class, or by specifying a title directly to any subclass of titled-pane.

3.2.1 Title panes

A title pane is a blank pane into which text can be placed in order to form a title.





Figure 3.2 A title pane

3.2.2 Specifying titles directly

You can specify a title directly to all CAPI panes, using the :title keyword. This is much easier than using title-panes, since it does not necessitate using a layout to group two elements together.

Any class that is a subclass of titled-pane supports the :title keyword. All of the standard CAPI panes inherit from this class. You can find out all the subclasses of titled-pane by graphing them using the class browser.

The position of any title can be specified by using the :title-position keyword. Most panes default the title-position to :top, although some use :left.

The title of a CAPI element may be changed interactively with the use of setf, if you wish.

1. Create a push button by evaluating the code below:

```
(setq button (make-instance 'push-button
            :text "Hello"
            :title "Press: "
            :title-position :left
            :callback 'hello))
```

(contain button)

2. Now evaluate the following:

(setf (titled-pane-title button) "Press here: ")

As soon as the form is evaluated, the title of the pane you just created changes. Notice how the window automatically resizes to make allowance for the size of the new title.

3.2.3 Display panes

Display panes can be used to display text messages on the screen. The text in these messages cannot be edited, so they can be used as by the application to present a message to the user. The :text keyword can be used to specify the message that is to appear in the pane.

1. Create a display pane by evaluating the code below:

```
(setq display (make-instance 'display-pane
            :text "This is a message"))
(contain display)
```



Figure 3.3 A display pane

3.3 Displaying and entering text

There are a variety of ways in which a user can type text into an application. Editor panes are commonly used for dealing with large amounts of text, such as files, and text input panes are used for entering short pieces of text which are to be acted upon immediately.

3.3.1 Text input panes

When you want to enter a line of text — for instance a search string — a text input pane can be used.

```
(setq text (make-instance 'text-input-pane
            :title "Search: "
            :callback 'test-callback))
(contain text)
```



Figure 3.4 A text input pane

Notice that the default title position for text input panes is :left.

Text may be placed in the text input pane itself by giving the :text keyword a string argument.

3.3.2 Editor panes

Editor panes can be created using the editor-pane element.

```
(setq editor (make-instance 'editor-pane))
(contain editor)
```

This creates an editor as described in the manual LispWorks Editor User Guide.



Figure 3.5 An editor pane

3.4 Stream panes

There are three subclasses of editor-pane which handle Common Lisp streams.

3.4.1 Collector panes

A collector pane displays anything printed to the stream associated with it. Background output windows, for instance, are examples of collector panes.

```
(contain (make-instance 'collector-pane
            :title "Example collector pane:"))
```

3.4.2 Interactive streams

An interactive stream is the building block on which a listener pane is built.

```
(contain (make-instance 'interactive-stream
            :title "Stream:"))
```

3.4.3 Listener panes

The listener-pane class is a subclass of interactive-stream, and allows you to create interactive Common Lisp sessions. You may occasionally want to include a listener pane in a tool (as, for instance, Common LispWorks does with the Debugger).

```
(contain (make-instance 'listener-pane
                             :title "Listener:"))
```

3.5 Miscellaneous button elements

A variety of different buttons can be created for use in an application. These include push buttons, which you have already seen, and check buttons. Button panels can also be created, and are described in Chapter 4, "Choices".

3.5.1 Push buttons

You have already seen push buttons in earlier examples. The **:enabled** keyword can be used to specify whether or not the button should be selectable when it is displayed. This can be useful for disabling a button in certain situations.

The following code creates a push button which cannot be selected.

```
(setq offbutton (make-instance 'push-button
        :data "Button"
        :enabled nil))
```

```
(contain offbutton)
```

These setf expansions enable and disable the button:

```
(setf (button-enabled offbutton) t)
(setf (button-enabled offbutton) nil)
```

All subclasses of the button class can be disabled in this way.

3.5.2 Check buttons

Check buttons can be produced with the check-button element.

1. Type the following in a Listener:

(contain check)

🛃 Co	_	Ð	×
🗹 Button			

Figure 3.6 A check button

Notice the use of **:retract-callback** in the example above, to specify a callback when the element is deselected.

Like push buttons, check buttons can be disabled by specifying :enabled nil.

3.5.3 Radio buttons

Radio buttons can be created explicitly although they are usually part of a button panel as described in Chapter 4, *Choices*. The <code>:selected</code> keyword is used to specify whether or not the button is selected, and the <code>:text</code> keyword can be used to label the button.

```
(contain (make-instance 'radio-button
    :text "Radio Button"
    :selected t))
```



Figure 3.7 An explicitly created radio button

Although a single radio button is of limited use, having an explicit radio button class gives you greater flexibility, since associated radio buttons need not be physically grouped together. Generally, the easiest way of creating a group of radio buttons is by using a button panel, but doing so means that they will be geometrically, as well as semantically, connected. Creating Common Windows

Choices

Some elements of a window interface contain collections of items, for example rows of buttons, lists of filenames, and groups of menu items. Such elements are known in the CAPI as *collections*.

In most collections, items may be selected by the user — for example, a row of buttons. Collections whose items can be selected are known as *choices*. Each button in a row of buttons is either checked or unchecked, showing something about the application's state — perhaps that color graphics are switched on and sound is switched off. This selection state came about as the result of a *choice* the user made when running the application, or default choices made by the application itself.

The CAPI provides a convenient way of producing groups of items from which collections and choices can be made. The abstract class collection provides a means of specifying a group of items. The subclass choice provides groups of selectable items, where you may specify what initial state they are in, and what happens when the selection is changed. Subclasses of collection and choice used for producing particular kinds of grouped elements are described in the sections that follow.

All the choices described in this chapter can be given a print function via the :print-function keyword. This allows you to control the way in which items in the element are displayed. For example, passing the argument 'string-

Choices

capitalize to :print-function would capitalize the initial letters of all the words of text that an instance of a choice displays.

Some of the examples in this chapter require the functions test-callback and hello which were introduced in Chapter 3, "Creating Common Windows".

4.1 Choice classes

This section discusses the immediate subclasses of choice which can be used to build button panels. If you have a group of several buttons, you can use the appropriate button-panel element to specify them all as a group, rather than using push-button or check-button to specify each one separately. There are three such elements altogether: push-button-panel, check-button-panel and radio-button-panel. The specifics of each are discussed below.

4.1.1 Push button panels

The arrangement of a number of push buttons into one group can be done with a push-button-panel. Since this provides a panel of buttons which do not maintain a selection when you click on them, push-button-panel is a choice that does not allow a selection. When a button is activated it causes a :selection-callback, but the button does not maintain the selected state.

Here is an example of a push button panel:

```
(make-instance 'push-button-panel
    :items '(one two three four five)
    :selection-callback 'test-callback
    :print-function 'string-capitalize)
(contain *)
```



Figure 4.1 A group of push-buttons

The layout of a button panel (for instance, whether items are listed vertically or horizontally) can be specified using the <code>:layout-class</code> keyword. This can take two values: <code>'column-layout</code> if you wish buttons to be listed vertically, and <code>'row-layout</code> if you wish them to be listed horizontally. The default value is <code>'row-layout</code>. If you define your own layout classes, you can also use these as values to <code>:layout-class</code>. Layouts, which apply to many other CAPI objects, are discussed in detail in Chapter 5, "Laying Out CAPI Panes".

4.1.2 Radio button panels

A group of radio buttons (a group of buttons of which only one at a time can be selected) is created with the radio-button-panel class. Here is an example of a radio button panel:

```
(setq radio (make-instance 'radio-button-panel
                :items (list 1 2 3 4 5)
               :selection-callback 'test-callback))
(ceptain radio)
```

(contain radio)



Figure 4.2 A group of radio-buttons

4.1.3 Check button panels

A group of check buttons can be created with the check-button-panel class. Any number of check buttons can be selected.

Here is an example of a check button panel:

```
(contain
(make-instance
    'check-button-panel
    :items '("Red" "Green" "Blue")))
```



Figure 4.3 A check button panel

4.2 List panels

Lists of selectable items can be created with the *list-panel* class. Here is a simple example of a list panel:

```
(setq list (make-instance 'list-panel
               :items '(one two three four)
               :print-function 'string-capitalize))
```

```
(contain list)
```



Figure 4.4 A list panel

Notice how the items in the list panel are passed as symbols, and a print-function is specified which controls how those items are displayed on the screen.

Any item on the list can be selected by clicking on it with the mouse.

By default, list panels are single selection — that is, only one item in the list may be selected at once. You can use the :interaction keyword to change this:

```
(make-instance 'list-panel
            :items (list "One" "Two" "Three" "Four")
            :interaction :multiple-selection)
(contain *)
```

You can add callbacks to any items in the list using the :selection-callback keyword.

```
(make-instance 'list-panel
    :items (list "One" "Two" "Three" "Four")
    :selection-callback 'test-callback)
(contain *)
```

4.2.1 List interaction

If you select different items in the list, only the last item you select remains highlighted. The way in which the items in a list panel interact upon selection can be controlled with the <code>:interaction</code> keyword.

The list produced in the example above is known as a single-selection list because only one item at a time may be selected. List panels are <code>:single-selection</code> by default.

There are also multiple-selection and extended-selection lists available. The possible interactions for list panels are:

- :single-selection only one item may be selected
- :multiple-selection more than one item may be selected
- :extended-selection see Section 4.2.2

To get a particular interaction, supply one of the values above to the :interaction keyword, like this:

```
(contain
(make-instance
  'list-panel
  :items '("Red" "Green" "Blue")
  :interaction :multiple-selection))
```

Note that :no-selection is not a supported choice for list panels. To display a list of items with no selection possible you should use a display pane.

4.2.2 Extended selection

Application users often want to make single *and* multiple selections from a list. Some of the time they want a new selection to deselect the previous one, so that only one selection remains — just like a :single-selection panel. On

other occasions, they want new selections to be added to the previous ones — just like a :multiple-selection panel.

The **:extended-selection** interaction combines these two interactions. Here is an extended-selection list panel:

```
(contain
(make-instance
  'list-panel
  :items '("Item" "Thing" "Object")
  :interaction :extended-selection))
```

Before continuing, here are the definitions of a few terms. The action you perform to select a single item is called the *selection gesture*. The action performed to select additional items is called the *extension gesture*. There are two extension gestures. To add a single item to the selection, the extension gesture is a click of the left button while holding down the Control key. For selecting a range of items, it is a click of the left button whilst holding down the Shift key.

4.2.3 Deselection, retraction, and actions

As well as selecting items, users often want to deselect them. Items in multiple-selection and extended-selection lists may be deselected.

In a multiple-selection list, deselection is done by clicking on the selected item again with either of the selection or extension gestures.

In an extended-selection list, deselection is done by performing the extension gesture upon the selected item. (If this was done using the selection gesture, the list would behave as a single-selection list and all other selections would be lost.)

Just like a selection, a deselection — or *retraction* — can have a callback associated with it.

