

VisualWorks

Cookbook

Part Number: DS14001002

Copyright © 1995 by ParcPlace-Digitalk, Inc. All rights reserved.

Part Number: DS14001002

Revision 2.0, October 1995 (Software Release 2.5)

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

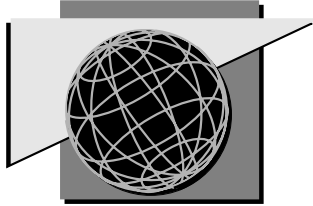
Trademark acknowledgments:

ObjectKit, ObjectWorks, ParcBench, ParcPlace, and VisualWorks are trademarks of ParcPlace Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. DataForms, MethodWorks, ObjectLens, ObjectSupport, ParcPlace Smalltalk, Visual Data Modeler, VisualWorks Advanced Tools, VisualWorks Business Graphics, VisualWorks Database Connect, VisualWorks DLL and C Connect, and VisualWorks ReportWriter are trademarks of ParcPlace Systems, Inc., its subsidiaries, or successors. ENVY is a registered trademark of Object Technology International, Inc. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 1995 by ParcPlace-Digitalk, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from ParcPlace-Digitalk.



Contents

	About This Book	xiii
	Audience	xiii
	Organization	xiii
	Conventions	xiv
	Additional Sources of Information	xviii
	Obtaining Technical Support	xix
Part I	Programming Fundamentals	1
Chapter 1	Smalltalk Basics	3
	Constructing a Message	4
	Combining Messages	7
	Deciding which Type of Variable to Use	10
	Declaring a Variable	13
	Removing a Variable	16
	Creating a Method	18
	Returning from a Method	20
	Creating an Instance of a Class	22
	Initializing an Object	24
	Creating a Class (Subclassing)	26
	Grouping Related Classes	29
	Grouping Related Methods	31
	Creating a Branch	33
	Creating a Loop	34
	Creating Complex True/False Tests	38
Chapter 2	Building Applications	41
	Designing the Application	42
	Painting the User Interface	43
	Creating the Domain Models	45

Connecting the Interface to the Models 47
 Connecting the Widgets to Each Other 50

Part II User Interface 51

Chapter 3 Widget Basics 53
 Accessing a Widget Programmatically 54
 Sizing a Widget 56
 Positioning a Widget 60
 Aligning a Group of Widgets 65
 Spacing a Group of Widgets 66
 Bordering a Widget 67
 Changing a Widget's Font 68
 Hiding a Widget 70
 Disabling a Widget 72
 Changing the Tabbing Order 74
 Coloring a Widget 75
 Adding and Removing Dependencies 78

Chapter 4 Windows 81
 Opening a Window 82
 Getting a Window from a Builder 85
 Sizing a Window 86
 Moving a Window 90
 Changing a Window's Label 92
 Refreshing a Window's Display 93
 Coloring a Window 94
 Adding and Removing Scroll Bars 96
 Adding a Menu Bar 98
 Getting the Active Window 99
 Getting the Window at a Specific Location 100
 Closing a Window 101
 Expanding and Collapsing a Window 103
 Hiding a Window 104
 Making a Window a Slave 105
 Setting a Window's Icon 108

Chapter 5 Labels 109
 Creating a Textual Label 110
 Creating a Graphic Label 111

	Supplying the Label at Run Time	113
	Changing Font, Emphasis, and Color	116
	Building a Registry of Labels	118
Chapter 6	Input Fields	121
	Creating an Input Field	122
	Restricting the Type of Input	125
	Formatting Displayed Data	129
	Validating the Input	132
	Modifying a Field's Pop-Up Menu	139
	Connecting a Field to Another Field	143
	Restricting Entries in a Field (Combo Box)	146
	Moving the Insertion Point	150
Chapter 7	Lines, Boxes, and Ovals	153
	Separating Widgets with a Line	154
	Grouping Widgets with a Box	156
	Grouping Widgets with an Ellipse	158
Chapter 8	Buttons	159
	Adding a Set of Radio Buttons	160
	Adding a Check Box	162
	Adding an Action Button	164
	Giving a Button a Graphic Label	167
	Turning Off Highlighting	168
Chapter 9	Text Editors	171
	Adding a Text Editor	172
	Accessing the Selected Text	174
	Highlighting Text Programmatically	176
	Aligning Text	178
	Making an Editor Read-Only	180
	Modifying an Editor's Menu	182
Chapter 10	Lists	183
	Adding a List	184
	Editing the List of Elements	187
	Allowing for Multiple Selections	189
	Finding Out What Is Selected	191
	Adding a Menu to a List	194
	Changing the Highlighting Style	196

	Connecting Two Lists	198	
	Connecting a List to a Text Editor	200	
Chapter 11	Datasets		203
	Adding a Dataset	204	
	Selecting Columns While Painting	209	
	Adding a Row	210	
	Connecting Data to a Dataset	212	
	Enhancing Column Labels	213	
Chapter 12	Tables		215
	Using TableInterface	216	
	Adding a Table	217	
	Connecting a Table to an Input Field	221	
	Labeling Columns and Rows	223	
Chapter 13	Menus		225
	Creating a Menu	226	
	Creating a Submenu	231	
	Adding a Menu Bar	233	
	Adding a Menu Button	236	
	Adding a Pop-Up Menu	240	
	Modifying a Menu Dynamically	243	
	Disabling a Menu Item	248	
	Adding a Divider to a Menu	250	
	Adding a Shortcut Key	252	
	Displaying an Icon in a Menu	254	
	Changing Menu Colors	257	
	Using a Menu Editor	259	
Chapter 14	Sliders		263
	Adding a Slider	264	
	Connecting a Slider to a Field	267	
	Changing the Range Dynamically	270	
	Changing the Length of the Marker	273	
	Making a Slider Two-Dimensional	274	
Chapter 15	Dialogs		277
	Displaying a Warning	278	
	Asking a Yes/No Question	280	
	Asking a Multiple-Choice Question	282	

	Requesting a Textual Response	284	
	Requesting a Filename	286	
	Choosing from a List of Items	289	
	Linking a Dialog to a Master Window	292	
	Creating a Custom Launcher	294	
	Creating a Custom Dialog	296	
Chapter 16	Subcanvases		301
	Inheriting an Application's Capabilities	302	
	Nesting One Application in Another	305	
	Reusing an Interface Only	308	
	Swapping Interfaces at Run Time	310	
	Accessing an Embedded Widget	313	
Chapter 17	Notebooks		315
	Adding a Notebook	316	
	Determining Which Tab Is Selected	319	
	Changing the Binding's Appearance	322	
	Changing the Size and Axis of the Tabs	324	
	Setting the Starting Page	326	
	Adding Secondary Tabs (Minor Keys)	328	
	Connecting Minor Tabs to Major Tabs	331	
	Changing the Page Layout (Subcanvas)	334	
	Connecting a Notebook to a Text Editor	336	
Chapter 18	Drag and Drop		339
	About Drag and Drop	340	
	Adding a Drop Source	343	
	Adding a Drop Target (General)	348	
	Providing Visual Feedback During a Drag	350	
	Responding to a Drop	359	
	Examining the Drag Context	365	
	Responding to Modifier Keys	366	
	Defining Custom Effect Symbols	371	
Chapter 19	Custom Views		375
	Creating a View Class	376	
	Connecting a View to a Domain Model	378	
	Defining What a View Displays	380	
	Updating a View When Its Model Changes	382	
	Connecting a View to a Controller	385	

	Redisplaying All or Part of a View	387
	Integrating a View into an Interface	389
Chapter 20	Custom Controllers	391
	Choosing an Input Architecture	392
	Creating a Controller Class	395
	Connecting a Controller to a Model	399
	Connecting a Controller to a View	400
	Defining When a Controller Has Control	402
	Defining What a Controller Does	405
	Equipping a Controller with a Menu	409
	Shifting Control to a Different Controller	411
	Sensing Mouse Activity	412
	Sensing Keyboard Activity	416
	Getting the Cursor's Location	419
Part III	Data Structures	423
Chapter 21	Numbers	425
	Creating a Number	426
	Adding and Subtracting	431
	Multiplying and Dividing	432
	Rounding	434
	Getting Squares and Roots	436
	Comparing Two Numbers	438
	Getting the Minimum and Maximum	441
	Performing Trigonometric Functions	442
	Performing Logarithmic Functions	444
	Testing Numberness, Evenness, Zeroness	445
	Accessing and Converting the Sign	447
	Converting a Number to Another Form	449
	Factoring	453
	Generating a Random Number	454
	Accessing Numeric Constants	458
Chapter 22	Dates	461
	Creating a Date	462
	Getting Information about a Day	465
	Getting Information about a Month	467
	Getting Information about a Year	469

	Adding and Subtracting with Dates	471	
	Comparing Dates	473	
	Formatting a Date	475	
Chapter 23	Times		477
	Creating a Time	478	
	Getting the Seconds, Minutes, and Hours	480	
	Adding and Subtracting Times	482	
	Creating a Time Stamp	483	
	Timing a Block of Code	484	
	Changing the Time Zone	486	
Chapter 24	Collections		489
	Choosing the Right Collection	490	
	Creating a Collection	491	
	Getting the Size	495	
	Adding Elements	497	
	Removing Elements	500	
	Replacing Elements	505	
	Copying Elements	508	
	Combining Two Collections	510	
	Finding Elements	511	
	Comparing Collections	517	
	Sorting a Collection	519	
	Converting to a Different Type of Collection	522	
	Looping through the Elements (Iterating)	524	
Chapter 25	Characters and Strings		529
	Creating a Character	530	
	Creating a String	532	
	Distinguishing Types of Characters	534	
	Changing the Case	537	
	Getting a String's Length and Width	539	
	Comparing	540	
	Searching	543	
	Combining Two Strings	545	
	Extracting a Substring	547	
	Removing or Replacing a Substring	549	
	Abbreviating a String	551	
	Inserting Line-End Characters	553	

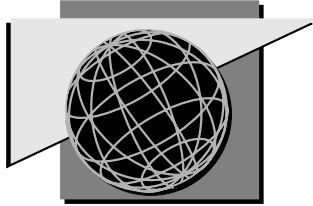
Chapter 26	Text and Fonts	555
	Creating a Text Object	556
	Displaying a Text Object	558
	Setting the Line Length	559
	Disabling Word Wrapping	560
	Controlling Alignment	561
	Setting Indents and Tabs	562
	Counting the Characters	564
	Printing a Text Object	565
	Searching for Strings	566
	Replacing a Range of Text	567
	Comparing Text Objects	568
	Copying a Range of Text	569
	Changing Case	571
	Applying Boldfacing and Other Emphases	572
	Using the Platform's Default Font	575
	Creating a Custom Text Style	576
	Changing Font Size	578
	Setting Font Family or Name	582
	Setting Text Color	585
	Changing the Fonts Menu	587
	Changing the Default Font	588
	Listing Platform Fonts	589
Chapter 27	Text Files	591
	Creating a File or Directory	592
	Getting Information about a File	594
	Getting File or Directory Contents	597
	Storing Text in a File	598
	Opening an Editor on a File	601
	Deleting a File or Directory	602
	Copying or Moving a File	603
	Comparing Two Files or Directories	605
	Printing a File	607
	Scanning Fields in a File (Stream)	609
	Setting File Permissions	611
Chapter 28	Object Files (BOSS)	613
	Storing Objects in a BOSS File	614
	Getting Objects from a BOSS File	617
	Storing and Getting a Class	621

	Converting Data After Changing a Class	624
	Customizing the Storage Representation	626
Chapter 29	Geometrics	629
	Displaying a Point	630
	Displaying a Straight or Jointed Line	631
	Displaying a Curved Line	634
	Displaying a Polygon	637
	Displaying an Arc, Circle, or Ellipse	640
	Changing the Line Thickness	644
	Changing the Line Cap Style	645
	Changing the Line Join Style	647
	Coloring a Geometric	649
	Integrating a Graphic into an Application	652
Chapter 30	Images, Cursors, and Icons	657
	Creating a Graphic Image	658
	Displaying an Image	662
	Coloring Pixels in an Image	664
	Masking Part of an Image	666
	Expanding or Shrinking an Image	668
	Flopping an Image	669
	Rotating an Image	670
	Layering Two Images	672
	Caching an Image	674
	Animating an Image	675
	Creating a Cursor	678
	Changing the Current Cursor	681
	Creating an Icon	682
	Associating an Icon with a Window	683
Chapter 31	Color	685
	Creating a Color	686
	Creating a Coverage	690
	Creating a Tiled Pattern	692
	Applying a Color or Pattern	694
	Changing an Image's Color Palette	696
	Changing the Policy for Rendering Colors	698
Chapter 32	Adapting Domain Models to Widgets	703
	Setting up Simple Value Models (ValueHolder)	704

Contents

Adapting Part of a Domain Model (AspectAdaptor)	706
Synchronizing Updates (Buffering)	710
Adapting a Collection (SelectionInList)	713
Adapting a Collection Element	715
Creating a Custom Adaptor (PluggableAdaptor)	717

Index	719
--------------	------------



About This Book

The *VisualWorks Cookbook* provides step-by-step instructions for performing hundreds of common tasks with VisualWorks®. VisualWorks is a fully object-oriented environment for constructing applications using the ParcPlace Smalltalk™ programming language.

Audience

This *Cookbook* is designed to help both new and experienced developers find and use the rich capabilities of the VisualWorks extensive class library.

This *Cookbook* assumes that you have a beginning familiarity with VisualWorks tools and Smalltalk syntax. It also assumes that you are familiar with the VisualWorks graphical user-interface application architecture. You can obtain that familiarity by using the *VisualWorks Tutorial*. In addition, ParcPlace-Digitalk and some of its partners provide VisualWorks training classes.

Organization

This *Cookbook* is organized around the set of tasks and subtasks that await the application developer—creating interfaces, storing data, and so on. *Cookbook* topics normally contain the following sections:

- **Strategy:** Explains concepts for understanding a task and choosing among alternative ways of performing the same task.

- **Basic Steps:** Describes the simplest way to accomplish the task and then provides example code. In some cases, the example can be executed by itself in a Workspace.
- **Variants:** Describes other ways to perform the same task or ways to perform closely related tasks. The “Variants” section also provides example code.
- **See Also:** Refers to related material, most often another task in the *Cookbook*.

The *Cookbook* uses a set of example classes to demonstrate various techniques. The example classes are contained in a set of files in the `online/examples` subdirectory of the product directory. The Online Documentation tool provides a convenient means of loading (if necessary) and browsing example classes using the File→Browse Example Class command. Filing in the example classes by other means, such as a File List tool, is not recommended because certain files rely on others.

Conventions

This section describes the notational conventions used to identify technical terms, computer-language constructs, mouse buttons, and mouse and keyboard operations.

Typographic Conventions



This book uses the following fonts to designate special terms:

Example	Description
<i>template</i>	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
<code>cover.doc</code>	Indicates filenames, pathnames, commands, and other C++, UNIX, or DOS constructs to be entered outside VisualWorks (for example, at a command line).

Example	Description
<i>filename.xwd</i>	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File→New command	Indicates the name of an item on a menu.
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.
 Caution:	Indicates information that, if ignored, could cause loss of data.
 Warning:	Indicates information that, if ignored, could damage the system.

Screen Conventions

This book contains a number of sample screens that illustrate the results of various tasks. The windows in these sample screens are shown in the default Smalltalk look, rather than the look of any particular platform. Consequently, the windows on your screen will differ slightly from those in the sample screens.

Mouse Buttons

Many hardware configurations supported by VisualWorks have a three-button mouse, but a one-button mouse is the standard for Macintosh users, and a two-button mouse is common for OS/2 and Windows users. To avoid the confusion that would result from referring to <Left>, <Middle>, and <Right> mouse buttons, this book instead employs the logical names <Select>, <Operate>, and <Window>.

The mouse buttons perform the following interactions:

<Select> button	Select (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of <i>operations</i> that are appropriate for the current view or selection. The menu that is displayed is referred to as the <i><Operate> menu</i> .
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks <i>window</i> (except dialogs), such as move and close. The menu that is displayed is referred to as the <i><Window> menu</i> .

Three-Button Mouse

VisualWorks uses the three-button mouse as the default:

- The left button is the <Select> button.
- The middle button is the <Operate> button.
- The right button is the <Window> button.

Two-Button Mouse

On a two-button mouse:

- The left button is the <Select> button.
- The right button is the <Operate> button.
- To access the <Window> menu, you press the <Control> key and the <Operate> button together.

One-Button Mouse

On a one-button mouse:

- The unmodified button is the <Select> button.
- To access the <Operate> menu, you press the <Option> key and the <Select> button together.
- To access the <Window> menu, you press the <Command> key and the <Select> button together.

Mouse Operations

The following table explains the terminology used to describe actions that you perform with mouse buttons.

When you see:	Do this:
click	Press and release the <Select> mouse button.
double-click	Press and release the <Select> mouse button twice without moving the pointer.
<Shift>-click	While holding down the <Shift> key, press and release the <Select> mouse button.
<Control>-click	While holding down the <Control> key, press and release the <Select> mouse button.
<Meta>-click	While holding down the <Meta> or <Alt> key, press and release the <Select> mouse button.

Additional Sources of Information

Printed Documentation

In addition to this *Cookbook*, the core VisualWorks documentation includes the following documents:

- *Installation Guide*: Provides instructions for the installation and testing of VisualWorks on your combination of hardware and operating system.
- *Release Notes*: Describes the new features of the current release of VisualWorks.
- *Tutorial*: Introduces the VisualWorks tools, class library, and approach to application design. It also introduces basic object-oriented concepts and the Smalltalk language.
- *User's Guide*: Provides an overview of object-oriented programming, a description of the Smalltalk language, a VisualWorks tools reference, and a description of various reusable software modules that are available in VisualWorks.
- *International User's Guide*: Describes the VisualWorks facilities that support the creation of nonEnglish and cross-cultural applications.
- *Object Reference*: Provides detailed information about the VisualWorks class library.

The documentation for the VisualWorks database tools consists of the following documents:

- *VisualWorks' Database Tools Tutorial and Cookbook*: Introduces the process and tools for creating applications that access relational databases. The "Cookbook" chapter describes how to programmatically customize various aspects of a database application.
- *Database Connect User's Guide*: Provides information about the external database interface. Versions of it exist for ORACLE7, SYBASE, and DB2 databases.

Online Documentation

To display the online documentation browser, open the Help pull-down menu from the VisualWorks main menu bar and select **Open Online Documentation**. Your choice of online books includes:

- *Database Cookbook*: Online version of the “Cookbook” part of the *VisualWorks’ Database Tools Tutorial and Cookbook* described above.
- *Database Quick Start Guides*: Describes how to build database applications. It covers such topics as data models, single- and multiwindow applications, and reusable data forms.
- *International User’s Guide*: Online version of the *International User’s Guide* described above.
- *VisualWorks Cookbook*: Online version of this book.
- *VisualWorks DLL and C Connect Reference*: Describes C data classes, object engine access functions, and user-primitive functions.

Obtaining Technical Support

If, after reading the documentation, you find that you need additional help, you can contact ParcPlace-Digitalk Technical Support. ParcPlace-Digitalk provides all customers with help on product installation. ParcPlace-Digitalk provides additional technical support to customers who have purchased the ObjectSupport package. VisualWorks distributors often provide similar services.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id*, which indicates the version of the product you are using. Choose **Help→About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id**.

- Any modifications (*patch files*) distributed by ParcPlace-Digitalk that you have imported into the standard image. Choose **Help→About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

How to Contact Technical Support

ParcPlace-Digitalk Technical Support provides assistance by:

- Electronic mail
- Electronic bulletin boards
- World Wide Web
- Telephone and fax

Electronic Mail

To get technical assistance on the VisualWorks line of products, send electronic mail to `support-vw@parcplace.com`.

Electronic Bulletin Boards

Information is available at any time through the electronic bulletin board CompuServe. If you have a CompuServe account, enter the ParcPlace-Digitalk forum by typing `go ppdforum` at the prompt.

World Wide Web

In addition to product and company information, technical support information is available via the World Wide Web:

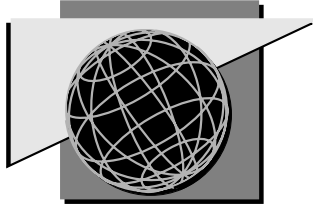
1. In your Web browser, open this location (URL):
`http://www.parcplace.com`
2. Click the link labeled "Tech Support."

Telephone and Fax

Within North America, you can:

- Call ParcPlace-Digitalk Technical Support at 408-773-7474 or 800-727-2555.
- Send questions and information via fax at 408-481-9096.
Operating hours are Monday through Thursday from 6:00 a.m. to 5:00 p.m., and Friday from 6:00 a.m. to 2:00 p.m., Pacific time.

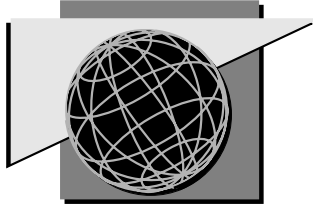
Outside North America, you must contact the local authorized reseller of ParcPlace-Digitalk products to find out the telephone numbers and hours for technical support.



Part I

Programming Fundamentals

Chapter 1: Smalltalk Basics	3
Chapter 2: Building Applications	41



Chapter 1

Smalltalk Basics

This chapter shows how to perform fundamental programming tasks, including:

Constructing a Message	4
Combining Messages	7
Deciding which Type of Variable to Use	10
Declaring a Variable	13
Removing a Variable	16
Creating a Method	18
Returning from a Method	20
Creating an Instance of a Class	22
Initializing an Object	24
Creating a Class (Subclassing)	26
Grouping Related Classes	29
Grouping Related Methods	31
Creating a Branch	33
Creating a Loop	34
Creating Complex True/False Tests	38

Constructing a Message

Strategy

A *message expression* is made up of two parts: a *receiver* and a *message*. The receiver is the object from which you desire a service. The message is the name of the receiver's method that provides the service, along with any necessary arguments.

Basic Steps

- Name the receiver (1.0) and then supply the message (sin).

```
"Print it"  
1.0 sin                                     "Basic Step"
```

Variants

V1. Storing the Result in a Variable

Every time a message is sent, the receiver sends an answer back. The answer itself is an object, perhaps the result of a computation. When this answer object is needed, you can assign it to a variable.

- Send a message to an object and store the result in a variable named sine.

```
"Print it"  
| sine |  
sine := 1.0 sin.  
^sine                                     "V1 Step"
```

V2. Naming a Variable as the Receiver

In variant 1, the receiver is a literal object, specifically a floating-point number. You can also send a message to an object that is stored in a variable, by naming the variable as the receiver.

- Send a message (squared) to the number held by a variable named sine.

```
"Print it"  
| sine |  
sine := 1.0 sin.  
sine squared.                                     "V2 Step"
```

V3. Naming a Class as the Receiver

You can also name a class as the receiver of a message. This is most often done when you are creating an instance of a class, as in the following example.

- Send a message (today) to the Date class.

```
"Print it"  
Date today                                     "V3 Step"
```

V4. Including One or More Arguments (Keyword Message)

When the message requires an argument, the message name ends in a colon. This is called a *keyword message*. For each argument, the message contains a separate keyword ending with a colon. By convention, each keyword and argument are indented on a new line below the receiver, if this improves readability of the code.

- Send a message that requires two arguments. Specifically, when copying a substring, you must specify the starting index and ending index of the desired substring.

```
"Print it"  
'9942-Steering wheel'                             "V4 Step"  
  copyFrom: 1  
  to: 4
```

V5. Using a Special Symbol (Binary Message)

For convenience, common operations such as addition and subtraction are invoked using the special symbols that are widely associated with those operations. These messages are called *binary messages* because you must supply one argument as well as the receiver (as with a one-keyword message).

- Multiply the receiver (12) by the argument (3.14159).

```
"Print it"
```

```
12 * 3.14159.
```

```
"V5 Step"
```

Combining Messages

Strategy

A simple message expression sends one message to a receiver. You can combine simple messages in several ways.

- You can create *complex* message expressions by using simple messages as the receivers or arguments of other messages (first three variants).
- You can rewrite a complex expression as a *sequence* of simpler ones (fourth variant).
- You can send multiple messages to the same receiver by *cascading* them (last variant). Cascaded expressions are generally used sparingly; they can be harder to read and debug than sequences of expressions.

Variants

V1. Using the Result of One Message as the Receiver in a Second Message

- Send a message (squared) to the result (0.841471) that is returned by the first message (1.0 sin).

```
"Print it"  
1.0 sin squared.                                     "V1 Step"
```

V2. Using the Result of One Message as the Argument in a Second Message

- Create a random-number generator (Random new) and then ask it for the next number in the random-number stream (next). The result is a random number between 0 and 1. Use that result as the argument in a multiplication.

```
"Print it"  
52 * Random new next.                               "V2 Step"
```

V3. Controlling Parsing Order

In a complex expression, messages are executed from left to right, starting with no-argument messages, then binary messages, and then keyword messages. You can use parenthesis to specify the parsing sequence. Expressions inside parentheses are executed before those outside. Expressions can be nested. Try executing the following expression both with and without the parentheses.

- Get a random number between 1 and 52 and then convert it from a floating-point number to the next-higher integer.

```
"Print it"  
(52 * Random new next) ceiling                                "V3 Step"
```

V4. Sending a Sequence of Messages

You can rewrite a complex expression as a *sequence* of simpler ones, typically by using one or more *temporary variables* that capture the result of one message for use in another message.

1. Declare a temporary variable for each result to be captured.
2. Create a random number generator and assign it to a temporary variable.
3. Get the next random number in the random number stream and assign it to another temporary variable.
4. Use the random number as the argument in a multiplication.

```
"Print it"  
| generator random |                                         "V4 Step 1"  
generator := Random new.                                    "V4 Step 2"  
random := generator next.                                    "V4 Step 3"  
52 * random.                                                "V4 Step 4"
```

V5. Sending Multiple Messages to the Same Receiver (Cascading)

When a series of messages are sent to the same receiver, you use a semicolon to separate the messages. Then you have to name the receiver only once, at the beginning of the series.

1. Create a collection.
2. Add five elements to the collection, using cascaded messages.

```
"Print it"
| flavors |
flavors := OrderedCollection new.                "V5 Step 1"

flavors                                         "V5 Step 2"
  add: 'Vanilla';
  add: 'Chocolate';
  add: 'Cookie Crumble';
  add: 'Rocky Road';
  add: 'Raspberry Swirl'.

^flavors
```

Deciding which Type of Variable to Use

Strategy

There are six types of variables:

- Temporary variables
- Instance variables
- Class instance variables
- Class variables
- Pool dictionaries
- Global variables

Scope: Each type of variable has a different scope—that is, it is available to a different range of methods. The list above is sorted from narrowest to widest scope.

In general, use the narrowest scope that suits your purpose.

Use Temporary Variables Freely

A temporary variable has the narrowest scope (a single method or `Workspace do it`). Use temporary variables freely.

Avoid Use of Global Variables

A global variable has the broadest scope—it can be referenced anywhere. This makes it hazardous to use, mainly because class names are also global in scope. The danger is that you may accidentally erase a class that happens to have the same name as your global, by associating a new value with the name. For this reason, you must be very careful when naming a global variable. A carefully named global can be useful in casual `Workspace` code, when you want to hold onto the result of one `do it` for use in a later `do it`.

Use an Accessing Method to Boost the Scope

Frequently, you can use an accessing method to give a variable wider scope. This is especially useful when you need to create a systemwide default that is accessed by a variety of objects. For example, the `LookPreferences` class implements a `defaultBorder`

method. Clients can ask for this default border by asking the `LookAndFeel` class for it, instead of relying on a global variable.

Use Instance Variables to Hold Object Data and Persistent Parameters

An instance variable is the primary means of associating data with an object, and it can be used freely. It is directly available to any instance method in the defining class and in any instance method of a subclass (that is, it is inherited).

A secondary role for the instance variable is as a persistent parameter. That is, if you are passing the same object as an argument to several methods within the same receiver, it may be helpful to create an instance variable as a central holder for that object.

Use Class Variables to Hold Defaults and Static Resources

A class variable is available to both class methods and instance methods, in the defining class and any subclasses. Because its value can be changed by multiple objects, a class variable is used mainly to hold a nonchanging or rarely changing value. For example, the `Date` class holds a collection of `MonthNames` as a class variable and makes that collection available to several of its methods.

Compared with instance variable: An alternative is to create an instance variable and initialize its value each time you create a new instance. The first advantage of a class variable is that you have to initialize it only once. The second advantage is that you need to have only one copy of the data, even when many instances of the class are accessing it.

Use Class Instance Variables within a Class Hierarchy

A class instance variable is rarely used. It is declared once, in a parent class. Each subclass then has its own copy of the variable and can assign to it independently.

Compared with class variable: One alternative is to declare a separate class variable in each subclass. Since each subclass would have to name its variable differently, each subclass

would need its own versions of the methods that accessed the variable. Thus, the advantage of the class instance variable is that all subclasses can use the same name for the variable and still be able to assign to it independently.

Compared with instance variable: Another alternative is to declare the variable as a regular instance variable in the parent class. This requires that you initialize the variable each time a new instance is created. Thus, the class instance variable is usually reserved for nonchanging resources whose initialization is too costly to repeat.

Use Pool Dictionaries to Create a Shared Lexicon among Classes

A pool dictionary is a lookup table shared by a related set of classes. The dictionary itself must first be declared as a global variable and initialized as a Dictionary. Each entry in the dictionary is then available directly to all class and instance methods in any class that declares the dictionary as its pool.

For example, the classes that manipulate text objects share a dictionary named `TextConstants`. This dictionary associates names such as “Space” and “Tab” with their character equivalents. As a result, the text classes can use the names for keyboard keys rather than the more obscure character codes.

Because the dictionary must be declared first as a global variable, pool dictionaries should be used very sparingly. Another negative for pool dictionaries is that, like globals, they are not automatically recreated when you file in the code that depends on them.

Compared with class variable: One alternative is to store a lookup dictionary in a class variable. The first disadvantage of this approach is that only instances of that class can access the dictionary directly. The second disadvantage is that lookups must be performed explicitly. With a pool dictionary, by contrast, naming the key is sufficient to summon its associated value. For example, instead of `TextConstants at: Space` you can simply use `Space`. Neither of these disadvantages is critical in most situations.

Declaring a Variable

Strategy

Data type: Any object can be assigned to any type of variable. In Smalltalk, variables are not declared as having a particular data type.

Default value: The value of any variable is nil until you assign a new value to it.

Naming: The name of a variable describes its purpose and sometimes also its intended data type. By convention, variable names are quite descriptive and rarely abbreviated except in casual usage. When multiple words are combined to form a name, each embedded initial is capitalized. Variable names may contain letters, numbers, and underscores, and may not begin with a number. By convention, the first letter is lowercase for local variables and uppercase for nonlocal variables.

Separating multiple declarations: When you are declaring two or more variables at the same time, use a space to separate them.

Undeclared variables: When a variable is referenced without being declared, it is entered in a system dictionary named Undeclared. If it is later declared, the entry in Undeclared remains and should be removed before you deploy your application. To do so, open an inspector on the dictionary by highlighting the word Undeclared and using the inspect command. You can use the dictionary inspector to check for references to each entry and to remove each entry that has no entries.

Variants

V1. Declaring a Temporary Variable

A temporary variable must be declared at the beginning of the method or Workspace do it in which it is used. To do so, place its name between vertical bars.

Naming. A temporary variable's name should begin with a lowercase letter, indicating its local scope.

Automated declarations: In practice, many Smalltalk programmers postpone declaring temporaries. They freely insert new variable names and rely on the system to prompt them when it encounters each undeclared variable name. They can then indicate its scope as “temporary” and the system will create the declaration.

- Declare temporary variables by enclosing them within vertical bars.

```
| numberOfDays date |                                "V1 Step"  
numberOfDays := 7.  
date := Date today addDays: numberOfDays.  
Transcript show: date printString.
```

V2. Declaring an Instance Variable

Naming. An instance variable’s name should begin with a lowercase letter.

1. In a System Browser, select the class.
2. Choose the **definition** command in the class view to display the class definition.
3. Add the desired instance variable name to the list of instance variables and then accept the new definition.

V3. Declaring a Class Instance Variable

Naming. A class instance variable’s should must begin with an uppercase letter.

1. In a System Browser, select the class and make sure the class switch is on.
2. In the pop-up menu provided by the class view, select the **definition** command to display the metaclass definition.
3. Add the desired variable name to the list of class instance variables and then accept the new definition.

V4. Declaring a Class Variable

Naming. A class variable’s name should begin with an uppercase letter.

1. In a System Browser, select the class.
2. Choose the **definition** command in the class view to display the class definition.
3. Add the desired class variable name to the list of class variables and then accept the new definition.

V5. Declaring a Pool Dictionary

When a group of related constants is to be made available to a class, a pool dictionary is an alternative to creating a separate class variable for each constant. Multiple classes can declare and use the same pool dictionary. For example, the text-related classes such as `Text` store constants such as the tab character in a pool dictionary, so they don't have to instantiate that character in their text-handling methods.

Naming: A pool dictionary's name should begin with an uppercase letter. The key in each element of the dictionary must also begin with an uppercase letter.

Creating the dictionary: The dictionary itself is a global variable and must be declared and initialized before you can declare it as a pool dictionary.

1. In a Workspace, verify that the global name you intend to give a new pool dictionary is not already in use as a global variable name by sending an `includesKey:` message to `Smalltalk`. The argument is the global name, expressed as a symbol (prefixed by a number sign).
2. In a Workspace, create a new dictionary by sending a new message to the `Dictionary` class. Add the desired constants and their lookup keys to the dictionary (now or later).
3. Create a global variable to hold the dictionary by sending an `at:put:` message to `Smalltalk`. The first argument is the global name, expressed as a symbol. The second argument is the dictionary.
4. For each class that will use the pool dictionary, display the class definition in a System Browser and add the global to the list of pool variables. Note that pool dictionaries are not inherited, so you must add them to each class that is to use them, even if they are declared in its superclass.

Removing a Variable

Strategy

Before you remove a variable, find and delete all references to that variable. For most types of variables, it's easier to find references before you remove the variable.

Variants

V1. Removing a Temporary Variable and Its References

Since a temporary variable can be referenced only in a single method or Workspace **do** it, you need to scan only that method for references. For a long method, use the **find** command to find each occurrence of that variable in the code. Rewrite the code as needed to remove each reference.

After you have removed all references, delete the variable declaration.

V2. Removing an Instance Variable and Its References

1. In a System Browser, select the class in which the variable is declared.
2. Select the **inst var refs** command in the class view.
3. In the resulting menu of instance variables, select the variable that you intend to remove.
4. In the resulting browser of all methods that reference the variable, edit the methods to remove the references.
5. In the class definition, delete the variable name and then accept the definition.

V3. Removing a Class Variable and Its References

Do steps 1 through 5 as above, except in step 2 use the **class var refs** command.

V4. Removing a Class Instance Variable and Its References

Do steps 1 through 5 as above, except in step 1 also turn on the System Browser's class switch.

V5. Removing a Pool Dictionary

1. For each entry in the pool dictionary, open a browser on all references to that pool variable by sending a `browseAllCallsOn:` message to the `Browser` class. The argument is the dictionary's entry, which is accessed by sending an `associationAt:` message to the global name of the dictionary; the argument is the lookup key for the dictionary entry.

```
Browser browseAllCallsOn: (TextConstants associationAt: #Centered). "V3 Step 1"
```

2. In each browser, edit each method, removing all references to the pool constants.
3. Use a System Browser to change the class definition of each class that declares the pool dictionary, removing the global dictionary from the definition.
4. Open a browser on all references to the global variable that holds the pool dictionary by sending a `browseAllCallsOn:` message to the `Browser` class. The argument is the Smalltalk dictionary's entry for the global, which is accessed by sending an `associationAt:` message to `Smalltalk`; the argument is the name of the global dictionary.

```
Browser browseAllCallsOn: (Smalltalk associationAt: #TextConstants). "V3 Step 4"
```

5. In the resulting browser, edit each method, removing all references to the global variable.
6. Remove the global variable from the global dictionary named `Smalltalk` by sending a `removeKey:` message to `Smalltalk`. The argument is the name of the global dictionary, expressed as a symbol.

V6. Removing a Global Variable

Do Steps 4 through 6 above.

Creating a Method

Strategy

The System Browser provides a template to help you create a new method. You can also use an existing method as your starting point.

Instance vs. class methods: An instance method is available to any instance of the defining class, whereas a class method is available only to the class itself. For that reason, instance methods outnumber class methods. Class methods are most often used for creating an instance of the class and for initializing and accessing class variables.

When to subdivide a large method: To promote reusability, keep Smalltalk methods short. For example, you can usually break a long method into smaller methods to isolate individual services that other clients may want to use. Similarly, when a subset of the code is repeated in a large method with only minor variations, you can usually make that subset into a separate method.

Naming. Method names may contain letters, numbers, and underscores, but may not begin with a number. The first letter should be lowercase.

Variants

V1. Creating an Instance Method

1. In a System Browser, turn on the instance switch.
2. Select the class.
3. Select the message category or add a new one.
4. Fill in the method template and then use the accept command in the code view.

V2. Creating a Class Method

In a System Browser, turn on the class switch and then do steps 2 through 4 above.

V3. Fixing Common Errors at Compile Time

Undeclared temporary variables: This is an “error” that you can commit on purpose, because the system will prompt you with a menu of variable types with which you can quickly and easily declare each of the temporary variables.

Undeclared class and instance variables: When you are prompted to declare an instance or class variable, it’s best to select **abort** in the menu and declare the variables before continuing. To save your uncompiled method while you use the System Browser to redefine the class, select **spawn** in the code view. This opens a new browser on the uncompiled code.

Missing period: When you have omitted a period, the system treats what should be two statements as though they were a single message expression. As a result, the error description is usually “Nothing more expected.”

Missing delimiters: When you have omitted a parenthesis or bracket, the error description is “Right parenthesis expected” or “Period or right bracket expected.”

Returning from a Method

Strategy

Every method returns a single object, which can be a collection of other objects.

By default, a method returns the object that received the message. This return object is simply ignored by clients that are interested in the effect of the method and not the return value.

When the return object is significant, you can specify that object by using a caret symbol (^).

Returning from a block—When a return character is enclosed within a block, it forces a return from the entire method. That is, it does not act as a return from the block back to the containing method.

Basic Steps

Online example: Customer1Example

- In a method, place the name of the return object after a caret.

```
accountID                                     "Basic Step"  
^accountID
```

Variants

V1. Returning the Result of a Message

A return character that is followed by a message causes the result of that message to be returned. This approach often circumvents the need to create a temporary variable for the message result.

- Place a caret in front of the message receiver.

displayString	"V1 Step"
^accountID printString, '--', name	

V2. Returning a Conditional Value

Frequently, a method performs a test and returns one value if the test result is true and a second value if the test result is false. Relying on the fact that a return character that is followed by a message returns the result of the message, you can use a single return caret to serve both forks of the branch, rather than placing a caret inside each block.

This approach has the advantage of combining two exit points into a single exit point, which is better programming style. It also makes the `ifTrue:` and `ifFalse:` blocks *clean* blocks—that is, blocks that do not contain a hard return character.

- Place a caret in front of the conditional expression. (The example is a hypothetical method that could be added to `Customer1Example`.)

accountPrefix	"V2 Step"
"Answer the first four characters of the accountID, or an empty string if the accountID is empty."	
id id := self accountID.	
^id isEmpty ifTrue: [String new] ifFalse: [id copyFrom: 1 to: 4].	

Creating an Instance of a Class

Strategy

Every class provides one or more messages for creating an instance of itself. By convention, these messages can be found in the instance creation protocol of the class.

The new method: All classes inherit a basic `new` method from the `Object` class. This method creates a raw instance whose instance variables each have the value `nil`.

Abstract classes and new: Abstract classes, such as `Boolean`, typically provide their own version of `new`, in which they announce an error such as “This class is not intended to be instantiated.”

Other flavors of new: Other classes frequently override `new` in order to initialize instance variables.

Basic Steps

- Send a new message to the class.

"Inspect"	
SourceFileManager new.	"Basic Step"

Variants

V1. Using a Class-Specific Creation Message

Other creation messages are specific to the implementing class. They frequently take arguments that are used to initialize the instance variables of the new instance. Such *parameterized* creation messages are typically a convenience for client objects, because the same effect usually can be achieved by first creating a `new` instance and then sending the parameters via *accessing* messages.

- Send a message that is listed in the class's *instance creation* protocol.

```
"Inspect"
  Date newDay:10                                "V1 Step"
  month:#June
  year:1995
```

V2. Accessing a Distinguished Instance

When a class is intended to provide just one instance of itself, that instance is referred to as a *distinguished instance*. Typically, it is stored in a class variable and accessed using an *accessing* message named `default`.

- Send a default message or other accessing message to the class. (Use the `inspect` command to open an Inspector on the instance, so you can see how the instance variables differ from those of a new instance.)

```
"Inspect"
SourceFileManager default.                       "V2 Step"
```

See Also

- “Initializing an Object” on page 24

Initializing an Object

Strategy

When you want a new instance to provide default values other than nil, create an initialize method in a protocol named initialize-release. The main advantage in doing so is that you prevent the errors that result when client methods send messages to the uninitialized instance variables.

Classes other than ApplicationModel and its subclasses must take the added step of invoking the initialize method in the instance-creation methods. (ApplicationModel already does so, because initialization is routinely used by its subclasses.)

Basic Steps

Online example: Customer1Example

1. Create an instance method named initialize in an *initialize-release* protocol. The method is responsible for assigning values to some or all of the instance variables.

```
initialize                                     "Basic Step 1"

    accountID := 0.
    name := String new.
    address := String new.
    phoneNumber := String new.
```

2. Create a class method named new in an *instance creation* protocol. The method is responsible for creating a new instance and then sending initialize to it.

```
new                                           "Basic Step 2"
    ^super new initialize
```

Variants

Including a Parent Class's Initialization

Online example: PreferredCustomerExample

When implementing an `initialize` method, be aware that a parent class may also have an `initialize` method. If so, invoke the parent class's `initialize` as a first step in the subclass's `initialize`.

- In the subclass's `initialize` method, send `initialize` to super, usually as the first step in the method.

<code>initialize</code>	"Variant Step"
<pre> super initialize. yearsOfPatronage := 3. </pre>	

Creating a Class (Subclassing)

Strategy

Every new class is a child of an existing class, so creating a new class consists of sending a subclassing message to the parent class. The System Browser provides a template for the most common subclassing message.

Choosing a parent class: Use the *Object Reference* to find existing classes that relate to the new class's behavior. Choose as a parent the class that you would need to modify the least in order to convert it to your purposes. Typically, this will be the *Object* class or a class in one of your own application-specific hierarchies.

Naming: A class name can contain letters, numbers and underscores, but cannot begin with a number. Because class names are Smalltalk global variables, they should begin with a capital letter.

Basic Steps

Online example: Customer1Example

1. In a System Browser, select the class category, and make sure no class is selected.
2. Modify the resulting class-creation template, entering at least the name of the parent class and the name of the new subclass.

```
Object subclass: #Customer1Example                                "Basic Step 2"  
  instanceVariableNames: 'accountID name address phoneNumber '  
  classVariableNames: "  
  poolDictionaries: "  
  category: 'Examples-Cookbook'
```

3. Select the accept command in the code view.

Variants

V1. Creating a Subclass of ApplicationModel or SimpleDialog

For the convenience of interface programmers, the `install` command in a Canvas provides a convenient dialog box for creating new subclasses of `ApplicationModel` and `SimpleDialog`. Choose `SimpleDialog` as the parent when instances of the new class will be used primarily to run one or more dialog windows. Choose `ApplicationModel` as the parent when instances of the new class will be used to run regular windows as well.

V2. Creating a Collection Class that Holds Pointers to Its Elements

- To create a class that holds a collection of indexable variables, each of which is a pointer to an object, use the following variant of the standard subclassing message.

```
ArrayedCollection variableSubclass: #ExampleArray          "V2 Step"
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Examples'
```

V3. Creating a Collection Class that Holds Byte-Sized Elements

- To create a variable-byte class, which holds a collection of indexable variables, each of which is a byte-sized object, use the following variant of the standard subclassing message. A variable-byte class can have no instance variables and can have only a variable-byte subclass.

```
ArrayedCollection variableByteSubclass: #ExampleByteArray    "V3 Step"
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Examples'
```

See Also

- “Declaring a Variable” on page 13
- “Grouping Related Classes” on page 29

Grouping Related Classes

Strategy

It is frequently useful to treat a group of classes as a single entity, known as a *class category*. The main advantage is that you can file out all of the classes in the category at once, for backing up your work or sharing it with another user. It also makes browsing the related code easier.

Keeping an application together: A common usage of class categories is to group all of the classes used by an application or by a module within a larger application.

Keeping support classes separate: Because a class cannot belong to multiple categories, support classes used in multiple applications are usually grouped in separate support categories. This allows you to easily create a set of files containing just the code needed for any given application.

Basic Steps

B1. Adding a Class Category

1. In a System Browser, select the **add** command in the class-category view.
2. In the resulting dialog, supply the name of the category (no harm is done if it already exists). Type a blank space to cancel the operation.

The new class category is inserted in the list above the category that was previously selected. To insert it at the bottom of the list, make sure no category is selected when you begin step 1.

B2. Removing a Class Category

If you remove a class category that still has classes in it, the classes will be removed also.

1. In a System Browser, select the category.
2. Select the **remove** command in the class-category view.
3. If the category contains classes, you will be asked to confirm the removal.

B3. Renaming a Class Category

When you rename a class category, the new name appears automatically in the definition of each class in that category.

1. In a System Browser, select the category.
2. Select the rename as command in the class-category view.
3. In the resulting dialog, supply the new name. Type a blank space to cancel the operation.

B4. Moving a Class to a Different Category

1. In a System Browser, select the class.
2. Select the move to command in the class view.
3. In the resulting dialog, supply the name of the destination category. If the category does not exist, it will be created.

B5. Changing the Order of Class Categories

1. In a System Browser, select the edit all command in the class-category view.
2. The categories and their members will be listed in the code view. Carefully cut and paste the listing to achieve the desired ordering.
3. Select the accept command in the code view. (To cancel the operation, select the cancel command in the code view.)

Grouping Related Methods

Strategy

Placing related methods in a message category, also known as a *protocol*, helps to document your code and makes it easier to find with a System Browser. Your choice of protocol name has no effect on your code's operation.

Public vs. private protocols: By convention, methods that are intended for use only by other methods of the current class are placed in a protocol named `private`. Some programmers use a broader definition of *private*, choosing to include any method that has a restricted set of intended clients. Some programmers also create multiple `private` protocols, each having a second part to its name that describes its contents (such as `private-accessing`).

Standard protocols: Because most methods fit into certain categories, a set of standard protocol names has come into use. Appendix A of the *VisualWorks User's Guide* lists these protocols.

Basic Steps

B1. Adding a Protocol

1. In a System Browser, select the class.
2. Select the add command in the protocol view.
3. In the resulting dialog, supply the name of the protocol (no harm is done if it already exists). Type a blank space to cancel the operation.

The new protocol is inserted in the list above the protocol that was previously selected. To insert it at the bottom of the list, make sure no protocol is selected when you begin step 1.

B2. Removing a Protocol

If you remove a protocol that still has methods in it, the methods will be removed also.

1. In a System Browser, select the protocol.
2. Select the remove command in the protocol view.

3. If the protocol contains methods, you will be asked to confirm the removal.

B3. Renaming a Protocol

1. In a System Browser, select the protocol.
2. Select the rename as command in the protocol view.
3. In the resulting dialog, supply the new name. Type a blank space to cancel the operation.

B4. Moving a Method to a Different Protocol

1. In a System Browser, select the method.
2. Select the move to command in the method view.
3. In the resulting dialog, supply the name of the destination protocol. If the protocol does not exist, it will be created. Type a blank space to cancel the operation.

B5. Copying a Method to a Different Class

1. In a System Browser, select the method.
2. Select the move to command in the method view.
3. In the resulting dialog, enter the name of the destination class, a greater-than symbol (>), and the name of the destination protocol. To copy the method to the class side rather than the instance side, insert "class" after the class name.

B6. Changing the Order of Protocols

1. In a System Browser, select the edit all command in the protocol view.
2. The protocols and their members will be listed in the code view. Carefully cut and paste the listing to achieve the desired ordering.
3. Select the accept command in the code view. (To cancel the operation, select the cancel command in the code view.)

Creating a Branch

Strategy

Branching, or conditional processing, is accomplished by sending a variant of the `ifTrue:` message to the result of a true/false test. The conditional statements are enclosed in a block.

Basic Steps

1. Get the width of the screen.
2. Test whether the screen's width is less than 1280 pixels.
3. If true, ring the bell.

screenWidth	
screenWidth := Screen default bounds width.	"Basic Step 1"
screenWidth < 1280	"Basic Step 2"
ifTrue: [Screen default ringBell]	"Basic Step 3"

Variants

The full set of variants is:

```
ifTrue:
ifFalse:
ifTrue: ifFalse:
ifFalse: ifTrue:
```

Creating a Loop

Strategy

Several ways of looping are provided in Smalltalk. They fall into the following categories:

- Simple repetition
- Conditional looping
- Processing each element in a collection (iteration)

In each case, a block is used to contain the statements that are repeated.

Use simple repetition when the block is to be repeated a certain number of times. Use conditional looping when the block is to be repeated only while a test condition is met. Use collection iteration when the block is to be repeated for each element in a collection.

Variants

V1. Looping a Fixed Number of Times (`timesRepeat:`)

- Send a message to the Transcript 10 times.

```
10 timesRepeat: [Transcript show: 'Testing!'; cr.]
```

"V1 Step"

V2. Looping with an Index Argument (`to:do:`)

- Repeat a block using each number in the interval from 65 to 122. This block includes a *block argument* (`:asciiNbr`), which is specified by an identifier preceded by a colon and separated from the block's expressions by a vertical bar. In each loop, a successive number in the interval is passed into the block and used where the block argument appears.

```
65 to: 122 do: [ :asciiNbr |  
  Transcript show: asciiNbr asCharacter printString]
```

"V2Step"

V3. Looping with an Index and Steps (to:by:do:)

- Repeat a block using each number in the interval from 10 to 65, counting by 5s.

```
10 to: 65 by: 5 do: [:marker |
  Transcript
    show: marker printString;
    show: '---' ].
```

"V3 Step"

V4. Looping until the Block Exits (repeat)

1. For each repetition of the block, increase a counter.
2. Test whether the counter is greater than 10. If so, exit from the loop.

```
| counter |
counter := 0.

[counter := counter + 1.
 counter > 10 ifTrue: [^true]
] repeat.
```

"V4 Step 1"
"V4 Step 2"

V5. Looping while a Condition is True or False (whileTrue: and whileFalse:)

1. Create an instance of Time that is 3 seconds from now.
2. Before each repetition of the block, test whether the endTime has been reached.
3. For each repetition, show the current time in the transcript.

```
| endTime |
endTime := Time now addTime: (Time fromSeconds: 3).

[Time now <= endTime] whileTrue: [
  Transcript show: Time now printString; cr].
```

"V5 Step 1"
"V5 Step 2"
"V5 Step 3"

V6. Processing Each Element of a Collection (do:)

1. Get an array containing the standard color names.
2. Print each color name in the Transcript.

```
| colors |  
colors := ColorValue constantNames.                                "V6 Step 1"  
  
colors do: [ :colorName |  
    Transcript show: colorName printString; cr]                    "V6 Step 2"
```

V7. Detecting the First Element that Meets a Test (detect:)

1. Get the color names.
2. Detect the first color that begins with the letter *m*.
3. Show that color name in the Transcript.

```
| colors mColor |  
colors := ColorValue constantNames.                                "V7 Step 1"  
  
mColor := colors detect: [ :colorName |  
    colorName first = $m].                                         "V7 Step 2"  
  
Transcript show: mColor printString; cr.                            "V7 Step 3"
```

V8. Selecting Elements that Meet a Test (select:)

1. Get the color names.
2. Get the subcollection of names beginning with the letter *d*.
3. Show each element of the subcollection in the Transcript.

```
| colors dColors |  
colors := ColorValue constantNames.                                "V8 Step 1"  
  
dColors := colors select: [ :colorName |  
    colorName first = $d].                                         "V8 Step 2"  
  
dColors do: [ :dColor |  
    Transcript show: dColor printString; cr].                       "V8 Step 3"
```

V9. Selecting Elements that Fail a Test (reject:)

1. Get the color names.

2. Get the subcollection of names that do not begin with *d*.
3. Show each element of the subcollection in the Transcript.

```
| colors nonDColors |
colors := ColorValue constantNames.                                "V9 Step 1"

nonDColors := colors reject: [ :colorName |
    colorName first = $d].                                         "V9 Step 2"

nonDColors do: [ :nonDColor |
    Transcript show: nonDColor printString; cr].                    "V9 Step 3"
```

V10. Operating on Each Element and Collecting the Results (collect:)

1. Get the color names.
2. For each color name, create a string equivalent and capitalize its initial.
3. Show each element of the resulting collection in the Transcript.

```
| colors colorsAsStrings string |
colors := ColorValue constantNames.                                "V10 Step 1"

colorsAsStrings := colors collect: [ :colorName |
    string := colorName asString.
    string at: 1 put: (string first asUppercase).
    string].                                                         "V10 Step 2"

colorsAsStrings do: [ :color |
    Transcript show: color; cr].                                     "V10 Step 3"
```

Creating Complex True/False Tests

Strategy

When two or more conditions need to be tested, use the logical *and* and *or* messages to combine the tests in a series. These messages come in two forms:

- `&` and `|` (vertical bar, not the letter *L*)
- `and:` and `or:` (the argument is a block containing the second test)

Use the second pair of messages when the second test depends on the first test. In a common situation involving such a dependency, the first test checks the data type of a variable and the second test sends a message that is appropriate only for the desired data type.

Using the second form, involving block arguments, is also appropriate when the second test is costly, because the second test is executed only when needed.

Variants

V1. Answering True Only When Both Tests are Met (Logical And)

1. Ask the user for a password.
2. Test the length of the response and respond appropriately.

```
| response message |  
response := Dialog request: 'What is your password'.           "V1 Step 1"
```

```
(response size > 0) & (response size <= 8)                   "V1 Step 2"  
  ifTrue: [message := 'Thank you. Have a safe journey']  
  ifFalse: [message := 'Sorry, I cannot let you pass'].
```

```
Transcript show: message; cr.
```

V2. Ignoring the Second Test, When Possible

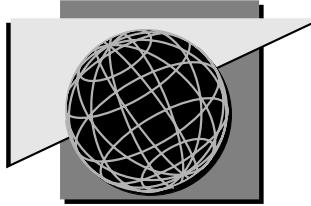
1. Ask for a password.

2. Test whether the response has four or more letters. If it does, test whether the fourth character is a percent sign.

```
| response message |  
response := Dialog request: 'What is your password'. "V2 Step 1"
```

```
((response size >= 4) and: [(response at: 4) = $%]) "V2 Step 2"  
  ifTrue: [message := 'Thank you. Have a safe journey']  
  ifFalse: [message := 'Sorry, I cannot let you pass'].
```

```
Transcript show: message; cr.
```



Chapter 2

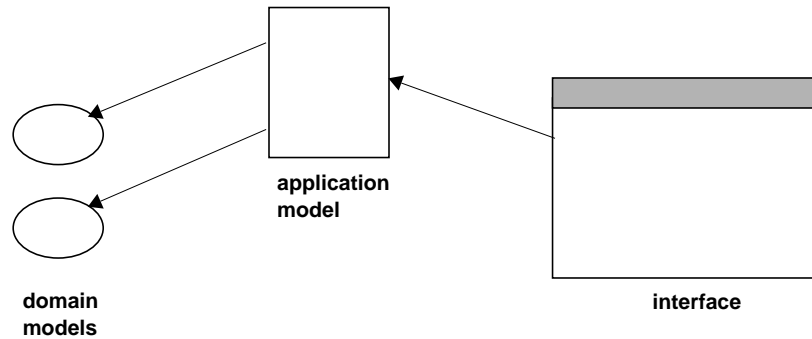
Building Applications

This chapter provides an overview of the major steps involved in building an application. You can use this chapter as a checklist as you create your first applications.

In keeping with its role as a checklist, this chapter does not go into detail about any given step. Other sections in the *Cookbook* supply the detail that is missing here, and those sections are referred to in the “See Also” notes. For in-depth explanations of the various application-building steps outlined here, see the *VisualWorks Tutorial*.

Designing the Application	42
Painting the User Interface	43
Creating the Domain Models	45
Connecting the Interface to the Models	47
Connecting the Widgets to Each Other	50

Designing the Application



Strategy

For simple applications, you can often “design” by painting the user interface. Even fairly complex applications that are heavy on interface and light on processing can be created this way.

For complicated applications, involving a complex information model and many windows, a formal design phase is usually helpful. Various methodologies have been proposed for analyzing and designing object-oriented applications.

ParcPlace-Digitalk offers training and consulting for a methodology called Object Behavior Analysis and Design (OBA/D). You can use this methodology to guide you through the process of:

- Creating an object-oriented requirements specification, based on the behaviors inherent in the system
- Creating an architectural design
- Defining reusable subsystems
- Creating a detailed design
- Choosing data-structure classes and supporting objects
- Evaluating trade-offs with respect to performance and understandability

Painting the User Interface



Strategy

Creating the user interface helps you understand the high-level data and processing requirements of your application. Using the ability of VisualWorks to define placeholder methods for the interface widgets (described later) you can even use the interface to demonstrate your concept to users and get valuable early feedback.

Basic Steps

1. For each window in your interface, open a blank canvas or an existing canvas that you want to extend.
2. For each desired widget, select it in the Palette and click to locate it on the desired canvas.
3. Click the **Install** button on the Canvas Tool to *install* each canvas in an *application model* (a new or existing subclass of `ApplicationModel`).
4. Click the **Open** button on the Canvas Tool to see the interface in action.

About the Application Model

Step 4 creates a bare-bones *application model*. This is the portion of your application that knows how to turn an installed canvas into an operational interface. It does this invoking an *interface builder*, which in turn invokes various windows and widgets, according to the interface specification provided by the installed canvas.

An application model is where you define the application-specific behavior of the widgets in the interface. That is, you

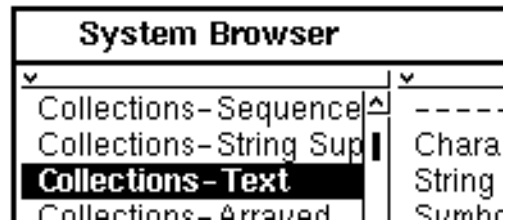
program an application model to establish the connection between each widget and the data or action it represents (in some cases, implementing the action, as well). You can also program an application model to set up interactions among multiple widgets in the interface. VisualWorks provides a number of tools that accelerate this level of programming, as you will see in later sections.

An application model typically binds widgets to data and actions that are defined in one or more *domain models*.

See Also

- “Creating the Domain Models” on page 45
- “Connecting the Interface to the Models” on page 47

Creating the Domain Models



Strategy

A *domain model* is an object that represents an entity in the application's domain. In a simple application, the entire application domain can often be represented by a single model. For example, a hypothetical class named `RolodexCard` could be the entire domain model for a small address-lookup application.

As the application domain becomes more complex, you will find that multiple domain models are necessary, each representing an entity that interacts with other models in the application. In a banking application, for example, the domain would be divided among model classes such as `Bank`, `Customer`, `FederalReserve`, and `MonetaryUnit`.

Role of the domain model: A domain model is intended to remain free of user-interface code. Any instance variables or methods that are necessary purely to support the mechanics of the user interface belong in the application model. This separation of responsibilities makes it easier to reuse your domain models with other interfaces.

Basic Steps

1. In a System Browser, define a domain model class (typically, a subclass of `Object`).
2. Create an `initialize` method to set default values for the instance variables.
3. Create accessing methods for accessing the instance variables.
4. Create actions methods defining the services that clients can request.

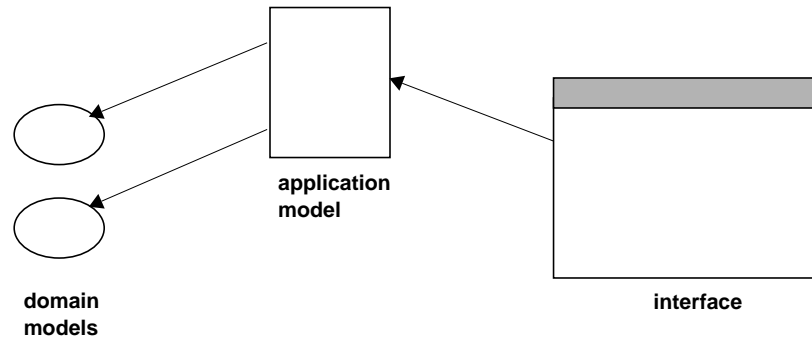
5. Create private methods, if necessary, to provide supporting mechanisms for the actions methods.

Variant

Combining Domain and Application Models

For simple applications in which the domain model is unlikely to be reused with a different interface, it is simpler to merge the responsibilities of the domain model and the application model in a single class. This is the approach taken in some of the sample applications, such as List1Example. For a merged model, define it as a subclass of ApplicationModel rather than of Object.

Connecting the Interface to the Models



Strategy

After you have created a user interface and appropriate domain models, you program the application model to establish the connections between them. The interface must be able to obtain data from the domain models and ask these models to perform actions. By programming the application model to establish these connections, you keep the domain models free of interface concerns.

A typical application model has an action method for each widget that will invoke an action (for example, a button). The application model may implement this action itself or forward a request to the appropriate domain model.

A typical application model has an instance variable (and accessor method) corresponding to each widget that will present an item of data (such as an input field). The application model initializes each such variable with a *value model*—an auxiliary object whose job is to manage the widget's access to the relevant data. In the running application, the widget will ask its value model for the data to be displayed and will send input data to the value model for storage. The widget will also depend on its value model to notify it when the relevant data changes; in response, the widget will update its display. The application model initializes the value model with the appropriate data, which is typically some *aspect* of the domain model.

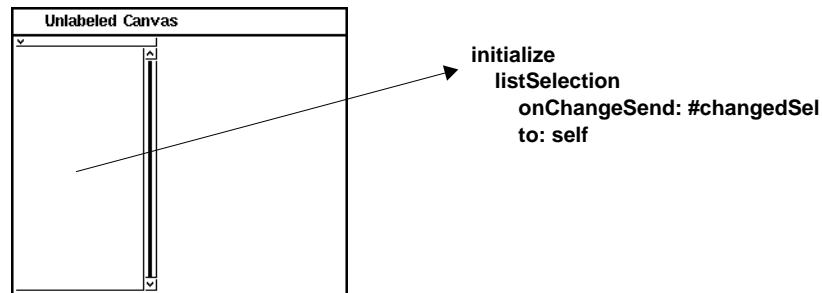
Basic Steps

1. Open a Properties Tool for the canvas.
2. For each data widget, select the widget in the canvas and fill in its **Aspect** property with the name of the method that will return a value model for the widget. **Apply** the property settings to the canvas.
3. For each action button, select the button in the canvas and fill in its **Action** property with the name of the method that will implement the button's action. **Apply** the properties.
4. For each widget that is to supply a menu of actions, select the widget and fill in its **Menu** property with the name of a method that will supply the menu. **Apply** the properties.
5. Install the canvas in the application model when all properties are applied.
6. In the application model in which the canvas is installed, use the canvas's **define** command to create an instance variable for each data aspect that is named by a widget. Alternatively, you can use a System Browser.
7. Use the canvas's **define** command or a System Browser to create **aspects** methods in the application model. Each aspect method returns the value of the corresponding aspect variable.
8. Use a System Browser to create an **initialize** method in the application model. This method creates and assigns a value model to each aspect variable, initializing each value model with the appropriate data from a domain model. (Alternatively, you can use the aspect methods to initialize their respective aspect variables.)
You can choose from among several kinds of value models, depending on your application's needs (see the chapter listed under "See Also").
9. Use the canvas's **define** command or a System Browser to create **actions** methods in the application model. Each action method either implements an action itself or requests an action from the appropriate domain model.
10. Use the Menu Editor to create and install each menu that was specified in a widget's properties.

See Also

- “Adapting Domain Models to Widgets” on page 703

Connecting the Widgets to Each Other



Strategy

Interface widgets frequently interact, so that when the user changes the data in one widget, it triggers a change in another widget. For example, when the user selects a customer name in a list, various field widgets might be updated to display details about the newly selected customer.

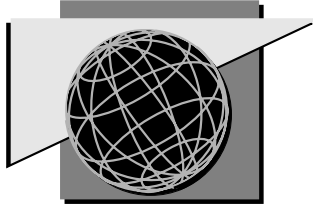
Arranging for such interactions is known as *defining dependencies*. VisualWorks provides a sophisticated dependency mechanism that makes it easy to accomplish this.

Basic Steps

1. In a System Browser, create or edit the `initialize` method of the application model.
2. For each data aspect whose change is intended to trigger a secondary effect, *register an interest* in the corresponding value model. Registering an interest tells the value model what message to send, and to which object, when its value is changed.
3. In a protocol named change messages, create a method for each message that was named in step 2. These methods implement the desired side effects that you want to associate with changes in the data.

See Also

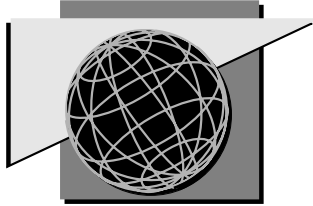
- “Adding and Removing Dependencies” on page 78



Part II

User Interface

Chapter 3: Widget Basics	53
Chapter 4: Windows	81
Chapter 5: Labels	109
Chapter 6: Input Fields	121
Chapter 7: Lines, Boxes, and Ovals	153
Chapter 8: Buttons	159
Chapter 9: Text Editors	171
Chapter 10: Lists	183
Chapter 11: Datasets	203
Chapter 12: Tables	215
Chapter 13: Menus	225
Chapter 14: Sliders	263
Chapter 15: Dialogs	277
Chapter 16: Subcanvases	301
Chapter 17: Notebooks	315
Chapter 18: Drag and Drop	339
Chapter 19: Custom Views	375
Chapter 20: Custom Controllers	391

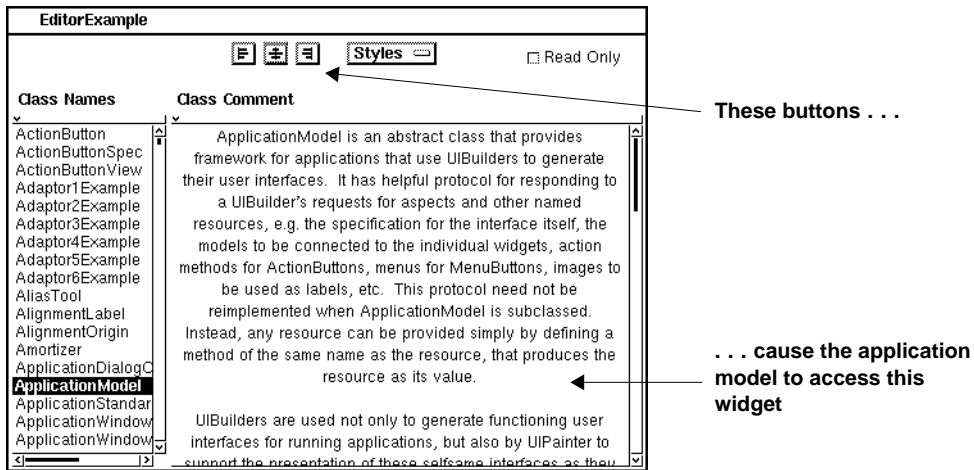


Chapter 3

Widget Basics

Accessing a Widget Programmatically	54
Sizing a Widget	56
Positioning a Widget	60
Aligning a Group of Widgets	65
Spacing a Group of Widgets	66
Bordering a Widget	67
Changing a Widget's Font	68
Hiding a Widget	70
Disabling a Widget	72
Changing the Tabbing Order	74
Coloring a Widget	75
Adding and Removing Dependencies	78

Accessing a Widget Programmatically



Strategy

In a variety of situations, it is useful to program an application model to send messages to a widget while your application is running. For example, you can program the application model to send messages to a text editor to change the alignment of the displayed text, as shown in the basic steps.

In some cases, the application model must send messages to the *wrapper* that surrounds the widget. A wrapper is an instance of *WidgetWrapper*, which controls various aspects of the widget's appearance, such as visibility, enablement, and layout. The variant shows how to access a widget's wrapper to enable, disable, hide, and redisplay the widget.

Basic Steps

Online example: Editor1Example

1. In a canvas, select the widget to be accessed. In the widget's ID property, enter an identifying name for the widget (in this case, #comment). Apply the properties and install the canvas.
2. In a System Browser, edit a method in the application model (in this example, alignCenter) so that it sends a

componentAt: message to the application model's builder. The argument is the ID.

3. Send a widget message to the object returned by step 2.

alignCenter

```
| widget style |
widget := (self builder componentAt: #comment) widget. "Basic Steps 2, 3"
style := widget textStyle copy.
style alignment: 2.
widget textStyle: style.
widget invalidate.
```

Variants

V1. Accessing the Widget's Wrapper

Online example: HideExample

1. In a canvas, select the widget to be accessed. In the widget's ID property, enter an identifying name for the widget. Apply the properties and install the canvas.
2. In a System Browser, edit a method in the application model (in this example, changedListVisibility) so that it sends a componentAt: message to the application model's builder. The argument is the ID.

changedListVisibility

```
| wrapper desiredState |
wrapper := self builder componentAt: #colorList. "V1 Step 2"
desiredState := self listVisibility value.

desiredState == #hidden
    ifTrue: [wrapper beInvisible].

desiredState == #disabled
    ifTrue: [
        wrapper beVisible.
        wrapper disable].
desiredState == #normal
    ifTrue: [wrapper enable; beVisible].
```

Sizing a Widget

Size1 Example

Fixed origin Fixed size	Fixed origin Relative size	Rel. origin Fixed size	Relative origin Relative size
<div style="border: 1px solid black; padding: 2px; width: 50px; height: 20px; display: flex; align-items: center; justify-content: center;">123456</div>	<div style="border: 1px solid black; padding: 2px; width: 150px; height: 20px; display: flex; align-items: center; justify-content: space-between;"> 123456789 123456789 1 </div>	<div style="border: 1px solid black; padding: 2px; width: 50px; height: 20px; display: flex; align-items: center; justify-content: center;">123456</div>	<div style="border: 1px solid black; padding: 2px; width: 150px; height: 20px; display: flex; align-items: center; justify-content: space-between;"> 123456789 123456789 123456789 </div>

Position Tool settings (proportion, offset)

Left	0 50	0 150	0.5 50
Top	0 50	0 50	0 50
Right	0 100	0.5 0	0.5 100
Bottom	0 100	1 -90	1 -90

← These lists . . .

. . . are sized and positioned by these settings in a Position Tool

Strategy

The basic way to set a widget's size is by dragging the widget's selection handles when you paint it on the canvas. You can also use the Canvas Tool's **Arrange→Equalize...** command to make a series of widgets adopt the same width, height, or both.

Widgets appear in their painted size when the window is opened. When the window size is fixed, nothing more normally needs to be done. However, when the window's size is variable, you may want to arrange for the widget to adjust its size in relation to that of the window. You can use the **Layout→Relative** command on the Canvas Tool to arrange for automatic resizing in both the vertical and the horizontal dimensions.

For more complicated situations, or for more precise control, you can set properties on the **Position** page of the Properties Tool. The first two variants show how to set these properties to make a widget's size fixed or relative to the size of its containing window.

The third variant shows how to convert an unbounded widget to a bounded widget so you can control its size. The final variant shows how to change the size of a widget programmatically.

Variants

V1. Making a Widget's Size Fixed

A widget's origin is controlled by the Left (L) and Top (T) property settings; its size is controlled by the Right (R) and Bottom (B) property settings. A fixed size is commonly used for buttons and labels.

Online example: [Size1Example](#)

1. In a canvas, select the widget whose size is to be fixed.
2. In a Properties Tool (Position page), set the Right Proportion to be equal to the Left Proportion (in this example, 0 and 0). Since proportions control variability, identical left and right proportions keep the right edge of the widget a fixed distance from the left edge.
3. Set the Right Offset to the width of the widget added to the Left Offset.
4. Set the Bottom Proportion equal to the Top Proportion.
5. Set the Bottom Offset to the height of the widget added to the Top Offset.
6. Apply the properties and install the canvas.

V2. Making a Widget's Size Relative

You can cause a widget to expand or shrink in concert with the window by setting its Right Proportion to be different from the Left Proportion, or by setting the Bottom Proportion to be different from the Top Proportion. This is especially useful for widgets that can use additional space, such as text editors, lists, and tables. Input fields are often made relative in the horizontal dimension only.

Online example: [Size1Example](#)

1. In a canvas, select the widget whose size is to be relative.
2. In a Properties Tool (Position page), set the Right Proportion to a value that is larger than the Left Proportion. (A right proportion of 0.5 keeps the right edge anchored at the window's midline while the left edge is anchored to the window's left edge.)

3. Set the Right Offset to the distance you want between the widget's right edge and the imaginary line identified by the Right Proportion.
4. Set the Bottom Proportion to a value that is larger than the Top Proportion.
5. Set the Bottom Offset to the distance between the widget's bottom edge and the imaginary line representing the Bottom Proportion.
6. Apply the properties and install the canvas.

V3. Applying Explicit Boundaries to an Unbounded Widget

Four widgets are inherently variable in size: labels, action buttons, radio buttons, and check boxes. These widgets change in size to accommodate their textual labels, which expand and shrink on different platforms because of font differences. Unlike most widgets, which have four boundaries, the variable-size widgets are said to be *unbounded*.

Sometimes it is preferable to convert an unbounded widget so it is bounded like other widgets. As shown in `Size2Example`, the advantage is that you can make a series of buttons have equal dimensions, for example. There is a slight hazard in converting an unbounded widget, however: on a different platform, a font change in the widget's label may cause the label to expand beyond the widget's unyielding boundaries.

Online example: `Size2Example`

1. In a canvas, select an unbounded widget such as a label.
2. In the Canvas Tool, select `Layout→Be Bounded`. Alternatively, select the Bounded button in the Properties Tool (Position page). The icon on the Bounded button shows a rectangle with solid lines on all four sides.
3. Apply the properties, if necessary, and install the canvas.
4. To reverse the operation, select `Layout→Unbounded`, or select the Unbounded button in the Properties Tool. Apply the properties, if necessary, and install the canvas.

V4. Changing a Widget's Size Programmatically

In some circumstances, your application may need to resize a widget while the application is running. In `Size3Example`, a colored region is resized in response to **Expand** and **Shrink** buttons.

Online example: `Size3Example`

1. Get the widget's wrapper from the application model's builder.
2. Send a `bounds` message to the wrapper to get the widget's existing size.
3. Create a rectangle having the desired origin and extent, using the widget's bounding rectangle to derive the new values.
4. Send a `newBounds:` message to the wrapper. The argument is the new bounding rectangle.

`expandBox`

```

| wrapper oldSize newSize |
wrapper := self builder componentAt: #box.           "V4 Step 1"
oldSize := wrapper bounds.                          "V4 Step 2"

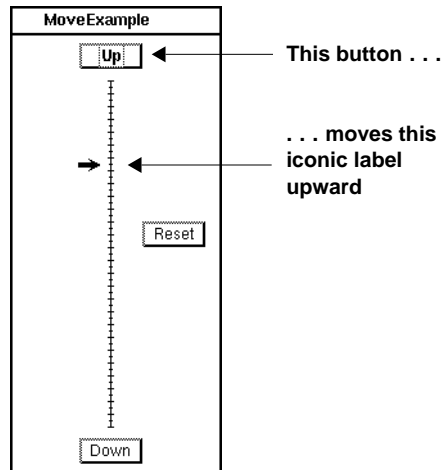
    "If the box is bigger than the window already, do nothing."
oldSize origin x < 0
    ifTrue: [^nil].

    "Expand the bounding rectangle by 10 pixels on each side."
newSize := Rectangle                                 "V4 Step 3"
    origin: oldSize origin - 10
    corner: oldSize corner + 10.

    "Assign the new bounding rectangle to the widget wrapper."
wrapper newBounds: newSize.                          "V4 Step 4"

```

Positioning a Widget



Strategy

The basic way to set a widget's position is by dragging it to the desired position in the canvas. This determines the widget's initial position relative to the window's upper left corner.

Widgets appear in their painted position when the window is opened. When the window size is fixed, nothing more normally needs to be done. However, when the window's size is variable, you may want to arrange for the widget to adjust its position relative to the size of the window. You can use the **Layout→Relative** command on the Canvas Tool to arrange for automatic repositioning in both the vertical and the horizontal dimensions.

For more complicated situations, or for more precise control, you can set properties on the **Position** page of the Properties Tool. The first two variants show how to set these properties to make a widget's position fixed or relative to the size of its containing window.

The final variant shows how to change the position of a widget programmatically.

Variants

V1. Making a Widget's Origin Fixed

Making a widget fixed is useful when the window's size is fixed. When the window's size is variable, this approach works best for a button or other fixed-size widget that is located along the left or top edges of the window.

Online example: [Size1Example](#) (start it and then resize the window to see the effect)

1. In a canvas, select the widget whose position is to be fixed.
2. In a Properties Tool (Position page), set the Left and Top Proportions to 0. These proportions control whether a widget moves relative to the window size. Setting these properties to 0 causes the widget's origin to remain fixed in place.
3. Set the Left Offset to the desired distance between the window's left edge and the widget's left edge (in the example, 50 pixels).
4. Set the Top Offset to the desired distance between the window's top edge and the widget's top edge (50).
5. Apply the properties and install the canvas.

V2. Making a Widget's Origin Relative

A relative origin causes the widget to move farther away from the left and top edges of the window when the window grows and closer when the window shrinks. This is useful for keeping an object centered in the window and for shifting one widget that is placed below or to the right of another widget that expands and shrinks in size.

You can also make the origin relative in only one dimension. In the example, the origin shifts horizontally as the window is resized, but it maintains a stable offset from the window's top edge.

Online example: [Size1Example](#)

1. In a canvas, select the widget whose position is to be relative.

2. In a Properties Tool (Position page), set the Left Proportion to the fraction of the window's width from which the Left Offset is to be measured. (In the example, a left proportion of 0.5 causes the widget to remain anchored at the window's midline.)
3. Set the Left Offset to the distance you want between the widget's left edge and the imaginary line identified by the Left Proportion (50 pixels).
4. Set the Top Proportion to the fraction of the window's height from which the Top Offset is to be measured. (In the example, a top proportion of 0 anchors the widget's top edge at the top edge of the window, which is the same as keeping the origin fixed in the vertical dimension.)
5. Set the Top Offset to the distance you want between the widget's top edge and the imaginary line identified by the Top Proportion (50 pixels).
6. Apply the properties and install the canvas.

V3. Giving an Unbounded Widget a Fixed Position

An unbounded widget has no left, right, top, and bottom sides because its boundaries are not fixed. However, it does have a reference point that can be positioned in either a fixed or relative location in the window. By default, the reference point is the origin of the widget (the top-left corner).

1. In a canvas, select an unbounded widget such as a label.
2. In the Properties Tool (Position page), set all of the Proportions to 0.
3. Set the X and Y Offsets to the coordinates of the widget's top-left corner relative to the top-left corner of the window.
4. Apply the properties and install the canvas.

V4. Giving an Unbounded Widget a Relative Position

Online example: Size2Example

1. In a canvas, select an unbounded widget (in the example, select one of the unbounded buttons on the left to examine its properties).

2. In a Properties Tool (Position page), set the x Proportion to the fraction of the widget's width at which the reference point is to be positioned (0.5).
3. Set the y Proportion to the fraction of the widget's height at which the reference point is to be positioned (0).
4. Set the X Proportion to the fraction of the window's width at which the widget's reference point is to be anchored. (In the example, an X Proportion of 0.25 keeps the widget's reference point anchored one-fourth of the way across the window.)
5. Set the X Offset to the distance you want between the widget's reference point and the imaginary line identified by the X proportion (0).
6. Set the Y Offset to the fraction of the window's height at which the widget's reference point is to be anchored. (In the example, a Y proportion of 0 keeps the widget's reference point a fixed distance from the window's top edge.)
7. Apply the properties and install the canvas.

V5. Positioning a Widget Programmatically

Although it is unusual for an application to need explicit control over a widget's location, it is possible to do so. In `MoveExample`, a graphic label is repositioned by three buttons, giving the effect of a pointer on a meter. The `Up` and `Down` buttons shift the position relative to the prior position, while the `Reset` button moves the widget to an absolute position.

Online example: `MoveExample`

1. Get the widget's wrapper from the application model's builder.
2. For a relative shift in position, send a `moveBy:` message to the wrapper. The argument is a `Point` whose `x` and `y` values indicate the number of pixels by which the widget is to be shifted.

```

moveArrowUp
| wrapper |
wrapper := (self builder componentAt: #arrow).           "V5 Step 1"

```

"If the arrow is not too high, raise it another notch."

```
wrapper bounds origin y > 30  
  ifTrue: [wrapper moveBy: 0@-5]
```

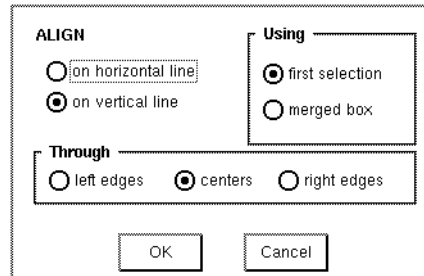
"V5 Step 2"

3. To apply an absolute position, send a `moveTo:` message to the wrapper. The argument is a `Point` whose coordinates are the desired position of the widget.
-

```
resetArrow  
| wrapper |  
wrapper := (self builder componentAt: #arrow).  
wrapper moveTo: self arrowOrigin
```

"V5 Step 3"

Aligning a Group of Widgets



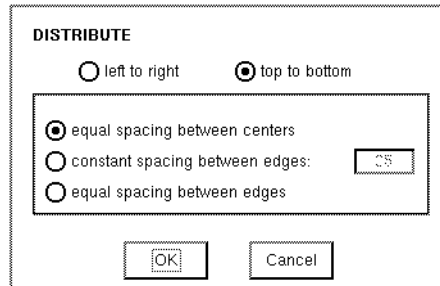
Strategy

When painting a canvas, you frequently need to make a group of widgets align along an imaginary vertical or horizontal line. Dragging the widgets is sufficient sometimes, but for precise control you can use the Align dialog, shown above. You can also use the Properties Tool (Position page), but it is less convenient unless you need to manipulate the position settings for other reasons.

Basic Steps

1. In a canvas, select the widgets to be aligned.
2. In the Canvas Tool, select the Arrange→Align command.
3. In the Align dialog, select on horizontal line when aligning side-by-side widgets. When aligning widgets in a column, select on vertical line.
4. In the Align dialog, select first selection when the widgets are to be aligned with the first widget that was selected. Select merged box to align the widgets on a line halfway between the two most extreme positions within the group of widgets.
5. In the Align dialog, select the edges, or the centers, to be aligned.
6. Install the canvas.

Spacing a Group of Widgets



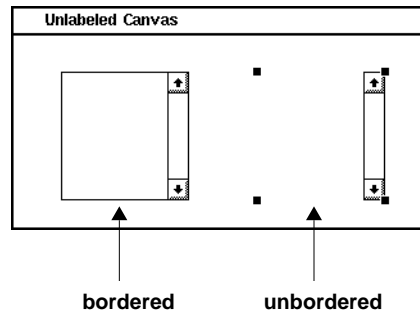
Strategy

When painting a canvas, you frequently need to make the spaces between a group of widgets equal. Dragging the widgets is sufficient sometimes, but for precise control you can use the Distribute dialog, shown above. You can also use the Position Tool (Position page), but it is less convenient unless you need to manipulate the position settings for other reasons.

Basic Steps

1. In a canvas, select the widgets to be spaced.
2. In the Canvas Tool, select the **Align→Distribute** command.
3. In the Distribute dialog, select **left to right** for widgets that are to be spaced in a horizontal row. Select **top to bottom** for columnar distribution.
4. In the Distribute dialog, select the type of spacing. For **constant spacing between edges**, you must specify the number of pixels to place between each pair of widgets.
5. Install the canvas.

Bordering a Widget



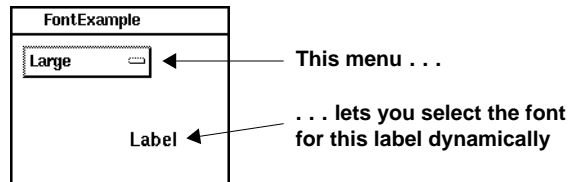
Strategy

Most widgets have a border by default. The appearance of the border changes according to the selected UILook.

Basic Steps

1. Select a widget in a canvas.
2. To apply a border to the widget, turn on its **Border** property.
3. To remove the border, turn off the **Border** property.
4. Apply the properties and install the canvas.

Changing a Widget's Font



Strategy

When a widget's default font is not suitable, you can use the Font menu in the widget's properties to choose an alternative font. The built-in fonts are:

- Default
- Fixed, for a fixed-width font that is useful when you want to align text in columns
- Large, for a font that is slightly larger than the default
- Small, for a font that is slightly smaller than the default
- System, for a font that matches the current platform's system font, when available

You can add or remove fonts in the menu, as referenced in "See Also."

You can also change a widget's font programmatically, as shown in the basic steps.

Basic Steps

Online example: Font1Example

1. In a method in the application model, get the widget from the application model's builder.
2. Create an instance of `TextAttributes` corresponding to the new font. If the font exists in the fonts menu, you can send a `styleNamed:` message to the `TextAttributes` class. The argument is the name of the font (for example, `#large` for the system's Large font).
3. Get the label from the widget by sending a `label` message; get the text of the label by sending a `text` message to it. Then

install a blank text temporarily as a means of erasing the old label if the new font is smaller.

4. Install the new font in the widget by sending a `textStyle:` message to the widget. The argument is the `TextAttributes` you created in step 2.
5. Reinstall the original label by sending a `labelString:` message to the widget.

changedFont

```
| widget newStyle oldLabel |
widget := (self builder componentAt: #label) widget.           "Basic Step 1"
newStyle := TextAttributes styleNamed: (self labelFont value). "Basic Step 2"
```

```
"Erase the existing label in case its font is larger than the new one."
oldLabel := widget label text.                                "Basic Step 3"
widget labelString: "".
```

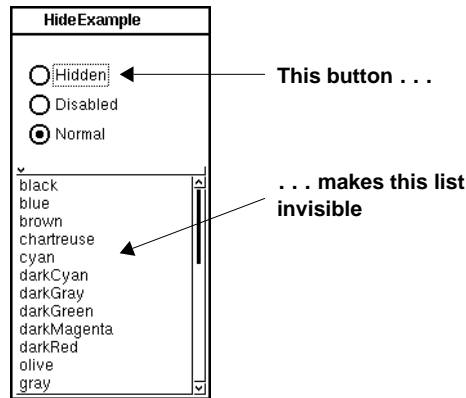
```
"Install the new font."
widget textStyle: newStyle.                                  "Basic Step 4"
```

```
"Reinstall the original label."
widget labelString: oldLabel.                                "Basic Step 5"
```

See Also

- “Creating a Custom Text Style” on page 576
- “Changing the Fonts Menu” on page 587

Hiding a Widget



Strategy

Sometimes a widget is useful only under certain conditions and needs to be hidden at other times to avoid confusing the user of your application. Action buttons need to be hidden when their actions are not appropriate.

A widget may also be hidden when two alternative widgets are layered on top of each other. For example, the Online Documentation window uses a text editor on top of a list editor and hides the view that is unneeded at any given time.

You can turn on a widget's `Initially Invisible` property to cause the widget to be hidden when the window opens. You can also program the application model to hide and show the widget while the application is running (shown in the basic steps).