For a multiple-selection list pane, there may be the following callbacks:

- :selection-callback called when a selection is made
- :retract-callback called when a selection is retracted

Consider the following example. The function set-title changes the title of the interface to the value of the argument passed to it. By using this as the callback to the check-button-panel, the title of the interface is set to the current
selection. The function retract-callback returns a message dialog with the name of the button retracted.

```
(defun set-title (data interface)
  (setf (interface-title interface)
       (format nil "~S" data)))
(make-instance 'check-button-panel
       :items '(one two three four five)
       :print-function 'string-capitalize
       :selection-callback 'set-title
       :retract-callback 'test-callback)
(contain *)
```



Figure 4.5 An example of a callback to a check-button panel

For an extended-selection list pane, there may be the following callbacks:

- :selection-callback called when a selection is made
- :retract-callback called when a selection is retracted
- :extend-callback called when a selection is extended

Also available in extended-selection and single-selection lists is the action callback. This is called when you double-click on an item.

:action-callback — called when a double-click occurs

4.2.4 Selections

List panels — all choices, in fact — can have selections, and you can set them from within Lisp. This is useful for providing default settings in a choice, or when a user selection has an effect on other settings than just the one they made.

Choices

The selection is represented as a vector of offsets into the list of the choice's items, unless it is a single-selection choice, in which case it is just represented as an offset.

The initial selection is controlled with the initiang :selection. The accessor choice-selection is provided.

4.3 Graph panes

Another kind of choice is the graph-pane. This is a special pane that can draw graphs, whose nodes and edges can be selected, and for which callbacks can be specified, as usual.

Here is a simple example of a graph pane. It draws a small rooted tree:

```
(contain
 (make-instance
 'graph-pane
 :roots '(1)
 :children-function
 #'(lambda (x)
            (when (< x 8)
               (list (* 2 x) (1+ (* 2 x)))))))
```



Figure 4.6 A graph pane

The graph pane is supplied with a :children-function which it uses to calculate the children of the root node, and from those children it continues to calculate more children until the termination condition is reached. For more details of this, see the *CAPI Reference Manual*.

You can associate selection, retraction, extension, and action callbacks with any or all elements of a graph. Here is a simple graph pane that has an action callback on its nodes.

First we need a pane for displaying the callback messages in. This is done by executing the following code:

```
(defvar *the-collector*
  (capi:contain (make-instance 'capi:collector-pane)))
```

Then, define the following four callback functions:

Choices

```
(defun test-selection-callback (&rest args)
                (format (capi:collector-pane-stream *the-collector*)
                     "Selection"))
(defun test-extend-callback (&rest args)
                    (format (capi:collector-pane-stream *the-collector*)
                    "Extend"))
(defun test-retract-callback (&rest args)
                    (format (capi:collector-pane-stream *the-collector*)
                    "Retract"))
```

Now create an extended selection graph pane which uses each of these callbacks, the callback used depending on the action taken:

The selection callback function is called whenever any node in the graph is selected.

The extension callback function is called when the selection is extended by middle clicking on another node (thus selecting it too).

The retract callback function is called whenever an already selected node is deselected.

The action callback function is called whenever an action is performed on a node (that is, whenever it is double clicked on).

4.4 Option panes and drop-down list boxes

Option panes, created with the option-pane class, display the current selection from a single-selection list. When you click on the option pane, the list appears and you can make another selection from it. Once the selection is made, it is displayed in the option pane.

In Windows, an option-pane produces a drop-down list box.

Here is an example option pane, which shows the choice of one of five numbers. The initial selection is controlled with :selected-item.

```
(contain
(make-instance
'option-pane
:items '(1 2 3 4 5)
:selected-item 3
:title "One of Five:"))
```



Figure 4.7 An option pane

4.5 Menu components

Menus (covered in Chapter 6) can have components that are also choices. These components are groups of items that have an interaction upon selection just like other choices. The :interaction keyword is used to associate radio or check buttons with the group — with the values :single-selection and :multiple-selection respectively. By default, a menu component has an interaction of :no-selection.

See "Grouping menu items together" on page 46 for more details.

4.6 General properties of choices

The behaviors you have seen so far are mostly general properties of choices rather than being specific to a particular choice. These general properties are summarized below.

4.6.1 Interaction

All choices have a selection interaction, controlled by the <code>:interaction</code> initarg. The <code>radio-button-panel</code> and <code>check-button-panel</code> are simply <code>button-panels</code> with their interactions set appropriately. The interaction possibilities are listed below.

Set :interaction to :single-selection to force single selection. Only one item may be selected at a time: selecting an item deselects any other selected item.

Set :interaction to the value :multiple-selection to create a multiple selection choice element. This lets you select as many items as you want. A selected item may be deselected by clicking on it again.

Set :interaction to the value :extended-selection to create an extended selection element. This is a combination of the other two: Only one item may be selected, but the selection may be extended to more than one item.

Set :interaction to the value :no-selection to force no interaction. Note that this option is not available for list panels. To display a list of items with no selection you should use a display pane instead.

Specifying an interaction that is invalid for a particular choice causes a compilation error.

The accessor choice-interaction is provided for inspecting a choice's interaction.

4.6.2 Selections

All choices have a selection. This is a state representing the items currently selected. The selection is represented as a vector of offsets into the list of the choice's items, unless it is a single-selection choice, in which case it is just represented as an offset.

The initial selection is controlled with the initiarg :selection. The accessor choice-selection is provided.

Generally, it is easier to refer to the selection in terms of the items selected, rather than by offsets, so the CAPI provides the notion of a *selected item* and

the *selected items*. The first of these is the selected item in a single-selection choice. The second is a list of the selected items in any choice.

The accessors choice-selected-item and choice-selected-items and the initargs :selected-item and :selected-items provide access to these conceptual slots.

4.6.3 Callbacks

All choices can have callbacks associated with them. These callbacks are activated when the application user makes a selection, and different sorts of gesture can have different sorts of callback associated with them.

The following callbacks are available: :selection-callback, :retract-callback (called when a deselection is made), :extend-callback, and :action-callback (called when a double-click occurs). What makes one choice different from another is that they permit different combinations of these callbacks. This is a consequence of the differing interactions. For example, you cannot have an :extend-callback in a radio button panel, because you cannot extend selection in one.

Callbacks pass data to the function they call. There are default arguments for each type of callback. Using the :callback-type keyword allows you to change these defaults. It can take any of the following arguments.

:interface	causes the interface to be passed as an argument to the callback function.
:data	causes the value of the selected data to be passed to the callback function.
:item	causes the selected item to be passed as an argument to the callback function.

The following combinations of two of the above are also allowed – :interface-data, :interface-item, :data-interface and :item-interface. In each of these cases two arguments are passed to the callback function.

A final option is to pass no arguments, using the **:none** keyword The following example uses a push button and a callback function to display the arguments it receives.

```
(defun show-callback-args (arg1 arg2)
  (display-message "The arguments were ~S and ~S" arg1 arg2))
(setq example-button (make-instance 'push-button
                         :text "Push Me"
                         :callback 'show-callback-args
                         :data "Here is some data"
                         :callback-type :data-interface))
(contain example-button)
```

Try changing the :callback-type to other types.

If you do not use the :callback-type argument and you do not know what the default is, you can include a &rest args statement in your callback function to account for all the arguments that might be passed.

Specifying a callback that is invalid for a particular choice causes a compilation error.

Laying Out CAPI Panes

So far, you have seen how you can create a variety of different window elements using the CAPI. Up to now, though, you have only created interfaces which contain one of these elements. The CAPI provides a series of layout elements which allow you to combine several elements in a single window. This chapter provides an introduction to the different types of layout available and the ways in which each can be used.

Layouts are created just like any other CAPI element, by using makeinstance. Each layout must contain a description of the CAPI elements it contains, given as a list to the :description keyword.

A layout is used to group any instances of simple-pane and its subclasses (for instance all the elements you met in the last chapter), and pinboard object and its subclasses (discussed in Chapter 9, "Creating Your Own Panes"). Once again, you should make sure you have defined the test-callback function before attempting any of the examples in this chapter. Its definition is repeated here for convenience.

5.1 Organizing panes in columns and rows

You will frequently need to organize a number of different elements in rows and columns. The column-layout and row-layout elements are provided to make this easy.

The following is a simple example showing the use of column-layout.

```
(contain (make-instance 'column-layout
  :description (list
                (make-instance 'text-input-pane)
                (make-instance 'list-panel
                :items '(1 2 3 4 5)))))
```



Figure 5.1 An example of using column-layout

1. Define the following elements:

```
(setq button1 (make-instance 'push-button
        :data "Button 1"
        :callback 'test-callback))
(setq button2 (make-instance 'push-button
        :data "Button 2"
        :callback 'test-callback))
(setq editor (make-instance 'editor-pane))
(setq message (make-instance 'display-pane
        :text "A display pane"))
```

```
(setq text (make-instance 'text-input-pane
                :title "Text: "
                :title-position :left
                :callback 'test-callback))
```

These will be used in the examples throughout the rest of this chapter.

To arrange any number of elements in a column, create a layout using columnlayout, listing the elements you wish to use. For instance, to display title, followed by text and button1, type the following into a Listener:



Figure 5.2 A number of elements displayed in a column

To arrange the same elements in a row, simply replace column-layout in the example above with row-layout. If you run this example, note that the elements in the original column layout disappear as soon as the row layout is created: each CAPI element can only be on the screen once at any time. Putting it into a new layout will remove it from its previous parent.

Layouts can be given horizontal and vertical scroll bars, if desired; the keywords :horizontal-scroll and :vertical-scroll can be set to t or nil, as necessary.

When creating panes which can be resized (for instance, list panels, editor panes and so on) you can specify the size of each pane relative to the others by listing the proportions of each. This can be done via either the <code>:y-ratios</code> keyword (for column layouts) or the <code>:x-ratios</code> keyword (for row layouts).

```
(contain (make-instance 'column-layout
  :description (list
                                 (make-instance 'display-pane)
                                 (make-instance 'editor-pane)
                                (make-instance 'listener-pane))
  :y-ratios '(1 5 3)))
```

You may need to resize this window in order to see the size of each pane.

Note that the heights of the three panes are in the proportions specified. The **:x-ratios** keyword will adjust the width of panes in a row-layout in a similar way.

5.2 Other types of layout

Row and column layouts are the most basic type of layout class available in the CAPI, and will be sufficient for many things you want to do. A variety of other layouts are available as well, as described in this section.

5.2.1 Grid layouts

Whereas row and column layouts only allow you to position a pane horizontally *or* vertically (depending on which class you use), grid layouts let you specify both, thus allowing you to create a complete grid of different CAPI panes.

5.2.2 Simple layouts

Simple layouts control the layout of only one pane. Where possible, the pane is resized to fit the layout. Simple layouts are sometimes useful when you need to encapsulate a pane.

5.2.3 Pinboard layouts

Pinboard layouts allow you to position a pane anywhere within a window, by specifying the *x* and *y* coordinates of the pane precisely. They are a means of letting you achieve any effect which you cannot create using the other available layouts, although their use can be correspondingly more complex. They are discussed in more detail in Chapter 9, "Creating Your Own Panes".

5.3 Combining different layouts

You will not always want to arrange all your elements in a single row or column. You can include other layouts in the list of elements used in any layout, thus enabling you to specify precisely how panes in a window should be arranged.

For instance, suppose you want to arrange the elements in your window as shown in Figure 5.3. The two buttons are shown on the right, with the text input pane and a message on the left. Immediately below this is the editor pane.



Figure 5.3 A sample layout

The layout in Figure 5.3 can be achieved by creating two row layouts: one containing the display pane and a button, and one containing the text input pane and the other button, and then creating a column layout which uses these two row layouts and the editor.

```
(setq row1 (make-instance 'row-layout
            :description (list message button1)))
(setq row2 (make-instance 'row-layout
            :description (list text button2)))
(contain (make-instance 'column-layout
            :description
            (list row1 row2 editor)))
```



Figure 5.4 An instantiation of the sample layout

As you can see, creating a variety of different layouts is simple. This means that it is easy to experiment with different layouts, allowing you to concentrate on the interface design, rather than its code.

5.4 Constraining the size of layouts

The size of a layout (often referred to as its *geometry*) is calculated automatically on the basis of the size of each of its children. The algorithm used takes account of *hints* provided by the children, and from the description of the layout itself. Hints are specified with keyword arguments given to the panes when they are created.

The following four hints are recognized by all layouts:

:visible-min-width — the minimum width of the child :visible-max-width — the maximum width of the child :visible-min-height — the minimum height of the child :visible-max-height — the maximum height of the child

Hints can take arguments in a number of formats, which are described in full in the *CAPI Reference Manual*. When given a number, the layout is constrained

to that number of pixels. A number of characters can also be specified, as shown in the next example.

In the last section, you created a window with five panes, by combining row and column layouts. Now consider changing the definition of the editor pane so that it required to have a minimum size. This would be a sensible change to make, because editor panes need to be large enough to work with comfortably.

```
(setq editor2 (make-instance 'editor-pane
    :visible-min-width '(:character 30)
    :visible-min-height '(:character 10)))
```

You will now have to place this new editor into the layouts that contained the old one. Only one layout is affected: the overall column layout which brings all the panes together.

Before typing the following into the listener, you should close down all the windows created in this chapter in order to free up the instances of Button1, Button2 and so forth.



Figure 5.5 The result of resizing the sample layout

5.5 Advanced pane layouts

Until now you have used layouts for CAPI elements in which the constituents were displayed in fixed positions set out by the CAPI. In this chapter we will be looking at a number of ways in which users can select the layout and display of CAPI elements in an interface once an instance of the interface has been displayed, through the use of dividers, switchable layouts and tab layouts. Throughout this section we will be using three predefined panes, which you should define before proceeding.

5.5.1 Switchable layouts

A switchable layout allows you to place CAPI objects on top of one another and determine which object is displayed on top through Lisp code, possibly linked to a button or menu option through a callback. Switchable layouts are set up using a switchable-layout element in a make-instance. As with the other layouts, such as column-layout and row-layout, the elements to be organized are given as a list to the :description keyword. Here is an example:

```
(setq switching-panes (make-instance
                       'switchable-layout
                       :description (list red-pane green-pane)))
```

```
(contain switching-panes)
```

Note that the default pane to be displayed is the red pane, which was the first pane in the description list. The two panes can now be switched between Using switchable-layout-visible-child:

```
(setf (switchable-layout-visible-child switching-panes)
      green-pane)
(setf (switchable-layout-visible-child switching-panes)
     red-pane)
```

5.5.2 Tab layouts

In its simplest mode, a tab layout is similar to a switchable layout, except that each pane is provided with a labelled tab, like the tabs on filing cabinet folders or address books. If the tab is clicked on by the user, the pane it is attached to is pulled to the front.

```
(make-instance 'tab-layout
               :items (list (list "one" red-pane)
                            (list "two" green-pane)
                            (list "three" blue-pane))
               :print-function 'car
               :visible-child-function 'second)
(contain *)
```

The example needs the :print-function to be car, or else the tabs will be labelled with the object numbers of the panes as well as the title provided in the list.

However, a tab layout can also be used in a non-switchable manner, with each tab responding with a callback to alter the appearance of only one pane. In this mode the <code>:description</code> keyword is used to describe the main layout of the tab pane. In the following example the tabs alter the choice of starting node for one graph pane, by using a callback to the <code>graph-pane-roots</code> accessor:

5.5.3 Dividers

Sometimes you may wish to have two or more panes presented in a column layout, with a horizontal divider between them. This is to allow the user the option of resizing one pane into the space of the other. By clicking on the bar between the two panes produced in the layout below, and then dragging it up or down the panes are resized.

```
(contain (make-instance 'column-layout
            :description (list green-pane
            :divider red-pane)))
```

Dividers can also be placed between panes in a row-layout or even combinations of row and column layouts.

Creating Menus

You can create menus for an application using the menu class.

You should make sure you have defined the test-callback and hello functions before attempting any of the examples in this chapter. Their definitions are repeated here for convenience.

6.1 Creating a menu

A menu can be created in much the same way as any of the CAPI classes you have already met.

1. Type the following into a Listener:

```
(make-instance 'menu
  :title "Foo"
  :items '("One" "Two" "Three" "Four")
  :callback 'test-callback)
```

```
(make-instance 'interface
  :menu-bar-items (list *))
(display *)
```

This creates a CAPI interface with a menu, **Foo**, which contains four items. Choosing any of these items displays its arguments. Each item has the callback specified by the :callback keyword.

A submenu can be created simply by specifying a menu as one of the items of the top-level menu.

2. Type the following into a Listener:

```
(make-instance 'menu
  :title "Bar"
  :items '("One" "Two" "Three" "Four")
  :callback 'test-callback)
(make-instance 'menu
  :title "Baz"
  :items (list 1 2 * 4 5)
  :callback 'test-callback)
(contain *)
```

This creates an interface which has a menu, called **Baz**, which itself contains five items. The third item is another menu, **Bar**, which contains four items. Once again, selecting any item returns its arguments.

Menus can be nested as deeply as required using this method.

6.2 Grouping menu items together

The menu-component class lets you group related items together in a menu. This allows similar menu items to share properties, such as callbacks, and to be visually separated from other items in the menus. Menu components are actually choices.

Here is a simple example of a menu component. This creates a menu called **Items**, which has four items. **Menu 1** and **Menu 2** are ordinary menu items, but **Item 1** and **Item 2** are created from a menu component, and are therefore grouped together on the menu bar.

<u>l</u>tem 1 Item 2

Menu 2

Menu components allow you to specify, via the <code>:interaction</code> keyword, selectable menu items — either as multiple-selection or single-selection items. This is like having radio buttons or check boxes as items in a menu, and is a popular technique among many GUI-based applications.

Figure 6.1 A menu

The following example shows you how to include a panel of radio buttons in a menu.

```
(setq radio (make-instance 'menu-component
                  :interaction :single-selection
                 :items '("This" "That")
                 :callback 'hello))
```

```
(setq commands (make-instance 'menu
    :title "Commands"
    :items
    (list "Command 1" radio "Command 2")
    :callback 'test-callback))
```

```
(contain commands)
```

🛃 Co 💶 🗙		
<u>C</u> ommands		
<u>C</u> ommand 1		
<u>T</u> his		
✓ T <u>h</u> at		
Co <u>m</u> mand 2		

Figure 6.2 Radio buttons included in a menu

The menu items **This** and **That** are radio buttons, only one of which may be selected at a time. The other menu items are just ordinary commands, as you saw in the previous examples. Note that the CAPI automatically groups the items which are parts of a menu component so that they are separated from other items in the menu.

This example also illustrates the use of more than one callback in a menu, which of course is the usual case when you are developing real applications. Choosing either of the radio buttons displays one message on the screen, and choosing either **Command1** or **Command2** returns the arguments of the callback.

Checked menu items can be created by specifying :multiple-selection to the :interaction keyword, as illustrated below.

```
(contain (make-instance 'menu
  :title "Greek"
  :items (list letters)
  :callback 'test-callback))
```

✓ <u>A</u>lpha
 ✓ <u>B</u>eta

Figure 6.3 An example of checked menu items

Note how the items in the menu component inherit the callback given to the parent, eliminating the need to specify a separate callback for each item or component in the menu.

6.3 Creating individual menu items

The menu-item class lets you create individual menu items. These items can be passed to menu-components or menus via the :items keyword. Using this class, you can assign different callbacks to different menu items.

```
(setq test (make-instance 'menu-item
                        :title "Test"
                       :callback 'test-callback))
(setq hello (make-instance 'menu-item
                         :title "Hello"
                         :callback 'hello))
(setq group (make-instance 'menu-component
                         :items (list test hello)))
(contain group)
```

🛃 Co 💶	□×
<u>M</u> enu	
<u>T</u> est	
<u>H</u> ello	

Figure 6.4 Individual menu items

6.4 The CAPI menu hierarchy

The combination of menu items, menu components and menus creates a hierarchical structure as shown in Figure 6.5. Items in a menu inherit values from their parent, allowing similar elements to share relevant properties whenever possible.

The menu below was created using 5 menu items, 4 of which were grouped together in 2 menu-components. These menu-components were then combined with the fifth item in the menu itself. Note that the <code>:interaction</code> keyword was used with the second menu component to make menu item number three a radio button.



Figure 6.5 An example of a menu hierarchy

6.5 Disabling menu items

Like buttons, individual menu items, menus, and menu-components can be disabled by setting the :enabled keyword to nil.

In addition, a function can be specified via the :enabled-function keyword, that determines whether or not the menu, menu item, or menu component is enabled. Consider the following example:

```
(defvar *on* t)
(setq menu
    (make-instance
        'menu-item
        :title "Foo"
        :enabled-function #'(lambda (menu) *on*)))
(contain menu)
```

Toggling the value of *on* between t and nil in the listener, using setq, results in the menu item toggling between an enabled and disabled state.

Creating Menus

Defining Interface Classes

So far we have looked at various components for building interfaces. The CAPI provides all these and more, but instead of continuing with our exploration of the various classes provided, let us see how what we have learned so far can be combined into a single, non-trivial interface class.

7.1 The define-interface macro

The macro define-interface is used to define subclasses of interface, the superclass of all CAPI interface classes.

It is an extension to defclass, which provides the functionality of that macro as well as the specification of the panes, layouts, and menus from which an interface is composed. It takes the same arguments as defclass, and supports the additional options :panes, :layouts, :menus, and :menu-bar.

Each component of the interface is named in the code, and a slot of that name is added to the class created. When an instance of the class is made, each component is created automatically and placed in its slot.

When defining a component, you can use other components within the definition simply by giving its name. You can refer to the interface itself by the special name interface.

7.2 An example interface

Here is a simple example of interface definition done with define-interface:

An instance of this interface can be displayed as follows:

```
(make-instance 'demo)
```

(display *)

At the moment the buttons do nothing, but they will eventually do the following:

- **Open File** will bring up a file prompter and allow you to select a filename from a directory. Later on, we will add an editor pane to display the chosen file's contents.
- **Page Down** will scroll downwards so that you can view the lower parts of the file that cannot be seen initially.
- **Page Up** will scroll upwards so that you can return to parts of the file seen before.