Basic Steps

Online example: HideExample

1. In a method in the application model, get the widget's wrapper from the application model's builder.
2. To hide the widget, send a `beInvisible` message to the wrapper.
3. To make the widget visible again, send a `beVisible` message to the wrapper.

```
changedListVisibility
| wrapper desiredState |
wrapper := self builder componentAt: #colorList.           "Basic Step 1"
desiredState := self listVisibility value.

desiredState == #hidden
  ifTrue: [wrapper beInvisible].                             "Basic Step 2"

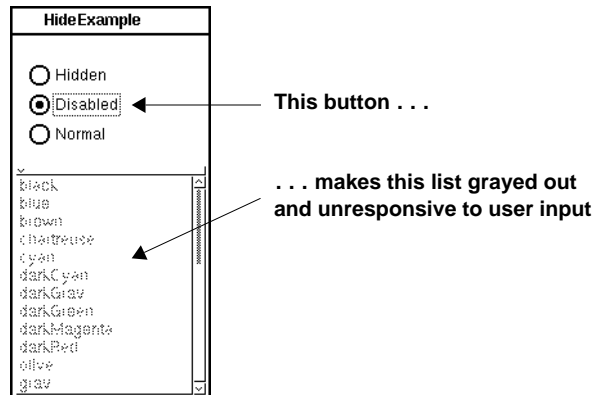
desiredState == #disabled
  ifTrue: [
    wrapper beVisible.
    wrapper disable].                                       "Basic Step 3"

desiredState == #normal
  ifTrue: [
    wrapper enable.
    wrapper beVisible].
```

See Also

- “Disabling a Widget” on page 72

Disabling a Widget



Strategy

Sometimes a widget is useful only under certain conditions, but making it invisible would be confusing to the user of your application. You can disable a widget, causing it to be displayed in gray. In addition, its controller is inactivated so the widget does not respond to user input. Action buttons are frequently “grayed out” when not needed.

You can turn on a widget’s `Initially Disabled` property to cause the widget to be disabled when the window opens. You can also program the application model to disable and enable the widget while the application is running (shown in the basic steps).

Basic Steps

Online example: HideExample

1. In a method in the application model, get the widget’s wrapper from the application model’s builder.
2. To disable the widget, send a `disable` message to the wrapper.
3. To make the widget active again, send an `enable` message to the wrapper.

```

changedListVisibility
| wrapper desiredState |
wrapper := self builder componentAt: #colorList.           "Basic Step 1"

```

```
desiredState := self listVisibility value.
```

```
desiredState == #hidden  
  ifTrue: [wrapper beInvisible].
```

```
desiredState == #disabled  
  ifTrue: [  
    wrapper beVisible.  
    wrapper disable].
```

"Basic Step 2"

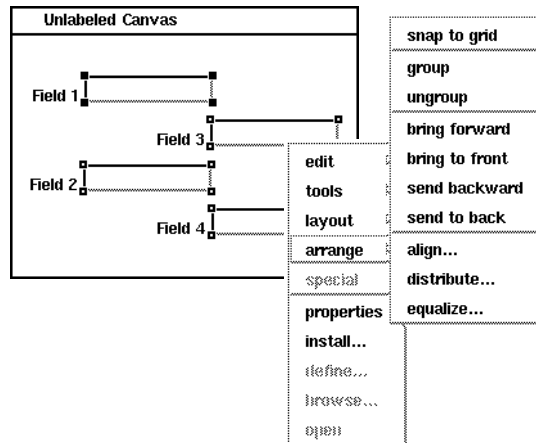
```
desiredState == #normal  
  ifTrue: [  
    wrapper enable.  
    wrapper beVisible].
```

"Basic Step 3"

See Also

- "Hiding a Widget" on page 70

Changing the Tabbing Order



Strategy

When an application is running, users can use the <Tab> key to shift the keyboard focus from one widget to the next in a window, without having to move the mouse.

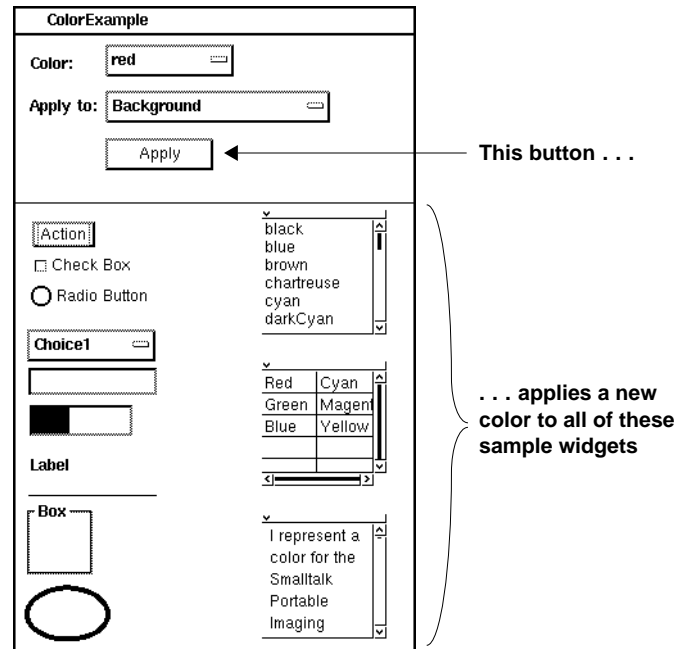
More specifically, the <Tab> key moves focus to each widget on the *tab chain*. You add a widget to the tab chain by turning on its Can Tab property. Passive widgets such as labels and dividers do not have a Can Tab property, so they cannot be put on the tab chain. Note that you should turn off the Can Tab property in a text editor, if you want the editor to interpret the <Tab> key as a literal character to be entered into the text.

Basic Steps

By default, the order in which the <Tab> key advances the focus is the order in which the widgets were drawn. The following steps show how to change the order of widgets in the tab chain.

1. Hold down a <Shift> key while you select the tabbing widgets in the desired order.
2. In the Canvas Tool, select the Arrange→Bring To Front command. Install the canvas.

Coloring a Widget



Strategy

A widget can have up to four color zones:

- Foreground
- Background
- Selection foreground
- Selection background

The Properties Tool (Color page) enables you to apply a color to any of these zones. On a monochrome or gray-scale monitor, the colors are rendered in gray patterns based on the luminosity of the color.

The variant shows how to change a widget's colors programmatically.

Basic Steps

1. In a canvas, select the widget whose color you want to set.
1. In a Properties Tool (Color page), select the desired color from the color chart. Alternatively, you can access one of the standard, named color constants from a pull-down menu in the color box of the Properties Tool. You can also revert to the widget's default colors through the policy colors submenu in the same pull-down menu.
2. Select the color zone in the Properties Tool.
3. Apply the properties and install the canvas.

Variant

Changing a Widget's Colors Programmatically

Online example: ColorExample

1. In a method in the application model, get the widget's wrapper from the application model's builder.
2. Get the LookPreferences from the wrapper and create a copy with the desired color. The copy is created when a color-zone message is sent: foregroundColor:, backgroundColor:, selectionForegroundColor:, or selectionBackgroundColor:. The argument is the desired new color.
3. Install the new LookPreferences by sending a lookPreferences: message to the wrapper. The argument is the new LookPreferences.

foregroundColor: aColor

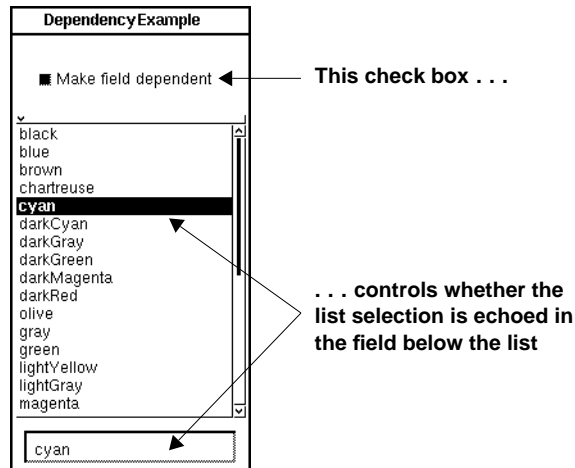
"For each sample widget, change the indicated color layer."

```
| wrapper lookPref |
self sampleWidgets do: [ :widgetID |
    wrapper := (self builder componentAt: widgetID).      "Variant Step 1"
    lookPref := wrapper
        lookPreferences foregroundColor: aColor.          "Variant Step 2"
    wrapper lookPreferences: lookPref].                    "Variant Step 3"
```

See Also

- “Creating a Color” on page 686

Adding and Removing Dependencies



Strategy

When a widget's value is changed, such as when an item is selected from a list, the application often needs to react in some way. A common reaction is to update other widgets based on the new value. You can arrange for such a reaction, typically as part of the initialization process. This is known as *setting up a dependency* or *registering an interest*.

You can also bypass the dependency when unusual circumstances arise. For example, when two widgets depend on each other, one of them must bypass the dependency mechanism to avoid infinite recursion. The variants show two ways of bypassing a dependency.

The first variant removes the dependency and relies on the application to reestablish it after the value has been changed.

The second variant bypasses all dependencies, including that of the widget's view. Thus, you must ask the widget to update its view programmatically. This variant also bypasses any dependencies that may have been established by objects other than the application model and the widget, but that is not a common situation.

Basic Step

Adding a Dependency

Online example: DependencyExample

- In the application model's initialize method (typically), send an `onChangeSend:to:` message to the widget's value holder. The first argument is a message, which will be sent to the second argument. The second argument is typically the application model itself.

initialize

```
colorNames := SelectionInList with: ColorValue constantNames.
selectedColor := String new asValue.
fieldsDependent := false asValue.
```

"Arrange for the application model to take action when the check box is turned on or off."

```
fieldsDependent
```

```
  onChangeSend: #changedDependency to: self.
```

"Basic Step"

Variants

V1. Removing a Dependency by Retracting the Interest

Online example: DependencyExample

1. Send a `retractInterestsFor:` message to the widget's value holder. The argument is the object that registered the interest, typically the application model itself.
2. After the value has been changed, register the interest again as shown in the basic step.

changedDependency

"Turn on or off the dependency link between the list and the input field, depending on the value of the check box."

```
| valueModel |
```

```
valueModel := self colorNames selectionIndexHolder.
```

```
self fieldsDependent value
  ifTrue:
    [valueModel onChangeSend: #changedSelection to: self]"V1 Step 2"
  ifFalse:
    [valueModel retractInterestsFor: self]. "V1 Step 1"
```

V2. Bypassing All Dependencies

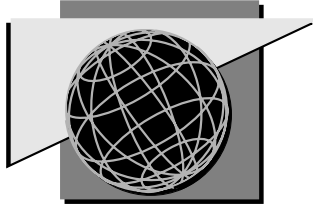
Online example: FieldConnectionExample

1. Send a `setValue:` message to the widget's value holder instead of the usual `value:` message. The argument is the widget's new value.
2. Get the widget from the application model's builder and ask the widget to update itself with the new value.

`changedB`

```
"Use setValue: to bypass dependents, thus avoiding circularity."
self bSquared setValue: (self b value raisedTo: 2). "V2 Step 1"
```

```
"Since dependents were bypassed when the model was updated,
update the view manually."
(self builder componentAt: #b2) widget update: #value. "V2 Step 2"
```

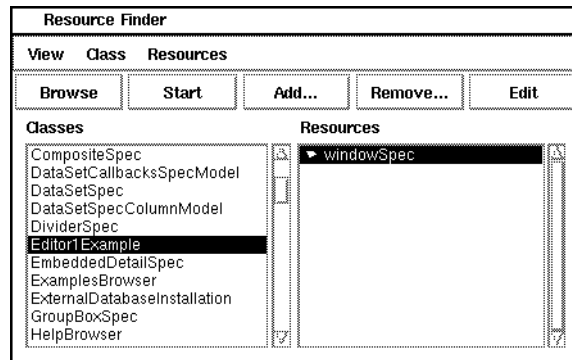


Chapter 4

Windows

Opening a Window	82
Getting a Window from a Builder	85
Sizing a Window	86
Moving a Window	90
Changing a Window's Label	92
Refreshing a Window's Display	93
Coloring a Window	94
Adding and Removing Scroll Bars	96
Adding a Menu Bar	98
Getting the Active Window	99
Getting the Window at a Specific Location	100
Closing a Window	101
Expanding and Collapsing a Window	103
Hiding a Window	104
Making a Window a Slave	105
Setting a Window's Icon	108

Opening a Window



Strategy

The usual way of opening a running window is to ask an application model to open one of its interface specifications (an interface specification is created when you install a painted canvas in an application model). You can do so programmatically or by using a Resource Finder.

You can also create an instance of `ApplicationWindow` and open it programmatically. This is rarely necessary, but it does offer more flexibility because you can control the window's type (normal, dialog, or pop-up) as well as its contents.

Basic Steps

1. In a Resource Finder, select the application (such as `Editor1Example`).
2. Click on the `Start` button in the Resource Finder.

Variants

V1. Opening a Default Canvas Programmatically

This is the programmatic equivalent of basic steps 1 and 2.

- Send an open message to the application model.

```
Editor1Example open "V1 Step"
```

V2. Opening a Main Canvas by Spec Name

When the spec name is `#windowSpec`, you can just send `open` as in the variant above. When you want to open a different spec on a new instance of an application, use this variant.

Some specs are meant to be opened only after the application has reached a certain state—that is, after the variables on which the widgets depend have been properly initialized. In those situations, use step 1 in variant 3. This variant is provided for situations when your main window's spec has to be named other than `#windowSpec`.

- Send `openWithSpec:` to the application class.

```
Editor1Example openWithSpec: #windowSpec "V2 Step"
```

V3. Opening a Secondary Canvas by Spec Name

When the same application model serves one or more secondary canvases in addition to the main canvas, you can open a secondary canvas with this variant. The example is the `openFinder` method implemented by the class named `HelpBrowser`.

A “secondary” canvas implies that the application has reached the proper state—that is, the instance variables required by the interface have been initialized. In the `HelpBrowser`, the main window must be opened before the secondary canvas named `#finderSpec` is opened.

This example creates a new `UIBuilder` the first time it is invoked, and it stores that builder in an instance variable. When your application needs to access widgets on the

secondary canvas later, storing this second builder assures you will have a means of accessing the widgets.

1. In a method in the application model, create a new `UIBuilder`.
2. Tell the builder which object will supply its menus, aspects, and other resources by sending it a `source:` message. The argument is typically the application model itself. (Alternatively, you can send a series of `aspectAt:put:` messages to install the resources directly.)
3. Create the spec object and add the spec to the builder.
4. Open the window.

openFinder

"Open the Search window. If already open, raise to top."

```
| bldr |
(self finderBuilder notNil and: [self finderBuilder window isOpen])
  ifTrue: [self finderBuilder window raise]
  ifFalse: [
    self finderBuilder: (bldr := UIBuilder new).           "V3 Step 1"
    bldr source: self.                                     "V3 Step 2"
    bldr add: (self class
              interfaceSpecFor: #finderSpec).           "V3 Step 3"
    bldr window
      application: self;
      beSlave.
    self adjustSearchScope.
    self searchStatus value: 0.
    (bldr componentAt: #searchStatus) widget
      setMarkerLength: 5.

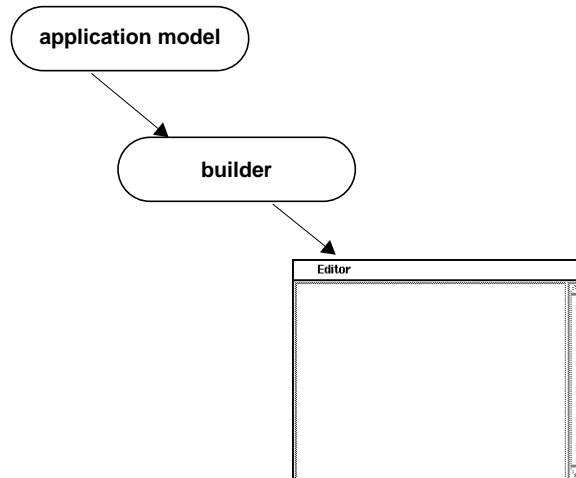
    bldr openAt: (self                                     "V3 Step 4"
                 originFor: bldr window
                 nextTo: #findButton)].

(self builder componentAt: #listView) takeKeyboardFocus.
```

See Also

- "Getting a Window from a Builder" on page 85
- "Dialogs" on page 277

Getting a Window from a Builder



Strategy

When you ask an application model to open an interface specification, the application model creates an interface builder, which in turn creates the specified window and its contents. Your application code can manipulate the window programmatically by obtaining the window from the builder and then sending it messages.

Each application model holds onto the builder for its primary window. In addition, you can arrange for your application model to hold onto additional builders created for assembling any secondary windows.

Basic Step

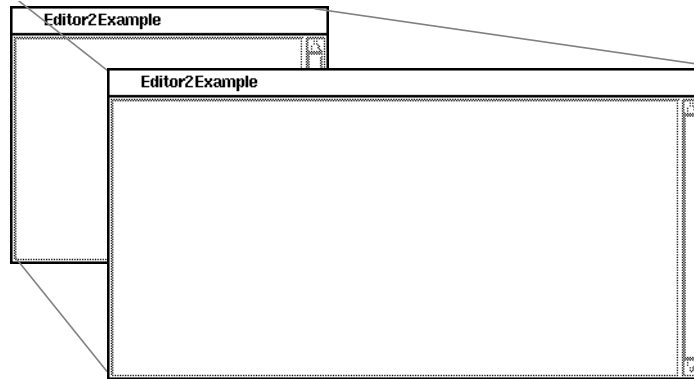
Online example: Editor2Example

- Ask the builder for its window.

```
| bldr win |
bldr := Editor2Example open.
win := bldr window.
win label: 'Editor'.
```

"Basic Step"

Sizing a Window



Strategy

You can control the initial size of a window as well as the minimum and maximum sizes. These sizes can be established either programmatically or by directly sizing a canvas.

Basic Steps

1. While editing a canvas, use the window manager to resize it.
2. Make sure no widget is selected in the canvas. This selects the canvas itself.
3. Select the `layout→window→pref size` command in the canvas's menu to set the initial size of the window.
4. Install the canvas.

Variants

V1. Setting the Initial Size Programmatically

Online example: Editor2Example

1. Build an interface up to the point of opening the window.
2. Get the window from the interface builder.
3. Ask the window to open with a specified size (extent).

```
| bldr win |
bldr := Editor2Example new allButOpenInterface: #windowSpec.
win := bldr window.
win openWithExtent: 500@220.                                     "V1 Step 3"
```

V2. Constraining the Size Using the Canvas

When the interface becomes unusable below a certain minimum size, or when larger than a certain maximum size, you can impose limits on the size. Then when the user tries to make the window larger or smaller than is reasonable, the window maintains a useful size.

1. While editing the canvas, use the window manager to resize the canvas to its intended minimum size.
2. Make sure no widget is selected in the canvas.
3. Select the layout→window→min size command in the canvas's menu.
4. Install the canvas.

For maximum size, in step 3 use the layout→window→max size command.

V3. Constraining the Size Programmatically

1. Give the window a minimum size and/or a maximum size.
2. Open the window. Then try to resize the window beyond its minimum or maximum size.

```
| bldr win |
bldr := Editor2Example new allButOpenInterface: #windowSpec.
win := bldr window.
win minimumSize: 100@100;                                     "V3 Step 1"
  maximumSize: 500@300;
  open.                                                       "V3 Step 2"
```

V4. Making the Size Unchangeable (Fixed)

In step 3 of variant 2, use the layout→window→fixed size command. To accomplish the same thing programmatically, in step 1 of variant 3 set the minimum size equal to the maximum size.

V5. Changing the Size of an Open Window

- Give the window a new display box, which is the rectangle within which it displays itself, using screen coordinates.

```
| bldr win |  
bldr := Editor2Example open.  
win := bldr window.  
win displayBox: (100@100 extent: 400@220). "V5 Step"
```

V6. Clearing All Size Constraints on a Canvas

1. In the window's canvas, make sure no widget is selected.
2. Select the `layout→window→clear all` command in the canvas's menu. Install the canvas.

V7. Determining a Window's Dimensions

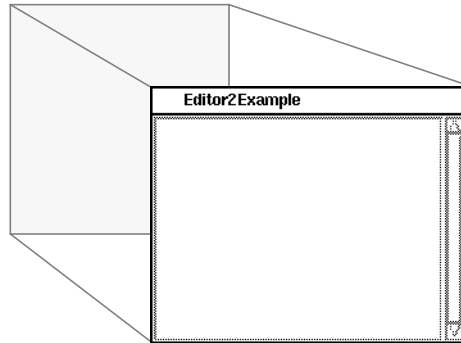
1. Ask the window for its minimum size.
2. Ask for its maximum size.
3. Ask for its display box.
4. Ask the display box for its width.
5. Ask the display box for its height.
6. Display the parameters in the Transcript.

```
| win min max box width height |  
win := (Editor2Example open) window.  
  
min := win minimumSize. "V7 Step 1"  
max := win maximumSize. "V7 Step 2"  
box := win displayBox. "V7 Step 3"  
width := box width. "V7 Step 4"  
height := box height. "V7 Step 5"  
  
Transcript "V7 Step 6"  
show: 'Min: ', min printString; cr;  
show: 'Max: ', max printString; cr;  
show: 'Box: ', box printString; cr;  
show: 'Width: ', width printString; cr;  
show: 'Height: ', height printString; cr
```

See Also

- “Moving a Window” on page 90

Moving a Window



Strategy

You can move a window that is already open and you can also set its location at startup. Both of these operations are performed programmatically.

Moving a canvas has no effect on the startup location of its window.

Your prompt-for-open preference also affects the startup location of a window unless you specify a location programmatically as shown here.

Variants

V1. Setting the Startup Location of a Window

1. Build the interface up to the point of opening the window.
2. Get the window from the interface builder.
3. Ask the window to open itself within a specified rectangle, using screen coordinates, in pixels.

```
| bldr win |  
bldr := Editor2Example new  
    allButOpenInterface: #windowSpec. "V1 Step 1"  
win := bldr window. "V1 Step 2"  
win openIn: (50@50 extent: win minimumSize). "V1 Step 3"
```

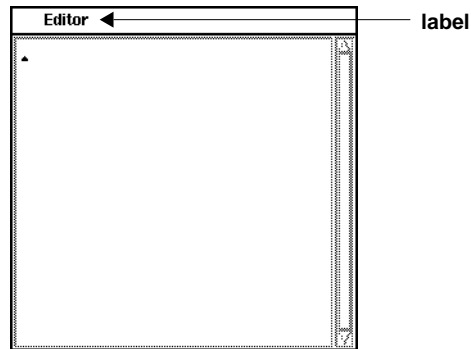
V2. Moving an Open Window

- Ask the window to relocate its origin (upper-left corner) to a specified point, using screen coordinates.

```
| win |  
win := (Editor2Example open) window.  
(Delay forSeconds: 1) wait.  
win moveTo: 300@50.
```

"V2 Step"

Changing a Window's Label



Strategy

You can modify a window's label using the canvas or, for dynamic control, by sending a message to the window.

Basic Steps

1. In the canvas for the window, make sure no widget is selected.
2. In the Properties Tool, fill in the window's Label property with the desired label.
3. Apply the properties and install the canvas.

Variant

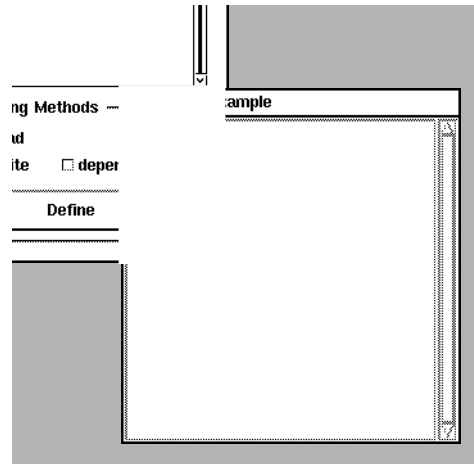
Changing the Label Programmatically

- Send a `label:` message to the window, with the new label as argument.

```
| win |  
win := (Editor2Example open) window.  
win label: 'Editor'.
```

"Variant Step"

Refreshing a Window's Display



Strategy

Under normal conditions, a window redisplay its contents whenever those contents change or whenever an overlapping window is moved. Sometimes you need to redisplay a window programmatically, as when you want to display an intermediate state of the window before a drawing operation has been completed.

Basic Step

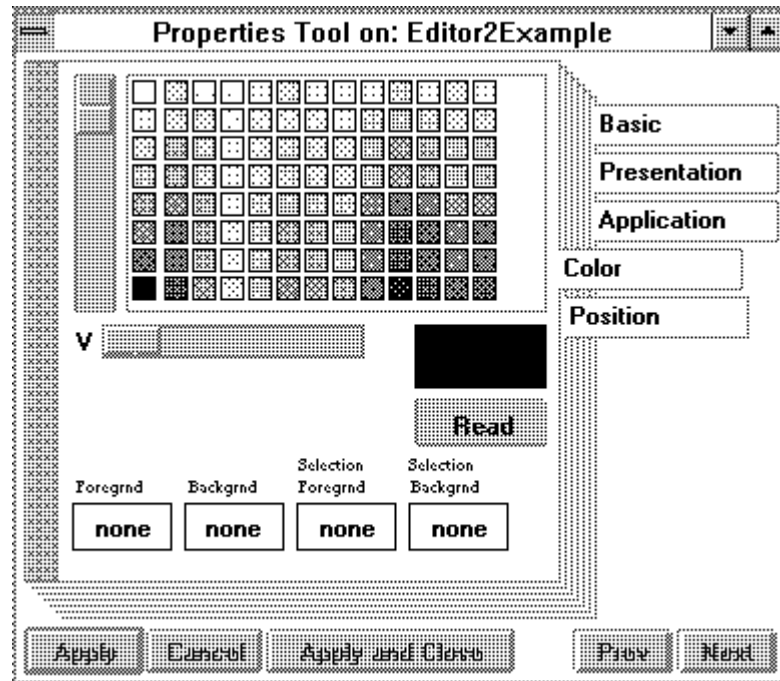
Online example: Editor2Example

- Send a display message to the window.

```
| win |
win := (Editor2Example open) window.
5 timesRepeat: [
  (Delay forMilliseconds: 400) wait.
  win display].
```

"Basic Step"

Coloring a Window



Strategy

You can use the Properties Tool to change the background color of a window. Changing the foreground or selection color has no effect in the case of a window.

You can also change the color of a window programmatically for run-time control. Doing so enables you to use window color as a visual cue to indicate a change in some property of your application.

Limitation: Under any look policy other than OS/2 or Windows, widgets inherit the background color of the window until you explicitly make them opaque and apply a different color to their backgrounds. Also, make sure you choose a background color for the window that contrasts sufficiently with scroll bars.

Basic Steps

1. In the canvas for the window, make sure no widget is selected.
2. In a Properties Tool (Color page), select the color for the background.
3. Apply the properties and install the canvas.

Variant

Changing the Color Programmatically

Online example: Editor2Example

In the following example, we create a loop that is repeated for each of the color constants. For each color, the window background is changed and the window is redisplayed.

- Send a background: message to the window, with the color as argument.

```
| win color |  
win := (Editor2Example open) window.
```

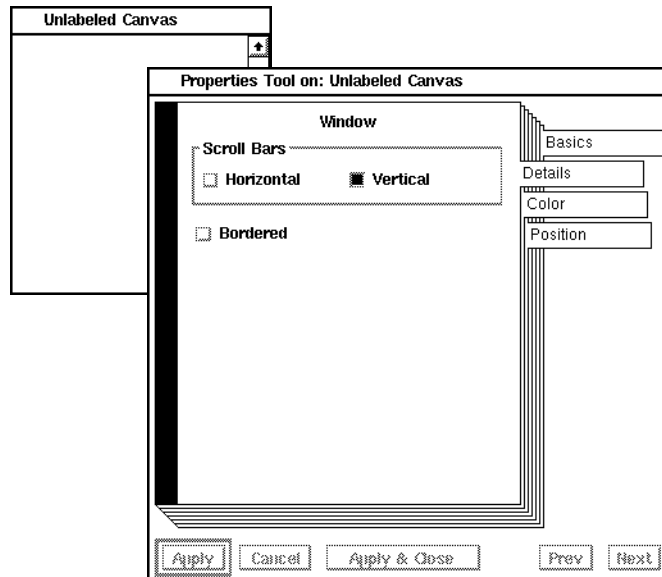
```
ColorValue constantNames do: [ :colorName |  
  (Delay forMilliseconds: 200) wait.  
  color := ColorValue perform: colorName.  
  win background: color.  
  win display]
```

"Variant Step"

See Also

- "Creating a Color" on page 686

Adding and Removing Scroll Bars



Strategy

The Properties Tool enables you to add vertical and/or horizontal scroll bars to a window.

You can also add and remove scroll bars programmatically. You can do so, however, only in the way shown here when the window had a scroll bar to start with. Thus, to add a vertical scroll bar while your application is running (but not before), you must turn on the vertical scroll bar property before installing the canvas and then remove the scroll bar programmatically before opening the window. This equips the window with a `BorderDecorator`, which is the object that is empowered to supply scroll bars.

Basic Steps

1. In the window's canvas, make sure no widget is selected.
2. In a Properties Tool (Details page), turn on the desired scroll bars.
3. Apply the properties and install the canvas.

Variant

Adding and Removing Scroll Bars Programmatically

1. After opening the window, remove the scroll bars that are meant to be displayed later.
2. Ask the window's component to add scroll bars.

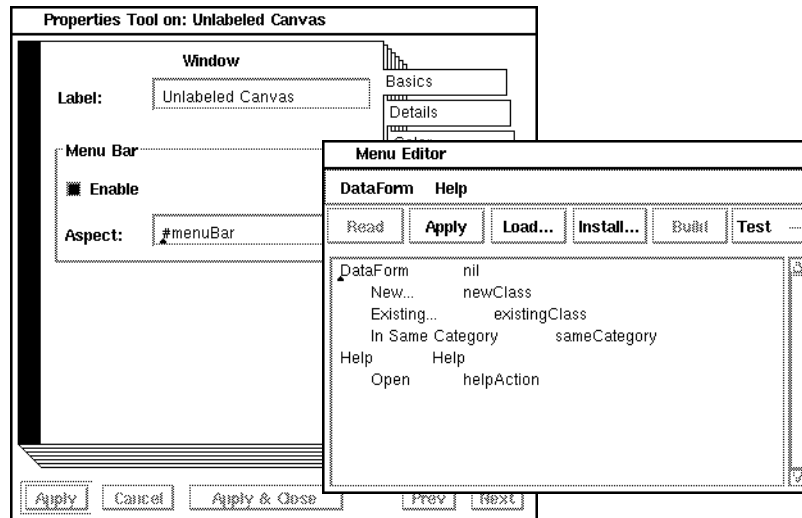
```
| win |  
win := ApplicationWindow new.  
win component: (BorderDecorator  
    on: Object comment asComposedText).  
win open.
```

```
win component                                "Variant Step 1"  
    noVerticalScrollBar;  
    noHorizontalScrollBar.  
win display.
```

```
Cursor wait showWhile: [  
    (Delay forSeconds: 2) wait].
```

```
win component                                "Variant Step 2"  
    useVerticalScrollBar;  
    useHorizontalScrollBar.
```

Adding a Menu Bar



Strategy

Adding a menu bar has two parts: turning on a menu bar property for the window and creating the underlying menu.

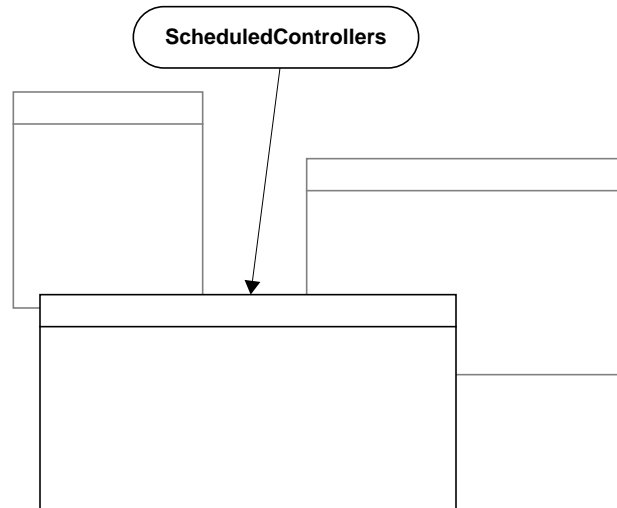
Basic Steps

1. In the canvas for the window, make sure no widget is selected.
2. In a Properties Tool, turn on the Enable switch for the Menu Bar property.
3. In the Menu field, enter the name of the menu-creation method.
4. Install the canvas.
5. Use the Menu Editor to create the menu. Each first-level entry in the menu appears in the menu bar, but *only* when it has a submenu. That is, a menu bar displays menu names, not command names.

See Also

- "Creating a Menu" on page 226

Getting the Active Window



Strategy

The `ScheduledControllers` object keeps track of all controllers, including the active controller. You can ask the active controller for its associated window.

Although this maneuver is rarely needed in application code, it is often useful in ad hoc experiments when you want to display an object on a Workspace window.

Basic Step

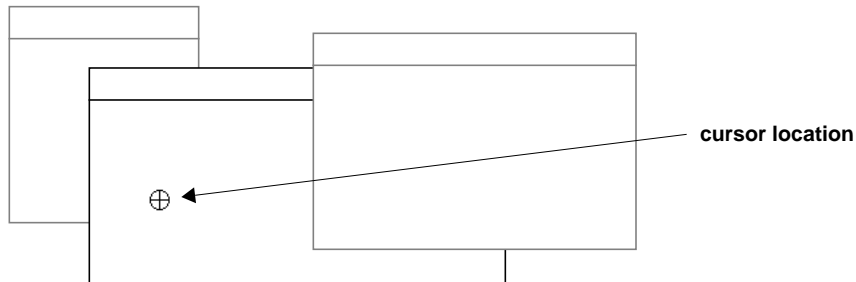
- Ask the active controller for its associated window (that is, the `topComponent` associated with the controller's view).

```
| win |
win := ScheduledControllers
    activeController view topComponent.           "Basic Step"
win moveTo: 20@20.
```

See Also

- "Getting a Window from a Builder" on page 85

Getting the Window at a Specific Location



Strategy

When your application performs an operation on a window that is pointed to by the user (using the mouse), you can access the window as shown in the basic steps. Drag-and-drop operations, in particular, rely on this technique.

Basic Steps

1. Prompt the user to point at a window by sending a `waitButton` message to the current controller's sensor. It's a good idea to change the cursor while waiting, so the user knows that input is expected.
2. Get the cursor location in screen coordinates by sending a `globalCursorPoint` message to the controller's sensor.
3. Get the window at the cursor point by sending a `windowAt:` message to the default `Screen`. The argument is the cursor location. (In the example, the window's component flashes so you can verify that the correct window was accessed.)

```

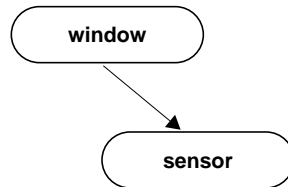
| sensor pt window |
sensor := ScheduledControllers activeController sensor.
Cursor bull showWhile: [sensor waitButton].           "Basic Step 1"
pt := sensor globalCursorPoint.                       "Basic Step 2"

window := Screen default windowAt: pt.                "Basic Step 3"

window component flash.

```

Closing a Window



Strategy

The window manager provides the user with a means of closing a window. Closing a window programmatically is useful mainly when the user exits from the application in some other way, such as clicking on a Quit button. You might also want to close a window as a side effect of some conclusive user action.

As with a window-closing event that is initiated using the window manager, the techniques shown below are safe—that is, the window’s model is notified in case it wants to take some precaution such as confirming the action. The variant shows how to set up such a confirmer.

Basic Step

When an application model is running one or more windows, you can close it (or all of them at once, if there is more than one) by sending `closeRequest` to the application.

- Ask the application model to close its associated windows.

```

| editor |
editor := Editor2Example new.
editor openInterface: #windowSpec.
(Delay forSeconds: 1) wait.
editor closeRequest.
  
```

"Basic Step"

Variant

Arranging for Final Actions When Closing a Window

When an application window has been asked to close, it first sends a `changeRequest` message to its application model. If the model answers `false`, the window won't close; if it answers `true`, the window proceeds to close itself. Thus, the model has a chance to verify that no damage will be done if the window is closed.

For example, as shown below, the Image Editor (`UIMaskEditor`) uses a `changeRequest` method to confirm the user's intent to abandon any unsaved changes in the image.

- Implement a `changeRequest` method in your application model, which answers `true` when the window can close and `false` otherwise.

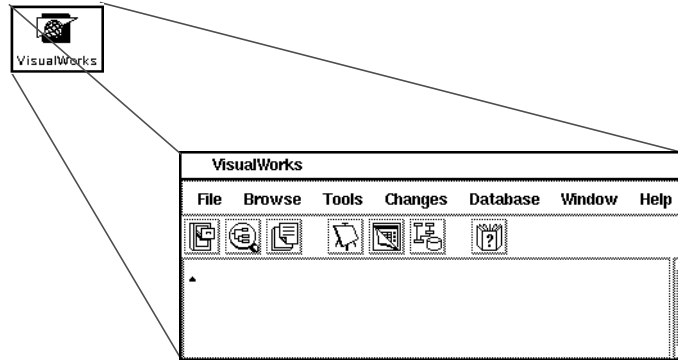
```
changeRequest                                     "Variant Step"
    ^super changeRequest
      ifFalse: [false]
      ifTrue: [(self modified or: [self magnifiedBitView controller
updateRequest not])
              ifTrue:
                [Dialog confirm: 'The image has been altered, but not installed.
Do you wish to discard the changes?']
              ifFalse: [true]]
```

Notice also in the example above that the inherited version of `changeRequest` is first invoked to preserve any precautions that a parent class may have implemented.

See Also

- “Making a Window a Slave” on page 105

Expanding and Collapsing a Window



Strategy

Window managers typically provide a means of collapsing (iconifying) a window and expanding it back to its normal state. You can also control that behavior programmatically.

Basic Steps

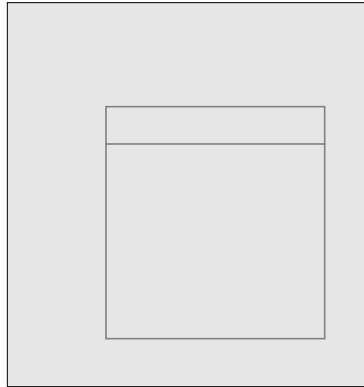
1. Send a collapse message to the window.
2. Send an expand message to the window.

```
| win |
win := (Editor2Example open) window.
win display.
(Delay forSeconds: 1) wait.
win collapse.                                     "Basic Step 1"
(Delay forSeconds: 1) wait.
win expand.                                       "Basic Step 2"
```

See Also

- “Hiding a Window” on page 104
- “Making a Window a Slave” on page 105

Hiding a Window



Strategy

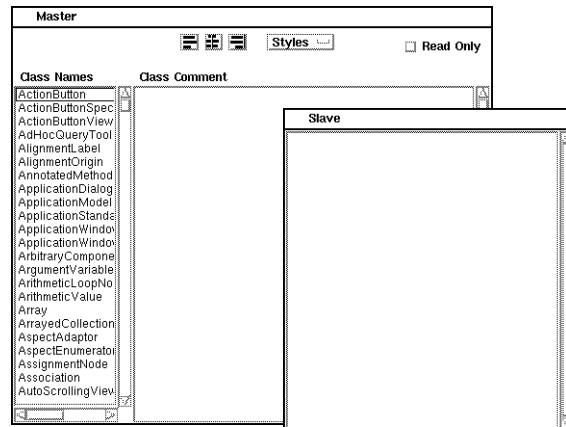
A window is a relatively expensive object, because it holds a visual component that is often bulky and because it allocates a display surface using the window manager. When your application needs to open and close a window repeatedly, it is not necessary to reconstruct it each time. Instead, you can unmap it, which hides the window without disassembling it. Then you can simply map it to redisplay it.

Basic Steps

1. Ask the window to unmap itself.
2. Ask the window to map itself.

```
| win |  
win := (Editor2Example open) window.  
win display.  
(Delay forSeconds: 1) wait.  
win unmap. "Basic Step 1"  
(Delay forSeconds: 1) wait.  
win map. "Basic Step 2"
```

Making a Window a Slave



Strategy

In a multiwindow application, it is often helpful to close all secondary windows automatically when the user closes the main window. In this situation, the main window is called the *master window* and the secondary windows are called *slave windows*.

Basic Steps

1. Tell the master window which application model to inform of its events.
2. Tell the master window to be a master.
3. Tell the slave window which application model will relay events from the master window.
4. Tell the slave window to be a slave.

```
| app masterWin slaveWin |
app := Editor1Example new.
masterWin := (app openInterface) window.
masterWin
    label: 'Master';
    application: app;
    beMaster.
```

"Basic Step 1"
"Basic Step 2"

```
slaveWin := (Editor2Example open) window.  
slaveWin  
  label: 'Slave';  
  application: app; "Basic Step 3"  
  beSlave. "Basic Step 4"
```

Variants

V1. Make Windows Equal Partners

When you want to be able to close all of your application's windows by closing any one of them, make them partners instead of master and slaves.

- Tell the windows to be partners.

```
| app win1 win2|  
app:= Editor1Example new.  
  
win1 := (app openInterface) window.  
win1  
  label: 'Partner 1';  
  application: app;  
  bePartner. "V1 Step"  
  
win2 := (Editor2Example open) window.  
win2  
  label: 'Partner 2';  
  application: app;  
  bePartner. "V1 Step"
```

V2. Choosing the Events That Are Sent

By default, master and partner windows broadcast the following events: #close, #collapse, and #expand. You can remove any of those events, and you can add any of the following: #bounds, #enter, #exit, #hibernate, #reopen, and #release.

- Tell the master or partner window which events to broadcast.

```
| app masterWin slaveWin |
app := Editor1Example new.

masterWin := (app openInterface) window.
masterWin
  label: 'Master';
  application: app;
  beMaster;
  sendWindowEvents: #( #close #collapse
    #expand #hibernate #reopen).                                "V2 Step"

slaveWin := (Editor2Example open) window.
slaveWin
  label: 'Slave';
  application: app;
  beSlave.
```

V3. Choosing the Events That Are Received

By default, slave and partner windows mimic the following events: #close, #collapse, and #expand. Controlling the events that are received lets each slave be selective according to its needs.

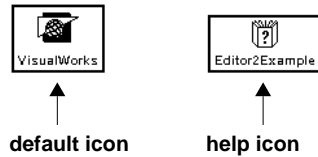
- Tell the slave or partner window which events to receive.

```
| app masterWin slaveWin |
app := Editor1Example new.

masterWin := (app openInterface) window.
masterWin
  label: 'Master';
  application: app;
  beMaster.

slaveWin := (Editor2Example open) window.
slaveWin
  label: 'Slave';
  application: app;
  beSlave;
  receiveWindowEvents: #( #close).                                "V3 Step"
```

Setting a Window's Icon



Strategy

Under window managers that support iconified windows, the default icon appears as shown in the illustration above. You can assign a different icon, perhaps a custom icon that you have created.

Basic Step

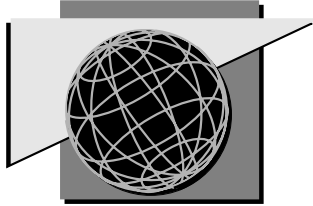
- Tell the window which icon to use. (If your window manager supports iconification, try collapsing the window after you open it.)

```
| helpIcon win |  
helpIcon := Icon image: VisualLauncher BWHelp24.  
win := (Editor2Example open) window.  
win icon: helpIcon.
```

"Basic Step"

See Also

- “Creating an Icon” on page 682



Chapter 5

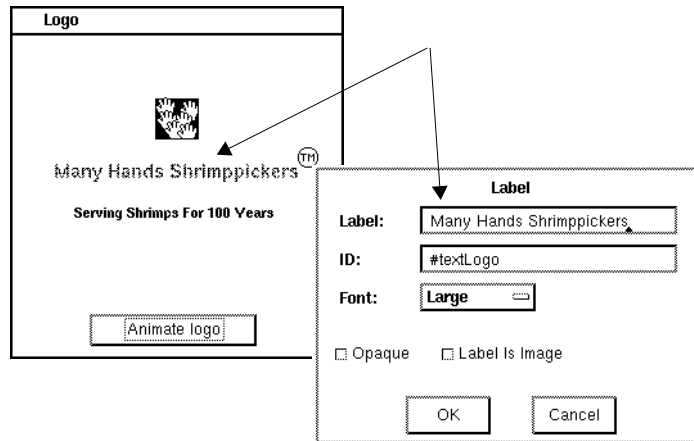
Labels

Creating a Textual Label	110
Creating a Graphic Label	111
Supplying the Label at Run Time	113
Changing Font, Emphasis, and Color	116
Building a Registry of Labels	118

See Also

- “Widget Basics” on page 53

Creating a Textual Label



Strategy

A label is most often used in conjunction with another widget, such as a field, to describe the purpose of the field. It is also used by itself as a title for a group of widgets or a window. Since the text of a label can be changed while the application is running, a label can also be used for read-only display.

Multiline label: A label accommodates only a single line of text. For a multiline label, use a separate label for each line or use a read-only text widget.

Basic Steps

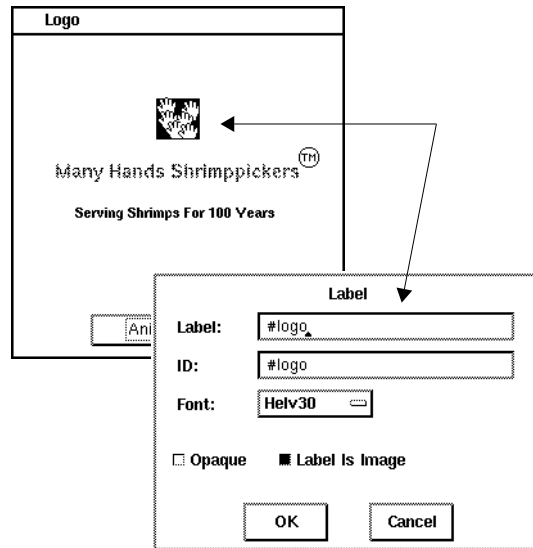
Online example: LogoExample

1. Use a Palette to place a label widget on the canvas. Don't worry about the size—it will expand to accommodate your text. Leave the label selected.
2. In a Properties Tool, fill in the label's Label property with the text of the label.
3. Apply the properties and install the canvas.

See Also

- "Changing the Fonts Menu" on page 587

Creating a Graphic Label



Strategy

Use a graphic label when you want to add a pictorial element to an interface. The graphic can be changed while the application is running, so you can also use a graphic label to represent a changing aspect of the model pictorially.

Passive vs. active: A graphic label is passive. Use a graphic button when the graphic is meant to respond to a mouse click.

Scroll bars: For a large graphic that requires scroll bars, insert the graphic in a view holder instead.

Basic Steps

Online example: LogoExample

1. Use a Palette to place a label widget on the canvas, and leave the label selected.
2. In a Properties Tool, fill in the label's Label property with the name of the method that you will create to supply the graphic image (in the example, logo). Do not prefix the name with the pound sign (#); this will be added automatically.

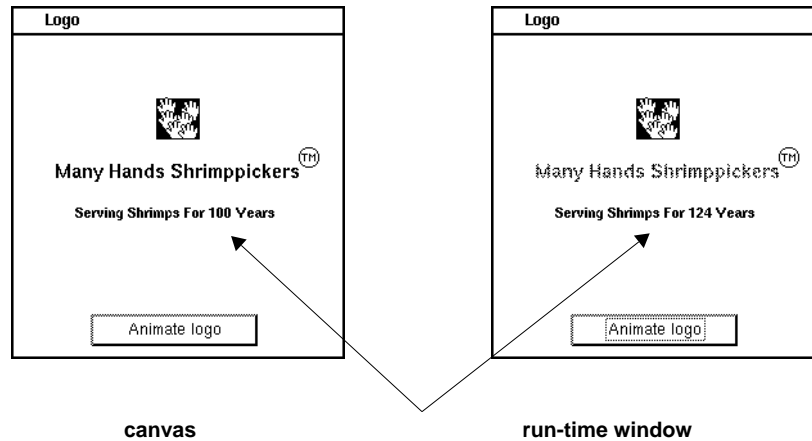
3. In Properties Tool, turn on the Label is Image property.
4. Apply the properties and install the canvas.
5. Use the Image Editor or other means to create the graphic image and install it in the application model, using the method name from step 2. Put the method in a class protocol named `resources`.

Hint: The graphic will appear in the running application. You can see the graphic in the canvas if you install the image resource before you fill in and apply the label's properties.

See Also

- “Giving a Button a Graphic Label” on page 167
- “Integrating a View into an Interface” on page 389
- “Creating a Graphic Image” on page 658

Supplying the Label at Run Time



Strategy

You can change the content of a label programmatically while the application is running.

This technique enables you to use a label instead of a field for read-only display of a text. The advantage is that a label requires less mechanism than a field (no instance variable, no accessing method, and no initialization code). The disadvantage is that updating a label is more awkward than updating a field.



Caution: When you supply a longer text or a larger graphic, you run the risk of overlapping neighboring widgets, if any.

Basic Steps

Online example: LogoExample

1. In a method in the application model (such as `postBuildWith()`), get the widget from the application model's builder and send a `labelString:` message to it. The argument is the new label string.
2. When the replacement label is in the form of a `ComposedText` (which can have boldness, color, etc.), get the widget from

the builder and send a label: message to it. The argument is the composed text.

postBuildWith: aBuilder

"Update the slogan's text, and make the company name bold and red."

| slogan txt emph label |

"Insert the years-in-business into the slogan."

slogan := 'Serving Shrimps For '

, (Date today year - 1869) printString, ' Years'.

aBuilder componentAt: #slogan labelString: slogan.

"Basic Step 1"

"Make the company name bold and red."

txt := 'Many Hands Shrimppickers' asText

emph := Array

with: #bold

with: #color->ColorValue red.

txt emphasizeFrom: 1 to: 10 with: emph.

label := Label

with: txt

attributes: (TextAttributes styleNamed: #large).

(aBuilder componentAt: #textLogo) label: label.

"Basic Step 2"

Hint: The example updates a label before the canvas is opened, but you can change the label string at any time after the interface has been built.

Variant

Supplying a Graphic Label's Image at Run Time

Online example: LogoExample

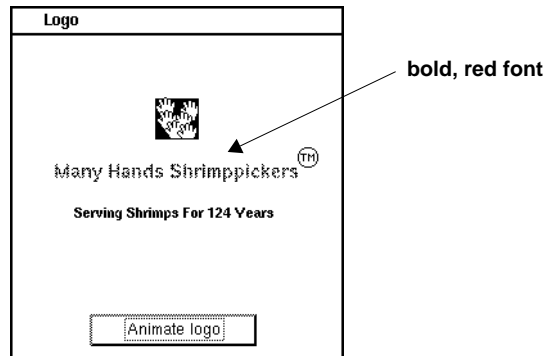
- Get the widget from the builder and send a label: message to it, with the new graphic as argument.

animateLogo

"Display the logo in successively larger sizes (as a way of demonstrating dynamic updating of a graphic label)."


```
| logo widget view animationRegion |  
logo := self class logo.  
widget := self builder componentAt: #logo.  
animationRegion := widget bounds.  
view := self builder composite.  
  
10 to: 1 by: -1 do: [ :factor |  
    (Delay forMilliseconds: 100) wait.  
    widget label: (logo shrunkenBy: factor @ factor).      "Variant Step"  
    view invalidateRectangle: animationRegion repairNow: true]
```

Changing Font, Emphasis, and Color



Strategy

You specify a label's font by choosing the font in the label's properties. The chosen font applies to the entire label. Alternatively, you can change the font programmatically or mix emphases (bold, italic, etc.) and colors.

The advantage of mixing font emphases within a single label rather than creating multiple labels is that the spacing between the parts of the label will be preserved even when you run the image on a platform that supplies different fonts.

Mixing fonts: Although you can mix font families in the same label to the limited extent that you can apply or remove serifs from a portion of the text, generally you must create a separate label for each font family.

Basic Steps

Online example: LogoExample

1. In a method in the application model (such as `postBuildWith:`), create a `Text` by sending an `asText` message to the label string.
2. Add the desired emphases to the text.
3. Create a `Label` with the text and the desired font.
4. Get the widget wrapper from the builder (with `componentAt:`) and install the new label with `label:`.

postBuildWith: aBuilder

"Update the slogan's text, and make the company name bold and red."

| slogan txt emph label |

"Insert the years-in-business into the slogan."

slogan := 'Serving Shrimps For '
 , (Date today year - 1869) printString, ' Years'.
 (aBuilder componentAt: #slogan) labelString: slogan.

"Make the company name bold and red."

txt := 'Many Hands Shrimppickers' asText.

"Basic Step 1"

emph := Array

"Basic Step 2"

with: #bold

with: #color->ColorValue red.

txt emphasizeFrom: 1 to: 10 with: emph.

label := Label

with: txt

attributes: (TextAttributes styleNamed: #large).

"Basic Step 3"

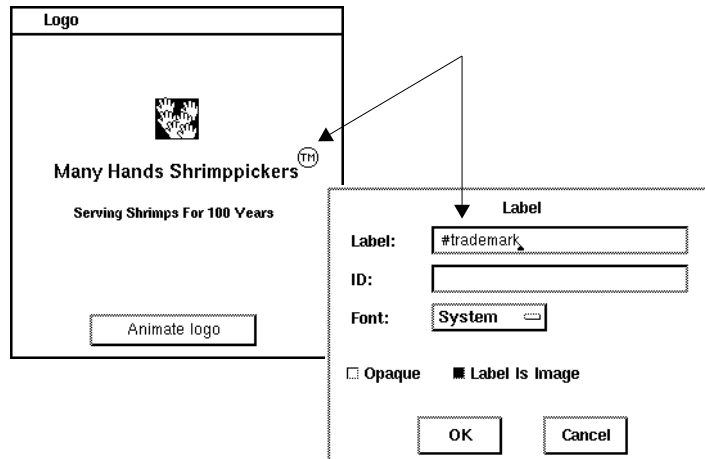
(aBuilder componentAt: #textLogo) label: label.

"Basic Step 4"

See Also

- "Applying Boldfacing and Other Emphases" on page 572
- "Creating a Custom Text Style" on page 576
- "Setting Text Color" on page 585

Building a Registry of Labels



Strategy

When you plan to use the same label (such as a company name or logo) in multiple interfaces, you can store it in a central registry. The system will look for the label there when it does not find the usual resource method.

Two separate registries are available, one for graphics and the other for strings. The basic steps show how to register both kinds of labels. The variant shows how to remove an entry from a registry.

Memory usage: Use these registries sparingly, especially when graphic images are involved rather than strings, because each entry occupies memory until it is explicitly removed.

Basic Steps

Registering a Graphic Label

Online example: LogoExample

1. To register a graphic image, send a `visualAt:put:` message to the `ApplicationModel` class. The first argument is the name of the label, as defined in the `Label` property of the widget. This

is usually done in a class-initialization method, so the registration will occur whenever the class is filed into a new image.

2. To register a string label, send a `labelAt:put:` message to the `ApplicationModel` class. The first argument is the name of the label as defined in the `Label` property of the widget.

initialize

"LogoExample initialize"

"Register the graphic image for the trademark symbol."

`ApplicationModel`

"Basic Step 1"

`visualAt: #trademark`

`put: self trademark.`

"Register the textual version of the trademark symbol."

`ApplicationModel`

"Basic Step 2"

`labelAt: #tm`

`put: '(TM)'.`

3. Execute the initialization method.

Variant

Removing an Entry from a Registry

1. To get the graphic labels registry, send a `visuals` message to the `ApplicationModel` class. To get the string labels registry, send a `labels` message.
2. The registry is a dictionary, so use the standard message (`removeKey:ifAbsent:`) for removing an entry from a dictionary. The first argument is the name of the label, as identified in the `Label` property of the widget. The second argument is a block containing the action to be taken if the label is not found, frequently an empty block for no action.

"Visual registry"

| registry |

`registry := ApplicationModel visuals.`

"Variant Step 1"

`registry removeKey: #trademark ifAbsent: [].`

"Variant Step 2"

Chapter 5 Labels

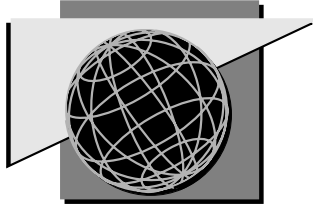
"Labels registry"

registry := ApplicationModel labels.

registry removeKey: #tm ifAbsent: [].

"Variant Step 1"

"Variant Step 2"



Chapter 6

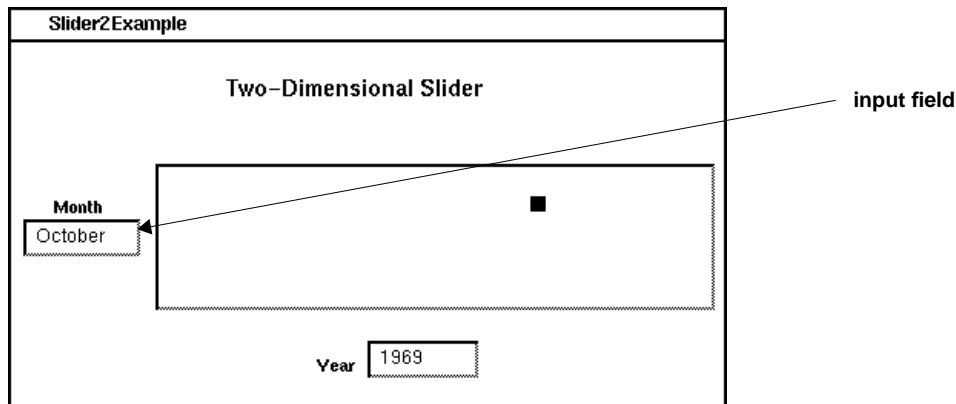
Input Fields

Creating an Input Field	122
Restricting the Type of Input	125
Formatting Displayed Data	129
Validating the Input	132
Modifying a Field's Pop-Up Menu	139
Connecting a Field to Another Field	143
Restricting Entries in a Field (Combo Box)	146
Moving the Insertion Point	150

See Also

- “Widget Basics” on page 53

Creating an Input Field



Strategy

An input field is used for both entering and displaying data. You can also use a field in read-only mode when you just want to display data. When a field has a short list of valid entries, consider using a menu button or a combo box instead.

A field is designed to use some kind of value model to manage the data it presents. When the field accepts input from a user, it sends this data to the value model for storage; when the field needs to update its display, it asks its value model for the data to be displayed.

The basic kind of value model for a field is a value holder (an instance of `ValueHolder`), which stores the data by holding it in an instance variable. A value holder is most appropriate for data that is not held elsewhere in the application. If the relevant data is to be held in a domain model, you can set up the field with another kind of value model, namely, an aspect adaptor (an instance of `AspectAdaptor`) which stores and retrieves the data directly from the domain model. Otherwise (if a value holder is used), the application model must be programmed to copy the relevant data between the domain model and the value holder.

Basic Steps

Online example: Slider2Example

1. Use a Palette to add a field to the canvas (such as the Month field).
2. Optionally, add a label to the canvas, and fill in the Label property to describe the field's contents.
3. Use the widget handles to size and position the field. Leave the field selected.
4. In a Properties Tool, fill in the field's Aspect property with the name of the method (month) that will return a value model for the field's data.
5. Apply the properties and install the canvas.
6. Use the canvas's define command or a System Browser to add an instance variable (month) to the application model. The instance variable will hold the value model for the field's data.
7. Use the canvas's define command or a System Browser to create the aspect method that you named in step 3 (month).

```
month                                     "Basic Step 7"
  ^month
```

8. Use a System Browser to initialize the instance variable you created in step 6 (month), either in the aspect method or in a separate initialize method. In this example, you use the latter to initialize the variable with a value holder that holds the desired month. (You create the value holder by sending asValue to the data object.)

```
initialize
  month := (Date nameOfMonth: 1) asValue.          "Basic Step 8"
  year := 1900 asValue.

  dateRange := (0@1) asValue.
  dateRange onChangeSend: #changedDate to: self.
```

Variants

V1. Aligning a Field's Contents

- Set the field's **Align** property to **Left** (to start the data at the left side of the field), **Center** (to center the data), or **Right** (to place the data against the right margin).

V2. Creating a Read-Only Field

- Turn on the field's **Read Only** property.

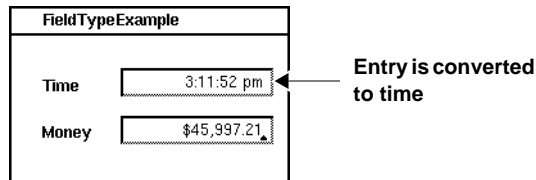
V3. Restricting the Size of User Input

- In the field's **Size** property, enter the number of characters that you want to allow. (When the user tries to enter characters beyond that limit, they are ignored.)

See Also

- “Adding a Menu Button” on page 236
- “Adapting Domain Models to Widgets” on page 703

Restricting the Type of Input



Strategy

You specify the type of input that a field is to accept by setting its `Type` property. This property tells the field to convert the user's input string into an appropriate kind of object before sending it to the value model. If the conversion cannot be performed, the field flashes and continues to display the unaccepted string without storing it in its value model. When the field updates its display with data from its value model, it converts the data object to a display string.

You can choose from the following data types:

- **String**—Input is stored as an instance of `ByteString`. This is the default property setting.
- **Symbol**—Input is stored as an instance of `Symbol` and is displayed with a prepended pound sign (#). Useful for programming applications that manipulate method selectors.
- **Text**—Input is stored as an instance of `Text`, which can have emphasis (boldness, etc.).
- **Number**—Input is stored as an appropriate subclass of `Number`. Acceptable input includes literal expressions for integers, single or double floating-point numbers, scientific notation, or radix notation.
- **Password**—Input is stored as a string, with an asterisk (*) displayed for each character the user enters. (The real characters are sent to the field's value model.)
- **Date**—Input is converted into an instance of `Date`.
- **Time**—Input is converted into an instance of `Time`.
- **Timestamp**—Input is converted into an instance of `Timestamp`, which combines a date and a time.

- **FixedPoint(2)**—Input is converted to an instance of `FixedPoint` that represents a fixed point number with two decimal places. Useful for applications that manipulate monetary amounts.
- **Boolean**—Input is stored as an instance of `Boolean`. Acceptable values are `true` and `false`.
- **ByteArray**—Input is stored as an instance of `ByteArray`.
- **Object**—Input is evaluated as a Smalltalk expression; the resulting object is stored as the field's value. The field redisplay this object using the object's `printString` method.

The `Format` property provides predefined formatting alternatives for some of these types.

You can also add a custom data type to the list, as described in the variant below.

Basic Steps

1. Select the field in the canvas.
2. In the Properties Tool, set the field's `Type` property to the desired data type.
Hint: Ensure that the field is initialized with the appropriate type of data.
3. Apply the properties and install the canvas.

Variant

Creating a Custom Data Converter

Online example: `FieldTypeExample`

This example shows how to create a data converter for handling an instance of `Time`. Note this converter, as given below, is unnecessary, because you can set the `Type` property to `Time`. However, you can use it as a model for building converters for other kinds of objects.

1. Use a Palette to add a field to the canvas. Leave the field selected.

2. In a Properties Tool, fill in the Aspect property with the name of the method (time) that will return a value model for the field. Apply the property and install the canvas.
3. Use a System Browser to create an instance method (timeToText) in the initialize algorithm protocol of the TypeConverter class. Use an existing method in that protocol as your template. The method is responsible for initializing the getBlock and putBlock of a PluggableAdaptor (ignore the updateBlock).
4. Use a System Browser to create a class method (onTimeValue:) in the instance creation protocol of TypeConverter. Use an existing method in that protocol as your template. The method is responsible for returning a new instance of TypeConverter, which is initialized using the method from the preceding step.
5. Ensure that the object responds to the messages sent to it by the method in step 3 (often printOn: will suffice).
6. Use a System Browser to create the instance variable (time) that you named in step 2 in the application model (FieldTypeExample).
7. Use a System Browser to create the aspect method (time) for the instance variable. The method is responsible for initializing the variable with a TypeConverter, using the instance creation method you defined in step 4.

```

timeToText                                     "Variant Step 3"
    "Initialize the receiver to perform the action
    when assigned a value."

    self
        getBlock: [:m | m value == nil
            ifTrue: [String new]
            ifFalse: [m value printString]]
        putBlock: [:m :v | v isEmpty
            ifTrue: [m value: nil]
            ifFalse: [m value: (Time readFrom: v readStream)]]
        updateBlock: [:m :a :p | true]

```

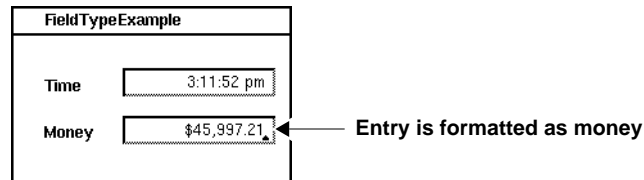
```

onTimeValue: aValue                             "Variant Step 4"
    ^(self on: aValue) timeToText

```

```
time                                     "Variant Step 7"  
  ^time isNil  
    ifTrue: [time := (TypeConverter onTimeValue: Time now asValue)]  
    ifFalse: [time]
```

Formatting Displayed Data



Strategy

A field displays a string representation of the data stored by its value model. For some types of data, the displayed string can be formatted in various ways. For example, numbers can be formatted as phone numbers, monetary units, percentages, and so on.

The basic steps show how to use property settings to choose among alternative predefined formats for certain types of data. The first three variants describe some useful predefined formats. The final variant describes how to create a custom format programmatically.

For applications that are to be deployed in locations other than the United States, VisualWorks offers a separate mechanism (Locale objects) for adapting to local formatting conventions. This mechanism is described in the *International User's Guide*. For this mechanism to take effect, you must set the *Type* property to *Number*, *Time*, or *Date*, and leave the *Format* property setting blank.

Basic Steps

1. Select the field in the canvas.
2. In the Properties Tool, set the field's *Format* property to the desired data format. Note that the field's *Type* property setting determines the kinds of available formats, if any.
3. Apply the properties and install the canvas.

Variants

V1. Displaying a U.S. Phone Number

1. Select the field in the canvas.
2. In the Properties Tool, set the field's **Type** property by selecting either **String** or **Number**, as desired.
3. In the Properties Tool, set the field's **Format** property by selecting either `(@@@) @@@-@@@@` (for type **String**) or `(000) 000-0000` (for type **Number**). In a **String** format, the `@` stands for a single character. In a **Number** format, the `0` stands for exactly one digit.

V2. Displaying a U.S. Dollar Amount

1. Select the field in the canvas.
2. In the Properties Tool, set the field's **Type** property by selecting either **Number** or **FixedPoint(2)**, as desired.
3. In the Properties Tool, set the field's **Format** property by selecting any of the formats that begin with `$`—for example, `##,##0.00;[Red]($#,##0.00)`

In formats such as this, the `0` stands for exactly one digit, and the `#` stands for zero or one digit. This example also specifies that negative numbers be displayed in red and enclosed in parentheses.

V3. Displaying a Date

1. Select the field in the canvas.
2. In the Properties Tool, set the field's **Type** property by selecting **Date**.
3. In the Properties Tool, set the field's **Format** property by selecting a date format such as `d-mmm-yy`.

In formats such as this, the symbol `d` stands for the minimum number of digits representing the day, `mmm` stands for a three-letter abbreviation of the month name, and `yy` stands for the two-digit year number.

V4. Creating a Custom Format

Online example: FieldTypeExample

A `TypeConverter` enables a field to display a number in a special format, such as a monetary format. You define the format as a string that uses the same conventions as the predefined formats. For more information about format conventions, use a System Browser to read the class comments for the `NumberPrintPolicy`, `TimestampPrintPolicy`, and `StringPrintPolicy` classes.

This example shows how to create a format for a monetary amount. Note this format, as given below, is unnecessary, because you can set the `Format` property to a predefined format. However, you can use it as a model for building other kinds of formats.

1. Use a Palette to add a field to the canvas.
2. Fill in the `Aspect` property, naming the method (`price`) that returns a value model for the field. Apply the property and install the canvas.
3. Use a System Browser to create the instance variable (`price`) in the application model.
4. Use a System Browser to create the aspect method (`price`), in which you initialize the value to a `TypeConverter` that uses the desired format string.

<code>price</code>		<code>"V4 Step 4"</code>
<code>^price isNil</code>		
<code> ifTrue: [price := (TypeConverter</code>		
<code> onNumberValue: 0 asValue</code>		
<code> format: '\$###,###,###.##')]</code>		
<code> ifFalse: [price]</code>		

Validating the Input

FieldValidInputExample	FieldValidationExample	FieldValidation2Example
Username: <input type="text" value="guest"/> Access Code: <input type="text" value="0"/>	<input type="text" value="PW4099"/> Punctuation is not accepted	<input type="text" value="HP700"/> Valid partial entry is completed for you (Try S, A, H or I)

Strategy

Frequently, only certain entries are valid for a particular field. For example, you might want to restrict input to a numeric range such as 0 to 999 or check for undesirable characters.

Validating whole entries: You can arrange for a typed entry to be validated when the user accepts it (that is, when the user chooses accept from the field's <Operate> menu, or presses the <Tab> or <Return> key to move focus out of the field). You arrange for validation by specifying one or more *validation callbacks* (messages for the widget to send when asked to accept input or change focus). You implement corresponding methods in the application model to test the input, warn the user if it is unacceptable, and, if desired, prevent further actions until valid input is entered. The basic steps implement the following callbacks for a field that accepts strings:

- A *change validation callback*, which prevents input from being passed to the value model unless the it is valid
- An *exit validation callback*, which prevents the user from moving focus out of the field until the input is corrected

The first variant implements the same kinds of callbacks for a field whose `Type` property is set to `Number`.

Character-by-character validation: You can arrange for the user to get immediate feedback after typing an invalid character. The character might be illegal under any circumstances, in which case you can simply intercept the keyboard event and check the character (second variant). Or you might want to validate the partly completed entry against a list of valid entries after each keystroke (third variant).

Basic Steps

Validating String Entries

Online example: FieldValidInputExample

1. Use the Palette to add a field labeled Username: to the canvas. Leave the field selected.
2. In a Properties Tool, fill in the Aspect property with the name of the method (username) that will return a value model for the field. Leave the Type property set to String.
3. On the Validation page of the Properties Tool, fill in the Change property with the name (validateUsername:) of the change validation callback. This specifies the method that will determine whether to accept input into the field's value model.
4. On the Validation page of the Properties Tool, fill in the Exit property with the name (validateUsername:) of the exit validation callback. This specifies the method that will determine whether the field can give up focus. In this example, the same method is used for both change and exit validation.
5. Apply properties and install the canvas.
6. Use the canvas's define command or a System Browser to add an instance variable (username) and aspect method (username) to the application model. Initialize the instance variable with a value model—for example, in an initialize method.
7. Use a System Browser to create the method (validateUsername:) corresponding to the callback you named in steps 3 and 4. Note that, because the name ends in a colon, the method must accept a Controller as an argument.
8. Send an editValue message to the field's Controller to obtain the user's entry. The entry must be obtained from the controller instead of the value model, because validation occurs before the entry has been passed to the value model.
9. Validate the entry (in this case, check its length).
10. If the entry is valid, return true. This permits the field to pass the entry to the value model and, if requested, give up focus.

11. If the user's entry is not valid, warn the user and return false. This tells the field to wait until the user enters a valid entry.

validateUsername: aController	"Basic Step 7"
"Check the length of the entered username. Warn the user if the entered input is too long."	
entry lengthLimit	
lengthLimit := 6.	
entry := aController editValue.	"Basic Step 8"
"If the username is too long, warn the user (and reject the input)."	
^entry size <= lengthLimit	"Basic Step 9"
ifTrue: [true]	"Basic Step 10"
ifFalse: [Dialog warn: 'Please enter only ', lengthLimit printString ,	
' characters.'	
false]	"Basic Step 11"

Variants

V1. Validating Non-String Entries

Online example: FieldValidInputExample

1. Use the Palette to add a field labeled **Access Code:** to the canvas. Leave the field selected.
2. In a Properties Tool, fill in the **Aspect** property (accessCode) and change the **Type** property (in this example, to **Number**).
3. On the **Validation** page of the Properties Tool, set the **Change** and **Exit** properties (in this case, enter validateAccessCode:).
4. Apply properties and install the canvas.
5. Use the canvas's **define** command or a **System Browser** to add an **instance variable** (accessCode) and **aspect method** (accessCode) to the application model. Initialize the instance variable with a value model.
6. Use a **System Browser** to create the method (validateAccessCode:) corresponding to the callback you named in steps 3.

7. Send a `hasEditValue` message to the field's Controller to find out whether the user's entry can be converted from a string to the specified type (in this case, a number).
8. If the input string was successfully converted, send `editValue` to the controller to obtain the converted entry.
9. If the input string could not be converted, warn the user and return `false` to tell the field to wait for valid input.
10. Validate the successfully converted entry, if any (in this case, check whether the number is in the correct range).
11. If the entry is valid, return `true` to pass the entry to the value model and, if requested, give up focus.
12. If the user's entry is not valid, warn the user and return `false` to wait until the user enters a valid entry.

```

validateAccessCode: aController                                "V1 Step 6"
"Check whether the user entered a number. If not, warn the user. If so, check
whether the number is in the right range. If not, warn the user."

```

```

| entry lowerLimit upperLimit |
lowerLimit := 50.
upperLimit := 100.

```

```

"Test whether the input can be converted to a number.
If not, warn the user and reject the input. If so, get the number."
aController hasEditValue                                        "V1 Step 7"
  ifTrue: [ entry := aController editValue.]                  "V1 Step 8"
  ifFalse: [Dialog warn: 'Enter a number.'.
            ^false].                                         "V1 Step 9"

```

```

"If the access code is in the wrong range, warn the user and
reject the input."
^(lowerLimit < entry) & (entry < upperLimit)                  "V1 Step 10"
  ifTrue: [true]                                             "V1 Step 11"
  ifFalse: [Dialog warn: 'Enter a number between', lowerLimit printString ,
            ' and ', upperLimit printString, '.'.
            false]                                          "V1 Step 12"

```

V2. Validating Each Keystroke

Online example: FieldValidation1Example

1. In a Properties Tool, give the field an ID property (in the example, the ID is codeField). Apply the property and install the canvas.
2. Use a System Browser to add a postBuildWith: instance method to the application model (FieldValidation1Example), in which the first task is to get the controller from the field.
3. In the postBuildWith: method, send a keyboardHook: message to the controller. The argument to keyboardHook: is a block that takes two arguments: the keyboard event and the controller.
4. Inside the block, invoke the method (keyPress:) that you will create to validate the keystroke. (You can also put the validation code directly inside the block.)

```
postBuildWith: aBuilder                                     "V2 Step 2"
| ctrlr |
ctrlr := (aBuilder componentAt: #codeField) widget controller.
ctrlr keyboardHook: [:ev :c |                               "V2 Step 3"
self keyPress: ev].                                       "V2 Step 4"
```

5. Use a System Browser to create the keyPress: method, which takes the keyboard event as its argument, extracts the character, and validates it.
6. As the last step in the keyPress: method, return the event when you want to forward the keyboard event for normal processing. Return nil to bypass normal processing.

```
keyPress: ev                                             "V2 Step 5"
"Validate the character."

| ch ascii |
ch := ev keyValue.

"Allow tab and cr."
ascii := ch asInteger.
(ascii == 9 or: [ascii == 13])
ifTrue: [^ev].
```

```

ch isAlphaNumeric
  ifFalse: [
    Dialog warn: 'Please enter only letters and digits'.
    ^nil].
^ev

```

"V2 Step 6"

V3. Validating a Partial Entry After Each Keystroke

Online example: FieldValidation2Example

1. Use a System Browser to add a `postBuildWith:` method to the application model (FieldValidation2Example) in an interface opening protocol.
2. In the `postBuildWith:` method, register an interest in the field's value holder (productCode).
3. In the `postBuildWith:` method, send a `continuousAccept:` message to the field's controller, with an argument of true.
4. Use a System Browser to add the change method that was named in step 2 (`codeChanged`), in a change messages protocol. This method is responsible for performing the validation.

```

postBuildWith: aBuilder
  "Ask the field's controller to accept continuously -- that is, to
  return the entry to the model after each keystroke."
  | ctrlr |
  self productCode onChangeSend: #codeChanged to: self.
  ctrlr := (aBuilder componentAt: #codeField) widget controller.
  ctrlr continuousAccept: true.

```

"V3 Step 1"

"V3 Step 2"

"V3 Step 3"

```

codeChanged
  "The code entry was changed -- if the (partial) entry
  uniquely identifies a valid product code, fill in the rest
  of the code for the user."
  | entry qualifiedCodes holder |
  holder := self productCode.
  entry := holder value.

```

"V3 Step 4"

"If the partial entry uniquely identifies a valid product code,

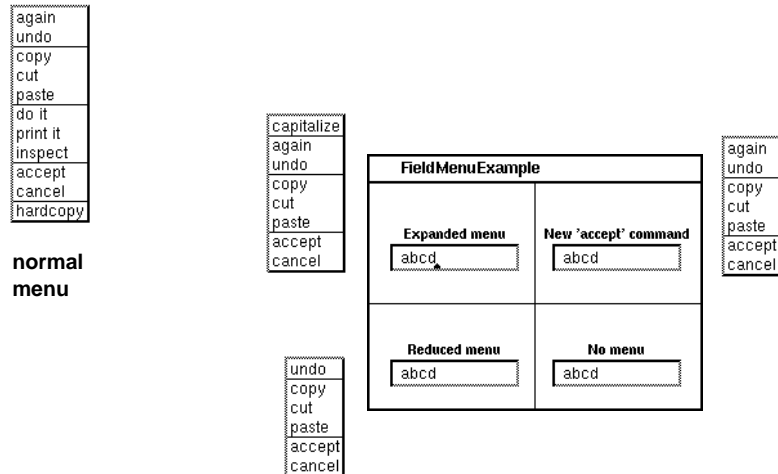
```
fill in the rest of the code for the user."
entry := entry, "".
qualifiedCodes := OrderedCollection new.
self validCodes do: [ :code |
    (entry match: code)
    ifTrue: [qualifiedCodes add: code]].

qualifiedCodes size == 1
ifTrue: [
    holder retractInterestsFor: self.
    holder value: qualifiedCodes first.
    holder onChangeSend: #codeChanged to: self].
```

See Also

- “Creating a Custom Adaptor (PluggableAdaptor)” on page 717

Modifying a Field's Pop-Up Menu



Strategy

By default, a field has the menu of text-editing commands that is shown above. You can add or omit commands, override the action that is associated with a command, or disable the menu entirely.

A field's menu is usually oriented toward commands. Although you can arrange for a field's menu to contain a list of valid entries, this is properly the job of a menu button.

Basic Steps

B1. Adding a Command

Online example: FieldMenuExample

1. In a canvas, select the field.
2. In a Properties Tool, fill in the field's Menu property with the name of the method that you will create to supply a custom menu (expandedMenu).

3. Use a System Browser to add the menu-creating method (expandedMenu) to the application model in a menu messages protocol.

```
expandedMenu                                     "B1 Step 3"
    "Add a command to the default text-editing menu."

    | mb |
    mb := MenuBuilder new.
    mb
        add: 'capitalize'->#capitalize;
        line;
        addDefaultTextMenu.

    ^mb menu
```

4. Use a System Browser to add the method (capitalize) that is invoked by the newly added command. Put the method in the menu messages protocol.

```
capitalize                                       "B1 Step 4"
    "Capitalize the field's contents."

    self field1 value: (self field1 value
        collect: [ :ch | ch asUppercase]).
```

B2. Omitting a Command

Do basic steps 1 through 3 above (substituting reducedMenu for expandedMenu).

- In the menu-creating method, build the default menu from its parts, omitting the command that you don't want to include.

```
reducedMenu                                     "B2 Step"
    "Omit one of the commands (#again) from the default text-editing menu."

    | mb |
    mb := MenuBuilder new.
    mb
        add: 'undo'->#undo;
```

```

line;
addCopyCutPaste;
line;
addAcceptCancel.
^mb menu

```

Variants

V1. Overriding a Default Command

Do basic steps 1 through 3 above (substituting `newAcceptMenu` for `expandedMenu`).

1. In the menu-creating method, build the default menu from its parts. For the command that you want to override, provide the name of your overriding method as the value (`#newAccept`).

`newAcceptMenu`

"Redefine the 'accept' command by invoking a local alternate."

```

| mb |
mb := MenuBuilder new.
mb
  addFindReplaceUndo;
  line;
  addCopyCutPaste;
  line;
  add: 'accept'->#newAccept;           "V1 Step 1"
  add: 'cancel'->#cancel.
^mb menu

```

2. Use a System Browser to create the overriding method (`newAccept`).

`newAccept`

Transcript show: self field3 value; cr.

V1 Step 2"

V2. Disabling a Field's Menu

Do basic steps 1 through 3 (substituting `noMenu` for `expandedMenu`).

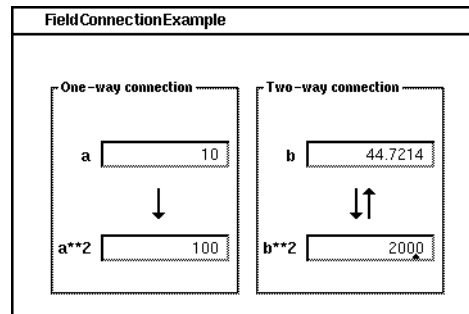
- In the menu-creating method (`noMenu`), return a block containing `nil`. When asked for its menu, the field will evaluate this block, and no menu is displayed.

<code>noMenu</code>	
<code>^[nil]</code>	"V2 Step"

See Also

- "Creating a Menu" on page 226
- "Adding a Menu Button" on page 236

Connecting a Field to Another Field



Strategy

When the value in a field depends on the value in another field, you can link them using the built-in dependency mechanism.

Use a one-way connection when only one of the fields can affect the other, as when a data field updates a total field.

Use a two-way connection when a change in either field affects the other field, as when a lookup of a customer record can be initiated by entering either the customer's name or the customer's ID number.

Basic Step

Creating a One-Way Connection

Online example: FieldConnectionExample

1. Use a System Browser to add a `postBuildWith:` method to the application model in an interface opening protocol.
2. In the `postBuildWith:` method, register an interest in the field that originates updates, naming a method to be invoked when that field is changed (`changedA`).

```
postBuildWith: aBuilder                                     "Basic Step 1"
  self a onChangeSend: #changedA to: self.                 "Basic Step 2"
```

3. Use a System Browser to add the change method (changedA) to the application model in a change messages protocol. That method updates the dependent field's model.

```
changedA                                     "Basic Step 3"
    self aSquared value: (self a value raisedTo: 2).
```

Variant

Creating a Two-Way Connection

Online example: FieldConnectionExample

1. Use a System Browser to add a postBuildWith: method to the application model (FieldConnectionExample) in an interface opening protocol.
2. In the postBuildWith: method, register interests in both fields, naming the method to be invoked when each field is changed (changedB, changedBSquared).

```
postBuildWith: aBuilder                       "Variant Step 1"
    self b onChangeSend: #changedB to: self.  "Variant Step 2"
    self bSquared onChangeSend: #changedBSquared to: self.
```

3. Use a System Browser to add the change methods (changedB, changedBSquared) to the application model in a change messages protocol. Those methods update the dependent field's model in a way that avoids circularity.

```
changedB                                     "Variant Step 3"
    "Use setValue: to bypass dependents, thus avoiding circularity."
    self bSquared setValue: (self b value raisedTo: 2).

    "Since dependents were bypassed when the model was updated,
    update the view manually."
    (self builder componentAt: #b2) widget update: #value.
```

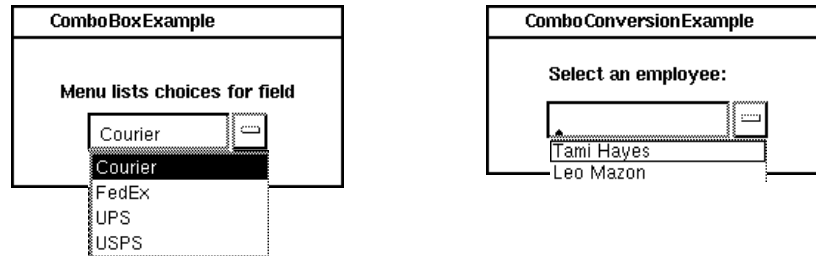
```
changedBSquared                             "Variant Step 3"
    "Use setValue: to bypass dependents, thus avoiding circularity."
    self b setValue: (self bSquared value raisedTo: (1/2)).
```

"Since dependents were bypassed when the model was updated,
update the view manually."
(self builder componentAt: #b) widget update: #value.

See Also

- "Connecting a Slider to a Field" on page 267

Restricting Entries in a Field (Combo Box)



Strategy

Frequently, an input field must be restricted to a group of standard entries. For example, a field that identifies the shipping instructions in an order-entry application might use a standard set of entries because there are a limited number of ways to ship a package. A combo box is ideally suited to this situation because it combines a field with a pull-down list of the standard entries for the field. The basic steps show how to create a combo box.

A menu button can be used in the same situation, but more programming effort is required to coordinate it with the field.

Basic Steps

Online example: `ComboBoxExample`

1. Use a Palette to add a combo-box widget to the canvas. Leave the combo box selected.
2. In a Properties Tool, fill in the combo box's **Aspect** property with the name of the method (in the example, `shipper`) that returns a value model for the combo box.
3. In the combo box's **Choices** property, enter the name of the method (`ShipperChoices`) that returns a collection of entry choices.
4. In the combo box's **Type** property, choose the type of input the widget is to accept (see "Restricting the Type of Input"). Set the **Format** property, as appropriate (see "Formatting Displayed Data").

5. Apply the properties and install the canvas.
6. Use a System Browser or the Define button to create the instance variable (shipper) and accessing method (shipper) for the aspect.

shipper	"Basic Step 6"
^shipper	

7. Use a System Browser to create the method that you named in step 3 (shipperChoices). The method returns a value holder containing the list of valid entries. The value holder can be held in an instance variable (as in the example).

shipperChoices	"Basic Step 7"
^shipperChoices	

8. Initialize the field's aspect variable, typically in an initialize method. Initialize the variable with a value model containing data of the type specified in step 4.
9. Initialize the choices variable with a value holder containing the list of valid entries. Initialize this list with data of the type specified in step 4.

```

initialize

    | list |
    shipper := 'Courier' asValue.                                     "Basic Step 8"

    list := List new.
    list add: 'Courier';                                           "Basic Step 9"
    list add: 'FedEx';
    list add: 'UPS';
    list add: 'USPS'.
    shipperChoices := list asValue.                                 "Basic Step 9"

```

Variant

You can arrange for a combo box to display a list of choices that are arbitrary objects (for example, a list of Employee objects). You do this by supplying a *print method* and a *read method* that translate the relevant objects into displayable elements (for

example, Strings or graphical images) and back. For example, in `ComboConversionExample`, the `print` method enables the combo box to display Employee names in the pull-down list and, when an Employee is selected, to display that Employee's name in the field. The `read` method enables the combo box to interpret the user's input as an Employee name, which can be matched with an existing Employee, or used to create a new one.

Online example: `ComboConversionExample`

1. In the Properties Tool, set the **Type** property of the combo box to **Object**.
2. Fill in the **Print** property with the name of a method for converting the relevant objects to strings (in this example, `employeeToString`). The name must end with a colon.
3. Fill in the **Read** property with the name of a method for converting strings to objects of the desired type (in this example, `stringToEmployee`). The name must end with a colon.
4. In the application model, create a print method with the name you specified in step 2 (`employeeToString`). This method accepts an object from the choices list as an argument (in this case, an instance of `Employee`).
5. In the print method, return a `String` that represents the object from the choices list. In this example, display the name of the `Employee`. The string is displayed in the combo box's pull-down list and also in the combo box's field when the choice is selected.

<code>employeeToString: anEmployee</code>	"Variant Step 4"
<code>"Return a String for representing the Employee in the combo box's list and field."</code>	

<code>^anEmployee name.</code>	"Variant Step 5"
--------------------------------	------------------

6. Create a read method with the name you specified in step 3 (`stringToEmployee`). This method accepts a `String` argument.
7. In the read method, return an object for the given `String`. In this example, determine whether the `String` is the name of an `Employee` in the choices list; if so return that `Employee`. Otherwise, create a new `Employee` and add it to the choices list.

stringToEmployee: aString "Variant Step 6"
"Return an Employee corresponding to the given String. If the String corresponds to the name of an Employee on the choices list, return that Employee. Otherwise, create a new Employee and add it to the list."

```
| theEmp |  
theEmp := self employeeChoices value  
         detect: [:each | each name = aString]  
         ifNone: [nil].
```

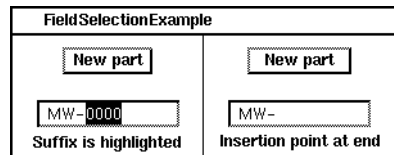
```
theEmp isNil  
  ifTrue:  
    [theEmp := Employee new name: aString.  
     self employeeChoices value addLast: theEmp].
```

```
^theEmp
```

See Also

- "Restricting the Type of Input" on page 125
- "Formatting Displayed Data" on page 129

Moving the Insertion Point



Strategy

You can control the position of the insertion point in a field programmatically. For example, the data in a field might have a prefix that rarely changes—you could highlight the suffix for convenient editing. In that case, the “insertion point” is actually a portion of the field’s text, which will be replaced by the user’s entry.

When the suffix has yet to be filled in, you can simply position the insertion point at the end of the prefix.

Variants

V1. Highlighting a Portion of a Field

Online example: FieldSelectionExample

1. In a method in the application model, ask the field’s wrapper to takeKeyboardFocus.
2. Tell the field’s controller the indices (character positions) of the substring that is to be highlighted.

addPart

"Put a template in the partID field, then highlight the suffix."

```
| wrapper |
self partID value: 'MW-0000'.
```

```
wrapper := self builder componentAt: #part1.
wrapper takeKeyboardFocus.
wrapper widget controller selectFrom: 4 to: 7.
```

"V1 Step 1"

"V1 Step 2"

Hint: When you want to select the entire contents of the field, just do step 1.

V2. Positioning the Insertion Point

1. In a method in the application model, ask the field's wrapper to takeKeyboardFocus.
2. Tell the field's controller the character position at which to place the insertion point.

addPart2

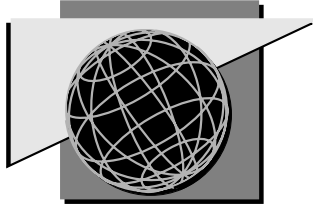
"Put a template in the partID2 field, then position the insertion point."

```
| wrapper |  
self partID2 value: 'MW-'
```

```
wrapper := self builder componentAt: #part2.  
wrapper takeKeyboardFocus.  
wrapper widget controller selectAt: 4.
```

"V2 Step 1"

"V2 Step 2"

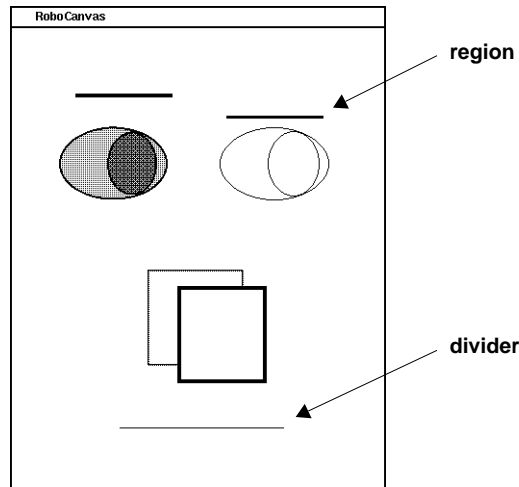


Chapter 7

Lines, Boxes, and Ovals

Separating Widgets with a Line	154
Grouping Widgets with a Box	156
Grouping Widgets with an Ellipse	158

Separating Widgets with a Line



Strategy

Use a divider to provide visual separation between two sets of interface components. It can be either vertical or horizontal.

A divider's thickness is one pixel—for a thicker line, use a region as described below.

Basic Steps

Online example: [LineExample](#)

1. Use a Palette to add a divider to the canvas.
2. Use the widget handles to size and position the divider.

Variants

V1. Adding a Vertical Line

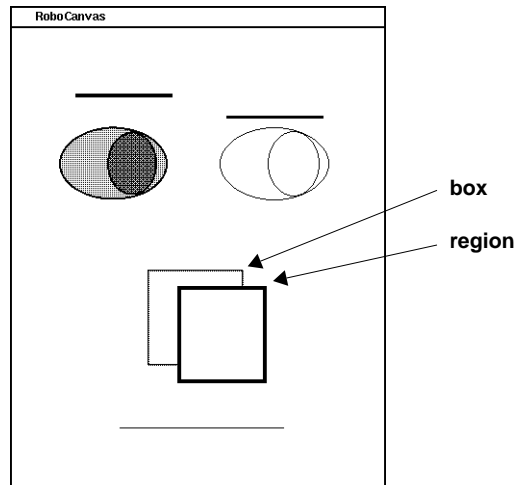
1. Use a Palette to add a divider to the canvas. Leave the divider selected.
2. In the Properties Tool, turn on the divider's Vertical orientation property. Apply the property.
3. Use the widget handles to size and position the divider.

4. Install the canvas.

V2. Simulating a Thicker Line

1. Use a Palette to add a region to the canvas. Leave the region selected.
2. Turn on the Thick property. Apply the property.
3. Use the widget handles to radically elongate the region, merging two opposite sides and making it appear to be a thick line.
4. Install the canvas.

Grouping Widgets with a Box



Strategy

When an interface begins to appear cluttered, the user of your application may have trouble understanding how the widgets relate to one another. As a visual aid, cluster the widgets in logical groups. Spacing is one way to group widgets; another way is to surround some groups with boxes.

A box can have a label embedded in its top border. Its line thickness is one pixel. For a thicker line, use a region as described below.

Basic Steps

Online example: LineExample

1. Use a Palette to add a box to the canvas. Leave the box selected.
2. Use the widget handles to size and position the box.
3. Fill in the Label property, if desired.
4. Choose the label's font.
5. Apply the properties and install the canvas.

Variants

V1. Adding a Box with Thicker Lines

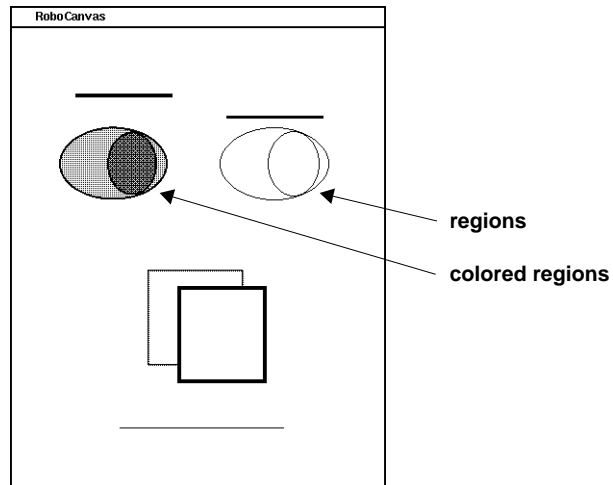
1. Use a Palette to add a region to the canvas. Leave the region selected.
2. Turn on the desired **Border** thickness property.
3. Use the widget handles to size and position the region.
4. If you want the region to have a label, use a Palette to add a label widget. Turn on the label's **opaque** property, fill in its **Label** property, and then position it as desired.
5. Apply the properties and install the canvas.

V2. Changing a Box's Colors

A box widget has no interior surface to color, so use a region when you want a filled box.

- To change the label color of a box, use a Properties Tool to apply foreground color.
- To change only the background color of the label, apply background color.
- To change the border color of a region, apply foreground color.
- To change the interior color of a region, apply background color.

Grouping Widgets with an Ellipse



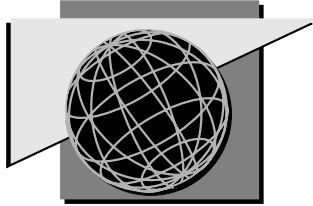
Strategy

For visual variety, you can make a region circular or elliptical in shape rather than rectangular.

Basic Steps

Online example: [LineExample](#)

1. Use a Palette to add a region to the canvas. Leave the region selected.
2. Turn on the region's Ellipse property.
3. Use the widget handles to size and position the region.
4. If desired, use the Properties Tool to apply color to the foreground (border) and/or background (interior).
5. Apply the properties and install the canvas.



Chapter 8

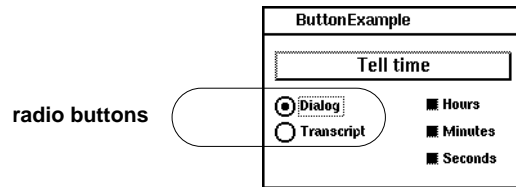
Buttons

Adding a Set of Radio Buttons	160
Adding a Check Box	162
Adding an Action Button	164
Giving a Button a Graphic Label	167
Turning Off Highlighting	168

See Also

- “Widget Basics” on page 53
- “Adding a Menu Button” on page 236

Adding a Set of Radio Buttons



Strategy

A group of radio buttons enables the user of your application to select from a limited list of choices. Selecting a radio button causes any other button in its group to be deselected. This characteristic makes radio buttons useful only where an exclusive selection is appropriate.

Alternatives to radio buttons. Radio buttons display the full set of choices at all times. If you need to save space, you can use a menu or menu button instead.

Radio buttons are typically used only for a very brief and static set of choices. If you want your application to reconfigure the list of choices programmatically, you use a list widget, instead. A list is also scrollable, making it more suitable for a long list of options.

If you want to allow users to select more than one choice, use either a group of check boxes or a list widget that has the Multi Select property turned on.

Basic Steps

Online example: ButtonExample

1. Use the Palette to add one radio button to the canvas for each item in the list of choices.
2. For each button, change the Label property to name the choice (in the example, "Dialog" and "Transcript").
3. For all buttons, enter the same Aspect property (outputMode).
4. For each button, enter a different Select property (#dialog and #transcript). This is the symbol that is stored in the Aspect value holder whenever the button is selected.

5. Apply the properties and install the canvas.
6. Use the canvas's `define` command or a System Browser to add an instance variable for the aspect that is shared by the buttons (`outputMode`).
7. Use the canvas's `define` command or a System Browser to create a method for accessing the aspect variable (`outputMode`), in an `aspects` protocol.
8. Use a System Browser to create an `initialize` method, in which you initialize the aspect variable so it holds a value holder containing one of the valid `Select` symbols (`#dialog`). Your choice of symbol determines which radio button will be selected as a default.

```

outputMode                                     "Basic Step 7"
  ^outputMode

```

```

initialize
  super initialize.
  outputMode := #dialog asValue.               "Basic Step 8"
  showMinutes := true asValue.
  showHours := true asValue.
  showSeconds := true asValue.

```

Variant

Relocating the Label

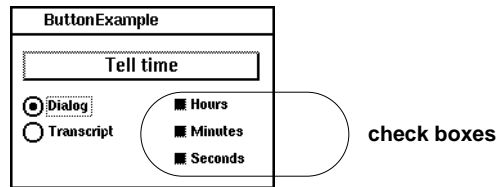
A radio button's built-in label appears to the right of the button under most window managers. To place the label in a different location, such as above the button:

1. Leave the button's `Label` property blank, so nothing appears in the button's default labeling location.
2. Use a separate label widget to label the button.

See Also

- "Adding a List" on page 184
- "Adding a Menu Button" on page 236

Adding a Check Box



Strategy

A check box is like a toggle button that enables the user of your application to turn on or turn off an attribute. Check boxes are often used in a group to represent a set of related attributes.

Selecting one check box has no effect on others in the set, so users can select as many as they want. When you want only one attribute to be selected at a time, use radio buttons instead.

Basic Steps

Online example: ButtonExample

1. Use a Palette to add a check box to the canvas. (The example uses three check boxes to control whether hours, minutes, and/or seconds are displayed.)
2. For each check box, enter a descriptive name in its Label property (for example, "Hours").
3. For each checkbox, fill in its Aspect property with the name of the method that accesses the check box's value holder (showHours). This value holder will contain true when the check box is selected and false when it is not selected.
4. Apply the properties and install the canvas.
5. Use the canvas' define command or a System Browser to create an instance variable in which to store the check box's value holder (showHours).
6. Use the canvas' define command or a System Browser to create the method(s) named in step 3 (showHours) in an aspects protocol.

```
showHours                                     "Basic Step 6"  
  ^showHours
```

7. In the initialize method, initialize the variable to a value holder containing true if you want the check box to be selected by default and false otherwise.

```
initialize  
  super initialize.  
  outputMode := #dialog asValue.  
  showHours := true asValue.                                     "Basic Step 7"  
  showMinutes := true asValue.  
  showSeconds := true asValue.
```

Variant

Relocating the Label

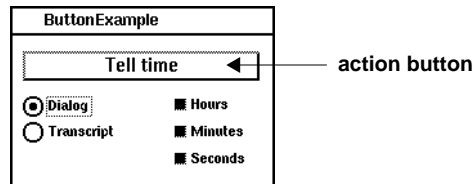
A check box's built-in label appears to the right of the check box. To place the label in a different location, such as above the check box:

1. Leave the check box's Label property blank, so nothing appears in the check box's default labeling location.
2. Use a separate label widget to label the check box.

See Also

- "Displaying an Icon in a Menu" on page 254

Adding an Action Button



Strategy

An action button triggers an action, such as opening a dialog window. If you want to save space in an interface, consider using a menu instead of multiple buttons.

Basic Steps

Online example: ButtonExample

1. Use a Palette to add an action button to the canvas. Leave the button selected.
2. In a Properties Tool, fill in the button's Label property with a descriptive label (in the example, "Tell time").
3. Fill in the button's Action property with the name of the method that performs the action (#tellTime).
4. Apply the properties and install the canvas.
5. Use a System Browser to create the method named in step 3 (tellTime) in an actions protocol.

```

tellTime                                     "Basic Step 5"
| t tString |
t := Time now.
tString := String new.

"Assemble the time string based on the check boxes."
self showHours value
  ifTrue: [tString := tString, t hours printString].
self showMinutes value
  ifTrue: [tString := tString, ':', t minutes printString, ':']
  ifFalse: [tString := tString, '::'].
self showSeconds value

```

```
ifTrue: [tString := tString, t seconds printString].
```

```
"Send the time string to the output channel set by the radio buttons."
```

```
self outputMode value == #transcript
```

```
ifTrue: [Transcript show: tString; cr]
```

```
ifFalse: [DialogView warn: tString]
```

Variants

V1. Using a Placeholder Action

Sometimes it is convenient to add a button to a canvas before you are ready to implement the action method. If you leave the Action property blank, the button will have no effect in the running interface, which can be disconcerting. This variant causes the button to display a dialog reminding you that the method has not yet been implemented.

- In the button's Action property, enter `unimplemented`.

V2. Designating a Default Button

In a canvas that is to be used as a dialog, it is common to enable the user to signify completion either by clicking on a particular button (such as OK or Done) or by pressing <Return> on the keyboard. To arrange for <Return> to activate a particular button (before the focus is shifted manually):

- Turn on the button's Be Default property.

V3. Sizing a Button as If It Were the Default Button

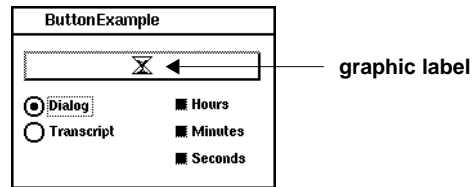
With some window managers, such as Windows and OSF Motif, a default button has a different appearance and the difference may affect the dimensions of the button's border. This can complicate matters when you try to align nondefault buttons with the default button, even after you have equalized their heights and widths. To make a nondefault button take on the sizing characteristics of a default button:

- Turn on the button's Size as Default property.

See Also

- “Adding a Menu Bar” on page 233

Giving a Button a Graphic Label



Strategy

Any of the three kinds of buttons—action buttons, radio buttons, or check boxes—can have a graphic label instead of a text label. In practice, graphic labels are used most often with action buttons, if only because the other two types of button already have a graphic component under most window managers.

Basic Steps

Online example: ButtonExample (You must do steps 1 and 2 first)

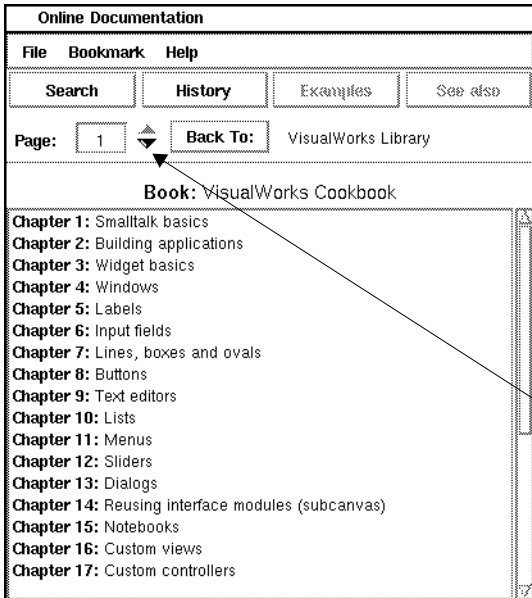
1. Select the button in the canvas.
2. In a Properties Tool, fill in the button's Label property with the name of the method that returns a graphic image (in the example, hourglass).
3. Turn on the button's Label is Image property.
4. Apply the properties and install the canvas.
5. Use an Image Editor or System Browser to create, in a resources protocol, the class method that returns the graphic image (hourglass).

<code>hourglass</code>	"Basic Step 5"
<code>^Cursor wait asOpaqueImage</code>	

See Also

- "Creating a Graphic Image" on page 658

Turning Off Highlighting



Highlighting has been turned off for nonrectangular action buttons

Strategy

By default, a button is highlighted when the user clicks on it. The highlighting is rectangular, like the button's border, even when the border is not displayed and the interior graphic is not rectangular. The basic steps show how to turn off the highlighting in such situations.

Basic Steps

Online example: HelpBrowser

1. Select the button in the canvas. In a Properties Tool, turn off the button's **Bordered** property.
2. In a method in the application model (typically `postBuildWith:`), get the widget from the application model's builder and send a `hiliteSelection:` message to it. The argument is `false`.

```
postBuildWith: aBuilder  
  | oddButtons |
```

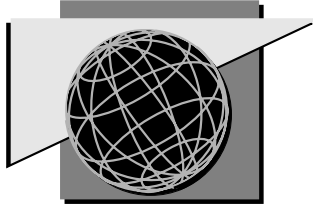
```
"Make the main window a master window."  
aBuilder window  
  application: self;  
  beMaster.
```

```
"Turn off highlighting for the nonrectangular buttons."  
oddButtons := #( #prevPageButton #nextPageButton ).  
oddButtons do: [ :buttonName |  
  (aBuilder componentAt: buttonName)  
    widget hiliteSelection: false].
```

"Basic Step 2"

```
"Disable the appropriate buttons."  
self adjustButtons.
```

```
"Set keyboard hook for special shortcut keys."  
aBuilder keyboardProcessor keyboardHook: [ :ev :ctrl |  
  self keyPress: ev].
```



Chapter 9

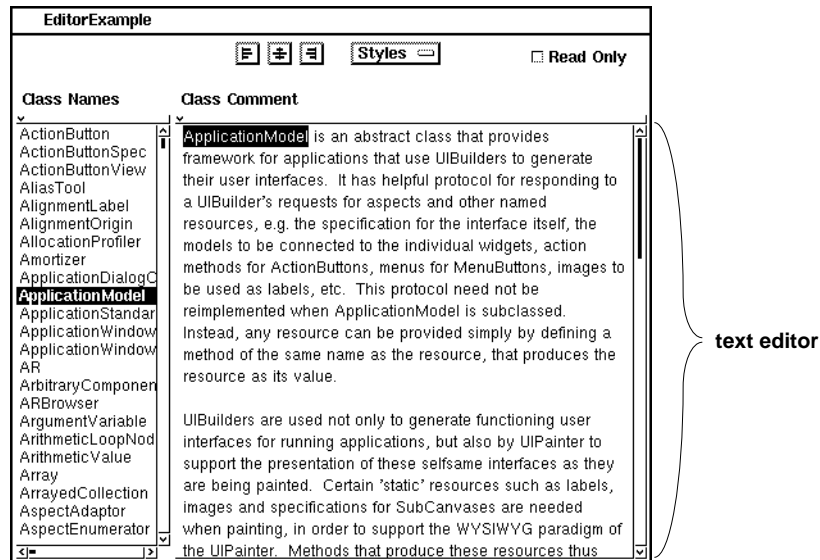
Text Editors

Adding a Text Editor	172
Accessing the Selected Text	174
Highlighting Text Programmatically	176
Aligning Text	178
Making an Editor Read-Only	180
Modifying an Editor's Menu	182

See Also

- “Characters and Strings” on page 529
- “Text and Fonts” on page 555

Adding a Text Editor



Strategy

A text editor is useful for displaying and editing text that does not fit comfortably within a field, especially when the text is expected to have multiple lines. The text editor has built-in facilities for:

- Line wrapping
- Changing the text style
- Cutting, copying, and pasting
- Undoing and reverting
- Searching and replacing
- Printing
- Executing Smalltalk expressions

Basic Steps

Online example: Editor1Example

1. Use a Palette to add a text editor to the canvas. Leave the text editor selected.
2. In a Properties Tool, fill in the editor's Aspect property with the name of the method (comment) that will return the value model for the text editor.
3. Use the define command or a System Browser to add an instance variable (comment) to the application model for storing the text editor's value model.
4. Use a System Browser to add, in an aspects protocol, a method (comment) that returns the contents of the instance variable.

```
comment                                     "Basic Step 4"
  ^comment
```

5. Use a System Browser to create an initialize method that initializes the aspect variable (comment) with a value holder containing the initial text to be displayed (an empty string).

```
initialize
  super initialize.

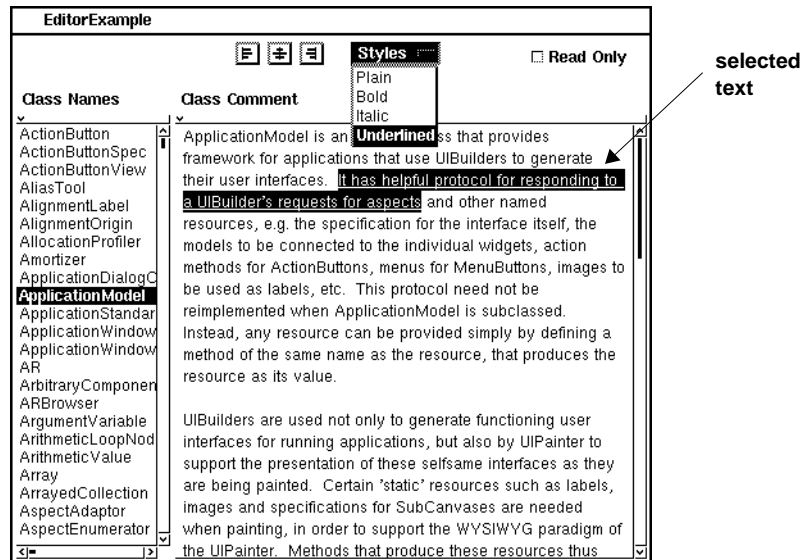
  comment := " asValue.                                     "Basic Step 5"

  classes := SelectionInList with: Smalltalk classNames.
  classes selectionIndexHolder
    onChangeSend: #changedClass to: self.

  textStyle := #plain asValue.
  textStyle onChangeSend: #changedStyle to: self.

  readOnly := false asValue.
  readOnly onChangeSend: #changedReadOnly to: self
```

Accessing the Selected Text



Strategy

When the user highlights a portion of the text in an editor, your application can find out what is highlighted. This is useful when the application needs to use the selected text in some way—as a parameter in a message send, for example.

Sometimes you need to modify the text in some way (in the example, we change the font) and then insert the new version into the main text. The variant shows how to do this.

Basic Steps

Online example: Editor1Example

1. In a method in the application model, get the controller from the widget.
2. Ask the controller for the selected text.

changedStyle

"A text style was selected -- apply it to the current selection in the comment."

```

| c selectedText style |

"Get the selected text."
c := (builder componentAt: #comment) widget controller. "Basic Step 1"
selectedText := c selection. "Basic Step 2"

"If nothing is selected, take no action."
selectedText isEmpty ifTrue: [^self].

"If 'Plain' was selected, remove all emphases;
otherwise add the new emphasis."
style := self textStyle value.
style == #plain
    ifTrue: [selectedText emphasizeAllWith: nil]
    ifFalse: [
        selectedText addEmphasis: (Array with: style)
        removeEmphasis: nil
        allowDuplicates: false].

"Ask the controller to insert the modified text, then update the view."
c replaceSelectionWith: selectedText. "Variant Step 1"
c view resetSelections. "Variant Step 2"
c view invalidate. "Variant Step 3"

```

Variant

Replacing the Selected Text

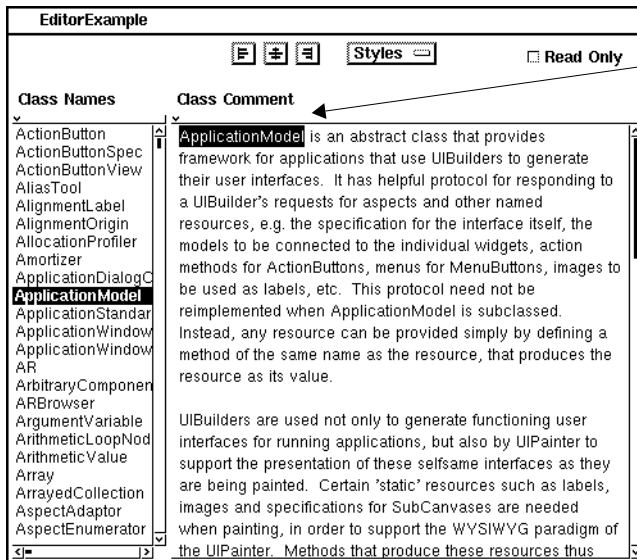
(See example method above.)

1. Ask the controller to replace the selection with a new text.
2. Ask the controller's view to reset its selections (to adjust for a possible width change in the selection).
3. Ask the view to redisplay itself.

See Also

- "Replacing a Range of Text" on page 567

Highlighting Text Programmatically



The name of the class is highlighted programmatically

Strategy

The user of your application can highlight text in an editor by dragging the mouse. Sometimes your application may need to highlight text for the user, perhaps as a way of drawing attention to a keyword or phrase.

Basic Steps

Online example: Editor1Example

1. In a method in the application model, get the controller from the widget.
2. Ask the controller to select the text between two endpoints (and ask it to scroll the selection into view if necessary).
3. Ask the builder's component to take the keyboard focus, so the highlighting will be displayed.

changedClass

"When the list selection changes, update the comment view."

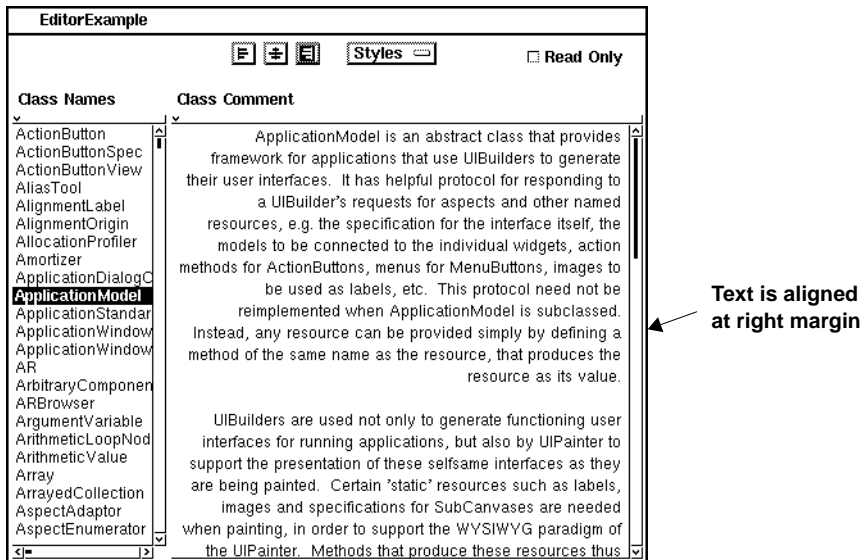
```
| selectedClass txt start wrapper |
selectedClass := self classes selection.

selectedClass isNil
  ifTrue: [self comment value: " asText]
  ifFalse: [
    txt := (Smalltalk at: selectedClass) comment.

    self comment
      value: txt.

    "Find and highlight the class name in the text."
    start := txt
      indexOfSubCollection: selectedClass asString
      startingAt: 1.
    start > 0 ifTrue: [
      wrapper := (self builder componentAt: #comment).
      wrapper widget controller          "Basic Step 1"
        selectAndScrollFrom: start      "Basic Step 2"
          to: start + selectedClass asString size - 1.
      wrapper takeKeyboardFocus]].      "Basic Step 3"
```

Aligning Text



Strategy

By default, text in an editor is aligned at the left margin. For word-processing applications, you may want to center the text or align it at the right margin. You can change the alignment by setting the text editor's **Align** property.

When you want to enable the user of your application to change the alignment, you can provide a button or menu item for doing so. The variant shows how to arrange it.

Limitation: Alignment applies to the entire text—it cannot be applied selectively to a portion of the text.

Basic Step

Online example: Editor1Example

1. Select the text editor in the canvas.
2. In the Properties Tool, set the editor's **Align** property to **Left**, **Center**, or **Right**.
3. Apply the property and install the canvas.

Variant

Changing the Alignment Programmatically

1. In a method in the application model, get the widget from the builder.
2. Get a *copy* of the widget's text style. (Do not modify the widget's text style directly, because that object is shared by many text editors in the system.)
3. Set the alignment of the text style to 0, 1, or 2 (0 is flush left, 1 is flush right, and 2 is centered).
4. Install the new text style in the widget.
5. Ask the widget to redisplay itself.

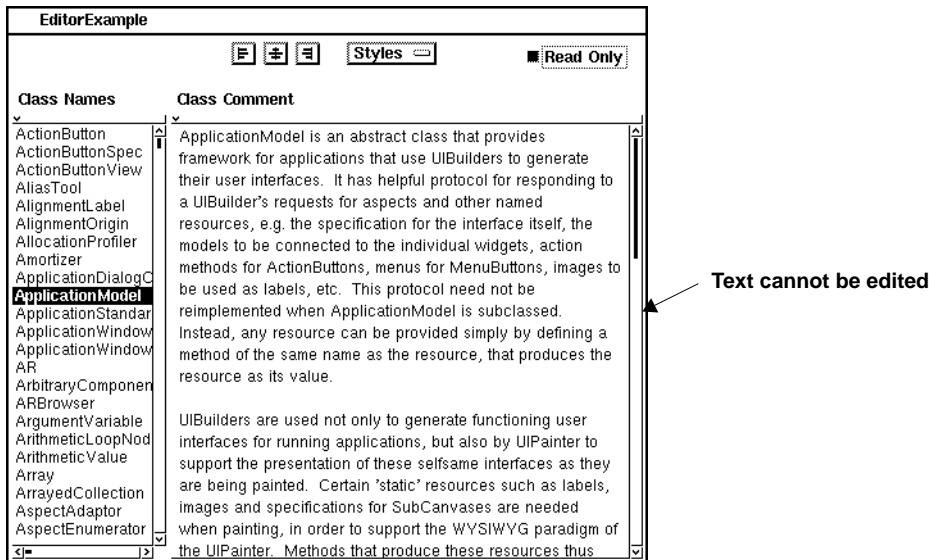
alignRight

```
| widget style |  
widget := (self builder componentAt: #comment) widget.    "Variant Step 1"  
style := widget textStyle copy.                            "Variant Step 2"  
style alignment: 1.                                        "Variant Step 3"  
widget textStyle: style.                                   "Variant Step 4"  
widget invalidate.                                       "Variant Step 5"
```

See Also

- “Controlling Alignment” on page 561

Making an Editor Read-Only



Strategy

By default, a text editor is both an output and an input device. You can turn off the input capability either at canvas-painting time or while the program is running. For example, you might want to disable input based on the user's security level.

Basic Step

1. Select the text editor in the canvas.
2. Turn on the editor's Read Only property.
3. Apply the property and install the canvas.

Variant

Changing the Read-Only Setting Programmatically

1. In a method in the application model, get the controller from the widget.
2. Ask the controller to change its readOnly setting to true or false.

changedReadOnly

| c |

c := (self builder componentAt: #comment) widget controller. "Variant Step 1"

c readOnly: (self readOnly value). "Variant Step 2"

Modifying an Editor's Menu

again	normal menu
undo	
copy	
cut	
paste	
do it	
print it	
inspect	
accept	
cancel	
hardcopy	

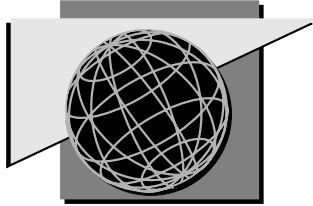
Strategy

By default, a text editor has the same menu of text-editing commands that the system tools have. You can add or remove commands, override the action that is associated with a command, or disable the menu entirely.

Detailed steps for modifying an editor's menu are the same as those for modifying an input field's menu.

See Also

- “Modifying a Field's Pop-Up Menu” on page 139



Chapter 10

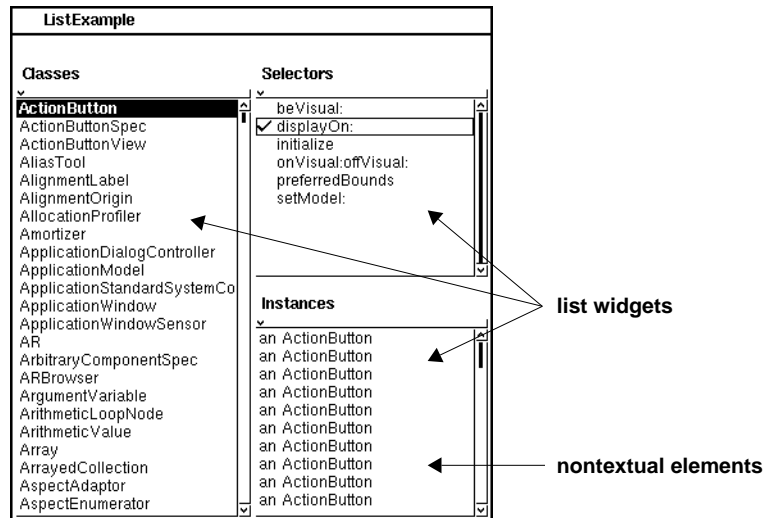
Lists

Adding a List	184
Editing the List of Elements	187
Allowing for Multiple Selections	189
Finding Out What Is Selected	191
Adding a Menu to a List	194
Changing the Highlighting Style	196
Connecting Two Lists	198
Connecting a List to a Text Editor	200

See Also

- “Widget Basics” on page 53

Adding a List



Strategy

A list widget is useful for displaying a collection of objects. As an input device, the list also enables the user to select one or more elements in the list as the targets for browsing and other operations.

A list widget is designed to depend on two value models, unlike most data widgets, which require only one. In particular, a list widget uses two value holders—one to hold the collection of objects to be displayed, and the other to hold the index of the current selection. Consequently, you program the application model to supply a `SelectionInList`, a complex object that contains both of the required value holders.

The elements in the collection need not be textual in nature, provided that they can display themselves textually. The variant shows how to control an object's textual representation in a list.

Basic Steps

Online example: List1Example

1. Use a Palette to add a list widget to the canvas. Leave the list selected.
2. In the Properties Tool, fill in the list's Aspect property with the name of the method that will return an instance of SelectionInList.
3. Use the canvas's define command or a System Browser to add an instance variable (classes) to the application model. This instance variable will hold the SelectionInList.
4. Use the canvas's define command or a System Browser to create the aspect method you named in step 2 (classes).
5. Use a System Browser to initialize the instance variable you created in step 3 (classes), usually in an initialize method. You initialize the variable with an instance of SelectionInList that is itself initialized with a list of Smalltalk class names.

```

classes                                     "Basic Step 4"
  ^classes

```

```

initialize
  super initialize.
  classes := SelectionInList with: Smalltalk classNames.    "Basic Step 5"
  classes selectionIndexHolder onChangeSend: #changedClass to: self.

  methodNames := MultiSelectionInList new.

  instances := SelectionInList new.

```

Variant

Controlling the Textual Display of List Elements

Online example: List1Example

A list sends a `displayString` message to its elements at display time. Every object responds to this message, because it is inherited from the `Object` class. However, the default implementation of `displayString`, which simply sends `printString` to the object, may not

be appropriate for your application. For example, if you use `List1Example` to list the instances of classes such as `ApplicationWindow`, you will find that most of the listings are uninformative because all that is displayed is a generic description such as “an `ApplicationWindow`.”

In contrast, if you use `List1Example` to list the instances of the `Association` class, you will find that the listings are more useful. Underlyingly, each `Association` instance consists of a key paired with a value, which would be too complex to display in a list. Consequently, the `Association` class has reimplemented the `displayString` method, so that each instance represents itself in a list using just the key displayed as a string.

In general, you can equip an object for its role as a list element by providing a reimplement of `displayString` that returns an appropriate string.

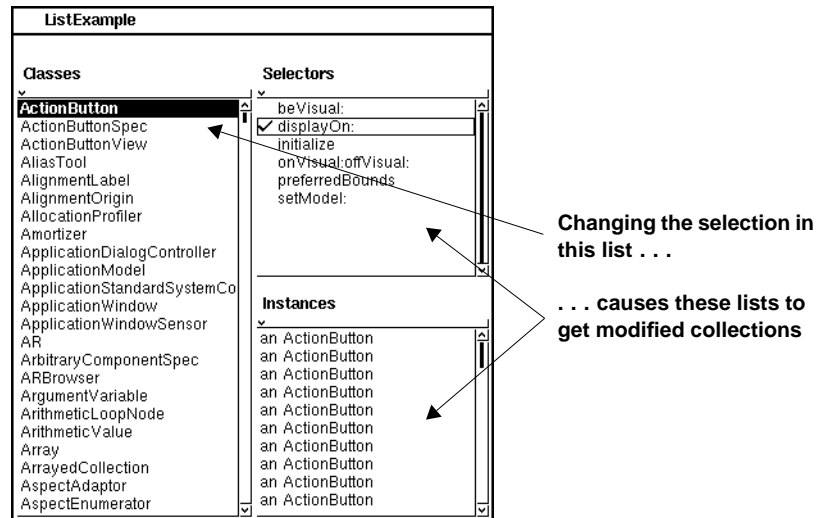
- In the class whose instances are to be displayed in a list, create a printing protocol that contains a `displayString` method that returns a descriptive string. (The following method is implemented in the `Association` class.)

```
displayString                                "Variant Step"  
    "Allows a value to be quietly associated with the key that is  
    displayed in a SequenceView."  
  
    ^key displayString
```

See Also

- “Adding a Notebook” on page 316
- “Creating a Collection” on page 491

Editing the List of Elements



Strategy

The contents of a list often change frequently, depending on other parts of the interface. In List1Example, both the Selectors view and the Instances view change whenever the selection in the Classes view is changed.

Changing the list is accomplished by giving the SelectionInList a new collection of elements. Note that this is not the same as installing an entirely new SelectionInList, which would have the effect of breaking the link with the list widget.

Basic Step

Online example: List1Example

- In the method that is responsible for updating the list, get the SelectionInList from the application model and send a list: message to it, with the new collection as the argument.

```
changedClass
| cls |
self classes selection isNil
```

```
"No class is selected -- empty the selector list."
ifTrue: [
    self methodNames list: List new.           "Basic Step"
    self instances list: List new]

"A class is selected"
ifFalse: [
    cls := Smalltalk at: self classes selection.

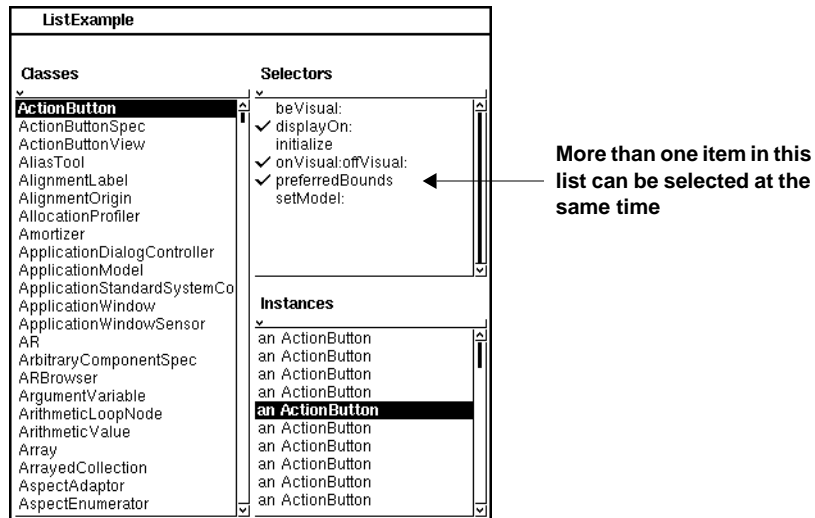
    "Update the selectors list."
    self methodNames list: cls selectors asSortedCollection.

    "Update the instances list."
    self instances list: cls allInstances].
```

See Also

- “Creating a Collection” on page 491

Allowing for Multiple Selections



Strategy

Sometimes it is appropriate for the user to select more than one item in a list as targets for an action. In List1Example, the Selectors list provides this capability so the user can open a Method Browser on several methods.

A list allows multiple selections when its **Multi Select** property is turned on. A second property, **Use Modifier Keys For Multi Select**, determines how selections are to be made. When this property is turned on (the default), the user:

- Clicks the <Select> mouse button to select a single item on the list
- <Shift>-clicks to select additional contiguous items
- <Control>-clicks to select additional noncontiguous items

When the **Use Modifier Keys For Multi Select** property is turned off, the user clicks the <Select> mouse button on each item to be selected. You normally turn this property off only when a multi-select list is to be compatible with other such lists in an older VisualWorks interface.

Basic Steps

Online example: List1Example

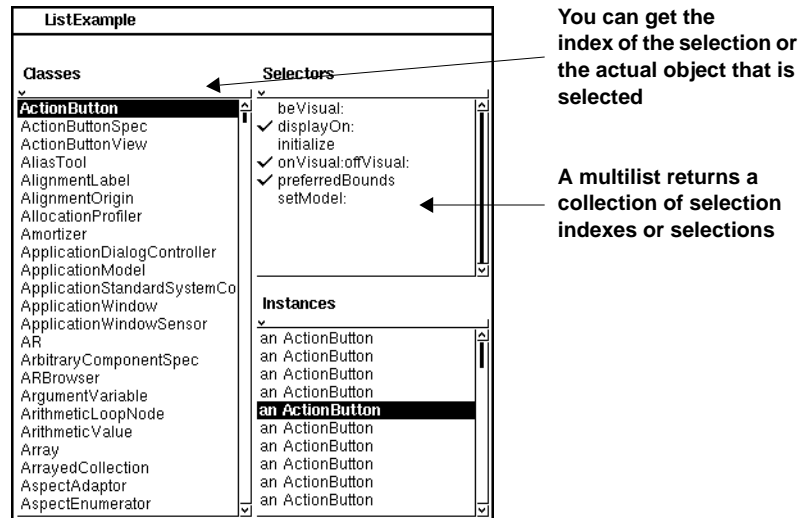
1. Select the list widget in the canvas.
2. In a Properties Tool, turn on the list widget's **Multi Select** property. (Leave the **Use Modifier Keys For Multi Select** property turned on.) Apply properties and install the canvas.
3. In the application model's initialize method, initialize the list widget's aspect variable to hold a **MultiSelectionInList** (instead of a **SelectionInList**).

initialize

```
super initialize.  
classes := SelectionInList with: Smalltalk classNames.  
classes selectionIndexHolder onChangeSend: #changedClass to: self.
```

```
methodNames := MultiSelectionInList new.           "Basic Step 3"  
instances := SelectionInList new.
```

Finding Out What Is Selected



Strategy

When a list widget serves as an input device, your application needs to be able to find out which object is selected. You can ask a `SelectionInList` for the selected object or for the index of the selected object in the list. You can also set the selection programmatically.

For a multiselect list, there may be multiple selections or selection indexes, so your application model must be prepared to handle a collection of objects rather than a single selection or index.

When nothing is selected, a `SelectionInList` returns a `nil` object as the selection and zero as the index; a `MultiSelectionInList` returns an empty collection for either the selections or the indexes.

Basic Step

Online example: List1Example

- In the method that needs to know the current selection in the list, get the `SelectionInList` from the application model and send a selection message to it. (To get the just index, send

selectionIndex. For a MultiSelectionInList, use a selections or selectionIndexes message.)

changedClass

```
| cls |
self classes selection isNil                                "Basic Step"

    "No class is selected -- empty the selector list."
    ifTrue: [
        self methodNames list: List new.
        self instances list: List new]

    "A class is selected"
    ifFalse: [
        cls := Smalltalk at: self classes selection.

        "Update the selectors list."
        self methodNames list: cls selectors asSortedCollection.

        "Update the instances list."
        self instances list: cls allInstances].
```

Variants

V1. Setting the Selection Programmatically

Online example: List1Example

- In the method that is to change the selection programmatically, get the SelectionInList from the application model and send it a selectionIndex: message with the desired index number as the argument.

Alternatively, send a selection: message with the desired object itself as the argument.

postOpenWith: aBuilder

```
super postOpenWith: aBuilder.

    "Uncomment the line below to auto-select the first class."
    self classes selectionIndex: 1.                                "V1 Step"
```

```
"Uncomment the lines below to auto-select the last class."
"self classes selection: self classes list last.
(aBuilder componentAt: #classes) widget controller
  cursorPointWithScrolling."
```

```
"In the classes list, use boxed highlighting instead of reverse-video."
(aBuilder componentAt: #classes) widget strokedSelection."
```

Note that, for a MultiSelectionInList, send selectionIndexes: or selections:, supplying as argument a collection of indexes or a collection of objects in the list.

V2. Selecting All Objects in a Multiple-Selection List

- Get the MultiSelectionInList from the application model and send a selectAll message to it.

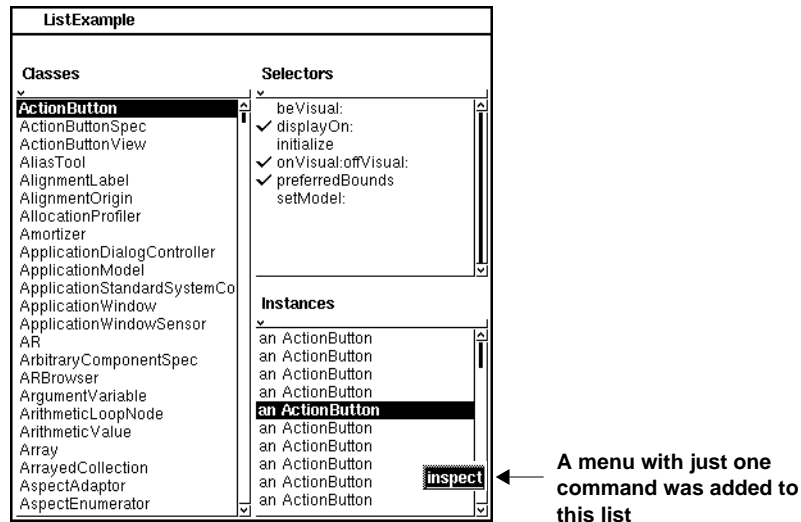
```
selectAll
  self methodNames selectAll.                                     "V2 Step"
```

V3. Clearing All Selections in a Multiple-Selection List

- Get the MultiSelectionInList from the application model and send a clearAll message to it.

```
clearAll
  self methodNames clearAll.                                     "V3 Step"
```

Adding a Menu to a List



Strategy

By default, a list does not provide a pop-up menu, but you can arrange for it to have a custom menu that is available through the <Operate> mouse button. Typically, a list's menu contains two kinds of commands:

- Commands that act on the selection(s).
- Commands that act on the list itself, usually by updating or filtering its contents.

Basic Steps

Online example: List1Example (Instances view)

1. Select the list widget in the canvas.
2. In the Properties Tool, fill in the list widget's Menu property with the name of the method that will supply the menu (instancesMenu).
3. Apply the property and install the canvas.
4. Use a Menu Editor or a System Browser to create the menu method (instancesMenu).

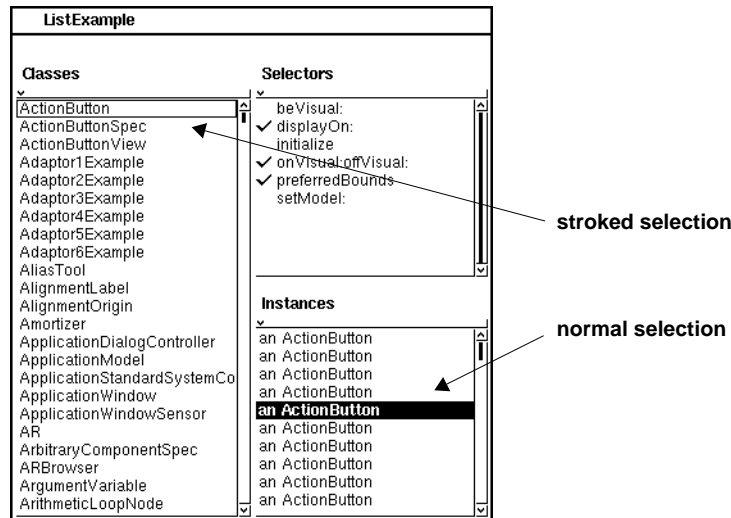
5. Use a System Browser to write the methods that are invoked by the menu (inspectInstance).

inspectInstance	"Basic Step 5"
"Open an inspector on the selected instance."	
inst	
inst := self instances selection.	
inst isNil iffFalse: [inst inspect].	

See Also

- "Creating a Menu" on page 226

Changing the Highlighting Style



Strategy

By default, the selected item in a list is highlighted through reverse video. You can use the list's **Selection Type** property to cause selected items to be indicated by check marks (basic step).

You can also arrange for selected items to be surrounded by a rectangular border (variant).

Basic Step

Online example: List1Example

1. In the canvas, select the list widget whose selection style you want to change. (In the example, this is the **Selectors** list.)
2. In a **Properties Tool**, set the list's **Selection Type** property to **Check Mark**. Apply the property and install the canvas.

Variant

Online example: List1Example

- **In a method in the application model (typically `postBuildWith:` or `postOpenWith:`), get the list widget from the application model's builder and send a `strokedSelection` message to it. (A `normalSelection` message causes the list to revert to normal highlighting.)**

postOpenWith: aBuilder

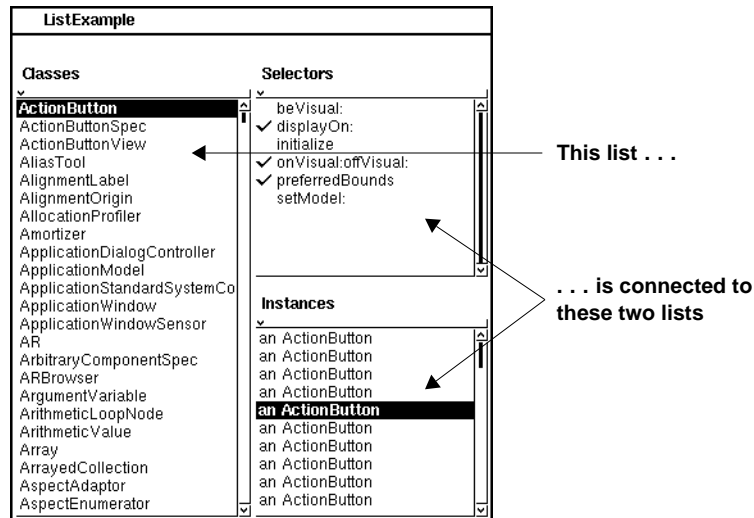
```
super postOpenWith: aBuilder.
```

```
"Uncomment the line below to auto-select the first class."  
self classes selectionIndex: 1.
```

```
"Uncomment the lines below to auto-select the last class."  
"self classes selection: self classes list last."  
(aBuilder componentAt: #classes) widget controller  
  cursorPointWithScrolling."
```

```
"In the classes list, use boxed highlighting instead of reverse-video."  
(aBuilder componentAt: #classes) widget strokedSelection.    "Basic Step"
```

Connecting Two Lists



Strategy

A list widget frequently interacts with another list. For example, in the Resource Finder, selecting a class in the Class list causes the Resource list to display all resources for that class.

Basic Steps

Online example: List1Example (Classes and Selectors lists)

1. In the application model's initialize method, arrange for a change message (changedClass) to be sent to the application model whenever the selection is changed in the first list.

initialize

```

super initialize.
classes := SelectionInList with: Smalltalk classNames.
classes selectionIndexHolder
    onChangeSend: #changedClass to: self.           "Basic Step 1"

methodNames := MultiSelectionInList new.
instances := SelectionInList new.

```

-
2. Use a System Browser to create the change method (changedClass) in the application model. This method tests whether anything is selected in the first list (classes) and then updates the second list (methodNames) appropriately.

```
changedClass                                     "Basic Step 2"
| cls |
self classes selection isNil

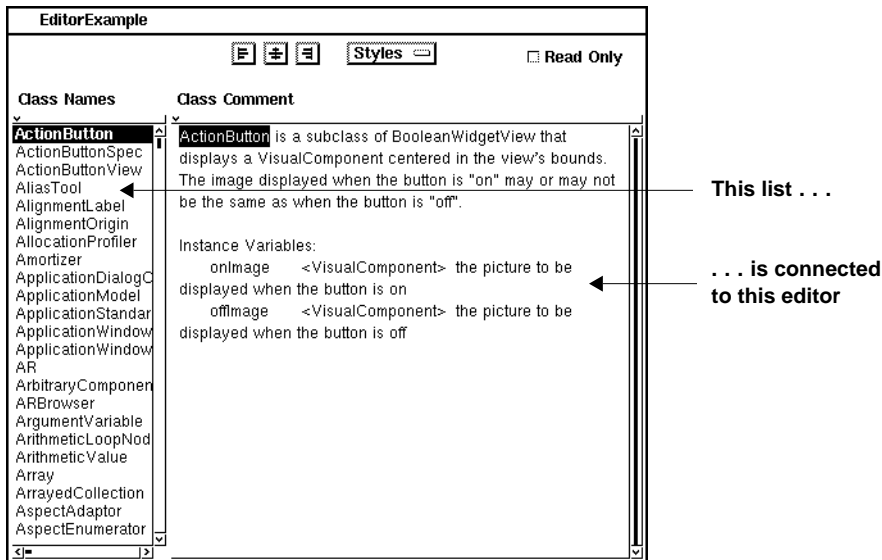
    "No class is selected -- empty the selector list."
    ifTrue: [
        self methodNames list: List new.
        self instances list: List new]

    "A class is selected"
    ifFalse: [
        cls := Smalltalk at: self classes selection.

        "Update the selectors list."
        self methodNames list: cls selectors asSortedCollection.

        "Update the instances list."
        self instances list: cls allInstances].
```

Connecting a List to a Text Editor



Strategy

A list widget often interacts with a text editor. For example, the VisualWorks browsers commonly use a list for choosing a class or method and a text view for displaying information about that class or method.

Basic Steps

Online example: Editor1Example

1. In the application model's initialize method, arrange for a change message (changedClass) to be sent to the application model whenever the list selection is changed.

initialize

```
super initialize.
```

```
comment := " asValue.
```

```
classes := SelectionInList with: Smalltalk classNames.
```

```

classes selectionIndexHolder
  onChangeSend: #changedClass to: self.           "Basic Step 1"

  textStyle := #plain asValue.
  textStyle onChangeSend: #changedStyle to: self.

  readOnly := false asValue.
  readOnly onChangeSend: #changedReadOnly to: self.

```

2. Use a System Browser to create the change method (changedClass) in the application model. This method tests whether anything is selected in the list (classes) and then updates the text editor's value holder (comment) appropriately.
-

```

changedClass                                     "Basic Step 2"
  "When the list selection changes, update the comment view."

  | selectedClass txt start wrapper |
  selectedClass := self classes selection.

  selectedClass isNil
    ifTrue: [self comment value: " asText]
    ifFalse: [
      txt := (Smalltalk at: selectedClass) comment.

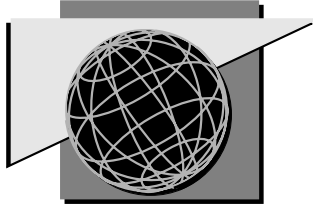
      self comment
        value: txt.

      "Find and highlight the class name in the text."
      start := txt
        indexOfSubCollection: selectedClass asString
        startingAt: 1.
      start > 0
        ifTrue: [
          wrapper := (self builder componentAt: #comment).
          wrapper widget controller
            selectAndScrollFrom: start
              to: start + selectedClass asString size - 1.
          wrapper takeKeyboardFocus]].

```

See Also

- “Adding a Text Editor” on page 172

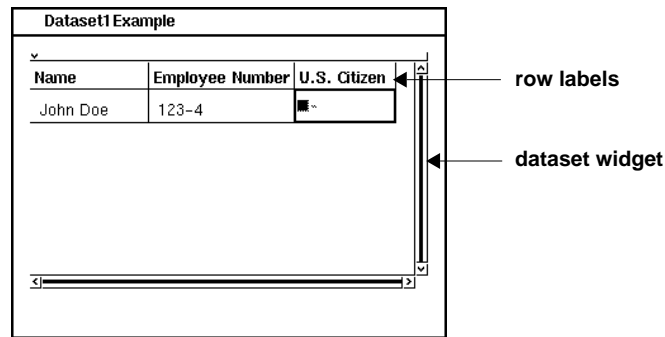


Chapter 11

Datasets

Adding a Dataset	204
Selecting Columns While Painting	209
Adding a Row	210
Connecting Data to a Dataset	212
Enhancing Column Labels	213

Adding a Dataset



Strategy

A dataset presents a list of similar objects for a user to edit. In appearance, datasets are similar to tables in that both kinds of widget presents information in tabular form. However, datasets present data in cells that can be edited directly, whereas tables require that changes be entered indirectly through separate input fields. Furthermore, datasets are best suited for presenting similar kinds of data, whereas a table can present a possibly disparate assortment of data in a collection that allows two-dimensional access.

A dataset uses a `SelectionInList` to hold the list of objects to be displayed, along with information about the current selection. Each object in the list is displayed in its own row, with individual aspects of the object displayed in their own columns. As shown in the basic steps, you use the Properties Tool to specify the means by which each column presents its data—through cells that contain read-only fields, editable fields, combo boxes, or checkboxes.

The variants show you how to resize the dataset's columns and change the order of the columns while painting the canvas.

Basic Steps

Online example: Dataset1Example

In this example, you create a dataset that displays instances of an `Employee` class. An `Employee` consists of three objects (name, `empNo`, and `citizen`), which are to be presented in three dataset columns.

1. Use the Palette to add the dataset widget to the canvas. Leave the dataset selected.
2. In the Properties Tool, fill in the dataset's `Aspect` property with the name of the method (`dsvList`) that will supply an instance of `SelectionInList`. Apply the property.
3. Use the `define` command to add the `dsvList` instance variable to the application model and to create the `dsvList` method in an `aspects` protocol.
The `dsvList` method returns a `SelectionInList` object that will eventually hold the list to be displayed. This method also sets up the `SelectionInList` so it will cause a user's selection to be put in a separate value holder (`selectedRow`).
4. In the dataset's properties, click the `New Column` button for each column you want in the dataset. In this example, click the button three times to add three columns to the canvas.
5. In the canvas, `<Control>`-click in the leftmost column to select it.
6. In the Properties Tool, display the dataset's `Column` property page. The properties you set on this page will apply to the currently selected column.
7. On the `Column` page, enter `Name` as the `Label` property. This creates a visual label above the selected column, which is to display employee names.
8. On the `Column` page, enter `selectedRow name` in the `Aspect` field. `selectedRow` refers to the value holder that will hold the object (the `Employee`) selected by the user. `name` refers to the aspect of `Employee` that is displayed in this column.
9. On the `Column` page, select `Input Field` as the `Type`. This causes each cell in the selected column to display its data in an editable input field. Note that you can optionally specify nondefault characteristics for these input fields by filling in properties on the `Column Type` page.

10. <Control>-click on the middle column to select it.
11. On the **Column** page, enter **Employee Number** as the **Label** and **selectedRow empNo** as the **Aspect**. Select **Input Field** as the **Type**.
12. <Control>-click on the rightmost column to select it.
13. On the **Column** page, enter **U.S. Citizen** as the **Label** and **selectedRow citizen** as the **Aspect**. Select **Check Box** as the **Type**.
14. When the all properties have been applied, install the canvas.
15. Use the **define** command to add the **selectedRow** instance variable to the application model and to create the **selectedRow** method in the **aspects** protocol.

The **selectedRow** method returns a value holder for holding the user-selected **Employee** object from the **SelectionInList**.

16. Use a browser to initialize the dataset (in an **initialize** method in an **initialize-release** protocol).

```

initialize                                     "Basic Step 16"
| aList |
aList := List new.
aList add: Employee new initialize.
self dsvList list: aList.

```

When you open the application, the dataset contains one empty row. You can type a name and number in the **Name** and **Employee Number** columns, and select the **U.S. Citizen** check box.

Note that the first part of the **Aspect** setting for each column must be the same as the message sent by the **SelectionInList** to obtain a value holder for storing the selected object. You used **selectedRow** in steps 8, 11, and 13, because that name is used in the generated **dsvList** method that sets up the **SelectionInList** (step 3). To use a name other than **selectedRow**, you must replace **selectedRow** with the desired name in each **Aspect** field *and* in the code generated for **dsvList** in step 3. Use the **define** command (as in step 15) to generate an instance variable and method with the new name.

Variants

V1. Changing Column Widths

By default, all columns have a width of 80 pixels. You can set specific widths in the dataset's Column properties. You can also change the column widths by editing the dataset in the canvas. For example, to resize the Employee Number column:

1. In the canvas, <Control>-click in the Employee Number column to select it.
2. Place the cursor near the right margin of the column.
3. <Control>-click and hold down the mouse button. If necessary, move the pointer toward the right margin of the selected column until the cursor changes appearance.
4. Drag the cursor to the right to widen the column; drag the cursor to the left to make the column narrower.
5. Install the canvas.

V2. Changing the Column Order

You can switch the order of a dataset's columns by editing it in the canvas. For example, to switch the order of the Employee Number and U.S. Citizen columns:

1. In the canvas, <Control>-click in the Employee Number column to select it.
2. Place the cursor on the drag handle within the selected column.
3. Click on the handle and drag it toward the U.S. Citizen column.
4. Install the canvas.

V3. Disabling Column Scrolling

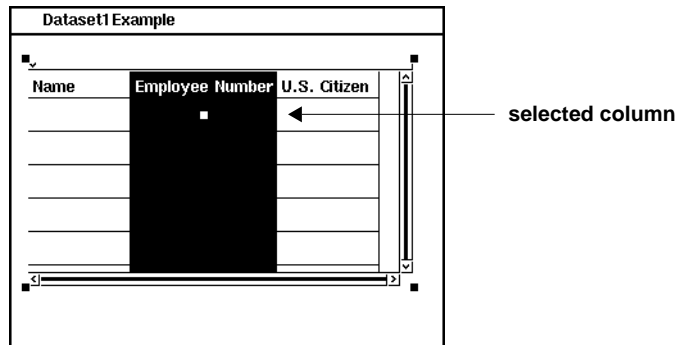
You can set a dataset's columns so that they cannot be scrolled horizontally. This is useful if you want to keep one or more columns displayed on the dataset at all times, while the others continue to scroll.

1. To disable scrolling for a column (and all columns to the left of it), select that column and click the Fixed check box in the Column properties.
2. Apply the property and install the canvas.

See Also

- “Selecting Columns While Painting” on page 209

Selecting Columns While Painting



Strategy

You must select a column before you can set its properties.

Basic Steps

1. Select the dataset on the canvas.
2. Place the cursor inside one of the columns of the dataset.
3. Hold down the <Control> key while clicking the <Select> mouse button.

Variant

V1. Moving the Selection to Another Column

1. Select a column in the dataset using the basic steps.
2. Click the <Select> mouse button for subsequent column selections.

If you then select another widget on the canvas, you must repeat the basic steps to reselect a dataset column.

V2. Scrolling Dataset Columns

You can scroll the columns in the dataset you are painting:

1. Select a column in the dataset.
2. Press <Control> while using the mouse to move the scroll bars on the dataset.

Adding a Row

Dataset2Example			
	Name	Employee Number	U.S. Citizen
	John Doe	123-4	<input checked="" type="checkbox"/>
	Rob Fell	567-8	<input type="checkbox"/>
	Mary Binary	101-1	<input checked="" type="checkbox"/>
▶	Lisa Chou	923-5	<input type="checkbox"/>
Add Row			

row marker

Strategy

When the number of rows needed for a dataset is not predetermined, you can program your application to add rows while it is running.

Basic Steps

Online example: Dataset2Example

1. Use the Palette to add an action button to a canvas containing a dataset. Leave the button selected.
2. In the Properties Tool, enter `Add Row` as the button's Label property and `addRow` as the button's Action property. Apply the properties and install the canvas.
3. Using the `define` command or a System Browser, add the instance method `addRow` in the actions protocol. This method adds a new object to the list displayed by the dataset. This, in turn, adds a new row to the dataset.

```
addRow                                     "Basic Step 3"
(dsvList list) add: Employee new
```

Variant

Adding a Row Marker

A row marker indicates which row is selected within a dataset. It is used in place of row highlighting. To add a row marker, select **Row Selector** on the dataset's **Details** properties. The marker appears as the first column within the dataset.

Connecting Data to a Dataset

Strategy

An initially empty dataset is sufficient if you want users to input the data after the application is open. However, some applications require their datasets to display data initially.

Basic Step

Online example: Dataset3Example

- In the application model, create an initialize method that provides the data for your dataset.

initialize

```
| aList anEmp |  
aList := List new.
```

```
"The aspect for the dataset should be a list of Employee instances.  
Create an employee to put in the list."  
anEmp := Employee new initialize.  
anEmp name: 'Tami Hayes'; empNo: '341-2'; citizen: true.  
aList add: anEmp.
```

```
"Create an employee to put in the list."  
anEmp := Employee new initialize.  
anEmp name: 'Leo Mazon'; empNo: '786-9'; citizen: false.  
aList add: anEmp.
```

```
"Set the list for the dataset aspect. This list appears when you start."  
self dsvList list: aList.  
super initialize.
```

Enhancing Column Labels

Name	Employee Number	U.S. Citizen
Tami Hayes	341-2	<input checked="" type="checkbox"/>
Leo Mazon	786-9	<input type="checkbox"/>

Strategy

When you specify a column label by entering a string in the Label property, it appears on one line. If a column label is particularly long, you can split the label so that it appears on two lines. You do this by providing a text that contains the appropriate carriage returns.

Basic Steps

Online example: Dataset4Example

To split the Employee Number column label:

1. Create a class method (number) in a resources protocol of the application model. This method returns a composed text that is to appear as the label.

```
number                                     "Basic Step 1"
    ^('Employee
    Number' asText allBold) asComposedText
```

2. In the canvas, select the Employee Number column of the dataset.
3. In the Properties Tool, enter number as the Label in the Column properties.

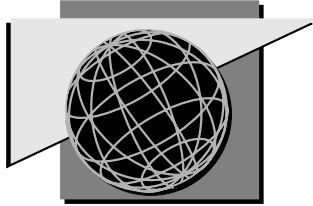
4. Select the Image check box next to Label. This specifies that the column label will come from the resource method named in the Label property.
5. Apply the properties and install the canvas.

Variant

Changing label colors

- To change the color of the column label, follow the basic steps, and then edit the number method to set the desired color.

```
number                                     "Variant Step"  
  ^('Employee  
  Number' asText emphasizeAllWith: (Array  
    with: #bold with: #color->ColorValue red)) asComposedText
```



Chapter 12

Tables

Using TableInterface	216
Adding a Table	217
Connecting a Table to an Input Field	221
Labeling Columns and Rows	223

Using TableInterface

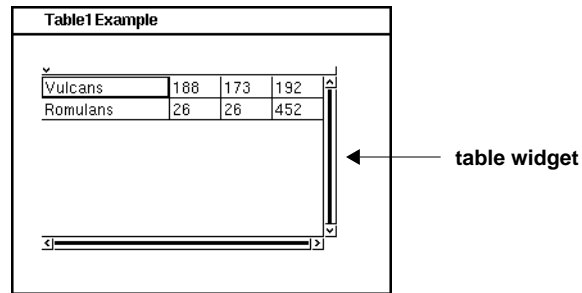
Strategy

Each basic widget, such as a field or label, requires only a simple value model for managing its data, which is usually just a single object such as a text, a number, and so on. In contrast, a table requires a relatively complex auxiliary object. This object, which is an instance of `TableInterface`, holds information about row and column labeling and formatting in addition to the table data itself.

Within a `TableInterface`, the table data is held by a composite object, an instance of `SelectionInTable`. This object holds the collection of cell contents and the selection index. The collection is expected to be a `TwoDList` (two-dimensional list), which converts a flat collection such as an array into a matrix of rows and columns. Alternatively, you can use a `TableAdaptor` to adapt a collection.

All of this interface machinery can be held by a single instance variable in the application model, and you can simply send messages to that object to fetch the table or the selection or any other aspect of it. However, you may find it economical to create instance variables to hold onto various aspects of the table interface. For example, the `SelectionInTable` is useful when your application model will need to access the contents of the table at run time.

Adding a Table



Strategy

A table is familiar to you if you have used a spreadsheet program. It is useful for presenting data that fits into a rows-and-columns structure. In appearance, tables are similar to dataset widgets. However, tables can present dissimilar kinds of data, provided that this data is in a collection that allows two-dimensional access. Furthermore, tables are best suited for presenting data that is unlikely to be edited. In contrast, a dataset is best for presenting a list of similar objects that a user can edit.

By default, a table is bordered and has both vertical and horizontal scroll bars. You can selectively turn off these features in the properties dialog. You can also set the font to be used with text that is displayed in the table cells, connect an <Operate> menu to the table, and turn on vertical and horizontal grid lines to separate rows and columns.

A table needs a special kind of container in which to store its collection of cells. Typically, it keeps the container in an instance variable of the application model. The first step in connecting the table to a model is to identify the method that the table must use to get the container from the model. To identify that method, enter its name in the Aspect field of the properties dialog. Then install the canvas.

In broad terms, you must create at least the following framework in the application model. Note that the canvas' define command creates an Aspect variable and accessing method.

- Add an instance variable for storing descriptive information about the table (a `TableInterface`) and, optionally, a second variable for storing the table's contents (a `SelectionInTable`).
- Create an initialize method in which the instance variables are initialized.
- Create the `Aspect` method, which simply returns the object held by the table-interface variable.

Basic Steps

`Table1Example` creates a table of UFO sightings for the past three years. This table will have a separate row for each type of spacecraft.

Online example: `Table1Example`

1. Use a Palette to add a table widget to the canvas. Leave the table selected.
2. In the Properties Tool, enter `tableInterface` as the `Aspect`. Turn on both horizontal and vertical grid lines. Apply the properties and install the canvas.
3. Use the canvas' `define` command or a System Browser to add the instance variables `sightingsTable` and `tableInterface`.
4. Use the canvas' `define` command or a System Browser to create the instance methods `sightingsTable` and `tableInterface` in an accessing protocol.

<code>sightingsTable</code>	"Basic Step 4"
<code>^sightingsTable</code>	

<code>tableInterface</code>	"Basic Step 4"
<code>^tableInterface</code>	

5. Using a System Browser, initialize the `SelectionInTable`, usually in an initialize method in the application model (initialize-release protocol).

<code>initialize</code>	"Basic Step 5"
list	
super initialize.	
"Create a collection of sightings data."	


```
list := TwoDList
  on: #("Vulcans' 188 173 192 'Romulans' 26 26 452) copy
  columns: 4
  rows: 2.
sightingsTable := SelectionInTable with: list.
"Create a table interface and load it with the sightings."
tableInterface := TableInterface new
  selectionInTable: sightingsTable.
```

Open the table interface. Although the data in a table cannot be edited directly, the next topic will describe how to use an input field to edit the highlighted cell.

Normally, of course, you wouldn't initialize the table with a hard-coded collection—the table data would be gathered from a database or some other source.

Variants

V1. Controlling Column Widths

By default, all columns have an equal width that is determined by the space available in the table. If the table expands with the window, the column widths will also expand. To set specific widths for the columns, send a `columnWidths:` message to the table interface. The argument is an array containing one number for each column. The number is the width in pixels. Any column for which no width is specified gets the width of the last entry in the array.

- Reset the widths at any time by adding to the `initialize` method.

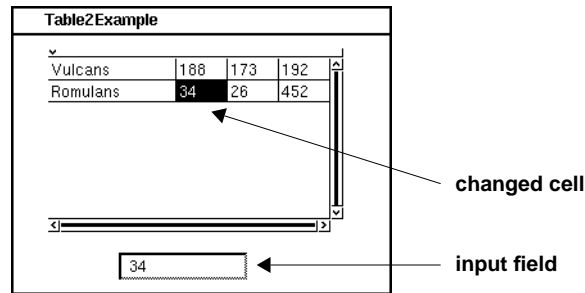
```
tableInterface columnWidths: #(100 40). "V1 Step"
```

In the above example, the first column has been changed so that it is wide enough to show the entire name of the alien race. These widths will remain in effect even if the window is expanded.

V2. Selecting by Row or Column

By default, a single cell in the table is highlighted when the user clicks in it. In some applications, it is more appropriate to highlight the entire row or column in which the cell is located. To arrange for this, simply turn on row or column Selection in the table's properties.

Connecting a Table to an Input Field



Strategy

A read-only table is sufficient for some applications, but in many situations the user needs a way to change the contents of a cell in the table. This can be arranged indirectly by placing an input field near the table and connecting it to the highlighted cell. This technique relies on a single cell being selected—although it still works when row or column selection is enabled, the effect is not very intuitive.

Basic Steps

Online example: Table2Example

1. In the canvas, add an input field below the table. Leave the field selected.
2. In a Properties Tool, enter `cellContents` as the field's Aspect property.
3. Use the canvas' `define` command or a System Browser to add an instance variable named `cellContents` to the `UFOtable` class.
4. Use the canvas' `define` command or a System Browser to create the instance method named in step 3 (`cellContents`) in an aspects protocol.

<code>cellContents</code>	"Basic Step 4"
<code>^cellContents</code>	

5. Add the instance method `changedCell` in a change messages protocol.

```

changedCell                                     "Basic Step 5"
| cellLocation |
"Get the coordinates of the highlighted cell."
cellLocation := self sightingsTable selectionIndex.
"If a cell is selected, update its contents from the input field."
cellLocation = Point zero
  ifFalse: [self sightingsTable table
            at: cellLocation
            put: self cellContents value]

```

6. In the application model's initialize method, initialize the input field (cellContents).

```

initialize
| list |
super initialize.
"Create a collection of sightings data."
list := TwoDList
  on: #('Vulcans' 188 173 192 'Romulans' 26 26 452) copy
  columns: 4
  rows: 2.
sightingsTable := SelectionInTable with: list.
"Create a table interface and load it with the sightings."
tableInterface := TableInterface new
  selectionInTable: sightingsTable.

cellContents := String new asValue.                                     "Basic Step 6"
self cellContents onChangeSend: #changedCell to: self.

```

When you run the application, you can use the input field to edit a selected cell. Notice that when you select a new cell, its contents are not shown in the input field. To make the field update its contents when the table selection changes, you must register interest in the table selection with `onChangeSend:` and trigger an update in the input field. In effect, the table selection and the input field would be watching each other for updates.

Labeling Columns and Rows

	Visiting Race	1992	1993	1994
1	Vulcans	188	173	192
2	Romulans	26	26	452

Labels: column labels (1992, 1993, 1994), row labels (1, 2)

Strategy

You can label one or more columns by sending an array of labels to the table interface. For row labels, you need to send an array of labels and also an indication of the width of those labels.

Basic Step

Online example: Table3Example

- Add code to the end of Table2Example's initialize method that initializes the row and column labels.

```
tableInterface                                     "Basic Step"
  columnLabelsArray: #('Visiting Race' '1992' '1993' '1994');
  rowLabelsArray: #(1 2);
  rowLabelsWidth: 20.
```

Variant

Aligning Data and Labels

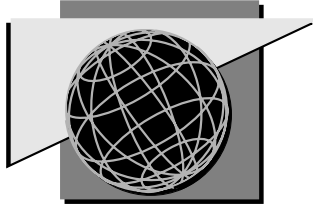
By default, all cells display their contents beginning at the left margin, and all labels are centered. You can align data and labels using any of three symbols: #left, #right, #centered, or

`#leftWrapped`. Using these symbols, you can control the alignment of a column's data, a column's labels, or a row's labels.

- Add code to the `initialize` method that initializes the label alignments.

```
tableInterface                                     "Variant Step"
  columnFormats: #(#left #right #right #right);
  columnLabelsFormats: #(#left #right #right #right);
  rowLabelsFormat: #right.
```

As the example shows, you can set row labels to the same alignment by passing a single symbol as argument, and the same applies to the column alignments. For column data and labels, however, you also have the option of setting each column's alignment individually, as we have done, by passing an array of symbols.



Chapter 13

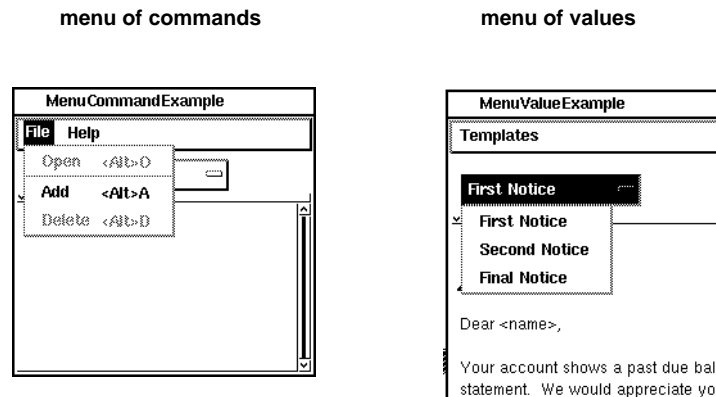
Menus

Creating a Menu	226
Creating a Submenu	231
Adding a Menu Bar	233
Adding a Menu Button	236
Adding a Pop-Up Menu	240
Modifying a Menu Dynamically	243
Disabling a Menu Item	248
Adding a Divider to a Menu	250
Adding a Shortcut Key	252
Displaying an Icon in a Menu	254
Changing Menu Colors	257
Using a Menu Editor	259

See Also

- “Widget Basics” on page 53

Creating a Menu



Strategy

Menu bars, menu buttons, and pop-up menus all rely on an underlying instance of `Menu`. The Menu Editor provides a convenient means of creating a menu and generating a resource method for recreating the menu on demand. Alternatively, you can assemble a menu programmatically, which is useful when the items in the menu must change depending on current conditions in the application. This chapter primarily shows how to assemble menus programmatically; the last topic provides an introduction to using the Menu Editor.

In this topic, the basic steps show how to create a menu of commands in which each command label is paired with the name of an action method that is to be sent to the application model.

The first variant shows how to create a menu of values, which inserts a value in a value holder rather than executing a command. Such menus are commonly used by menu buttons. The second variant shows an alternative menu in which each item uses a block to perform an action rather than a method.

A menu typically is created by a method in a `resources` protocol on the class side of an application model. The resource method can also be an instance method, which is useful when it relies on data supplied by a running application.

Basic Steps

Creating a Menu of Commands

This type of menu is typically used with a menu bar or popup menu. In such menus, selecting a menu item causes the associated symbol to be sent as a message to the application model.

Online example: MenuCommandExample

1. In a resource method that is responsible for creating the menu (in the example, fileMenu), create a MenuBuilder by sending a new message to that class.
2. For each item in the menu, send an add: message to the menu builder. The argument is an association in which the label string is paired with the name of an action method defined in the application model.
3. Get the menu by sending a menu message to the menu builder. Return that menu as the result of the method.

```
fileMenu
| mb menu submenu |
mb := MenuBuilder new.                                "Basic Step 1"

mb
beginSubMenuLabeled: 'File';
add: 'Open' -> #openFile;                             "Basic Step 2"
line;
add: 'Add' -> #addFile;
add: 'Delete' -> #deleteFile;
endSubMenu.

mb
beginSubMenuLabeled: 'Help';
add: 'Usage' -> #explainUsage;
endSubMenu.

"Add shortcut keys."
menu := mb menu.                                     "Basic Step 3"
submenu := (menu menuItemLabeled: 'File') submenu.
(submenu menuItemLabeled: 'Open') shortcutKeyCharacter: $O.
(submenu menuItemLabeled: 'Add') shortcutKeyCharacter: $A.
```

```
(submenu menuItemLabeled: 'Delete') shortcutKeyCharacter: $D.
```

```
^menu "Basic Step 3"
```

Variants

V1. Creating a Menu of Values

This type of menu is typically used with a menu button. In such menus, selecting a menu item causes the associated value to be sent to a value holder (see “Adding a Menu Button” in this chapter).

Online example: MenuValueExample

1. In a menu-creating instance method (in this example, `templatesMenuForMenuButton`), create a `MenuBuilder` by sending a new message to that class.
2. For each menu item, send an `add:` message to the menu builder. The argument is an association between the item’s label string and the value to be sent to a value holder. In this case, the value is a textual template that the menu button puts into its value holder.
3. Get the menu from the menu builder by sending a `menu` message to the menu builder, and return the menu as the result of the method.

`templatesMenuForMenuButton`

```
| mb |
mb := MenuBuilder new. "V1 Step 1"

mb
  add: 'First Notice' -> self class firstNotice; "V1 Step 2"
  add: 'Second Notice' -> self class secondNotice;
  add: 'Final Notice' -> self class finalNotice.

^mb menu "V1 Step 3"
```

V2. Creating a Menu of Action Blocks

You can use a menu of action blocks to provide a menu of values for a menu bar or pop-up menu. Each block causes the menu item to put a particular value in a value holder in response to the user's selection.

Online example: MenuValueExample

1. In a menu-creating instance method (`templatesMenuForMenuBar`), create a menu builder by sending a new message to the `MenuBuilder` class. An instance method is used here because information is needed from the application model instance.
2. For each menu item, send an `add:` message to the menu builder. The argument is an association between the item's label string and the block that is to perform the desired action. In this case, the block inserts a textual template in the value holder for a text editor.
3. Get the menu from the menu builder using a `menu` message and return it as the result of the menu-creating method.

```

templatesMenuForMenuBar
| mb menu submenu |
mb := MenuBuilder new.                                     "V2 Step 1"

mb
  beginSubMenuLabeled: 'Templates';
  add: ' ' -> [self letter value: self class firstNotice];   "V2 Step 2"
  add: ' ' -> [self letter value: self class secondNotice];
  add: ' ' -> [self letter value: self class finalNotice];
  endSubMenu.

"Add graphic labels."
menu := mb menu.                                          "V2 Step 3"
submenu := (menu menuItemLabeled: 'Templates') submenu.
(submenu menuItemAt: 1)
  labelImage: (self class oneImage).
(submenu menuItemAt: 2)
  labelImage: (self class twoImage).
(submenu menuItemAt: 3)
  labelImage: (self class threeImage).

"Set the background color."

```

submenu backgroundColor: ColorValue chartreuse.

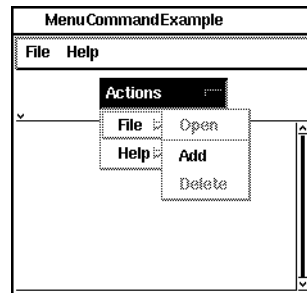
^menu

"V2 Step 3"

See Also

- “Adding a Menu Bar” on page 233
- “Adding a Menu Button” on page 236
- “Adding a Pop-Up Menu” on page 240
- “Using a Menu Editor” on page 259

Creating a Submenu



Strategy

A menu can be nested inside another menu, as shown in the basic steps. This nesting can be repeated, creating a hierarchical menu structure. Nesting beyond a second level begins to decrease the usability of your application, however.

Submenus are used in the construction of menu bars, which use a submenu for the contents of each menu bar item.

Basic Steps

Online example: MenuCommandExample

1. Send a `beginSubMenuLabeled:` message to the menu builder that you have created to assemble the menu. The argument is the label for the submenu, which appears in the parent menu.
2. For each submenu item, send an `add:` message to the menu builder. The argument is an association in which a label string is paired with a method name, value, or block.
3. Send an `endSubMenu` message to the menu builder.

fileMenu

```
| mb menu submenu |
mb := MenuBuilder new.
```

```
mb
```

```
beginSubMenuLabeled: 'File';
add: 'Open' -> #openFile;
```

```
"Basic Step 1"
```

```
"Basic Step 2"
```

```
line;  
add: 'Add' -> #addFile;  
add: 'Delete' -> #deleteFile;  
endSubMenu.
```

"Basic Step 3"

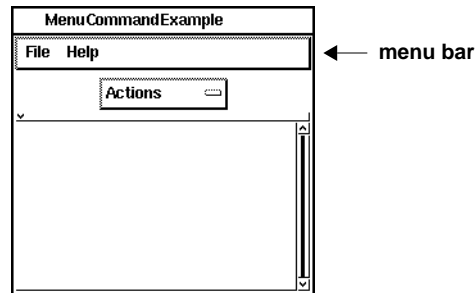
```
mb  
beginSubMenuLabeled: 'Help';  
add: 'Usage' -> #explainUsage;  
endSubMenu.
```

"Add shortcut keys."

```
menu := mb menu.  
submenu := (menu menuItemLabeled: 'File') submenu.  
(submenu menuItemLabeled: 'Open') shortcutKeyCharacter: $O.  
(submenu menuItemLabeled: 'Add') shortcutKeyCharacter: $A.  
(submenu menuItemLabeled: 'Delete') shortcutKeyCharacter: $D.
```

```
^menu
```

Adding a Menu Bar



Strategy

A menu bar appears to the user as a set of separate menus across the top edge of a window. The items in these menus typically give the user access to the commands that are available in the application window.

A menu bar is actually implemented with a single menu object. The menu labels displayed across the menu bar are top-level menu items in the menu object. The contents of the apparently separate menus are actually submenus associated with the top-level menu items.

A menu bar is normally a menu of commands, although you can implement it to behave like a menu of values. As a menu of commands (basic steps), the menu bar responds to the selection of a menu item by sending the associated symbol as a message to the application model. To implement a menu of values (variant), you program each menu item to put a value into a value holder.

Basic Steps

Online example: MenuCommandExample

1. In the canvas for the window, make sure no widget is selected.
2. In a Properties Tool, turn on the Enable switch for the Menu Bar property.