Figure 7.1 A demonstration of a CAPI interface

Later on, we will specify callbacks for these buttons to provide this functionality.

The (:default-initargs :title "Demo") part at the end is necessary to give the interface a title. If no title is given, the default name is "Untitled CAPI Interface".

7.2.1 How the example works

Examine the define-interface call to see how this interface was built. The first part of the call to define interface is shown below:

(define-interface demo ()
 ()

This part of the macro is identical to defclass — you provide:

- The name of the interface class being defined
- The superclasses of the interface (defaulting to interface)
- The slot descriptions

The interesting part of the define-interface call occurs after these defclasslike preliminaries. The remainder of a define-interface call lists all elements that define the interface's appearance. Here is the :panes part of the definition:

```
(:panes
(page-up push-button
        :text "Page Up")
(page-down push-button
        :text "Page Down")
(open-file push-button
        :text "Open File"))
```

Two arguments — the name and the class — are required to produce a pane. You can supply slot values as you would for any pane.

Here is the :layouts part of the definition:

Three arguments — the name, the class, and any child layouts — are required to produce a layout. Notice how the children of the layout are specified by using their component names.

The interface information given so far is a series of specifications for panes and layouts. It could also specify menus and a menu bar. In this case, three buttons are defined. The layout chosen is a row layout, which displays the three buttons side by side at the top of the pane.

7.3 Adapting the example

The :panes and :layouts keywords can take a number of panes and layouts, each specified one after the other. By listing several panes, menus, and so on, complicated interfaces can be constructed quickly.

To see how simply this is done, let us add an editor pane to our interface. We need this to display the text contained in the file chosen with the **Open File** button.

The editor pane needs a layout. It could be added to the row-layout already built, or another layout could be made for it. Then, the two layouts would have to be put inside a third to contain them (see Chapter 5, *Laying Out CAPI Panes*).

The first thing to do is add the editor pane to the panes description. The old panes description read:

```
(:panes
(page-up push-button
        :text "Page Up")
(page-down push-button
        :text "Page Down")
(open-file push-button
        :text "Open File"))
```

The new one includes an editor pane named viewer.

```
(:panes
(page-up push-button
        :text "Page Up")
(page-down push-button
        :text "Page Down")
(open-file push-button
        :text "Open File")
(viewer editor-pane
        :title "File:"
        :text "No file selected."
        :visible-min-height '(:character 8)
        :reader viewer-pane))
```

This specifies the editor pane, with a stipulation that it must be at least 8 characters high. This allows you to see a worthwhile amount of the file being viewed in the pane.

Note the use of **:reader**, which defines a reader method for the interface which returns the editor pane. You can also specify writers or accessors in this way.

The interface also needs a layout for the editor pane in the layouts section. The old layouts description read:

The new one reads:

This creates another row-layout for the new pane and then encapsulates the two row layouts into a third column-layout called main-layout. This is used as the default layout, specified by setting the :layout initarg to main-layout in the :default-initargs section. If there is no default layout specified, define-interface uses the first one listed.

By putting the layout of buttons and the layout with the editor pane in a column layout, their relative position has been controlled: the buttons appear in a row above the editor pane.

The code for the new interface is now as follows:

```
(define-interface demo ()
  ()
 (:panes
  (page-up push-button
           :text "Page Up")
  (page-down push-button
             :text "Page Down")
  (open-file push-button
             :text "Open File")
  (viewer editor-pane
          :title "File:"
          :text "No file selected."
          :visible-min-height '(:character 8)
          :reader viewer-pane))
 (:layouts
   (main-layout column-layout
                '(row-of-buttons row-with-editor-pane))
   (row-of-buttons row-layout
                   '(page-up page-down open-file))
   (row-with-editor-pane row-layout
                         (viewer)))
  (:default-initargs :title "Demo"))
```

Displaying an instance of the interface by entering the line of code below produces the window in Figure 7.2:

(display (make-instance 'demo))



Figure 7.2 A CAPI interface with editor pane

7.3.1 Adding menus

To add menus to your interface you must first specify the menus themselves, and then a menu bar of which they will be a part.

Let us add some menus that duplicate the proposed functionality for the buttons. We will add:

- A File menu with a Open option, to do the same thing as Open File
- A **Page** menu with **Page Up** and **Page Down** options, to do the same things as the buttons with those names

The extra code needed in the define-interface call is this:

Menu definitions give a slot name for the menu, followed by the title of the menu, a list of menu item descriptions, and then, optionally, a list of keyword arguments for the menu.

In this instance the menu item descriptions are just strings naming each item, but you may wish to supply initialization arguments for an item — in which case you would enclose the name and those arguments in a list.

The menu bar definition simply names all the menus that will be on the bar, in the order that they will appear. By default, of course, the environment may add menus of its own to an interface — for example the **File** menu in the Common LispWorks environment.

The code for the new interface is:

```
(define-interface demo ()
 ()
 (:panes
   (page-up push-button
           :text "Page Up")
   (page-down push-button
             :text "Page Down")
   (open-file push-button
             :text "Open File")
   (viewer editor-pane
          :title "File:"
           :text "No file selected."
           :visible-min-height '(:character 8)
           :reader viewer-pane))
  (:layouts
    (main-layout column-layout
                '(row-of-buttons row-with-editor-pane))
    (row-of-buttons row-layout
                    '(page-up page-down open-file))
    (row-with-editor-pane row-layout
                          '(viewer)))
  (:menus
   (file-menu "File"
             ("Open"))
   (page-menu "Page"
              ("Page Up" "Page Down")))
 (:menu-bar file-menu page-menu)
  (:default-initargs :title "Demo"))
```

🛃 Demo		_ 🗆 🗙
<u>File Page</u>	Page Down	Open File
File:	- ago bonni	

Figure 7.3 A CAPI interface with menu items

The menus contain the items specified — try it out to be sure.

7.4 Connecting an interface to an application

Having defined an interface in this way, you can connect it up to your program using callbacks, as described in earlier chapters. Here we define some functions to perform the operations we required for the buttons and menus, and then hook them up to the buttons and menus as callbacks.

The functions to perform the page scrolling operations are given below:

```
(defun scroll-up (data interface)
 (call-editor (viewer-pane interface)
          "Scroll Window Up"))
(defun scroll-down (data interface)
     (call-editor (viewer-pane interface)
          "Scroll Window Down"))
```

The functions use the CAPI function call-editor which calls an editor command (given as a string) on an instance of an editor-pane. The editor commands Scroll Window Up and Scroll Window Down perform the necessary operations for Page Up and Page Down respectively.

The function to perform the file-opening operation is given below:

```
(defun file-choice (data interface)
  (let ((file (prompt-for-file "Select A File:")))
    (when file
    (setf (titled-pane-title (viewer-pane interface))
    (format nil "File: ~S" file))
    (setf (editor-pane-text (viewer-pane interface))
    (with-open-file (stream file)
       (let ((buffer (make-array 1024 :element-type (stream-
element-type stream)
                    :adjustable t
                    :fill-pointer 0)))
         (do ((char (read-char stream nil nil) (read-char stream
nil nil)))
       ((null char))
       (vector-push-extend char buffer))
         (subseq buffer 0)))))))
```

This function prompts for a filename and then displays the file in the editor pane.

The function first produces a file prompter through which a file may be selected. Then, the selected file name is shown in the title of the editor pane (using titled-pane-title). Finally, the file name is used to get the contents of the file and display them in the editor pane (using editor-pane-text).

The correct callback information for the buttons is specified as shown below:
```
(:panes
 (page-up push-button
        :text "Page Up"
        :selection-callback 'scroll-up)
 (page-down push-button
        :text "Page Down"
        :selection-callback 'scroll-down)
 (open-file push-button
        :text "Open File"
        :selection-callback 'file-choice)
 (viewer editor-pane
        :title "File:"
        :text "No file selected."
        :visible-min-height '(:character 8)
        :reader viewer-pane))
```

All the buttons and menu items operate on the editor pane viewer. A reader is set up to allow access to it.

The correct callback information for the menus is specified as shown below:

```
(:menus
 (file-menu "File"
                (("Open"))
               :selection-callback 'file-choice)
 (page-menu "Page"
                (("Page Up"
                    :selection-callback 'scroll-up)
                ("Page Down"
                    :selection-callback 'scroll-down))))
```

In this case, each item in the menu has a different callback. The complete code for the interface is listed below — try it out.

```
(define-interface demo ()
  ()
  (:panes
      (page-up push-button
               :text "Page Up"
               :selection-callback 'scroll-up)
      (page-down push-button
                 :text "Page Down"
                 :selection-callback 'scroll-down)
      (open-file push-button
                 :text "Open File"
                 :selection-callback 'file-choice)
      (viewer editor-pane
              :title "File:"
              :text "No file selected."
              :visible-min-height '(:character 8)
              :reader viewer-pane))
   (:layouts
     (main-layout column-layout
                  '(row-of-buttons row-with-editor-pane))
     (row-of-buttons row-layout
                     '(page-up page-down open-file))
     (row-with-editor-pane row-layout
                           (viewer)))
   (:menus
     (file-menu "File"
                (("Open"))
                :selection-callback 'file-choice)
     (page-menu "Page"
                (("Page Up"
                  :selection-callback 'scroll-up)
                 ("Page Down"
                  :selection-callback 'scroll-down))))
   (:menu-bar file-menu page-menu)
   (:default-initargs :title "Demo"))
```

Prompting for Input

A dialog is a window that receives some input from the user and returns it to the application. For instance, if the application wants to know where to save a file, it could prompt the user with a file dialog. Dialogs can also be cancelled, meaning that the application should cancel the current operation.

In order to let you know whether or not the dialog was cancelled, CAPI dialogs always return two values. The first value is the return value itself, and the second value is t if the dialog returned normally and mil if the dialog was cancelled.

The CAPI provides both a large set of predefined dialogs and the means to create your own. This chapter takes you through some example uses of the predefined dialogs, and then shows you how to create custom built dialogs.

8.1 Some simple dialogs

The simplest form of dialog is a message dialog. The function displaymessage behaves very much like format.

(display-message "Hello world")



Figure 8.1 A message dialog

```
(display-message
"This function is ~S"
'display-message)
```



Figure 8.2 A second message dialog

Another simple dialog asks the user a question and returns t or mil depending on whether the user has chosen yes or no. This function is confirm-yesor-no.

```
(confirm-yes-or-no
    "Do you own a pet?")
```



Figure 8.3 A message dialog prompting for confirmation

8.2 Prompting for values

The CAPI provides a number of different dialogs for accepting values from the user, ranging from accepting strings to accepting whole Lisp forms to be evaluated.

8.2.1 Prompting for strings

The simplest of the CAPI prompting dialogs is prompt-for-string which returns the string you enter into the dialog.

```
(prompt-for-string
"Enter a string:")
```

×	(
	1
<u>C</u> ancel]
	× <u>C</u> ancel

Figure 8.4 A dialog prompting for a string

An initial value can be placed in the dialog by specifying the keyword argument :initial-value.