3. In the **Menu** field, enter the name of the menu-creation method (in the example, `fileMenu`).
4. Create the resource method that you named in step 3 (`fileMenu`). This method must return an instance of `Menu` in which each top-level label is associated with a submenu.

```
fileMenu                                     "Basic Step 4"
| mb menu submenu |
mb := MenuBuilder new.

mb
  beginSubMenuLabeled: 'File';
  add: 'Open' -> #openFile;
  line;
  add: 'Add' -> #addFile;
  add: 'Delete' -> #deleteFile;
  endSubMenu.

mb
  beginSubMenuLabeled: 'Help';
  add: 'Usage' -> #explainUsage;
  endSubMenu.

"Add shortcut keys."
menu := mb menu.
submenu := (menu menuItemLabeled: 'File') submenu.
(submenu menuItemLabeled: 'Open') shortcutKeyCharacter: $O.
(submenu menuItemLabeled: 'Add') shortcutKeyCharacter: $A.
(submenu menuItemLabeled: 'Delete') shortcutKeyCharacter: $D.

^menu
```

5. Create the action methods that are invoked by the menu items.

Variant

Creating a Menu Bar That Inserts a Value

Online example: MenuValueExample

- In the method that is responsible for creating the menu (`templatesMenuForMenuBar`), send an `add:` message to the menu builder for each item in the menu. The argument is an association in which a label string is paired with a block. The block is responsible for inserting the desired value in the desired value holder.

`templatesMenuForMenuBar`

```

| mb menu submenu |
mb := MenuBuilder new.

mb
  beginSubMenuLabeled: 'Templates';
  add: ' ' -> [self letter value: self class firstNotice];           "Variant Step"
  add: ' ' -> [self letter value: self class secondNotice];
  add: ' ' -> [self letter value: self class finalNotice];
  endSubMenu.

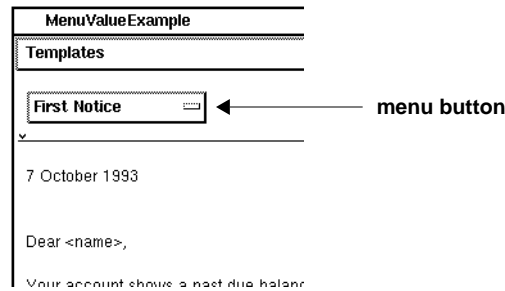
"Add graphic labels."
menu := mb menu.
submenu := (menu menuItemLabeled: 'Templates') submenu.
(submenu menuItemAt: 1)
  labelImage: (self class oneImage).
(submenu menuItemAt: 2)
  labelImage: (self class twoImage).
(submenu menuItemAt: 3)
  labelImage: (self class threeImage).

"Set the background color."
submenu backgroundColor: ColorValue chartreuse.

^menu

```

Adding a Menu Button



Strategy

A menu button is a visual representation of a set of menu items. It is similar to a submenu in a menu bar, but with two advantages: it can be placed anywhere in the canvas, and its label can change to reflect the current selection. A menu button is more visible to the user than a pop-up menu, but it uses space in the canvas.

A menu button can present either a menu of values (first variant) or a menu of commands (second variant). In either case, the menu button sends a value to a value model (usually a value holder) in response to a user's selection. For a menu of values, you program the application model to process the values as desired. For a menu of commands, you program the application model to treat these values as messages that invoke action methods.

Variants

V1. Adding a Menu Button with a Menu of Values

Online example: MenuValueExample

1. Use the Palette to add a menu button widget to the canvas.
2. In the button's Label property, enter the desired label, which will appear on the menu button. Leaving the Label blank causes the current selection to appear on the menu button while the application is running.

3. In the button's **Aspect** property, enter the name of the method that returns the value holder (in the example, letter).
4. In the menu button's **Menu** property, enter the name of the resource method that returns a menu of values (templatesMenuForMenuButton).
5. Use the **define** command or a **System Browser** to add an **instance variable** (letter) to the application model (MenuValueExample).
6. Use a **System Browser** to add a method (letter), in an aspects protocol, that returns the contents of the instance variable.

```
letter                                     "V1 Step 6"
  ^letter
```

7. Use a **System Browser** to create an **initialize** method, in an initialize-release protocol, that initializes the aspect variable .

```
initialize                                 "V1 Step 7"
  letter := self class firstNotice asValue.
  letter onChangeSend: #setCheckMark to: self.
```

8. Use a **Menu Editor** or a **System Browser** to create a **resource method** (templatesMenuForMenuButton) that creates a menu of values.

```
templatesMenuForMenuButton                "V1 Step 8"
  | mb |
  mb := MenuBuilder new.

  mb
    add: 'First Notice' -> self class firstNotice;
    add: 'Second Notice' -> self class secondNotice;
    add: 'Final Notice' -> self class finalNotice.

  ^mb menu
```

V2. Adding a Menu Button with a Menu of Commands

Online example: MenuCommandExample

1. Use the Palette to add a menu-button widget to the canvas.
2. In the button's Label property, enter the desired label, which will appear on the menu button. Leaving the Label blank causes the current selection to appear on the menu button while the application is running.
3. In the button's Aspect property, enter the name of the method that returns a value holder (in the example, action).
4. In the menu button's Menu property, enter the name of the resource method that returns a menu of commands (templatesMenuForMenuButton).
5. Use the **define** command or a System Browser to add an instance variable (action) to the application model (MenuCommandExample).
6. Use a System Browser to add a method (action), in an aspects protocol, that returns the contents of the instance variable .

```
action                                     "V2 Step 6"
  ^action
```

7. Use a System Browser to create an initialize method, in an initialize-release protocol, that initializes the aspect variable. Also in the initialize method, send an onChangeSend:to: message to the value holder (action); the first argument is the name of a method that performs the command (performAction), and the second argument is typically the application model.

```
initialize                                 "V2 Step 7"
  files := SelectionInList new.
  files selectionIndexHolder onChangeSend: #configureMenu to: self.

  action := nil asValue.
  action onChangeSend: #performAction to: self.
```

8. Use a System Browser to add a method that performs the currently selected action (performAction).

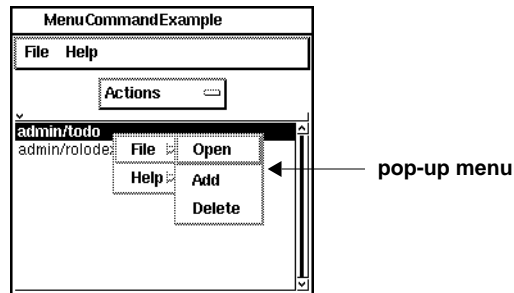
```
performAction "V2 Step 8"  
    self perform: self action value.
```

9. Use a Menu Editor or a System Browser to create a resource method (fileMenu) that creates a menu of commands.

```
fileMenu "V2 Step 9"  
    | mb menu submenu |  
    mb := MenuBuilder new.  
  
    mb  
        beginSubMenuLabeled: 'File';  
        add: 'Open' -> #openFile;  
        line;  
        add: 'Add' -> #addFile;  
        add: 'Delete' -> #deleteFile;  
        endSubMenu.  
  
    mb  
        beginSubMenuLabeled: 'Help';  
        add: 'Usage' -> #explainUsage;  
        endSubMenu.  
  
    "Add shortcut keys."  
    menu := mb menu.  
    submenu := (menu menuItemLabeled: 'File') submenu.  
    (submenu menuItemLabeled: 'Open') shortcutKeyCharacter: $O.  
    (submenu menuItemLabeled: 'Add') shortcutKeyCharacter: $A.  
    (submenu menuItemLabeled: 'Delete') shortcutKeyCharacter: $D.  
  
    ^menu
```

10. Use a System Browser to create each of the action methods.

Adding a Pop-Up Menu



Strategy

Several widgets, notably lists and text editors, provide a pop-up menu in response to the <Operate> mouse button.

The underlying menu is typically a menu of commands, although you can implement it to behave like a menu of values. As a menu of commands (basic steps), the pop-up menu responds to the selection of a menu item by sending the associated symbol as a message to the application model. To implement a menu of values (variant), you program each menu item to put a value into a value holder.

Basic Steps

Online example: MenuCommandExample

1. In the **Menu** property of the widget, enter the name of the method that returns a menu of commands (in the example, fileMenu).
2. Use a Menu Editor or a System Browser to create a method that returns a menu such as fileMenu.

```
fileMenu "Basic Step 2"
| mb menu submenu |
mb := MenuBuilder new.

mb
beginSubMenuItemLabeled: 'File';
add: 'Open' -> #openFile;
```

```

line;
add: 'Add' -> #addFile;
add: 'Delete' -> #deleteFile;
endSubMenu.

mb
beginSubMenuLabeled: 'Help';
add: 'Usage' -> #explainUsage;
endSubMenu.

"Add shortcut keys."
menu := mb menu.
submenu := (menu menuItemLabeled: 'File') submenu.
(submenu menuItemLabeled: 'Open') shortcutKeyCharacter: $O.
(submenu menuItemLabeled: 'Add') shortcutKeyCharacter: $A.
(submenu menuItemLabeled: 'Delete') shortcutKeyCharacter: $D.

^menu

```

3. Use a System Browser to create each of the action methods named in the menu of commands.

Variant

Adding a Pop-Up Menu of Values

Online example: MenuValueExample

1. In the Menu property of the widget, enter the name of the method that returns a menu of values (templatesMenuForPopUp).
2. Use a Menu Editor or System Browser to create the menu method (templatesMenuForPopUp). In the menu, each item label is paired with a block in which the widget's aspect variable (letter) is updated with the desired value.

```

templatesMenuForPopUp                                     "Variant Step 2"
| mb |
mb := MenuBuilder new.

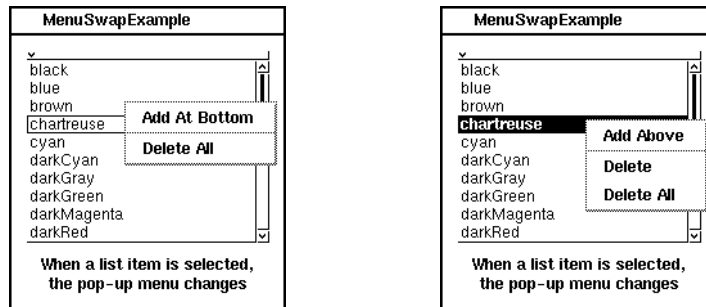
mb
add: 'First Notice' -> [self letter value: self class firstNotice];
add: 'Second Notice' -> [self letter value: self class secondNotice];

```

```
add: 'Final Notice' -> [self letter value: self class finalNotice].
```

```
^mb menu
```

Modifying a Menu Dynamically



Strategy

Sometimes a menu needs to change depending on conditions within the application. In the Resource Finder, for example, one set of menu items is displayed when an application is selected and another set is displayed when there is no selection.

The first variant shows how to substitute one menu for another. This is useful when the changes in a menu are major.

The second and third variants show how to add and remove menu items individually. This is useful when the changes are minor; however, this approach has the limitation that items are appended to the end of the menu.

The fourth variant shows how to temporarily hide and later reveal a menu item, preserving its position in the menu. This is useful when an item is to be repeatedly removed and reinstated.

Variants

V1. Substituting a Different Menu

Online example: MenuSwapExample

1. In the Menu property of the widget, enter the name of a method that returns a value holder containing a menu (in the example, menuHolder).

2. In the application model, create an instance variable to hold the menu in a value holder (menuHolder).
3. Use a System Browser to create a method (menuHolder) that returns the value of the instance variable.

```

menuHolder                                     "V1 Step3"
  ^menuHolder
    
```

4. Use a Menu Editor or System Browser to create the starting menu (nothingSelectedMenu) and the alternate menu (colorSelectedMenu).

```

nothingSelectedMenu                             "V1 Step 4"
  | mb |
  mb := MenuBuilder new.
  mb add: 'Add At Bottom' -> #unimplemented;
    line;
    add: 'Delete All' -> #unimplemented.
  ^mb menu
    
```

```

colorSelectedMenu                               "V1 Step 4"
  | mb |
  mb := MenuBuilder new.
  mb add: 'Add Above' -> #unimplemented;
    line;
    add: 'Delete' -> #unimplemented;
    add: 'Delete All' -> #unimplemented.
  ^mb menu
    
```

5. Use a System Browser to create an initialize method that gets the starting menu, puts it in a value holder, and assigns the holder to the instance variable.

```

initialize
  colors := SelectionInList with: ColorValue constantNames.
  colors selectionIndexHolder onChangeSend: #selectionChanged to: self.

  menuHolder := self nothingSelectedMenu asValue.
    
```

6. Create a method (selectionChanged) that tests to see which menu should be used and then puts the correct menu in the menu holder.

```
selectionChanged                                     "V1 Step 6"
  self colors selection isNil
    ifTrue: [self menuHolder value: self nothingSelectedMenu]
    ifFalse: [self menuHolder value: self colorSelectedMenu]
```

7. Arrange for the menu-changing method to be invoked when the relevant condition changes in the application. (In the example, an onChangeSend:to: message in the initialize method accomplishes this.)

V2. Adding an Item to a Menu

Online example: MenuModifyExample

1. Get the menu by sending a menuAt: message to the application model's builder. The argument is the name of the menu as identified in the Menu property.
2. Send an addItemLabel:value: message to the menu. The first argument is the label string and the second argument is a command, a value, or an action block.

```
addTitle
  "Prompt for a new job title and add it to the list."

  | newTitle jMenu |
  newTitle := Dialog request: 'New title?'.
  newTitle isEmpty ifTrue: [^self].

  jMenu := self builder menuAt: #jobTitlesMenu.           "V2 Step 1"
  jMenu addItemLabel: newTitle value: newTitle asSymbol. "V2 Step 2"

  self jobTitle value: newTitle asSymbol.
```

V3. Removing an Item from a Menu

Online example: MenuModifyExample

1. Get the menu by sending a `menuAt:` message to the application model's builder. The argument is the name of the menu as specified in the `Menu` property.
2. Get the item to be deleted by sending a `menuItemLabeled:` message to the menu. The argument is the label string of the menu item. (If the item is in a submenu, you must first access the submenu and get the item from it.)
3. Send a `removeItem:` message to the menu. The argument is the menu item from the previous step.
4. In the case of a menu button in which the current selection is displayed (that is, a menu button whose `Label` property is blank), make sure the button's value holder has a valid value. If necessary, choose a new value to displace the deleted menu item's value.

deleteTitle

"Prompt for a title and remove it from the list."

```
| jMenu removableTitles title item |
jMenu := self builder menuAt: #jobTitlesMenu.           "V3 Step 1"
```

```
"Don't permit the president to be overthrown."
removableTitles := jMenu labels
reject: [ :nextTitle | nextTitle = 'President'].
```

```
title := Dialog
choose: 'Delete Title'
fromList: removableTitles
values: removableTitles
lines: 8
cancel: [^nil]
for: ScheduledControllers activeController view.
```

```
item := jMenu menuItemLabeled: title.           "V3 Step 2"
jMenu removeItem: item.                         "V3 Step 3"
```

"If the deleted title is showing, pick the first title."

```
self jobTitle value == title asSymbol
  ifTrue: [self jobTitle value: #President].
```

"V3 Step 4"

V4. Hiding a Menu Item

Online example: MenuModifyExample

1. Get the menu by sending a `menuAt:` message to the application model's builder. The argument is the name of the menu as specified in the `Menu` property.
2. Get the item to be deleted by sending a `menuItemLabeled:` message to the menu. The argument is the label string of the menu item. (If the item is in a submenu, you must first access the submenu and get the item from it.)
3. To hide the item, send a `hideItem:` message to the menu. The argument is the menu item from the previous step. If the item is already hidden, no error occurs.
4. To reveal an item, send an `unhideItem:` message to the menu. The argument is the menu item from the previous step. If the item is already revealed, no error occurs.

adjustBenefitList

"Hide benefit items that are not available to the currently selected job title."

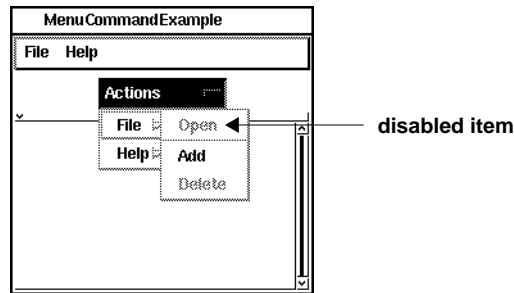
```
| bMenu item |
bMenu := self builder menuAt: #benefitsMenu.           "V4 Step 1"
item := bMenu menuItemLabeled: 'Golden Parachute'.    "V4 Step 2"
```

```
"Only the President gets the Golden Parachute."
self jobTitle value == #President
  ifTrue: [bMenu unhideItem: item]                     "V4 Step 4"
  ifFalse: [bMenu hideItem: item].                     "V4 Step 3"
```

See Also

- "Disabling a Menu Item" on page 248

Disabling a Menu Item



Strategy

Rather than removing a menu item when it is not appropriate for the user to select it, you can disable it. When disabled, it appears in a different color in the menu as a visual cue to the user, and it does nothing when the user tries to select it. Disabling a menu item is usually preferable to removing it, especially when the item is likely to be reinstated later.

In `MenuCommandExample`, menu items are enabled and disabled depending on whether a file name in a list is selected. The method that performs this service (`configureMenu`) is invoked at two different times: by a `postBuildWith:` method (to configure the menu at startup) and whenever the selection changes in the list (as arranged in the `initialize` method).

Basic Steps

Online example: `MenuCommandExample`

1. To disable a menu item, send a `disable` message to it. This is typically done after testing some condition in the application (in the example, after testing whether anything is selected in the list).
2. To enable a menu item, send an `enable` message to it.

`configureMenu`

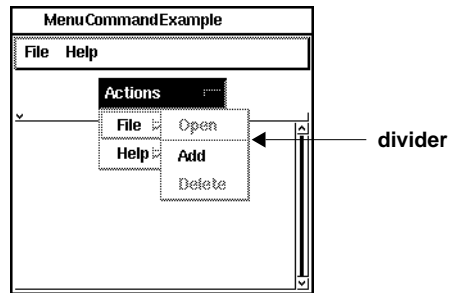
"Disable or enable the menu items depending on whether a file is selected."

```
| menu submenu |  
menu := self builder menuAt: #fileMenu.  
submenu := (menu menuItemLabeled: 'File') submenu.  
  
self files selection isNil  
  ifTrue: [  
    (submenu menuItemLabeled: 'Open') disable.      "Basic Step 1"  
    (submenu menuItemLabeled: 'Delete') disable]  
  ifFalse: [  
    (submenu menuItemLabeled: 'Open') enable.      "Basic Step 2"  
    (submenu menuItemLabeled: 'Delete') enable]
```

See Also

- “Modifying a Menu Dynamically” on page 243
- “Using a Menu Editor” on page 259

Adding a Divider to a Menu



Strategy

When a menu contains several items, it is often helpful to the user to group the items into functional sets. A submenu is one way of subdividing a large menu, but a divider line that provides visual separation is often adequate.

Basic Step

Online example: MenuCommandExample

- When creating a menu using a menu builder, send a line message to the menu builder before adding each new group of items.

fileMenu

```
| mb menu submenu |
mb := MenuBuilder new.
```

```
mb
```

```
beginSubMenuLabeled: 'File';
add: 'Open' -> #openFile;
line;
add: 'Add' -> #addFile;
add: 'Delete' -> #deleteFile;
endSubMenu.
```

"Basic Step"

```
mb
```

```
beginSubMenuLabeled: 'Help';
add: 'Usage' -> #explainUsage;
```

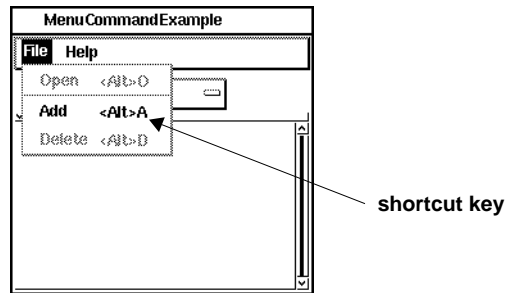


```
endSubMenu.  
  
"Add shortcut keys."  
menu := mb menu.  
submenu := (menu menuItemLabeled: 'File') submenu.  
(submenu menuItemLabeled: 'Open') shortcutKeyCharacter: $O.  
(submenu menuItemLabeled: 'Add') shortcutKeyCharacter: $A.  
(submenu menuItemLabeled: 'Delete') shortcutKeyCharacter: $D.  
  
^menu
```

See Also

- “Using a Menu Editor” on page 259

Adding a Shortcut Key



Strategy

For frequently used commands, it is helpful to provide a *keyboard shortcut* or *keyboard accelerator*—that is, a key sequence that invokes the command just as if the user had selected it from the menu. In some operating environments, every menu command is expected to have a keyboard equivalent. The basic steps show how to add a shortcut key to a menu bar.

Only a menu bar displays shortcut keys—not a menu button or a pop-up menu. This is true because only the menu bar is capable of responding to a keypress no matter where the cursor is located.

Basic Step

Online example: MenuCommandExample

- Send a `shortcutKeyCharacter:` message to a menu item. The argument is a character. The uppercase form of the character will appear in the menu, prefixed by `<Alt>`, indicating that the user must press the `<Alt>` key and the letter key simultaneously. (The `<Alt>` key has a different name on some keyboards.)

`fileMenu`

```
| mb menu submenu |
mb := MenuBuilder new.
```

```
mb
  beginSubMenuLabeled: 'File';
  add: 'Open' -> #openFile;
  line;
  add: 'Add' -> #addFile;
  add: 'Delete' -> #deleteFile;
endSubMenu.
```

```
mb
  beginSubMenuLabeled: 'Help';
  add: 'Usage' -> #explainUsage;
endSubMenu.
```

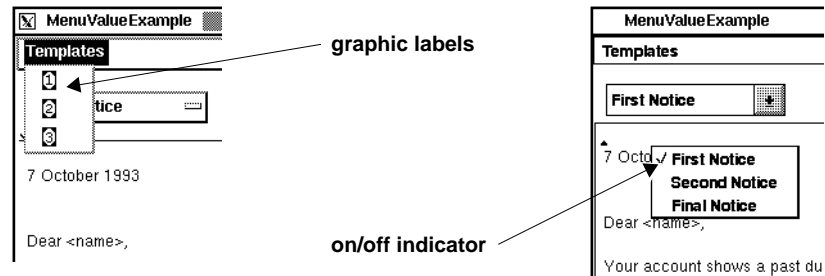
```
"Add shortcut keys."
menu := mb menu.
submenu := (menu menuItemLabeled: 'File') submenu.
(submenu menuItemLabeled: 'Open')
  shortcutKeyCharacter: $O.           "Basic Step"
(submenu menuItemLabeled: 'Add')
  shortcutKeyCharacter: $A.
(submenu menuItemLabeled: 'Delete')
  shortcutKeyCharacter: $D.
```

```
^menu
```

See Also

- “Sensing Keyboard Activity” on page 416

Displaying an Icon in a Menu



Strategy

Menu items can have a textual or graphical label.

The basic step shows how to substitute a graphic label for a textual label or combine the two.

The variant shows a special case in which a check mark or check box is prefixed to the textual label as a toggle indicator. This technique is frequently used with a menu item that represents a setting to indicate whether the condition is on or off. You can also use it to simulate a set of radio buttons in a menu, as MenuValueExample does.

Basic Step

Online example: MenuValueExample

- Send a `labelImage:` message to the menu item. The argument is any visual component, but typically it is a graphic image. The label string will be displaced to the right to make room for the image. The label string must have at least one character (even just a space).

```
templatesMenuForMenuBar
| mb menu submenu |
mb := MenuBuilder new.

mb
beginSubMenuItemLabeled: 'Templates';
add: ' ' -> [self letter value: self class firstNotice];
```

```
add: '' -> [self letter value: self class secondNotice];
add: '' -> [self letter value: self class finalNotice];
endSubMenu.
```

```
"Add graphic labels."
menu := mb menu.
submenu := (menu menuItemLabeled: 'Templates') submenu.
(submenu menuItemAt: 1)                                "Basic Step"
    labelImage: (self class oneImage).
(submenu menuItemAt: 2)
    labelImage: (self class twoImage).
(submenu menuItemAt: 3)
    labelImage: (self class threeImage).
```

```
"Set the background color."
submenu backgroundColor: ColorValue chartreuse.
```

```
^menu
```

Variant

Displaying an On/Off Indicator

Online example: MenuValueExample

1. To display an "on" indicator, send a `beOn` message to the menu item. The indicator is a check mark in some looks and a box in others.
2. To display an "off" indicator, send a `beOff` message. In some looks, `beOff` simply removes the "on" indicator; in others it displays a different image.

setCheckMark

"In the pop-up menu, set the check box to indicate the currently displayed template."

```
| menu item |
menu := self builder menuItemAt: #templatesMenuForPopUp.
```

```
item := menu menuItemAt: 1.
self letter value = self class firstNotice
```

```
ifTrue: [item beOn]
ifFalse: [item beOff].
```

```
"Variant Step 1"
"Variant Step 2"
```

```
item := menu menuItemAt: 2.
self letter value = self class secondNotice
ifTrue: [item beOn]
ifFalse: [item beOff].
```

```
item := menu menuItemAt: 3.
self letter value = self class finalNotice
ifTrue: [item beOn]
ifFalse: [item beOff].
```

See Also

- “Creating a Graphic Image” on page 658
- “Using a Menu Editor” on page 259

Changing Menu Colors

Strategy

You can modify the background color of a menu, as shown in the basic steps. This technique can be used to make all menus of a particular type appear similar. For example, you might make the Help menu a distinctive color wherever it occurs.

You can also group related items in a menu by applying a color to their labels, as shown in the basic steps. This approach is especially effective when you want to bring attention to the relationship among items that are not adjacent to one another.

Basic Steps

Online example: MenuModifyExample

1. To color a menu's background, send a `backgroundColor:` message to the menu. The argument is a paint, typically an instance of `ColorValue`.
2. To color a menu item's label, send a `color:` message to the menu item. The argument is a paint, typically a `ColorValue`.

benefitsMenu

```
| mb menu |
mb := MenuBuilder new.
mb add: 'Health Insurance' -> #health;
  add: 'Retirement Fund' -> #retirement;
  add: 'Life Insurance' -> #life;
  add: 'Stock Options' -> #stock;
  add: 'Golden Parachute' -> #parachute.
```

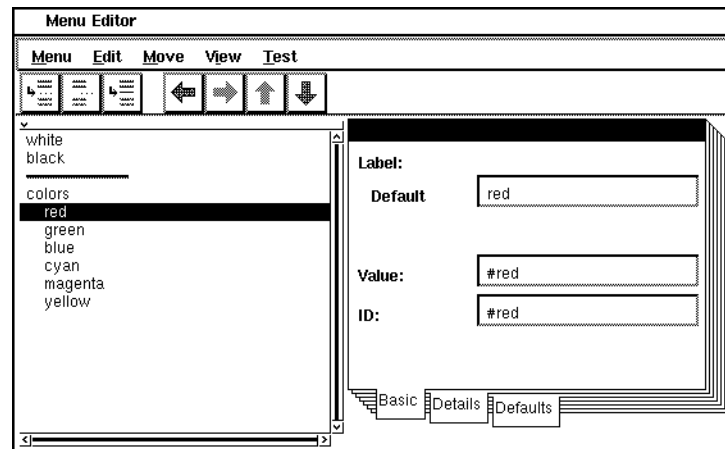
```
menu := mb menu.
menu backgroundColor: ColorValue chartreuse.           "Basic Step 1"
(menu menuItemLabeled: 'Golden Parachute')
  color: ColorValue red.                               "Basic Step 2"
```

```
^menu
```

See Also

- “Creating a Color” on page 686

Using a Menu Editor



Strategy

You can use a Menu Editor to create menus for menu bars, menu buttons, and pop-up menus. With a Menu Editor, you can create menus of commands and values (you cannot create menus of action blocks).

This topic assumes you are using the enhanced Menu Editor. To use the enhanced menu editor, turn on **Use Enhanced Tools** on the **UI Options** page of the **Settings Tool**. The enhanced Menu Editor provides a display in which the menu items appear as you create them, plus a notebook in which you can specify various properties for each menu item (label, value, identifier for programmatic use, shortcut character, graphical image for use in a label, on/off indicator, and initial states such as hiddenness).

The basic steps build a menu of values for a menu button. This menu includes a submenu and has a divider. To display a Menu Editor for the provided example, make sure that `MenuEditorExample` is filed in, then locate it in a Resource Finder, select the `colorMenu` resource and click **Edit**.

The variant shows how to access a menu, a submenu, and their menu items programmatically.

Basic Steps

Online example: MenuEditorExample

1. Open an enhanced Menu Editor (for example, from a Canvas Tool, choose **Tools**→**Menu Editor**).
2. Create the top-level menu. For each item that is to appear in it, choose **Edit**→**New Item** to create an empty item labeled <new item>.
3. Select each top-level <new item> and enter its string label in the **Label Default** property. Each label appears in the text area as soon as you accept input in the field (for example, by tabbing to the next field or pressing the <Return> key). In the example, create items labeled white, black, and colors.
4. Select each top-level item and enter its value in the **Value** property. Each value is turned into a symbol (with a prepended pound sign #) when you accept input in the field. In the example, enter values white and black for the first two items (colors does not need a value because it is the label for a submenu).
5. Create the submenu. Select the label for the submenu (colors) and choose **Edit**→**New Submenu Item** to create the first item in the submenu. Then, choose **Edit**→**New Item** once for each subsequent submenu item. In this example, create six items labeled <new item>, all at the same indentation level under the submenu label.
6. Select each indented <new item> and enter its string label in the **Label** property. In this example, create items labeled red, green, blue, cyan, magenta, and yellow.
7. Select each labeled submenu item and enter its value in the **Value** property. In this sample, enter values red, green, blue, cyan, magenta, and yellow.
8. Add a divider line below an item by selecting that item and choosing **Edit**→**Add Line**. In the example, add a divider line below the item labeled black.
9. Pull down the Test menu in the menu bar of the Menu Editor to test the menu you have created. Adjust any menu items as needed—for example, use **Move**→**Left** and **Move**→**Right** to change the level of indentation of the items; use **Move**→**Up** and **Move**→**Down** to change the order of items.

10. Choose **Menu→Install...** to install a specification for the menu in a resource method of the application model. In this example, install the menu specification in a `colorMenu` class method in `MenuEditorExample`.
11. In the canvas, select the menu button to which you want to apply the menu.
12. In a Properties tool, fill in the menu button's **Menu** property with the name of the menu resource (`colorMenu`). Apply properties and install the canvas.

Variant

Accessing Menus Programmatically

Online example: `MenuEditorExample`

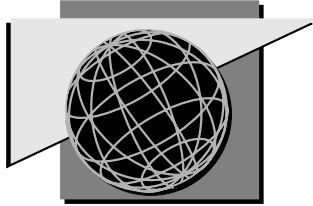
1. In an enhanced Menu Editor, select the menu item to be accessed and fill in its **ID:** property with an identifying *name key*. In the example, use the corresponding color name as the name key for each item. Install the menu.
2. In a System Browser, create the method that is to access the menu programmatically (in this example, `disableDarkColors`). Get the menu by sending a `menuAt:` message to the builder. The argument is the name of the menu's resource method (`#colorMenu`).
3. Get the menu's collection of menu items by sending a `menuItems` message to the menu.
4. Send a `nameKey` message to each menu item to obtain its name key. In this example, disable each menu item whose name key is in the `darkColors` array.
5. Get the menu item that serves as the label for the submenu (in this example, get the menu item whose name key is `#colors`). To do this, send an `atNameKey:` message to the menu, specifying the desired name key (`#colors`) as the argument.
6. Get the submenu by sending a `submenu` message to the menu item returned in step 5.
7. Get the submenu's menu items by sending a `menuItems` message to the submenu. In this example, disable each submenu item whose name key is in the `darkColors` array.

disableDarkColors

```
| menu submenu darkColors |  
darkColors := #(#black #red #blue #magenta).
```

```
menu := self builder menuAt: #colorMenu.           "Variant Step 2"  
menu menuItems do: [ :menuItem |                 "Variant Step 3"  
    (darkColors includes: menuItem nameKey)       "Variant Step 4"  
    ifTrue: [menuItem disable]].
```

```
submenu := (menu atNameKey: #colors) submenu.     "Variant Step 5, 6"  
submenu menuItems do: [ :menuItem |              "Variant Step 7"  
    (darkColors includes: menuItem nameKey)  
    ifTrue: [menuItem disable]].
```



Chapter 14

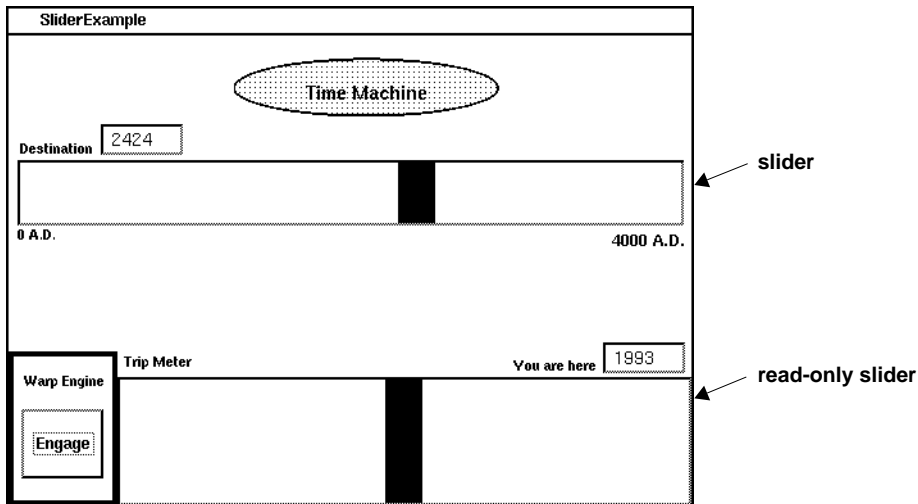
Sliders

Adding a Slider	264
Connecting a Slider to a Field	267
Changing the Range Dynamically	270
Changing the Length of the Marker	273
Making a Slider Two-Dimensional	274

See Also

- “Widget Basics” on page 53

Adding a Slider



Strategy

A slider widget simulates the sliding switch that some electronic devices use for controlling volume, bass level, and other properties. A slider enables you as the designer of an application to define a specific range of legal values, and it enables the user to conveniently select a value within that range.

Basic Steps

Online example: Slider1Example (the Destination slider)

1. Use a Palette to add a slider widget to the canvas. Leave the slider selected.
2. In the Properties Tool, fill in the slider's **Aspect** property with the name of the method (destination) that will supply a value model for the slider.
3. Optionally change the **Start** (0), **Stop** (4000), and **Step** (10) properties, which control the endpoints of the range and the increment by which the marker will move. The default **Start** is 0 and the default **Stop** is 1. The default **Step** is nil, giving the effect of continuous marker motion.
4. Apply the properties and install the canvas.

5. Use the canvas's **define** command or a System Browser to create an instance variable to hold the slider's value model (destination).
6. Use the canvas's **define** command or a System Browser to create a method, in an aspects protocol, for accessing the instance variable (destination).

```
destination                                     "Basic Step 6"
  ^destination
```

7. Use a System Browser to initialize the variable, usually in an initialize method, in an initialize release protocol. Initialize the variable with a value holder whose initial value is the current year.

```
initialize
  "Destination"
  destination := Date today year asValue.           "Basic Step 7"

  "Current year"
  currentYear := Date today year asValue.

  "Trip meter"
  tripRange := RangeAdaptor
    on: currentYear
    stop: 4000
    grid: 1.
```

Variants

V1. Making a Slider Vertical

By default, a slider is horizontal in shape, and the marker moves horizontally as well.

1. To alter the shape of a slider, drag the selection handles of the widget.
2. To alter the marker's direction of movement, select the slider's **Vertical** or **Horizontal** property.
3. Apply the properties and install the canvas.

Note that it is possible to have a slider that is horizontal in shape but vertical in operation.

V2. Making a Slider Read-Only

Although it is normally an input device, a slider can be used purely as an output device. In `Slider1Example`, we use a read-only slider as a meter to display the progress of the user's time-traveling adventure.

1. Select the slider in the canvas.
2. In a Properties Tool, fill in the slider's ID property with an identifying name (`tripRange`).
3. In a method in the application model (typically `postBuildWith:`), get the slider component from the builder and disable it.

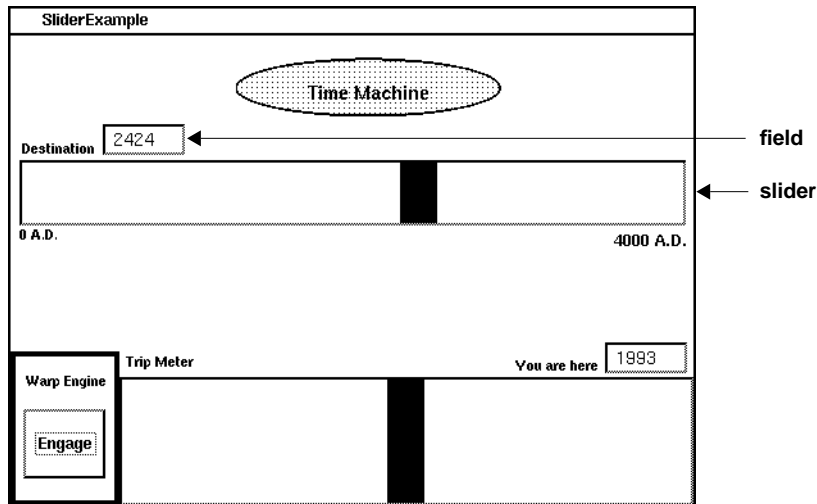
postBuildWith: aBuilder

"Disable the trip meter, making it read-only."

(aBuilder componentAt: #tripRange) disable.

"V2 Step 3"

Connecting a Slider to a Field



Strategy

Although a slider is both an input and an output device, it typically gives the user only a rough idea of the current value. Frequently, a field is used to display the same value precisely.

For example, the `Slider1Example` uses a field to display the destination year, because the slider covers such a large range (zero to 4000) that the user can only guess at its current value.

Unless you make the field read-only, the user has the option of changing the value by using either the slider or the field.

Nonnumeric slider: By its nature, a slider always manipulates a numeric value. You can make it appear to manipulate a nonnumeric value, however, by using a field to display the transformed value. The variant shows how to do so.

Basic Steps

Online example: `Slider1Example` (the `Destination` slider and field)

1. Use a Palette to add a field to the canvas. Leave the field selected.

2. In the Properties Tool, fill in the field's Aspect property with the same name that the slider uses for its Aspect (destination).
3. In the field's Type property, select Number.
4. Apply the properties and install the canvas.

Variant

Displaying a Transformed Value in the Field

Online example: Slider2Example (the Month field)

1. In the field's Aspect property, enter a *different* method name than the slider's Aspect (in the example, the slider's Aspect is dateRange while the field's Aspect is month).
2. In the field's Type property, select the type that corresponds to the transformed value (in the example, a month name will be displayed, so we use a String type field).
3. Use the canvas' define command or a System Browser to create the field's instance variable (month) and accessing method (month).

```
month                                     "Variant Step 3"
  ^month
```

4. In a method in the application model (typically initialize), initialize the field's variable.
5. In the initialize method, arrange for a change message (changedDate) to be sent to the application model when the slider's value changes.

```
initialize
  month :=(Date nameOfMonth: 1) asValue.           "Variant Step 4"
  year := 1900 asValue.

  dateRange := (0@1) asValue.
  dateRange onChangeSend: #changedDate to: self.  "Variant Step 5"
```

6. Use a System Browser to create the change method (changedDate) in the application model. This method is responsible for changing the field's value based on the slider's new value.

```

changedDate
"Convert the y-axis value to a month."                                "Variant Step 6"
| y x |
y := self dateRange value y.
y := (12 - (y * 12) asInteger) max: 1.                                "(12 months)"
self month value: (Date nameOfMonth: y).

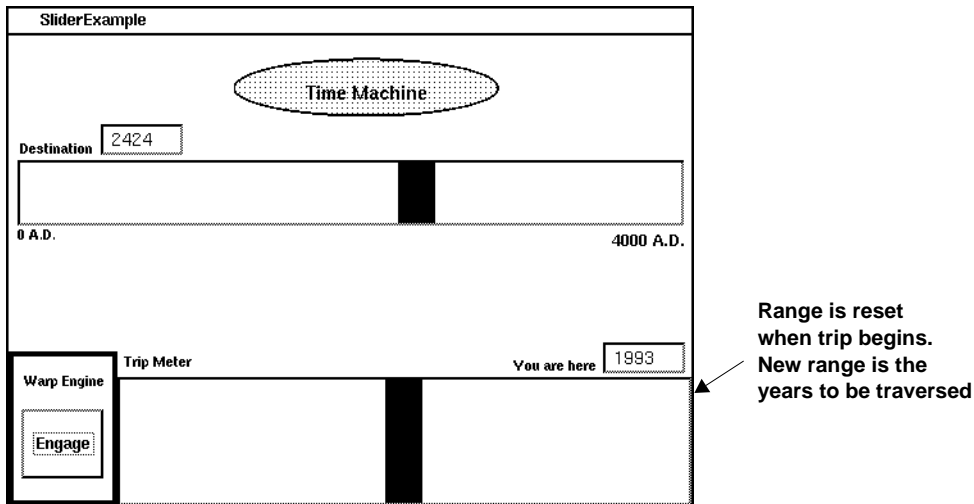
"Convert the x-axis value to a year."
x := self dateRange value x.
x := 1900 + (x * 100) asInteger.                                     "(100 years)"
self year value: x.

```

See Also

- “Creating an Input Field” on page 122

Changing the Range Dynamically



Strategy

When the slider's range is unchanging, you can use the slider's **Start**, **Stop**, and **Step** properties to set the range and the step value. When the range or step varies, however, this approach is not sufficient.

A `RangeAdaptor` provides the required flexibility. It is a specialized value model that also keeps track of the range and step values. You can change those values by sending messages to the adaptor. This can be done anytime—in the `Slider1Example`, the trip meter's range is modified every time the `Engage` button is pressed.

Basic Steps

Online example: `Slider1Example` (the `Trip Meter` slider)

1. In a method in the application model (typically in an `initialize` method), initialize the slider's aspect variable with a `RangeAdaptor` by sending the instance creation message (`on:start:stop:grid:`). The first argument (`currentYear`) is a value holder containing the number that the slider manipulates. (When a field is connected to the slider, as in the example,

this argument is the field's aspect variable.) The grid argument is the step value.

initialize

```
"Destination"
destination := Date today year asValue.

"Current year"
currentYear := Date today year asValue.

"Trip meter"
tripRange := RangeAdaptor                                "Basic Step 1"
  on: currentYear
  start: 0
  stop: 4000
  grid: 1.
```

2. Whenever the range or step must change, send a rangeStart:, rangeStop:, or grid: message to the adaptor. (In the example, this is done in the engage method.)

engage

```
"Start the time trip."

| startingYear destinationYear direction |
startingYear := self currentYear value.
destinationYear := self destination value.

destinationYear == startingYear
  ifTrue: [^Dialog warn: 'Please select a new destination.'].

"Set the endpoints on the trip meter."
self tripRange                                "Basic Step 2"
  rangeStart: startingYear;
  rangeStop: destinationYear;
  grid: 1.

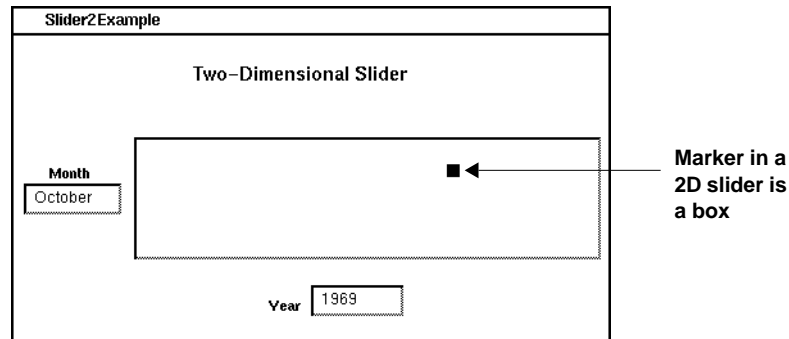
"Reset the meter to the starting position."
currentYear value: startingYear.

"Set up a step value for the loop that follows (-1 = backward in time)."
destinationYear > startingYear
```

```
ifTrue: [direction := 1]  
ifFalse: [direction := -1].
```

```
"For each year of time travel, update the current year."  
startingYear to: destinationYear by: direction do: [ :yr |  
currentYear value: yr].
```

Changing the Length of the Marker



Strategy

By default, a slider's marker is 29 pixels long. This length is suitable for most purposes. For a very short slider, however, a shorter marker may be more pleasing.

Before you change the marker width, be aware that the marker's appearance changes under different window-manager looks. In particular, the beveled appearance used by some window managers makes a marker that is less than 3 pixels wide display incorrectly.

Basic Step

Online example: Slider2Example

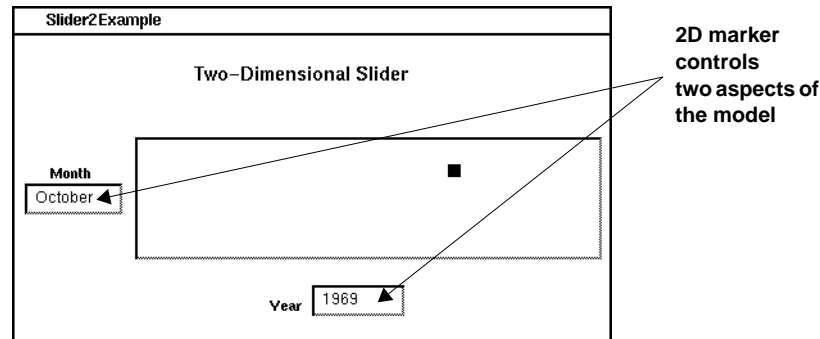
- In a message in the application model (typically `postBuildWith:`), get the slider widget from the builder and send a `setMarkerLength:` message to it, with the length in pixels as argument.

`postBuildWith: aBuilder`

```
(aBuilder componentAt: #dateRange) widget
  beTwoDimensional;
  setMarkerLength: 10.
```

"Basic Step"

Making a Slider Two-Dimensional



Strategy

By default, a slider operates in one dimension, changing a value along a linear scale. You can arrange for a slider to manipulate a point in two dimensions and then use the *x*-axis and *y*-axis components of that point to control two separate parameters.

In `Slider2Example`, a two-dimensional slider is used to alter two fields simultaneously. The first field, which uses the *y*-axis component of the slider's value, displays one of the 12 months. The second field uses the *x*-axis component of the slider's value to arrive at a year between 1900 and 2000.

Basic Steps

Online example: `Slider2Example`

1. In a method in the application model (typically `initialize`), initialize the slider's variable to an instance of `Point` that is held by a value holder.

`initialize`

```
month := (Date nameOfMonth: 1) asValue.
```

```
year := 1900 asValue.
```

```
dateRange := (0@1) asValue.
```

```
dateRange onChangeSend: #changedDate to: self.
```

"Basic Step 1"

2. In a `postBuildWith:` method, get the slider from the builder and ask it to `beTwoDimensional`.

postBuildWith: aBuilder

```
(aBuilder componentAt: #dateRange) widget
    beTwoDimensional;
    setMarkerLength: 10.
```

"Basic Step 2"

Variant

Connecting a Two-Dimensional Slider to Two Fields

1. In a method in the application model (typically initialize), arrange for a change message (changedDate) to be sent to the application model when the slider's value changes.

initialize

```
month :=(Date nameOfMonth: 1) asValue.
year := 1900 asValue.

dateRange := (0@1) asValue.
dateRange onChangeSend: #changedDate to: self.
```

"Variant Step 1"

2. Use a System Browser to create the change method (changedDate) in the application model. This method splits the slider's value into its x-axis and y-axis components. Each component is a value between 0 and 1 and is transformed as needed to produce a suitable value for the related field.

changedDate "Variant Step 2"

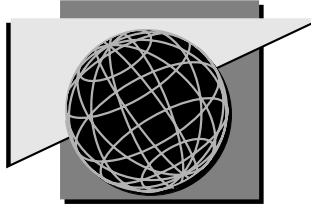
```
"Convert the y-axis value to a month."
| y x |
y := self dateRange value y.
y := (12 - (y * 12) asInteger) max: 1. "12 months"
self month value: (Date nameOfMonth: y).

"Convert the x-axis value to a year."
x := self dateRange value x.
x := 1900 + (x * 100) asInteger.
self year value: x.
```

"(100 years)"

See Also

- “Creating an Input Field” on page 122



Chapter 15

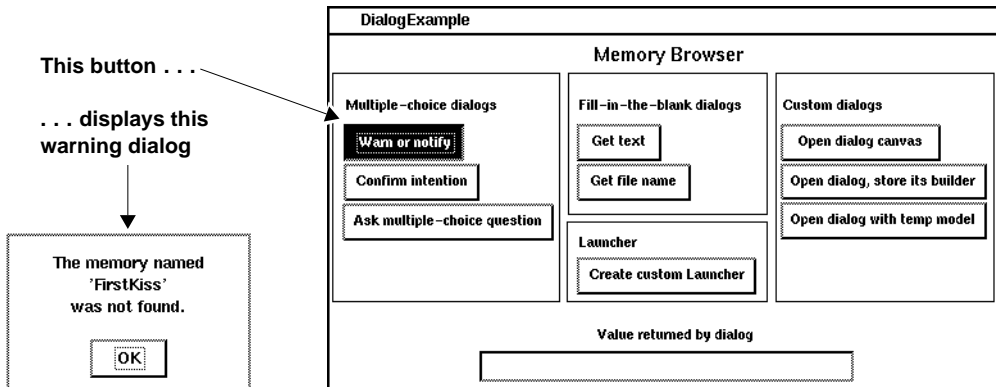
Dialogs

Displaying a Warning	278
Asking a Yes/No Question	280
Asking a Multiple-Choice Question	282
Requesting a Textual Response	284
Requesting a Filename	286
Choosing from a List of Items	289
Linking a Dialog to a Master Window	292
Creating a Custom Launcher	294
Creating a Custom Dialog	296

See Also

- “Widget Basics” on page 53

Displaying a Warning



Strategy

A warning dialog is frequently used when an action cannot be completed. For example, when a search command cannot find a user-specified string, the command normally reports this in a warning dialog. In general, a warning dialog can be used to display any simple textual message. The message can have embedded carriage returns and multiple text styles.

The dialog provides an OK button with which the user can dismiss the dialog. For this reason, a warning dialog is often referred to as an *OK dialog*.

A warning dialog is displayed by sending a `warn:` message to the Dialog class. When the user clicks OK, the message returns the value `nil`.

Basic Step

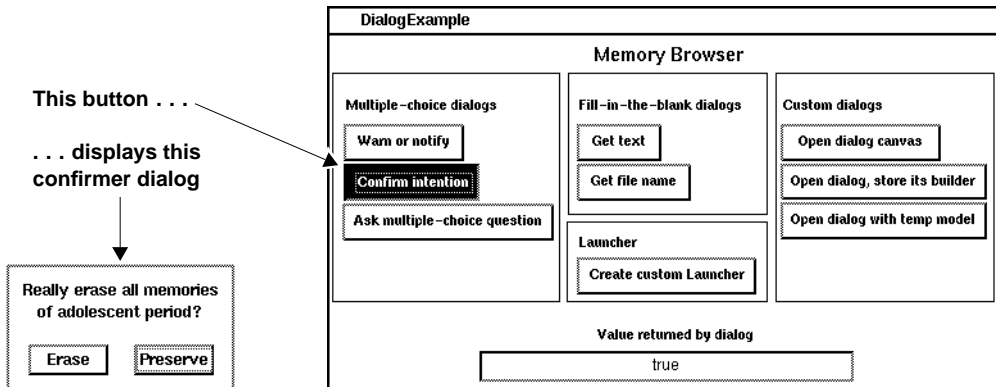
Online example: DialogExample

- In the method responsible for bringing up the dialog, send a `warn:` message to the Dialog class. The argument is the dialog's label string. Note that the backslash characters in this string are converted to carriage returns by the `withCRs` message.

```
warn
| returnVal |
returnVal := Dialog
  warn: 'The memory named\'FirstKiss\'was not found.!'
  withCRs. "Basic Step"

"Update the text field in the main window."
self returnedValue value: returnVal printString.
```

Asking a Yes/No Question



Strategy

Frequently an application needs to ask the user a yes/no question. A common situation for using such a dialog is when the user is initiating an action that may have unintended side effects, such as closing a file editor before saving edits. Because a yes/no dialog is so often used to confirm a dangerous action, it is often referred to as a *confirmer*.

By convention, the question is phrased in such a way that a Yes answer causes the action to proceed. Except in the most hazardous situations, Yes is also the default answer.

A confirmer dialog is displayed by sending a `confirm:` message to the Dialog class. When the user clicks Yes, the message returns the value `true`. When the user clicks No, the message returns the value `false`.

The basic steps show how to use a `confirm:` message. The first variant shows how to specify the default answer. The second variant specifies a master window from which the dialog adopts certain look-specific features such as its colors.

Basic Step

- Send a `confirm:` message to the Dialog class. The argument is the question to be asked.

```
Dialog confirm: 'Really erase all memories\of adolescent period?'
withCRs. "Basic Step"
```

Variants

V1. Supplying a Default Answer

- Send a `confirm:initialAnswer:` message to `Dialog`. The second argument is either `true` or `false`.

```
Dialog "V1 Step"
confirm: 'Really erase all memories\of adolescent period?' withCRs
initialAnswer: false
```

V2. Adopting the Look of a Master Window

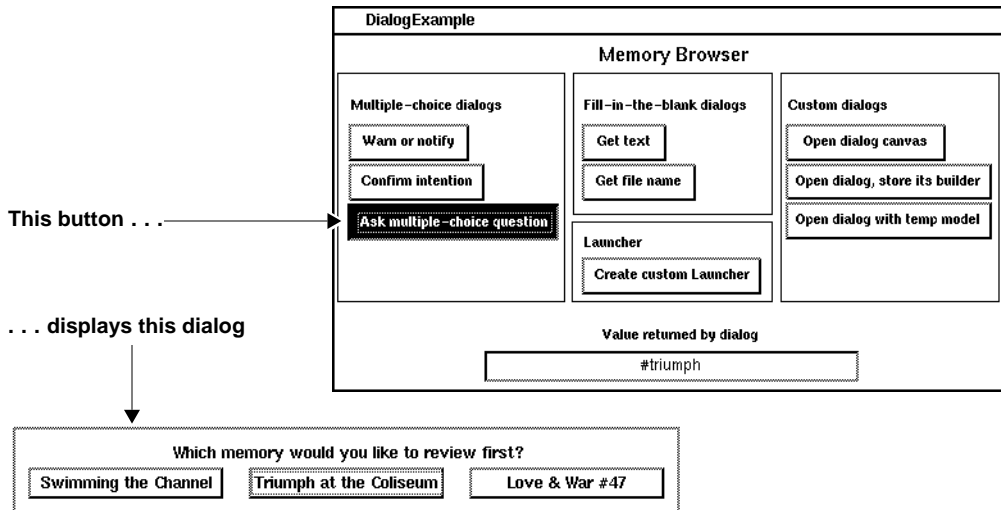
Online example: `DialogExample`

- Send a `confirm:initialAnswer:for:` message to `Dialog`. The third argument is the master window, typically the currently active window.

```
confirm
| returnVal |
returnVal := Dialog "V2 Step"
confirm: 'Really erase all memories\of adolescent period?' withCRs
initialAnswer: false
for: ScheduledControllers activeController view.

"Update the text field in the main window."
self returnedValue value: returnVal printString.
```

Asking a Multiple-Choice Question



Strategy

Frequently an application requires a means of offering the user a small set of choices. The Dialog class provides a dialog that accommodates any number of choices. In practice, because the choices are arrayed as a horizontal row of buttons, this dialog is useful only for a very few choices.

In the message that creates a multiple-choice dialog, you assign a symbol to each choice (basic steps). When the user clicks a choice, the message returns the corresponding symbol to the application. You can use a multiple-choice dialog to simulate a yes/no dialog when you want the dialog to return values other than true and false.

The variant indicates a master window from which the dialog adopts certain look-specific features such as its colors.

Basic Step

- Send a `choose:labels:values:default:` message to the Dialog class. The `choose` argument is the question. The `labels` argument is an array of strings to be displayed on the answer buttons. The `values` argument is an array of Symbols to be used as

return values by the answer buttons. The default argument is the Symbol that is associated with the desired default answer.

```
Dialog                                     "Basic Step"
  choose: 'Which memory would you like to review first?'
  labels: #('Swimming the Channel'
           'Triumph at the Coliseum'
           'Love & War #47')
  values: #(#swim #triumph #love47)
  default: #triumph
```

Variant

Adopting the Look of a Master Window

Online example: DialogExample

- Send a choose:labels:values:default:for: message to Dialog. The first four arguments are as described above. The for: argument is typically the active window.

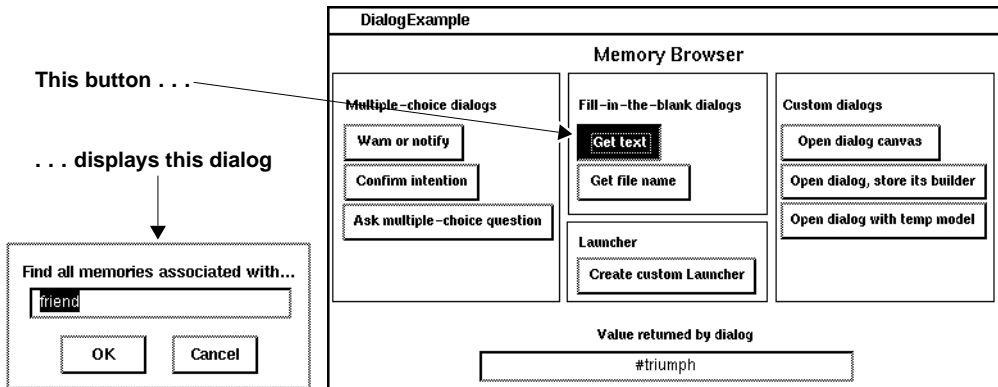
```
askMultiChoice
| returnVal |
returnVal := Dialog                                     "Variant Step"
  choose: 'Which memory would you like to review first?'
  labels: #('Swimming the Channel'
           'Triumph at the Coliseum'
           'Love & War #47')
  values: #(#swim #triumph #love47)
  default: #triumph
  for: ScheduledControllers activeController view.

self returnedValue value: returnVal printString.
```

See Also

- “Choosing from a List of Items” on page 289

Requesting a Textual Response



Strategy

A fill-in-the-blank dialog contains an input field and a label. It is commonly used to prompt for a string, such as a search string. By default, an empty string appears in the input field (basic steps). The first variant shows how to supply a different default.

When the user fills in a string and clicks OK, the message that creates the dialog returns the user-specified string. When the user clicks Cancel, an empty string is returned by default. The second variant shows how to arrange for a different value to be returned for canceling. The block that is used to return a canceling value can also be used to take other action, such as prompting the user for a nonblank response.

Basic Step

- Send a request: message to the Dialog class, with the question as the argument.

Dialog request: 'Find all memories associated with...' "Basic Step"

Variants

V1. Supplying a Default Answer

- Send a request:initialAnswer: message to Dialog. The second argument is the default answer string.

```
Dialog
  request: 'Find all memories associated with...'
  initialAnswer: 'friend'                                     "V1 Step"
```

V2. Supplying a Cancel Block

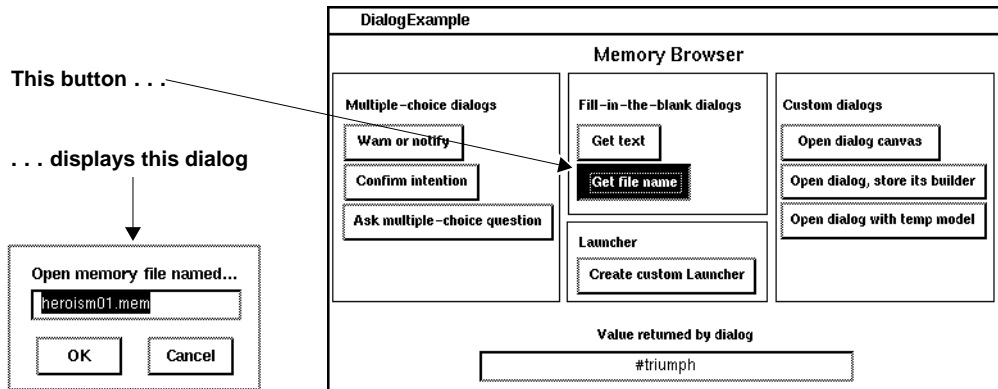
Online example: DialogExample

- Send a request:initialAnswer:onCancel: message to Dialog. The third argument is a block containing the action to be taken, the value to be returned, or both.

```
getText
| returnVal |
returnVal := Dialog
  request: 'Find all memories associated with...'
  initialAnswer: 'friend'
  onCancel: [self defaultRuminationTopic].                   "V2 Step"

"Update the text field in the main window."
self returnedValue value: returnVal printString.
```

Requesting a Filename



Strategy

A filename is a special case for a fill-in-the-blank dialog because it is frequently desirable to test for the existence of the named file. The built-in dialog performs this service automatically and re-prompts as needed. In addition, it responds to wildcard characters (* and #) by displaying a list of all files that match the pattern.

By default, the dialog accepts any filename that is accepted by the operating system. The variants show how to arrange for the dialog to take various actions depending on whether the file is supposed to exist already.

When the user clicks Cancel, an empty string is returned by default. The final variant shows how to arrange for a different value to be returned, an action to be taken, or both.

Basic Step

- Send a `requestFileName:` message to the Dialog class. The argument is a label string for the dialog.

Dialog `requestFileName:` 'Open memory file named...' "Basic Step"

Variants

V1. Supplying a Default Filename

- Send a `requestFileName:default:` message to Dialog. The second argument is a string containing the name of the default file.

```
Dialog
  requestFileName: 'Open memory file named...'
  default: 'hero01.mem'                                     "V1 Step"
```

V2. Confirming When the File Already Exists

- Send a `requestFileName:default:version:` message to Dialog. The third argument is the `#new` symbol, which indicates that you expect the file to be a new one.

```
Dialog
  requestFileName: 'Open memory file named...'
  default: 'hero01.mem'
  version: #new                                           "V2 Step"
```

V3. Confirming When the File Does Not Exist

- Send a `requestFileName:default:version:` message to Dialog. The third argument is the `#old` symbol, which indicates that you expect the file to exist.

```
Dialog
  requestFileName: 'Open memory file named...'
  default: 'hero01.mem'
  version: #old                                           "V3 Step"
```

V4. Canceling When the File Already Exists

- Send a `requestFileName:default:version:` message to Dialog. The third argument is the `#mustBeNew` symbol, which indicates that you require the file to be a new one.

```
Dialog
  requestFileName: 'Open memory file named...'
```

```

default: 'hero01.mem'
version: #mustBeNew

```

"V4 Step"

V5. Canceling When the File Does Not Exist

- Send a `requestFileName:default:version:` message to Dialog. The third argument is the `#mustBeOld` symbol, which indicates that you require the file to be an existing one.

```

Dialog
  requestFileName: 'Open memory file named...'
  default: 'hero01.mem'
  version: #mustBeOld

```

"V5 Step"

V6. Supplying a Cancel Block

Online example: DialogExample

- Send a `requestFileName:default:version:ifFail:` message to Dialog. The final argument is a block containing the action to be taken, the value to be returned, or both.

```

getFilename
| returnVal |
returnVal := Dialog
  requestFileName: 'Open memory file named...'
  default: 'hero01.mem'
  version: #mustBeOld
  ifFail: [Transcript show: 'Memory file access canceled'. "]. "V6 Step"
  "Update the text field in the main window."
  self returnedValue value: returnVal printString.

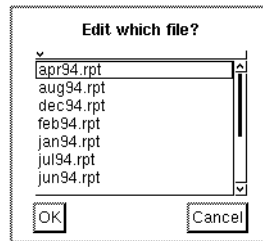
```

See Also

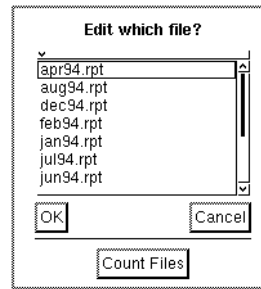
- “Creating a File or Directory” on page 592

Choosing from a List of Items

basic dialog



dialog with custom button



Strategy

You can display a dialog with a built-in list of commands or data values. This dialog is used as a kind of stand-alone menu. Each item in the list is associated with a value, just as a menu item is, and your application can either insert the selected value in a value holder or trigger an action.

By default, the dialog contains a list of items, an OK button, and a Cancel button (basic step). The variant shows how to add custom buttons to the dialog, for situations when neither selecting an item in the list nor canceling the dialog is acceptable. For example, when you enter a wildcard pattern in a file-pathname dialog, a list dialog shows the files that match your pattern and offers a Try again button in case you want to try a different pattern.

Basic Step

- Send a `choose:fromList:values:lines:cancel:` message to the Dialog class. The `choose:` argument is a prompt string. The `fromList:` argument is a collection of strings—either command names or value descriptions (in the example, filenames). The `values:` argument is a collection of the same size as the `fromList:` collection, containing the values to be associated with the list items. The `lines:` argument is an integer indicating the maximum number of list items to display (for a long list). The `cancel:` argument is a block containing the action to be

taken or the value to be supplied when the Cancel button is selected by the user.

```
| files response |
files := Filename defaultDirectory directoryContents
reject: [ :name | name asFilename isDirectory].

response := Dialog                                     "Basic Step"
  choose: 'Edit which file?'
  fromList: files
  values: files
  lines: 8
  cancel: [^nil].

response asFilename edit.
```

Variant

Supplying Extra Action Buttons Below the List

- Send a choose:fromList:values:buttons:values:lines:cancel: message to the Dialog class. The arguments are the same as in the basic step, with the addition of buttons: and values:. The buttons: argument is a collection of strings to be used as button labels. The values: argument is a collection of values to be associated with the button labels.

```
| files response |
files := Filename defaultDirectory directoryContents
reject: [ :name | name asFilename isDirectory].

response := Dialog
  choose: 'Edit which file?'
  fromList: files
  values: files
  buttons: #('Count Files')
  values: #(#count)
  lines: 12
  cancel: [^nil].                                     "Variant Step"
```



```
response == #count  
  ifTrue: [Dialog warn: files size printString]  
  ifFalse: [response asFilename edit]
```

See Also

- “Asking a Multiple-Choice Question” on page 282

Linking a Dialog to a Master Window

Strategy

By default, the built-in dialogs use system defaults for their colors and UI Look. When your application employs a special set of colors or a nondefault UI Look, you can arrange for dialogs to mimic the colors and UI Look of a master window. In addition, some window systems create a visual connection between a dialog and its master window.

The basic steps shows how to link a warning dialog to the currently active window with a `warn:for:` message. A master window with a yellow background color is opened. You can add a `for:` argument to other dialog-creation messages. The master window is typically the main application window, which an application model can access through `self builder window`.

Basic Steps

1. Send a `useColorOverridesFromParent:` message to the `SimpleDialog` class. The argument `true` causes subsequently opened dialogs to adopt the colors of their master window, in addition to the UI look. By default, instances of `SimpleDialog` and its subclasses adopt only the UI look of the master window. (Note that 'Overrides' is misspelled in the method name and must therefore be misspelled here.)
2. Send a `warn:for:` message to the `Dialog` class. The first argument is the message string, and the second argument is the master window.

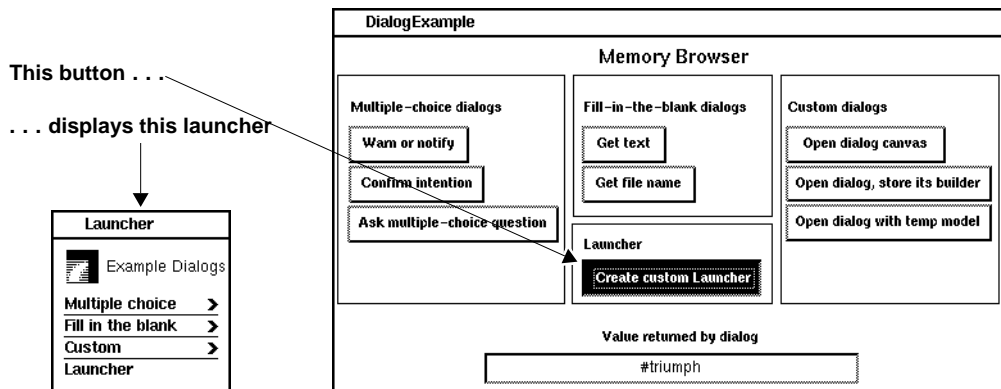
```
| masterWindow |
SimpleDialog useColorOverridesFromParent: true.           "Basic Step 1"
masterWindow := ScheduledWindow new.
masterWindow background: ColorValue yellow.
masterWindow open.

Dialog                                                    "Basic Step 2"
  warn: 'This dialog has a yellow background, too.'
  for: masterWindow.
```

`masterWindow sensor eventQuit: nil.`

Note that you may want to reset the SimpleDialog class to its default behavior by sending it the `useColorOverridesFromParent:` message with the argument `false`.

Creating a Custom Launcher



Strategy

A Launcher is a window whose widgets provide access to other parts of an application. Launchers offer similar functionality to dialogs. You can create a custom Launcher for each of your applications or a single custom Launcher for all of them.

By default, the Launcher's window label is "Launcher." The second variant shows how to arrange for an alternative window label.

You can also arrange for a heading within the Launcher window, as shown in the second variant. (The second variant presents the fullest form of the message for creating a Launcher, so only that variant has example code.)

Basic Step

- Send an `openOnMenu:` message to the `LauncherView` class. The argument is an instance of `Menu`.

Variants

V1. Supplying an Alternative Window Label

- Send an `openOnMenu:withLabel:` message to `LauncherView`. The second argument is the window's label string.

V2. Supplying a Heading

Online example: DialogExample

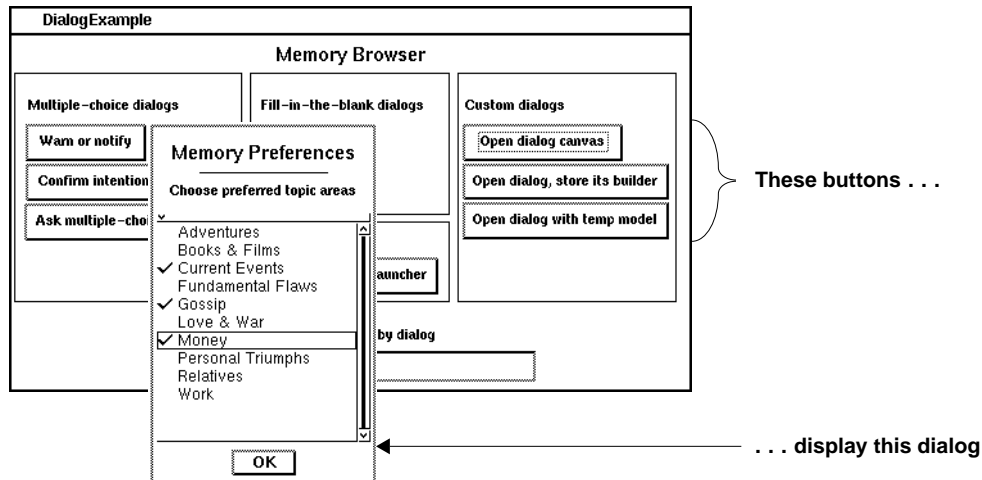
- Send an `openOnMenu:withLabel:andHeader:` message to `LauncherView`. The third argument is a string containing the desired header. The string can contain embedded carriage returns, which cause the header to be displayed on multiple lines.

```
createLauncher
  LauncherView                                "V2 Step"
    openOnMenu: self dialogMenu
    withLabel: 'Launcher'
    andHeader: 'Example Dialogs'.
```

See Also

- “Creating a Menu” on page 226

Creating a Custom Dialog



Strategy

When a built-in dialog is not sufficient, you can paint a canvas that has the desired widgets on it. You can then open the resulting interface specification in a dialog window—that is, in a window whose controller yields control only after the dialog has been closed.

The basic technique is to ask the application model to open a dialog window from an installed interface specification (basic steps). The dialog is created as an instance of `SimpleDialog`, which provides its own interface builder for setting up the dialog's widgets. This builder obtains any needed value models, actions, and resources for the widgets from the application model. Note, however, that buttons whose `Action` properties are `#accept` or `#cancel` obtain their actions from the `SimpleDialog` instance instead. These predefined actions are useful for OK and Cancel buttons on the dialog (first variant).

By default, the dialog's builder is discarded after the interface is constructed. If your application will need to access any widgets in the dialog (for disabling, etc.), you should save the builder in an instance variable of the application model for later use in any method (second variant).

A second technique for creating a custom dialog (not illustrated here) is to create a separate model for the dialog (typically, a subclass of `SimpleDialog`). You install the dialog's canvas in this subclass and then program the subclass to provide the value models, actions, and resources needed by the dialog's widgets. A method in the main application model asks the dialog's model to open itself and use itself as the source of value models, actions, and resources. This technique enables you to reuse the dialog more easily in further applications.

A third technique for creating a custom dialog is to program the application model to create an instance of `SimpleDialog` and configure its interface builder dynamically (third variant). This has the effect of creating a temporary model for the dialog, which is useful when the value models for the dialog's widgets are not needed beyond the lifetime of the dialog. For example, a file-finding dialog might employ several widgets, each requiring a value model, but only the ultimate filename is of interest to the application.

Basic Step

Online example: `DialogExample`

- In the method that is to open the dialog, send an `openDialogInterface:` message to the application model. The argument is the symbol that identifies the dialog's interface specification.

```
openDialogCanvas
| returnVal |
returnVal := self openDialogInterface: #memoryZonesDialog.    "Basic Step"
"Update the text field in the main window."
self returnedValue value: returnVal printString.
```

Variants

V1. Requesting Actions for OK and Cancel Buttons

Online example: DialogExample

When a custom dialog has OK and Cancel buttons, you can arrange for them to invoke predefined methods that close the dialog and return the appropriate value (true or false).

1. In the canvas for the dialog, select the action button that is to accept the dialog (typically labeled OK).
2. In the Properties Tool, enter `accept` in the button's Action property.
3. In the canvas, select the button that is to cancel the dialog (typically labeled Cancel).
4. In the Properties Tool, enter `cancel` in the button's Action property.
5. Apply the properties and install the canvas.

These Action settings cause the buttons to send `accept` and `cancel` messages to the `SimpleDialog` instance. Consequently, if you define methods named `accept` or `cancel` in the application model, they will be ignored. (Other dialog buttons with other Action settings do rely on the application model for their action methods, however.)

V2. Storing the Dialog's Builder for Later Use

Online example: DialogExample

1. In the method that is to open the dialog, create an instance of `SimpleDialog`.
2. Get the builder from the `SimpleDialog` and store it, typically in an instance variable of the application model (`dialogBuilder`).
3. Send an `openFor:interface:` message to the `SimpleDialog`. The first argument is the application model so that the dialog's widgets can obtain their value models, actions, and resources from it. The second argument is the name of the dialog's interface specification.

```

openDialogStoreBuilder
  | returnVal dialogModel |
  dialogModel := SimpleDialog new.           "V2 Step 1"
  self dialogBuilder: dialogModel builder.   "V2 Step 2"

  returnVal := dialogModel                   "V2 Step 3"
  openFor: self
  interface: #memoryZonesDialog.

  "Update the text field in the main window."
  self returnedValue value: returnVal printString.

```

V3. Providing a Temporary Model for the Dialog

Online example: DialogExample

In the example, the properties you set for the dialog's list widget tell the dialog's builder that the list widget needs a `MultiSelectionInList` to supply its value holders. In the other variants, the builder obtains the required `MultiSelectionInList` by sending the `memoryZones` aspect message to the application model. In this variant, the builder does not need to send this message, because it has been preconfigured with the required `MultiSelectionInList` through an `aspectAt:put:` message.

1. In the method that is to open the dialog, create an instance of `SimpleDialog`.
2. Get the builder from the `SimpleModel` and preload it with one binding for each active widget. The `aspectAt:` argument is the symbol you specified in the widget's `Aspect` property. The `put:` argument is an appropriate value model.
3. Ask the `SimpleDialog` to open the interface.

```

openTempModelDialog
  | returnVal dialogModel list |
  dialogModel := SimpleDialog new.           "V3 Step 1"
  dialogBuilder := dialogModel builder.

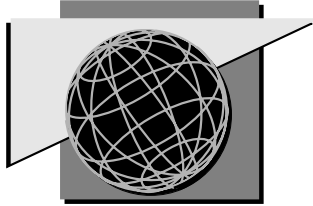
  "Since the simple model does not respond to a #memoryZones message,
  its builder must be preloaded with a multilist."
  list := MultiSelectionInList new

```

```
list: self memoryZones list copy.  
dialogBuilder aspectAt: #memoryZones put: list. "V3 Step 2"
```

```
"Open the interface."  
returnVal := dialogModel  
openFor: self  
interface: #memoryZonesDialog. "V3 Step 3"
```

```
"Update the text field in the main window."  
self returnedValue value: returnVal printString.
```



Chapter 16

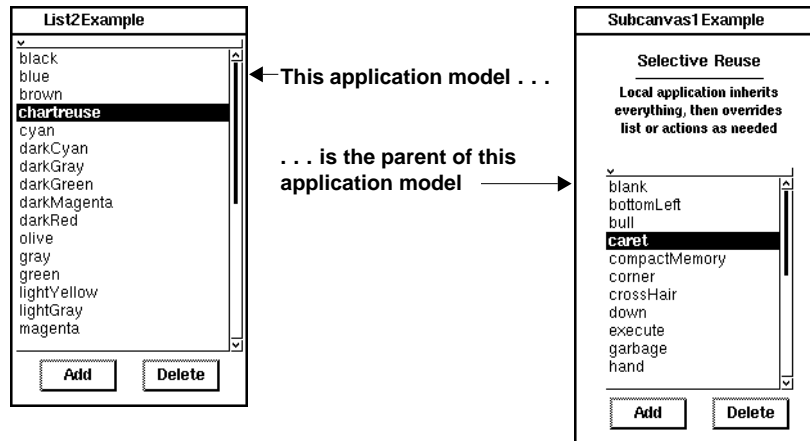
Subcanvases

Inheriting an Application's Capabilities	302
Nesting One Application in Another	305
Reusing an Interface Only	308
Swapping Interfaces at Run Time	310
Accessing an Embedded Widget	313

See Also

- “Widget Basics” on page 53

Inheriting an Application's Capabilities



Strategy

The `ApplicationModel` class provides a wealth of functionality that is inherited by any subclass, which is why you must make any new application model a subclass of `ApplicationModel`. In the same way, you can use your own subclass as a parent class, so that its children will inherit standard interface modules, value holders, and action methods. For example, `Subcanvas1Example` is a subclass of `List2Example`, so it can reuse the `List2Example` interface, value holders, and actions.

Overriding actions is possible: Although a subclass need not reimplement anything that the parent class has implemented, it *can* override an inherited action. (That is not always possible when you nest one application inside another without the aid of inheritance.)

No multiples: A limitation of the inheritance approach is that you cannot reuse an inherited interface more than once on the same canvas. For example, `Subcanvas1Example` could not use two subcanvases that each contained the same inherited `List2Example` interface, because both would reference the same value holder (`selectionInList`). (More precisely, you *can* use the same inherited interface twice, but both will display the same thing.)

Basic Steps

Online example: List2Example (parent) and Subcanvas1Example

1. Use a System Browser to create a new application model (Subcanvas1Example) as a subclass of the application model from which it is to inherit (List2Example).
2. Use a Palette to place a subcanvas widget on the inheriting canvas (the canvas for Subcanvas1Example). Leave the subcanvas widget selected.
3. In the subcanvas's Canvas property, enter the name of the inherited interface specification to be used by the subcanvas (listSpec). This name must be unique within the inheritance chain—for example, you could not embed an inherited canvas named windowSpec in a local canvas named windowSpec.
4. Apply the property and install the inheriting canvas in its application model (Subcanvas1Example).

Variants

V1. Installing a Different Value in an Inherited Widget

The power of reuse is fully realized when you provide local values for the inherited widgets. For example, List2Example initializes its list to display a collection of color names. Now the inheriting application, Subcanvas1Example, provides its own collection, causing the reused list to display cursor names instead.

1. Use a System Browser to create an initialize method in the inheriting application model (Subcanvas1Example).
2. In the initialize method, invoke the inherited initialize method.
3. In the initialize method, use the inherited aspect message (selectionInList) to access the desired valued model. Then send an accessing message (in this case, list:) to the value model to install the desired value (cursorNames).

initialize

```
"Install a different list (cursor names) than
the inherited default (color names)."
```

```
| cursorNames |
super initialize.
```

"V1 Step 2"

```
cursorNames := Cursor class organization
listAtCategoryNamed: #constants.
self selectionInList list: cursorNames. "V1 Step 3"
```

V2. Overriding an Inherited Action Method

- In the inheriting application model (Subcanvas1Example), create a method with the same name as the inherited method that you want to override (add).

```
add "V2Step"
    "Override the inherited implementation of this method,
    refining the prompt in the dialog."

    | entry newList |

    "Prompt for the name to add."
    entry := Dialog request: 'Add cursor name'.

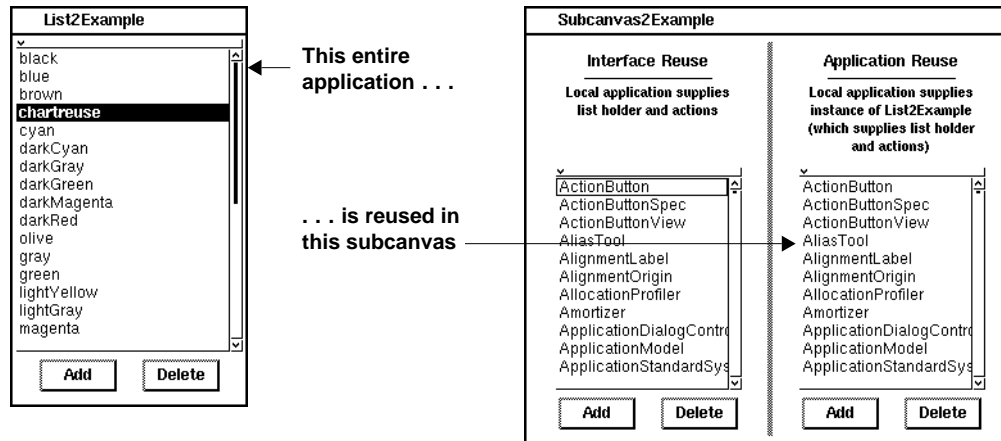
    "If the entry is blank, exit."
    entry isEmpty
        ifTrue: [^nil].

    "Update the list."
    newList := SortedCollection withAll: self selectionInList list.
    newList add: entry.
    self selectionInList list: newList.
```

See Also

- “Creating a Class (Subclassing)” on page 26

Nesting One Application in Another



Strategy

With a subcanvas, you can embed one application in another. In this way, you can create a set of application modules that can be plugged into larger applications as needed. This approach avoids wasteful duplication of effort for generic modules, enforces interface-design uniformity, and makes changes much easier to implement, because you have to change only the embedded application to effect a change in all reusing applications.

Overriding actions is not possible: The embedded application supplies all of its own value models and action methods. This feature makes it simple to implement but slightly more difficult to customize than an application with inherited capabilities. In particular, you cannot override an embedded application's action methods. In truly generic modules, however, this is not a serious limitation.

Multiples are possible: You can embed the same application any number of times in the same canvas. For example, you could reuse List2Example four times in creating a System Browser's four list views.

Basic Steps

Online example: List2Example embedded in Subcanvas2Example

1. Use a Palette to place a subcanvas in the reusing canvas (the canvas for Subcanvas2Example). Leave the subcanvas selected.
2. In the subcanvas's Name property, enter the name of the method (classNames) that will supply an instance of the embedded application.
3. In the subcanvas's Class property, enter the name of the application (List2Example) that you are embedding.
4. In the subcanvas's Canvas property, enter the name of the interface specification (listSpec) that you are using from the embedded application (List2Example).
5. Apply the properties and install the reusing canvas in its application model (Subcanvas2Example).
6. Use a System Browser to create an instance variable (classNames) in the reusing application model (Subcanvas2Example), for holding onto the embedded application.
7. Use a System Browser to create an initialize method in the reusing application model, in which the embedded application is created and assigned to the variable that you created in step 6.

initialize

"Reusing List2Example's interface only -- initialize the list holder."
selectionInList := SelectionInList with: Smalltalk classNames.

"Reusing List2Example application -- initialize the application instance."
classNames := List2Example new. "Basic Step 7"
classNames list: Smalltalk classNames.

Variant

Installing a Different Value in an Embedded Widget

An embedded widget uses the value with which its host application initializes it.

1. In the initialize method of basic step 7, send a message (list:) to the embedded application, installing the desired value.

initialize

```
"Reusing List2Example's interface only -- initialize the list holder."  
selectionInList := SelectionInList with: Smalltalk classNames.
```

```
"Reusing List2Example application -- initialize the application instance."  
classNames := List2Example new.
```

```
classNames list: Smalltalk classNames. "Variant Step 1"
```

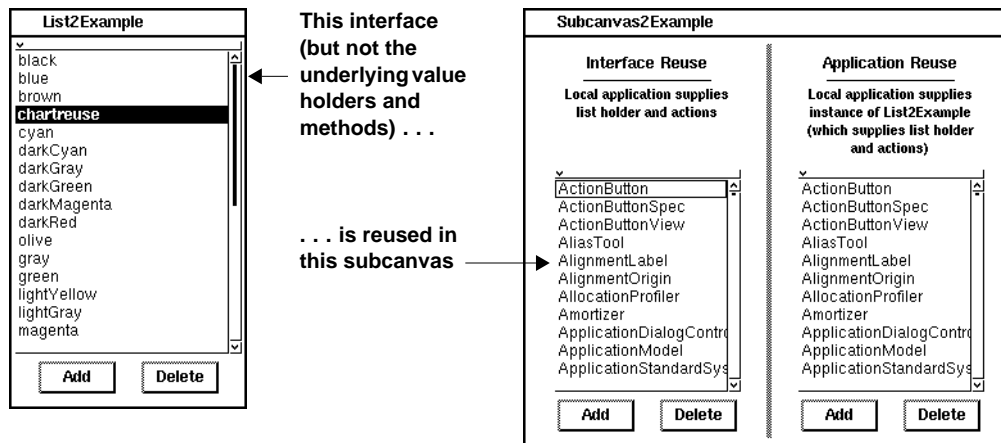
2. In some situations, as in the example, you will have to create a method (list:) in the embedded application model that enables an outside application to supply a new value.

list: aCollection "Variant Step 2"

```
"Install aCollection in the list. This message is provided so reusers  
can install a list that is different than the default list (color names)."
```

```
self selectionInList list: aCollection.
```

Reusing an Interface Only



Strategy

You can use a subcanvas to embed one canvas inside another. This is similar to embedding an entire subapplication, but the difference is that all value models and methods must be supplied by the reusing application. This is duplicative, but it is sometimes necessary, especially when you need to override action methods.

Overriding actions is possible: Because you are reusing only the interface and have to reimplement all of the supporting value holders and methods, you also have to supply actions for any buttons in the embedded interface.

Multiples are not possible: Because you are forced to use the aspect names that the embedded interface expects, you can have only one set of those names. So you cannot reuse an interface more than once on the same canvas.