8.2.2 Prompting for integers

The CAPI also provides a number of more specific dialogs that allow you to enter other types of data. For example, to enter an integer, use the function prompt-for-integer. Only integers are accepted as valid input for this function.

```
(prompt-for-integer
"Enter an integer:")
```

There are a number of extra options which allow you to specify more strictly which integers are acceptable. Firstly, there are two arguments :min and :max which specify the minimum and maximum acceptable integers.

```
(prompt-for-integer
"Enter an integer:"
:min 10 :max 20)
```

If this does not provide enough flexibility you can specify a function that validates the result with the keyword argument :ok-check. This function is passed the current value and must return non-nil if it is a valid result.

```
(prompt-for-integer
"Enter an integer:"
:ok-check 'oddp)
```

8.2.3 Prompting for an item in a list

If you would like the user to select an item from a list of items, the function prompt-with-list should handle the majority of cases. The simplest form just passes a list to the function and expects a single item to be returned.

```
(prompt-with-list
'(:red :yellow :blue)
"Select a color:")
```



Figure 8.5 A dialog prompting for a selection from a list

You can also specify the interaction style that you would like for your dialog, which can be any of the interactions accepted by a choice. The specification of the interaction style to this choice is made using the keyword argument :interaction:

```
(prompt-with-list
'(:red :yellow :blue)
"Select a color:"
:interaction :multiple-selection)
```

By default, the dialog is created using a list-panel to display the items, but the keyword argument :choice-class can be specified with any choice pane. Thus, for instance, you can present a list of buttons.

```
(prompt-with-list
  '(:red :yellow :blue)
  "Select a color:"
  :interaction :multiple-selection
  :choice-class 'button-panel)
```



Figure 8.6 Selection from a button panel

Finally, as with any of the prompting functions, you can specify additional arguments to the pane that has been created in the dialog. Thus to create a column of buttons instead of the default row, use:

```
(prompt-with-list
 '(:red :yellow :blue)
 "Select a color:"
 :interaction :multiple-selection
 :choice-class 'button-panel
 :pane-args
 '(:layout-class column-layout))
```

🛃 Listener 1	×
Select a color:	<u>0</u> K
	All
	<u>N</u> one
	<u>C</u> ancel

Figure 8.7 Selection from a column of buttons

8.2.4 Prompting for files

To prompt for a file, use the function prompt-for-file:

```
(prompt-for-file
 "Enter a file:")
```

You can also specify a starting pathname:

```
(prompt-for-file
  "Enter a filename:"
  :pathname "c:\\windows\\")
```

Enter a filen	ame:			? ×
Look <u>i</u> n:	🗐 System (C	2)	▼ €	
Hqbin Lispwork S Mosaic Program Softquad	cs Files d	unzipped Users Windows Autoexec.bat Autoexec.syd Command.com	E N	letlog.txt
•				Þ
File <u>n</u> ame: Files of <u>t</u> ype:	Windows All Files (*.*)		_	<u>O</u> pen Cancel

Figure 8.8 Selection of a file

8.2.5 Prompting for Lisp objects

The CAPI provides a number of dialogs specifically designed for creating Lisp aware applications. The simplest is the function prompt-for-form which accepts an arbitrary Lisp form and optionally evaluates it.

```
(prompt-for-form
 "Enter a form to evaluate:"
 :evaluate t)
(prompt-for-form
 "Enter a form to evaluate:"
 :evaluate nil)
```

Another useful function is prompt-for-symbol which prompts the user for an existing symbol. The simplest usage accepts any symbol, as follows:

```
(prompt-for-symbol
 "Enter a symbol:")
```

If you have a list of symbols from which to choose, then you can pass promptfor-symbol this list with the keyword argument :symbols. Finally, using :ok-check you can accept only certain symbols. For example, to only accept a symbol which names a class, use:

```
(prompt-for-symbol
"Enter a symbol:"
:ok-check #'(lambda (symbol)
                                 (find-class symbol nil)))
```

8.3 Creating your own dialogs

The CAPI provides a number of built-in dialogs which should cover the majority of most peoples needs. However, there is always the occasional need to create custom built dialogs, and the CAPI makes this very simple, using the function display-dialog which displays any CAPI interface as a dialog, and the functions exit-dialog and abort-dialog as the means to return from such a dialog.

8.3.1 Using display-dialog

Here is a very simple example that displays a **Cancel** button in a dialog, and when that button is pressed the dialog is cancelled. Note that display-dialog must receive an interface, so an interface is created for the button by using the function make-container.

Figure 8.9 A cancel button

The function abort-dialog cancels the dialog returning the values nil and nil, which represent a return result of nil and the fact that the dialog was cancelled, respectively. Note also that abort-dialog accepts any values and just ignores them.

The next problem is to create a dialog that can return a result. Use the function exit-dialog which returns the value passed to it from the dialog. The example below shows a simple string prompter.

Both of these examples are very simple, so here is a slightly more complicated one which creates a column containing both a text-input-pane and a **Cancel** button.

```
(display-dialog
(make-container
(list
  (make-instance
    'text-input-pane
    :callback-type :data
    :callback 'exit-dialog)
  (make-instance
    'push-button
    :text "Cancel"
    :callback 'abort-dialog))
:title "Enter a string:"))
```

Note that this looks very similar to the dialog created by prompt-for-string except for the fact that it does not provide the standard **OK** button. It would be simple enough to add the **OK** button, but since almost every dialog needs these standard buttons, the CAPI provides a higher level function called popup-confirmer that adds all of the standard buttons for you. This function is discussed in the next section.

8.3.2 Using popup-confirmer

The function popup-confirmer is a higher level function provided to add the standard buttons to user dialogs, and it is nearly always used in preference to display-dialog. In order to create a dialog using popup-confirmer, all you need to do is to supply a pane to be placed inside the dialog along with the buttons and the title. The function also expects a title, like all of the prompter functions described earlier.

```
(popup-confirmer
(make-instance
  'text-input-pane
  :callback-type :data
  :callback 'exit-dialog)
"Enter a string")
```

A common thing to want to do with a dialog is to get the return value from some state in the pane specified. For instance, in order to create a dialog that prompts for an integer the string entered into the text-input-pane would need to be converted into an integer. It is possible to do this once the dialog has returned, but popup-confirmer has a more convenient mechanism. The function provides a keyword argument, :value-function, which gets passed the pane, and this function should return the value to return from the dialog. It can also indicate that the dialog cannot return by returning a second value which is non-nil.

In order to do this conversion, popup-confirmer provides an alternative exit function to the usual exit-dialog. This is called exit-confirmer, and it does all of the necessary work on exiting.

You now have enough information to write a primitive version of promptfor-integer.

```
(defun text-input-pane-integer (pane)
 (let* ((text
            (text-input-pane-text pane))
            (integer
               (parse-integer
               text
               :junk-allowed t)))
 (or (and (integerp integer) integer)
            (values nil t))))
```

```
(popup-confirmer
 (make-instance
  'text-input-pane
  :callback 'exit-confirmer)
 "Enter an integer:"
  :value-function 'text-input-pane-integer)
```

Note that the dialog's **OK** button never becomes activated, yet pressing Return once you have entered a valid integer *will* return the correct value. This is because the **OK** button is not being dynamically updated on each keystroke in the text-input-pane so that it activates when the text-input-pane contains a valid integer. The activation of the **OK** button is recalculated by the function redisplay-interface, and the CAPI provides a standard callback, :redisplay-interface, which calls this as appropriate.

Thus, to have an **OK** button that becomes activated and deactivated dynamically, you need to specify the change-callback for the text-input-pane to be **:redisplay-interface**.

```
(popup-confirmer
(make-instance
    'text-input-pane
    :change-callback :redisplay-interface
    :callback 'exit-confirmer)
"Enter an integer:"
    :value-function 'text-input-pane-integer)
```

Note that the **OK** button now changes dynamically so that it is only ever active when the text in the text-input-pane is a valid integer.

The next thing that you might want to do with your integer prompter is to make it accept only certain values. For instance, you may only want to accept negative numbers. This can be specified to popup-confirmer by providing a validation function with the keyword argument :ok-check. This function receives the potential return value (the value returned by the value function) and it must return non-nil if that value is valid. Thus to accept only negative numbers we could pass minusp as the :ok-check.

(popup-confirmer (make-instance 'text-input-pane :change-callback :redisplay-interface :callback 'exit-confirmer) "Enter an integer:" :value-function 'text-input-pane-integer :ok-check 'minusp) Prompting for Input

Creating Your Own Panes

The CAPI provides a wide range of built-in panes, but it is still fairly common to need to create panes of your own. In order to do this, you need to specify both the input behavior of the pane (how it reacts to keyboard and mouse events) and its output behavior (how it displays itself). The class output-pane is provided for this purpose.

An output-pane is a fully functional graphics port. This allows it to use all of the graphics ports functionality to create graphics, and it also has a powerful input model which allows it to receive mouse and keyboard input.

9.1 Displaying graphics

The following is a simple example demonstrating how to create an outputpane and then how to draw a circle on it.

```
(setq output-pane
  (contain
      (make-instance 'output-pane)
     :best-width 300
     :best-height 300)))
```



Figure 9.1 An empty output pane

Now you can draw a circle in the empty output pane by using the drawcircle function from the graphics package.

```
(gp:draw-circle output-pane 100 100 50)
```



Figure 9.2 An output pane containing a circle

Notice that this circle is not permanently drawn on the output-pane, and when the window is next redisplayed it vanishes. To prove this to yourself, force the window to be redisplayed (for example by iconizing or resizing it). At this point, you can draw the circle again yourself but it will not happen automatically.

```
(gp:draw-circle output-pane 100 100 50)
```

In order to create a permanent display, you need to provide a function to the output-pane that is called to redraw sections of the output-pane when they are exposed. This function is called the *display callback*. When the CAPI needs to redisplay a region of an output-pane, it calls that output-pane's display-callback function, passing it the output-pane and the region in question.

For example, to create a pane that has a permanent circle drawn inside it, do the following:

Notice that the callback in this example ignores the region that needs redrawing and just redraws everything. This is possible because the CAPI clips the drawing to the region that needs redisplaying, and hence only the needed part of the drawing gets done. For maximum efficiency, it would be better to only draw the minimum area necessary.

The keywords :best-width and :best-height keywords specify the initial width and height of the circle pinboard object. More detail can be found in the *CAPI Reference Manual*.

Now that we can create output-panes with our own display functions, we can create a new class of window by using defclass as follows.

```
(defclass circle-pane (output-pane)
  ()
  (:default-initargs
  :display-callback 'draw-a-circle))
(contain
  (make-instance 'circle-pane))
```

9.2 Receiving input from the user

You now know enough to be able to create new classes of window which can display arbitrary graphics, but to be able to create interactive windows you need to be able to receive events. The CAPI supports this through the use of an input model, which is a mapping of events to the callbacks that should be run when they occur.

When the event callback is called, it gets passed the output-pane and the *x* and *y* coordinates of the mouse pointer at the time of the event. A few events also pass additional information as necessary; for example, keyboard events also pass the key that was pressed.

For example, we can create a very simple drawing pane by adding a callback to draw a point whenever the left button is dragged across the pane. This is done as follows:

```
(contain
(make-instance
    'output-pane
    :input-model '(((:motion :button-1)
               gp:draw-point))))
```



Figure 9.3 An interactive output pane

The input model above seems quite complicated, but it is just a list of event to callback mappings, where each one of these mappings is a list containing an event specification and a callback. An event specification is also a list containing keywords specifying the type of event required.

9.3 Creating graphical objects

A common feature needed by an application is to have a number of objects displayed in a window and to make events affect the object underneath the cursor. The CAPI provides the ability to create graphical objects, to place them into a window at a specified size and position, and to display them as necessary. Also a function is provided to determine which object is under any given point so that events can be dispatched correctly.

These graphical objects are called *pinboard objects*, as they can only be displayed if they are contained within a pinboard-layout. To define a pinboardobject, you define a subclass of drawn-pinboard-object and specify a drawing routine for it (and you can also specify constraints on the size of your object). You can then make instances of these objects and place them into layouts just as if they were ordinary panes. You can also place these objects inside layouts as long as there is a pinboard-layout higher up the layout hierarchy that contains the panes.

Here is an example of the built-in pinboard object class item-pinboardobject which displays its text like a title-pane. Note that the function contain always creates a pinboard-layout as part of the wrapper for the object to be contained, and so it is possible to contain pinboard-objects in just the same way as you can contain other classes of CAPI object.



Figure 9.4 A pinboard object

9.3.1 The implementation of graph panes

One of the major uses the CAPI itself makes of pinboard-objects is to implement graph-panes. The graph-pane itself is a pinboard-layout and it is built using pinboard-objects for the nodes and edges. This is because each node (and sometimes each edge) of the graph needs to react individually to the user. For instance, when an event is received by the graph-pane, it is told which pinboard object was under the pointer at the time, and it can then use this information to change the selection. Create the following graph-pane and notice that every node in the graph is made from an *item-pinboard-object* as described in the previous section and that each edge is made from a *line-pinboard-object*.

```
(defun node-children (node)
  (when (< node 16)
      (list (* node 2)
                               (1+ (* node 2)))))
(contain
  (make-instance
      'graph-pane
    :roots '(1)
    :children-function 'node-children)
:best-width 300 :best-height 400)
```



Figure 9.5 A graph pane with pinboard object nodes

As mentioned before, pinboard-layouts can just as easily display ordinary panes inside themselves, and so the graph-pane provides the ability to specify the class used to represent the nodes. As an example, here is a graph-pane with the nodes made from push-buttons.

```
(contain
 (make-instance
 'graph-pane
 :roots '(1)
 :children-function 'node-children
 :node-pinboard-class 'push-button)
 :best-width 300 :best-height 400)
```



Figure 9.6 A graph pane with push-button nodes

9.3.2 An example pinboard object

To create your own pinboard objects, the class drawn-pinboard-object is provided, which is a pinboard-object that accepts a display-callback to display itself. The following example uses this class to create a new class of pinboardobject that displays an ellipse.

```
(defun draw-ellipse-pane (gp pane
                          ху
                          width height)
 (with-geometry pane
    (let ((x-radius
           (floor %width% 2))
          (y-radius
           (floor %height% 2)))
      (gp:draw-ellipse
      gp
       (+ %x% x-radius)
       (+ %y% y-radius)
      x-radius y-radius))))
(defclass ellipse-pane
     (drawn-pinboard-object)
 ()
 (:default-initargs
   :display-callback 'draw-ellipse-pane
   :min-width 50
   :min-height 50))
(contain
 (make-instance 'ellipse-pane))
```



Figure 9.7 An ellipse-pane class

The with-geometry macro is used to set the size and position, or geometry, of the ellipse drawn by the draw-ellipse-pane function. See the *CAPI Reference Manual* for more details.

Now that you have a new ellipse-pane class, you can create instances of them and place them inside layouts. For instance, the example below creates nine ellipse panes and place them in a three by three grid.



Figure 9.8 Nine ellipse-pane classes in a layout

10

Graphics Ports

10.1 Introduction

Graphics Ports allow users to write source-compatible applications for different host window systems. Graphics Ports are the destinations for drawing primitives. They are implemented with a generic host-independent part and a small host-specific part.

Graphics Ports implement a set of drawing functions and a mechanism for specifying the graphics state to be used in each drawing function call. There are two types of Graphics Port: on-screen ports and off-screen ports. Onscreen graphics ports correspond to visible windows, while off-screen graphics ports can be used for building up graphical images for subsequent copying to the screen.

See the *CAPI Reference Manual* for full reference entries on all the Graphics Port functions, macros, classes and types.

10.1.1 The package

All graphics port symbols are interned in and exported from the graphicsports package, nicknamed gp.

10.1.2 The system

The graphics ports system is available in the default LispWorks image. All symbols in the gp package are available to you as soon as you start LispWorks.

10.1.3 Creating instances

Pixmap graphics ports are usually required only temporarily and the macro with-pixmap-graphics-port allocates one for you from a cache of such objects.

10.2 Features

The main features of graphics-ports are:

- 1. Each port has a "graphics state" which holds all the information about drawing parameters such as line thickness, fill pattern, line-end-style etc. A graphics state object can also be created independently of any particular graphics port.
- **2.** The graphics state contents can either be enumerated in each drawing function call, bound to values for the entirety of a set of calls, or permanently changed.
- **3.** The graphics state includes a transform which implements generalized coordinate transformations on the port's coordinates.
- **4.** Off-screen ports can compute the horizontal and vertical bounds of the results of a set of drawing function calls, thus facilitating image or pixmap generation.

10.3 Graphics state

The graphics-state object associated with each port holds values for the following parameters:

Parameter Name	Default Value	Allowed Values
transform	the unit trans- form	Anything returned by the transform functions. (See below)
foreground	:black	A color spec, pixel, or symbol
background	:white	A color spec, pixel, or symbol
operation	boole-1	Boole constants (Chapter 12 of the ANSI standard)
thickness	1	number
scale-thickness	t	(member nil t)
dashed	nil	(member nil t)
dash	′(44)	(sequence integer integer)
line-end-style	:butt	(member :butt :round :projecting)
line-joint-style	:miter	(member :bevel :miter :round)
mask	nil	<pre>nil or a list of the form (x y width height)</pre>
font	nil	nil or a valid font

Table 10.1 Parameters held in a graphics-state object

transform An object which determines the coordinate transformation applying to the graphics port. The default value leaves the port coordinates unchanged from those used by the host window system — origin at top left, X increasing to the right and Y increasing down the screen. Section 10.4 on page 93 describes these objects.

Graphics Ports

foreground	Determines the foreground color used in drawing func- tions. A color can be a pixel value, a color name symbol, a color name string or a color object. Using pixel values results in better performance.
background	Determines the background color used in drawing functions which use a stipple. Valid values are the same as for foreground.
operation	The combination function used in the drawing primi- tives. Valid values are 0 to 15, being the same logical values as the op arg to the Common Lisp function boole (CLtL2).
thickness	The thickness of lines drawn. If scale-thickness is non-nil, the value is in port (transformed) coordinates, otherwise it is in pixels.
scale-thickness	If non-nil means interpret the thickness parameter in transformed port coordinates, otherwise interpret it in pixels.
dashed	If non-nil draws a dashed line using dash as the mark- space specifier.
dash	If non-nil should be a two element list specifying the mark and space for dashed lines. The mark and space values are interpreted in pixels only.
line-end-style	One of :butt :round Or :projecting and specifies how to draw the ends of lines.
line-joint-style	One of :bevel :miter :round and specifies how to draw the areas where the edges of polygons meet.
mask	Either nil or a list of the form (x y width height), defin- ing a rectangle inside which the drawing is done. The mask is not tiled. A mask is not transformed by the transform parameter.
mask-x	An integer specifying where in the port the X coordi- nate of the mask origin is to be considered to be. The value is in window coordinates.

fontEither nil or a valid font name or font object to be used
by the draw-character and draw-string functions. A
valid font is a portable font description. See Section 10.6
on page 96.

10.3.1 Setting the graphics state

The graphics state values associated with a drawing function call to a graphics port are set by one of three mechanisms.

1. Enumeration in the drawing function call. For example:

```
(draw-line port 1 1 100 100
    :thickness 10
    :scale-thickness nil
    :foreground :red)
```

2. Bound using the with-graphics-state macro. For example:

```
(with-graphics-state (port :thickness 10
                                :scale-thickness nil
                         :foreground :red)
        (draw-line port 1 1 100 100)
        (draw-rectangle port 2 2 40 50 :filled t))
```

3. Set by the set-graphics-state function. For example:

The first two mechanisms change the graphics state temporarily. The last one changes it permanently, effectively altering the "default" state.

10.4 Graphics state transforms

Coordinate systems for windows generally have the origin (0,0) positioned at the upper left corner of the window with X positive to the right and Y positive downwards. This is the "window coordinates" system. Generalized coordinates are implemented using scaling, rotation and translation operations such that any Cartesian coordinates can be used within a window. The Graphics Ports system uses a transform object to achieve this.

10.4.1 Generalized points

An (x, y) coordinate pair can be transformed to another pair in another coordinate system by scaling, rotation and transformation. The first two can be implemented using 2 x 2 matrices to hold the coefficients:

If the point *P* is (a, b) and it is transformed to the point Q(a', b')

$$P \Rightarrow Q \text{ or } (a, b) \text{ fi } (a', b')$$
$$a' = pa + rb, b' = qa + sb.$$
$$Q = PM, \text{ where } M = \left| \begin{array}{c} p & q \\ r & s \end{array} \right|$$

Translation can be included in this if the points *P* and *Q* are regarded as 3-vectors instead of 2-vectors, with the 3rd element being unity:

$$Q = PM$$

$$= (a \ b \ 1) \qquad p \ q \ 0$$

$$r \ s \ 0$$

$$u \ v \ 1$$

The coefficients *u* and *v* specify the translation.

So, the six elements (p, q, r, s, u, and v) of the 3 x 3 matrix contain all the transformation information. These elements are stored in a list in the graphics state slot transform.

Transforms can be combined by matrix multiplication to effect successions of translation, scaling and rotation operations.

Functions are provided in Graphics Ports which apply translation, scaling and rotation to a transform, combine transforms by pre- or post-multiplication, invert a transform, and so on.

10.4.2 Drawing functions

The scan-line conversions of the drawing functions are very much hostdependent. In other words, you cannot assume that, for example (draw-point port x y) has exactly the same effect on all machines. Some machines might put pixels down and to the right of integer coordinates (x y) while others may center the pixel at (x y).

See also the *LispWorks Reference Manual* entries for draw-circle (which draws a circle) and draw-ellipse (which draws an ellipse), and the *CAPI Reference Manual* for full reference entries of all the drawing functions.