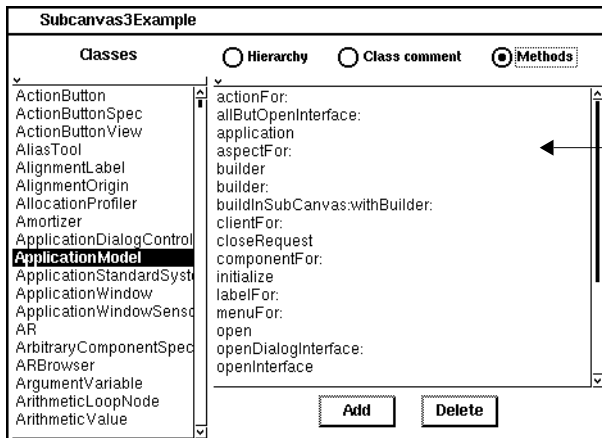
Basic Steps

Online example: Subcanvas2Example (which reuses List2Example's listSpec)

1. Use a Palette to place a subcanvas in the reusing canvas (the canvas for Subcanvas2Example).

2. In the subcanvas's **Class** property, enter the name of the application (List2Example) that defines the interface to be embedded.
3. In the subcanvas's **Canvas** property, enter the name of the interface specification (listSpec) to be embedded.
4. Apply the properties and install the reusing canvas in its application model (Subcanvas2Example).
5. Use a **System Browser** to edit the reusing application model (Subcanvas2Example), **creating instance variables** (selectionInList) **and methods** (selectionInList, initialize, add, and delete) **to support the embedded interface. These instance variables and methods must have the same names as the corresponding ones in the reused class (List2Example). Modify values and action methods as desired.**

Swapping Interfaces at Run Time



This subcanvas holds a List2Editor when the Methods button is chosen and an Editor2Example when another button is chosen

Strategy

A subcanvas makes it easy to change the widgets that appear in a larger canvas, depending on the circumstances. In Subcanvas3Example, a subcanvas is used to hold either a text editor or a list view, depending on whether the user wants to see textual or listed material related to a selected class.

An alternative approach is to layer the multiple sets of widgets in the main canvas (without using subcanvases at all) and then make the desired widgets visible as needed.

Basic Steps

Online example: Subcanvas3Example (which swaps Editor2Example and List2Example)

1. Use a Palette to place a subcanvas in the reusing canvas (the canvas for Subcanvas3Example).
2. In the subcanvas's Name property, enter the name of the method (embeddedApplication) that supplies the embedded application at startup time.
3. Apply the properties and install the reusing canvas in its application model (Subcanvas3Example).

-
4. Use a System Browser to create the method (embeddedApplication) that you named in step 2. You create this method in the reusing application model (Subcanvas3Example). This method can supply either a nil value (for a blank subcanvas) or one of the subapplications.

```
embeddedApplication                                "Basic Step 4"
  ^nil asValue
```

5. In a change message (presumably triggered by a change in some other widget such as a button), create an instance of the desired application model (Editor2Example) and initialize it. (Or you can create and initialize the application model once at startup and store it in an instance variable.)
6. Continuing in the change message, get the spec object for the interface you want to use by sending an interfaceSpecFor: message to the embedded application model's class (Editor2Example). The argument is the name of the interface specification (#windowSpec).
7. Continuing in the change message, get the subcanvas from the builder and send a client:spec: message to it. The first argument is the application you created in step 5. The second argument is the spec object you obtained in step 6.

showComment

```
| selectedClass subcanvas spec application |
selectedClass := Smalltalk at: self classNames selection.
```

```
"Create the subapplication and initialize it."                                "Basic Step 5"
application := Editor2Example new.
application text value: selectedClass comment.
```

```
"Get the spec object for the embedded canvas."
spec := Editor2Example interfaceSpecFor: #windowSpec.                        "Basic Step 6"
```

```
"Get the subcanvas and install the editing application."                    "Basic Step 7"
subcanvas := (self builder componentAt: #subcanvas) widget.
subcanvas client: application spec: spec.
```

Variants

Blanking the Subcanvas

In `Subcanvas3Example`, the subcanvas goes blank when no class is selected. You may encounter a similar situation that requires you to empty a subcanvas at run time.

- Get the subcanvas from the builder and send a client: message to it. The argument is `nil`.

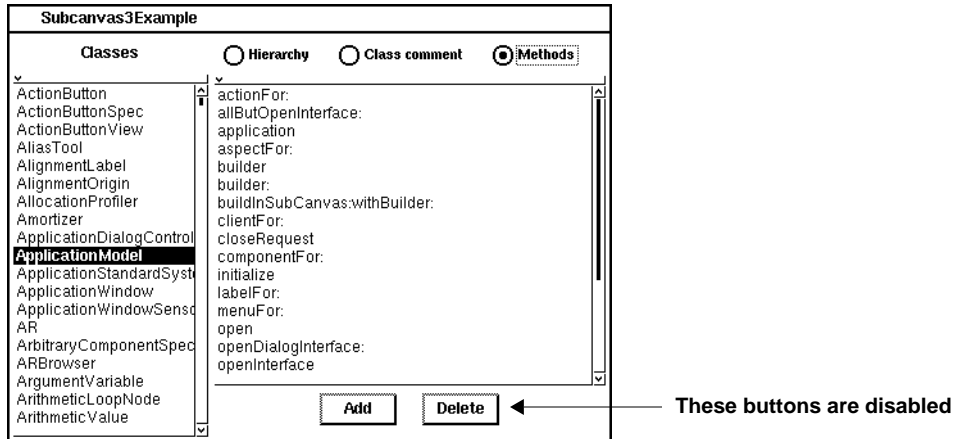
`showNothing`

```
| subcanvas |  
subcanvas := (self builder componentAt: #subcanvas) widget. "Variant Step"  
subcanvas client: nil.
```

See Also

- “Hiding a Widget” on page 70

Accessing an Embedded Widget



Strategy

Frequently an embedded or inherited interface contains more than you need. For example, when an embedded action button is not appropriate in the local application, you could make it invisible or disable it. Before you can manipulate embedded widgets, however, you need to access them.

Basic Steps

Online example: Subcanvas3Example

1. Before installing the new subapplication using `client:spec;`, initialize the subapplication's builder to `nil`. (Otherwise, the subapplication will continue to hold the old builder even after a new builder is created to assemble the new interface.)
2. Ask the subapplication for its builder and then send `componentAt:` to that builder. The argument is the ID of the desired widget.

showMethods

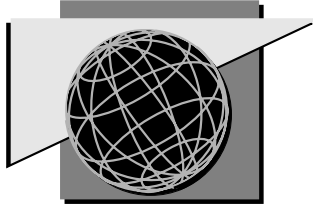
```
| selectedClass subcanvas spec |
selectedClass := Smalltalk at: self classNames selection.
spec := List2Example interfaceSpecFor: #listSpec.
```

"Install the method names as the collection in the list application."
self listApplication list: selectedClass selectors asSortedCollection.

"Set the subbuilder to nil to discard the old builder. This is only
necessary when the application uses the builder later to access widgets."
listApplication builder: nil. "Basic Step 1"

"Get the subcanvas and install the list application."
subcanvas := (self builder componentAt: #subcanvas) widget.
subcanvas client: listApplication spec: spec.

"Disable the embedded buttons (just to show that we can)."
(listApplication builder componentAt: #addButton) disable. "Basic Step 2"
(listApplication builder componentAt: #deleteButton) disable.



Chapter 17

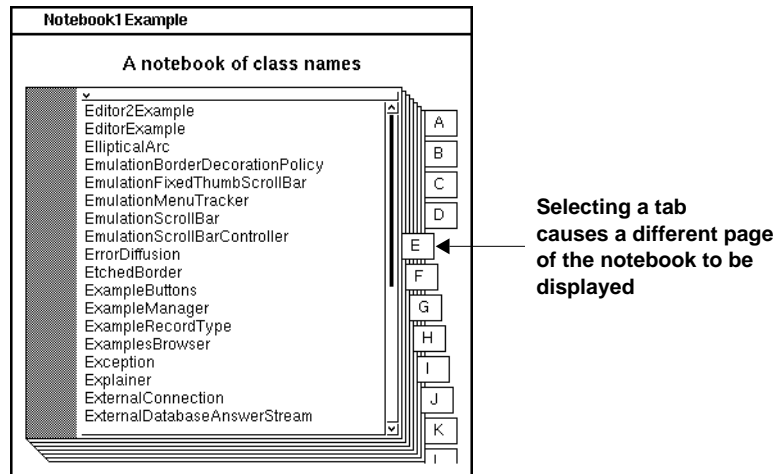
Notebooks

Adding a Notebook	316
Determining Which Tab Is Selected	319
Changing the Binding's Appearance	322
Changing the Size and Axis of the Tabs	324
Setting the Starting Page	326
Adding Secondary Tabs (Minor Keys)	328
Connecting Minor Tabs to Major Tabs	331
Changing the Page Layout (Subcanvas)	334
Connecting a Notebook to a Text Editor	336

See Also

- “Widget Basics” on page 53

Adding a Notebook



Strategy

A notebook is a powerful navigational widget. At its simplest, as shown here, it provides a list in the form of index tabs. When the user selects an index tab, the effect is the same as selecting an item in a conventional list—in fact, both a list and a notebook's tabs use a `SelectionInList` to provide their value models. A notebook can be used in many of the same situations in which a list or a menu might be used, though its richer set of capabilities (such as minor keys) extend its range of uses.

A notebook also contains a subcanvas. This subcanvas can be used to display a different interface for each index tab or, as in this simple example, the same interface. In `Notebook1Example`, the subcanvas contains a list widget, and the list is changed each time an index tab is selected.

Basic Steps

Online example: `Notebook1Example`

1. Use a Palette to paint a notebook widget on your canvas. Leave the notebook selected.

2. In a Properties Tool (Basics page), fill in the notebook's Major property with the name of the method (majorKeys) that returns a SelectionInList containing the labels for the index tabs.
3. In the notebook's ID property, enter an identifying name (pageHolder).
4. Apply the properties and install the canvas.
5. Create a second canvas for the interface that is to be displayed inside the notebook. Install this canvas in its own resource method (listSpec).
6. Use a System Browser or the canvas's define command to create the instance variable (majorKeys) and accessing method (majorKeys) for the notebook's list of index labels. Initialize the variable, either in the accessing method or in an initialize method (as in the example), with a SelectionInList containing either strings or associations.

```
majorKeys                                     "Basic Step 6"
  ^majorKeys
```

7. Use a System Browser or the canvas's define command to create any variables and methods needed by the subcanvas. (In the example, these are the classNames variable, the classNames method, and the initialize method.)

```
classNames                                     "Basic Step 7"
  ^classNames
```

8. In the initialize method, use an onChangeSend:to: message to arrange for the notebook to send a message (changedLetter) to the application model when the user selects an index tab.

```
initialize
| letters |
letters := #(' A' ' B' ' C' ' D' ' E' ' F' ' G' ' H' ' I' ' J' ' K' ' L' ' M'
            ' N' ' O' ' P' ' Q' ' R' ' S' ' T' ' U' ' V' ' W' ' X' ' Y' ' Z').
majorKeys := SelectionInList with: letters.                                     "Basic Step 6"
majorKeys selectionIndexHolder
  onChangeSend: #changedLetter to: self.                                       "Basic Step 8"
classNames := SelectionInList new.                                             "Basic Step 7"
```

9. Create the change message (`changedLetter`) in which the subcanvas is updated based on the index tab that has been selected. (In the example, the `classNames` list is updated with classes beginning with the letter on the index tab.)

```
changedLetter                                     "Basic Step 9"  
  | chosenLetter list |  
  chosenLetter := self majorKeys selection last.  
  list := Smalltalk classNames select: [ :name | name first == chosenLetter].  
  self classNames list: list.
```

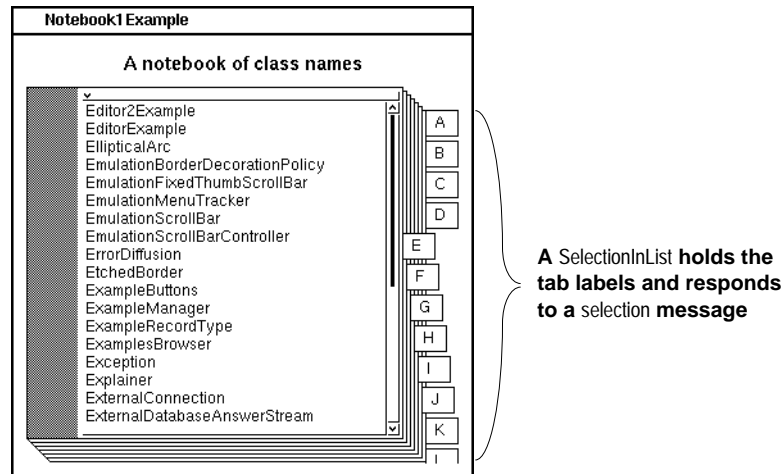
10. Create a `postOpenWith:` method. In this method, first get the notebook from the application model's builder, using the notebook's ID (`pageHolder`). Then install the subcanvas by sending a `client:spec:` message to the notebook. The first argument is the subapplication's application model (in the example, `self`). The second argument is the name of the spec method (`listSpec`) that defines the desired canvas.

```
postOpenWith: aBuilder                             "Basic Step 10"  
  (aBuilder componentAt: #pageHolder) widget  
  client: self  
  spec: #listSpec.  
  majorKeys selectionIndex: 1.
```

See Also

- “Adding a List” on page 184
- “Creating a Collection” on page 491

Determining Which Tab Is Selected



Strategy

When the user selects an index tab on a notebook widget, the selection changes in the underlying `SelectionInList`. Accessing that selection is a fundamental operation because the application model must know which tab is selected before it can take the appropriate action.

The basic step shows how to access the label on the index tab.

The first variant shows how to access an object that has been associated with the selected index tab. This assumes that you have associated an object with each label, much as a menu does. The associated object is typically a `Symbol` that identifies a method to be performed, a canvas to be installed, or an application-specific attribute.

The second variant shows how to access the relative position of the index tab. The resulting index number can be used to find the appropriate object in a separate collection. Because the separate collection can be changing dynamically, this approach is one way to vary the action associated with each index tab.

Basic Step

Online example: Notebook1Example

- In a method in the application model, get the selected index tab's string or association by sending a selection message to the notebook's major SelectionInList. (In the example, the resulting string contains a leading space, so a last message is sent to get the index letter that follows the space).

```
changedLetter
| chosenLetter list |
chosenLetter := self majorKeys selection last.           "Basic Step"
list := Smalltalk classNames
    select: [ :name | name first == chosenLetter].
self classNames list: list.
```

Variants

V1. Getting a Value Associated with an Index Tab

Online example: Notebook2Example

1. In a method in the application model, get the selected tab's association by sending a selection message to the SelectionInList (in the example, minorKeys, to which the initialize method assigned a SelectionInList with a collection of associations.)
2. Send a value message to the resulting association. (In the example, the value is a Symbol—#all or #examples—which is used to filter the list of class names.)

```
changedPage
| chosenLetter list filter filteredList |
chosenLetter := self majorKeys selection last.
filter := self minorKeys selection value.                 "V1 Step 2"

list := Smalltalk classNames
    select: [ :name | name first == chosenLetter].

filter == #all
    ifTrue: [filteredList := list]
    ifFalse: [filteredList := list]
```

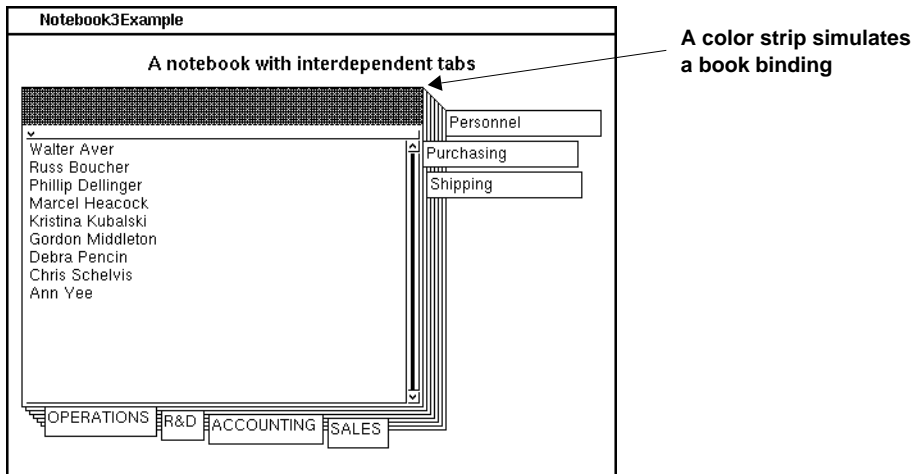
```
select: [ :name | '**Example' match: name]].
```

```
self classNames list: filteredList.
```

V2. Getting the Index Number of the Tab

- **Send selectionIndex to the SelectionInList (instead of a selection message).**

Changing the Binding's Appearance



Strategy

A solid color strip at the left or top edge of a notebook is used to simulate the appearance of a book binding. By default, the binding is along the left edge—the first variant shows how to move it to the top edge.

By default, the binding strip is 18 pixels wide. The variant shows how to change the width of the binding. A width of zero can be used to eliminate the binding strip altogether.

Basic Step

Changing the Location

Online example: Notebook3Example

1. Select the notebook in the canvas.
2. In Properties Tool, go to the notebook's Binding property and select top. This moves the binding to the top edge. (To move it back, select left.)
3. Apply properties and install the canvas.

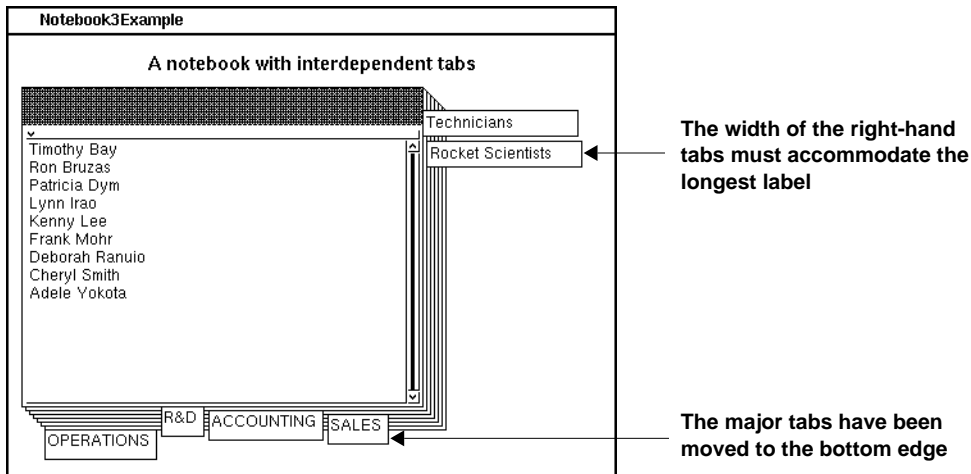
Variant

Changing the Width

Online example: Notebook3Example

- ▶ In the **Width** field of the notebook's **Binding** property, enter the desired number of pixels of width (in the example, 30). A zero setting makes the binding disappear.

Changing the Size and Axis of the Tabs



Strategy

By default, the major index tabs are aligned along the right-hand edge of the notebook, and the minor tabs are along the bottom. The basic step shows how to reverse that orientation.

By default, the right-hand tabs are 60 pixels wide and the bottom tabs are 24 pixels high. These values are called *insets*, because the notebook page is inset by those amounts from the widget's allotted area. When a label does not fit in those insets, the user cannot see the end of the label. The second variant shows how to adjust the insets to allow for the longest label on the right. (Allowing for the highest label on the bottom is less often a concern, unless you are using a nonstandard font.)

Basic Step

Changing the Axis

Online example: Notebook3Example

1. Select the notebook in the canvas.

2. In the Properties Tool, go to the notebook's **Major Tabs** property and select **bottom**. (The minor tabs, if any, will move to the right-hand edge.)
3. Apply the property and install the canvas.

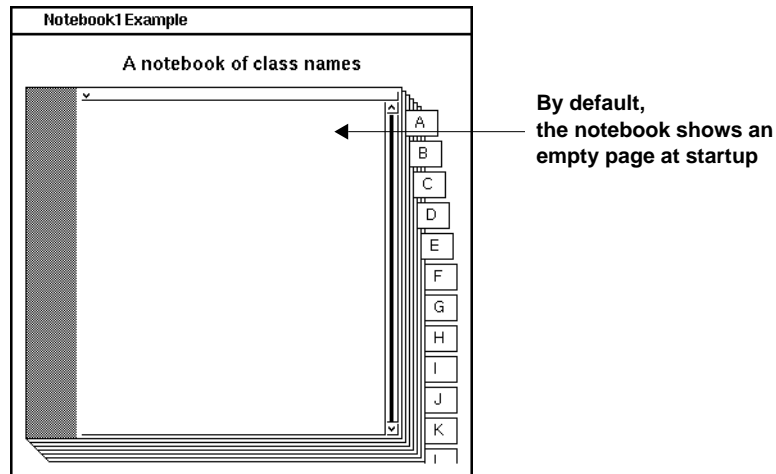
Variant

Setting the Size

Online example: [Notebook3Example](#)

1. In **Right** field of the notebook's **Insets** property, enter the number of pixels of width for the right-hand index tabs. (The height adjusts automatically to fit the label font).
2. In **Bottom** field of the notebook's **Insets** property, enter the number of pixels of height for the bottom tabs. (The width adjusts to fit each tab's label.)

Setting the Starting Page



Strategy

By default, a notebook opens with a blank page showing. This can be regarded as the cover of the notebook, and it is properly left blank if choosing a particular page to display at startup would be arbitrary and therefore confusing. However, displaying a nonblank page often provides better visual clues to the user as to the nature of the notebook. The basic steps show how to choose a default page by setting the selection indexes of the major and minor `SelectionInLists`.

The variant shows how to set the selection by specifying the list element itself (rather than the index number of that element). This approach is more convenient when your application has held onto the list element from an earlier operation.

Basic Steps

Online example: `Notebook1Example`

1. In a method in the application model (such as `postOpenWith()`), send a `selectionIndex:` message to the `SelectionInList` that holds the major keys (in the example, `majorKeys`). The argument is the index number of the desired tab in the list.

```
postOpenWith: aBuilder
  (aBuilder componentAt: #pageHolder) widget
  client: self
  spec: #listSpec.
  majorKeys selectionIndex: 1.                                     "Basic Step 1"
```

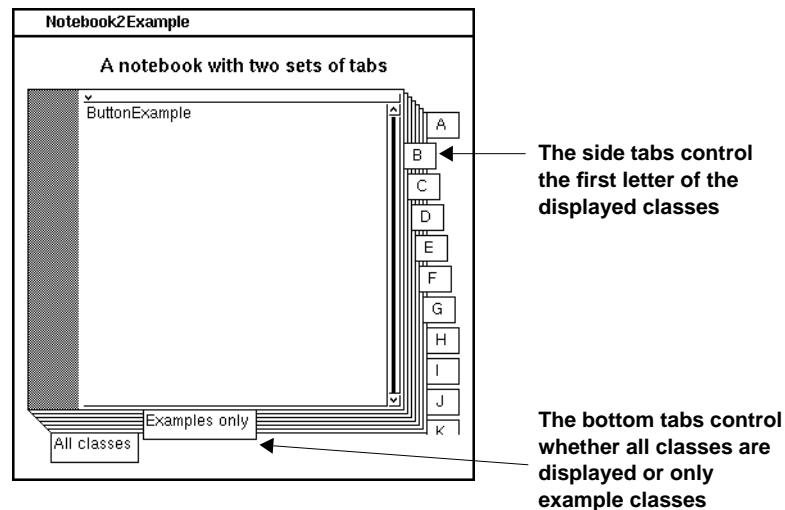
2. If the notebook has minor keys, also set the selection index in the SelectionInList that contains the minor keys.

Variant

Setting the Page by Specifying the List Element

- Send a selection: message to the SelectionInList that holds the major keys and, if applicable, another such message to the minor list. The argument is the desired element in the list.

Adding Secondary Tabs (Minor Keys)



Strategy

In addition to the first set of index tabs, a second row of tabs can be added along another edge of the notebook. This second set of tabs is referred to as the *minor keys*. The minor keys can be used either to refine the subdivisions implied by the major keys or to filter the content of the notebook along a separate dimension.

In Notebook2Example, which lists the classes in the system, two minor keys are used to control whether the page shows all classes beginning with the selected letter or just the example classes.

Basic Steps

Online example: Notebook2Example

1. Select the notebook in the canvas.
1. In the Properties Tool, fill in the notebook's **Minor** property with the name of the method that returns a `SelectionInList` containing the labels for the secondary tabs (in the example, `minorKeys`).

2. Use a System Browser or the canvas's define command to create the instance variable (minorKeys) and accessing method (minorKeys) for the notebook's list of index labels.

```
minorKeys                                     "Basic Step 2"
  ^minorKeys
```

3. Initialize the variable, either in the accessing method or in an initialize method (as in the example), with a SelectionInList containing either strings or associations (the example uses associations).
4. In the initialize method, use an onChangeSend:to: message to arrange for the notebook to send a message (changedPage) to the application model when the user selects a secondary tab. (In the example, both the major and minor tabs trigger the same message: changedPage.)

```
initialize
| letters |
letters := #('A''B''C''D''E''F''G''H''I''J''K''L''M'
            'N''O''P''Q''R''S''T''U''V''W''X''Y''Z').
majorKeys := SelectionInList with: letters.
majorKeys selectionIndexHolder
  onChangeSend: #changedPage to: self.

minorKeys := SelectionInList with: (Array
  with: 'All classes'-> #all
  with: 'Examples only' -> #examples).
minorKeys selectionIndexHolder
  onChangeSend: #changedPage to: self.                                     "Basic Step 4"

classNames := SelectionInList new.
```

5. Create the change message (changedPage) in which the subcanvas is updated based on the selected index tab. (In the example, the classNames list is updated with all class names or only example classes, based on the minor key.)

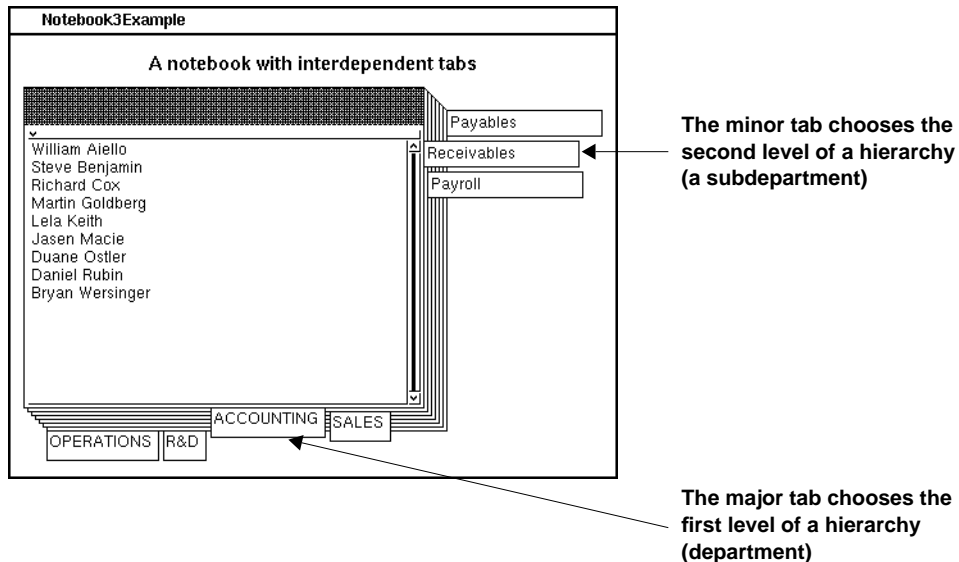
```
changedPage
| chosenLetter list filter filteredList |
```

```
"Major key."  
chosenLetter := self majorKeys selection last.  
list := Smalltalk classNames  
  select: [ :name | name first == chosenLetter].
```

```
"Minor key."                                     "Basic Step 5"  
filter := self minorKeys selection value.  
filter == #all  
  ifTrue: [filteredList := list]  
  ifFalse: [filteredList := list  
    select: [ :name | '**Example' match: name]].
```

```
self classNames list: filteredList.
```

Connecting Minor Tabs to Major Tabs



Strategy

The major and minor keys of a notebook can be used to navigate through a two-tiered hierarchy of information. The minor keys depend on the major keys—that is, when the user selects a different major key, a different set of minor keys is presented.

In Notebook3Example, the major keys represent departments in a company. The minor keys represent subdepartments, and they change depending on which department is selected. (Compare this with Notebook2Example, in which the minor keys remain unchanged when a new major key is selected.)

Basic Steps

Online example: Notebook3Example

1. In the application model's initialize method, use `onChangeSendTo:` messages to arrange for the two `SelectionInLists` to notify the application model when their selections are changed.

initialize

```
self initializeDepartments.
self initializeEmployees.
```

```
majorKeys := SelectionInList with: departments keys asArray.
majorKeys selectionIndexHolder
  onChangeSend: #changedDepartment
  to: self.                                     "Basic Step 1"
```

```
minorKeys := SelectionInList new.
minorKeys selectionIndexHolder
  onChangeSend: #changedSubdepartment
  to: self.                                     "Basic Step 1"
```

```
employeeList := SelectionInList new.
```

2. **Create the change message (changedDepartment) for the major keys. In this method, get the selection from the major SelectionInList and verify that it is not nil. Then use that selection as the basis for choosing the new labels for the minor tabs. Typically, this method will also reset the minor key selection so the notebook displays the first subpage.**

changedDepartment "Basic Step 2"

```
| subdepts sel |
sel := self majorKeys selection.
sel isNil ifTrue: [^self].
```

```
"Display the appropriate subdepartments as minor keys."
subdepts := self departments at: sel.
self minorKeys list: subdepts.
self minorKeys selectionIndex: 1.
```

3. **Create the change message (changedSubdepartment) for the minor keys. In this method, get the minor selection and verify that it is not nil. Then use that selection as the basis for updating the canvas in the notebook or, as in the example, its model (employeeList).**

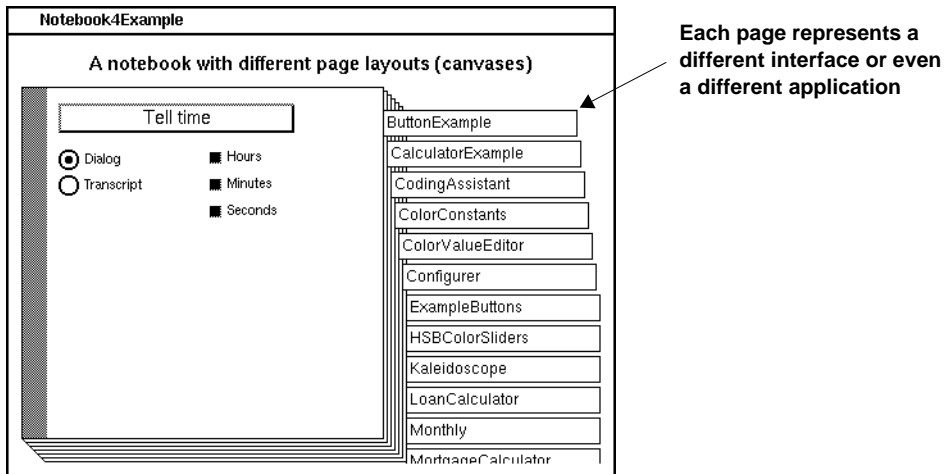
changedSubdepartment "Basic Step 3"

```
"Display the appropriate employees in the list."
```

```
| emps sel |  
sel := self minorKeys selection.  
sel isNil ifTrue: [^self].
```

```
emps := self employees at: sel.  
self employeeList list: emps.
```

Changing the Page Layout (Subcanvas)



Strategy

In some applications, a notebook can be used to present a different interface on each page. This approach can be used as an alternative to placing each subinterface in its own window, and thus it reduces the number of windows cluttering the application user's screen. There is some loss of convenience, however, which is especially noticeable when the user might want to view two subinterfaces simultaneously.

In Notebook4Example, each index tab represents an example application. Selecting a tab causes a working instance of that application to be contained in the notebook.

Basic Steps

Online example: Notebook4Example

1. In the application model's initialize method, arrange for the SelectionInList that holds the major keys to notify the application model when a tab is selected. (In the example, a changedExample message is triggered.)

```
initialize
  | exampleClasses |
```

```
exampleClasses := OrderedCollection new.
exampleClasses := Smalltalk keys select: [ :c |
    (**Example' match: c)
    and: [(Smalltalk at: c) isVisualStartable])
    and: [(Notebook** match: c) not]].
majorKeys := SelectionInList
with: exampleClasses asSortedCollection.
```

```
majorKeys selectionIndexHolder
onChangeSend: #changedExample to: self. "Basic Step 1"
```

2. In the `change` method (`changedExample`), get the notebook widget from the application model's builder by sending a `componentAt:` message.
3. Still in the `change` method, send a `client:spec:` message to the notebook. The first argument is an instance of the desired application model (in the example, `exampleClass`). The second argument is the name of the desired canvas (in the example, each example class's `windowSpec` is used).

changedExample

```
| sel exampleClass |
sel := self majorKeys selection.
sel isNil ifTrue: [^self].
```

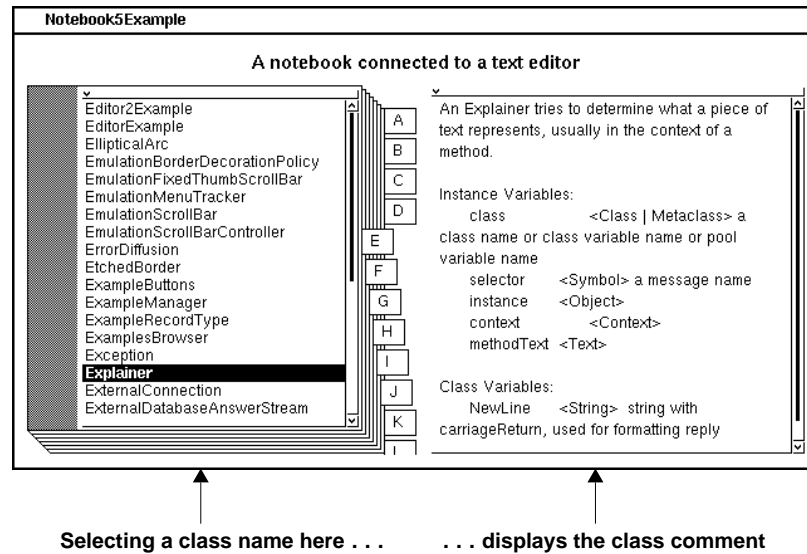
```
exampleClass := Smalltalk at: sel value.
```

```
(self builder componentAt: #pageHolder) widget "Basic Step 2"
client: exampleClass new "Basic Step 3"
spec: #windowSpec.
```

See Also

- “Nesting One Application in Another” on page 305

Connecting a Notebook to a Text Editor



Strategy

A common use for a notebook is for navigating through a set of textual pages. The text editor can be separate from the notebook or displayed as a subcanvas within the notebook. Using a separate text editor permits the user of your application to see the navigational hierarchy and the text at the same time.

In Notebook5Example, a notebook is used for finding a class. The text editor displays the class comment for the selected class.

Since the major and minor keys of a notebook can represent two levels of a hierarchy, a list widget inside the notebook can represent a third level. Thus, a notebook is a convenient means of navigating up to three levels deep in a hierarchy. The text editor is typically connected to the lowest level of the hierarchy—in the example, the classNames list that is displayed inside the notebook.

Basic Steps

Online example: Notebook5Example

1. In the application model's initialize method, arrange for the SelectionInList that holds the lowest level of selections to notify the application model when a text is selected. (In the example, a changedClass message is triggered by a selection in the classNames list.)

```
initialize
| letters |
letters := #(' A' ' B' ' C' ' D' ' E' ' F' ' G' ' H' ' I' ' J' ' K' ' L' ' M'
            ' N' ' O' ' P' ' Q' ' R' ' S' ' T' ' U' ' V' ' W' ' X' ' Y' ' Z').
majorKeys := SelectionInList with: letters.
majorKeys selectionIndexHolder
    onChangeSend: #changedLetter to: self.

classNames := SelectionInList new.
classNames selectionIndexHolder
    onChangeSend: #changedClass to: self.           "Basic Step 1"

classComment := " asValue.
```

2. In the change method (changedClass), get the selection from the lowest-level SelectionInList (classNames). Then use that selection as the basis for choosing the text and install the new text in the text holder (classComment).

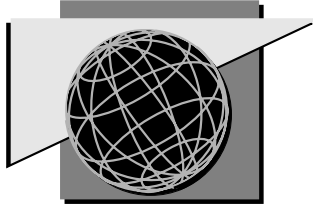
```
changedClass
| chosenClass newText |
chosenClass := self classNames selection.           "Basic Step 2"

newText := chosenClass isNil
    ifTrue: [""]
    ifFalse: [(Smalltalk at: chosenClass) comment].

self classComment value: newText.
```

See Also

- “Adding a Text Editor” on page 172

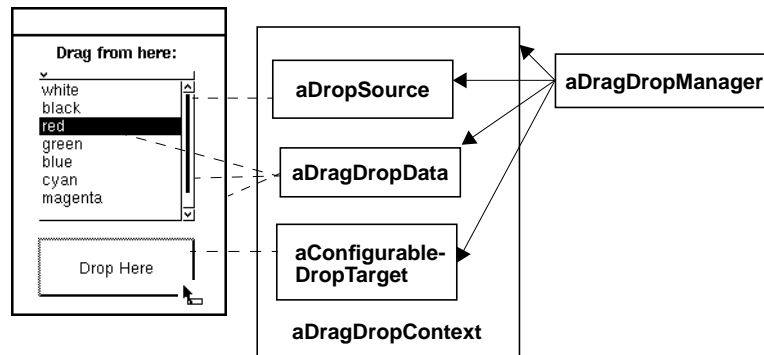


Chapter 18

Drag and Drop

About Drag and Drop	340
Adding a Drop Source	343
Adding a Drop Target (General)	348
Providing Visual Feedback During a Drag	350
Responding to a Drop	359
Examining the Drag Context	365
Responding to Modifier Keys	366
Defining Custom Effect Symbols	371

About Drag and Drop



Strategy

You can program application interfaces to enable users to transfer data using *drag and drop*. A user performs drag and drop by using the mouse pointer to:

1. Grab the object to be transferred
2. Drag that object to another location on the screen
3. Release (drop) the object there

For example, a user might copy an object such as a file by dragging that object to a target location and dropping it there. Throughout such an interaction, the pointer typically changes shape to indicate whether it is over a valid target location.

You arrange for drag and drop by setting up one or more widgets to be *drop sources* (widgets that present data to be transferred) and/or *drop targets* (widgets that respond in some way to the transferred data). Currently, a drop source must be a list widget, while a drop target can be any widget except a linked or embedded dataform. Drop sources and drop targets may be in the same interface, or they may be in the interfaces of different applications.

As shown in the topics in this chapter, you set up drop sources and drop targets by specifying various message names in their properties, and then programming the relevant application model(s) to respond to these messages.

Drag and drop framework classes. In a running application, a drag-and-drop interaction is carried out by instances of several framework classes. Some of these instances are created as a result of the code you write, while others are created automatically when the interface is built or when drag and drop is underway. Each drag-and-drop interaction involves:

- An instance of `DragDropData`, which holds the data to be transferred, plus information about where the drag originated (the widget's controller, the containing window, and the associated application model).
- An instance of `DropSource`, which defines the shapes that the pointer can have during a drag originating from this drop source. By default, the `DropSource` defines pointer shapes for signaling whether a move, a copy, or no transfer would take place if a drop were to occur at any point during the drag.
- An instance of `DragDropManager`, which tracks the mouse pointer throughout the drag. When the pointer encounters a potential drop target, the `DragDropManager` sends messages to find out whether the dragged data can be dropped there; the `DragDropManager` asks the `DropSource` to set the pointer's shape based on the response. When a drop occurs in a drop target, the `DragDropManager` sends a message to process the transferred data.
- Instances of `ConfigurableDropTarget`, which identify the widgets that have been set up as drop targets. When the pointer moves to a particular drop target, the associated `ConfigurableDropTarget` receives the messages that are sent by the `DragDropManager` and forwards them to the associated application model, which provides the actual response.
- An instance of `DragDropContext`, in which the `DragDropManager` combines the `DragDropData`, the `DropSource`, and the `ConfigurableDropTarget` to create a convenient object for passing as an argument in various messages. The `DragDropManager` also adds other information to the `DragDropContext` during a drag, such as the current pointer position and modifier key state.

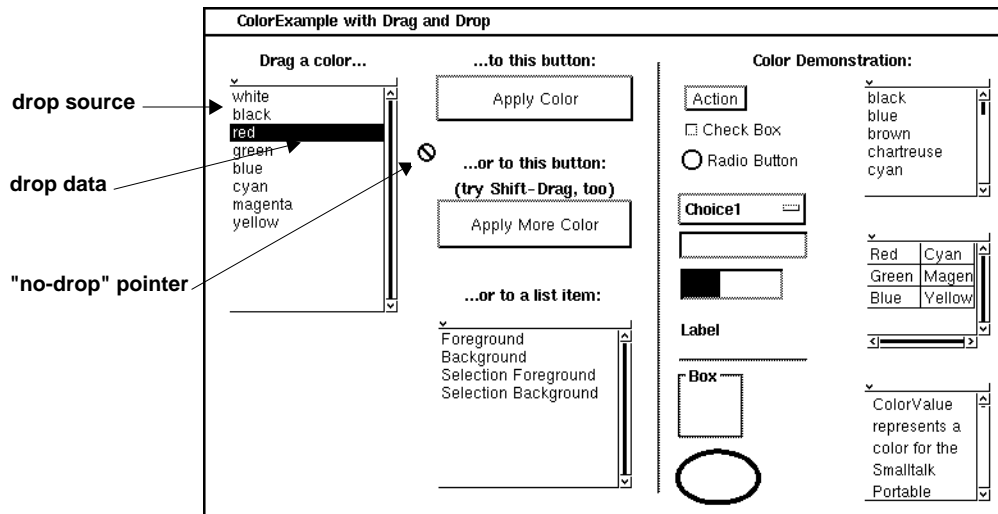
Extended example. You can obtain an example of an extended drag and drop implementation by filing in `tooldd.st` from the `extras` subdirectory of your VisualWorks installation directory

or folder. This example modifies the `Browser`, `UIPropertiesTool`, and `VisualLauncher` classes so that you can:

- Drag a selector to move it to a different protocol in a browser.
- Drag a class to move it to a different category in a browser.
- Drag a category, class, protocol, or selector to the System Browser button on the toolbar of the VisualWorks main window. This opens a System Browser on the dragged object.
- Drag a category, class, protocol, or selector to the File List button on the toolbar of the VisualWorks main window. This files out the dragged object.
- Drag a selector for an interface specification (for example, `windowSpec`) to the Canvas button on the toolbar of the VisualWorks main window. This opens the canvas for editing.
- Drag a selector from a browser to a Properties Tool to copy it into a field that accepts selectors.

After filing in `tooldd.st`, close any open browsers and Properties Tools. You must also rebuild the VisualWorks main window, for example, by exiting and restarting VisualWorks.

Adding a Drop Source



Strategy

A drop source is a widget from which a drag can originate; it displays the data that is to be transferred. Currently, only a list can serve as a drop source.

You set up a drop source by filling in the Drag OK and Drag Start properties on the Drop Source page of the Properties Tool. These properties specify the names of the messages that the widget will send whenever the user presses a mouse button down and starts to move the pointer within the widget's bounds. You program the widget's application model to respond to these messages as follows:

- The drag-ok method must return a Boolean to indicate whether-drag and drop is appropriate from this drop source.
- The drag-start method must create DragDropData, DropSource, and DragDropManager instances and set them to work.

The basic steps show how to set up a list of colors from which a single color can be dragged. The variant outlines steps for setting up a multi-selection list as a drop source.

Basic Steps

Online example: ColorDDEExample

1. Add a list widget to the canvas and set its **Aspect** and **ID** properties (in this example, enter `color` and `colorList`, respectively). Apply the properties and install the canvas.
2. Use the canvas' **define** command or a **System Browser** to add an instance variable (`color`) to the application model to hold the list's `SelectionInList`.
3. Use the canvas' **define** command or a **System Browser** to create an aspect method (`color`) in an aspects protocol.

```
color                                     "Basic Step 3"
  ^color
```

4. Using a **System Browser**, initialize the `SelectionInList` in an initialize method in the application model (initialize-release protocol).

```
initialize
| tableList |

color := SelectionInList
  with: (#white #black #red #green #blue
        #cyan #magenta #yellow ) asList.          "Basic Step 4"

colorLayer := SelectionInList
  with: #('Foreground' 'Background' 'Selection Foreground'
        'Selection Background' ) asList.

"Sample data for demonstration widgets"
sampleList := SelectionInList with: ColorValue constantNames asList.
tableList := TwoDList
  on: #('Red' 'Cyan' 'Green' 'Magenta' 'Blue' 'Yellow' )
  columns: 2
  rows: 3.
sampleTable := TableInterface new
  selectionInTable: (SelectionInTable with: tableList).
sampleText := ColorValue comment asValue.
```

5. In a Properties Tool, fill in the list's Drag OK property with the name of the method that will determine whether a drag can proceed from this list. In this example, enter `colorWantToDrag:` (the selector must end with a colon).
6. In a Properties Tool, fill in the list's Drag Start property with the name of the method that will initiate drag and drop. In this example, enter `colorDrag:` (the selector must end with a colon).
7. Leave the Select On Down property selected. This causes a selection to occur when the mouse is pressed down to start the drag (rather than waiting for the mouse to be released). Apply the properties and install the canvas.
8. In a System Browser, add a drag-ok method (`colorWantToDrag:`) in a suitable protocol (drag source). This method must return a Boolean (true to permit drag and drop, false to prevent it). Note that this method must accept a Controller as an argument, even if that controller isn't used.

```
colorWantToDrag: aController "Basic Step 8"  

"Determine whether to permit a drag to start from this widget. In this case,  

make sure that there is data to drag and that the drag starts a selection."
```

```
^self color list size > 0 and: [self color selection notNil]
```

9. In a System Browser, add a drag-start method (`colorDrag:`) in the drag source protocol. Note that this method must accept a Controller as an argument.
10. In the drag-start method, create a `DragDropData` instance. This instance will hold various pieces of information about the dragged data and where it came from.
11. Send a `key:` message to the `DragDropData` instance to specify a symbol (`#colorChoice`) that identifies the kind of data being stored. A drop target can use this key to filter out inappropriate kinds of data (for example, data being dragged from an unrelated drop source).
12. Send messages to the `DragDropData` instance to specify any further information that a drop target might use when evaluating this drag. Typically, you send a `contextWindow:` message specifying the containing window, a `contextWidget:` message

specifying the list widget; and a contextApplication: message specifying the application model.

13. Send a clientData: message to the DragDropData instance to store the object to be transferred (in this case, the color that is currently selected). Notice that the color choice is stored in an IdentityDictionary, which is a general technique for storing multiple related pieces of data. (The utility of this technique is not exploited here, however, so the color selection could have been stored directly in the DragDropData).
14. Create an instance of DropSource to make predefined kinds of visual feedback available during the drag.
15. Create an instance of DragDropManager and initialize it with the DropSource and DragDropData instances.
16. Send a doDragDrop message to the DragDropManager to start the drag and drop.

```
colorDrag: aController "Basic Step 9"
"Drag the currently selected color. Provide all available information about the
context of the color so that the drop target can use whatever it needs."
```

```
| ds dm data |
data := DragDropData new. "Basic Step 10"
data key: #colorChoice. "Basic Step 11"
data contextWindow: self builder window. "Basic Step 12"
data contextWidget: aController view.
data contextApplication: self.
data clientData: IdentityDictionary new. "Basic Step 13"
data clientData at: #colorChoice put: self color selection. "Basic Step 13"

ds := DropSource new. "Basic Step 14"

dm := DragDropManager
    withDropSource: ds
    withData: data. "Basic Step 15"
dm doDragDrop "Basic Step 16"
```

Note that when the drag and drop completes, the doDragDrop message returns a symbol which can be stored in a temporary variable and then used to trigger further actions (such as cutting the dragged data out of the drop source list). This symbol is ignored in the colorDrag: method.

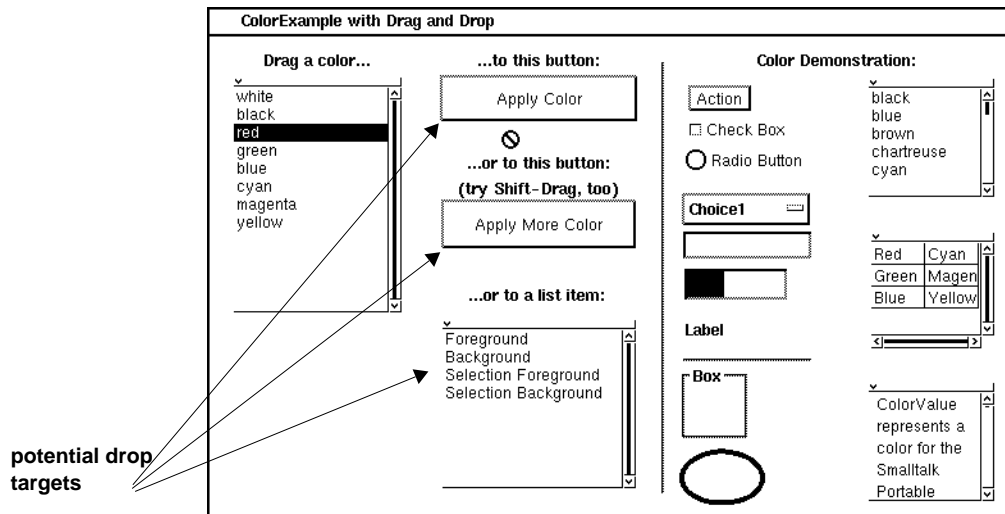
Variants

V1. Dragging Multiple Selections

You can use most of the same basic steps to set up a drop source so that multiple selected items can be dragged from it. The exceptions are listed below:

1. In Basic Step 1, click the list's **Multi Select** property.
2. In Basic Step 4, initialize the list's aspect variable with a `MultiSelectionInList` instead of a `SelectionInList`.
3. In Basic Step 13, send the `selections` message (instead of `selection`) to obtain the selected data to store in the `DragDropData` instance. The `selections` message returns an ordered collection of objects. You may want to store individual members of this collection as separate elements of an `IdentityDictionary`.

Adding a Drop Target (General)



Strategy

A drop target is a widget in which dragged data can potentially be dropped. You can set up any window or widget as a drop target (except a linked or embedded dataform).

In general, you set up a drop target by filling in one or more of its properties on the Drop Target page of the Properties Tool, and then implementing corresponding methods in the application model. Each of a widget's DropTarget properties specifies the name of a message that the DragDropManager sends at various points after the drag encounters this widget.

At a minimum, you must fill in the widget's Drop property to specify the name of a method that implements the desired response when a drop occurs in that widget. Filling in the Drop property causes the builder to set up the widget with a ConfigurableDropTarget instance so that the DragDropManager can recognize the widget as a drop target.

In addition, you normally fill in the widget's Entry, Over, and Exit properties to specify the names of methods that provide visual feedback when the pointer is dragged across the widget. Typically, these methods specify the pointer's shape and adjust the

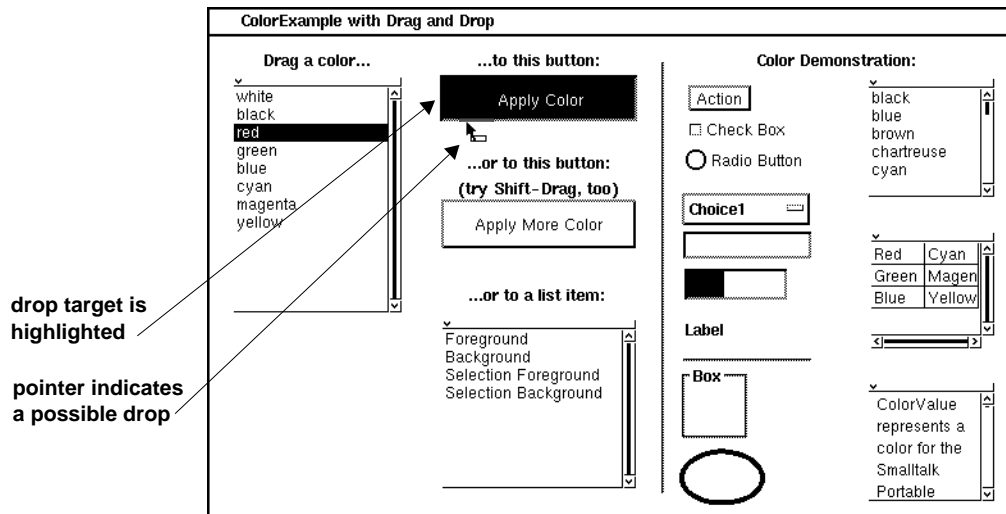
drop target's appearance to signal whether the drop target can accept a drop from this particular drag. Strictly speaking, these properties are optional, in that drag and drop can function without them. However, providing visual feedback is normally required by user interface design guidelines.

Specific steps are given in the topics listed under See Also.

See Also

- “Providing Visual Feedback During a Drag” on page 350
- “Responding to a Drop” on page 359

Providing Visual Feedback During a Drag



Strategy

As part of adding a drop target, you normally arrange for visual feedback to be given to users when they drag the pointer over it. The purpose of this feedback is to let users know whether a drop can be accepted from this particular drag, and, if so, what kind of transfer may result. Visual feedback typically includes changing the pointer's shape and adjusting the drop target's appearance—for example, by highlighting a button (basic steps), changing a label (first variant), or scrolling a list to track the pointer's movement over it (second variant).

Drop target messages. You arrange for visual feedback by filling in the drop target's `Entry`, `Over`, and `Exit` properties with the names of messages to be sent by the `DragDropManager` at various points during a drag. You then implement methods in the application model to respond to these messages:

- The entry message is sent as soon as the pointer enters the widget's bounds. The method typically saves the drop target's visual state for restoring later, and/or toggles simple visual characteristics such as highlighting.

- The `over` message is sent immediately after the `entry` message, and then every time the pointer moves within the widget's bounds. The method typically adjusts the drop target's appearance in response to pointer location or a modifier key press.
- The `exit` message is sent whenever the pointer is dragged out of the widget's bounds before the mouse button is released. The method typically restores the widget's original appearance.

Each method has access to the dragged data through the `DragDropContext` instance that is passed to it by the `DragDropManager`. The methods query the `DragDropContext` to decide what kind of visual feedback to provide. Furthermore, these methods use the `DragDropContext` to save and restore drop target characteristics (both variants).

Note that no changes are made to a drop target's appearance unless you implement them in these methods. Programmatic techniques for changing widget appearance are described in the specific widget chapters of this book.

Pointer shapes. Each of the `entry`, `over`, and `exit` methods control the pointer shape by returning an *effect symbol*. The `DragDropManager` passes each effect symbol to the operation's `DropSource` instance, which sets the pointer shape accordingly. A standard `DropSource` recognizes these basic effect symbols:

- `#dropEffectNone`—produces a pointer shaped like a circle with a slash through it; usually indicates that no transfer is possible in the pointer's current location.
- `#dropEffectMove`—produces an arrow-shaped pointer with an open box below it; usually indicates a simple transfer such as a move (data is cut from the source after the transfer).
- `#dropEffectCopy`—produces the same pointer as `#dropEffectMove`, but with a plus sign; usually indicates a modified transfer such as a copy (data is left in the source after the transfer).

Basic Steps

Online example: `ColorDDEExample`

This example highlights the `Apply Color` button and changes the pointer's shape while the pointer is in the button.

1. In the canvas, select the **Apply Color** button, and set its ID property (in this case, enter #applyColorButton).
2. On the **Drop Target** page of a **Properties Tool**, fill in the widget's **Entry**, **Over**, and **Exit** properties with the names of the messages to be sent during the drag (applyColorEnter:, applyColorOver:, and applyColorExit:, respectively). Each selector must end with a colon. Apply the properties and install the canvas.
3. In a **System Browser**, add an entry method (applyColorEnter:) in an appropriate protocol (in this case, drop target - button1). The method must accept a **DragDropContext** instance as an argument.
4. In the entry method, test the dragged data to determine what kind of feedback to provide (positive feedback if the data is a color choice, and negative feedback otherwise). Send a key message to the **DragDropContext** instance to obtain the identifying symbol that was assigned when the drag started. If the data's key is not #colorChoice, return an effect symbol (#dropEffectNone) to signal that a drop is not allowed.
5. If the dragged data is acceptable, highlight the button as if it were pressed and return an effect symbol (#dropEffectMove) that signals permission to drop.

```

applyColorEnter: aDragContext                                "Basic Step 3"
"A drag has entered the bounds of the Apply Color button. Test whether a drop
would be permitted here with this data. If so, cause the button to be highlighted
as if it were pressed, and return a symbol that indicates the feedback to be
given to the user."

    aDragContext key == #colorChoice
        ifFalse: [#dropEffectNone].                                "Basic Step 4"

    (self builder componentAt: #applyColorButton) widget
        isInTransition: true.                                       "Basic Step 5"

    ^#dropEffectMove.                                             "Basic Step 5"

```

6. Add an over method (applyColorOver:) that accepts a **DragDropContext** instance as an argument.

7. In the `over` method, test the dragged data and return the appropriate effect symbols. No other processing is necessary in this method because the button's highlighting does not vary with the pointer's movement.

applyColorOver: aDragContext "Basic Step 6"
 "A drag is over the Apply Color button. Test whether a drop would be permitted here with this data. If so, return a symbol that indicates the feedback to be given to the user. The DragDropManager uses this symbol to determine the pointer shape."

```
aDragContext key == #colorChoice
    ifFalse: [^#dropEffectNone]. "Basic Step 7"
    ^#dropEffectMove "Basic Step 7"
```

8. Add an exit method (`applyColorExit:`) that accepts a `DragDropContext` instance as an argument.
9. In the exit method, test the dragged data and return `#dropEffectNone` if the dragged data is not a color choice.
10. If the dragged data is acceptable, reverse any visual effect that was set in the entry method (in this case, unhighlight the button).
11. Return `#dropEffectNone`, to signal that no drop has occurred (this method executes only if the pointer leaves the widget without dropping).

applyColorExit: aDragContext "Basic Step 8"
 "A drag has exited the Apply Color button without dropping. Test whether a drop would have been permitted here with this data. If so, restore the button to its former state, and return a symbol that indicates the feedback to be given to the user."

```
aDragContext key == #colorChoice "Basic Step 9"
    ifFalse: [^#dropEffectNone].
```

```
(self builder componentAt: #applyColorButton) widget
    isInTransition: false. "Basic Step 10"
```

```
^#dropEffectNone "Basic Step 11"
```

Variant

V1. Changing a Button Label During a Drag

Online example: ColorDDExample

This example saves and changes the label of the **Apply More Color** button when the pointer enters the button, restoring the original label when the pointer exits.

1. In the canvas, select the **Apply More Color** button, and set its ID property (in this case, enter `#applyMoreColorButton`).
2. On the **Drop Target** page of a **Properties Tool**, set the widget's **Entry**, **Over**, and **Exit** properties (enter `applyMoreColorEnter`., `applyMoreColorOver`., and `applyMoreColorExit`).
3. In an entry method (`applyMoreColorEnter`), create an **IdentityDictionary** in which to save the drop target's original state.
4. Save any button characteristics in the **IdentityDictionary** that are to be restored later. In this case, store the widget and its label. (Storing the widget is a stylistic option that enables the widget to be accessed later through the **DragDropContext** rather than through the builder.)
5. Get the widget's **ConfigurableDropTarget** instance from the **DragDropContext**, and set the **IdentityDictionary** as its client data.
6. Change the button's label by sending the `labelString`: `message` to the button. The message argument is the string to be displayed. Note that a different string is specified depending on the state of the shift key.

`applyMoreColorEnter`: `aDragContext`

"A drag has entered the bounds of the **Apply More Color** button. Test whether a drop would be permitted here with this data. If so, store the current label of the button. Then test whether the shift key is down. Based on this test, change the button's label and return a symbol that indicates the feedback to be given to the user."

```
| widget dict |
aDragContext key == #colorChoice
  ifFalse: [#dropEffectNone].
```

```
widget := (self builder componentAt: #applyMoreColorButton) widget.
```



```

dict := IdentityDictionary new.                "V1 Step 3"
dict at: #widget put: widget.                 "V1 Step 4"
dict at: #label put: widget label.           "V1 Step 4"
aDragContext dropTarget clientData: dict.    "V1 Step 5"

aDragContext shiftDown
  ifTrue:
    [widget labelString: 'Background'.        "V1 Step 6"
     ^#dropEffectCopy].
  widget labelString: 'Foreground'.           "V1 Step 6"
  ^#dropEffectMove.

```

7. In an exit method (applyMoreColorExit:), get the drop target's IdentityDictionary from the DragDropContext.
 8. Retrieve the button from the IdentityDictionary. (Alternatively, you could obtain the button from the builder.)
 9. Retrieve the original label from the IdentityDictionary and put it back on the button. (Note that argument of the label: message is a label object, not a string.)
 10. Remove the drop target data from the DragDropContext. This prepares the DragDropContext for the next drop target the pointer may encounter.
-

applyMoreColorExit: aDragContext

"A drag has exited the Apply More Color button without dropping. Test whether a drop would have been permitted here with this data. If so, restore the button to its former state, and return a symbol that indicates the feedback to be given to the user."

```

| dict widget |
aDragContext key == #colorChoice
  ifFalse: [^#dropEffectNone].

dict := aDragContext dropTarget clientData.  "V1 Step 7"
widget := dict at: #widget.                  "V1 Step 8"
widget label: (dict at: #label).             "V1 Step 9"
aDragContext dropTarget clientData: nil.    "V1 Step 10"

^#dropEffectNone.

```

V2. Tracking a Targeted List Item

Online example: ColorDDExample

This example provides visual feedback for a list in which a drop is intended for a particular item rather than the list as a whole. The steps cause the pointer's location to be indicated by *target emphasis*, a rectangular border around the item containing the pointer. The target emphasis tracks the pointer, scrolling if necessary. (Target emphasis is also used in keyboard traversal of lists to indicate the target for selection.)

1. In the canvas, select the list of color layers and set its ID property (in this case, enter #colorLayerList).
2. On the Drop Target page of a Properties Tool, set the widget's Entry, Over, and Exit properties (enter colorLayerEnter:, colorLayerOver:, and colorLayerExit:).
3. In an entry method (colorLayerEnter:), create an IdentityDictionary in which to save the drop target's original state.
4. Save any characteristics into the IdentityDictionary that are to be restored later. In this case, store the widget, the location of any target emphasis resulting from keyboard traversal, and a Boolean indicating whether the list has focus.
5. Get the widget's ConfigurableDropTarget instance from the DragDropContext. Store the IdentityDictionary as its client data.
6. Give focus to the list to prepare it for tracking the pointer with target emphasis.

colorLayerEnter: aDragContext

"A drag has entered the bounds of the list of color layers. Test whether a drop would be permitted here with this data. If so, save the initial state of the color layer list, give focus to the list, and return a symbol that indicates the feedback to be given to the user."

```
| dict widget |
aDragContext key == #colorChoice
  ifFalse: [#dropEffectNone].
```

```
widget := (self builder componentAt: #colorLayerList) widget.
dict := IdentityDictionary new.           "V2 Step 3"
dict at: #widget put: widget.           "V2 Step 4"
dict at: #targetIndex put: widget targetIndex. "V2 Step 4"
```

```
dict at: #hasFocus put: widget hasFocus.           "V2 Step 4"
aDragContext dropTarget clientData: dict.         "V2 Step 5"

widget hasFocus: true.                             "V2 Step 6"
^#dropEffectMove
```

7. In an over method (colorLayerOver:), retrieve the list widget from the DragDropContext.
 8. Send the showDropFeedbackIn:allowScrolling: message to the list to display target emphasis at the pointer's current position, scrolling, if necessary. (Remember, this message gets sent each time the pointer moves in the list).
-

colorLayerOver: aDragContext

"A drag is over the list of color layers. Test whether a drop would be permitted here with this data. If so, tell the list to scroll the target emphasis when the pointer moves. Return a symbol that indicates the feedback to be given to the user. The DragDropManager uses this symbol to determine the pointer shape."

```
| list |
aDragContext key == #colorChoice
  iffFalse: [#dropEffectNone].

list := aDragContext dropTarget clientData at: #widget.   "V2 Step 7"
list
  showDropFeedbackIn: aDragContext
  allowScrolling: true.                                   "V2 Step 8"

^#dropEffectMove
```

9. In an exit method (colorLayerExit:), get the drop target's IdentityDictionary from the DragDropContext and retrieve the list widget.
 10. Restore the list's original target emphasis and focus state.
 11. Remove the drop target data from the DragDropContext. This prepares the DragDropContext for the next drop target the pointer may encounter.
-

colorLayerExit: aDragContext

"A drag has exited the list of color layers without dropping. Test whether a drop would have been permitted here with this data. If so, restore the initial state of

the color layer list, and return a symbol that indicates the feedback to be given to the user."

```
| dict widget |  
aDragContext key == #colorChoice  
  ifFalse: [#dropEffectNone].
```

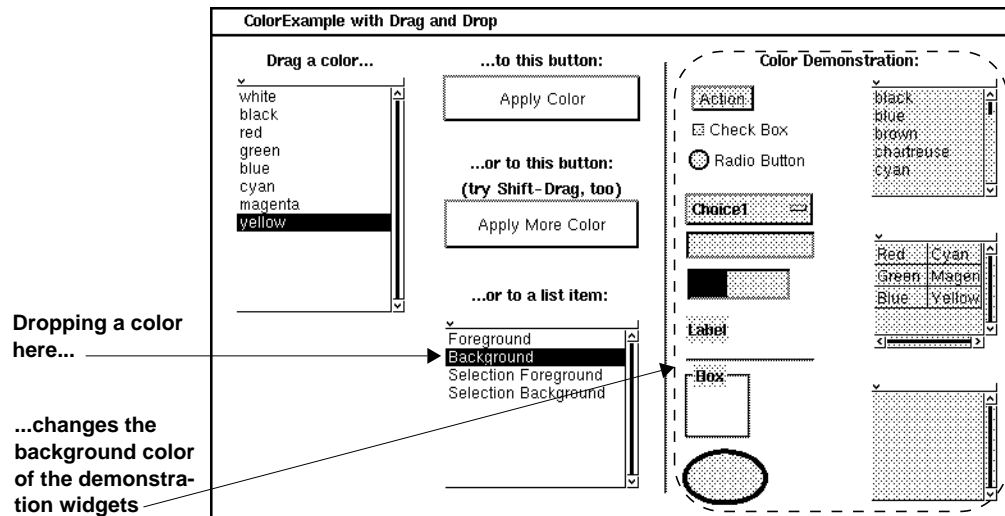
```
dict := aDragContext dropTarget clientData.           "V2 Step 9"  
widget := dict at: #widget.                            "V2 Step 9"  
widget targetIndex: (dict at: #targetIndex).          "V2 Step 10"  
widget hasFocus: (dict at: #hasFocus).                "V2 Step 10"
```

```
aDragContext dropTarget clientData: nil.             "V2 Step 11"  
^#dropEffectNone
```

See Also

- “Adding a Drop Target (General)” on page 348
- “Examining the Drag Context” on page 365
- “Responding to Modifier Keys” on page 366
- “Defining Custom Effect Symbols” on page 371

Responding to a Drop



Strategy

A central part of creating a drop target is to implement the action to be taken whenever a drop occurs in it. A typical action is to verify that the dragged data is acceptable to this drop target, and then process that data accordingly. You arrange for this by filling in the widget's Drop property on the Drop Target page of the Property Tool. This property specifies the name of the message that the DragDropManager will send when a drop occurs in the widget. You then create a corresponding method in the application model to implement the response.

Like the entry, over, and exit methods, a drop method receives a DragDropContext instance as an argument from the DragDropManager. The method can examine this instance to determine whether to accept the dragged data and, if so, how to process it.

A typical drop method adjusts the appearance of the drop target widget to reverse any visual feedback caused by the enter or over method or to provide visual evidence of the completed drop. The drop method may also need to clean up any drop target data that was created by the entry method.

A drop method returns an effect symbol for the `DragDropManager` to return to the drag-start method that initiated the drag.

The basic steps set up the **Apply Color** button in `ColorDDEExample` so that dropping a color on this button applies that color to the foreground layer of the demonstration widgets. The variant sets up the color layer list so that dropping a color on a targeted layer applies the dragged color to that layer.

Basic Steps

Online example: `ColorDDEExample`

1. In the canvas, select the widget you want to use as a drop target. In this case, select the **Apply Color** button.
2. On the **Drop Target** page of a **Properties Tool**, set the widget's **Drop** property (`applyColorDrop:`). The selector must end with a colon. Apply properties and install the canvas.
3. In a **System Browser**, add a drop method (`applyColorDrop:`) in an appropriate protocol (in this case, `drop target - button 1`). The method must accept a `DragDropContext` instance as an argument.
4. In the drop method, determine whether the dragged data should be accepted for processing (that is, whether it is a color choice). To do this, send a key message to the `DragDropContext` instance to obtain the identifying symbol that was assigned when the drag started. If the key is not `#colorChoice`, return an effect symbol (`#dropEffectNone`) to signal that no drop is allowed.
5. If the dragged data is acceptable, perform the processing that is to result from the drop. In this example, send a `sourceData` message to the `DragDropContext` to obtain the `DragDropData`; then send this object a `clientData` message to obtain the selected color. Turn the selected color into a color value and set it as the foreground color of the demonstration widgets.
6. Restore the widget's original appearance (turn off the highlighting that was turned on by the `applyColorEnter:` method).
7. Return an effect symbol indicating the result of the drop. This symbol is passed to the drag-start method (`colorDrag:`),

where it can be used to trigger further actions at the drop source.

applyColorDrop: aDragContext	"Basic Step 3"
"A drop has occurred in the Apply Color button. If the drop is permitted, set the foreground color of the demonstration widgets to be the dragged color choice. Restore the button to its former visual state and return an effect symbol for possible use in the colorDrag method."	
dict aColor aDragContext key == #colorChoice ifFalse: [#dropEffectNone].	"Basic Step 4"
dict := aDragContext sourceData clientData. aColor := ColorValue perform: (dict at: #colorChoice).	"Basic Step 5"
(self builder componentAt: #applyColorButton) widget isInTransition: false.	"Basic Step 6"
self foregroundColor: aColor.	"Basic Step 5"
^#dropEffectMove.	"Basic Step 7"

Variant

Dropping data on a particular list item requires that you enable *target emphasis* in the list through the `entry` and `over` methods. Target emphasis tracks the location of the pointer, displaying a rectangular border around the currently targeted list item.

Online example: [ColorDDExample](#)

1. In the canvas, select the list of color layers and set its ID property (in this case, enter `#colorLayerList`).
2. On the Drop Target page of a Properties Tool, set the widget's Entry, Over and Drop properties (`colorLayerEnter:`, `colorLayerOver:` and `colorLayerDrop:`).
3. In an entry method (`colorLayerEnter:`), give focus to the list to prepare it for displaying target emphasis.

colorLayerEnter: aDragContext

"A drag has entered the bounds of the list of color layers. Test whether a drop would be permitted here with this data. If so, save the initial state of the color layer list, give focus to the list, and return a symbol that indicates the feedback to be given to the user."

```
| dict widget |
aDragContext key == #colorChoice
  ifFalse: [#dropEffectNone].
```

```
widget := (self builder componentAt: #colorLayerList) widget.
dict := IdentityDictionary new.
dict at: #widget put: widget.
dict at: #targetIndex put: widget targetIndex.
dict at: #hasFocus put: widget hasFocus.
aDragContext dropTarget clientData: dict.
```

```
widget hasFocus: true.                                "Variant Step 3"
^#dropEffectMove
```

4. In an over method (colorLayerOver:), retrieve the list widget from the DragDropContext and send it the showDropFeedbackIn:allowScrolling: message to display target emphasis at the pointer's current position.

colorLayerOver: aDragContext

"A drag is over the list of color layers. Test whether a drop would be permitted here with this data. If so, tell the list to scroll the target emphasis when the pointer moves. Return a symbol that indicates the feedback to be given to the user. The DragDropManager uses this symbol to determine the pointer shape."

```
| list |
aDragContext key == #colorChoice
  ifFalse: [#dropEffectNone].
```

```
list := aDragContext dropTarget clientData at: #widget.    "Variant Step 4"
```

```
list
  showDropFeedbackIn: aDragContext
  allowScrolling: true.                                    "Variant Step 4"
```


`^#dropEffectMove`

5. In a drop method (`colorLayerDrop:`), test whether the dragged data is a color choice; if so, obtain the selected color from the dragged data and turn it into a color value.
6. Send a `targetIndex` message to the list to get the index of the targeted list item (the item containing the pointer when the drop occurs).
7. Get the color layer that is shown in the list at the targeted index.
8. Give visual feedback to indicate a successful drop. In this case, cause the targeted list item to appear selected (set the list's selection index to be the targeted index). Alternatively, you could restore the list to its original visual state, as is done in the `colorLayerExit:` method.
9. Use the targeted color layer to choose the appropriate message for changing the color of the demonstration widgets.

colorLayerDrop: aDragContext

"A drop has occur in the list of color layers. If the drop is permitted, combine the dragged color choice and the targeted color layer to change the color of the appropriate parts of the demonstration widgets. Return an effect symbol for possible use in the `colorDrag` method."

```

| dict aColor widget idx aLayer |
aDragContext key == #colorChoice
    ifFalse: [^#dropEffectNone].                                "Variant Step 5"

dict := aDragContext sourceData clientData.                  "Variant Step 5"
aColor := ColorValue perform: (dict at: #colorChoice).

widget := aDragContext dropTarget clientData at: #widget.
idx := widget targetIndex.                                    "Variant Step 6"
idx = 0 ifTrue: [^#dropEffectNone].
aLayer := self colorLayer listHolder value at: idx.          "Variant Step 7"

self colorLayer selectionIndexHolder value: idx.             "Variant Step 8"

aDragContext dropTarget clientData: nil.

```

```
aLayer = 'Foreground'                                "Variant Step 9"
    ifTrue: [self foregroundColor: aColor].
aLayer = 'Background'
    ifTrue: [self backgroundColor: aColor].
aLayer = 'Selection Foreground'
    ifTrue: [self selectionForegroundColor: aColor].
aLayer = 'Selection Background'
    ifTrue: [self selectionBackgroundColor: aColor].

^#dropEffectMove.
```

See Also

- “Adding a Drop Target (General)” on page 348
- “Providing Visual Feedback During a Drag” on page 350
- “Examining the Drag Context” on page 365
- “Responding to Modifier Keys” on page 366

Examining the Drag Context

Strategy

In `ColorDDEExample`, the various `Entry`, `Over`, `Exit`, and `Drop` methods determine what kind of action or visual feedback to provide by examining the dragged data. Specifically, these methods test the key symbol that was assigned to the dragged data in the `Drag Start` method.

In general, drop target methods can examine a variety of information by querying the `DragDropContext` instance that is passed to them by the `DragDropManager`. Among other things, the `DragDropContext` instance contains the `DragDropData` you created in the `Drag Start` method, plus the current pointer location and modifier key states.

Basic Steps

- In a drop target method, obtain information about the drag from the `DragDropContext` instance that is passed to the method (assume the argument name is `aDragContext`).

"Get the key that was assigned to the dragged data."

`aDragContext key.`

"Get the application model from which the drag originated."

`aDragContext sourceData contextApplication.`

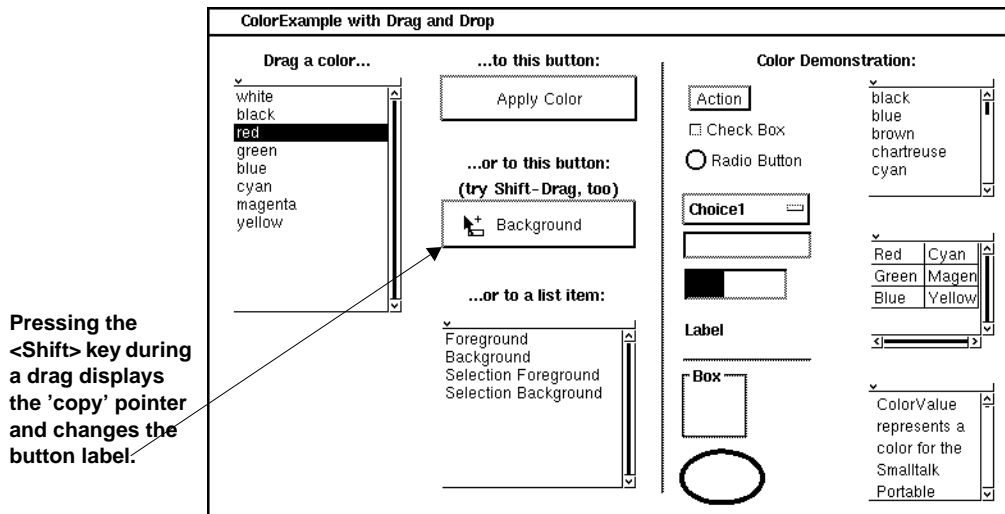
"Get the window from which the drag originated."

`aDragContext sourceData contextWindow.`

"Get the current pointer location."

`aDragContext mousePoint.`

Responding to Modifier Keys



Pressing the <Shift> key during a drag displays the 'copy' pointer and changes the button label.

Strategy

You can make drag and drop sensitive to the state of the <Control>, <Shift>, <Alt>, and <Meta> modifier keys. For example, in many applications, a user can move a file by dragging it and copy a file by <Shift>-dragging it.

Making drag and drop sensitive to modifier keys involves:

- Providing the appropriate visual feedback in the drop target's entry, over, and exit methods.
- Providing appropriate processing in the drop target's drop method.

The basic steps set up the Apply More Color button in ColorDDEExample so that dragging changes the foreground color of the demonstration widgets, while <Shift>-dragging changes the background color.

Basic Steps

Online example: ColorDDEExample

1. In the canvas, select the list of color layers and set its ID property (in this case, enter #applyMoreColorButton).
2. On the Drop Target page of a Properties Tool, set the widget's Entry, Over, Exit and Drop properties (applyMoreColorEnter:, applyMoreColorOver:, applyMoreColorExit:, and applyMoreColorDrop:). Apply properties and install the canvas.
3. In an entry method (applyMoreColorEnter:), send a shiftDown message to the DragDropContext instance to find out whether the user is pressing the <Shift> key down.
4. If the <Shift> key is down, provide appropriate visual feedback. In this case, change the button's label to indicate that background colors will be set, and return an effect symbol (#dropEffectCopy) to signal a modified transfer.
5. If the <Shift> key is not down, change the button's label to indicate that foreground colors will be set, and return an effect symbol (#dropEffectMove) to signal a regular transfer.

applyMoreColorEnter: aDragContext

"A drag has entered the bounds of the Apply More Color button. Test whether a drop would be permitted here with this data. If so, store the current label of the button. Then test whether the shift key is down. Based on this test, change the button's label and return a symbol that indicates the feedback to be given to the user."

```
| widget dict |
aDragContext key == #colorChoice
  ifFalse: [^#dropEffectNone].

widget := (self builder componentAt: #applyMoreColorButton) widget.
dict := IdentityDictionary new.
dict at: #widget put: widget.
dict at: #label put: widget label.
aDragContext dropTarget clientData: dict.

aDragContext shiftDown
  ifTrue:
    [widget labelString: 'Background'.
     ^#dropEffectCopy].
```

"Basic Steps 3"

"Basic Steps 4"

```

widget labelString: 'Foreground'.
^#dropEffectMove.

```

"Basic Steps 5"

6. In an over method (applyMoreColorOver:), find out whether the user has changed the <Shift> key state while dragging the pointer within the widget. (Remember, this method executes each time the pointer moves in the widget.) If the <Shift> key is down, test whether the button's label needs to change; if so, change it. Return the #dropEffectCopy symbol.
 7. If the <Shift> key is not down, test whether the button's label needs to change; if so, change it. Return the #dropEffectMove symbol.
-

applyMoreColorOver: aDragContext

"A drag is over the Apply More Color button. Test whether a drop would be permitted here with this data. If so, test whether the shift key is down. Based on this test, return a symbol that indicates the feedback to be given to the user. The DragDropManager uses this symbol to determine the pointer shape."

```

| widget |
aDragContext key == #colorChoice
  ifFalse: [#dropEffectNone].

widget := aDragContext dropTarget clientData at: #widget.

aDragContext shiftDown
  ifTrue:
    [widget label text string = 'Background'
     ifFalse: [widget labelString: 'Background'].
     ^#dropEffectCopy].

widget label text string = 'Foreground'
  ifFalse: [widget labelString: 'Foreground'].
^#dropEffectMove.

```

"Basic Step 6"

"Basic Step 7"

8. In an exit method (applyMoreColorExit:), restore the button's original label. In this example, the same label is restored, regardless of the <Shift> key's state.

applyMoreColorExit: aDragContext

"A drag has exited the Apply More Color button without dropping. Test whether a drop would have been permitted here with this data. If so, restore the button to its former state, and return a symbol that indicates the feedback to be given to the user."

```
| dict widget |
aDragContext key == #colorChoice
  ifFalse: [#dropEffectNone].

dict := aDragContext dropTarget clientData.
widget := dict at: #widget.
widget label: (dict at: #label).
aDragContext dropTarget clientData: nil.

^#dropEffectNone.
```

"Basic Steps 8"

9. In a drop method (applyMoreColorDrop:), test whether the <Shift> key is down.
10. If the <Shift> key is down, apply the dragged color choice to the background color layer of the demonstration widgets. Return #dropEffectCopy to signal a modified transfer.
11. If the <Shift> key is not down, apply the dragged color choice to the foreground color layer. Return #dropEffectMove to signal a regular transfer. Note that a drag-start method could respond differently depending on which symbol is returned.

applyMoreColorDrop: aDragContext

"A drop has occurred in the Apply More Color button. If the drop is permitted, obtain the dragged color. Then test whether the shift key is down. If so, set the background color of the demonstration widgets. If not, set their foreground color. Restore the button to its former visual state and return an effect symbol for possible use in the colorDrag method."

```
| dict widget aColor |

aDragContext key == #colorChoice
  ifFalse: [#dropEffectNone].
```

```
dict := aDragContext sourceData clientData.  
aColor := ColorValue perform: (dict at: #colorChoice).
```

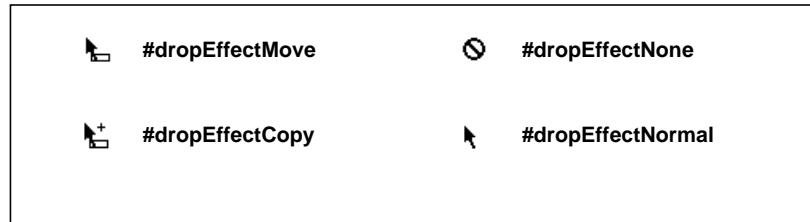
```
dict := aDragContext dropTarget clientData.  
widget := dict at: #widget.  
widget label: (dict at: #label).  
aDragContext dropTarget clientData: nil.
```

```
aDragContext shiftDown                                "Basic Steps 9"  
  ifTrue:  
    [self backgroundColor: aColor.  
     ^#dropEffectCopy].                                "Basic Steps 10"  
  self foregroundColor: aColor.  
  ^#dropEffectMove.                                   "Basic Steps 11"
```

See Also

- “Adding a Drop Target (General)” on page 348
- “Providing Visual Feedback During a Drag” on page 350
- “Responding to a Drop” on page 359

Defining Custom Effect Symbols



Strategy

As part of providing visual feedback during drag and drop, you set the pointer's shape by returning effect symbols from the drop target's entry, over, and exit methods. The `DragDropManager` passes each symbol to the operation's `DropSource` instance, which converts it into a pointer shape.

A standard `DropSource` recognizes the following effect symbols:

- `#dropEffectNone`—produces a pointer shaped like a circle with a slash through it; usually indicates that no transfer is possible in the pointer's current location.
- `#dropEffectMove`—produces an arrow-shaped pointer with an open box below it; usually indicates a simple transfer such as a move (data is cut from the source after the transfer).
- `#dropEffectCopy`—produces the same pointer as `#dropEffectMove`, but with a plus sign; usually indicates a modified transfer such as a copy (data is left in the source after the transfer).
- `#dropEffectNormal`—produces a plain arrow-shaped pointer; useful for situations in which other visual feedback besides pointer shape is provided.

If you want to use different pointer shapes in your application, you can add your own effect symbols (basic steps) or override the existing ones (variant).

Basic Steps

Adding a New Effect Symbol

1. In the application model, create a `ConfigurableDropSource` instance (instead of a `DropSource` instance) in a drag-start method.
2. Initialize the `ConfigurableDropSource` instance with the name of the message to be sent to convert an effect symbol into a pointer shape. In this example, specify `giveFeedback:for:`. Note that the message name must consist of two keywords.
3. Initialize the `ConfigurableDropSource` instance with the object that is to receive the message specified in step 2. In this case, specify the application model itself.
4. Initialize the `DragDropManager` with the `ConfigurableDropSource` instance.

`someDragStartMethod: aController`

"This is a hypothetical method that doesn't really exist in any example."

```
| ds dm data |
data := DragDropData new.
data key: #someKindOfData.
data contextWindow: self builder window.
data contextWidget: aController.
data contextApplication: self.
data clientData: IdentityDictionary new.
data clientData at: #someKindOfData put: self someData selection.

ds := ConfigurableDropSource new.           "Basic Step 1"
ds giveFeedbackSelector: #giveFeedback:for: . "Basic Step 2"
ds receiver: self.                         "Basic Step 3"

dm := DragDropManager
    withDropSource: ds
    withData: data.                          "Basic Step 4"

dm doDragDrop.
```

5. In the application model, create a `giveFeedback:for:` method that accepts an effect symbol and a `ConfigurableDropSource` instance as its arguments.
6. In the `giveFeedback:for:` method, define the desired new effect symbols. In this case, the symbol `#dropEffectDelete` results in a pointer shaped like a garbage can.
7. Return true to preserve the standard interpretation for the effect symbols `#dropEffectNone`, `#dropEffectMove`, `#dropEffectCopy`, and `#dropEffectNormal`.

```
giveFeedBack: anEffect for: aDropSource           "Basic Step 5"
  anEffect == #dropEffectDelete
    ifTrue: [Cursor garbage show].                "Basic Step 6"
  ^true                                           "Basic Step 7"
```

Variant

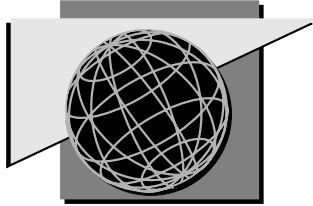
Redefining a Standard Effect Symbol

1. Repeat Basic Steps 1-5 to create and initialize a `ConfigurableDropTarget` instance in the `drag-start` method.
2. In the `giveFeedback:for:` method, define all of the effect symbol(s) you plan to use. In this case, cause the `#dropEffectMove` symbol to result in a pointer shaped like a hand, but keep the standard shapes for the other standard symbols.
3. Return false to prevent your custom definitions from being overridden by the standard definitions.

```
giveFeedBack: anEffect for: aDropSource           "Variant Step 2"
  anEffect == #dropEffectMove ifTrue: [Cursor hand show].
  anEffect == #dropEffectCopy ifTrue: [Cursor standardDragCopy show].
  anEffect == #dropEffectNone ifTrue: [Cursor dropNotOK show].
  anEffect == #dropEffectNormal ifTrue: [Cursor normal show].
  ^false                                           "Variant Step 3"
```

See Also

- “Providing Visual Feedback During a Drag” on page 350



Chapter 19

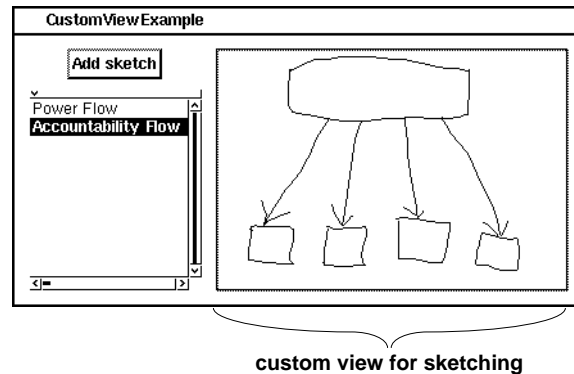
Custom Views

Creating a View Class	376
Connecting a View to a Domain Model	378
Defining What a View Displays	380
Updating a View When Its Model Changes	382
Connecting a View to a Controller	385
Redisplaying All or Part of a View	387
Integrating a View into an Interface	389

See Also

- “Creating a Custom Dialog” on page 296
- “Custom Controllers” on page 391

Creating a View Class



Strategy

A view displays text or graphics representing all or part of a data model. Each of the existing widgets uses a view to display a data model. When an existing widget does not serve your purpose, you can create a custom view. You begin that process by creating a view class.