10.5 Pixmap graphics ports

Pixmap graphics ports are drawing destinations which exist only as pixel arrays whose contents are not directly accessible. They can be drawn to using the draw-thing functions, and their contents can be copied onto other graphics ports. However this copying can be meaningless unless the conversion of colors uses the same color device on both ports. Because color devices are associated with regular graphics ports (Windows) rather than pixmap graphics ports, you have to connect a pixmap graphics port to a regular graphics port for color conversion. This is the purpose of the *owner* slot in pixmapgraphics-port-mixin. The conversion of colors to pixel values is done in the same way as for regular graphics ports, but the pixmap graphics port's owner is used to find a color device. You can draw to pixmap graphics ports using pre-converted colors to avoid color conversion altogether, in which case a null color owner is OK for a pixmap graphics port.

10.5.1 Relative drawing in pixmap graphics ports

Many of the drawing functions have a *relative* argument. If non-nil, it specifies that when drawing functions draw to the pixmap, the extremes of the pixel coordinates reached are accumulated. If the drawing strays beyond any edge of the pixmap port (into negative coordinates or beyond its width or height), then the drawing origin is shifted so that it all fits on the port. If the drawing extremes exceed the total size available, some are inevitably lost. If *relative* is nil, any part of the drawing which extends beyond the edges of the pixmap is lost. If *relative* is nil and *collect* non-nil, the drawing bounds are collected for later reading, but no relative shifting of the drawing is performed. The collected bounds are useful when you need to know the graphics motion a series of drawing calls causes. The *rest* args are host-dependent. They usually include a :width and :height pair.

10.6 Portable font descriptions

Portable font descriptions are designed to solve the following problems:

- Specify enough information to uniquely determine a real font.
- Query which real fonts match a partial specification.
- Allow font specification to be recorded and reused in a later run.

All the symbols described below are exported from the gp package. Font objects returned by find-matching-fonts and find-best-font can be used in calls to the Graphics Ports drawing functions and as the :font argument for CAPI panes.

10.6.1 Font attributes and font descriptions

Font attributes are properties of a font, which can be combined to uniquely specify a font on a given platform. There are some portable attributes which can be used on all platforms; other attributes are platform specific and will be ignored or signal errors when used on the wrong platform.

Font descriptions are externalizable objects which contain a set of font attributes. When using a font description in a font lookup operation, missing attributes are treated as wildcards (as are those with value :wild) and invalid attributes signal errors. The result of a font lookup contains all the attributes needed to uniquely specify a font on that platform.

Fonts are the objects used in drawing operations. They are made by a font lookup operation on a pane, using a font description as a pattern.

These are the current set of portable font attributes and their portable types:

Attribute	Possible values	Comments
:family	string	Values are not portable.
:weight	(member :normal :bold)	
:slant	(member :roman :italic)	
:size	(or (eql :any) (integer 0 *))	:any means a scalable font
:charset	keyword	

Table 10.2 Set of portable font attributes

Graphics Ports
11

The Color System

11.1 Introduction

The LispWorks Color System allows applications to use keyword symbols as aliases for colors in Graphics Ports drawing functions. They can also be used for backgrounds and foregrounds of windows and CAPI objects.

For example, the call

```
(gp:draw-line my-port x1 y1 x2 y2
:foreground :navyblue)
```

uses the keyword symbol :navyblue for the color of the line.

Colors are looked up in a color database. The LispWorks image is delivered with a large color database already loaded (approximately 650 entries.) The color database contains color-specs which define the colors in terms of a standard color model. When the drawing function is executed, the color-spec is converted into a colormap index (or "pixel" value).

The LispWorks Color System has facilities for:

- Defining new color aliases in one of several color models
- Loading the color database from a file of color descriptions
- Converting color specifications between color models

• Defining new color models

It is accessible from the COLOR package, and all symbols described in this chapter are assumed to be external to this package unless otherwise stated.

The color-models available by default are RGB, HSV and GRAY.

11.2 Reading the color database

To find out what colors are defined in the color database, use the following functions:

apropos-color-names

Function

Function

Function

apropos-color-names substring

This returns a list of symbols whose symbol-names contain *substring* and which are present in the color-database defining color aliases. By convention these are in the keyword package.

```
TEST-4> (color:apropos-color-names "RED")
(:ORANGERED3 :ORANGERED1 :INDIANRED3 :INDIANRED1
:PALEVIOLETRED :RED :INDIANRED :INDIANRED2
:INDIANRED4 :ORANGERED :MEDIUMVIOLETRED
:VIOLETRED :ORANGERED2 :ORANGERED4 :RED1 :RED2 :RED3
:RED4 :PALEVIOLETRED1 :PALEVIOLETRED2 :PALEVIOLETRED3
:PALEVIOLETRED4 :VIOLETRED3 :VIOLETRED1 :VIOLETRED2
:VIOLETRED4)
```

apropos-color-alias-names

apropos-color-alias-names substring

This functions like apropos-color-names but returns only those symbols that have been defined as color aliases.

apropos-color-spec-names

apropos-color-spec-names substring

This functions like apropos-color-names but returns only those symbols that were defined as original entries in the color database.

Function

get-all-color-names

get-all-color-names & optional sort

This returns a list of all color-names in the color database. By convention these are symbols in the keyword package. The returned list is alphanumerically sorted on the symbol-names if the optional argument is non-nil.

11.3 Color specs

A color spec is an object which numerically defines a color in some colormodel. For example the object returned by the call:

```
(color:make-rgb 0.0s0 1.0s0 0.0s0) =>
#(:RGB 0.0s0 1.0s0 0.0s0)
```

defines the color green in the RGB color model. (Note that short-floats are used; this results in the most efficient color conversion process. However, any floating-point number type can be used.)

To find out what color-spec is associated with a color name, use the following function.

get-color-spec

Function

```
get-color-spec color-name
```

Returns the color-spec associated with the symbol *color-name*. If there is no color-spec associated with *color-name*, this function returns nil. If *color-name* is the name of a color alias, the color alias is dereferenced until a color-spec is found.

Color-specs are made using standard functions make-rgb, make-hsv and make-gray. For Example:

```
(color:make-rgb 0.0s0 1.0s0 0.0s0)
(color:make-hsv 1.2s0 0.5s0 0.9s0)
(color:make-gray 0.66667s0)
```

The predicate color-spec-p can be used to test for color-spec objects. The function color-spec-model returns the model in which a color-spec object has been defined.

11.4 Color aliases

You can enter a color alias in the color database using the function definecolor-alias. You can remove an entry in the color database using deletecolor-translation.

define-color-alias

This makes an entry in the color database under *name*, which should be a symbol. LispWorks by convention uses keyword symbols. *alias-for* is either a color-spec or another color name (symbol). Attempting to make an alias for an original entry in the color database results in an error. An original entry means one present in a new LispWorks image or one entered via the color database loading functions described below. If the entry is already present and is a color alias, then the value of the *if-exists* argument is used to determine how to continue.

```
(color:define-color-alias :wire-color :darkslategray)
```

delete-color-translation

delete-color-translation color-name

This removes an entry from the color-database. Both original entries and aliases can be removed.

As described in Section 11.3 on page 101, the function get-color-spec returns the color-spec associated with a color alias. The function get-color-alias-translation returns the ultimate color name for an alias.

get-color-alias-translation

Function

Function

Function

get-color-alias-translation color-alias

This returns the ultimate color name associated with *color-alias*.

11.5 Color models

Three color models are defined by default: RGB, HSV and GRAY. RGB and HSV allow specification of any color within conventional color space using three orthogonal coordinate axes, while gray restricts colors to one hue between white and black.

Model	Name	Component: Range
RGB	Red Green Blue	RED (0.0 to 1.0) GREEN (0.0 to 1.0) BLUE (0.0 to 1.0)
HSV	Hue Saturation Value	HUE (0.0 to 5.99999) SATURATION (0.0 to 1.0) VALUE (0.0 to 1.0)
GRAY	Gray	GRAY (0.0 to 1.0)

Table 11.1 Color models defined by default

The Hue value in HSV is mathematically in the open interval [0.0 6.0). All values must be specified in floating point values.

You can convert color-specs between models using the available ensurefunctions. For example:

```
(setf green (make-rgb 0.0 1.0 0.0)
    => #(:RGB 0.0 1.0 0.0))
(eq green (ensure-rgb green)) => T
(ensure-hsv green) => #(:HSV 3.0 0.0 1.0)
(eq green (ensure-hsv green) => NIL
(ensure-rgb (ensure-hsv green)) => #(:RGB 0.0 1.0 0.0)
(eq green (ensure-rgb (ensure-hsv green))) => NIL
```

Of course, information can be lost when converting to GRAY:

There is also ensure-color which takes two color-spec arguments. It converts if necessary the first argument to the same model as the second. For example:

```
(ensure-color (make-gray 0.3) green)
=> #(:RGB 0.3 0.3 0.3)
```

ensure-model-color takes a model as the second argument. For example:

```
(ensure-model-color (make-gray 0.3) :hsv)
=> #(:HSV 0 1.0 0.3)
```

The following function compares two color-spec objects for color equality.

colors=

Function

Function

colors= color1 color2 &optional (tolerance 0.001s0)

colors= returns t if the two colors are equal to the given tolerance.

Conversion to pixel values used by CLX is done by convert-color.

convert-color

convert-color port color &key (errorp t)

This returns the representation of *color* on the given Graphics Port *port*. In CLX, this is the "pixel" value, which corresponds to an index into the default colormap. It is more efficient to use the result of convert-color in place of its argument in drawing function calls, but the penalty is the risk of erroneous colors being displayed should the colormap or the colormap entry be changed.

11.6 Loading the color database

You can load new color definitions into the color database.

read-color-db

Function

read-color-db &optional file

This reads color definitions from the given *file* (a filename or pathname). If no *file* is given, it uses the default color definitions file in the Lisp-Works library. The returned data structure can be passed to load-color-database. The format of the file is:

```
#(:RGB 1.0s0 0.980391s0 0.980391s0) snow
#(:RGB 0.972548s0 0.972548s0 1.0s0) GhostWhite
etc
```

Each line contains a color definition which consists of a color-spec and a name. The names are converted to uppercase and interned in the keyword package. Whitespace in names is preserved.

load-color-database

Function

load-color-database data

This loads the color database with color definitions contained in *data*, which should have been obtained via the function read-color-db. The colors thus defined may not be replaced by color aliases.

To clear the color database use the form:

(setf *color-database* (make-color-db))

Warning: You should do this before starting Common LispWorks (that is, before env:start-environment is called) and be sure to load new color definitions for all the colors used in the environment when you do start it. Those colors are determined from the config\colors.db file.

You can remove a color database entry with delete-color-translation.

11.7 Defining new color models

Before using the definition described here, you should evaluate the form:

(require "color-defmodel")

The macro define-color-models can be used to define new color models for use in the color system.

define-color-models

Macro

define-color-models model-descriptors

This defines *all* the color models. *model-descriptors* is a list, each element being a model-descriptor. A model descriptor has the syntax:

(model-name component-descr*)

A component-descr is a list:

(component-name lowest-value highest-value)

The default color models are defined by the following form:

For example, to define a new color model YMC and keep the existing RGB, HSV and GRAY models:

You must then define some functions to convert YMC color-specs to other color-specs. In this example, those functions are named

make-ymc-from-rgb
make-ymc-from-hsv
make-ymc-from-gray

and

```
make-rgb-from-ymc
make-hsv-from-ymc
make-gray-from-ymc
```

You can make this easier, of course, by defining the functions

```
make-ymc-from-hsv
make-ymc-from-gray
make-hsv-from-ymc
make-gray-from-ymc
```

in terms of make-ymc-from-rgb and make-rgb-from-ymc.