About the example: In `CustomView1Example`, a simple sketch pad is created using a `SketchView1`. A `SketchView1` uses a `Sketch` as its model—a `Sketch` has a name and a collection of points representing a series of sketched strokes. A `SketchView1` uses a `SketchController1` to handle mouse and keyboard input. These four classes are all example classes that have been created to demonstrate the interactions among a domain model (`Sketch`), a custom view (`SketchView1`), a custom controller (`SketchController1`), and an application model (`CustomView1Example`).

Basic Steps

Online example: `SketchView1`

1. In a System Browser, display the class-definition template by selecting a class category and making sure no class is selected.
2. In the template, replace “NameOfSuperclass” with the name of the view’s parent class (in the example, `View`). For

guidance in choosing a superclass, refer to the View class in the *VisualWorks Object Reference*.

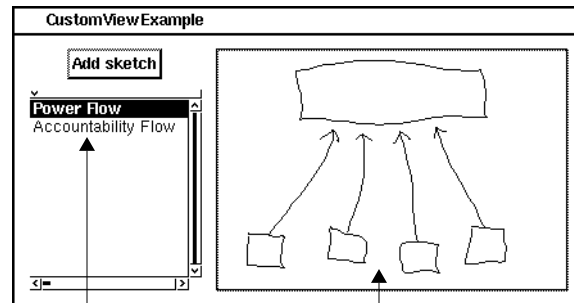
3. Replace “NameOfClass” with the new class’s name (SketchView1).
4. Supply variable names, if any, and then accept the definition.

```
View subclass: #SketchView1
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Examples-Cookbook'
```

See Also

- “Creating a Class (Subclassing)” on page 26

Connecting a View to a Domain Model



When a different domain model is selected . . .

. . . the customer view is given the new model and displays it

Strategy

A view displays text or graphics that communicate the state of its domain model, or at least a portion of its domain model. Since a view must communicate frequently with the domain model, it needs a way of accessing that object. As a subclass of `DependentPart`, every view inherits an instance variable for storing its model. Sending a `model: message` to the view, typically when the view is created, stores the model in this instance variable, where it can be accessed easily.

A side effect of the `model: message` is that the view is registered as a dependent of the model. This link sets the stage for the view to update its display when the model changes.

About the example: Although some views have the same domain model for their whole lifetimes, `SketchView1` changes its model each time the user selects a different Sketch. For that reason, `SketchView1` reimplements the `model:` method so it can update its display after storing the new model.

Basic Steps

Online example: CustomView1Example, SketchView1

1. Tell the view which object to use as its domain model. This is done in an initialization method or, as in the example, the application model (CustomView1Example) can notify the view whenever the domain model changes.

changedSketch

self sketchView model: self sketches selection.

"Basic Step 1"

2. If the view needs to take action when its model is changed, such as redisplaying itself, override the inherited model: method (as in SketchView1).

model: aModel

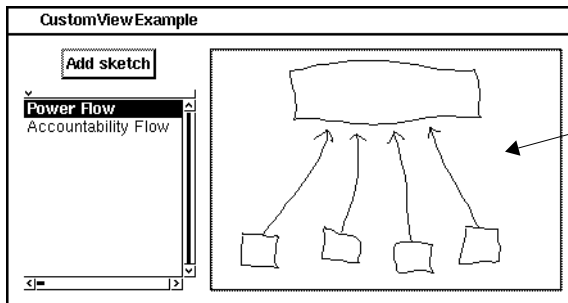
super model: aModel.
self invalidate.

"Basic Step 2"

"Tell the controller where to send menu messages."

self controller performer: aModel.

Defining What a View Displays



The view gets a collection of points from the model and displays the points as line segments

Strategy

A view's purpose is to display text or graphics. It does so in a method named `displayOn:`, which is sent to the view whenever circumstances require that it update its display.

The view decides what to display based on the state of its domain model.

It displays the text and/or graphics on a `GraphicsContext`, which is an object that windows and other display surfaces use for rendering objects.

About the example: In `CustomView1Example`, a `SketchView1` is used to display the line segments that are stored in its domain model, a `Sketch`.

Basic Steps

Online example: `SketchView1`

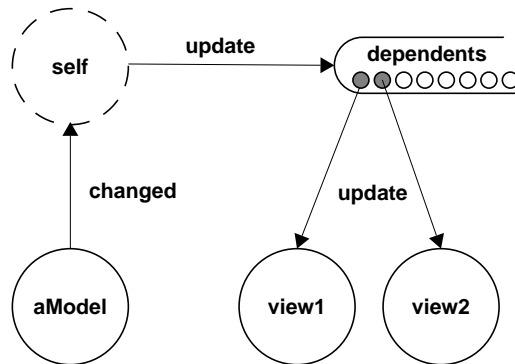
1. In a displaying protocol, add a `displayOn:` method to the view. The argument is a `GraphicsContext`.
2. In the `displayOn:` method, get the required data from the model (in the example, a set of line segments, each represented as a collection of points).
3. In the `displayOn:` method, display the appropriate text or graphics, based on the data from step 2 (in the example, each collection of points is displayed as a `Polyline`).

displayOn: aGraphicsContext self model isNil ifTrue: [^self].	"Basic Step 1"
self model strokes do: [:stroke aGraphicsContext displayPolyline: stroke].	"Basic Step 2" "Basic Step 3"

See Also

- “Displaying a Text Object” on page 558
- “Integrating a Graphic into an Application” on page 652

Updating a View When Its Model Changes



Strategy

Since the purpose of a view is to display some aspect of its domain model, it must be prepared to change its display when the model is changed.

When the domain model changes its state, it is responsible for notifying all of its dependents. It does so by sending a variant of the `changed:with:` message to itself. The first argument is a Symbol indicating what was changed, and the second argument is the new value.

The `changed:with:` message is inherited, and it sends an `update:with:` message to each dependent, passing along the same two arguments. Thus, the view must implement an `update:with:` method in which it gets the new data from the model and displays it.

Basic Steps

Online example: Sketch, SketchView1

1. In any method in the domain model that changes the model in a way that affects the view, send a variant of the `changed:with:` message to the model. (In the example, Sketch sends three such messages, one when it adds a point and the others when it erases some or all of its contents.)

add: aPoint
"Add aPoint to the current stroke."

```
self strokes last add: aPoint.
self changed: #stroke with: self currentLineSegment.      "Basic Step 1"
```

eraseLine

```
"Erase the last stroke that was drawn."

self strokes isEmpty
  ifFalse: [
    self strokes removeLast.
    self changed: #erase with: nil].      "Basic Step 1"
```

eraseAll

```
"Erase my contents."

self strokes removeAll: self strokes copy.
self changed: #erase with: nil.      "Basic Step 1"
```

2. In the view, implement a variant of the `update:with:` method to take the appropriate action in response to a change in the model. (In the example, the same `update:with:` method responds to either of the `changed:with:` messages sent by the model.)

```
update: anAspect with: anObject      "Basic Step 2"
"When a point is added to the model..."
anAspect == #stroke
  ifTrue: [anObject asStroker displayOn: self graphicsContext].

"When the model erases its contents..."
anAspect == #erase
  ifTrue: [self invalidate].
```

Variant

Using Shorter Forms of the `changed:with:` Message

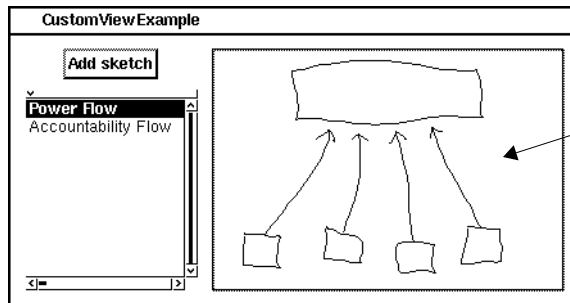
The `changed:with:` message sent by the model can be shortened to `changed:` when it only needs to tell dependents what changed without sending a new value.

The message can be further shortened to `changed` when it simply wants dependents to know that it has changed without telling them what aspect of itself has changed.

When two different forms are to be sent, send the fuller form in both places. Otherwise, the simpler message is ignored. In the example, the `erase` method uses `nil` as the second argument in `changed:with:.` That makes it conform with the `changed:with:` message sent by the `add:` method—otherwise, the `erase` method could have sent `changed:.`

The view must implement the corresponding form of the `update:with:` message.

Connecting a View to a Controller



This view uses a custom controller, so it names that controller as its `defaultControllerClass`

Strategy

A passive view—a view that does not respond to mouse or keyboard input—does not need a controller. An active view uses a controller to process mouse and keyboard input. A view is often closely allied with its controller, so an inherited mechanism installs the desired controller when the view is created. You can control which type of controller is installed.

About the example: `SketchView1` uses a `SketchController1`, which changes the cursor to a crosshair, notifies the model when the user draws with the `<Select>` button, and provides a menu when the `<Operate>` button is pressed.

Basic Step

Online example: `SketchView1`, `SketchController1`

- Use a System Browser to create a `defaultControllerClass` method for the view. This method returns the name of the desired controller class.

```
defaultControllerClass                                     "Basic Step"
    ^SketchController1
```

Variants

V1. Making a View Passive

1. Do the basic step.
2. Return `NoController` from the `defaultControllerClass` method.

V2. Connecting a Composite View to a Controller

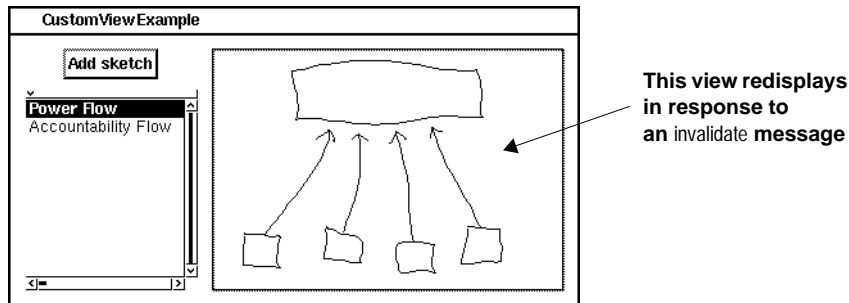
When multiple views inhabit the same window, normally they have separate controllers. In some situations, the composite object that groups them needs its own controller, either instead of the individual controllers or in addition to them. A `CompositeView` is intended for grouping views that need a common controller. To initialize its controller:

- Send a `controller:` message to the composite that groups the views. The argument is an instance of the desired type of controller (not the class name as with `defaultControllerClass`).

See Also

- “Creating a Controller Class” on page 395

Redisplaying All or Part of a View



Strategy

A view can redisplay its entire contents or just a portion of them. For example, when one window overlaps another, the overlap region is all that needs to be redisplayed when the lower window is no longer obscured by the upper window. This overlap region is called a *damage rectangle*, because it is a rectangular region that was damaged by an overlapping window.

The window's sensor keeps track of such damage rectangles and repairs them in a batch to avoid repairing the same region twice. Sending *invalidate* to a view causes the entire view to be treated as a damage rectangle, as shown in the basic steps. The first variant shows how to limit the damage rectangle to a portion of the window.

By default, damage rectangles are accumulated until the window's controller reaches a certain point in its cycle of activity. That is sufficient in most situations. However, when a competing process is monopolizing the processor, the delay can be significant. The second variant shows how to force the damage to be repaired immediately.

Invalidating a view is done in a view method, when the view updates its model. It can also be done by an application model that has changed a widget's data model in a way that bypasses the normal dependencies.

Basic Step

Online example: SketchView1

- Send `invalidate` to the view. This is typically done in a view method that changes the model (as in the example).

```
model: aModel
super model: aModel.
self invalidate.                                     "Basic Step"

"Tell the controller where to send menu messages."
self controller performer: aModel.
```

Variants

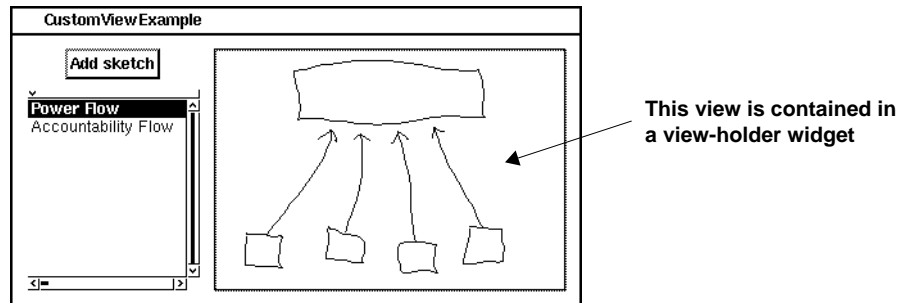
V1. Redisplaying Part of a View

- Send `invalidateRectangle:` to the view. The argument is a rectangle that represents all or part of the view's bounding box. The bounding box can be accessed by sending `bounds` to the view.

V2. Arranging for Immediate Redisplay

- Send `invalidateRectangle:repairNow:` to the view. The first argument is a rectangle that represents all or part of a view's bounding box. The second argument is `true` when immediate redisplay is desired, and `false` for the default behavior.

Integrating a View into an Interface



Strategy

A view-holder widget is provided on the Palette for integrating a custom view into a canvas. This view holder enables you to treat your custom view like a standard widget in that you can paint its layout and apply borders and scroll bars. However, your application is responsible for connecting the view to a domain model.

Basic Steps

Online example: CustomView1Example, SketchView1

1. Use a Palette to place a view-holder widget on the canvas.
2. In the view holder's **View** property, enter the name of the application-model method that supplies an instance of the desired view (`sketchView`).
3. If the application model will need to access the custom view while the application is running, use a System Browser to create an instance variable (`sketchView`) in which to store the custom view.
4. Use a System Browser to create the application-model method that you named in step 2 (`sketchView`). This method typically answers the contents of the instance variable that you created in step 3.

<code>sketchView</code>	"Basic Step 4"
<code>^sketchView</code>	

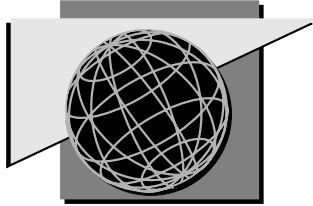
5. In an `initialize` method in the application model, create an instance of the custom view. If appropriate, connect the custom view to a data model. (In the example, there is no model to be connected until the user adds the first Sketch object.)

initialize

```
sketches := SelectionInList with: OrderedCollection new.  
sketches selectionIndexHolder onChangeSend: #changedSketch to: self.
```

```
sketchView := SketchView1 new.
```

"Basic Step 5"



Chapter 20

Custom Controllers

Choosing an Input Architecture	392
Creating a Controller Class	395
Connecting a Controller to a Model	399
Connecting a Controller to a View	400
Defining When a Controller Has Control	402
Defining What a Controller Does	405
Equipping a Controller with a Menu	409
Shifting Control to a Different Controller	411
Sensing Mouse Activity	412
Sensing Keyboard Activity	416
Getting the Cursor's Location	419

See Also

- “Custom Views” on page 375

Choosing an Input Architecture

Strategy

An input controller is the part of a user interface that responds to user-input events, typically mouse actions and keyboard activity. VisualWorks supports one type of input controller that uses a polling architecture and another type that uses an event-driven architecture. Within a given user-interface canvas, all views must employ the same input architecture. The basic steps show how to specify the desired input architecture for a canvas.

Polling controller: One type of input controller uses a loop to repeatedly check for input events, for as long as the controller retains control. (Typically, a controller retains control while the mouse cursor is within the boundaries of the associated view.) Each time the controller asks for events that have occurred since the previous iteration, it is said to be *polling* for events, hence the name *polling controller*.

Event-driven controller: The second type of input controller does not use a loop to poll for input events. Instead, it relies on the `ControlManager` to notify it whenever an input event occurs. It then decides whether the event is of interest—for example, a button widget's controller cares about mouse-button events but ignores most keyboard events. Because this type of controller is inactive except when there is a relevant input event for it to process, it is said to be driven by events, or *event-driven*. While the flow of control is dispatched to a polling controller, individual events are dispatched to an event-driven controller.

Missed mouse events: When a mouse button is pressed and released rapidly while the controller is busy processing an earlier event, the button-pressing event can be displaced in the object that stores button states (an `InputState`) before a polling controller has a chance to poll for events. As a result, a polling controller can miss some events and thus appear unresponsive. This phenomenon is usually associated with single-clicking and double-clicking maneuvers, and is often referred to as *missed mouse events*. An event-driven controller captures all events faithfully. This is considered the primary advantage of an event-driven controller.

Processor burden: Because a polling controller runs its polling loop even when there are no events to process, it places a greater burden on the processor. In practice, VisualWorks minimizes this difference by putting a polling controller to sleep after a brief period of fruitless polling, then awakening it when fresh input arrives.

Cross-platform portability: An event-driven controller is more sensitive to differences in event sequences across platforms, which requires more care in multi-platform applications. If you only want to be faithful to a single platform, an event-driven controller is generally preferred because the host window manager's sequence of input events is preserved.

Filing in the input events code: By default, all canvases use the polling architecture because that is the only architecture that is available in the standard VisualWorks environment. A file named `events.st` in the `extras` directory contains the code for the event-driven architecture. After you file in that code, all canvases use the event-driven architecture by default. Canvases that use a standard VisualWorks controller can use either type of input architecture, but canvases that use custom controllers will usually need to have the architecture set individually, as shown in the basic steps.

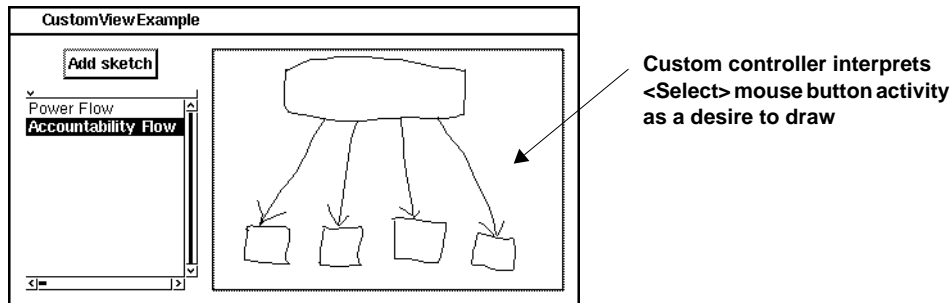
Dual-architecture controllers: A given controller class can be equipped for both input architectures. After you file in the input events code, standard VisualWorks controllers are capable of servicing either a polling or an event-driven canvas. A custom controller class also inherits from `Controller` most of the machinery for fitting into either architecture, though you will typically need to add custom protocol for each architecture. Thus, for example, you can continue to use a custom controller within the older polling style while you layer on the event-handling methods, then you can switch any canvases that use the controller to the event-driven architecture. The remaining topics in this chapter discuss the functional customizations needed to suit each input architecture.

Basic Steps

1. If necessary, file in the input events code that is supplied with VisualWorks, as described above.

2. If a Resource Finder is not already open, open one by choosing **Browse→Resources** from the VisualWorks main window.
3. In the Resource Finder, select the application class and the interface spec whose input architecture you want to set, then click on the **Edit** button. The canvas will be opened in edit mode.
4. In the Canvas Tool, click on the **Properties** button. A **Properties Tool** will be opened. Make sure no widget is selected in the canvas, so the **Properties Tool** is displaying the window properties.
5. In the **Properties Tool**, turn on the **Event Driven** check box if you want to use the event-driven input architecture for this canvas. Turn it off to use the polling architecture.
6. Save the change by clicking on **Install** in the **Canvas Tool**.

Creating a Controller Class



Strategy

Because an input controller defines the interactive character of a view, changing the controller can have a dramatic impact on the operation of a view. When an existing controller class does not serve your purposes, you can create a custom controller class, as shown in the basic steps.

Event-driven variant: An event-driven controller that responds to keyboard input needs to have an instance variable named `keyboardProcessor`, and accessor methods for that variable (`keyboardProcessor` and `keyboardProcessor:`), as shown in the variant. The controller should not initialize the variable, however, because a `KeyboardProcessor` will be supplied by the window via a `keyboardProcessor:` message.

By default, an event-driven controller does not accept keyboard focus. When it handles keyboard input, it must respond true to a `desiresFocus` message, as shown in the variant.

About the examples: In `CustomView1Example`, a simple sketch pad is created using a `SketchView1`. A `SketchView1` uses a `Sketch` as its model—a `Sketch` has a name and a collection of points representing a series of sketched strokes. A `SketchView1` uses a `SketchController1` to handle mouse and keyboard input. These four classes are all example classes that have been created to demonstrate the interactions among a domain model (`Sketch`), a custom view (`SketchView1`), a custom controller (`SketchController1`), and an application model (`CustomView1Example`).

A parallel set of example classes demonstrates the event-driven way of doing things. CustomView2Example is identical to CustomView1Example, except that its canvas has the Event Driven property turned on. SketchView2 is identical to SketchView1, except that its defaultControllerClass is SketchController2 instead of SketchController1. SketchController2 has the event-specific methods needed to service an event-driven canvas. (Separate example classes have been used for clarity — a single controller class can easily have both polling and event-driven protocol.)

To file in these classes, choose File→Browse Example Class from the Online Documentation browser and select CustomView1Example or CustomView2Example, or both.

Basic Steps

Online example: SketchController1

1. In a System Browser, display the class-definition template by selecting a class category and making sure no class is selected.
2. In the template, replace “NameOfSuperclass” with the name of the controller’s parent class (in the example, ControllerWithMenu). For guidance in choosing a superclass, consult the entry for Controller in the *VisualWorks Object Reference*.
3. Replace “NameOfClass” with the new class’s name (SketchController1).
4. Supply variable names, if any, and then accept the definition. (In the example, a variable named strokeInProgress is created to store a true/false indication of whether the user is actively sketching.)

```
ControllerWithMenu subclass: #SketchController1           "Basic Steps 1-4"
instanceVariableNames: 'strokeInProgress '
classVariableNames: "
poolDictionaries: "
category: 'Examples-Cookbook'
```

5. Add an initialize method in an instance protocol named *initialize-release*. The method is responsible for assigning an initial value to the instance variables.

```
initialize                                     "Basic Step 5"
    super initialize.
    strokeInProgress := false.
```

6. Add accessor methods in an instance protocol named *accessing*. The methods are responsible for getting and setting the value of each instance variable.

```
strokeInProgress                             "Basic Step 6"
    ^strokeInProgress
```

```
strokeInProgress: aBoolean                   "Basic Step 6"
    strokeInProgress := aBoolean
```

Variant

Creating an Event-Driven Controller Class

Online example: SketchController2

1. Do the basic steps, adding an instance variable named `keyboardProcessor` as well as accessor methods for that variable (when keyboard input is to be handled).

```
ControllerWithMenu subclass: #SketchController2    "Variant Step 1"
    instanceVariableNames: 'keyboardProcessor strokeInProgress '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Examples-Cookbook'
```

```
keyboardProcessor                             "Variant Step 1"
    ^keyboardProcessor
```

```
keyboardProcessor: kp                         "Variant Step 1"
    keyboardProcessor := kp
```

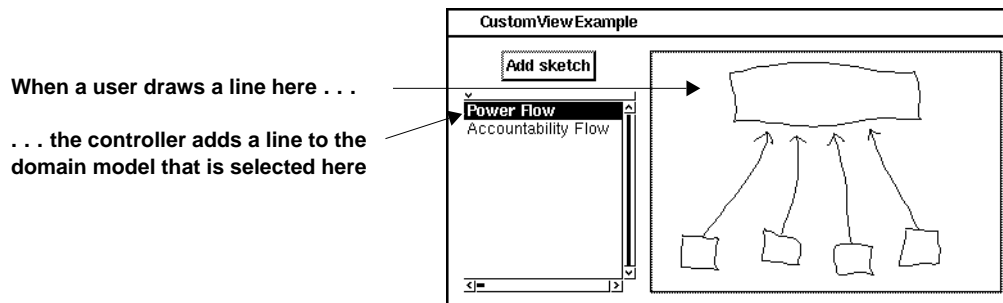
2. When keyboard input is to be handled, add a `desiresFocus` message to the controller, in a protocol named *event driven*. The method simply returns `true`, overriding the inherited method, which returns `false`.

<code>desiresFocus</code>	"Variant Step 2"
<code>^true</code>	

See Also

- “Creating a Class (Subclassing)” on page 26

Connecting a Controller to a Model



Strategy

A controller's purpose is to respond to input events. Frequently, the response involves sending a message to the view's domain model. A controller inherits a model instance variable for storing the model so it can easily access that object.

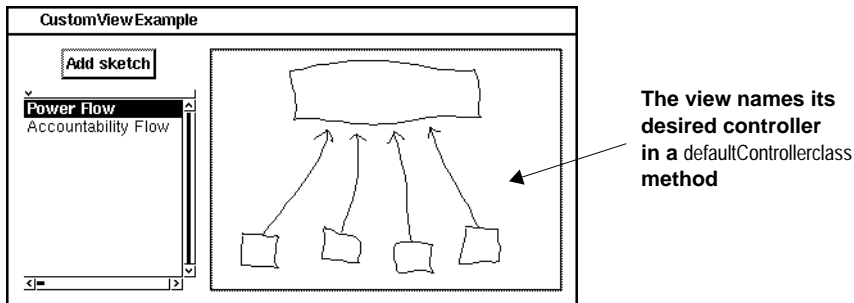
By default, a view automatically sends a model: message to its controller when it receives that message. In effect, installing the model in the view also installs it in the controller.

You can also set the controller's model explicitly, as shown in the step.

Basic Step

- Send a model: message to the controller. The argument is the model.

Connecting a Controller to a View



Strategy

An application model sometimes needs to access a view's controller. The usual way of accessing the controller is to ask the view for it. Thus, the view must itself be able to access its controller.

A view inherits an instance variable for storing its controller. By default, this instance variable is initialized automatically when a view is opened. The basic step shows how to arrange for a view to be initialized with your custom controller.

By default, the view simply sends `new` to the `defaultControllerClass` to get the controller instance. The first variant shows how to arrange for the default controller to be created in a different manner.

The second variant shows how to install a controller in a view directly.

Basic Step

Online example: `SketchView1`, `SketchController1`

- Use a System Browser to add to the view an instance method named `defaultControllerClass`, in a message category named *controller accessing*. This method must return the name of the desired controller class.

<code>defaultControllerClass</code>	"Basic Step"
<code>^SketchController1</code>	

Variants

V1. Initializing the Controller in a Nonstandard Way

Online example: CUARadioButtonView (not an example class)

- Use a System Browser to add to the view an instance method named `defaultController`, in a message category named *controller accessing*. This method must return a properly initialized instance of the desired controller class.

<code>defaultController</code>	"V1 Step"
<code>^super defaultController beTriggerOnUp</code>	

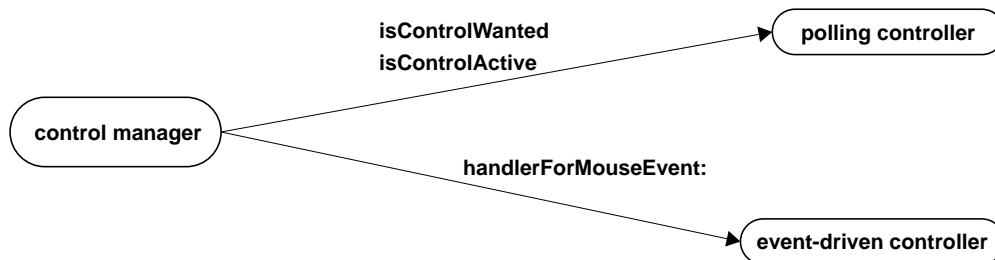
V2. Setting the View's Controller Directly

- Send a `controller: message` to the view, typically within the view's `initialize` method. The argument is an instance of the desired controller class.

See Also

- "Creating a View Class" on page 376

Defining When a Controller Has Control



Strategy

Taking control: By default, a polling controller takes control and an event-driven controller accepts mouse events whenever the mouse cursor enters the boundaries of the view. (Window and keyboard events are directed to the active widget in the active window, regardless of where the cursor is located.) This is inherited behavior, which may differ depending on the superclass you choose for your custom controller. The B1 step shows how to override that inherited behavior for a polling controller, and the B2 step shows how to override it for an event-driven controller.

Each of the example controllers, `SketchController1` and `SketchController2`, adds a stipulation that a `Sketch` must be selected in the neighboring list of sketches. It tests this by verifying that its model is not `nil`.

Yielding control: During its cycle of activity, a polling controller continually tests whether it should yield control. By default, it yields when the cursor goes outside the view's boundaries or when the window menu button is pressed. The variant shows how to redefine the test for yielding control. An event-driven controller needs no such test because it yields control whenever there are no events for it to process.

Basic Steps

B1. Taking Control (Polling Controller)

Online example: SketchController1

- Use a System Browser to create an `isControlWanted` method in the controller. This method must answer true when its conditions for taking control are met, and false otherwise.

```
isControlWanted                                     "B1 Step"
    "Answer true when the cursor is inside the view and the model is not nil."

    ^self viewHasCursor and: [self model notNil].
```

B2. Taking Control (Event-Driven Controller)

Online example: SketchController2

- Use a System Browser to create a `handlerForMouseEvent:` method in the controller. The argument is a mouse event. This method must return the controller when its conditions for handling the mouse event are met, and false otherwise. Note that this method uses `viewHasCursorWithEvent:` instead of the polling variant, `viewHasCursor`.

```
handlerForMouseEvent: aMouseEvent                   "B2 Step"
    "Answer true when the cursor is inside the view and the model is not nil."

    ^((self viewHasCursorWithEvent: event)
       and: [self model notNil])
       ifTrue: [self]
       ifFalse: [nil]
```

Variant

Defining the Test for Yielding Control

Online example: ParagraphEditor (not an example class)

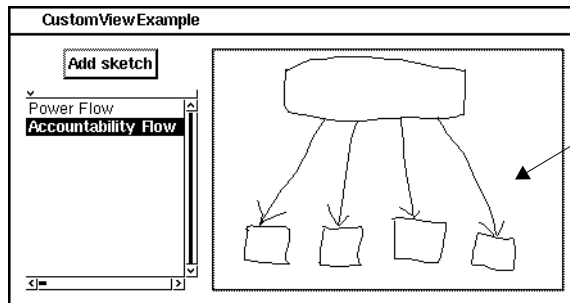
- Use a System Browser to create an `isControlActive` method in the controller. This method must answer true when its

conditions for holding onto control are met, and false when it is ready to yield.

```
isControlActive                                "Variant Step"
    "I want control only if there is a key waiting or a mouse
    button other than blue button is pressed"

    ^super isControlActive
    and: [self hasControl.
          self sensor keyboardPressed
          or: [self sensor redButtonPressed
              or: [self sensor yellowButtonPressed]]]
```

Defining What a Controller Does



This controller changes the cursor, provides a menu, provides keyboard shortcuts, and adds a stroke to the model when the user drags the mouse

Strategy

Polling controller: A controller's purpose is to sense user input and respond appropriately. For a polling controller, the inherited machinery employs a cycle of activity known as a *control loop*. During each cycle, the controller checks to make sure it still has control and then sends `controlActivity` to itself.

The inherited control activity varies depending on which superclass your custom controller has. The B1 step shows how to override the inherited behavior. Because this method is the core of a controller and defines its basic character, you will almost always need to create such a method for a custom polling controller.

Event-driven controller: Instead of using a control loop that checks for user input continuously, an event-driven controller must be prepared to respond to each type of input event individually as shown in B2. To do so, it must be equipped with a set of event methods, such as `enterEvent:`, `exitEvent:` and `keyPressedEvent:`. The `Controller` class provides default methods (see the *events* protocol), which generally do nothing, so you only need to define methods for events that your custom controller cares about. In the example, `SketchController2` responds to the following events: cursor entering or exiting the view, keyboard key being pressed, mouse being moved, or `<Select>` button being pressed or released.

Initializing and terminating: The first variant shows how to arrange for actions that need to take place only when a polling

controller accepts or yields control. An event-driven controller performs such actions in its `enterEvent:` and `exitEvent:` methods.

Changing the cursor: The second variant shows how to override the inherited control loop for a polling controller. As in the example, this technique is used mainly for adding to the basic loop by displaying a different cursor while the controller has control. The `SketchController1` displays a cross-hair cursor to indicate to the user that drawing is enabled. An event-driven controller implements a cursor change in the `enterEvent:` and `exitEvent:` methods, as shown in B2.

Event consumption: When an event-driven controller receives an event, it has the choice of passing the event on to subordinate controllers or consuming the event. An event-handling method consumes the event by returning `nil`, and passes the event on by returning the event.

Basic Steps

B1. Control Loop (Polling Controller)

Online example: `SketchController1`

- Use a System Browser to create a `controlActivity` method in the controller. This method must find out what user input event has occurred, if any, and take the appropriate action. User input is sensed by the controller's sensor.

<code>controlActivity</code>	"B1 Step"
"Check for mouse button and keyboard activity."	
"If the <Select> mouse button is being pressed, draw."	
<code>self sensor redButtonPressed</code>	
<code>ifTrue: [^self redButtonActivity].</code>	
"When not drawing, end the stroke."	
<code>self strokeInProgress: false.</code>	
"If the <Operate> button is being pressed, display the menu."	
<code>self sensor yellowButtonPressed</code>	
<code>ifTrue: [^self yellowButtonActivity].</code>	

```
"Check for keyboard input (specifically, the shortcut keys)."  
self checkForAccelerators.
```

B2. Event Methods (Event-Driven Controller)

Online example: SketchController2

- Use a System Browser to create an event-handling method for each type of event that needs custom handling. See the *events* protocol in the Controller class for the names of event methods. (The example methods define what happens when the cursor enters or exits the controller's view; other event methods are discussed later in this chapter.)

```
enterEvent: anEnterEvent "B2 Step"  
    "Change the cursor and request keyboard focus."  
  
    self keyboardProcessor requestActivationFor: self.  
    Cursor crossHair show.
```

```
exitEvent: anExitEvent "B2 Step"  
    "Change the cursor shape back to normal.  
    Also end current stroke in case red button is still being pressed."  
  
    Cursor normal show.  
    self strokeInProgress: false.
```

Variants

V1. Initializing and Terminating (Polling Controller)

Online example: ApplicationDialogController (not an example class)

1. Use a System Browser to create a `controlInitialize` method in the controller. This method has no typical usage—it can take any action that is appropriate when the view becomes active.

```
controlInitialize                                "V1 Step 1"  
    self locked: false.  
    super controlInitialize
```

2. Create a **matching controlTerminate method in the controller**. This method is a convenient place to reverse whatever actions are invoked by controlInitialize.

```
controlTerminate                                "V1 Step 2"  
    self locked: true.  
    super controlTerminate
```

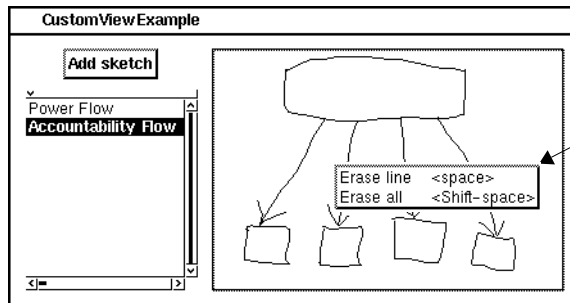
V2. Displaying a Different Cursor (Polling Controller)

Online example: SketchController1

- Use a System Browser to create a controlLoop method in the controller. This method typically displays an alternate cursor while it invokes the inherited implementation.

```
controlLoop                                    "V2 Step"  
    "Change the cursor to a cross-hair for drawing."  
  
    Cursor crossHair showWhile: [super controlLoop].
```

Equipping a Controller with a Menu



This controller provides a custom menu when the <Operate> button is pressed on the mouse

Strategy

The pop-up menu that is offered in many of the system views is maintained by each view's controller. A common motivation for creating a new controller, in fact, is simply to equip it with a menu that differs from the menu provided by the view's usual controller.

The basic steps show how to install a custom menu, for either a polling or event-driven controller. Those steps assume that your custom controller is a subclass of `ControllerWithMenu`, which provides an instance variable for a menu holder, and supporting mechanisms.

Routing menu messages: When the user of an application selects a menu item, a corresponding message is sent to the *performer*—the object that performs the associated action. By default, the performer is the controller itself. Because a controller frequently just forwards the message to the model, you can arrange for the model to receive the messages directly, saving you the trouble of defining unnecessary action methods in the controller. The variant shows how to make the model the performer.

Basic Steps

Online example: `SketchController1`

1. Use a System Browser to create an `initializeMenu` method in the controller. This has the effect of overriding the parent class's implementation.

2. In the `initializeMenu` method, send a `menuHolder:` message to the controller. The argument is a value holder containing an instance of `Menu`.

```
initializeMenu                                     "Basic Step 1"
  | mb |

  "Build the menu."
  mb := MenuBuilder new.
  mb add: 'Erase line <space>' -> #eraseLine.
  mb add: 'Erase all <Shift-space>' -> #eraseAll.

  "Install the menu."
  self menuHolder: mb menu asValue.                                     "Basic Step 2"
```

Variant

Routing Menu Messages

Online example: `SketchView1`

- Send a `performer:` message to the controller. The argument is the object to which menu messages should be sent—typically, the domain model. This message is typically sent by either the application model or the view. (In the example, the view updates the controller's performer whenever its model changes.)

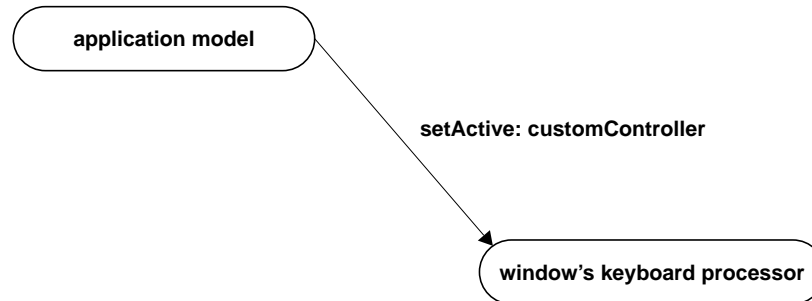
```
model: aModel
  super model: aModel.
  self invalidate.

  "Tell the controller where to send menu messages."
  self controller performer: aModel.                                     "Variant Step"
```

See Also

- “Creating a Menu” on page 226

Shifting Control to a Different Controller



Strategy

In most situations, the user determines which view is the *focus* of keyboard and mouse activity by moving the cursor into that view. Sometimes the application itself needs to shift the focus. For example, the `Can Tab` property of the standard views is implemented with this technique, enabling the user to shift focus to the next widget by pressing the `<Tab>` key.

Unless you also move the cursor to the view that is to become active, this technique assumes that the controller can take control even when the cursor is outside its view.

Basic Steps

Online example: `WidgetWrapper` (not an example class)

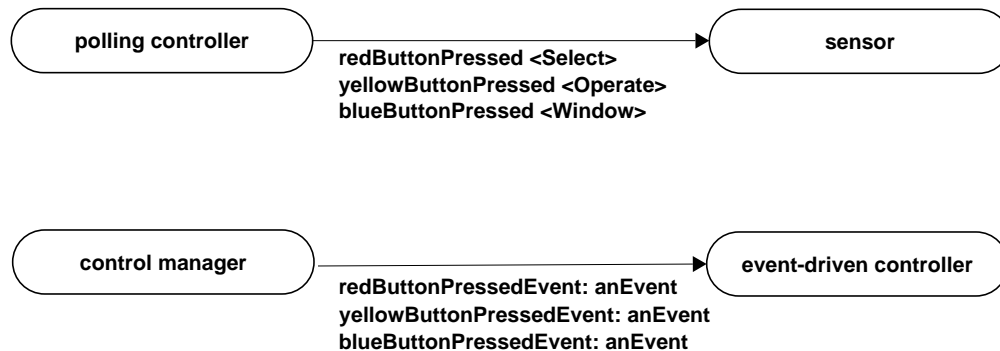
1. Get the window that contains the controller's view by sending `topComponent` to the view.
2. Get the `KeyboardProcessor` from the window by sending `keyboardProcessor` to the window.
3. Send a `requestActivationFor:` message to the keyboard processor. The argument is the controller that is to take control.

`takeKeyboardFocus`

```

(self isVisible and: [self isEnabled and: [widgetState canTakeFocus]])
  ifTrue: [self topComponent keyboardProcessor      "Basic Steps 1,2"
           requestActivationFor: self widget controller] "Basic Step 3"
  
```

Sensing Mouse Activity



Strategy

Polling controller: As part of its `controlActivity` loop, a polling controller asks its sensor whether a mouse button has been pressed. It must inquire separately for each mouse button (red, yellow or blue) that it cares about. If a button has been pressed, the controller responds appropriately, typically by invoking a `red-`, `yellow-` or `blueButtonActivity` method, as shown in B1.

Mouse movement is detected by caching the cursor location on each pass through the `controlActivity` loop and comparing that cached location with the cursor location on the next pass through the loop.

Event-driven controller: While a polling controller must ask its sensor whether each mouse button was pressed, an event-driven controller is notified directly when the control manager sends a `red-`, `yellow-` or `blueButtonPressedEvent: message`, as shown in B2. Similarly messages notify the controller of button-released events and mouse-moved events.

Button names: For both polling and event-driven controllers, the mouse buttons are identified as *redButton* (the `<Select>` button), *yellowButton* (the `<Operate>` menu button), and *blueButton* (the `<Window>` menu button). These colors correspond to colors that were on the mouse buttons on the development machine that was used by the original Smalltalk developers.

Other polling sensor messages: The variants show other tests that are available to a polling controller when asking its sensor about mouse-button activity.

Basic Steps

B1. Mouse Activity (Polling Controller)

Online example: SketchController1

1. Get the sensor from the controller.
2. Send a `redButtonPressed` message to the sensor. If true is returned, the user pressed the <Select> button on the mouse. Send `yellowButtonPressed` to find out about the <Operate> button, and `blueButtonPressed` to find out about the <Window> button.

controlActivity

"Check for mouse button and keyboard activity."

"If the <Select> mouse button is being pressed, draw."

```
self sensor redButtonPressed
  ifTrue: [^self redButtonActivity].
```

"B1 Steps 1, 2"

"When not drawing, end the stroke."

```
self strokeInProgress: false.
```

"If the <Operate> button is being pressed, display the menu."

```
self sensor yellowButtonPressed
  ifTrue: [^self yellowButtonActivity].
```

"Check for keyboard input (specifically, the shortcut keys)."

```
self checkForAccelerators.
```

B2. Mouse Activity (Event-Driven Controller)

Online example: SketchController2

1. Use a System Browser to add `red-`, `yellow-` and `blueButtonPressedEvent:` methods to the controller (assuming the controller has a separate response for each of the three mouse buttons), in an instance protocol named *events*. The

method is responsible for taking the appropriate action. (In the example, a new stroke is initiated in the sketch when the red button is pressed.)

```
redButtonPressedEvent: aRedButtonPressedEvent          "B2 Step 1"  
    "Start drawing a new line when the <Select> button is pressed."
```

```
self model beginStroke.  
self strokeInProgress: true.
```

```
self model add: (self sensor cursorPointFor: aRedButtonPressedEvent)
```

2. Add similar red-, yellow- or blueButtonReleasedEvent: methods if the controller takes separate actions when the buttons are released. (In the example, the current stroke in the sketch is ended when the red button is released.)

```
redButtonReleasedEvent: aRedButtonReleasedEvent        "B2 Step 2"  
    "Stop drawing when the <Select> button is released."
```

```
self strokeInProgress: false.
```

3. Add a similar mouseMovedEvent: method if the controller takes a separate action each time the mouse is moved. (In the example, the controller tests whether a stroke is in progress and, if so, it records the cursor location as a new point on the current stroke.)

```
mouseMovedEvent: aMouseMovedEvent                      "B2 Step 3"  
    "Add a new point for every mouse movement when drawing is in progress."
```

```
self strokeInProgress  
    ifTrue: [self model  
        add: (self sensor cursorPointFor: aMouseMovedEvent)]
```

Variants

V1. Any Mouse Button (Polling Controller)

- Send anyButtonPressed to the controller's sensor. If true is returned, one of the mouse buttons was pressed.

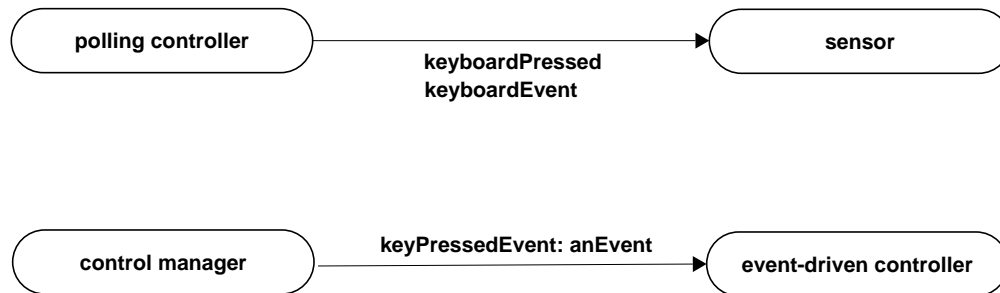
V2. No Mouse Button (Polling Controller)

- Send `noButtonPressed` to the controller's sensor. If true is returned, none of the mouse buttons was pressed.

V3. Non-`<Window>` Button (Polling Controller)

- Send `nonBlueButtonPressed` to the controller's sensor. Because a view's controller rarely responds to the `<Window>` button, this is used to distinguish window-related events from view-related events.

Sensing Keyboard Activity



Strategy

When a controller's view manipulates text, the controller needs to detect keyboard activity and respond appropriately. A controller might also want to detect keyboard activity in a nontextual view to invoke the actions associated with command shortcuts, as shown in the basic steps.

In `SketchController1`, for example, the `<Space>` key is interpreted as a shortcut for the Erase line command in its menu. Similarly, `<Shift><Space>` is a shortcut for Erase all.

Polling controller: As part of its `controlActivity` loop, a polling controller asks its sensor whether a key has been pressed. If so, it asks the sensor for the queued `KeyboardEvent`, as shown in B1.

Event-driven controller: While a polling controller must ask its sensor whether a key was pressed and, if so, ask for the queued `KeyboardEvent`, an event-driven controller is handed the event directly as the argument to a `keyPressedEvent: message`, as shown in B2.

Querying an event: You can find out from the `KeyboardEvent` which character was pressed (`keyValue`) and whether a modifier key such as `<Shift>` or `<Meta>` was pressed at the same time (`hasAlt` or `hasMeta`). See the *accessing* and *testing* protocols in the `KeyboardEvent` class for messages that can be sent to the event.

Dispatch table: The example shows a technique for enabling menu shortcuts when the set of shortcuts is small. A more sophisticated approach involves the use of a dispatch table to

associate actions with keyboard keys. That approach is used by `ParagraphEditor` and its subclasses, which are used by the text views in the system. For an example of that approach, see the class method named `initializeDispatchTable` in `ParagraphEditor`.

Basic Steps

B1. Keyboard Input (Polling Controller)

Online example: `SketchController1`

1. Get the sensor by sending `sensor` to the controller. This is typically done in the controller's `controlActivity` method or, as in the example, a method invoked by `controlActivity`.
2. Find out whether a key was pressed by sending a `keyboardPressed` message to the sensor. If `true` is returned, a keyboard key was pressed since the last time the controller checked.
3. Get the keyboard event by sending `keyboardEvent` to the sensor. (To get the event without removing it from the event queue, send a `keyboardPeek` message.)
4. Get the character by sending `keyValue` to the keyboard event. If a known character was pressed, that character is returned. If a known special key was pressed, such as `<Help>`, a symbol naming that key (`#Help`) is returned. Otherwise, an integer is returned that identifies the key in a platform-dependent way.
5. Find out whether a modifier key was pressed at the same time. Valid messages for testing the state of modifier keys are: `hasAlt`, `hasCtrl`, `hasLock`, `hasMeta`, and `hasShift`.

`checkForAccelerators`

"If a keyboard shortcut was used, tell the model to take the appropriate action."

```
| event char |
self sensor keyboardPressed                                "B1 Step 2"

"A keyboard key was pressed -- check for <Space>."
ifTrue: [
    event := self sensor keyboardEvent.                    "B1 Step 3"
    char := event keyValue.                                 "B1 Step 4"
```

```

(char == Character space)

"<Space> was pressed -- check for <Shift> key."
ifTrue: [event hasShift                                "B1 Step 5"

    "<Shift> was pressed -- erase all."
    ifTrue: [self model eraseAll]

    "<Shift> was not pressed -- erase line."
    ifFalse: [self model eraseLine]].

```

B2. Keyboard Input (Event-Driven Controller)

Online example: SketchController2

- Use a System Browser to add a `keyPressedEvent:` method to the controller, in an instance protocol named *events*. The method is responsible for testing the given `KeyPressedEvent` for conditions that the controller cares about, and then taking the appropriate action. (In the example, a space character causes the most recent stroke to be erased from the sketch, and a shifted space erases the entire sketch.)

```

keyPressedEvent: aKeyPressedEvent                    "B2 Step"
    "Respond to the <Space> key."

    | char |
    self model == nil ifTrue: [^nil].

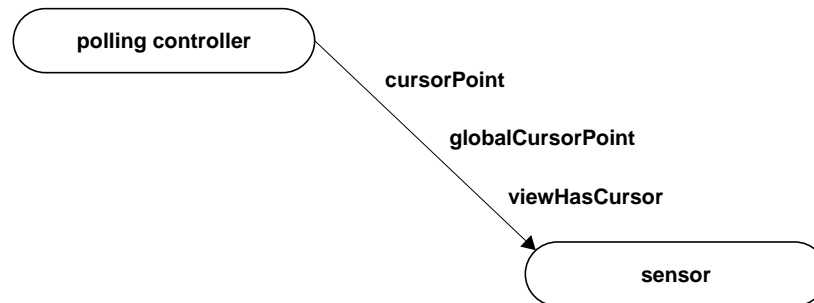
    char := aKeyPressedEvent keyValue.
    (char == Character space)
        ifTrue: [aKeyPressedEvent hasShift
            ifTrue: [self model eraseAll]
            ifFalse: [self model eraseLine]].

```

See Also

- “Adding a Shortcut Key” on page 252

Getting the Cursor's Location



Strategy

Polling controller: In drawing applications, especially, the controller often needs to supply the model with the location of the cursor. In `SketchController1`, for example, the cursor location determines the coordinates of a point that is added to the model's collection of sketch points.

The basic steps show how to get the coordinates of the cursor relative to the upper-left corner of the view. The first variant shows how to get the location relative to the upper-left corner of the screen.

Event-driven controller: For an event-driven controller, the cursor location is obtained from a mouse event. For example, after the user moves the mouse, a `mouseMovedEvent:` message is sent to the controller. The argument `MouseMovedEvent` knows the cursor location and the state of the mouse buttons at the time of the event. See the *accessing* and *testing* protocols in `MouseEvent` and `MouseMovedEvent`.

View has cursor: In some situations, the controller does not need to have the cursor's coordinates, it only needs to know whether the cursor is within the boundaries of the view. The second variant shows how to determine this conveniently.

Waiting for a button: The third variant shows how to put a polling controller into a waiting state until the user presses a button, releases a button, or both presses and releases a button. The cursor's location is returned. This can be used to retain control while the user clicks on a target outside the view,

such as another window. An event-driven controller can achieve the equivalent

Basic Steps

B1. Cursor Location (Polling Controller)

Online example: SketchController1

1. Get the sensor by sending `sensor` to the controller.
2. Send a `cursorPoint` message to the sensor. An instance of `Point` is returned, containing the coordinates of the cursor relative to the origin of the view.

`redButtonActivity`

"If necessary, begin a new stroke."

`self strokeInProgress`

`ifFalse: [`

`self model beginStroke.`

`self strokeInProgress: true].`

"Give the cursor's location to the model."

`self model add: self sensor cursorPoint.`

"B1 Step 2"

B2. Cursor Location (Event-Driven Controller)

Online example: SketchController2

1. Get the sensor by sending `sensor` to the controller.
2. Send a `cursorPointFor: message` to the sensor, with a mouse event as the argument. An instance of `Point` is returned, containing the coordinates of the cursor relative to the origin of the view.

`mouseMovedEvent: aMouseEvent`

"Add a new point for every mouse movement

when drawing is in progress."

`self strokeInProgress`

`ifTrue: [self model`

`add: (self sensor cursorPointFor: aMouseEvent)] "B2 Step 2"`

Variants

V1. Getting the Global Cursor Location

- For a polling controller, send a `globalCursorPoint` message to the controller's sensor to get the cursor location relative to the screen's origin. For an event-driven controller, send a `globalPoint` message to the mouse event. (To get the cursor location relative to the containing window, send a `point` message to the mouse event.)

V2. Testing Whether the Cursor Is Within the View

- For a polling controller, send a `viewHasCursor` message to the controller. If `true` is returned, the cursor is inside the view's boundaries. For an event-driven controller, send a `viewHasCursorWithEvent:` message to the controller, with the input event as the argument.

V3. Waiting for a Mouse Button

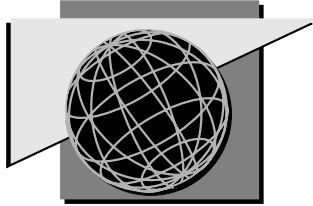
- Send a `waitButton` message to the controller's sensor. The user can then move the cursor anywhere, including beyond the controller's view, without disrupting the current controller's focus. When the user depresses any mouse button, the cursor's location is returned, relative to the view's origin. If a button is already depressed, send `waitNoButton` to wait until the button is released. Send `waitClickButton` to wait until a button is both pressed and released.

```
"Inspect"
| sensor window |
sensor := ScheduledControllers activeController sensor.
```

```
Cursor bull showWhile: [sensor waitButton].                                "V3 Step"
window := Screen default
    windowAt: sensor globalCursorPoint.
^window
```

See Also

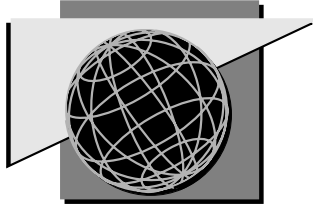
- “Creating a Cursor” on page 678
- “Changing the Current Cursor” on page 681



Part III

Data Structures

Chapter 21: Numbers	425
Chapter 22: Dates	461
Chapter 23: Times	477
Chapter 24: Collections	489
Chapter 25: Characters and Strings	529
Chapter 26: Text and Fonts	555
Chapter 27: Text Files	591
Chapter 28: Object Files (BOSS)	613
Chapter 29: Geometrics	629
Chapter 30: Images, Cursors, and Icons	657
Chapter 31: Color	685
Chapter 32: Adapting Domain Models to Widgets	703



Chapter 21

Numbers

Creating a Number	426
Adding and Subtracting	431
Multiplying and Dividing	432
Rounding	434
Getting Squares and Roots	436
Comparing Two Numbers	438
Getting the Minimum and Maximum	441
Performing Trigonometric Functions	442
Performing Logarithmic Functions	444
Testing Numberness, Evenness, Zeroness	445
Accessing and Converting the Sign	447
Converting a Number to Another Form	449
Factoring	453
Generating a Random Number	454
Accessing Numeric Constants	458

See Also

- “Formatting Displayed Data” on page 129

Creating a Number

Strategy

VisualWorks provides classes for integers, floating-point numbers, fractions, and fixed-point numbers.

Integers come in three varieties: `SmallInteger`, `LargePositiveInteger`, and `LargeNegativeInteger`. In practice, only the most demanding numeric applications need to be aware of the distinctions among the three types of integer. VisualWorks converts a value from one form to another as needed.

A floating-point number has a decimal point and at least one digit before the decimal and at least one digit after the decimal. A `Float` is a single-precision number, with six to seven digits of precision. A `Double` is a double-precision number having 14 to 15 digits of precision.

A `Fraction` has an integral numerator and an integral denominator, separated by a division slash. Fractions are always reduced to lowest terms—for example, $4/6$ is reduced to $2/3$.

A fixed-point number (an instance of `FixedPoint`) is useful for business applications in which a fixed number of decimal places is required.

Basic Steps

Creating an Integer

You can create integers as numeric literals or as the result of arithmetic operations involving one or more integers.

1. Create an integer using a literal expression.
2. Create another integer as a result of the arithmetic operation `+`.

```
"Print it"
|x y |
x := 100.                                "Basic Step 1"
y := 5.
^x + y                                   "Basic Step 2"
```

Variants

V1. Creating a Floating-Point Number

You can create floating-point numbers (instances of `Float`) as numeric literals or as the result of arithmetic operations involving one or more instances of `Float`. Note that after you create a floating-point number, you can use `integerPart` and `fractionPart` to access its parts separately.

1. Create a floating-point number using a literal expression that includes the predecimal digits, the decimal point, and the postdecimal digits.
2. Create another floating-point number as a result of the arithmetic operation `+`.

"Print it"	
x y	
x := 100.2.	"V1 Step 1"
y := 5.3.	
^x + y	"V1 Step 2"

V2. Creating a Double-Precision Floating-Point Number

You can create double-precision floating-point numbers (instances of `Double`) as numeric literals or as the result of arithmetic operations involving one or more instances of `Double`.

1. Create a double-precision floating-point number by placing the letter `d` after a floating-point number.
2. Create another `Double` as the result of the arithmetic operation `+`.

"Print it"	
x y	
x := 5.5d.	"V2 Step 1"
y := 5555.5555d.	
^x + y	"V2 Step 2"

V3. Creating a Fraction

You can create fractions by dividing integer literals or as the result of arithmetic operations involving one or more fractions. After you create a fraction, you can use `numerator` and `denominator` to access its parts separately.

1. Create a fraction by dividing one integer by another. The division slash, like other binary messages, can either be separated from the two integers by white space or not.
2. Create another fraction by sending a `numerator:denominator:` message with integer arguments to the `Fraction` class. If a noninteger argument is provided, it will be converted to an integer (the fractional part will be ignored).
3. Create another `Fraction` as the result of the arithmetic operation `+`.

```

"Print it"
|x y|
x := 1/2.                                "V3 Step 1"
y := Fraction                             "V3 Step 2"
    numerator: 3
    denominator: 4.
^x + y                                    "V3 Step 3"

```

V4. Creating a Fixed-Point Number

You can create fixed-point numbers as numeric literals or as the result of arithmetic operations involving one or more such numbers.

1. Create a fixed-point number by placing the letter *s* after a literal integer or a floating-point number. The number of decimal places preceding the *s* implicitly specifies *scale* of the number (the number of decimal places to be preserved).
2. Create another fixed-point number by specifying the scale explicitly. To do this, append the letter *s* as in step 1, and specify the desired scale after the letter *s*. Note that an explicit scale takes precedence over an implicit one, so that `99.95s4` is the same as `99.9500s`, while `99.9500s2` is an error.
3. Create another fixed-point number by coercing a nonliteral number. To do this, send an `asFixedPoint:` message to the

original number. The message argument is the scale (the number of decimal places to be preserved).

```
"Print it"
| fixed1 fixed2 fixed3 |

fixed1 := 99.95s.                                "V4 Step 1"

fixed2 := 99.95s4.                              "V4 Step 2"

fixed3 := (fixed1 * fixed2 * 0.075) asFixedPoint: 2. "V4 Step 3"

^fixed3
```

V5. Creating a Number with Scientific Notation

You can create numbers with scientific notation by using numeric literals or by obtaining the result of arithmetic operations involving one or more such numbers. A number that is created with scientific notation is stored and displayed in its normal form, as either an integer or a floating-point number.

1. Create a number with scientific notation by placing the letter *e* after a literal integer or a floating-point number. Indicate the power of 10 for the number after the letter *e*.
2. Create a Double with scientific notation by using the letter *d* in place of the letter *e*. (Note that you can also use the letter *q* instead of *d*. The letter *q* stands for quad-precision, and is available for portability to other Smalltalk systems; however, in VisualWorks, *q* has the same effect as *d*.)

```
"Print it"
| x y |
x := 1.555e3.                                    "V5 Step 1"
y := -3.955d2.                                  "V5 Step 2"
^x + y
```

V6. Creating a Number with Radix Notation

You use radix notation for a number whose base is not 10. You can create numbers with radix notation by using numeric

literals or by obtaining the result of arithmetic operations involving one or more such numbers. A number that is created using radix notation is stored and displayed in its normal form, as either an integer or a floating-point number.

1. Create a number with radix notation by indicating the base, followed by the letter *r* and the number as represented in that base.

"Print it"

| x y |

x := 2r101.

"V6 Step 1"

y := 16r1A.

"V6 Step 1"

^x + y

Adding and Subtracting

Variants

V1. Adding (+)

- Send a + message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 100.  
y := 5.  
^x + y
```

"V1 Step"

V2. Subtracting (-)

- Send a - message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 100.  
y := 5.  
^x - y
```

"V2 Step"

Multiplying and Dividing

Variants

V1. Multiplying (*)

- Send an * message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 100.  
y := 5.  
^x * y
```

"V1 Step"

V2. Dividing (/)

- Send a / message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 99.  
y := 5.  
^x / y
```

"V2 Step"

V3. Dividing and Discarding the Remainder (//)

- Send a // message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 99.  
y := 5.  
^x // y
```

"V3 Step"

V4. Dividing and Answering the Remainder (\)

- Send a \ message to a number. The argument is another number.

"Print it"

| x y |

x := 99.

y := 5.

^x \y

"V4 Step"

Rounding

Variants

V1. Rounding to the Nearest Integer

- Send a rounded message to the number.

```
"Print it"  
| x |  
x := -5.5.  
^x rounded
```

"V1 Step"

V2. Rounding Down (toward Negative Infinity)

- Send a floor message to the number.

```
"Print it"  
| x |  
x := 5.8.  
^x floor
```

"V2 Step"

V3. Rounding Up (toward Positive Infinity)

- Send a ceiling message to the number.

```
"Print it"  
| x |  
x := 5.2.  
^x ceiling
```

"V3 Step"

V4. Rounding Toward Zero

- Send a truncated message to the number.

```
"Print it"  
| x |  
x := -5.8.  
^x truncated
```

"V4 Step"

V5. Rounding to the Nearest Multiple of a Value

- Send a `roundTo:` message to the number. The argument is a number indicating the desired granularity. (In the example, `x` is rounded to the nearest hundredth.)

```
"Print it"  
| x |  
x := 5555.55555.  
^x roundTo: 0.01
```

"V5 Step"

V6. Truncating to the Nearest Multiple of a Value

- Send a `truncateTo:` message to the number. The argument is a number indicating the desired granularity. In the example, `x` is truncated (rounded toward zero) to the nearest hundredth.

```
"Print it"  
| x |  
x := 5555.55555.  
^x truncateTo: 0.01
```

"V6 Step"

See Also

- “Formatting Displayed Data” on page 129

Getting Squares and Roots

Variants

V1. Squaring a Number

- Send a squared message to the number.

```
"Print it"  
| x |  
x := 5.5.  
^x squared                                     "V1 Step"
```

V2. Taking the Square Root of a Number

- Send a sqrt message to the number.

```
"Print it"  
| x |  
x := 5.5.  
^x sqrt                                       "V2 Step"
```

V3. Raising a Number to a Power (**)

- Send a ** message to the number. The argument is the power to which the number is to be raised.

```
"Print it"  
| x |  
x := 5.  
^x ** 3                                       "V3 Step"
```

V4. Taking a Root of a Number (**)

- Send a ** message to the number. The argument is a fractional value indicating the root. (In the example, the third root of x is derived.)

```
"Print it"  
| x |
```

x := 125.
 $x^{1/3}$

"V4 Step"

Comparing Two Numbers

Variants

V1. Testing for Equality

Be careful when testing two limited-precision numbers (Float and Double) for equality—the limits of their precision must be taken into account.

- Send an = message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 5.5.  
y := 5.5.  
^x = y
```

"V1 Step"

V2. Testing for Identity

This test is faster than testing for equality. It answers true when the receiver and the argument are the same object, which is always true with equal numbers. Do not use this test, however, with limited-precision numbers.

- Send an == message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 5.  
y := 5.  
^x == y
```

"V2 Step"

V3. Testing for Inequality

- Send a ~= message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 5.5.  
y := 5.6.  
^x ~= y
```

"V3 Step"

V4. Testing for Nonidentity

This test is faster than testing for inequality, but it should not be used with limited-precision numbers (Float and Double).

- Send a `~~` message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 5.  
y := 6.  
^x ~~ y
```

"V4 Step"

V5. Testing for “Less Than”

- Send a `<` message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 5.  
y := 6.  
^x < y
```

"V5 Step"

V6. Testing for “Less Than or Equal To”

- Send a `<=` message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 5.
```

```
y := 5.  
^x <= y
```

"V6 Step"

V7. Testing for "Greater Than"

- Send a > message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 6.  
y := 5.  
^x > y
```

"V7 Step"

V8. Testing for "Greater Than or Equal To"

- Send a >= message to a number. The argument is another number.

```
"Print it"  
| x y |  
x := 5.  
y := 5.  
^x >= y
```

"V8 Step"

V9. Testing for Inclusion in a Range

- Send a between:and: message to a number. The first argument is a number indicating the beginning of the range. The second argument is a number indicating the end of the range. The arguments are included in the range.

```
"Print it"  
| x y z |  
x := 50.  
y := 1.  
z := 100.  
^x between: y and: z
```

"V9 Step"

Getting the Minimum and Maximum

Variants

V1. Getting the Minimum of Two Numbers

- Send a `min:` message to one of the numbers to be compared. The argument is the second number to be compared. The lesser number is returned.

```
"Print it"  
| x y |  
x := 5.5.  
y := 6.5.  
^x min: y
```

"V1 Step"

V2. Getting the Maximum of Two Numbers

- Send a `max:` message to one of the numbers to be compared. The argument is the second number to be compared. The greater number is returned.

```
"Print it"  
| x y |  
x := 5.5.  
y := 6.5.  
^x max: y
```

"V2 Step"

Performing Trigonometric Functions

Variants

V1. Sine

1. Convert to radians an angle expressed in degrees.
2. Send a sin message to the angle.

```
"Print it"  
| x |  
x := 45 degreesToRadians.           "V1 Step 1"  
^x sin                             "V2 Step 2"
```

V2. Cosine

- Send a cos message to a number representing an angle, expressed in radians.

```
"Print it"  
| x |  
x := 45 degreesToRadians.  
^x cos                             "V2 Step"
```

V3. Tangent

- Send a tan message to a number representing an angle, expressed in radians.

```
"Print it"  
| x |  
x := 45 degreesToRadians.  
^x tan                             "V3 Step"
```

V4. ArcSine

- Send an arcSin message to a number representing a sine value. The result is an angle expressed in radians, which can be converted to degrees if desired.

```
"Print it"
|x y |
x := 45 degreesToRadians sin.
y := x arcSin radiansToDegrees.
^y
```

V5. ArcCosine

- Send an arcCos message to a number representing a cosine value. The result is an angle expressed in radians, which can be converted to degrees if desired.

```
"Print it"
|x y |
x := 45 degreesToRadians cos.
y := x arcCos radiansToDegrees.
^y
```

V6. ArcTangent

- Send an arcTan message to a number representing a tangent value. The result is an angle expressed in radians, which can be converted to degrees if desired.

```
"Print it"
|x y |
x := 45 degreesToRadians tan.
y := x arcTan radiansToDegrees.
^y
```

Performing Logarithmic Functions

V1. Getting the Base 10 Log of a Number

- Send a log message to the number.

```
"Print it"  
| x |  
x := 5.5.  
^x log                                     "V1 Step"
```

V2. Getting a Log with a Specified Base

- Send a log: message to the number. The argument is a number indicating the base.

```
"Print it"  
| x |  
x := 5.5.  
^x log: 7                                 "V2 Step"
```

V3. Getting a Natural Log

- Send an ln message (with a lowercase *l*, not an uppercase *l*) to the number.

```
"Print it"  
| x y |  
x := 5.5 ln.  
y := x exp.                               "V3 Step"  
^y
```

V4. Getting a Number's Exponential

- Send an exp message to the number.

```
"Print it"  
| x |  
x := 5.5.  
^x exp                                     "V4 Step"
```

Testing Numberness, Evenness, Zeroness

Strategy

Because variables have no declared type in VisualWorks, it is sometimes prudent to test a variable that is expected to hold a number, as shown in the first variant. If it does hold a number, you can safely send arithmetic and other number messages to it.

The other variants show how to test various qualities of a number: whether it is an integer, whether it is even or odd, and whether it is zero.

Basic Step

Testing for Numberness

- Send a `respondsToArithmetic` message to the object (which can be any type of object). If the object is a number, it responds `true`.

```
"Print it"
|x|
x := 55.
^x respondsToArithmetic                                "Basic Step"
```

Variants

V1. Testing for Integerness

- Send an `isInteger` message to the number.

```
"Print it"
|x|
x := 55.
^x isInteger                                           "V1 Step"
```

V2. Testing for Evenness

- Send an `even` message to the number.

```
"Print it"  
| x |  
x := 56.  
^x even
```

"V2 Step"

V3. Testing for Oddness

- Send an odd message to the number.

```
"Print it"  
| x |  
x := 55.  
^x odd
```

"V3 Step"

V4. Testing for Zeroness

- Send an isZero message to the number.

```
"Print it"  
| x |  
x := 0.  
^x isZero
```

"V4 Step"

Accessing and Converting the Sign

Variants

V1. Testing for Zero or Greater

- Send a positive message to the number.

```
"Print it"  
| x |  
x := 55.  
^x positive
```

"V1 Step"

V2. Testing for Greater Than Zero

- Send a strictlyPositive message to the number.

```
"Print it"  
| x |  
x := 55.  
^x strictlyPositive
```

"V2 Step"

V3. Testing for Less Than Zero

- Send a negative message to the number.

```
"Print it"  
| x |  
x := -55.  
^x negative
```

"V3 Step"

V4. Accessing the Sign as a Multiplier

- Send a sign message to the number. The returned value is 1 when the number is greater than zero, -1 when the number is less than zero, or 0. This value can be used to convert another number, such as the absolute value of the receiver, to the same sign.

```
"Print it"  
| x y |
```

```
x := -55.  
y := x abs.  
^y * (x sign) "V4 Step"
```

V5. Reversing the Sign

- Send a negated message to the number.

```
"Print it"  
| x |  
x := -55.  
^x negated "V5 Step"
```

Converting a Number to Another Form

Variants

V1. Converting to Fixed Point

- Send an `asFixedPoint:` message to the number; the argument is the number of decimal places.

```
"Print it"
|x|
x := 99.95 asFixedPoint: 2.           "V1 Step"
^x
```

V2. Converting to Single-Precision Float

- Send an `asFloat` message to the number.

```
"Print it"
|x|
x := 55.
^x asFloat                           "V2 Step"
```

V3. Converting to Double-Precision Float

- Send an `asDouble` message to the number.

```
"Print it"
|x|
x := 55.
^x asDouble                           "V3 Step"
```

V4. Converting to Integer or Fraction (Rational Number)

- Send an `asRational` message to the number.

```
"Print it"
|x|
```

```
x := 55.5.  
^x asRational
```

"V4 Step"

V5. Converting to Absolute Number

- Send an abs message to the number.

```
"Print it"  
| x |  
x := -55.5.  
^x abs
```

"V5 Step"

V6. Converting to Reciprocal

- Send a reciprocal message to the number.

```
"Print it"  
| x |  
x := 0.5.  
^x reciprocal
```

"V6 Step"

V7. Converting from Degrees to Radians

- Send a degreesToRadians message to the number.

```
"Print it"  
| x |  
x := 90.  
^x degreesToRadians
```

"V7 Step"

V8. Converting from Radians to Degrees

- Send a radiansToDegrees message to the number.

```
"Print it"  
| x |  
x := 1.5.  
^x radiansToDegrees
```

"V8 Step"

V9. Converting to Symmetric Point

- Send an `asPoint` message to the number. The point that is returned has the number as both its *x* and *y* coordinates.

```
"Print it"
|x|
x := 55.
^x asPoint                                     "V9 Step"
```

V10. Converting to One Axis of a Point

- Send an `@` message to the number. The receiver is the *x* coordinate and the argument is the *y* coordinate of the resulting point. Like other binary messages, the `@` message can either have white space before and after it or not.

```
"Print it"
|x y|
x := 55.
y := 100.
^x @ y                                         "V10 Step"
```

V11. Converting to Character

- Send an `asCharacter` message to an integer. If the receiver is the numeric representation of a valid character, the character is returned; otherwise, an error results.

```
"Print it"
|x|
x := 55.
^x asCharacter                                 "V11 Step"
```

V12. Converting to String

- Send a `printString` message to the number.

```
"Print it"
|x|
```

```
x := 55.  
^x printString
```

"V12 Step"

V13. Converting to String, Using a Base Other Than 10

- Send a `printStringRadix: message` to the number. The argument is a number indicating the base.

```
"Print it"  
| x |  
x := 55.  
^x printStringRadix: 16
```

"V13 Step"

See Also

- "Formatting Displayed Data" on page 129

Factoring

Variants

V1. Getting the Greatest Common Divisor

- Send a gcd: message to one of the two numbers. The argument is the second number.

```
"Print it"  
| x y |  
x := 5.  
y := 10.  
^x gcd: y
```

"V1 Step"

V2. Getting the Least Common Multiple

- Send an lcm: message to one of the two numbers. The argument is the second number.

```
"Print it"  
| x y |  
x := 5.  
y := 8.  
^x lcm: y
```

"V2 Step"

V3. Getting the Factorial of a Number

- Send a factorial message to the number.

```
"Print it"  
| x |  
  
x := 5.  
^x factorial
```

"V3 Step"

Generating a Random Number

Strategy

A random number can be generated by an instance of `Random`. This object is a kind of stream, so it responds to stream messages—in particular, the `next` message gets the next number in the sequence, as shown in the basic steps.

By default, a random stream returns fractional values between 0 and 1. The first variant shows how to convert those values to a range, in effect generating a value between, say, 1 and 52.

`VisualWorks` provides seven different streams of random numbers, identified as generators 1 through 7. You can also choose a position in the stream by supplying a *seed* value—often, `Time millisecondClockValue` is used for this purpose because it is a value that varies with time. By default, a different seed value is used each time you send a new message to the `Random` class.

The second variant shows how to use the same generator and seed value when you want a reproducible sequence of “random” numbers. The third variant shows how to ensure that a second sequence of numbers is not the same as the first by changing the generator, the seed value, or both.

Basic Steps

Generating a Random Number between 0 and 1

1. Create a random stream of numbers by sending `new` to the `Random` class.
2. Get the next number in the stream by sending `next` to the random stream.

```
"Print it"
| randomStream x |
randomStream := Random new.                                "Basic Step 1"
x := randomStream next.                                     "Basic Step 2"
^x
```

Variants

V1. Generating a Random Integer in a Specified Range

1. Create a random stream of numbers by sending `new` to `Random`.
2. Define the beginning and ending values of the range.
3. Derive the extent of the range.
4. Get the next value from the random stream, then multiply it by the extent of the range, add the range's beginning value, and round the result.

```
"Print it"
| randomStream rangeStart rangeEnd rangeExtent x |
randomStream := Random new.           "V1 Step 1"
rangeStart := 1.                       "V1 Step 2"
rangeEnd := 52.
rangeExtent := rangeEnd - rangeStart.  "V1 Step 3"
x := (randomStream next * rangeExtent + rangeStart) rounded. "V1 Step 4"
^x
```

V2. Reproducing a Sequence of Random Numbers

1. Create the first random stream by sending a `fromGenerator:seededWith:` message to the `Random` class. The first argument is an integer from 1 to 7, identifying one of the seven streams that `VisualWorks` provides. The second argument is a number that is used to select a position in the stream.
2. Create the second random stream exactly the same way as in step 1—that is, use the same two values for the arguments to assure that the same generator and the same seed value are used.

```
"Print it"
| rangeStart rangeEnd rangeExtent randomStream1 randomStream2
firstSequence secondSequence |
rangeStart := 1.
rangeEnd := 52.
rangeExtent := rangeEnd - rangeStart.
```

```
"Create the two equivalent generators."
randomStream1 := Random                                "V2 Step 1"
  fromGenerator: 1
  seededWith: 1.
randomStream2 := Random                                "V2 Step 2"
  fromGenerator: 1
  seededWith: 1.

"Collect the first 1000 numbers from generator 1."
firstSequence := OrderedCollection new.
1000 timesRepeat: [
  firstSequence
  add: (randomStream1 next * rangeExtent + rangeStart) rounded].

"Collect the first 1000 numbers from generator 2."
secondSequence := OrderedCollection new.
1000 timesRepeat: [
  secondSequence
  add: (randomStream2 next * rangeExtent + rangeStart) rounded].

"Answer whether the two collections are the same."
^firstSequence = secondSequence
```

V3. Encouraging Randomness in Multiple Sequences

1. Create the first random stream by sending a `fromGenerator:seededWith:` message to the `Random` class. The first argument is an integer from 1 to 7, identifying one of the seven streams that `VisualWorks` provides. The second argument is a number that is used to select a position in the stream.
2. Create each of the other random streams in the same way, but use a different value for the generator, the seed value, or both.

```
"Print it"
| rangeStart rangeEnd rangeExtent randomStream1 randomStream2
randomStream3 numbers outStream |
rangeStart := 1.
```

```

rangeEnd := 52.
rangeExtent := rangeEnd - rangeStart.

"Create three nonequivalent generators."
randomStream1 := Random "V3 Step 1"
    fromGenerator: 1
    seededWith: 1.
randomStream2 := Random "V3 Step 2"
    fromGenerator: 2
    seededWith: 1.
randomStream3 := Random "V3 Step 2"
    fromGenerator: 1
    seededWith: Time millisecondClockValue.

"Collect the first 10 numbers from each generator."
numbers := OrderedCollection new.
10 timesRepeat: [
    numbers add: (randomStream1 next * rangeExtent + rangeStart) rounded.
    numbers add: (randomStream2 next * rangeExtent + rangeStart) rounded.
    numbers add: (randomStream3 next * rangeExtent + rangeStart) rounded].

"Arrange the random numbers in columns."
outStream := (" writeStream) cr.
1 to: 30 do: [:i |
    outStream nextPutAll: (numbers at: i) printString; tab.
    (i \ 3) isZero ifTrue: [outStream cr]].
^outStream contents

```

Accessing Numeric Constants

Variants

V1. Accessing Zero

1. Send a zero message to any numeric class. Note that the type of zero returned varies—for example, Float returns 0.0 and Integer returns 0.
2. To get a zero of the same class as an existing number, first get the class of that number by sending a class message to it and then send zero to the resulting object.

```
"Print it"  
| x y z |  
x := Float zero.           "V1 Step 1"  
y := Integer zero.  
z := x class zero.        "V1 Step 2"  
^x + y + z
```

V2. Accessing One

1. Send a unity message to any numeric class. Note that the type of one returned varies—for example, Float returns 1.0 and Integer returns 1.
2. To get a one of the same class as an existing number, first get the class of that number and then send unity to the resulting object.

```
"Print it"  
| x y z |  
x := Float unity.         "V2 Step 1"  
y := Integer unity.  
z := x class unity.      "V2 Step 2"  
^x + y + z
```

V3. Accessing Pi

1. Send a pi message to the Float or Double class. Note that Float returns a single-precision version while Double returns a double-precision version.
2. To get a pi of the same class as an existing number, first get the class of that number and then send pi to the resulting object.

```
"Print it"
| x y z |
x := Float pi.           "V3 Step 1"
y := Double pi.
z := x class pi.       "V3 Step 2"
^x + y + z
```

V4. Accessing the Largest SmallInteger

This value is frequently used to specify an arbitrarily large number whose exact value is not important. For example, the ComposedText class uses this value to set its default composition width—in effect, turning off any semblance of word wrapping.

- Send a maxVal message to the SmallInteger class.

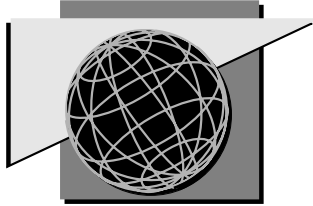
```
"Print it"
| x |
x := SmallInteger maxVal.   "V4 Step"
^x
```

V5. Accessing the Smallest SmallInteger

This value is seldom used.

- Send a minVal message to the SmallInteger class.

```
"Print it"
| x |
x := SmallInteger minVal.  "V5 Step"
^x
```



Chapter 22

Dates

Creating a Date	462
Getting Information about a Day	465
Getting Information about a Month	467
Getting Information about a Year	469
Adding and Subtracting with Dates	471
Comparing Dates	473
Formatting a Date	475

See Also

- “Times” on page 477

Creating a Date

Basic Step

Creating Today's Date

- Send a today message to the Date class.

```
"Print it"
| date |
date := Date today.                                "Basic Step"
^date
```

Variants

V1. Creating a Date from a String

- Send a readFromString: message to Date. The argument is a string containing the month, day, and year in any of several formats. The string can begin with either the month or the day—if the month is expressed as an integer, it must be in the first position. The year is always last. The month can be either a number (1 through 12) or the unique first letters of the name (case is irrelevant). The month, day, and year can be separated by a space, comma, hyphen, slash, period, or nothing.

```
"Print it"
| dates |
dates := OrderedCollection new.

dates
  add: (Date readFromString: 'January 31, 1994');    "V1 Step"
  add: (Date readFromString: '31 January 1994');    "V1 Step"
  add: (Date readFromString: '1/31/94');            "V1 Step"
  add: (Date readFromString: '1.31.1994');          "V1 Step"
  add: (Date readFromString: '1-31-1994');          "V1 Step"
  add: (Date readFromString: '31JAN94').            "V1 Step"
^dates
```

V2. Creating a Date from a Day, Month, and Year

1. Send a `newDay:monthNumber:year:` message to the `Date` class. The `newDay` argument is the day number. The `monthNumber` argument is the month number. The year argument is the year, with or without the century part.
2. To specify the month by name, send a `newDay:month:year:` message to `Date`. The month argument is the unique first letters of a month name expressed as a Symbol.

```
"Print it"
| date1 date2 |

date1 := Date
    newDay: 31
    monthNumber: 1
    year: 1994.                                     "V2 Step 1"

date2 := Date
    newDay: 31
    month: #Jan
    year: 1994.                                     "V2 Step 2"

^date1 = date2
```

V3. Creating a Date by Specifying the Days Since January 1

During a series of date computations that span several months in the same year, it can be helpful to treat a date as the number of days that it represents since the beginning of the year. After the computation is completed, you can convert the day-count back into a date.

- Send a `newDay:year:` message to `Date`. The first argument is the number of days from the beginning of the year. The second argument is the year number.

```
"Print it"
| date |
date := Date
    newDay: 32                                     "V3 Step"
```

```
year: 1994.  
^date
```

V4. Creating a Date by Specifying the Days Since 1901

During a series of date computations that span multiple years, it can be helpful to treat a date as the number of days that it represents since 1901 (in effect, the beginning of recorded time). After the computation is completed, you can convert the day-count back into a date.

- Send a `fromDays:` message to `Date`. The argument is the number of days from the beginning of 1901.

```
"Print it"  
| date |  
date := Date  
    fromDays: (94 * 366).  
^date
```

"V4 Step"

See Also

- “Creating a Time” on page 478

Getting Information about a Day

Basic Step

Getting the Day of the Week

- Send a weekday message to a date. The name of the week day is expressed as a Symbol, such as #Friday.

```
"Print it"  
| date |  
date := Date today.  
^date weekday
```

"Basic Step"

Variants

V1. Getting the Day of the Month

- Send a dayOfMonth message to a date. The day number within the month is returned.

```
"Print it"  
| date |  
date := Date today.  
^date dayOfMonth
```

"V1 Step"

V2. Getting the Day of the Year

- Send a day message to a date. The day number within the year is returned.

```
"Print it"  
| date |  
date := Date today.  
^date day
```

"V2 Step"

V3. Counting the Days Since 1901 Began

- Send an asDays message to a date. The day number within the century is returned.

```
"Print it"  
| date |  
date := Date today.  
^date asDays
```

"V3 Step"

V4. Counting the Seconds Since 1901 Began

In computations that involve both times and dates, it can be useful to represent both as a quantity of seconds.

- Send an `asSeconds` message to a date. The number of seconds elapsed prior to the date in the century is returned.

```
"Print it"  
| date |  
date := Date today.  
^date asSeconds
```

"V4Step"

Getting Information about a Month

Basic Step

Getting the Name of the Month

- Send a `monthName` message to a date. The month name is expressed as a Symbol, as in `#January`.

```
"Print it"  
| date |  
date := Date today.  
^date monthName
```

"Basic Step"

Variants

V1. Getting the Number of the Month

- Send a `monthIndex` message to a date.

```
"Print it"  
| date |  
date := Date today.  
^date monthIndex
```

"V1 Step"

V2. Counting the Days in the Month

- Send a `daysInMonth` message to a date.

```
"Print it"  
| date |  
date := Date today.  
^date daysInMonth
```

"V2 Step"

V3. Getting the Number of the First Day Relative to the Year

In computations involving dates that span months, it can be useful to know how many days have elapsed in the year at the month's beginning.

- Send a `firstDayOfMonth` message to a date.

```
"Print it"  
| date |  
date := Date today.  
^date firstDayOfMonth
```

"V3 Step"

Getting Information about a Year

Basic Step

Getting the Year Number from a Date

- Send a year message to a date.

```
"Print it"  
| date |  
date := Date today.  
^date year
```

"Basic Step"

Variants

V1. Counting the Days in the Year

- Send a daysInYear message to a date.

```
"Print it"  
| date |  
date := Date today.  
^date daysInYear
```

"V1 Step"

V2. Counting the Days Remaining in the Year

- Send a daysLeftInYear message to a date.

```
"Print it"  
| date |  
date := Date today.  
^date daysLeftInYear
```

"V2 Step"

V3. Finding Out Whether a Year Is a Leap Year

- Send a leap message to a date. The result is 1 in a leap year and zero otherwise.

```
"Print it"  
| date |
```

date := Date today.
^date leap

"V3 Step"

Adding and Subtracting with Dates

Basic Step

Adding Days to a Date

- Send an `addDays:` message to a date. The argument is the number of days to be added, and can be a negative number.

```
"Print it"
| date daysToAdd |
date := Date today.
daysToAdd := 60.
^date addDays: daysToAdd
```

"Basic Step"

Variants

V1. Subtracting Days from a Date

- Send a `subtractDays:` message to a date. The argument is the number of days to be subtracted, and it can be a negative number.

```
"Print it"
| date daysToSubtract |
date := Date today.
daysToSubtract := 60.
^date subtractDays: daysToSubtract
```

"V1 Step"

V2. Getting the Number of Days between Two Dates

- Send a `subtractDate:` message to a date. The argument is the date to be subtracted, which can be either before or after the first date.

```
"Print it"
| date1 date2 |
date1 := Date today.
date2 := Date readFromString: '31 December 1999'.
^date2 subtractDate: date1
```

"V2 Step"

V3. Getting a Previous Day of the Week

- Send a previous: message to a date. The argument is the name of the preceding weekday whose date you desire, expressed as a Symbol.

```
"Print it"  
| date dayOfWeek |  
date := Date today.  
dayOfWeek := #Monday.  
^date previous: dayOfWeek
```

"V3 Step"

Comparing Dates

Basic Step

Testing Whether Two Dates Are Equal

- Send an = message to a date. The argument is the date to be compared. If the dates are equal, true is returned; otherwise, false is returned.

```
"Print it"  
| date1 date2 |  
date1 := Date today.  
date2 := Date fromDays: 1.  
^date1 = date2
```

"Basic Step"

Variants

V1. Testing Whether Two Dates Are Unequal

- Send a ~= message to a date. The argument is the date to be compared. If the dates are unequal, true is returned; otherwise, false is returned.

```
"Print it"  
| date1 date2 |  
date1 := Date today.  
date2 := Date fromDays: 1.  
^date1 ~= date2
```

"V1 Step"

V2. Testing for “Less Than”

- Send a < message to a date. The argument is the date to be compared. If the first date is earlier, true is returned; otherwise, false is returned.

```
"Print it"  
| date1 date2 |  
date1 := Date today.
```

```
date2 := Date fromDays: 1.  
^date1 < date2
```

"V2 Step"

V3. Testing for "Less Than or Equal"

- Send a <= message to a date. The argument is the date to be compared. If the first date is earlier or equal, true is returned; otherwise, false is returned.

```
"Print it"  
| date1 date2 |  
date1 := Date today.  
date2 := Date fromDays: 1.  
^date1 <= date2
```

"V3 Step"

V4. Testing for "Greater Than"

- Send a > message to a date. The argument is the date to be compared. If the first date is later, true is returned; otherwise, false is returned.

```
"Print it"  
| date1 date2 |  
date1 := Date today.  
date2 := Date fromDays: 1.  
^date1 > date2
```

"V4 Step"

V5. Testing for "Greater Than or Equal"

- Send a >= message to a date. The argument is the date to be compared. If the first date is earlier or equal, true is returned; otherwise, false is returned.

```
"Print it"  
| date1 date2 |  
date1 := Date today.  
date2 := Date fromDays: 1.  
^date1 >= date2
```

"V5 Step"

Formatting a Date

Strategy

A date can describe itself in a string having a variety of formats. The `printFormat: message` takes as its argument an array containing six elements. The six elements are interpreted as follows:

- Day's position in the string (1, 2, or 3)
- Month's position in the string (1, 2, or 3)
- Year's position in the string (1, 2, or 3)
- The separator character
- Month's format: 1 (numeric), 2 (abbreviation), or 3 (full name)
- Year's format: 1 (with century) or 2 (without century)

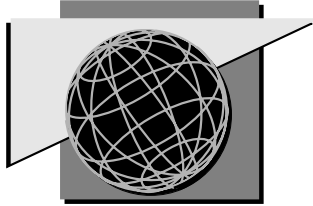
Basic Step

- Send a `printFormat: message` to the date. The argument is an array of six elements.

```
"Print it"
| date |
date := Date today.
^date printFormat: #(2 1 3 $- 3 1)                                "Basic Step"
```

See Also

- “Restricting the Type of Input” on page 125



Chapter 23

Times

Creating a Time	478
Getting the Seconds, Minutes, and Hours	480
Adding and Subtracting Times	482
Creating a Time Stamp	483
Timing a Block of Code	484
Changing the Time Zone	486

See Also

- “Dates” on page 461

Creating a Time

Basic Step

Creating the Current Time

- Send a `now` message to the `Time` class.

```
"Print it"
| time |
time := Time now.                                "Basic Step"
^time
```

Variants

V1. Creating a Time from a String

- Send a `readFromString:` message to `Time`. The argument is a string containing the hours, minutes, and seconds, separated by colons. The minutes and/or seconds can be omitted. The “am/pm” designation can be omitted (“am” is the default) and can be in upper- or lowercase.

```
"Print it"
| times |
times := OrderedCollection new.

times
  add: (Time readFromString: '3:47:26 pm');    "V1 Step"
  add: (Time readFromString: '03:47');        "V1 Step"
  add: (Time readFromString: '::26 PM').      "V1 Step"
^times
```

V2. Creating a Time by Specifying the Seconds Since Midnight

In computations involving times on different dates, it is sometimes useful to represent each time as a number of seconds

since midnight. At the end of the computation, you can convert the number of seconds back into an instance of Time.

- Send a `fromSeconds:` message to `Time`. The argument is the number of seconds that have elapsed since midnight.

```
"Print it"  
| time |  
time := Time fromSeconds: (60 * 60 * 4).           "V2 Step"  
^time
```

See Also

- “Creating a Date” on page 462

Getting the Seconds, Minutes, and Hours

Basic Step

Getting the Seconds Since the Minute Began

- Send a seconds message to the time.

```
"Print it"  
| time |  
time := Time now.  
^time seconds.                                     "Basic Step"
```

Variants

V1. Getting the Seconds Since the Day Began

In computations involving time, it is sometimes convenient to represent each time as the number of seconds since midnight.

- Send an asSeconds message to the time.

```
"Print it"  
| time |  
time := Time now.  
^time asSeconds.                                   "V1 Step"
```

V2. Getting the Seconds Since the Century Began

In time computations that span multiple days, it is sometimes convenient to represent each time as the number of seconds since 1901 began.

- Send a totalSeconds message to Time.

```
"Print it"  
| x |  
x := Time totalSeconds.  
^x                                               "V2 Step"
```

V3. Getting the Seconds Since the Millisecond Clock Was Reset

When you want to measure the number of milliseconds required by some process, you can take a reading of the millisecond clock both before and after the process. This reading is also sometimes used as a simple random number for temporary file naming and as a seed value for a random stream.

- Send a `millisecondClockValue` message to `Time`.

```
"Print it"
"Time 1000 repetitions of the millisecondClockValue method"
| x |
x := Time millisecondClockValue.
1000 timesRepeat: [Time millisecondClockValue].
^Time millisecondClockValue - x
```

"V3 Step"

V4. Getting the Minutes Since the Hour Began

- Send a `minutes` message to the time.

```
"Print it"
| time |
time := Time now.
^time minutes.
```

"V4 Step"

V5. Getting the Hours Since the Day Began

- Send an `hours` message to the time.

```
"Print it"
| time |
time := Time now.
^time hours.
```

"V5 Step"

Adding and Subtracting Times

Variants

V1. Adding Times (and Dates)

- Send an `addTime:` message to a time. The argument is either a time or a date.

```
"Print it"  
| time1 time2 |  
time1 := Time readFromString: '5'.  
time2 := Time readFromString: '8:51:39 am'.  
^time1 addTime: time2
```

"V1 Step"

V2. Subtracting Times (and Dates)

- Send a `subtractTime:` message to a time. The argument is either a time or a date.

```
"Print it"  
| time1 time2 |  
time1 := Time readFromString: '5'.  
time2 := Time readFromString: '8:51:39 am'.  
^time2 subtractTime: time1
```

"V2 Step"

Creating a Time Stamp

Strategy

When an application needs to record the date and time that an event occurred, the `Time` class provides a `dateAndTimeNow` method for that purpose. It returns an array containing two elements: the current date and the present time. You can store the array itself as a time stamp, or you can convert it to a string as shown in the basic steps.

You can achieve the same result by using `Date today` and `Time now` separately to obtain the current date and the present time. However, since each of those methods invokes `dateAndTimeNow`, the cost is an extra invocation of `dateAndTimeNow`.

Basic Steps

1. Send a `dateAndTimeNow` message to the `Time` class.
2. If desired, convert the resulting array to a string.

```
"Print it"
| dateTime |
dateTime := Time dateAndTimeNow.           "Basic Step 1"
^(dateTime at: 1) printString, ' ', (dateTime at: 2) printString. "Basic Step 2"
```

Timing a Block of Code

Strategy

During the optimization phase of application development, it is frequently useful to compare the running time of alternate versions of the same code. The Advanced Programming ObjectKit, which is available as an add-on to VisualWorks, provides a Profiler tool for measuring execution times and reporting the consumption in a detailed breakdown by method. That tool helps you isolate the methods that consume suspicious amounts of execution time, so you can focus your optimizing efforts.

In simple circumstances, when you already know which method you want to test, you can use a utility provided by the `Time` class. That class provides a `millisecondsToRun:` method, which reports the number of milliseconds it takes to execute a block containing one or more Smalltalk expressions.

Basic Steps

1. Create a `BlockClosure` containing one or more expressions to be tested. Repeating the expressions through `timesRepeat:` usually improves the validity of the comparison.
2. Send a `millisecondsToRun:` message to the `Time` class. The argument is the block you created in step 1.
3. Repeat steps 1 and 2 for the second version of the code. If the second version is not ready yet, you can simply record the value from step 2 for later comparison.

```
"Print it"
| block1 block2 ms1 ms2 |
```

```
"Test the speed of Time now and Date today."
block1 := [100 timesRepeat: [Time now. Date today]].      "Basic Step 1"
ms1 := Time                                             "Basic Step 2"
      millisecondsToRun: block1.
```

```
"Test the speed of dateAndTimeNow, which does the same thing."
block2 := [100 timesRepeat: [Time dateAndTimeNow]].      "Basic Step 3"
```

```
ms2 := Time  
  millisecondsToRun: block2.  
  
^ms1 printString, ", ms2 printString
```

Changing the Time Zone

Strategy

On machines that report Greenwich Mean Time (GMT) rather than local time, the `Time` class converts GMT to local time with the aid of another class, `TimeZone`. A `TimeZone` stores an offset from GMT for local time. In some parts of the world, this offset is an integral number of hours; in other places it is not. Both kinds of offset are handled by `TimeZone`.

`TimeZone` provides an algorithm for determining whether Daylight Saving Time (DST) is in effect. The algorithm relies on parameters that you can change to suit your needs. By default, DST is in effect from 2 a.m. on the Sunday preceding April 7 to 2 a.m. on the Sunday preceding October 31.

The basic steps show how to change the day of the week on which DST begins and ends.

The first variant shows how to create a new `TimeZone` and install it as the system default.

The second variant shows how to install a null time zone, for machines that return local time rather than GMT.

In a few locations, the algorithm for determining the beginning and ending of DST is different from the algorithm that `TimeZone` uses. To accommodate such a time zone, you will need to modify the instance method named `convertGMT:do:` in `TimeZone`.

Basic Steps

1. Get the default time zone by sending a default message to the `TimeZone` class.
2. Send a `weekDayToStartDST` message to the default time zone. The argument is the name of the day on which DST is to begin and end, in the form of a `Symbol`.
3. Use the `File→Save As` command to save your image.

zone	
zone := TimeZone default.	"Basic Step 1"
zone weekDayToStartDST: #Saturday.	"Basic Step 2"

Variants

V1. Installing a New Time Zone as the System Default

The example shows the parameters for California (the default settings in the delivered product).

1. Create a new instance of `TimeZone` by sending a `timeDifference:DST:at:from:to:startDay:` message to the `TimeZone` class. The arguments are:
 - `timeDifference` is the offset in hours from GMT.
 - `DST` is the number of hours by which DST differs.
 - `at` is the number of hours after midnight at which DST begins and ends.
 - `from` is the number of the day on which DST begins in a nonleap year (it will be adjusted automatically during leap years).
 - `to` is the number of the day on which DST ends in a nonleap year.
 - `startDay` is the day of the week on which DST begins and ends.
2. Send a `setDefaultTimeZone:` message to the `TimeZone` class. The argument is the time zone you created in step 1.
3. Use the `File→Save As` command to save your image.

```
| newZone |
newZone := TimeZone
timeDifference: -8
DST: 1
at: 2
from: 97
to: 304
startDay: #Sunday.

```

"V1 Step 1"

```
TimeZone setDefaultTimeZone: newZone.

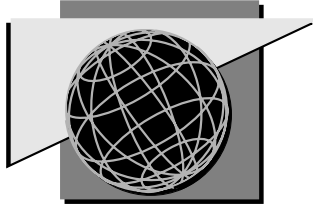
```

"V1 Step 2"

V2. Installing a Null TimeZone

1. Create a null time zone—one that does not alter the time returned by your computer—by sending a null message to the `TimeZone` class.
2. Send a `setDefaultTimeZone:` message to the `TimeZone` class. The argument is the time zone you created in step 1.
3. Use the `File→Save As` command to save your image.

nullZone	
nullZone := TimeZone null.	"V2 Step 1"
TimeZone setDefaultTimeZone: nullZone.	"V2 Step 2"



Chapter 24

Collections

Choosing the Right Collection	490
Creating a Collection	491
Getting the Size	495
Adding Elements	497
Removing Elements	500
Replacing Elements	505
Copying Elements	508
Combining Two Collections	510
Finding Elements	511
Comparing Collections	517
Sorting a Collection	519
Converting to a Different Type of Collection	522
Looping through the Elements (Iterating)	524

See Also

- “Characters and Strings” on page 529
- “Lists” on page 183

Choosing the Right Collection

Strategy

VisualWorks contains a wealth of specialized collection classes, as described in the *VisualWorks Object Reference*. For most situations, however, one of four types of collection will suffice:

- List
- Array
- Set
- Dictionary

A List is the most flexible type of collection and the most commonly used. It keeps elements in the order in which they were added. You can add a single element or a collection of elements, and the additions can be inserted anywhere. A List can also be sorted. A List is typically the collection that is held by a SelectionInList or MultiSelectionInList, as the model for a list or notebook widget. (List supersedes the older OrderedCollection and SortedCollection.)

An Array is a very simple collection that is efficient in situations that do not require adding, removing, or sorting elements. However, an array does support replacing of elements.

A Set is another simple collection whose main distinguishing feature is that it discards duplicate elements. This is useful when you want to be able to add an element without first having to make sure that element was not added previously. Unlike the other three types of collections, a set does not support replacing of elements, so it is not useful in situations requiring such changes.

A Dictionary is useful for lookup tables and similar collections. Each element in a Dictionary is an Association, which is a lookup key (usually a Symbol) paired with a value (any type of object).

Creating a Collection

Strategy

Typically, you create an empty collection, as shown in the basic step.

For an `Array`, which cannot add elements, the equivalent is to specify the size of the array, as shown in the first variant. Each element will be `nil` until your application replaces it with another object. The second variant shows how to specify an alternate object with which to initialize the new collection. These techniques can be used for collection classes other than `Array` also.

You can also create a collection by specifying up to four elements, as shown in the third variant. This approach is typically used to create a small array, because it is less cumbersome than creating a `nil` array and then replacing the elements.

When an array contains only literal elements, such as numbers and strings, you can also create the array using its literal form, as shown in the fourth variant.

Sometimes a new collection needs to be created from an existing collection. For example, a nongrowing array might need to be expanded to accommodate more elements. Or a dictionary's keys might be placed in a list for sorting. The fifth variant shows how to create a new collection that has the elements of an existing collection.

The sixth variant shows how to create a new collection from an existing collection, but with a starting size that is larger than the existing set. This is mainly useful as a means of expanding an array (which cannot grow to accept new elements).

Basic Step

- Send a new message to the desired collection class (in the example, `List`).

```
"Inspect"  
| list |  
list := List new.
```

"Basic Step"

```
list add: 'Leonardo';  
      add: 'Michelangelo';  
      add: 'Donatello';  
      add: 'Raphael'.  
^list.
```

Variants

V1. Creating a Collection of a Certain Size

- Send a new: message to the desired collection class (typically Array, but useful with other types of collections to avoid time-consuming grow operations as elements are added).

```
"Inspect"  
| array |  
array := Array new: 4.                                     "V1 Step"  
  
array at: 1 put: 'Leonardo';  
      at: 2 put: 'Michelangelo';  
      at: 3 put: 'Donatello';  
      at: 4 put: 'Raphael'.  
^array.
```

V2. Creating an Initialized Collection of a Certain Size

- Send a new:withAll: message to the desired collection class (in the example, Array).

```
"Inspect"  
^Array new: 16 withAll: 0.                                 "V2 Step"
```

V3. Creating an Array with Up to Four Elements

- Send a with:with:with:with: message to the Array class, or a variant of that message containing as many with: keywords as needed, up to four. The argument of each with: keyword can be any object. (This variant can be used with any collection class but is most often used with arrays.)

```

"Inspect"
| array1 array2 array3 array4 |
array1 := Array with: 'Leonardo'.                                "V3 Step"

array2 := Array                                                "V3 Step"
  with: 'Leonardo'
  with: 'Michelangelo'.

array3 := Array                                                "V3 Step"
  with: 'Leonardo'
  with: 'Michelangelo'
  with: 'Donatello'.

array4 := Array                                                "V3 Step"
  with: 'Leonardo'
  with: 'Michelangelo'
  with: 'Donatello'
  with: 'Raphael'.

^Array                                                         "V3 Step"
  with: array1
  with: array2
  with: array3
  with: array4.

```

V4. Creating a Literal Array

- Enclose the list of literal elements in parentheses, with a number-sign prefix. Any white-space character can be used to separate the elements.

```

"Inspect"
^#( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael' )          "V4 Step"

```

V5. Creating a New Collection from an Old Collection

- Send a withAll: message to the desired collection class (List).

"Inspect"

^List withAll: Smalltalk keys

"V5 Step"

Getting the Size

Strategy

The basic step shows how to count the elements in a collection.

Each position in which an element can be stored is known as a *slot*. A collection often has more slots than elements to avoid having to expand the collection each time a new element is added. The first variant shows how to count the slots in a collection.

Frequently, it is useful to know whether a collection is empty of elements, as shown in the second variant.

Basic Step

- Send a size message to the collection. The response is an integer (in the example, 31).

```
"Print it"
| array |
array := ColorValue constantNames.
^array size                                     "Basic Step"
```

Variants

V1. Getting the Capacity

- Send a capacity message to the collection. The response is an integer (47).

```
"Print it"
| set |
set := Set withAll: ColorValue constantNames.
^set capacity                                   "V1 Step"
```

V2. Testing for Emptiness

- Send an isEmpty message to the collection. The response is true when the collection has no elements and false otherwise.

```
| list |  
list := List allInstances.
```

```
list isEmpty                                "V2 Step"  
  ifFalse: [^list first]
```

Adding Elements

Strategy

Although an Array can contain only the elements with which it was created, a List, Set, or Dictionary can add elements at any time, as shown in the basic steps.

By default, a List adds new elements to the end of the collection. The first variant shows how to position the additional element at the beginning of the collection, before a particular element, or before a particular index. (A Set and a Dictionary do not keep their elements in an externally visible order, so the notion of inserting a new element does not apply.)

When a List or Set is used to accumulate the contents of other collections, the additions can be added in batches, as shown in the second variant. Each batch can be inserted at a specific location, as with a single-element addition.

The third variant shows a technique for expanding an array by creating a copy that has a new element appended to it. The copy can then be substituted for the original.

Basic Steps

1. Send an `add:` message to a List or Set. The argument can be any object.
2. Send an `at:put:` message to a Dictionary. The first argument is the lookup key, typically but not necessarily a Symbol. The second argument is the object that is associated with the key. (Note: A Dictionary also responds to `add:`, with an Association as the argument, but this is considered bad style by experts.)

```
"Inspect"
| list dict |
list := List new.
dict := Dictionary new.
```

```
list add: 'Leonardo';
      add: 'Michelangelo';
      add: 'Donatello';
```

"Basic Step 1"

```
add: 'Raphael'.
```

```
dict at: #Leader put: 'Leonardo';  
    at: #Member1 put: 'Michelangelo';  
    at: #Member2 put: 'Donatello';  
    at: #Member3 put: 'Raphael'.
```

"Basic Step 2"

```
^Array with: list with: dict
```

Variants

V1. Inserting an Element at a Specific Location

1. Send an `addFirst:` message to a List. The argument is the element to be inserted at the beginning of the collection.
2. Send an `add:before:` message to a List. The first argument is the element to be inserted. The second argument is the element before which the insertion is to take place.
3. Send an `add:beforeIndex:` message to a List. The first argument is the element to be inserted. The second argument is the index of the element before which the insertion is to take place.

```
"Inspect"  
| list |  
list := List new.
```

```
list add: 'Raphael';  
    addFirst: 'Leonardo';  
    add: 'Michelangelo' before: 'Raphael';  
    add: 'Donatello' beforeIndex: 3.
```

"V1 Step 1"
"V1 Step 2"
"V1 Step 3"

```
^list
```

V2. Adding a Collection of Elements

1. Send an `addAll:` message to a List or Set. The argument is a collection of elements to be added. Remember that a Set will discard duplicate elements.

2. Send an `addAllFirst:` message to a List (not a Set). The argument is the collection of elements to be inserted at the beginning of the list.
3. Send an `addAll:beforeIndex:` message to a List. The first argument is the collection to be inserted. The second argument is the index number of the element before which the new batch is to be inserted.

```

"Print it"
| sizes totalElements |
sizes := List new: 10000.

sizes addAll: (List allInstances collect: [ :list | list size]).           "V2 Step 1"
sizes addAllFirst: (Dictionary allInstances collect: [ :dict | dict size]). "V2 Step 2"
sizes                                                                    "V2 Step 3"
    addAll: (Array allInstances collect: [ :array | array size])
    beforeIndex: 2.

totalElements := 0.
sizes do: [ :sz | totalElements := totalElements + sz].
^totalElements

```

V3. Expanding an Array

1. Send a `copyWith:` message to a List or Array. The argument is the object that is to be appended to the end of the copy.
2. Replace the original array with the expanded copy.

```

"Print it"
| array copy |
array := #(1 2 3 4 5 6 7 8 9).

copy := array copyWith: 10.                                             "V3 Step 1"
array := copy.                                                         "V3 Step 2"
^array

```

Removing Elements

Strategy

The basic step shows how to remove a specified element from a List or Set. If the specified object is not an element in the collection, an error results. The first variant shows how to supply an alternative action (including doing nothing) when the object is not found.

The second variant shows how to remove a subset of a List or Set when the subset is held in a separate collection.

A List provides several messages for removing a single element at a specified position or a range of elements. The third variant shows how to use the available messages.

The fourth variant shows how to test each element in a List and remove those that meet the test conditions.

The fifth variant shows how to remove an association from a Dictionary by supplying the association's key.

The sixth variant shows a technique for removing occurrences of an object from an array. It involves making a copy of the array, omitting each occurrence of a specified object. The copy can then be substituted for the original array.

Basic Step

- Send a `remove:` message to a List or Set. The argument is the object to be removed.

```
"Print it"
```

```
| list |
```

```
list := List withAll: ColorValue constantNames.
```

```
list remove: #red.
```

```
^list
```

```
"Basic Step"
```

Variants

V1. Supplying an Alternative Element-Not-Found Response

- Send a `remove:ifAbsent:` message to a List or Set. The first argument is the object to be removed. The second argument is a block containing the action or actions.

An empty block is an effective means of taking no action—that is, simply shutting off the error notifier that is displayed by default.

```
| list |
list := List withAll: ColorValue constantNames.

list remove: #brickRed                                "V1 Step"
  ifAbsent: [Dialog warn: 'You must be kidding -- brickRed?'].

list remove: #moonbeam                                "V1 Step"
  ifAbsent: [ ].
^list
```

V2. Removing a Subset

- Send a `removeAll:` message to a List or Set. The argument is a collection containing the elements to be removed. If an element is not found, an error results. The subset can be contained in a different type of collection.

```
"Print it"
| list |
list := List withAll: ColorValue constantNames.

list removeAll: #( #red #green #blue ).                "V2 Step"
^list
```

V3. Removing an Element or Range of Elements by Index

1. Send a `removeFirst` message to a List (but not a Set, Array, or Dictionary). The first element in the list will be removed. If the list is empty, an error results.
2. Send a `removeFirst: message` to a List. The argument is the number of elements to be removed from the front of the list.
3. Send a `removeLast` message to remove the last element.
4. Send a `removeLast: message`. The argument is the number of elements to be removed from the end of the list.
5. Send a `removeFrom:to: message` to a List. The first argument is the starting index of the range and the second argument is the ending index. An array containing the deleted elements is returned.
6. Send a `removeFrom:to:returnElements: message` to a List. The first and second arguments are index numbers identifying the range to be removed. The third argument is `false` when you want `nil` to be returned instead of an array containing the deleted elements—for large removal operations, this is more efficient.

```
"Print it"
| list |
list := List new: 25.
1 to: 25 do: [ :i | list add: i].
```

```
"Removes 1"
list removeFirst.                                "V3 Step 1"
```

```
"Removes 2 3 4 5 6"
list removeFirst: 5.                              "V3 Step 2"
```

```
"Removes 25"
list removeLast.                                  "V3 Step 3"
```

```
"Removes 20 21 22 23 24"
list removeLast: 5.                               "V3 Step 4"
```

```
"Removes 14 15 16 17 18"
list removeFrom: 8 to: 12.                         "V3 Step 5"
```

```
"Removes 9 10 11 12 13"
list removeFrom: 3 to: 7 returnElements: false.           "V3 Step 6"

^list
```

V4. Removing All Elements That Pass a Test

- Send a `removeAllSuchThat:` message to a List. The argument is a block containing the test. The block must declare one argument variable for the element to be tested. (In the example, all color names beginning with the letter *r* are removed.)

```
| list |
list := List withAll: ColorValue constantNames.

list removeAllSuchThat: [ :name | name first == $r].       "V4 Step"
^list
```

V5. Removing an Association from a Dictionary

1. Send a `removeKey:` message to the dictionary. The argument is the key of the association that you want to remove. The removed value is returned. If the key is not found, an error results.
2. To provide an alternative response to the key-not-found condition, send a `removeKey:ifAbsent:` message to the dictionary. The first argument is the key to be removed and the second argument is a block that specifies the action to take if the key is not found. An empty block causes no action, which is the same as silently ignoring the condition.

```
"Inspect"
| dict |
dict := Dictionary new.
dict at: #Leader put: 'Leonardo';
      at: #Member1 put: 'Michelangelo';
      at: #Member2 put: 'Donatello';
      at: #Member3 put: 'Raphael'.
```

```
dict removeKey: #Member2. "V5 Step 1"
```

```
dict removeKey: #Villain ifAbsent: []. "V5 Step 2"  
^dict
```

V6. Removing an Element from an Array

1. Send a `copyWithout:` message to an Array (or a List). The argument is the object to be removed. Every occurrence of that object will be removed from the copy.
2. Replace the original array with the copy.

```
"Print it"  
| array copy |  
array := #(1 8 3 4 5 6 7 8 9).
```

```
copy := array copyWithout: 8. "V6 Step 1"  
array := copy. "V6 Step 2"  
^array
```

Replacing Elements

Strategy

A Set does not support replacing of elements, because there is no index number or key for specifying which element to replace. A Dictionary supports a single form of replacement, in which you specify the lookup key and provide a new value, as shown in the basic steps. List and Array support a similar form of replacement, but the lookup key is the index number of the element you want to replace, as shown in the basic steps.

When you want to replace all elements of a List or Array with a particular object, use the technique shown in the first variant. To replace only those elements with specified index numbers, use the second variant.

To replace all occurrences of a specified object with a new object, use the third variant.

To replace a subset of a List or Array with a new set of elements, use the fourth variant.

Basic Steps

1. Send an `at:put:` message to a Dictionary. The first argument is the lookup key. The second argument is the value to be placed at that key. If the key does not exist, it will be added.
2. Send an `at:put:` message to a List or Array. The first argument is the index number of the element to be replaced. The second argument is the object that is to replace the old element.

```
"Inspect"
| list dict |
dict := Dictionary new.
dict at: #Leader put: 'Leonardo';
    at: #Member1 put: 'Michelangelo';
    at: #Member2 put: 'Donatello';
    at: #Member3 put: 'Raphael'.
list := List withAll: dict values.
list sort.
```

```
dict at: #Leader put: 'Rembrandt'.
```

```
"Basic Step 1"
```

```
list at: 1 put: 'Rembrandt'.  
^Array with: list with: dict.
```

Variants

V7. Replacing All Elements

- Send an `atAllPut:` message to a List or Array. The argument is the object that is to replace all existing elements.

```
"Print it"  
| list |  
list := List new.  
1 to: 10 do: [ :number | list add: number ].
```

```
list atAllPut: 0.  
^list
```

V8. Replacing Specified Elements

- Send an `atAll:put:` message to a List or Array. The first argument is a collection containing the index numbers of the elements to be replaced. The second argument is the object to be placed in those slots.

```
"Print it"  
| list |  
list := List new.  
list  
  add: 'red';  
  add: 'ghoulishGreen';  
  add: 'red';  
  add: 'blackAndBlue'.
```

```
list atAll: #( 1 3 ) put: 'bloodRed'.  
^list
```

V9. Replacing All Occurrences of an Object

- Send a `replaceAll:with:` message to a List or Array. The first argument is the object whose occurrences you want to replace. The second argument is the replacement object.

```
"Print it"
| list |
list := List new.
list
  add: 'red';
  add: 'ghoulishGreen';
  add: 'red';
  add: 'blackAndBlue'.
```

```
list replaceAll: 'red' with: 'bloodRed'.
^list
```

"V9 Step"

V10. Replacing a Subset with a New Subset

- Send a `replaceFrom:to:with:startingAt:` message to a List or Array. The first and second arguments are index numbers identifying the replacement range. The `with:` argument is a collection containing the new elements. The `startingAt:` argument is the index number in the new collection at which to begin copying the replacement elements.

```
"Print it"
| mainList replacements |
mainList := #( 1 2 3 4 5 6 7 8 9 ).
replacements := #( 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 ).
```

```
mainList
  replaceFrom: 1
  to: mainList size
  with: replacements
  startingAt: 7.
```

```
^mainList
```

"V10 Step"

Copying Elements

Strategy

A collection, like any other object, can provide a copy of itself, as shown in the basic step. You can modify literal elements such as numbers and strings without affecting the copy. For a nonliteral element, however, the copied collection holds onto the same object rather than a copy and will reflect any changes you make to that object. One way around this problem is to replace the element with a copy of itself—you can modify this copied object freely without affecting the similar element in the copied collection. In the case of a Dictionary, you must remove the key and then add it with the new value to avoid affecting the copied association, as shown in the basic step.

To copy a subset of a List or Array, use the technique shown in the variant.

Basic Step

- Send a copy message to the collection.

```
"Inspect"
| dict1 dict2 |
dict1 := Dictionary new.
dict1 at: #Leader put: 'Leonardo';
      at: #Member1 put: 'Michelangelo';
      at: #Member2 put: 'Donatello';
      at: #Member3 put: 'Raphael'.

dict2 := dict1 copy.                                     "Basic Step"

"Change the original without changing the copy."
dict1 removeKey: #Leader.
dict1 at: #Leader put: 'Rembrandt'.

^Array with: dict1 with: dict2
```

Variants

V1. Copying a Subset

- Send a `copyFrom:to:` message to a List or Array. The first argument is the starting index of the range you want to copy, and the second argument is the ending index.

```
"Print it"  
| list copy |  
list := List new.  
1 to: 10 do: [ :number | list add: number ].
```

```
copy := list copyFrom: 1 to: 3.                                     "V1 Step"  
^copy
```

Combining Two Collections

Strategy

Two ordered collections, such as a List and an Array, can be combined, as shown in the basic step. This technique is widely used with strings (which are ordered collections of characters), but it can also be used with other kinds of ordered collections.

Basic Step

- Send a comma (,) message to a List or Array. The argument is another ordered collection. A new collection will be returned, of the same type as the first collection, containing the elements of both collections.

```
"Print it"  
| list array combinedList |  
list := List withAll: ColorValue constantNames.  
array := #( #bloodRed #ghoulishGreen #blackAndBlue ).  
  
combinedList := list, array.  
^combinedList
```

Finding Elements

Strategy

Given an index location, a List or Array can supply the corresponding element, as shown in the basic step.

A Dictionary can find the value for a specified lookup key, as the first variant shows. By default, an error results if the key does not exist, so using an alternate not-found action is also shown.

A Dictionary can also find the key corresponding to a given value (that is, a reverse lookup), also shown in the first variant.

A List or Array can also do a reverse lookup, returning the index number that corresponds to a given element, as shown in the second variant. The search can be initiated at the beginning of the collection or later, or it can go backward from the end.

Any collection can tell you whether it includes a specific object, as shown in the third variant. It can either answer true when it finds the object, or it can count the occurrences.

A List can find the element that is either before or after a specified object, as shown in the fourth variant.

A List or Array can return the first or last element, as shown in the fifth variant.

To find the starting index of a subset in a List or Array, use the sixth variant.

To find elements that meet your custom conditions in any collection, use the seventh variant.

Basic Step

- Send an `at:` message to a List or Array. The argument is an index number. If the object is not found, zero is returned.

```
"Print it"
| list |
list := List withAll: Smalltalk classNames.
```

```
^list at: 1
```

```
"Basic Step"
```

Variants

V1. Searching a Dictionary

1. Send an `at:` message to the dictionary. The argument is the lookup key. If the key does not exist, an error results.
2. To avoid the key-not-found error, send an `at:ifAbsent:` message. The second argument is an empty block (for no action) or a block containing actions to be taken if the key does not exist.
3. To find the key that corresponds to a value, send a `keyAtValue:ifAbsent:` message to the dictionary. The first argument is the object whose key is to be found. The second argument is a block containing the value-not-found action.

```
"Print it"
| dict found1 found2 found3 |
dict := Smalltalk.
```

```
found1 := dict at: #List.                "V1 Step 1"
found2 := dict at: #UnlikelyClassName ifAbsent: [nil]. "V1 Step 2"
found3 := dict keyAtValue: List ifAbsent: [nil].      "V1 Step 3"
```

```
^Array with: found1 with: found2 with: found3
```

V2. Finding the Index of an Object

1. Send an `indexOf:` message to a List or Array. The argument is the object to be found. If the object is not an element, zero is returned.
2. To search a subset of the List or Array, send a `nextIndexOf:from:to:` message. The first argument is the object to be found. The second and third arguments are indexes that define the search range. The returned index is relative to the beginning of the collection.
3. To search backward from the end, send a `lastIndexOf:` message. The index of the last occurrence is returned, or zero if none exists. The returned index is relative to the beginning of the collection.

```

"Print it"
| list found1 found2 found3 |
list := List withAll: #( #red #green #blue #red #yellow #blue).

found1 := list indexOf: #red.                "V2 Step 1"
found2 := list nextIndexOf: #red from: 2 to: 6. "V2 Step 2"
found3 := list lastIndexOf: #red.           "V2 Step 3"

^Array with: found1 with: found2 with: found3

```

V3. Learning Whether an Object Is in a Collection

1. Send an `includes:` message to the collection. The argument is the object to be found. The response is `true` if the collection contains at least one matching object, and `false` otherwise. (A Dictionary can also be sent `includesKey:` and `includesAssociation:` messages.)
2. Send an `occurrencesOf:` message to the collection. The argument is the object whose matching instances among the elements are to be counted. If the object is not found, zero is returned.

```

"Print it"
| list found1 found2 |
list := List withAll: #( #red #green #blue #red #yellow #blue).

found1 := list includes: #red.                "V3 Step 1"
found2 := list occurrencesOf: #red.          "V3 Step 2"

^Array with: found1 with: found2

```

V4. Finding the Element Before or After an Object

1. Send a `before:` message to a List. The argument is the element before which the desired element is located. If the argument matches the first element, an error results.
2. Send an `after:` message to a List. The argument is the element after which the desired element is located. If the argument matches only the last element, an error results.

```
"Print it"
| list found1 found2 |
list := List withAll: #( #red #green #blue #red #yellow #blue).

found1 := list before: #blue.           "V4 Step 1"
found2 := list after: #yellow.         "V4 Step 2"

^Array with: found1 with: found2
```

V5. Finding the First or Last Element

1. To get the first element, send a first message to a List or Array. If the collection is empty, an error results.
2. To get the last element, send a last message to a List or Array. If the collection is empty, an error results.

```
"Print it"
| list found1 found2 |
list := List withAll: #( #red #green #blue).

found1 := list first.                 "V5 Step 1"
found2 := list last.                  "V5 Step 2"

^Array with: found1 with: found2
```

V6. Finding a Subset

- Send an `indexOfSubCollection:startingAt:` message to a List or Array. The first argument is the subset to be found, which need not be the same type of collection. The second argument is the index number at which the search is to begin. The returned index number is relative to the beginning of the collection. If the subset is not found, zero is returned.

```
"Print it"
| list subset found |
list := List withAll: #( #red #green #blue #red #yellow #blue).
subset := #( #red #yellow #blue).
```

```
found := list indexOfSubCollection: subset startingAt: 1.           "V6 Step"
^found
```

V7. Finding Elements That Pass or Fail a Test

1. To find all elements that pass a test, send a `select: message` to any type of collection. The argument is a block containing a test to determine whether each element is to be selected. The block is expected to declare one argument for the next collection element to be tested. A collection containing the elements that passed the test is returned.
2. To find all elements that fail a test, send a `reject: message` to any type of collection. The argument is a block containing a test to determine whether each element is to be rejected. The block is expected to declare one argument for the next collection element to be tested. A collection containing the elements that failed the test is returned.
3. To find the first element that passes a test, send a `detect:ifNone: message` to any type of collection. The first argument is a block containing the test. The block must declare one argument for the next collection element to be tested. The second argument is a no-argument block containing the action to take if no element passes the test.

```
"Inspect"
| list found1 found2 found3 |
list := List withAll: Smalltalk classNames.
```

```
"Select classes with 'Example' in their names."
found1 := list                                     "V7 Step 1"
  select: [ :nextElement |
    (nextElement indexOfSubCollection: 'Example'
      startingAt: 1) > 0].
```

```
"Reject classes with 'Example' in their names."
found2 := list                                     "V7 Step 2"
  reject: [ :nextElement |
    (nextElement indexOfSubCollection: 'Example'
      startingAt: 1) > 0].
```

```
"Detect the first class beginning with 'R'."
```

```
found3 := list                                     "V7 Step 3"  
  detect: [ :nextElement | nextElement first == $R]  
  ifNone: [0].  
  
^Array with: found1 with: found2 with: found3
```

Comparing Collections

Strategy

A collection can be compared to another collection or even to a noncollection object. The basic step shows how to test whether a collection is equal to another collection, meaning it is the same type of collection, has the same number of elements, and all of the elements are equal.

The first variant shows a more stringent test, but one that is much faster. It tests whether the two collections are the same object. Note that two collections will fail this test even if they are of the same type, have the same number of elements, and all of their elements are the same. For that reason, this more stringent test is rarely used with collections, but it is often used to compare other kinds of objects.

When you want to know which elements are unique to one of two sets or dictionaries, use the second variant.

Basic Step

- Send an = message to one of the collections. The argument is the other collection. The response is true when both collections are of the same type, have the same number of elements, and all of the elements are equal. (The example shows that a copy is equal, but a copy with one changed element is not equal.)

```
"Print it"
| list1 list2 copyIsEqual copyWithChangedElementsEqual |
list1 := List withAll: ColorValue constantNames.
list2 := list1 copy.

copyIsEqual := list1 = list2.                                     "Basic Step"

list2 at: 1 put: #burntOrange.
copyWithChangedElementsEqual := list1 = list2.

^Array with: copyIsEqual with: copyWithChangedElementsEqual.
```

Variants

V1. Comparing with the Same-Object Test

- Send an `==` message to one of the collections. The argument is the other collection (typically, another variable, which may be holding the same collection as the receiver). The response is true when the argument is the same object as the receiver. (The example shows that a copy is not the same, nor, of course, is a copy with a changed value.)

```
"Print it"
| list1 list2 copyIsSame copyWithChangedElementsSame |
list1 := List withAll: ColorValue constantNames.
list2 := list1 copy.

copyIsSame := list1 == list2.                                     "V1 Step"

list2 at: 1 put: #burntOrange.
copyWithChangedElementsSame := list1 == list2.

^Array with: copyIsSame with: copyWithChangedElementsSame.
```

V2. Subtracting One Set from Another

- Send a `-` (minus) message to a Set or Dictionary. The argument is another set or dictionary. A similar type of collection is returned, containing the elements that occur in the first set but not the second.

```
"Print it"
| set1 set2 |
set1 := Set withAll: ColorValue constantNames.
set2 := set1 select: [ :name |
    (name indexOfSubCollection: 'light' startingAt: 1) > 0].

^set1 - set2                                                     "V2 Step"
```

Sorting a Collection

Strategy

A List can be asked to rearrange its elements in ascending order, as shown in the basic step. It is assumed that the elements respond to < and = messages, which are used to compare elements during the sorting.

You can customize the sorting order by supplying a block containing the test for determining whether one element comes before another, as shown in the first variant. The block is given two elements to compare, and is expected to answer true when the first element should precede the second element.

Any collection can be sorted by converting it to an instance of SortedCollection, as shown in the second variant. Again, the default sort order is ascending, and you can supply a block to customize the sort order.

A List or Array can be inverted by creating a copy with its elements in reverse order, as shown in the third variant. No sorting is implied by this operation—when the elements are unsorted to begin with, they will remain unsorted in the reversed copy, but their order will be inverted.

Basic Step

- Send a sort message to a List. Its elements are rearranged in ascending order.

```
"Print it"
| list |
list := List withAll: #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael').

list sort.                                     "Basic Step"
^list
```

Variants

V1. Customizing the Sort Order

- Send a `sortWith:` message to a List. The argument is a block containing the test for determining whether one element precedes another. The block must declare two arguments to contain the two elements being compared. (In the example, the test causes the elements to be sorted in descending order.)

```
"Print it"
| list |
list := List withAll: #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael').

list sortWith: [ :element1 :element2 | element1 > element2].      "V1 Step"
^list
```

V2. Sorting a nonList Collection

1. Send an `asSortedCollection` message to the collection. A `SortedCollection` is returned, with the collection's elements in ascending order.
2. To customize the sort order, send an `asSortedCollection:` message to the collection. The argument is a block that compares two elements and answers `true` when the first element is to precede the second element.

```
"Inspect"
| array1 sort1 array2 sort2 |

array1 := #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael').
sort1 := array1 asSortedCollection.                                "V2 Step 1"

array2 := #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael').
sort2 := array2 asSortedCollection: [ :name1 :name2 | name1 > name2].
                                                                    "V2 Step 2"

^Array with: sort1 with: sort2.
```

V3. Reversing the Elements

- Send a reverse message to a List or Array (or any ordered collection). A new instance of the same type of collection is returned, with the elements in reverse order (but not explicitly sorted).

```
"Print it"  
| array reversedArray |  
array := #('Leonardo' 'Michelangelo' 'Donatello' 'Raphael').
```

```
reversedArray := array reverse.                                "V3 Step"  
^reversedArray
```

Converting to a Different Type of Collection

Strategy

Any collection can be converted to a List, an Array, or a Set, as shown in the three variants. Strictly speaking, the collection is not converted. Instead, an instance of the desired type of collection is returned containing the original collection's elements, and the original collection remains unchanged.

The order of the elements is random when the original collection is a Set, Dictionary, or any other type of unordered collection. One practical implication of this limitation is that a later conversion of the same collection may return a collection with the elements in a different order, which would make it unequal to the first conversion.

When a Dictionary is converted, its keys are ignored—only its values are contained in the new collection.

Variants

V1. Converting to a List

- Send an `asList` message to a collection. A List is returned that contains the original collection's elements, in the same order when possible.

```
"Inspect"
| array list |
array := ColorValue constantNames.
```

```
list := array asList.
^list.
```

"V1 Step"

V2. Converting to an Array

- Send an `asArray` message to a collection. An Array is returned that contains the original collection's elements, in the same order when possible.


```
"Inspect"  
| dict array |  
dict := Smalltalk.
```

```
array := dict asArray.                                "V2 Step"  
^array.
```

V3. Converting to a Set

- Send an `asSet` message to a collection. A Set is returned that contains the original collection's elements, minus any duplicates. This is a useful technique for removing duplicates from a collection that normally allows duplicates.

```
"Inspect"  
| array set |  
array := #( #red #green #blue #red #yellow #blue).
```

```
set := array asSet.                                  "V3 Step"  
^set
```

Looping through the Elements (Iterating)

Strategy

Frequently an application needs to perform a set of actions for each element in a collection. For example, a sales processing application might want to generate a packing slip for each element in a list of sales orders. The basic step shows how to create a loop that repeats a series of steps for each element in a collection.

Occasionally the elements in a collection need to be processed in reverse order, starting with the final element and proceeding toward the first element, as shown in the first variant.

For collections that provide an index number (List, Array) or other lookup key (Dictionary) for accessing the elements, the second variant shows how to loop through the keys instead of the values, or both the keys and values. This is especially useful with dictionaries, whose values are sometimes meaningless without the associated keys.

Frequently the actions inside a block transform the element or create a related object, and the new object is then added to a separate collection. The third variant shows a shortcut for collecting the objects that result from the processing without having to explicitly create the separate collection and add the objects to it.

Often two collections need to be processed in tandem. The fourth variant shows how to pass corresponding elements from two ordered collections into a two-argument block.

Basic Step

- Send a `do:` message to a collection. The argument is a block that performs a series of operations on an element. The block is repeated for each element in the collection, and it is expected to declare one argument variable to hold the next element to be processed.

```
| list color |  
list := List withAll: ColorValue constantNames.
```

list sort.

```
list do: [ :colorName |
    Transcript show: colorName asString; cr.
    color := ColorValue perform: colorName.
    Transcript
        show: color red printString;
        tab;
        show: color green printString;
        tab;
        show: color blue printString;
        cr; cr].
```

"Basic Step"

Variants

V1. Looping in Reverse Order

- Send a `reverseDo:` message to a collection. The argument is a block that performs a series of operations on an element. The block is repeated for each element in the collection, starting with the last and proceeding toward the first element. The block is expected to declare one argument variable to hold the next element to be processed.

```
| list color |
list := List withAll: ColorValue constantNames.
list sort.
```

```
list reverseDo: [ :colorName |
    Transcript show: colorName asString; cr.
    color := ColorValue perform: colorName.
    Transcript
        show: color red printString;
        tab;
        show: color green printString;
        tab;
        show: color blue printString;
        cr; cr].
```

"V1 Step"

V2. Looping through Lookup Keys

1. Send a `keysDo:` message to an ordered collection such as a Dictionary, List, or Array. The argument is a block that performs a series of operations using the lookup key for each element. The block is expected to declare one argument variable to hold the next key to be processed.
2. Send a `keysAndValuesDo:` message to an ordered collection such as a Dictionary, List, or Array. The argument is a block that performs a series of operations using the lookup key and associated value for each element. The block is expected to declare two argument variables to hold the next key-value pair to be processed.

```
| dict randomGenerator gc randomX randomY colorValue |
randomGenerator := Random new.
gc := (ExamplesBrowser prepareScratchWindowOfSize: 300@400)
graphicsContext.
```

```
dict := Dictionary new.
ColorValue constantNames do: [ :colorName |
colorValue := ColorValue perform: colorName.
dict at: colorName put: colorValue].
```

```
dict keysDo: [ :colorName |                                     "V2 Step 1"
randomX := randomGenerator next * 300.
randomY := randomGenerator next * 300.
colorName displayOn: gc at: (randomX @ randomY)].
```

```
dict keysAndValuesDo: [ :colorName :color |                   "V2 Step 2"
randomX := randomGenerator next * 300.
randomY := randomGenerator next * 300.
gc paint: color.
colorName displayOn: gc at: (randomX @ randomY)].
```

V3. Collecting the Results of the Processing

- Send a `collect:` message to a collection. The argument is a block that processes an element and returns an object that is to become an element in the result collection. The block is expected to declare one argument variable for the next element to be processed.

```

"Inspect"
| list capitalizedName initial |
list := List withAll: ColorValue constantNames.
list sort.

list collect: [ :colorName |                               "V3 Step"
  capitalizedName := colorName asString.
  initial := (capitalizedName at: 1) asUppercase.
  capitalizedName at: 1 put: initial.
  capitalizedName].

```

V4. Looping through Two Parallel Collections

- Send a `with:do:` message to a List or Array. The first argument is another List or Array. The second argument is a block that performs a series of operations on a pair of elements, one from each of the two collections. The block is expected to declare two argument variables, one for each of the elements. (The example creates key-value pairs for a dictionary, taking the keys from one array and the associated values from a second array.)

```

"Inspect"
| array1 array2 dict |
array1 := #( #Leader #Member1 #Member2 #Member3).
array2 := #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael' ).
dict := Dictionary new.

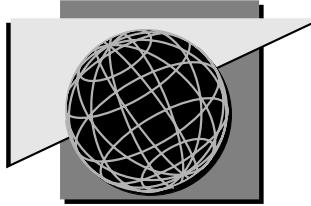
array1 with: array2 do: [ :array1Element :array2Element |       "V4 Step"
  dict at: array1Element put: array2Element].

^dict

```

See Also

- “Searching” on page 543
- “Finding Elements” on page 511



Chapter 25

Characters and Strings

Creating a Character	530
Creating a String	532
Distinguishing Types of Characters	534
Changing the Case	537
Getting a String's Length and Width	539
Comparing	540
Searching	543
Combining Two Strings	545
Extracting a Substring	547
Removing or Replacing a Substring	549
Abbreviating a String	551
Inserting Line-End Characters	553

See Also

- “Collections” on page 489
- “Text and Fonts” on page 555

Creating a Character

Strategy

To create a character as a separate entity, rather than as part of a string, use its literal form (basic step).

Certain characters cannot be created as keyboard literals, such as <Delete> and <Return>. The `Character` class provides convenience messages for creating such nondisplaying characters (first variant).

A character can also be created from its numeric equivalent (second variant). The numeric value is displayed in a character's print string.

Finally, you can create a composed character (third variant). A composed character has a base character plus a diacritical mark.

Note: Any application that manipulates characters should be prepared to encounter any character value from 0 through 65535.

Basic Step

- Precede the desired character with a dollar sign.

```
"Print it"  
| char |  
char := $C.                                "Basic Step"  
^char
```

Variants

V1. Creating a Nondisplaying Character

- Send one of the following messages to the `Character` class to create the corresponding character: `backspace`, `cr`, `del`, `esc`, `leftArrow`, `lf`, `newPage`, `space`, `tab`.

```
"Print it"  
| char |
```



```
char := Character cr.                                "V1 Step"
^char
```

V2. Creating a Character from a Numeric Code

- Send a value: message to the Character class. The argument is the numeric Unicode representation for the character.

```
"Print it"
| char |
char := Character value: 67.                          "V2 Step"
^char
```

V3. Creating a Composed Character

- Send a composeDiacritical: message to a character. The argument is a diacritical character, which can be obtained by sending diacriticalNamed: to the Character class. The argument is a symbol naming a diacritical character—see the method comment for the list of valid names.

```
"Print it"
| baseChar diacrit composedChar |
baseChar := $a.
diacrit := Character diacriticalNamed: #grave.
composedChar := baseChar composeDiacritical: diacrit.  "V3 Step"
^composedChar
```

Creating a String

Strategy

A string is usually created using its literal form, enclosed between single quotes, as shown in the basic step. (Double quotes are used to enclose a code comment in Smalltalk.) Note that the returned object will be a platform-specific subclass of String.

For initializing a string variable, an empty string is frequently used, as shown in the first variant.

Although a string will grow to accommodate newly added characters, it is more efficient to create a string of the appropriate size and then change its characters. The second variant shows how to create a string containing a specified number of characters. By default, the string is filled with null characters, but you can specify the default character.

In parsing situations, it is sometimes necessary to convert a single character into a one-character string. The third variant shows how to do this.

Basic Step

- Enclose the desired characters in single quotes.

```
"Inspect"  
| string |  
string := 'This is a string.'  
^string
```

"Basic Step"

Variants

V1. Creating an Empty String

- Send a new message to the String class. This is equivalent to enclosing nothing between single quotes.

```
"Inspect"  
| emptyString |
```

```
emptyString := String new.                                "V1 Step"
^emptyString
```

V2. Creating a String of a Certain Size

1. Send a `new:` message to the `String` class. The argument is an integer indicating how many characters are to be in the string. Each character is a null, which you can replace with another character as a separate operation.
2. To supply a default character, send a `new:withAll:` message to the `String` class. The first argument is the number of characters. The second argument is the default character that is to fill all of the string's slots.

```
"Inspect"
| nullString zeroString |

nullString := String new: 10.                            "V2 Step 1"

zeroString := String new: 10 withAll: $0.                "V2 Step 2"

^Array with: nullString with: zeroString
```

V3. Creating a String from a Character

- Send a `with:` message to the `String` class. The argument is the character that is to be the sole element of the string.

```
"Print it"
| oneCharString |
oneCharString := String with: Character tab.            "V3 Step"
^oneCharString
```

Distinguishing Types of Characters

Strategy

VisualWorks uses one encoding internally (Unicode) to represent characters, allowing you to translate to other character encodings as needed (see the *International User's Guide*). Unicode supports characters from nearly all modern and classical alphabets, as well as other special characters such as math operators and monetary symbols.

Within this extended character set, VisualWorks distinguishes several subsets, and applications that manipulate characters sometimes need to know whether a specific character belongs to a particular subset. For example, a numeric application might want to verify that a character is numeric. Characters provide a series of testing methods that conveniently answer the common queries regarding group membership.

Variants

V1. Testing for Letterness

1. Send an `isAlphabetic` message to a character. The response is true when the character is a-z or A-Z in the English alphabet.
2. Send an `isAlphaNumeric` message to a character. The response is true when the character is in a-z, A-Z, or 0-9.
3. Send an `isLetter` message to a character. The response is true when the character is in a-z, A-Z, or the set of non-English letters.
4. Send an `isVowel` message to a character. The response is true when the character is in *aeiou* or *AEIOU*, with or without diacritical marks.

```
"Print it"  
| char responses |  
char := $a.  
responses := Array new: 4.
```

```
responses
```

```

at: 1 put: char isAlphabetic;           "V1 Step 1"
at: 2 put: char isAlphaNumeric;       "V1 Step 2"
at: 3 put: char isLetter;             "V1 Step 3"
at: 4 put: char isVowel.              "V1 Step 4"
^responses

```

V2. Testing for Numberness

- Send an isDigit message to a character. The response is true when the character is in the range 0–9.

```

"Print it"
| char |
char := $5.
^char isDigit                               "V2 Step"

```

V3. Testing for Case

1. Send an isLowercase message to the character. The response is true when the character is a lowercase letter.
2. Send an isUppercase message to find out whether the character is an uppercase letter.

```

"Print it"
| char isLower isUpper |
char := $C.

isLower := char isLowercase.           "V3 Step 1"
isUpper := char isUppercase.          "V3 Step 2"

^Array with: isLower with: isUpper.

```

V4. Testing for White Spaceness

- Send an isSeparator message to the character. The response is true when the character is a space, tab, carriage return, line feed, form feed, or null.

```

"Print it"
| char |

```

```
char := Character cr.  
^char isSeparator "V4 Step"
```

V5. Testing for Composedness

1. Send an `isComposed` message to the character. The response is true when the character is composed of a base character plus a diacritical mark.
2. To find out whether a character is a diacritical mark (alone), send an `isDiacritical` message.

```
"Print it"  
| char |  
char := Character diacriticalNamed: #grave.  
^Array  
  with: char isComposed "V5 Step 1"  
  with: char isDiacritical "V5 Step 2"
```

Changing the Case

Strategy

Applications that manipulate strings sometimes need to convert one or more lowercase letters to uppercase, or vice versa. You can change the case of an entire string (basic steps). You can also change the case of a selected letter (variant)—for example, in a loop that capitalizes the initial letter in each word in a string.

Do not use case-changing protocol with strings whose characters are caseless (for example, Japanese Katakana characters).

Basic Steps

1. To convert a string to all lowercase letters, send an `asLowercase` message to the string.
2. To convert a string to all uppercase, send an `asUppercase` message.

```
"Print it"
| string |
string := 'North American Fertilizer Company'.
```

```
^string asUppercase                                     "Basic Step 2"
```

Variant

Changing the Case of a Selected Letter

1. Send an `asUppercase` message to the character. An uppercase equivalent will be returned.
2. Send an `asLowercase` message to get the lowercase equivalent.

```
"Print it"
| string prevCharIsSeparator newChar |
string := 'NORTH AMERICAN FERTILIZER COMPANY'.
prevCharIsSeparator := true.
```

```
string keysAndValuesDo: [ :index :char |
```

```
prevCharIsSeparator
  if True: [newChar := char asUppercase]      "Variant Step 1"
  if False: [newChar := char asLowercase].    "Variant Step 2"
string at: index put: newChar.
prevCharIsSeparator := char isSeparator].
```

^string

Some character sets contain single lowercase characters that become multiple characters in their uppercase form. If you are working with such a character set, your code should handle the results of asUppercase accordingly.

Getting a String's Length and Width

Strategy

A String is a kind of Collection. Its elements are characters. Counting the characters in a string is accomplished by getting the size of the collection, as shown in the first variant.

The width of a string changes depending on the font that is used to display it. Because the font choice is controlled by the graphics context of the display surface, that object can compute the width of a string, in pixels, as shown in the second variant.

Variants

V1. Counting the Characters

- Send a size message to the string.

```
"Print it"
| string |
string := '123456789'.
^string size                                     "V1 Step"
```

V2. Getting the Width in Pixels

- Send a widthOfString: message to the graphics context of the display surface on which the string will be displayed. The argument is the string. The width in pixels is returned.

```
"Print it"
| window string width |
window := ScheduledWindow new.
string := 'Hello, world'.

width := window graphicsContext
  widthOfString: string.
^width                                           "V2 Step"
```

Comparing

Strategy

Characters and strings respond to the same comparison messages as most objects: `=`, `==`, `<`, `>`, and so on (basic step and first two variants).

Characters are compared based on their numeric equivalents. Thus, `$a` is greater than `$A`.

Strings are compared as other collections are compared. They are equal when both are strings, both have the same number of characters, and both have the same characters in the same order.

Case difference makes two strings unequal but does not make one string greater than the other. For example, `'abc'` is not less than, equal to, or greater than `'ABC'`. You can treat two strings as being equal in spite of case difference (third variant).

To find out how similar two strings are, you can either count the number of leading characters that are the same, or you can derive a similarity rating on a scale of 100 (fourth variant).

Basic Step

- Send an `=` or `~=` (not equal) message to the object (character or string). The argument is a similar object.

```
"Print it"
| char1 char2 |
char1 := $a.
char2 := $A.
^char1 = char2                                     "Basic Step"
```

Variants

V1. Comparing Identities

- To compare based on identity, send an `==` or `~~` (not identical) message to the object. The argument is a similar

object. Two different strings cannot be identical, though two variables that refer to the same string are identical.

```
"Print it"
| str1 str2 str3 |
str1 := 'Excellent'.
str2 := 'Excellent'.
str3 := str1.
```

```
^Array
  with: (str1 == str2)
  with: (str1 == str3)
"V1 Step"
```

V2. Comparing by Sorting Order

1. Send a `<` (less than) or `<=` (less than or equal to) message to the object. The argument is a similar object. Remember that case differences make two strings unequal but not less than or greater than each other.
2. Send a `>` or `>=` message to compare for greater than.

```
"Print it"
| str1 str2 str3 |
str1 := 'north'.
str2 := 'North'.
str3 := 'northwest'.
```

```
^Array
  with: (str1 < str2)
  with: (str1 < str3)
  with: (str2 < str3)
"V2 Step 1"
```

V3. Comparing Strings While Ignoring Case Differences

- Send a `sameAs:` message to one of the strings. The argument is the second string.

```
"Print it"
| str1 str2 str3 |
str1 := 'north'.
```

```
str2 := 'North'.
str3 := 'northwest'.
```

```
^Array
  with: (str1 sameAs: str2)           "V3 Step"
  with: (str1 sameAs: str3)
  with: (str2 sameAs: str3)
```

V4. Rating the Similarity of Two Strings

1. Send a `sameCharacters:` message to one of the strings. The argument is the second string. An integer is returned, indicating how many of the beginning characters are the same (including case) in the two strings.
2. Send a `spellAgainst:` message to one of the strings, with the second string as argument. An integer from 1 (entirely different) through 100 (equal) is returned.

```
"Print it"
| str1 str2 str3 |
str1 := 'north'.
str2 := 'North'.
str3 := 'northwest'.
```

```
^Array
  with: (str1 sameCharacters: str2)   "V4 Step 1"
  with: (str1 sameCharacters: str3)
  with: (str1 spellAgainst: str2)    "V4 Step 2"
  with: (str1 spellAgainst: str3)
```

Searching

Strategy

The ability to find a specific character or substring is essential in applications that parse strings. Often a special character or series of characters identifies a field within a string, especially when the string represents the contents of a structured text file. The basic steps show how to find either a character (a less-than character) or a substring ('Class Variables:') within the class comment for the String class.

By default, searching is case-sensitive. The variant shows how to ignore case during a search.

The variant also shows how to use wildcard characters during a search. A pound sign (#) takes the place of any single character, and an asterisk (*) takes the place of zero or more characters.

Basic Steps

1. To get the index of a character, send an `indexOf:` message to the string. The argument is the search character. If it is not found, zero is returned.
2. To find the starting index of a substring, send a `findString:startingAt:ifAbsent:` message to the string. The first argument is the substring to be found. The second argument is the character position at which the search is to begin. The third argument is a block containing actions to be taken if the substring is not found (often an empty block, to avoid the default error).

```
"Print it"
| classComment searchChar searchString index1 index2 |
classComment := String comment.
searchChar := $<.
searchString := 'Class Variables:'.
```

```
index1 := classComment indexOf: searchChar.
index2 := classComment
    findString: searchString
```

"Basic Step 1"

"Basic Step 2"

```
startingAt: 1  
ifAbsent: [ ].
```

```
^Array with: index1 with: index2
```

Variant

Searching While Ignoring Case Difference

- Send a `findString:ignoreCase:useWildcards: message` to the string. The `findString` argument is the substring to be found. The `ignoreCase` argument is true when case difference is to be ignored. The `useWildcards` argument is true when the number sign and asterisk are to be interpreted as wildcard characters rather than literal characters. Because the presence of an asterisk wildcard affects the endpoint of the found string, this variant returns an `Interval` identifying the index range of the found string. A zero interval is returned when the search string is not found.

```
"Print it"  
| classComment searchString interval |  
classComment := String comment.  
searchString := 'Var*:'.
```

```
interval := classComment  
    findString: searchString  
    startingAt: 1  
    ignoreCase: true  
    useWildcards: true.                                "Variant Step"
```

```
^classComment  
    copyFrom: interval first  
    to: interval last
```

See Also

- “Scanning Fields in a File (Stream)” on page 609

Combining Two Strings

Strategy

In simple situations, you can combine two strings using a comma (basic step). For situations involving a large number of such concatenations, it is more efficient to use a stream, as shown in the variant. For example, when assembling a series of strings for a report, the variant would be preferable.

For cross-cultural applications, use the string expansion facility described in the *International User's Guide*.

Basic Step

- Send a , (comma) message to the first string. The argument is the second string. A new string is returned, containing the first string followed by the second string.

```
"Print it"
| firstName lastName fullName space |
firstName := 'Bill'.
lastName := 'Clinton'.
space := String with: Character space.
```

```
fullName := firstName, space, lastName.           "Basic Step"
^fullName
```

Variant

Combining Strings Using a Stream

1. Create a stream by sending an `on:` message to the `WriteStream` class. The argument is typically an empty string, but it could be any string, such as a preassembled report heading.
2. Append each string in the series to the stream by sending a `nextPutAll:` message to the stream, with the string as argument.
3. Get the stream contents in the form of a string by sending a `contents` message to the stream.

```
"Print it"
| classNames formallist |
classNames := Smalltalk classNames.
formallist := WriteStream on: String new.           "Variant Step 1"

classNames do: [ :name |
    formallist nextPutAll: 'Class: ';
    nextPutAll: name;
    cr].                                           "Variant Step 2"

^formallist contents                               "Variant Step 3"
```

Extracting a Substring

Strategy

When a string contains two or more parts, getting the parts as separate strings is a common requirement. For example, you might need to extract the first and last names from a string containing a full name. You can copy a portion of a string, using the starting and stopping character locations (basic step).

In certain situations, the only part of a string that you need is a prefix that ends at a specific character. You can copy the characters that precede a specific endpoint character (variant).

Basic Step

- Send a `copyFrom:to:` message to the string. The first argument is the starting index and the second argument is the ending index of the desired substring.

```
"Print it"
| fullName firstName lastName spaceIndex |
fullName := 'Mahatma Gandhi'.
spaceIndex := fullName indexOf: Character space.
```

```
firstName := fullName                                     "Basic Step"
  copyFrom: 1
  to: spaceIndex - 1.
lastName := fullName
  copyFrom: spaceIndex + 1
  to: fullName size.
```

```
^Array with: firstName with: lastName
```

Variant

Copying a Prefix

- Send a `copyUpTo:` message to the string. The argument is the character that marks the end of the prefix (but is not included in it).

```
"Print it"  
| fullName firstName |  
fullName := 'Boris Yeltsin'.
```

```
firstName := fullName copyUpTo: Character space.          "Variant Step"  
^firstName
```

Removing or Replacing a Substring

Strategy

A string can be quite long and complicated, representing an entire report or the contents of a lengthy text file. In long strings especially, replacing a portion of the string with a new substring is frequently useful. The basic steps show how to replace a substring based on the starting and stopping indexes.

Removing characters is accomplished by creating a copy in which the unwanted characters have been replaced by an empty string, as shown in the basic steps.

When a string contains multiple occurrences of a substring, you can replace all occurrences by using the technique shown in the variant.

Basic Steps

1. Send a `copyReplaceFrom:to:with:` message to the string. The first and second arguments are the index locations of the starting and stopping characters in the substring that is to be replaced. The `with:` argument is the new substring, which need not be the same size as the original substring.
2. To insert a substring without removing any characters in the existing string, make the ending index less than the starting index.
3. To remove characters, replace them with an empty string.

```
"Print it"
| colorNames magentaStart yellowStart |
colorNames := 'cyan magenta yellow'.
magentaStart := colorNames findString: 'magenta' startingAt: 1.
```

```
"Replace magenta with oddDarkReddishColor."
colorNames := colorNames
  copyReplaceFrom: magentaStart
  to: magentaStart + 'magenta' size - 1
  with: 'oddDarkReddishColor'.
"Basic Step 1"
```

```
"Insert newColor before oddDarkReddishColor."
```

```
colorNames := colorNames
  copyReplaceFrom: magentaStart
  to: magentaStart - 1
  with: 'newColor '.
```

"Basic Step 2"

```
"Remove yellow."
yellowStart := colorNames findString: 'yellow' startingAt: 1.
colorNames := colorNames
  copyReplaceFrom: yellowStart
  to: yellowStart + 'yellow' size - 1
  with: String new.
```

"Basic Step 3"

```
^colorNames
```

Variant

Replacing All Occurrences of a Substring

- Send a `copyReplaceAll:with:` message to the string. The first argument is the substring that is to be replaced. The second argument is the replacement substring.

```
"Print it"
| colorNames |
colorNames := String new.
ColorValue constantNames do: [ :name |
  colorNames := colorNames, name asString, ' '].
```

```
colorNames := colorNames
  copyReplaceAll: 'Gray'
  with: 'Grey'.
```

"Variant Step"

```
^colorNames
```

Abbreviating a String

Strategy

Abbreviations are rarely as comprehensible as the full form of a string, and automatically derived abbreviations tend to be even less readable. In some situations, however, an abbreviation is preferable, as when a field is too short to display the full string. For example, an extra-long filename might well extend beyond the width of the field that is intended to display it. In such a situation, displaying the beginning and ending of the string is often more effective than simply truncating it. The basic step shows how to abbreviate a string, inserting an ellipsis (. . .) in place of the missing characters.

Another common abbreviation technique involves removing all vowels. The variant shows how to remove all vowels except the first letter of the string.

Basic Step

- Send a `contractTo:` message to the string. The argument is the number of characters in the abbreviation, including three for the ellipsis. Half of the abbreviation will be taken from the beginning of the string and the other half from the end.

```
"Print it"
| string contractedString |
string := 'North American Free Trade Agreement'.
```

```
contractedString := string contractTo: 15.                                     "Basic Step"
^contractedString
```

Variant

Removing All Vowels

- Send a `dropFinalVowels` message to the string. An abbreviated string will be returned, in which only the leading vowel (if any) remains.

```
"Print it"  
| string noVowelString |  
string := 'North American Free Trade Agreement'.  
noVowelString := string dropFinalVowels.           "Variant Step"  
^noVowelString
```

Inserting Line-End Characters

Strategy

In Smalltalk methods, certain conventions of indentation and line wrapping make the code more readable. Sometimes a string disrupts the readability of the code because it contains embedded carriage returns. The basic steps show how to keep the entire string on one line without sacrificing the embedded returns.

Basic Steps

1. For each embedded carriage return in the string, substitute a backslash character (\).
2. Send a `withCRs` message to the string to convert the backslashes back to carriage returns.

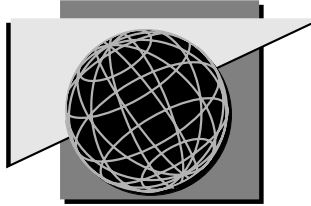
Dialog

request: 'This string\has 3 lines\when displayed.' withCRs

"Basic Steps 1, 2"

initialAnswer: 'No response needed'.

This technique is not recommended for cross-cultural applications, because it interferes with text lookup in message catalogs. Instead, use separate strings and recombine them with literal line-end characters (use the string expansion facility described in the *International User's Guide*).



Chapter 26

Text and Fonts

Creating a Text Object	556
Displaying a Text Object	558
Setting the Line Length	559
Disabling Word Wrapping	560
Controlling Alignment	561
Setting Indents and Tabs	562
Counting the Characters	564
Printing a Text Object	565
Searching for Strings	566
Replacing a Range of Text	567
Comparing Text Objects	568
Copying a Range of Text	569
Changing Case	571
Applying Boldfacing and Other Emphases	572
Using the Platform's Default Font	575
Creating a Custom Text Style	576
Changing Font Size	578
Setting Font Family or Name	582
Setting Text Color	585
Changing the Fonts Menu	587
Changing the Default Font	588
Listing Platform Fonts	589

See Also

- “Characters and Strings” on page 529

Creating a Text Object

Strategy

A `ComposedText` is the displayable counterpart of a `String`. A `ComposedText` consists of a string plus a set of attributes that control the appearance of that string, such as boldness and font. Typically, a composed text is created when you want to customize the appearance of the text that is displayed in a textual widget such as a text editor or a label.

The basic step shows how to convert a string into a composed text. The default display attributes then can be changed separately.

The composed text's display attributes are controlled by an instance of `TextAttributes`. To supply a custom `TextAttributes` while creating a composed text, use the variant.

A `Text` is an intermediate text object between a string and a composed text. It holds a string plus an array of emphasis values that apply to the string. Because the emphasis values can be interpreted only by a composed text, a `Text` is rarely used in applications directly. However, it is frequently manipulated during operations that involve applying boldness or other emphasis values to a composed text.

Basic Step

- Send an `asComposedText` message to a string.

```
| string txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
string := ComposedText comment.  
  
txt := string asComposedText.                                "Basic Step"  
  
txt displayOn: gc at: 5@5.
```

Variant

Creating a Text with Custom Display Attributes

1. Create an instance of `Text`, typically by sending an `asText` message to the string that is the basis for the composed text.
2. Create a `TextAttributes`, or get one from the dictionary that is held by the `TextAttributes` class by sending a `styleNamed:` message to `TextAttributes`.
3. Send a `withText:style:` message to the `ComposedText` class. The first argument is the text. The second argument is the `TextAttributes`.

```
| txt gc textStyle |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := ComposedText comment asText.           "Variant Step 1"
textStyle := TextAttributes styleNamed: #large. "Variant Step 2"

txt := ComposedText                           "Variant Step 3"
    withText: txt
    style: textStyle.

txt displayOn: gc at: 5@5.
```

See Also

- “Creating a String” on page 532

Displaying a Text Object

Strategy

Because a `ComposedText` is a visual component, you can ask it to display itself on a window or other display surface, as shown in the basic steps. A variety of textual widgets are provided in `VisualWorks`, however, so displaying a text directly on a window is usually necessary only when you are creating a new kind of textual widget.

Basic Steps

1. Get the graphics context from the display surface by sending a `graphicsContext` message.
2. Send a `displayOn:` message to the composed text. The argument is the graphics context of the display surface.

```
| txt gc |  
txt := ComposedText comment asComposedText.  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt displayOn: gc at: 5@5.
```

"Basic Step 1"
"Basic Step 2"

See Also

- “Defining What a View Displays” on page 380

Setting the Line Length

Strategy

By default, a composed text wraps long sentences onto multiple lines to avoid running off the right edge of the display area. The line length is determined by the composition width of the composed text, as shown in the basic steps.

Normally the composition width is adjusted automatically when a composed text is installed in a text widget. The example does not use a text widget; it simply displays the text directly on a scratch window. Thus, this technique would be useful mainly when you are creating the displaying method (`displayOn:`) for a new text widget.

Changing the composition width has no effect when word wrapping has been disabled in the text.

Basic Steps

1. Send a `compositionWidth:` message to the composed text. The argument is the line length in pixels.
2. To get the current line length, send a `compositionWidth` message.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := VisualComponent comment asComposedText.
```

```
txt compositionWidth: 380.
```

```
"Basic Step 1"
```

```
txt displayOn: gc at: 5@5.
```

See Also

- “Getting a String’s Length and Width” on page 539

Disabling Word Wrapping

Strategy

By default, a composed text wraps long sentences onto multiple lines to avoid running off the right edge of the display area. This word-wrapping feature can be disabled for columnar material or other text that would be disrupted by wrapping, as shown in the basic steps. If you turn off word wrapping, however, be sure to provide a horizontal scroll bar on the text widget, or fix the size of the widget to ensure that it is wide enough.

Note that VisualWorks text widgets do not consult the text about word wrapping, because frequently a string is the “text” of a widget and a string has no notion of wrappability. So when you turn off word wrapping in a text that is held by a text widget, you must turn off word wrapping in the text widget itself.

Basic Step

- Send a `wordWrap:` message to the composed text. The argument is `false` to disable wrapping and `true` to turn it on.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := ComposedText comment asComposedText.  
txt compositionWidth: 380.
```

```
txt wordWrap: false.                                     "Basic Step"
```

```
txt displayOn: gc at: 5@5.
```

Controlling Alignment

Strategy

By default, a composed text starts each new line flush against the left margin. In some situations, it is more appropriate to align the text flush at the right margin, or centered, or flush with both margins. The basic steps show how to change the alignment of a composed text.

Basic Steps

1. For flush-left text (the default), send a `leftFlush` message to the composed text.
2. For flush-right text, send `rightFlush`.
3. For centered text, send `centered`.
4. For text that aligns with both left and right margins, send `justified`.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := VisualComponent comment asComposedText.  
txt compositionWidth: 380.
```

```
txt rightFlush.
```

"Basic Step 2"

```
txt displayOn: gc at: 5@5.
```

See Also

- “Aligning Text” on page 178

Setting Indents and Tabs

Strategy

With a composed text, you can set two indents and any number of tab stops. All of these settings are measured in pixels.

By default, all lines in a composed text begin at the left edge of the containing view. The first variant shows how to set one indent that affects only the first line and another that affects all subsequent lines of text.

You can set any number of tab stops, as shown in the second variant. The tab settings are controlled by the `TextAttributes` object that is held by the composed text. Notice that you must make a copy of the attributes object (called a *text style*) because the default text style for any composed text is a systemwide object—changing that object affects all texts that do not already have custom attributes, possibly with disruptive effects.

Variants

V1. Setting the First and Subsequent Indents

1. Send a `firstIndent:` message to the composed text. The argument is the width in pixels of the first line's indentation from the left edge.
2. To set the indent for later lines, send a `restIndent:` message to the composed text. The argument is the width of the indentation from the left edge for all lines after the first line.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := 'Line 1\\Line 2\\Line 3\\Line 4'  
      withCRs asComposedText.  
txt compositionWidth: 380.
```

```
txt firstIndent: 50.                                "V1 Step 1"  
txt restIndent: 100.                                "V1 Step 2"
```

```
txt displayOn: gc at: 5@5.
```

V2. Setting Tab Stops

1. Get a copy of the `TextAttributes` from the composed text by sending a `textStyle` message followed by a `copy` message.
2. Send a `useTabs:` message to the text style. The argument is an array containing one or more tab settings. Each setting is an integer indicating how many pixels separate that tab stop from the `restIndent` setting. When each tab is an equal distance from its predecessor, the array can contain a single integer indicating that separation distance.
3. Install the modified text style in the composed text by sending a `textStyle:` message to the text, with the style as the argument.

```
| txt gc style tab |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
tab := String with: Character tab.
txt := ('Line 1\Line 2\Line 3',
       tab, '1 Tab',
       tab, tab, '2 Tabs',
       tab, tab, tab, '3 Tabs').
txt := txt withCRs asComposedText.
txt compositionWidth: 380.

txt firstIndent: 50.
txt restIndent: 100.

style := txt textStyle copy.
style useTabs: #( 15 ).
txt textStyle: style.

txt displayOn: gc at: 5@5.
```

"V2 Step 1"
 "V2 Step 2"
 "V2 Step 3"

Counting the Characters

Strategy

A `ComposedText` holds a `Text`, which in turn holds a `String`. Like a `String`, a `Text` can supply its size, measured in characters. When you need to know how many characters a `ComposedText` contains, the basic steps show how to query the underlying `Text`.

Basic Steps

1. Get the underlying `Text` from the composed text by sending a text message.
2. Send a size message to the `Text`.

"Print it"

| `composedText plainText` |

`composedText := Object comment asComposedText.`

`plainText := composedText text.`

"Basic Step 1"

`^plainText size`

"Basic Step 2"

Printing a Text Object

Strategy

A composed text can be printed on paper very simply, as shown in the basic step. This technique assumes that you have configured your system to send output to a printer. If you can successfully print by using the `hardcopy` command in a System Browser, you can also print a composed text as shown here.

Basic Step

- Send a `hardcopy` message to a composed text.

```
| txt |  
txt := Object comment asComposedText.
```

```
txt hardcopy. "Basic Step"
```

See Also

- “Printing a File” on page 607

Searching for Strings

Strategy

A `ComposedText` has a `Text`, which has a `String`. Strings support a variety of flexible searching techniques. The basic steps show the fullest form of the string searching message.

Basic Steps

1. Get the string from the composed text by sending a string message to the text.
2. Send a `findString:startingAt:ignoreCase:useWildcards: message` to the string. The `findString` argument is the substring to be found. The `startingAt` argument is the index position at which the search is to begin. The `ignoreCase` argument is true when case difference is to be disregarded. The `useWildcards` argument is true when the pound sign (#) and asterisk (*) are to be treated as wildcard characters, with the pound sign taking the place of any single character and the asterisk taking the place of zero or more characters.

```
"Print it"  
| composedText string |  
composedText := Object comment asComposedText.
```

```
string := composedText string.                                     "Basic Step 1"
```

```
^string                                                             "Basic Step 2"  
  findString: 'Var*:'  
  startingAt: 1  
  ignoreCase: true  
  useWildcards: true.
```

See Also

- “Searching” on page 543

Replacing a Range of Text

Strategy

Replacing part of a `ComposedText` is done very much as with a string. Either a string or a text can be substituted for part of an existing text. If the replacement text has boldfacing or other emphasis values, they will be preserved.

Basic Step

- Send a `replaceFrom:to:with:` message to the composed text. The first and second arguments are integers indicating the range of text to be replaced. The third argument is the replacement text, which can be either a string or a text.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := 'Red Green Blue' asComposedText.  
txt compositionWidth: 300.
```

```
txt replaceFrom: 1                                "Basic Step"  
    to: 3  
    with: 'BloodRed' asText allBold.
```

```
txt displayOn: gc at: 5@5.
```

See Also

- “Removing or Replacing a Substring” on page 549

Comparing Text Objects

Strategy

A `ComposedText` can only tell whether it is the same object as another text. In technical terms, the `=` comparison has the same effect as an `==` comparison. In many situations, it is more useful to test the underlying `Text` objects, which compare their underlying strings.

Basic Steps

1. To test whether two variables reference the same `ComposedText` object, send an `=` message to one variable, with the second variable as the argument.
2. To test whether two different instances of `ComposedText` have equal `Text` objects and hence equal strings, get the text from each composed text and compare using an `=` message.

```
"Print it"  
| txt1 txt2 equal equivalent |  
txt1 := Object comment asComposedText.  
txt2 := Object comment asComposedText.
```

```
equal := txt1 = txt2. "Basic Step"  
equivalent := txt1 text = txt2 text. "Basic Step"
```

```
^Array with: equal with: equivalent
```

See Also

- “Comparing” on page 540

Copying a Range of Text

Strategy

A `ComposedText` does not directly support copying a range of it. The basic steps demonstrate a technique that is based on copying a range of its underlying `Text` and then creating a new composed text with that range. The text style is also transferred to the new composed text.

The composition width and word-wrap setting are not copied in this approach. That is rarely necessary because the width is often set by a text view, and word wrap typically remains in the default “on” condition. The variant shows how to preserve those settings, just in case.

Basic Steps

1. Get the underlying `Text` by sending a text message to the composed text.
2. Copy the desired range of text by sending a `copyFrom:to:` message. The first argument is the beginning index and the second argument is the ending index.
3. Convert the text to a composed text by sending an `asComposedText` message.

```
| composedText plainText descriptionEnd copy gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
composedText := Object comment asComposedText.
composedText compositionWidth: 300.
```

```
plainText := composedText text.                                "Basic Step 1"
descriptionEnd := plainText
    findString: 'Class Variables'
    startingAt: 1.
descriptionEnd := descriptionEnd - 1.
```

```
copy := plainText copyFrom: 1 to: descriptionEnd.              "Basic Step 2"
copy asComposedText displayOn: gc at: 5@15.                    "Basic Step 3"
```

Variant

Copying the Width and Word Wrap, Too

After doing the basic steps above:

1. Send a `compositionWidth:` message to the copy. The argument is the composition width of the original composed text, which can be obtained by sending a `width` message to it.
2. Send a `wordWrap:` message to the copy. The argument is the word-wrap setting of the original composed text, which can be accessed using a `wordWrap` message.

```
| composedText plainText descriptionEnd copy gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
composedText := Object comment asComposedText.  
composedText compositionWidth: 300.
```

```
plainText := composedText text.  
descriptionEnd := plainText  
    findString: 'Class Variables'  
    startingAt: 1.  
descriptionEnd := descriptionEnd - 1.
```

```
copy := plainText copyFrom: 1 to: descriptionEnd.  
copy := copy asComposedText.
```

```
copy compositionWidth: composedText width.  
copy wordWrap: composedText wordWrap.  
copy displayOn: gc at: 5@15.
```

"Variant Step 1"

"Variant Step 2"

Changing Case

Strategy

The underlying `Text` that is held by a composed text can be converted to uppercase or lowercase. The basic steps show how to extract the underlying text, change its case, and insert the modified text back into the composed text.

In the example, the “HELLO, WORLD” text retains the composition width of the shorter “Hello, World” text. As a result, the capitalized version is displayed on two lines because it no longer fits on a single line. You can adjust the composition width to compensate for the increased size of the text, but normally this is handled automatically by the text widget that is responsible for displaying the text.

Basic Steps

1. Get the underlying `Text` from the composed text by sending a text message, and then change the case by sending either an `asUppercase` or `asLowercase` message.
2. Install the changed text by sending a `text:` message to the composed text. The argument is the changed text.

```
| composedText capText gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
composedText := 'Hello, World' asComposedText.
```

```
capText := composedText text asUppercase.
composedText text: capText.
```

"Basic Step 1"

"Basic Step 2"

```
composedText displayOn: gc at: 5@5.
```

Applying Boldfacing and Other Emphases

Strategy

A `Text`, the underlying text in a composed text, has two parts: a `String` and an array of modifiers that indicate how each character in the string is to be displayed. Each modifier is called an *emphasis* because the commonly used modifiers such as `#bold` and `#italic` are often used to emphasize a portion of a text.

Emphasis values are implemented by a `TextAttributes`, also known as a *text style*, which is held by a composed text. When a `Text` is displayed directly, rather than with a containing `ComposedText`, the system provides a default `TextAttributes`. The same default is used by a composed text unless you supply an alternate. This default set of attributes defines several standard emphases, as shown in the basic step.

When two or more emphases apply to the same range of characters, such as bold and italic, an array containing the emphases can be used, as shown in the first variant.

When an entire text is to be given the same emphasis, you can apply it without having to specify the range explicitly, as shown in the second variant.

Because boldfacing an entire text is a common operation, a convenient means of applying the `#bold` emphasis to a text is available, as shown in the third variant.

Basic Step

- Send an `emphasizeFrom:to:with:` message to a `Text`. The first and second arguments identify the character range to be modified. The third argument is the emphasis value. Standard emphases are `#bold`, `#italic`, `#serif`, `#underline`, `#strikeout`, `#large`, and `#small`.

```
| txt gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := 'normal bold italic serif underline strikeout large small' asText.
```

```
txt emphasizeFrom: 8 to: 11 with: #bold.
```

"Basic Step"

```
txt emphasizeFrom: 13 to: 18 with: #italic.  
txt emphasizeFrom: 20 to: 24 with: #serif.  
txt emphasizeFrom: 26 to: 34 with: #underline.  
txt emphasizeFrom: 36 to: 44 with: #strikeout.  
txt emphasizeFrom: 46 to: 50 with: #large.  
txt emphasizeFrom: 52 to: 56 with: #small.
```

```
txt displayOn: gc at: 5@25.
```

Variants

V1. Applying Multiple Emphases to the Same Characters

- Send an `emphasizeFrom:to:with:` message to a `Text`. The first and second arguments identify the character range. The third argument is an array containing the emphasis values.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := 'normal bold&italic large&bold&italic&underline' asText.
```

```
txt emphasizeFrom: 8 to: 18 with: #( #bold #italic). "V1 Step"  
txt emphasizeFrom: 20 to: txt size with: #( #large #bold #italic #underline).
```

```
txt displayOn: gc at: 5@25.
```

V2. Applying Emphasis to an Entire Text

- Send an `emphasizeAllWith:` message to a `Text`. The argument is the emphasis value or an array containing multiple emphasis values.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := 'Hello, World' asText.
```

```
txt emphasizeAllWith: #( #bold #italic). "V2 Step"
```

```
txt displayOn: gc at: 5@25.
```

V3. Boldfacing an Entire Text

- ▶ Send an allBold message to a Text.

```
| txt gc |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := 'Hello, World' asText.
```

```
txt allBold displayOn: gc at: 5@25.
```

"V3 Step"

Using the Platform's Default Font

Strategy

Among the built-in text styles that are available is a virtual text style, which corresponds to the default font that is supplied by the window manager, when applicable. When the Look Selection is set to something other than the host window manager, a font is selected that mimics the appearance of the default font for that look. In the fonts menu, this is the System font. Thus, a widget that uses the System font has the best chance of looking like other applications on any platform on which it is deployed.

The System text style can be applied to any composed text, as shown in the basic steps.

Basic Steps

1. Get the text style nearest the platform default by sending a `styleNamed:` message to the `TextAttributes` class, with the argument `#systemDefault`.
2. Send a `textStyle:` message to the composed text. The argument is the text style from step 1.