If you never convert between YMC and any other model, you need only define the function make-rgb-from-ymc.

The Color System

Printing from the CAPI—the Hardcopy API

The CAPI hardcopy API is a mechanism for printing a Graphics Port (and hence a CAPI output-pane) to a printer. It is arranged in a hierarchy of concepts: printers, print jobs, pagination and outputting.

Printers correspond to the hardware accessible to the OS. Print jobs control connection to a printer and any printer-specific initialization. Pagination controls the number of pages and which output appears on which page. Outputting is the operation of drawing to a page. This is accomplished using the standard Graphics Ports drawing functions and is not discussed here.

12.1 Printers

You can obtain the current printer, or ask the user to select one, by using current-printer. You can ask the user about configuration by using the functions page-setup-dialog and print-dialog which display the standard Page Setup and Print dialogs.

12.2 Print jobs

A Print job is contained within a use of the macro with-print-job, which handles connection to the printer and sets up a graphics port for drawing to the printer.

12.3 Handling pages—page on demand printing

In *Page on Demand Printing*, the application provides code to print an arbitrary page. The application should be prepared to print pages in any order. This is the preferred means of implementing printing. Page on Demand printing uses the with-document-pages macro, which iterates novel all pages in the document.

12.4 Handling pages—page sequential printing

Page Sequential Printing may be used when it is inconvenient for the application to implement Page on Demand printing. In Page Sequential Printing, the application prints each page of the document in order. Page on Demand printing uses the with-page macro, with each invocation of the with-page macro contributing a new page to the document.

12.5 Printing a page

In either mode of printing, the way in which a page is printed is the same. A suitable transformation must be established between the coordinate system of the output-pane or printer-port object and the physical page being printed. The page is then drawn using normal Graphics Ports operations.

12.5.1 Establishing a page transform

The with-page-transform macro can be used to establish a page transform that maps a rectangular region of the document to the whole page being printed. Any number of invocations of with-page-transform may occur during the printing of a page. For instance, it may be convenient to use a different page transform when printing headers and footers to the page from that used when printing the main body of the page.

A helper function, get-page-area, is provided to simplify the calculation of suitable rectangles for use with with-page-transform. It calculates the width and height of the rectangle in the user's coordinate space that correspond to one printable page, based on the logical resolution of the user's coordinate space in dpi.

For more specific control over the page transform, the printer metrics can be queried using get-printer-metrics and the various printer-metrics accessors.

12.6 Other printing functions

A simple printing API is available via simple-print-port, which prints the contents of an output-pane to a printer.

The Hardcopy API also provides a means of printing plain text to a printer. The functions printer-text, print-file and print-editor-buffer can be used for this. Printing from the CAPI—the Hardcopy API

Index

A

abort-dialog function 73, 74 action-callback keyword 27, 30, 33 apropos-color-alias-names function 100 apropos-color-names function 100 apropos-color-spec-names function 100

B

:background keyword 92 best-height keyword 82 best-width keyword 82 bold keyword 12 button panels orientation 23 prompting with 69-71 button-1 keyword 83 button-enabled accessor 17 button-panel class 22 buttons check 18 push 17 radio 18

С

callback keyword 15 callbacks description of 8 general properties 33 graph panes 29 in interfaces 61–64 used for choices 26–27 using callback functions 11 callback-type keyword 33, 74 call-editor function 62 CAPI basic objects 3 description of 1–3 linking code into 8 loading the ??-6 menu hierarchy 50 changing titles interactively 14 character keyword 41 check button panels 23 check buttons 18 check-button class 18, 22 check-button-panel class 22, 23, 27, 32 children-function keyword 28, 29 choice class 21 choice-class keyword 69 choice-interaction accessor 32 choices 21-34 callbacks available 33 description of 21-34 general properties 31-34 relationship to menus 31 choice-selected-item accessor 33 choice-selected-items accessor 33 choice-selection accessor 28, 32 classes collections 22 creating your own 79-88 CLUE 2 CLX₂ collection class 21 collections description of 21 collector panes 16 collector-pane class 16 colors= function 104 column-layout class 23, 36, 57

column-layout function 37 combo boxes 30 confirm-yes-or-no function 66 contain function 6, 7, 84 convert-color function 104 creating menus 45 creating submenus 46

D

:dash keyword 92 :dashed keyword 92 data keyword 7, 17, 33 data-interface keyword 33 default settings selections 27 default-initargs keyword 55, 57, 82 defclass macro 53, 55, 82 define-color-alias function 102 define-color-models function 106 define-interface macro 53 arguments supplied to 55 defpackage function 6 delete-color-translation function 102 demibold keyword 12 description keyword 36 description of the CAPI 1-3 dialogs creating your own 73-77 description of 65–77 display callback 81 display function 6,7 display panes 14 display-callback keyword 82 display-dialog function 73, 75 displaying text on screen 14 display-message function 8,65 display-pane class 15, 38 draw-circle function 80 drawn-pinboard-object class 84,87 draw-point function 83 drop-down list box 31

E

editor panes 16 editor-pane class 16, 38, 62 subclasses 16 editor-pane-text accessor 62 elements creating your own 79-88 generic properties of 11-12 enabled keyword 17, 51 enabled-function keyword 51 evaluate keyword 72 exit-confirmer function 75 exit-dialog function 73, 74, 75 extend-callback keyword 27, 30, 33 extended selection specifying 32 using on diferent platforms 32 extended-selection keyword 25-26, 32 extension gesture 26

F

files prompting for 71–72 :font keyword 93 font keyword 12 fonts 12 :foreground keyword 92 functions apropos-color-alias-names 100 apropos-color-names 100 apropos-color-spec-names 100 colors= 104 convert-color 104 define-color-alias 102 define-color-models 106 delete-color-translation 102get-all-color-names 101 get-color-alias-translation 102 get-color-spec 101 load-color-database 105 read-color-db 105 sample 11

G

generic properties of elements 11–12 geometry of layouts, specifying 40–42 get-all-color-names function 101 get-color-alias-translation function 102 get-color-spec function 101 graph panes callbacks 29 graphics changing interactively 82–83 creating permanent displays 81 displaying 79–82 graphics ports 89 drawing functions 95 pixmap 95 graph-pane class 28 implementation of 84 grid-layout class 38

Н

hardcopy API 109-111 hierarchy of menus 50 hints 40 horizontal-scroll keyword 12, 37

Ι

initial-value keyword 68 input-model keyword 83 integers prompting for 68 interaction general properties 32 in lists 25 interaction keyword 25, 31, 32, 47, 69 interactive streams 17 interactive-stream class 17 interface class 3, 6, 53 interface keyword 33 interface-data keyword 33 interface-item keyword 33 interfaces defining 53-64 description of 53 layouts, specifying 55 menus, specifying 59–61 panes, specifying 55 title, specifying 55 interface-title accessor 27 italic keyword 12 item keyword 33 item-interface keyword 33 item-pinboard-object class 84 items keyword 24, 45, 49

K

```
keywords
:background 92
:dash 92
:dashed 92
:font 93
:foreground 92
:line-end-style 92
:line-joint-style 92
:mask 92
:mask-x 92
:operation 92
```

:scale-thickness 92
:thickness 92
:transform 91

L

layout-class keyword 23 layouts combining different 39-40 description of 35-42 introduction to 7 specifying geometry 40–42 specifying size of panes in 37 layouts keyword 53 left keyword 15 light keyword 12 :line-end-style keyword 92 :line-joint-style keyword 92 Lisp forms prompting for 72 list function 36 list items, specifying 24 list panels 24 listener panes 17 listener-pane class 17, 38 list-panel class 24 lists actions in 26 deselection in 26 extended selection in 25 extended selections 25 interaction in 25 multiple selection in 25 prompting with 68–71 retraction in 26 single selection in 25 load-color-database function 105 loading the CAPI 6

M

make-container function 73 make-instance function 5 :mask keyword 92 :mask-x keyword 92 max keyword 68 medium keyword 12 menu class 3, 45 menu-bar keyword 53, 59, 60 menu-bar-items keyword 46 menu-component class 46 menu-component class 3 menu-item class 3, 49 menus components 31 creating 45 creating submenus 46 description of 45–51 disabling items in 51 grouping items together 46–49 individual items in 49 menu hierarchy 50 nesting 46 menus keyword 53, 59, 60 min keyword 68 motion keyword 83 multiple-selection keyword 25, 31, 32, 48

Ν

none keyword 33 no-selection keyword 31, 32

0

oblique keyword 12 ok-check keyword 68, 73, 76 :operation keyword 92 option panes 30 option-pane class 30 organizing panes 36 output-pane class 79

P

pane-args keyword 70 panel button layout 23 panels check button 23 list 24 push button 22 radio button 23 panes collector 16 creating your own 79-88 default title position 14 display 14 editor 16 graphs 28 listener 17 option 30 organizing 36 sizing 37 text input 15 title 13

panes keyword 53 pathname keyword 71 pinboard objects 84 creating your own 87-88 pinboard-layout class 38,84 pinboard-object class 84 popup-confirmer function 74,75 portable font descriptions 96 print function 21 print-function keyword 21, 24 prompt-for-file function 62,71 prompt-for-form function 72 prompt-for-integer function 68,75 prompt-for-string function 67,74 prompt-for-symbol function 72 prompt-with-list function 68 push button panels creating 22 push buttons 17 push-button class 7, 17, 22 push-button-panel class 22

R

radio button panels creating 23 radio buttons 18 radio-button class 18 radio-button-panel class 22, 23, 32 read-color-db function 105 reader keyword 57 redisplay-interface function 76 retract-callback keyword 18, 26, 30, 33 reverse-italic keyword 12 reverse-oblique keyword 12 roman keyword 12 row-layout class 23, 36, 57 row-layout function 37

S

selection-callback, keyword 25 selections 25-28 default settings 27 extending 25 general properties 32 specifying multiple 32 setf function 14, 17 single selection specifying 32 single-selection keyword 25, 31, 32, 47 slot 5 slot-value function 5 specifying window titles 13-15 streams interactive 17 strings prompting for 67 subclasses finding 14 subclasses, finding 14 symbols prompting for 72-73

Т

text displaying 15 displaying on screen 14 entering 15 text input panes 15 text keyword 12, 13, 14 text-input-pane class 15 :thickness keyword 92 title keyword 14,55 title panes 13 titled-pane class 13 titled-pane-title accessor 62 title-pane class 13 title-position keyword 14,37 titles changing interactively 14 specifying 13, 13–15 top keyword 14 :transform keyword 91

U

user input 65–77 using callback functions 11

V

value-function keyword 75 values

prompting for 67-73 vertical-scroll keyword 12, 37 visible-max-height keyword 40 visible-max-width keyword 40 visible-min-height keyword 40

W

window titles specifying 13-15

X

x-ratios keyword 37

Y

y-ratios keyword 37