```
| txt gc style |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
txt := 'Hello, World' asComposedText.
```

```
style := TextAttributes styleNamed: #systemDefault.           "Basic Step 1"  
txt textStyle: style.                                         "Basic Step 2"
```

```
txt displayOn: gc at: 5@25.
```

Creating a Custom Text Style

Strategy

A composed text uses an instance of `TextAttributes` to control various properties of its text: line spacing, alignment, indents, tabs, and font properties. The `TextAttributes` in turn holds an instance of `CharacterAttributes`, which defines the emphases that are available to the text. Associated with each emphasis symbol is a block that operates on a `FontDescription`. The font description is also held by the `CharacterAttributes` and controls font selection by specifying the font size, family, boldness, and so on. Thus, creating a custom text style, although not a simple task, gives you great flexibility in controlling font selection. The basic steps show how to assemble a custom text style that is equipped to provide a large (24-pixel) font.

A limitation to bear in mind is that a composed text applies the same line spacing to its entire text, so mixing font sizes is effective within only a narrow range for each composed text. Separate instances of `ComposedText` are recommended in such situations.

Basic Steps

1. Create a new instance of `CharacterAttributes` by sending a `newWithDefaultAttributes` message to the `CharacterAttributes` class. This message initializes the `CharacterAttributes` with the standard emphases such as `#bold` and `#italic`, so you don't have to redefine them.
2. Install an instance of `FontDescription` in the new `CharacterAttributes` by sending a `setDefaultQuery:` message. The argument can be either a new instance of `FontDescription` or, as in the example, a copy of the default font description from an existing text style's character attributes. The advantage of copying an existing font description is that you retain the existing settings.
3. Customize the `CharacterAttributes` as desired. The example defines a new emphasis called `#title`, which specifies that the font must be 24 pixels in height.

4. Create a new `TextAttributes` by sending a `characterAttributes:` message to the `TextAttributes` class. The argument is the `CharacterAttributes` that you customized in step 3.
5. If you intend to display unusually large or small text, as in the example, adjust the line spacing and baseline of the text style. The line spacing is set by sending a `lineGrid:` message to the text style, with an argument at least a few pixels larger than the largest font size. To set the baseline, which is the distance between the top of the line and the imaginary line on which capital letters rest, send a `baseline:` message to the text style; the argument is the distance in pixels.
6. Install the custom text style by sending a `textStyle:` message to the composed text. The argument is the custom `TextAttributes` from step 5.
7. Apply the new emphasis to the desired portions of the composed text's underlying `Text`.

```
| txt gc ca ta |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := Object comment asComposedText.
txt compositionWidth: 300.
```

"Create and install a custom text style."

ca := CharacterAttributes newWithDefaultAttributes.	"Basic Step 1"
ca setDefaultQuery: txt textStyle defaultFont.	"Basic Step 2"
ca at: #title put: [:fontDesc fontDesc pixelSize: 24].	"Basic Step 3"
ta := TextAttributes characterAttributes: ca.	"Basic Step 4"
ta lineGrid: 27; baseline: 18.	"Basic Step 5"
txt textStyle: ta.	"Basic Step 6"

txt text emphasizeAllWith: #title.	"Basic Step 7"
txt displayOn: gc at: 5@25.	

See Also

- "Changing Font Size" on page 578
- "Setting Font Family or Name" on page 582

Changing Font Size

Strategy

Two of the standard text emphases, `#small` and `#large`, give you control over the font size within a narrow range, as shown in the basic steps.

Because fonts are supplied by the operating system, and VisualWorks runs on several different operating systems, fonts are specified flexibly by describing the desired properties. This font description is held by a `CharacterAttributes`, which in turn is held by a composed text's text style. Font size is just one of the properties you can set by modifying the font description.

The first variant shows how to define a `#title` emphasis, which modifies the pixel size in the font description for any parts of the text that have the `#title` emphasis.

When mixing font sizes in the same composed text, bear in mind that a single text can have only one setting for line spacing. The second variant shows how to adjust the line spacing and the baseline to suit the largest font you are using. When this produces unsatisfactory results for smaller text, put the smaller text in its own `ComposedText`, with appropriate line spacing.

The built-in text styles (`#large` and `#small`, for example) automatically adjust their pixel sizes to suit the pixel density of the display device. This resizing feature is especially useful when deploying your application on different types of hardware. To incorporate it into your custom text style, use `VariableSizeTextAttributes` instead of its parent class, `TextAttributes`, in the following examples.

Basic Steps

1. Send an `emphasizeFrom:to:with:` message to the composed text's underlying `Text`. The first and second arguments define the character range by specifying the starting and stopping indexes. The third argument is `#small` or `#large`, depending on whether you want the font size to be slightly smaller or slightly larger than normal. The actual size depends on the

fonts available from the operating system, and on some platforms it may not differ at all.

2. To return to the default size, apply a nil emphasis to the text.

```
| txt gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := 'large small' asText.
```

```
txt emphasizeFrom: 1 to: 5 with: #large.           "Basic Step 1"
txt emphasizeFrom: 7 to: 11 with: #small.
txt displayOn: gc at: 5@25.
```

```
txt emphasizeAllWith: nil.                         "Basic Step 2"
txt displayOn: gc at: 5@40.
```

Variants

V1. Defining an Emphasis for a Custom Size

1. Create a new instance of CharacterAttributes by sending a newWithDefaultAttributes message to the CharacterAttributes class.
2. Install an instance of FontDescription in the new CharacterAttributes by sending a setDefaultQuery: message. The argument can be either a new instance of FontDescription or, as in the example, a copy of the default font description from an existing text style. The advantage of copying an existing font description is that you retain the existing settings.
3. Define a new emphasis by sending an at:put: message to the character attributes. The first argument is the name of the emphasis (#title). The second argument is a block that sends a pixelSize: message to the block argument, with the desired size of the font (in pixels, not in points).
4. Create a new TextAttributes by sending a characterAttributes: message to the TextAttributes class. The argument is the CharacterAttributes that you customized in step 3.
5. Install the custom text style in the composed text by sending a textStyle: message to the composed text. The argument is the custom TextAttributes from step 4.

6. Apply the new emphasis to the desired portions of the composed text's underlying Text.

```
| txt gc ca ta |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := Object comment asComposedText.
txt compositionWidth: 300.

"Create and install a custom text style."
ca := CharacterAttributes newWithDefaultAttributes.           "V1 Step 1"
ca setDefaultQuery: txt textStyle defaultFont.              "V1 Step 2"
ca at: #title put: [ :fontDesc | fontDesc pixelSize: 24].  "V1 Step 3"
ta := TextAttributes characterAttributes: ca.                 "V1 Step 4"
txt textStyle: ta.                                           "V1 Step 5"

txt text emphasizeFrom: 1 to: 6 with: #title.                 "V1 Step 6"
txt displayOn: gc at: 5@25.
```

V2. Adjusting the Line Spacing and Baseline

- Send a `gridForFont:withLead:` message to the `TextAttributes` that is held by the composed text. The first argument is the name of the text emphasis (`#title`). The second argument is the *leading*, which is the vertical space to be left between one line and the next—typically zero to two pixels. This adjusts both the line spacing and the baseline to suit the font's size.

```
| txt gc ca ta |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := Object comment asComposedText.

"Create and install a custom text style."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: txt textStyle defaultFont.
ca at: #title put: [ :fontDesc | fontDesc pixelSize: 24].
ta := TextAttributes characterAttributes: ca.
ta gridForFont: #title                                       "V2 Step"
    withLead: 2.
txt textStyle: ta.

txt text emphasizeAllWith: #title.
```

```
txt compositionWidth: 300.  
txt displayOn: gc at: 5@25.
```

See Also

- “Creating a Custom Text Style” on page 576

Setting Font Family or Name

Strategy

The default font is Helvetica, Arial, or a similar font, depending on the operating system. Two of the built-in text emphases give you some control over the choice of font family: `#serif` (for a serif font such as Times) and `#fixedWidth` (for a font whose letters all occupy the same horizontal space, such as Courier). The basic steps show how to apply one of these emphases.

When you want to be more specific about the font family, you can create a custom emphasis to do so. That emphasis can then be applied to all or part of a text, as shown in the variant.

The most specific technique is to provide the name string that the operating system uses to identify a particular font. This approach is useful when, for example, you want to examine the operating system's fonts.

Because some operating systems may not supply the font family or name that you specify, it's a good idea to specify alternatives. You can also specify a wildcard pattern for any of the three attributes, such as `helv*` to indicate that a partial match is acceptable. You can also use the `#serif` and `#fixedWidth` emphases to guide the selection of an alternative—the family attribute supersedes those settings, and the name attribute supersedes the family.

Basic Steps

1. Create a new `FontDescription` and send a `family:` message to it. The argument is an array containing one or more strings. Each string names a font family or a wildcard pattern for partial matching. A string containing an asterisk is frequently used as the final element in the array to indicate that any alternate is preferable to a “font not found” error.
2. Create a `CharacterAttributes` by sending a `newWithDefaultAttributes` message to the class.
3. Install the custom font description by sending a `setDefaultQuery:` message to the character attributes. The argument is the font description from step 2.

4. Create a text style by sending a `characterAttributes: message` to the `TextAttributes` class. The argument is the character attributes from step 3.
5. Adjust the line spacing to suit the font by sending a `gridForFont:withLead: message` to the text style. The first argument is `nil` in this case. The second argument is the amount of leading (space) between lines of text (typically zero to two pixels).
6. Install the text style in the composed text by sending a `textStyle: message` to the composed text. The argument is the text style from step 5.

```
| txt gc ca ta fd |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := Object comment asComposedText.

"Create and install a custom text style."
fd := FontDescription new
    family: #( 'bookman' 'times' '*' );           "Basic Step 1"
    serif: true;
    fixedWidth: false;
    pixelSize: 14.
ca := CharacterAttributes newWithDefaultAttributes. "Basic Step 2"
ca setDefaultQuery: fd.                         "Basic Step 3"
ta := TextAttributes characterAttributes: ca.    "Basic Step 4"
ta gridForFont: nil                             "Basic Step 5"
    withLead: 2.
txt textStyle: ta.                               "Basic Step 6"

txt compositionWidth: 300.
txt displayOn: gc at: 5@25.
```

Variant

Setting the Font Name

- Create a new `FontDescription` and send a `name: message` to it. The argument is a string that names a font family or a wildcard pattern for partial matching. (The example takes the list of available fonts from the operating system and uses the first one.)

```
| txt gc ca ta fd |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := Object comment asComposedText.

"Create and install a custom text style."
fd := FontDescription new
    name: (Screen default listFontNames at: 1).           "Variant Step"
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: fd.
ta := TextAttributes characterAttributes: ca.
ta gridForFont: nil
    withLead: 2.
txt textStyle: ta.

txt compositionWidth: 300.
txt displayOn: gc at: 5@25.
```

See Also

- “Creating a Custom Text Style” on page 576

Setting Text Color

Strategy

The default text style that supports standard emphasis values for text objects also supports color and patterns. Unlike emphases such as `#bold` and `#italic`, a color emphasis requires an argument (the specific color desired). The `#color` emphasis is paired with its argument by making an association out of them, as shown in the basic step.

Patterned text is assembled in the same way by providing a `Pattern` as the color argument, as shown in the variant.

Basic Step

- Send an `emphasizeFrom:to:with:` message to the underlying `Text` of a composed text. The first and second arguments identify the range of characters to be affected. The third argument is an association, which is created by sending a `->` message to the lookup key (`#color`), with the desired color as the argument.

```
| txt gc boldBlue |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := 'BLACK RED GRAY BOLDBLUE' asText.
```

```
txt emphasizeFrom: 7 to: 9 with: #color -> ColorValue red.           "Basic Step"
txt emphasizeFrom: 11 to: 14 with: #color -> ColorValue gray.
```

```
boldBlue := Array with: #bold with: #color -> ColorValue blue.
txt emphasizeFrom: 16 to: 23 with: boldBlue.
```

```
txt displayOn: gc at: 5@25.
```

Variant

Patterned Text

1. Create the `Pattern`. One way to do so, as shown in the example, is to create a `Pixmap`, display the graphic elements

that make up a tile in the pattern, and then convert the Pixmap to a Pattern by sending an `asPattern` message to it.

2. Send an `emphasizeAllWith:` message to the underlying Text of the composed text, or a more selective `emphasizeFrom:to:with:` message. The `with` argument is an association between the lookup key (`#color`) and the pattern. (In the example, a large font is used, so the pattern will be more visible.)

```
| txt gc fd ca ta dotTile dotPattern |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
txt := Object comment asComposedText.

"Create a 48-pixel font description."
fd := FontDescription new
    pixelSize: 48.
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: fd.
ta := TextAttributes characterAttributes: ca.
ta gridForFont: fd withLead: 2.
txt textStyle: ta.

"Create a dotted pattern."
dotTile := Pixmap extent: 6@6.
dotTile graphicsContext
    displayDotOfDiameter: 5 at: 3@3.
dotPattern := dotTile asPattern.

txt text emphasizeAllWith: #color->dotPattern.

txt compositionWidth: 300.
txt displayOn: gc at: 5@25.
```

See Also

- “Creating a Color” on page 686

Changing the Fonts Menu

Strategy

Each of the textual widgets, such as label and field, provides a menu of fonts in its property sheet. You can expand the menu to include a custom font, as shown in the basic steps. The technique involves adding a new `TextAttributes` to the system's dictionary of text styles.

Removing a text style from the system's dictionary can be troublesome when existing widgets specify that font. For that reason, no supported mechanism for removing a font exists. The best you can do is to replace the text style that is associated with a particular name, in the same way that you added the original text style. For this reason, we recommend that you expand the fonts menu with caution.

Basic Steps

1. Create the desired text style. In the example, a 24-pixel font is created.
2. Install the text style in the system's dictionary of styles by sending a `styleNamed:put:` message to the `TextAttributes` class. The first argument is a lookup name, specified as a `Symbol`—a capitalized version of the name will appear in the fonts menu. The second argument is the custom text style.

```
| fd ca ta |
fd := FontDescription new
    pixelSize: 24.
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: fd.
ta := TextAttributes characterAttributes: ca.                "Basic Step 1"
ta gridForFont: fd withLead: 2.

TextAttributes styleNamed: #title put: ta.                "Basic Step 2"
```

Changing the Default Font

Strategy

The default font that is used by VisualWorks tools to display textual information can be changed as shown in the basic steps. Widgets in which the Default font has been selected, both in system tools and in your applications, will also be affected. Because many of the widgets use the System font by default, they will not be affected unless you change their font property to Default.

Basic Steps

1. Send a `setDefaultTo:` message to the `TextAttributes` class. The argument is the `Symbol` that names the desired text style. The text style must have been defined and installed in the fonts menu previously.
2. Refresh the windows that are already open by sending a `resetViews` message to the `TextAttributes` class. When they are redisplayed, they will use the new default.

`TextAttributes setDefaultTo: #default.`

"Basic Step 1"

`TextAttributes resetViews.`

"Basic Step 2"

Listing Platform Fonts

Strategy

In VisualWorks, fonts are usually described in general terms that allow the system to choose a matching font from those provided by the operating system. This approach enables you to move an application to a different operating system, possibly with a different set of fonts, without having to adjust fonts manually. When you are developing an application for a single platform, however, specifying a platform-specific font directly gives you the greatest control over font selection. The basic steps show how to obtain a list of the platform's font names in the form of encoded strings. The font name can be used to set the font for a text style.

Basic Steps

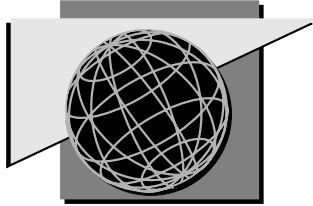
Online example: Font2Example

1. Get the default Screen by sending a default message to that class.
2. Get the list of platform font names by sending a listPlatformFonts message to the default screen.

initialize

```
platformFonts := SelectionInList
    with: Screen default listFontNames.           "Basic Steps 1, 2"

platformFonts selectionIndexHolder onChangeSend: #changedFont to: self.
```



Chapter 27

Text Files

Creating a File or Directory	592
Getting Information about a File	594
Getting File or Directory Contents	597
Storing Text in a File	598
Opening an Editor on a File	601
Deleting a File or Directory	602
Copying or Moving a File	603
Comparing Two Files or Directories	605
Printing a File	607
Scanning Fields in a File (Stream)	609
Setting File Permissions	611

See Also

- “Requesting a Filename” on page 286
- “Characters and Strings” on page 529
- “Object Files (BOSS)” on page 613

Creating a File or Directory

Strategy

The File List and File Editor provide convenient ways of creating files and directories interactively. This chapter describes the programmatic means of doing so.

The `Filename` class supports operations involving disk files and directories. Although `Filename` is an abstract class—the real work is done by its platform-specific subclasses—it directs the creation message to the appropriate subclass. This enables you to keep your file-creating code general enough to run on any of the supported platforms. A `Filename` can represent either a directory or a file (basic step).

When the disk file does not already exist, it is created when the first character is written to it. A directory must be explicitly created (first variant).

The technique shown in the basic step works well for creating a file in the working directory. You can also use that approach with a full pathname that includes directory separators, but the separator character differs across platforms, so you would be compromising the portability of your application. You can preserve portability (second variant) by using a technique that never mentions the separator character explicitly but instead supplies it through the platform-specific subclass of `Filename`.

Basic Step

- Send an `asFilename` message to the string identifying the desired file or directory. The disk file or directory is not affected by the mere creation of a `Filename` object. Because no explicit link exists to the disk file or directory, you need not do anything explicit to release the external resource when you are finished with it.

```
"Inspect"  
| name filename |  
name := 'test.tmp'.
```

```
filename := name asFilename. "Basic Step"  
^filename
```

Variants

V1. Creating a New Disk Directory

- Send a `makeDirectory` message to the `Filename` representing the desired directory. If the disk directory already exists, an error results.

```
"Print it"  
| directory |  
directory := 'test' asFilename.  
directory makeDirectory. "V1 Step"  
^directory exists
```

V2. Constructing a Filename in a Portable Way

- Send a `construct:` message to the `Filename` representing the parent directory. When a pathname is to represent a hierarchy of nested parent directories, use a series of such `construct:` messages.

```
| unixDir portableDir |  
unixDir := 'visual/utills' asFilename.  
  
portableDir := 'visual' asFilename "V2 Step"  
  construct: 'utills'.  
  
unixDir inspect.  
portableDir inspect.
```

Getting Information about a File

Basic Step

Finding Out Whether a File or Directory Exists

- Send an `exists` message to the `Filename`. If the disk file or directory exists, `true` is returned.

```
"Print it"
| unlikelyFile |
unlikelyFile := 'qqqqzzzz' asFilename.
^unlikelyFile exists
```

"Basic Step"

Variants

V1. Counting the Characters in a File

- Send a `fileSize` message to the `Filename`. If the file exists, the number of characters it contains is returned. If the file does not exist, an error results. If the `Filename` represents a disk directory rather than a disk file, zero is returned.

```
"Print it"
| newFile stream |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.

^newFile fileSize.
```

"V1 Step"

V2. Getting the Working Directory

- Send a `defaultDirectory` message to the `Filename` class. A `Filename` representing the working directory is returned.

```
"Inspect"
| workingDir |
```

```
workingDir := Filename defaultDirectory.           "V2 Step"
^workingDir
```

V3. Getting the Parent Directory

- Send a directory message to the Filename. A Filename representing the parent directory is returned.

```
"Print it"
| dir parentDir |
dir := Filename defaultDirectory.
```

```
parentDir := dir directory.           "V3 Step"
^parentDir
```

V4. Getting the Parts of a Pathname

1. To get the entire pathname as a string, send an asString message to the Filename.
2. To get the directory part of a pathname, send a head message to the Filename. A string containing the directory's pathname is returned.
3. To get the file part of the pathname, send a tail message. A string containing the file's name is returned.

```
"Print it"
| filename pathString dirString fileString |
filename := Filename defaultDirectory.
```

```
pathString := filename asString.           "V4 Step 1"
dirString := filename head.               "V4 Step 2"
fileString := filename tail.              "V4 Step 3"
```

```
^
PATH: ', pathString, '
DIRECTORY: ', dirString, '
FILE: ', fileString
```

V5. Distinguishing a File from a Directory

- Send an `isDirectory` message to the `Filename`. If the `Filename` represents a disk directory, `true` is returned. If it represents a disk file, `false` is returned. If neither a file nor a directory with a matching name exists, an error results.

```
"Print it"
| dir |
dir := Filename defaultDirectory.
^dir isDirectory                                     "V5 Step"
```

V6. Getting the Access and Modification Times

1. Get a dictionary containing dates and times associated with a file or directory by sending a `dates` message to the `Filename`.
2. Get the desired date-time pair by sending an `at:` message to the dictionary. The argument is `#accessed` for the time at which the file's contents were most recently accessed. The argument is `#modified` for the time of the most recent modification to the file's contents. The argument is `#statusChanged` for the time of the most recent change in external attributes of the file, such as ownership and permissions.

If the operating system does not support the requested type of information, `nil` is returned; otherwise, an array containing a date and a time is returned.

```
"Print it"
| newFile stream datesDict modifyDates modifyDate modifyTime |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
datesDict := newFile dates.                                     "V6 Step 1"
modifyDates := datesDict at: #modified.                         "V6 Step 2"
modifyDates isNil
  ifFalse: [
    modifyDate := modifyDates first.
    modifyTime := modifyDates last].
^
MODIFIED: ', modifyDate printString, ' at ', modifyTime printString
```

Getting File or Directory Contents

Strategy

The contents of a disk file can be accessed in the form of a string, as shown in the first variant. The second variant shows how to obtain the contents of a directory in the form of an array of strings naming files and subdirectories.

Variants

V1. Getting the Contents of a File

- Send a `contentsOfEntireFile` message to a `Filename` representing a disk file. A string is returned.

```
"Inspect"  
| newFile stream contents |  
newFile := 'testFile' asFilename.  
stream := newFile writeStream.  
stream nextPutAll: Object comment.  
stream close.
```

```
contents := newFile contentsOfEntireFile.           "V1 Step"  
^contents
```

V2. Getting the Contents of a Directory

- Send a `directoryContents` message to a `Filename` representing a disk directory. An array of file and subdirectory names is returned.

```
"Inspect"  
| workingDir contents |  
workingDir := Filename defaultDirectory.
```

```
contents := workingDir directoryContents.         "V2 Step"  
^contents
```

Storing Text in a File

Strategy

Putting a string into a disk file involves using a stream to funnel the characters to the file, as shown in the basic steps.

The stream that is used in the basic steps causes any existing contents in the file to be erased. The first variant shows how to append a string to the existing contents, if any.

A stream holds onto an external resource, which must be released. The second variant shows a technique for assuring that the stream is closed gracefully under any conditions short of a system failure.

When your intention is to create a new disk file, it's a good idea to test the `Filename` to make sure a file with the same name does not already exist. When your application will be deployed on a UNIX system, it's also advisable to make sure the user has the appropriate file permissions, as shown in the second variant.

Basic Steps

1. Create a `Filename` by sending an `asFilename` message to a string containing the pathname.
2. Create a stream for writing characters onto the file by sending a `writeStream` message to the `Filename`.
3. Send the string's characters to the file by sending a `nextPutAll:` message to the stream. The argument is the string. This operation can be repeated for a series of strings.
4. Close the stream by sending a `close` message to it.

```
"Inspect"
| newFile stream |
newFile := 'testFile' asFilename.           "Basic Step 1"
stream := newFile writeStream.             "Basic Step 2"
stream nextPutAll: Object comment.         "Basic Step 3"
stream close.                              "Basic Step 4"
```

```
^newFile contentsOfEntireFile
```

Variants

V1. Appending Text to a File

- When creating the stream, send an `appendStream` message to the `Filename`.

```
"Print it"  
| filename stream |  
filename := 'testFile' asFilename.
```

```
"Creating the file."  
stream := filename writeStream.  
stream nextPutAll: 'FIRST STRING'.  
stream close.
```

```
"Appending"  
stream := filename appendStream.  
stream nextPutAll: ' -- SECOND STRING'.  
stream close.
```

"V1 Step"

```
^filename contentsOfEntireFile
```

V2. Storing Text with Safeguards

1. After creating the `Filename`, test whether a disk file or directory with a matching name already exists by sending an `exists` message to it.
2. If the `Filename` already exists, test whether it is a directory by sending an `isDirectory` message to it.
3. If the `Filename` represents a directory, warn the user and cancel the operation.
4. If the `Filename` represents a file, warn the user that the existing contents of the file will be overwritten (this is not always necessary).
5. Test whether the user has permission to write onto the file (especially when your application will be deployed on UNIX systems) by sending a `canBeWritten` message to the `Filename`.
6. If the user does not have write permission on the file, warn the user and cancel the operation.

7. To assure that the stream is closed and the external resource is released, even if an abnormal interruption occurs, enclose the stream-writing operation in a block and send a `valueNowOrOnUnwindDo:` message to it; the argument is another block containing the stream close expression.

```

"Inspect"
| filename stream response |
filename := 'testFile' asFilename.

filename exists                                "V2 Step 1"
  ifTrue: [filename isDirectory                 "V2 Step 2"
    ifTrue: [
      Dialog warn: 'The file is a directory'.  "V2 Step 3"
      ^self]
    ifFalse: [
      response := Dialog                       "V2 Step 4"
        confirm: 'All right to overwrite the existing file?'
        initialAnswer: false.
      response ifFalse: [^self]].

filename canBeWritten                          "V2 Step 5"
  ifFalse: [
    Dialog warn: 'You do not have the necessary permissions'. "V2 Step 6"
    ^self].

stream := filename writeStream.
[stream nextPutAll: Object comment]
  valueNowOrOnUnwindDo: [stream close].      "V2 Step 7"

^filename contentsOfEntireFile

```

Opening an Editor on a File

Strategy

The main VisualWorks window provides a convenient means of opening a File Editor. The basic step shows how to open an editor programmatically. The editor gives the user of your application the ability to alter the contents of the file. For read-only access to the file, create a canvas containing a read-only text editor or a text editor with a limited menu.

Basic Step

- Send an edit message to the Filename. If the Filename represents a disk directory, an error results. If the Filename represents a nonexistent file, an editor is opened with which the user can create the contents of the file.

```
| newFile stream |  
newFile := 'testFile' asFilename.  
stream := newFile writeStream.  
stream nextPutAll: Object comment.  
stream close.
```

```
newFile edit.
```

```
"Basic Step"
```

Deleting a File or Directory

Strategy

The File List enables you to delete a file interactively. The basic steps show how to do so programmatically (in the example, the target file is first created).

On operating systems such as UNIX that support multiple pathnames for the same physical disk file or directory, deleting as shown here removes the reference that is identified by the pathname, but it does not delete the physical file or directory if another reference exists.

Basic Steps

1. If necessary, confirm that the disk file or directory to be deleted exists by sending an exists message to the Filename.
2. Send a delete message to the Filename.

```
"Print it"
| newFile stream pretest posttest |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
pretest := newFile exists.                                     "Basic Step 1"

newFile delete.                                             "Basic Step 2"
posttest := newFile exists.

^
EXISTS BEFORE DELETION: ', pretest printString, '
EXISTS AFTER DELETION: ', posttest printString.
```

Copying or Moving a File

Strategy

The basic step shows how to make a copy of a disk file, giving the copy a new name.

The first variant shows how to move a disk file, which has the same effect as making a copy and then deleting the original file.

The second variant shows how to rename a file. On operating systems that support this, such as UNIX, renaming a file is more efficient than moving it.

Basic Step

Copying a File

- Send a `copyTo:` message to the `Filename`. The argument is a string containing the pathname of the copy. If the `Filename` represents a directory or a nonexistent disk file, an error results.

```
"Print it"
| newFile stream |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
```

```
newFile copyTo: 'testFile.tmp'.
```

```
"Basic Step"
```

```
^'testFile.tmp' asFilename exists.
```

Variants

V1. Moving a File

- Send a `moveTo:` message to the `Filename`. The argument is a string containing the new pathname, which can include a different directory. If the `Filename` represents a directory or a nonexistent disk file, an error results.

```
"Print it"  
| newFile stream |  
newFile := 'testFile' asFilename.  
stream := newFile writeStream.  
stream nextPutAll: Object comment.  
stream close.
```

```
newFile moveTo: 'testFile.tmp'.
```

"V1 Step"

```
^'testFile.tmp' asFilename exists.
```

V2. Renaming a File

- Send a `renameTo:` message to the `Filename`. The argument is a string containing the new pathname, which can include a different directory. If the `Filename` represents a directory or a nonexistent disk file, an error results.

```
"Print it"  
| newFile stream |  
newFile := 'testFile' asFilename.  
stream := newFile writeStream.  
stream nextPutAll: Object comment.  
stream close.
```

```
newFile renameTo: 'testFile2.tmp'.
```

"V2 Step"

```
^'testFile2.tmp' asFilename exists.
```

Comparing Two Files or Directories

Strategy

When comparing two files or directories, it is important to remember the distinction between a `Filename` and the disk object that it represents. Two `FileNames` are equal when they have the same pathname and not equal when their pathnames differ. When you want to know whether the contents of two disk files or two directories are the same, you must explicitly compare the contents. The basic steps show both types of comparison for files, and the first variant does the same for directories.

Basic Steps

Comparing Two Filenames or Two Files

1. To compare two filenames, send an `=` message to one `Filename`. The argument is the second `Filename`. If they have the same pathname (that is, they point to the same physical disk file), `true` is returned.
2. To compare the contents of two disk files, get the contents of each file by sending `contentsOfEntireFile` messages to the `FileNames`. Then send an `=` message to one of the resulting strings, with the other string as the argument.

```
"Print it"
| file1 file2 stream pathsAreEqual contentsAreEqual |
file1 := 'fileOne' asFilename.
file2 := 'fileTwo' asFilename.
stream := file1 writeStream.
stream nextPutAll: Object comment.
stream close.
file1 copyTo: file2 asString.

pathsAreEqual := (
    file1 = file2).                                "Basic Step 1"
contentsAreEqual := (
    file1 contentsOfEntireFile = file2 contentsOfEntireFile). "Basic Step 2"
```

^^

```
PATHS ARE EQUAL: ', pathsAreEqual printString, '  
CONTENTS ARE EQUAL: ', contentsAreEqual printString.
```

Variant

Comparing Two Filenames or Two Directories

1. To compare two Filenames, send an = message to one Filename. The argument is the second Filename. If they have the same pathname (that is, they point to the same physical disk directory), true is returned.
2. To compare the contents of two disk directories, get the contents of each directory by sending directoryContents messages to the Filenames. Then send an = message to one of the resulting arrays, with the other array as the argument.

```
"Print it"  
| dir1 dir2 pathsAreEqual contentsAreEqual |  
dir1 := Filename defaultDirectory.  
dir2 := dir1 directory.  
  
pathsAreEqual := (  
    dir1 = dir2).                                "Variant Step 1"  
contentsAreEqual := (  
    dir1 directoryContents = dir2 directoryContents). "Variant Step 2"  
  
^  
PATHS ARE EQUAL: ', pathsAreEqual printString, '  
CONTENTS ARE EQUAL: ', contentsAreEqual printString.
```

Printing a File

Strategy

Some operating systems support printing a text file directly, and others require that it first be converted to PostScript or another printer-specific format. The basic steps show a technique for printing a file that works regardless of the operating system. This approach involves converting the file contents to a composed text, so it has the added benefit of providing margins and line wrapping.

The technique of converting the contents of the file to a `ComposedText` takes extra time and memory (especially for large files). The variant shows how to print a text file directly. If you try this on an operating system that does not support this, an error will result.

Basic Steps

1. Get the contents of the file by sending a `contentsOfEntireFile` message to the `Filename`. Convert the resulting string to a `ComposedText` by sending an `asComposedText` message to it.
2. Print the composed text by sending a `hardcopy` message to it.

```
| newFile stream contents composedText |
newFile := 'testFile' asFilename printTextFile.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
```

```
contents := newFile contentsOfEntireFile. "Basic Step 1"
```

```
composedText := contents asComposedText.
composedText hardcopy. "Basic Step 2"
```

Variant

Printing a File Directly

- Send a `printTextFile` message to the `Filename`. If text file printing is not supported by the operating system, an error results.

```
| newFile stream |  
newFile := 'testFile' asFilename printTextFile.  
stream := newFile writeStream.  
stream nextPutAll: Object comment.  
stream close.
```

```
newFile printTextFile
```

"Variant Step"

Scanning Fields in a File (Stream)

Strategy

A stream is a device for finding an element in a collection, scanning a certain number of elements from that position, and so on. In the case of a text file, each element is a character in the file. Thus, by using a special character such as a comma or a colon to separate fields of textual data, you can use a text file as a crude form of database. More to the point, you can use a stream to read the fields in a textual data file that has been created by another application.

The first variant shows how to create and edit a data file that contains comma-delimited fields. The second variant shows how to read such a data file back in.

Variants

V1. Writing Fields to a Data File

1. Create a write stream on the file by sending a `writeStream` message to the `Filename`.
2. Create a block in which, for each field of data, a `nextPutAll:` message is sent to the stream with the data string as argument, followed by a `nextPut:` message with the separator character as argument.
3. Send a `valueNowOrOnUnwindDo:` message to the data-writing block. The argument is another block that closes the stream by sending a `close` message to it.
4. To confirm the operation, open an editor on the data file.

```
| dataFile stream separator writingBlock |
dataFile := 'dataFile' asFilename.
separator := $,,"comma"
```

```
stream := dataFile writeStream.                                     "V1 Step 1"
```

```
writingBlock := [
    ColorValue constantNames do: [ :color |
        stream nextPutAll: color.                                     "V1 Step 2"
        stream nextPut: separator]].
```

writingBlock valueNowOrOnUnwindDo: [stream close]. "V1 Step 3"

dataFile edit. "V1 Step 4"

V2. Reading Fields in a Data File

1. Create a read stream on the file by sending a readStream message to the filename.
 2. Create a block in which the next field of data is fetched by sending an upTo: message to the stream, with the separator character as the argument. This is repeated by placing it within an inner block that is repeated until the end of the stream is encountered.
 3. Send a valueNowOrOnUnwindDo: message to the data-reading block. The argument is another block that closes the stream by sending a close message to it.
-

```
"Inspect"
| dataFile stream separator writingBlock colorNames readingBlock |
dataFile := 'dataFile' asFilename.
separator := $, "comma"
```

```
"Write data"
stream := dataFile writeStream.
writingBlock := [
    ColorValue constantNames do: [ :color |
        stream nextPutAll: color.
        stream nextPut: separator]].
writingBlock valueNowOrOnUnwindDo: [stream close].
```

```
"Read data"
stream := dataFile readStream. "V2 Step 1"
colorNames := OrderedCollection new.
readingBlock := [
    [stream atEnd] whileFalse: [
        colorNames add: (stream upTo: separator)]. "V2 Step 2"
readingBlock valueNowOrOnUnwindDo: [stream close]. "V2 Step 3"
```

```
^colorNames
```

Setting File Permissions

Strategy

On operating systems such as UNIX that support file and directory permissions, the permission to change a file can be added or removed as shown in the basic steps. The most general permission is affected—when possible, the permission change applies to everyone else in addition to the current user. The basic steps also show how to ask a `Filename` whether the associated disk file or directory can be written to, which is a portable operation that can be used on any operating system.

Basic Steps

1. To remove the permission to change the contents of a file or directory, send a `makeUnwritable` message to the `Filename`.
2. To restore the writing permission, send a `makeWritable` message.
3. To find out whether the writing permission is enabled, send a `canBeWritten` message. If the file or directory does not exist, a response of `true` indicates that the parent directory is writable. The `canBeWritten` test works on all operating systems.

```
"Print it"
| newFile stream removed restored |
newFile := 'testFile' asFilename.
stream := newFile writeStream.
stream nextPutAll: Object comment.
stream close.
```

```
newFile makeUnwritable.
removed := newFile canBeWritten.
```

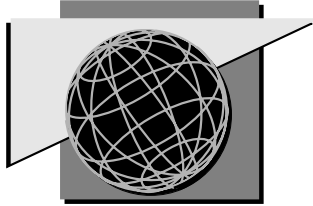
```
"Basic Step 1"
```

```
newFile makeWritable.
restored := newFile canBeWritten.
```

```
"Basic Step 2"
```

```
"Basic Step 3"
```

```
^
PERMISSION REMOVED: ', removed printString, '
PERMISSION RESTORED: ', restored printString.
```



Chapter 28

Object Files (BOSS)

Storing Objects in a BOSS File	614
Getting Objects from a BOSS File	617
Storing and Getting a Class	621
Converting Data After Changing a Class	624
Customizing the Storage Representation	626

See Also

- “Text Files” on page 591

Storing Objects in a BOSS File

Strategy

When you need to store an object's data, connecting to a database is the usual solution. When a database is not available, you can use the Binary Object Streaming Service (BOSS) to store one or more objects in a file. Each object is stored in a compact, encoded format, along with any objects that it holds. The basic steps show how to store a single instance of `PointExample` in a file.

When you need to store a collection of objects, you can either store the collection as a unit or store each of its elements individually, as shown in the first variant. Storing the elements individually enables you to stop reading in the file after the desired element is found, which is useful in applications involving large data files and selective access.

You can also append more objects to the end of an existing BOSS file, as shown in the second variant.

Limitation: Avoid BOSSing out objects that are tied to the windowing system or the execution machinery, such as `Window`, `Context`, and `BlockClosure`. Avoid circular references, such as an application model that holds onto a window that holds onto the application model, and so on. BOSS is intended for data objects, not interface objects.

Basic Steps

1. Create a data stream, typically a write stream on a `Filename`.
2. Create a `BinaryObjectStorage` by sending an `onNew:` message to that class. The argument is the data stream.
3. Store each data object by sending a `nextPut:` message to the `BinaryObjectStorage`. The argument is the data object. This operation is safer when enclosed in a block and with a `valueNowOrOnUnwindDo:` message sent to that block. The argument is another block in which the stream is closed. This protects against leaving the file open when an error or interrupt occurs.

```

| dataObject dataStream bos |
dataObject := PointExample x: 3 y: 4 z: 5.

dataStream := 'points.b' asFilename writeStream.           "Basic Step 1"
bos := BinaryObjectStorage onNew: dataStream.             "Basic Step 2"

[bos nextPut: dataObject]                                  "Basic Step 3"
valueNowOrOnUnwindDo: [bos close].

```

Variants

V1. Storing a Collection of Objects

After doing basic steps 1 and 2:

- **Send a nextPutAll: message to the BinaryObjectStorage. The argument is a collection of objects. Each element in the collection will be stored separately, enabling you to access them separately later.**

```

| dataCollection bos |
dataCollection := ColorValue constantNames.
bos := BinaryObjectStorage
onNew: 'colors.b' asFilename writeStream.

[bos nextPutAll: dataCollection]                            "V1 Step"
valueNowOrOnUnwindDo: [bos close].

```

V2. Appending an Object to a File

1. **Create a read-append data stream, typically by sending a readAppendStream message to a Filename.**
2. **Create a BinaryObjectStorage by sending an onOld: message to that class. The argument is the data stream.**
3. **Set the writing position to the end of the file by sending a setToEnd message to the BinaryObjectStorage.**
4. **For each object to be appended, send a nextPut: message to the BinaryObjectStorage. The argument is the data object.**

```
"Inspect"
| colorNames newColor bos |

"First create a file containing color names."
colorNames := ColorValue constantNames.
bos := BinaryObjectStorage
    onNew: 'colors.b' asFilename writeStream.
[bos nextPutAll: colorNames]
    valueNowOrOnUnwindDo: [bos close].

"Then append a new color name."
newColor := #mudBrown.
bos := BinaryObjectStorage
    onOld: 'colors.b' asFilename readAppendStream.
bos setToEnd.
[bos nextPut: newColor]
    valueNowOrOnUnwindDo: [bos close].
```

Getting Objects from a BOSS File

Strategy

The basic steps show how to get the entire contents of a BOSS file, with each stored object as an element in an array. (In the example, a BOSS file is created first.)

For selective access to the objects in the data stream, you can read them sequentially until you find the desired object, as shown in the first variant.

Another selective approach is to position the stream at the beginning of the desired object, as shown in the second variant. This technique, although swifter than reading each object sequentially, assumes that your application keeps a position index for each object in the file when the objects are stored.

Basic Steps

1. Create a data stream, typically by sending a `readStream` message to a `Filename` that represents the data file.
2. Create a `BinaryObjectStorage` by sending an `onOld:` message to that class, with the data stream as argument. (When you do not intend to write new objects onto the file, send an `onOldNoScan:` message instead; this is faster because it does not scan the data file as it must before writing more data.)
3. Get the objects in the file by sending a `contents` message to the `BinaryObjectStorage`. An array containing the stored objects will be returned.
4. Close the `BinaryObjectStorage` (which also closes the data stream).

```
"Inspect"
| colorNames bos array |
```

```
"First create a file containing color names."
colorNames := ColorValue constantNames.
bos := BinaryObjectStorage
    onNew: 'colors.b' asFilename writeStream.
[bos nextPutAll: colorNames]
valueNowOrOnUnwindDo: [bos close].
```

```

"Read the file contents"
bos := BinaryObjectStorage
  onOldNoScan: 'colors.b' asFilename readStream.
[array := bos contents]
  valueNowOrOnUnwindDo: [bos close].

^array

```

"Basic Step 2"

"Basic Step 3"

"Basic Step 4"

Variants

V1. Searching Sequentially for an Object

1. Create a block in which you test whether the end of the data stream has been reached by sending an `atEnd` message to the `BinaryObjectStorage`.
2. Send a `whileFalse:` message to the block. The argument is another block, in which you get the next object in the data stream by sending a `next` message to the `BinaryObjectStorage`. Test the object to find out whether it is the desired object; if so, send a `setToEnd` message to the `BinaryObjectStorage` to break out of the loop.
3. Close the `BinaryObjectStorage`.

```

"Inspect"
| points bos foundObject nextObject |

```

```

"First create a file containing points."
points := OrderedCollection new.
1 to: 100 do: [:coord |
  points add: (PointExample x: coord y: coord z: coord)].
bos := BinaryObjectStorage
  onNew: 'points.b' asFilename writeStream.
[bos nextPutAll: points]
  valueNowOrOnUnwindDo: [bos close].

```

```

"Search sequentially."
foundObject := nil.
bos := BinaryObjectStorage
  onOldNoScan: 'points.b' asFilename readStream.
[[bos atEnd]

```

"V1 Step 1"


```

whileFalse: [
    nextObject := bos next.
    (nextObject z > 45)
    ifTrue: [
        foundObject := nextObject.
        bos setToEnd]]
valueNowOrOnUnwindDo: [bos close].

```

"V1 Step 2"

"V1 Step 3"

^foundObject

V2. Getting an Object at a Specific Position

1. Create a dictionary to be used as a lookup table. Each entry in the dictionary will associate an object's identifier with that object's position in the BOSS file.
2. Before each object-writing operation, record the binary stream's position in the lookup table.
3. After each object-writing operation, send a `forgetInterval:` message to the binary stream. The argument is an `Interval` beginning with the binary stream's index before the write operation and ending with the next index. This assures that the `BinaryObjectStorage` will not make use of back-references to the object just stored when storing future objects; such back-references thwart random access to stored objects.
4. When reading the desired object, first send a `position:` message to the binary stream. The argument is the object's position, as recorded in the lookup table.
5. To get the object at that position, send a `next` message to the binary stream.

```

"Print it"
| bos foundObject positions prevIndex |
positions := Dictionary new.
bos := BinaryObjectStorage onNew: 'colors.b' asFilename writeStream.
prevIndex := bos nextIndex.

```

"V2 Step 1"

```

"First create a file containing colors."
[ColorValue constantNames do: [ :name |
    positions at: name put: bos position.
    bos nextPut: (ColorValue perform: name).
    bos forgetInterval: (prevIndex to: bos nextIndex).

```

"V2 Step 2"

"V2 Step 3"

```
prevIndex := bos nextIndex]]  
valueNowOrOnUnwindDo: [bos close].
```

```
"Get the object at a certain location."  
bos := BinaryObjectStorage onOld: 'colors.b' asFilename readStream.  
[bos position: (positions at: #chartreuse). "V2 Step 4"  
foundObject := bos next] "V2 Step 5"  
valueNowOrOnUnwindDo: [bos close].
```

```
^foundObject
```

Storing and Getting a Class

Strategy

A `BinaryObjectStorage` is most often used to store instances rather than classes, relying on the virtual image to contain the class definitions. When the virtual image that is to read a BOSS file does not contain the necessary classes, you can use BOSS, parcels, or the conventional file-out/file-in procedure to transfer the necessary class definitions.

Unlike the file-in procedure, the BOSS technique does not normally require the presence of any compilers in the receiving image. Thus, you can use BOSS to introduce a new or redefined class into a deployment image, perhaps as a means of delivering a patch that fixes a bug.

Note, however, that BOSSing in a class requires the Smalltalk compiler to be present when any superclass of that class varies in structure between the receiving image and the original image (the image from which the class was originally BOSSed out). In particular, if any superclass varies between these two images with respect to the number or order of its instance variables, BOSS will attempt to invoke the Smalltalk compiler to recompile the class's methods. When a collection of classes is stored using BOSS, they are automatically sorted into superclass order. BOSS writes the same information that `fileOut` does: the class definition, method definitions, and an expression that initializes the class if a class `initialize` method is present.

By default, BOSS stores the source code for methods, the class comment, and the protocols. The variant shows how to arrange for BOSS to omit the source code, which is useful when you want to discourage users of your application from modifying it. For this reason, even classes that have nothing to do with BOSS data are sometimes transferred from one image to another using BOSS.

Basic Steps

1. To store a collection of classes in a BOSS file, send a `nextPutClasses:` message to a binary stream. The argument is a collection containing the desired classes.

2. To load a collection of classes from a BOSS file, send a next-Classes message to a binary stream on the file. (In the example, loading the Date class has no effect because the image already contains the same definition of that class.)

```
"Print it"
| file bos |
file := 'date.b' asFilename.
bos := BinaryObjectStorage onNew: file writeStream.

"Write the Date class to a file."
[bos nextPutClasses: (Array with: Date)]
    valueNowOrOnUnwindDo: [bos close].
"Basic Step 1"

"Read the file contents"
bos := BinaryObjectStorage onOldNoScan: file readStream.
[bos nextClasses]
    valueNowOrOnUnwindDo: [bos close].
"Basic Step 2"

^file fileSize
```

Variant

Omitting the Source Code

- Before storing the classes, send a sourceMode: message to the binary stream. The argument is #discard. (An argument of #keep causes sources to be stored, which is the default.)

```
"Print it"
| file bos |
file := 'date.b' asFilename.
bos := BinaryObjectStorage onNew: file writeStream.

"Write the Date class to a file."
[bos sourceMode: #discard.
bos nextPutClasses: (Array with: Date)]
    valueNowOrOnUnwindDo: [bos close].
"Variant Step"

"Read the file contents"
bos := BinaryObjectStorage onOldNoScan: file readStream.
```

```
[bos nextClasses]
  valueNowOrOnUnwindDo: [bos close].
```

```
^file fileSize
```

Converting Data After Changing a Class

Strategy

When you store instances of an object in a BOSS file and then add an instance variable or otherwise change the definition of that object's class, BOSS detects the incompatibility when it tries to read the old data file. For example, suppose the `PointExample` class began its life representing a two-dimensional point; later you extend it to represent three-dimensional points by adding a `z` instance variable in addition to the `x` and `y` variables. The basic steps show how to arrange for old files containing two-dimensional instances of `PointExample` to be read without error.

Basic Steps

Online example: `PointExample`

1. In the class whose definition has been changed, create a class method named `binaryRepresentationVersion`. This method is responsible for returning a version identifier, commonly a sequential number or a descriptive string. (The method must be rewritten each time the class definition is changed, assuming BOSS files relying on the prior version of the class definition will need to be read.)
2. Create a class method named `binaryReaderBlockForVersion:format:`. This method must return a block that converts the old object to a new instance. The block takes one argument, an array of the instance variables (for pointer-type objects) or a `ByteString` (for byte-type objects). The block typically assigns the data values from the old instance variables and then sends a `become:` message to the old object; the argument is the new instance. The first method argument (`oldVersion`) identifies the version (`nil`, by default, and later defined by the method you created in the preceding step) and enables you to distinguish between old data and current data. The second method argument (`oldFormat`) is typically ignored except for internal system purposes.

```
binaryRepresentationVersion "Basic Step 1"  
"First version (nil) had x and y coordinates.  
Second version (2) added a z coordinate."  
  
^2
```

```
binaryReaderBlockForVersion: oldVersion format: oldFormat "Basic Step 2"  
| newPoint |  
oldVersion isNil ifTrue: [  
  ^[:oldPoint |  
    newPoint := PointExample new.  
  
    "Each oldPoint obtained from the BOSS file is an Array  
    that contains the state of an old instance of PointExample.  
    The array elements are the values of the old instance's  
    variables, in the order in which the old version of PointExample  
    defined them."  
  
    newPoint x: (oldPoint at: 1).  
    newPoint y: (oldPoint at: 2).  
    newPoint z: 0. "oldPoint has no z"  
  
    oldPoint become: newPoint]].
```

Customizing the Storage Representation

Strategy

By default, BOSS stores the entire contents of an object, including its dependents and the dependents of its variables. Although this default is appropriate for most data objects, it results in a BOSS error when an interface object is a dependent of a data object that is being BOSSed out. This kind of dependency is often encountered in the case of an instance variable that holds onto a collection when the collection is displayed in a list widget. BOSSing a copy of the collection is one way to remove the dependency.

The basic step shows how to control which parts of an object are BOSSed out. This technique is also useful when an instance variable holds an object that points back to the original object, creating a circular reference that causes endless repetition when BOSS attempts to store either object.

In the example, the custom storage representation contains all of the instance variables, but it nevertheless shows the technique for customizing.

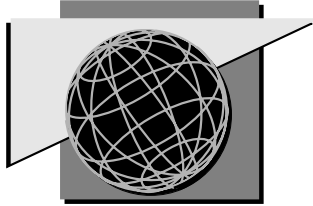
Basic Step

- Create an instance method named `representBinaryOn:` in the class whose BOSS representation you want to customize. The method typically returns a `MessageSend`, which is created by sending a `receiver:selector:arguments: message` to that class. The receiver argument identifies the class that is to create an instance, typically the object's class. The selector argument is the name of the instance-creation method that is to be used when the data is read by BOSS. The arguments argument is a collection of data values, typically the values of the object's instance variables.

```
representBinaryOn: bos                                "Basic Step"  
    "Represent a PointExample by its x, y and z coordinates  
    plus the message and receiver for creating an instance from  
    those coordinates."
```



```
^MessageSend  
  receiver: self class  
  selector: #x:y:z:  
  arguments: (Array with: x with: y with: z).
```



Chapter 29

Geometrics

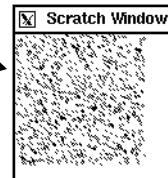
Displaying a Point	630
Displaying a Straight or Jointed Line	631
Displaying a Curved Line	634
Displaying a Polygon	637
Displaying an Arc, Circle, or Ellipse	640
Changing the Line Thickness	644
Changing the Line Cap Style	645
Changing the Line Join Style	647
Coloring a Geometric	649
Integrating a Graphic into an Application	652

See Also

- “Images, Cursors, and Icons” on page 657
- “Color” on page 685

Displaying a Point

Each point is displayed as a line segment to a neighboring point



Strategy

Displaying a single point is rarely done except in batches, as when you are building up a dotted pattern. The basic steps show how to display a dot in the context of a loop that creates a random dot pattern. The technique relies on displaying a line segment from the desired point to a neighboring point.

Basic Steps

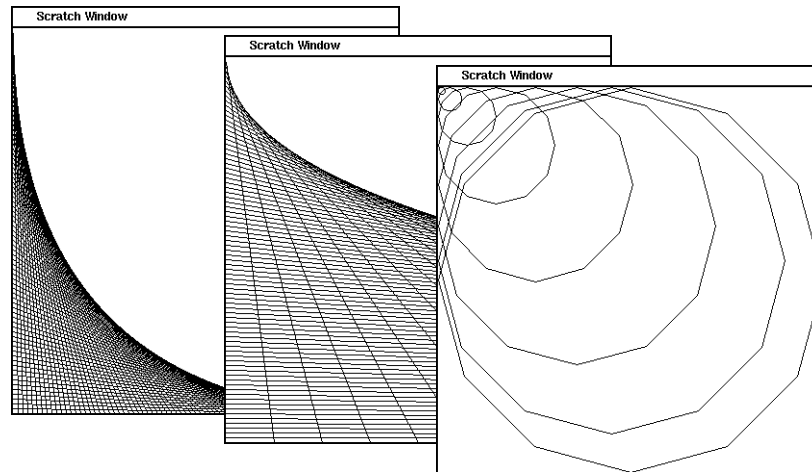
1. Create a Point by sending an @ message to the integer representing the x coordinate. The argument is the y coordinate.
2. Display the point by sending a displayLineFrom:to: message to the graphics context. The first argument is the point and the second argument is a neighboring point, which can be derived by adding 1 to the point.

```
| gc random points |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
random := Random new.
points := OrderedCollection new.
```

```
"Create 1000 random points in a 100-pixel square."
1000 timesRepeat: [
  points add: ((random next * 100) @ (random next * 100)). "Basic Step 1"
```

```
"Display each random point."
points do: [ :pt |
  gc displayLineFrom: pt to: pt + 1] "Basic Step 2"
```

Displaying a Straight or Jointed Line



Strategy

You can draw a straight line directly on a display surface, as shown in the basic steps. Or you can create an instance of `LineSegment` and display it, as shown in the first variant. Creating a `LineSegment` is useful when your application needs to perform an operation on the line, such as determining its length or scaling it.

A jointed line, or polyline, can also be drawn directly or instantiated as a `PolyLine`, as shown in the second variant.

Basic Steps

1. Get the graphics context of the display surface by sending a `graphicsContext` message.
2. Send a `displayLineFrom:to:` message to the graphics context. The first argument is the starting point of the line and the second argument is the endpoint.

```
| gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
```

"Basic Step 1"

```
5 to: 400 by: 5 do: [ :i |
  gc displayLineFrom: 0@i to: i@400].
```

"Basic Step 2"

Variants

V1. Creating and Displaying a Line Segment

1. Create a line segment by sending a `from:to:` message to the `LineSegment` class. The first argument is the starting point of the line and the second argument is the endpoint.
2. Perform any desired operations on the line (in the example, the `x` dimension is exaggerated by a factor of 10).
3. Wrap the line segment in a stroking wrapper by sending an `asStroker` message to it. This equips the line with the ability to render itself.
4. Display the wrapped line segment by sending a `displayOn:` message to its stroking wrapper. The argument is the graphics context of the display surface.

```
| gc line scaleFactor |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
scaleFactor := 10@1.
```

```
5 to: 400 by: 5 do: [ :i |
  line := LineSegment from: 0@i to: i@400.
  line := line scaledBy: scaleFactor.
  line asStroker displayOn: gc].
```

"V1 Step 1"
"V1 Steps 3, 4"

V2. Displaying a Polyline

1. Send a `displayPolyLine:` message to the graphics context. The argument is a collection of points defining the endpoints and vertices of the polyline.
2. Alternatively, create a `Polyline` by sending a `vertices:` message to the `Polyline` class. The argument is the collection of vertices. Then wrap the polyline in a stroking wrapper (using `asStroker`) and display it on the graphics context (using `displayOn:`).

```
| gc points x y radians polyline |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
points := OrderedCollection new.  
0 to: 360 by: 30 do: [ :angle |  
    radians := angle degreesToRadians.  
    x := 200 - (200 * radians cos).  
    y := 200 - (200 * radians sin).  
    points add: x@y].  
gc displayPolyline: points.                                     "V2 Step 1"
```

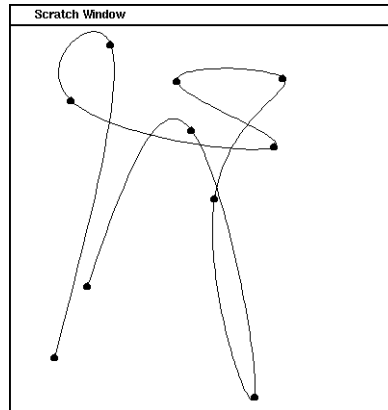


```
polyline := Polyline vertices: points.                         "V2 Step 2"  
0.9 to: 0.1 by: -0.1 do: [ :scale |  
    polyline := polyline scaledBy: scale.  
    polyline asStroker displayOn: gc].
```

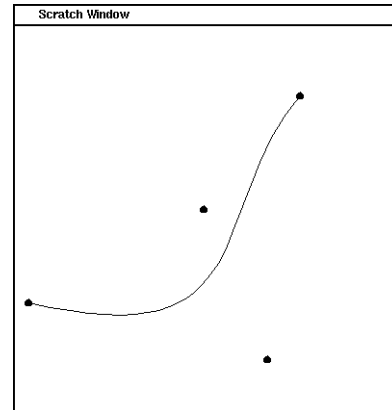
See Also

- “Changing the Line Thickness” on page 644
- “Changing the Line Cap Style” on page 645
- “Changing the Line Join Style” on page 647

Displaying a Curved Line



spline



Bezier curve

Strategy

Frequently a smoothly curved line is preferable to the jointed line provided by a Polyline. A Spline is like a Polyline except that it curves the joints in its collection of points, as shown in the basic steps.

For scientific purposes, a Bezier curve is also available. A Bezier curve has a start, an end, and two control points. Each control point causes the line to curve toward it, as if exerting gravity on the line, as shown in the variant.

Basic Steps

1. Create a Spline by sending a `controlPoints:` message to the Spline class. The argument is a collection of points.
2. Wrap the spline in a stroking wrapper by sending an `asStroker` message to it.
3. Display the wrapped spline by sending a `displayOn:` message to the stroking wrapper. The argument is the graphics context.

```
| gc points spline random x y |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
```



```
points := OrderedCollection new.
random := Random new.
```

```
"Collect 10 random points."
10 timesRepeat: [
  x := random next * 400.
  y := random next * 400.
  points add: x@y.
  gc displayDotOfDiameter: 8 at: points last].
```

```
spline := Spline controlPoints: points.
spline asStroker displayOn: gc.
```

"Basic Step 1"
"Basic Steps 2, 3"

Variant

Displaying a Bezier Curve

1. **Create a Bezier by sending a start:end:controlPoint1:controlPoint2: message to the Bezier class. Each of the arguments is a point.**
2. **Wrap the Bezier curve in a stroking wrapper by sending an asStroker message to it.**
3. **Display the wrapped spline by sending a displayOn: message to the stroking wrapper. The argument is the graphics context.**

```
| gc points bezier random x y |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
points := OrderedCollection new.
random := Random new.
```

```
"Collect 10 random points."
4 timesRepeat: [
  x := random next * 400.
  y := random next * 400.
  points add: x@y.
  gc displayDotOfDiameter: 8 at: points last].
```

```
bezier := Bezier
start: (points at: 1)
end: (points at: 2)
```

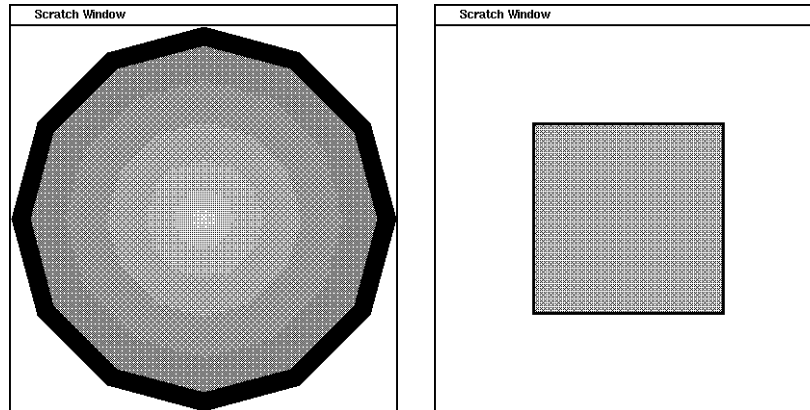
"Variant Step 1"

```
controlPoint1: (points at: 3)
controlPoint2: (points at: 4).
bezier asStroker displayOn: gc. "Variant Steps 2, 3"
```

See Also

- “Changing the Line Thickness” on page 644
- “Changing the Line Cap Style” on page 645

Displaying a Polygon



Strategy

A polygon is a filled Polyline. A polygon can be created and displayed from a collection of vertices, as shown in the basic steps.

A Rectangle is a special case that provides an extended set of operations because it is so commonly used in constructing complex views. A rectangle is commonly created by specifying its origin point and either its lower-right corner or its extent, as shown in the variant.

Basic Steps

1. Send a `displayPolygon:` message to the graphics context of the display surface. The argument is a collection of points, each point representing one vertex of the polygon.
2. Alternatively, create an instance of Polyline by sending a `vertices:` message to the Polyline class, with the vertex points as the argument. Wrap the polyline in a stroking or filling wrapper (using `asStroker` or `asFiller`) and display the wrapped polygon by sending `displayOn:` to the wrapper with the graphics context as argument. A variant of `displayOn:` (used here) enables you to specify the origin—that is, the upper-left corner of the rectangle containing the polygon.

```

| gc points x y radians polyline origin |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
points := OrderedCollection new.
0 to: 360 by: 30 do: [ :angle |
    radians := angle degreesToRadians.
    x := 200 - (200 * radians cos).
    y := 200 - (200 * radians sin).
    points add: x@y].

gc displayPolygon: points.                                     "Basic Step 1"

polyline := Polyline vertices: points.                       "Basic Step 2"
0.9 to: 0.1 by: -0.1 do: [ :scale |
    gc paint: (ColorValue brightness: 1 - scale).
    polyline := polyline scaledBy: scale.
    origin := 200@200 - (polyline bounds width / 2).
    polyline asFiller displayOn: gc at: origin].

```

Variant

Displaying a Rectangle

1. Create a rectangle (in the example, `rect1`) by sending an `extent:` message to the point representing the origin. The argument is a point whose `x` value indicates the width of the rectangle and whose `y` value indicates the height.
2. Alternatively, create a rectangle (`rect2`) by sending a `corner:` message to the origin point. The argument is the lower-right corner point.
3. Wrap the rectangle in a stroking or filling wrapper (using `asStroker` or `asFiller`) and display the resulting wrapper on a graphics context.

```

| gc rect1 rect2 border |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

"Black rectangle"
rect1 := 100@100 extent: 200@200.
rect1 asFiller displayOn: gc.

```

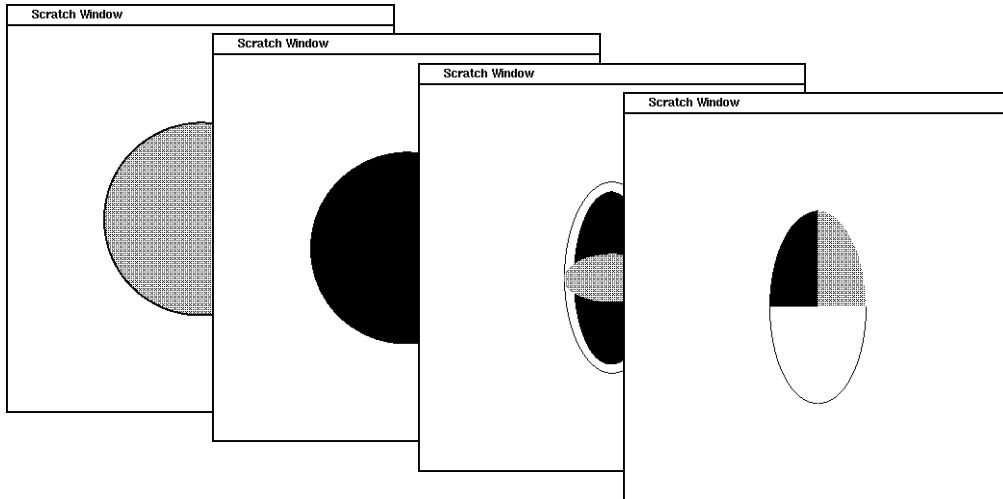
"Variant Step 1"
"Variant Step 3"

```
"Gray rectangle"  
border := 3.  
rect2 := (rect1 origin + border) corner: (rect1 corner - border). "Variant Step 2"  
rect2 asFiller displayOn: (gc paint: ColorValue gray). "Variant Step 3"
```

See Also

- “Changing the Line Thickness” on page 644

Displaying an Arc, Circle, or Ellipse



Strategy

A circle is created by specifying its center point and radius, as shown in the basic steps. The first variant shows an alternative technique that avoids creating an instance of `Circle`, but it is useful only for filled circles (not stroked circles).

An ellipse is created by specifying the rectangle that encloses it, as well as the beginning angle and the number of degrees traversed (the sweep angle) from that starting angle. For a complete ellipse, the angles are 0 and 360, as shown in the second variant. When the bounding rectangle is a square, the ellipse is circular.

An arc is created in the same way as a full ellipse, except that the beginning and sweep angles specify only a portion of the full 360 degrees, as shown in the third variant.

Basic Steps

1. Send a `center:radius:` message to the `Circle` class. The first argument is the center point of the circle. The second argument is an integer indicating the radius of the circle.

2. Wrap the circle in a stroking or filling wrapper by sending `asStroker` or `asFiller` to it.
3. Display the wrapped circle by sending `displayOn:` to it, with the graphics context as argument.

```
| gc circle |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

"Blue filled circle"
circle := Circle center: 200@200 radius: 100.           "Basic Step 1"
circle asFiller displayOn: (gc paint: ColorValue blue). "Basic Steps 2, 3"

"Black stroked circle"
gc paint: ColorValue black; lineWidth: 2.
circle asStroker displayOn: gc.
```

Variants

V1. Displaying a Filled Dot

- Send a `displayDotOfDiameter:at:` message to the graphics context of the display surface. The first argument is the diameter of the circle. The second argument is the center point.

```
| gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

gc displayDotOfDiameter: 200 at: 200@200.           "V1 Step"
```

V2. Displaying an Ellipse

1. For a stroked ellipse, send a `displayArcBoundedBy:startAngle:sweepAngle:` message to the graphics context. The first argument is the rectangle that encloses the ellipse. The second argument is 0 and the third argument is 360.
2. For a filled ellipse, send a `displayWedgeBoundedBy:startAngle::sweepAngle:` message to the graphics context, with the same arguments as above.
3. Alternatively, create an instance of `EllipticalArc` by sending a `boundingBox:startAngle:sweepAngle:` message to that class. The

arguments are the same as above. Then wrap the ellipse in a stroking or filling wrapper and display it on the graphics context.

```
| gc ellipse |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

"Black stroked ellipse"
gc displayArcBoundedBy: (150@100 extent: 100@200)          "V2 Step 1"
  startAngle: 0
  sweepAngle: 360.

"Black filled ellipse"
gc displayWedgeBoundedBy: (160@110 extent: 80@180)         "V2 Step 2"
  startAngle: 0
  sweepAngle: 360.

"Red ellipse"
ellipse := EllipticalArc                                  "V2 Step 3"
  boundingBox: (150@175 extent: 100@50)
  startAngle: 0
  sweepAngle: 360.
ellipse asFiller displayOn: (gc paint: ColorValue red)
```

V3. Displaying an Arc

- Use the same technique as for displaying a full ellipse, but the `startAngle` argument is the angle at which the arc or wedge begins, measured in degrees clockwise from the 3 o'clock position. The `sweepAngle` argument is the number of degrees spanned by the arc, measured clockwise from the starting angle.

```
| gc arc box |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
box := 150@100 extent: 100@200.

"Black stroked arc"
gc displayArcBoundedBy: box                                "V3 Step"
  startAngle: 0
  sweepAngle: 180.
```



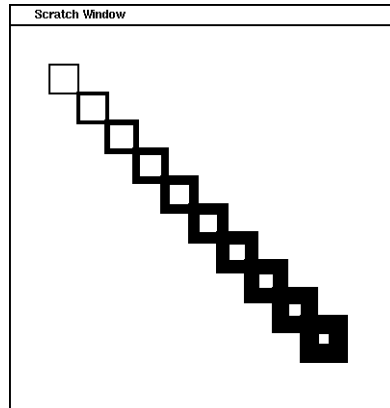
```
"Black filled arc"
gc displayWedgeBoundedBy: box
  startAngle: 180
  sweepAngle: 90. "V3 Step"
```

```
"Red arc"
arc := EllipticalArc
  boundingBox: box
  startAngle: 270
  sweepAngle: 90. "V3 Step"
arc asFiller displayOn: (gc paint: ColorValue red)
```

See Also

- “Changing the Line Thickness” on page 644

Changing the Line Thickness



Strategy

By default, lines, arcs, and polygons are drawn with a one-pixel line. The basic step shows how to increase the line width. Extra thickness is spread evenly on both sides of the actual line, so a horizontal line that is 20 pixels thick has 10 pixels above the line and 10 pixels below.

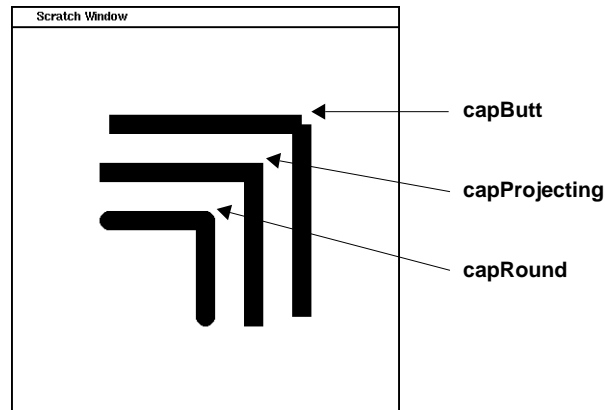
Basic Step

- Send a `lineWidth:` message to the graphics context of the display surface. The argument is an integer indicating the number of pixels of thickness.

```
| gc rect |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
rect := 10@10 extent: 30@30.
```

```
2 to: 20 by: 2 do: [ :width |  
  gc lineWidth: width.                                     "Basic Step"  
  rect moveBy: 30@30.  
  rect asStroker displayOn: gc].
```

Changing the Line Cap Style



Strategy

By default, lines and arcs are drawn with butt ends, which means each end stops abruptly at the specified endpoint. When two thick lines share an endpoint, butt ends produce a notched joint. Changing the cap style to projecting fixes this by extending each end of the line by half of its thickness. Another solution is to use round ends, which extend the ends in a semicircle.

Basic Step

- Send a `capStyle:` message to the graphics context of the display surface. The argument is derived by sending a `capButt`, `capProjecting`, or `capRound` message to the `GraphicsContext` class.

```
| gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
gc lineWidth: 20.
```

```
"Butt line caps -- the default"
gc capStyle: GraphicsContext capButt.
gc displayLineFrom: 100@100 to: 300@100.
gc displayLineFrom: 300@100 to: 300@300.
```

"Basic Step"

"Projecting line caps"

```
gc capStyle: GraphicsContext capProjecting.  
gc displayLineFrom: 100@150 to: 250@150.  
gc displayLineFrom: 250@150 to: 250@300.
```

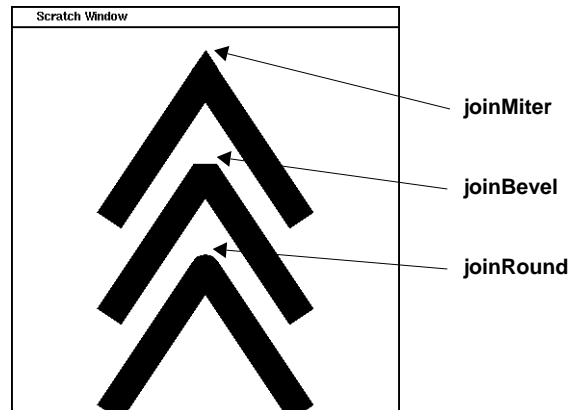
"Basic Step"

"Round line caps"

```
gc capStyle: GraphicsContext capRound.  
gc displayLineFrom: 100@200 to: 200@200.  
gc displayLineFrom: 200@200 to: 200@300.
```

"Basic Step"

Changing the Line Join Style



Strategy

By default, a polyline or polygon is drawn with mitered joints. In some situations, a beveled or rounded joint is preferable. The basic step shows how to change the join style.

Basic Step

- Send a `joinStyle:` message to the graphics context of the display surface. The argument is derived by sending a `joinMiter`, `joinBevel`, or `joinRound` message to the `GraphicsContext` class.

```
| gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
gc lineWidth: 30.
```

"Miter joins -- the default"

```
gc joinStyle: GraphicsContext joinMiter. "Basic Step"
gc displayPolyline: (Array with: 100@200 with: 200@50 with: 300@200).
```

"Bevel joins"

```
gc joinStyle: GraphicsContext joinBevel. "Basic Step"
gc displayPolyline: (Array with: 100@300 with: 200@150 with: 300@300).
```

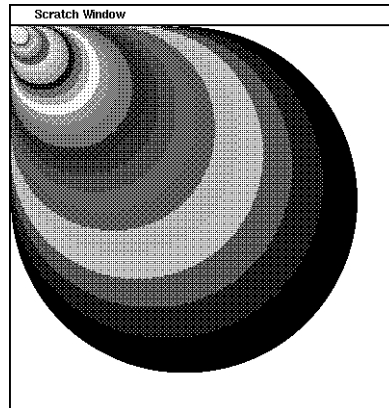
"Round joins"

gc joinStyle: GraphicsContext joinRound.

"Basic Step"

gc displayPolyline: (Array with: 100@400 with: 200@250 with: 300@400).

Coloring a Geometric



Strategy

By default, a color-based display surface (`ApplicationWindow` or `Pixmap`) displays geometric objects in black. The basic step shows how to change the color by installing a new paint (color or pattern) in the graphics context.

When the graphic object is going to be reused and the color information needs to be kept with it, the variant shows how to wrap the geometric object in a wrapper that keeps track of the paint to be used for its component.

Basic Step

- Send a `paint:` message to the graphics context of the display surface. The argument is a color or pattern.

```
| gc circle colors |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
circle := Circle center: 200@200 radius: 200.
colors := ColorValue constantNames.
```

```
colors do: [ :colorName |
  gc paint: (ColorValue perform: colorName).           "Basic Step"
  circle := circle scaledBy: 0.9.
  circle asFiller displayOn: gc]
```

Variant

Storing the Paint with the Geometric Object

1. Wrap the geometric object in a stroking or filling wrapper by sending `asStroker` or `asFiller` to it.
2. Wrap the stroking or filling wrapper in a `GraphicsAttributesWrapper` by sending an `on:` message to that class, with the wrapper from the basic step as the argument.
3. Create a new `GraphicsAttributes` and send a `paint:` message to it. The argument is a color or pattern.
4. Install the graphics attributes in the `GraphicsAttributesWrapper` by sending an `attributes:` message with the attributes as the argument.
5. Display the graphics attributes wrapper by sending a `displayOn:at:` message to it. The first argument is the graphics context of the display surface. The second argument is the origin point at which the geometric object is to be displayed.

```
| gc circle wrapper1 wrapper2 random pt attributes1 attributes2 |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
circle := Circle center: 0@0 radius: 50.
```

```
wrapper1 := GraphicsAttributesWrapper on: circle asFiller. "Variant Steps 1, 2"
attributes1 := GraphicsAttributes new paint: ColorValue red. "Variant Step 3"
wrapper1 attributes: attributes1. "Variant Step 4"
```

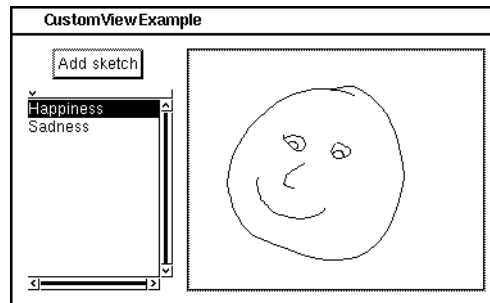
```
wrapper2 := GraphicsAttributesWrapper on: circle asFiller.
attributes2 := GraphicsAttributes new paint: ColorValue blue.
wrapper2 attributes: attributes2.
```

```
random := Random new.
100 timesRepeat: [
    pt := random next * 300 + 50 @ (random next * 300 + 50).
    wrapper1 displayOn: gc at: pt. "Variant Step 5"
    pt := random next * 300 + 50 @ (random next * 300 + 50).
    wrapper2 displayOn: gc at: pt]
```

See Also

- “Creating a Color” on page 686
- “Creating a Tiled Pattern” on page 692

Integrating a Graphic into an Application



Strategy

Displaying graphic objects directly onto a window, as in the examples shown previously, is fine for ad hoc testing. However, an overlapping window quickly damages the displayed object because it is not integrated into the *damage repair* mechanism provided by VisualWorks.

The technique for integrating a graphic relies on the fact that a view is automatically sent a `displayOn:` message whenever its containing window perceives that the view's display area is in need of repair. For a graphic that changes when the model changes, as in the example, the application model can trigger the displaying method whenever necessary. Thus, a view gets display requests from two sources: the window-repair mechanism and the application. Requests of the first kind happen automatically; you arrange for the second in your application, as shown in the basic steps.

In the example, a `SketchView1` updates its display when any of three changes occur in the model (a `Sketch`): a point is added to the current stroke (step), the sketch is erased (step 3), or a new sketch is selected in the list of sketches (step 5). Each of these three events demonstrates a variant in the basic mechanism for keeping the displayed graphic up to date.

Basic Steps

Online example: `CustomView1Example`, `Sketch` and `SketchView1`

1. In the view that is responsible for displaying the graphic (in the example, `SketchView1`), create a `displayOn:` method. This method is responsible for creating the graphic objects based on data from the model. The method displays the graphic objects on the graphics context that is supplied as the method argument. This method is triggered whenever an `invalidate` message is received by the view, as when window damage occurs or the view is notified of a change in the model.

```

displayOn: aGraphicsContext                                "Basic Step 1"
    self model isNil ifTrue: [^self].

    self model strokes do: [ :stroke |
        aGraphicsContext displayPolyline: stroke].

```

2. In any method in the domain model (`Sketch`) that affects the graphics being displayed, send a `changed:with:` message to `self`. The first argument is a symbol identifying the nature of the change (`#stroke`, because a point has been added to the current stroke in the sketch). The second argument is a data or control parameter that will be needed by the view to display the appropriate graphic (in the example, a line segment is sent, which is all that the view needs to add to its display of the sketch). The `changed:with:` message causes an `update:with:` message with the same parameters to be sent to all dependents of the model—the view is the primary and often the only dependent.

```

add: aPoint
    "Add aPoint to the current stroke."

    self strokes last add: aPoint.
    self changed: #stroke with: self currentLineSegment.    "Basic Step "

```

3. When the change in the model is such that the view needs no data or control parameter, use `nil` as the second argument in the `changed:with:` message. In the example, when the `Sketch` model erases all or part of itself, it specifies `#erase` as the first argument in the `changed:with:` message and `nil` as the second argument, because the view has no way of

removing part of the drawing except to display the new sketch entirely.

eraseAll

"Erase my contents."

self strokes removeAll: self strokes copy.

self changed: #erase with: nil.

"Basic Step 3"

4. In the view (SketchView1), create an `update:with:` method in an updating protocol. This method is invoked by the model whenever it changes and is responsible for updating its display based on the aspect of the model that changed. In the example, it displays a new line when the `#stroke` aspect is changed. When the sketch is `#erased`, the `update:with:` method sends `invalidate` to the view. This inherited method causes a `displayOn: message` to be sent to the view with the appropriate graphics context.

update: anAspect with: anObject

"Basic Step 4"

"When a point is added to the model..."

anAspect == #stroke

ifTrue: [anObject asStroker displayOn: self graphicsContext].

"When the model erases its contents..."

anAspect == #erase

ifTrue: [self invalidate].

5. When an entirely new model is given to the view using its `model: method`, the view sends `invalidate` to itself, again causing a `displayOn: message` to be sent to the view with the appropriate graphics context. Because `model:` overrides an inherited method with that name, begin the method by invoking the inherited version by sending a `model: message` to `super`.

model: aModel

super model: aModel.

self invalidate.

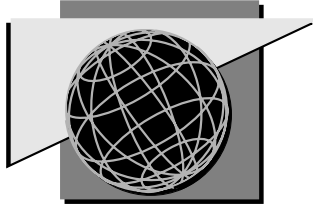
"Basic Step 5"

"Tell the controller where to send menu messages."

self controller performer: aModel.

See Also

- “Defining What a View Displays” on page 380



Chapter 30

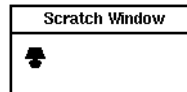
Images, Cursors, and Icons

Creating a Graphic Image	658
Displaying an Image	662
Coloring Pixels in an Image	664
Masking Part of an Image	666
Expanding or Shrinking an Image	668
Flopping an Image	669
Rotating an Image	670
Layering Two Images	672
Caching an Image	674
Animating an Image	675
Creating a Cursor	678
Changing the Current Cursor	681
Creating an Icon	682
Associating an Icon with a Window	683

See Also

- “Geometrics” on page 629
- “Color” on page 685

Creating a Graphic Image



Strategy

A graphic image is a rectangular painting made up of colored pixels arranged in rows. Complex graphics that involve non-geometric elements are typically graphic images. The Image Editor enables you to paint an image pixel by pixel, and then store it, encoded textually, in a compilable resource method (basic steps). Because of the size of this encoding, the Image Editor is best suited for producing small images (such as for cursor shapes or icons).

You can also capture a graphic image from the screen, either using an Image Editor (first variant) or programmatically (second variant). Images captured through the Image Editor are limited to 128 pixels square; images captured programmatically can be of arbitrary size. An image can be captured from a non-VisualWorks window, although only the colors that are in the VisualWorks color palette will be represented accurately.

Because you can color the pixels in an image in a variety of ways, the third variant shows how to create a blank image of a particular size.

A display surface can be converted into an image, as shown in the fourth variant. This is useful when you want to assemble an image by displaying a set of graphic objects on a window or Pixmap. It is also a convenient way of capturing the graphic contents of an existing window.

When you want to use an image that was created with another application, but you have no means of displaying it for a capture operation, you may have to convert its stored numeric data into a `ByteArray`. Then you can use the fifth variant to create an image from the array of bytes.

Basic Steps

1. Open an Image Editor, for example, by choosing Tools→Image Editor from the VisualWorks main window.
2. Paint the desired image in the scrollable pixel grid. To do this, click on a color and then click on each pixel to be painted that color.
3. Use the Image Editor's Install button to create a method that returns the image, typically a class method in a resource protocol of an application model class.

Variants

V1. Capturing an Image from the Screen

1. In an Image Editor, choose the Image→Capture command. The cursor changes to a cross-hair.
2. Press the <Select> mouse button at the upper-left corner of the desired rectangle, drag to the lower-right corner, and then release the mouse button. The rectangle is limited to 1024 pixels on a side.
3. If desired, edit the captured image by changing the color of individual pixels.
4. Use the Image Editor's Install button to create a method that returns the image, typically a class method in a resource protocol of an application model class.

V2. Capturing a Screen Image Programmatically

1. In a Workspace, send a fromUser message to the Image class. The cursor changes to a cross-hair.

```
| gc capturedImage |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
capturedImage := Image fromUser.                                "V1 Step 1"
capturedImage displayOn: gc.
```

2. Press the <Select> mouse button at the upper-left corner of the desired rectangle, drag to the lower-right corner, and then release the mouse button.

V3. Creating a Blank Image

- Send an `extent:depth:palette:` message to the `Image` class. The `extent` argument is a `Point` whose `x` coordinate controls the width of the image (in pixels) and whose `y` coordinate controls the height. The `depth` argument is an integer indicating the color depth of the image—that is, the number of bits required to represent a color in the palette that the image uses. The `palette` argument is a color palette from which the image draws its colors.

```
"Inspect"
| blankImage palette |
palette := Screen default colorPalette.
```

```
blankImage := Image                                     "V3 Step"
  extent: 8@8
  depth: (palette depth)
  palette: palette.
^blankImage
```

V4. Creating an Image from a Display Surface

- Send an `asImage` message to a display surface (window, `Pixmap`, or `Mask`). In the case of a window, the window must not be overlapped by other windows.

```
| gc window image |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
window := Window currentWindow.
  window raise.
```

```
image := window asImage.                               "V4 Step"
image displayOn: gc.
```

V5. Creating an Image from a Byte Array

- Send an `extent:depth:palette:bits:pad:` message to the `Image` class. The first three arguments are the same as in step V3. The `bits` argument is a `ByteArray` specifying the color for each pixel, using the color encodings from the palette with a multiple of 32 bits per row. The `pad` argument is 8, 16, or 32.

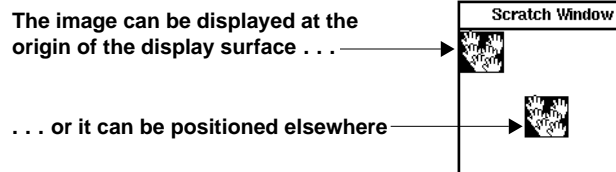
Because each row in the byte array must contain a multiple of 32 bits, the pad size appends 8 bits to a 24-bit row, 16 bits to a 16-bit row (as in the example), or none to a 32-bit row, as a convenience.

```
| gc lamplImage |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

lamplImage := Image                                     "V5 Step"
  extent: 16@16
  depth: 1
  palette: MappedPalette whiteBlack
  bits: #[
    2r00011111 2r11111000
    2r00011111 2r11111000
    2r00111111 2r11111100
    2r00111111 2r11111100
    2r00111111 2r11111100
    2r01111111 2r11111110
    2r01111111 2r11111110
    2r11111111 2r11111111
    2r11111111 2r11111111
    2r00000011 2r11000000
    2r00001111 2r11110000
    2r00011111 2r11111000
    2r00011111 2r11111000
    2r00001111 2r11110000
    2r00001111 2r11110000
    2r00000111 2r11100000]
  pad: 16.

lamplImage displayOn: gc at: 10@10.
```

Displaying an Image



Strategy

As with other visual objects, an image can display itself on a graphics context, as shown in the basic steps. Note that an image's palette cannot be color-based if you intend to display it on a coverage-based Mask rather than a color-based Window or Pixmap.

A common situation requires creating a hidden display surface (Mask or Pixmap) of the same size as an image and then displaying the image on it. The variant shows how to accomplish this in a single step.

Basic Steps

1. Send a `displayOn:` message to the image. The argument is the graphics context of the display surface on which the image is to be displayed.
2. To specify a display origin other than the default `0@0`, send a `displayOn:at:` message to the image. The first argument is the graphics context and the second argument is a Point indicating the origin of the image relative to the display surface's origin.

```
| gc logo |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
```

```
logo displayOn: gc.                                "Basic Step 1"
logo displayOn: gc at: 50@50.                       "Basic Step 2"
```

Variant

Creating a Display Surface Bearing an Image

- Send an `asRetainedMedium` message to the image. If the image has a color-based palette, a `Pixmap` will be returned. If the image has a coverage-based palette, a `Mask` will be returned.

```
"Inspect"  
| image pixmap |  
image := LogoExample logo.
```

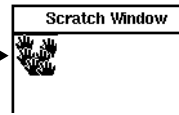
```
pixmap := image asRetainedMedium. "Variant Step"  
^pixmap
```

See Also

- “Defining What a View Displays” on page 380
- “Integrating a Graphic into an Application” on page 652

Coloring Pixels in an Image

The colors have been inverted by changing white to black and black to white



Strategy

In an application such as the Image Editor, which enables you to paint a new image or edit an existing image, individual pixel colors can be changed as shown in the basic steps. You can also specify the color to be applied using the numeric equivalent, as shown in the variant.

The colors that you substitute, however, must exist in the image's palette. For example, if the image's palette contains only black and white, as in the example image used in the basic steps, the most you can do is reverse a given pixel from white to black or from black to white.

Basic Steps

1. To get the current color of a pixel, send a `valueAtPoint:` message to the image. The argument is a `Point` indicating the coordinates of the pixel in the image.
2. To change the color of a pixel, send a `valueAtPoint:put:` message to the image. The first argument is the location of the pixel, and the second is a color that exists in the image's palette.

```
| gc logo oldColor newColor white black |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
white := ColorValue white.
black := ColorValue black.
```

"Change each black pixel to white, and vice versa."

```
0 to: logo height - 1 do: [:y |
  0 to: logo width - 1 do: [:x |
    oldColor := logo valueAtPoint: x@y.           "Basic Step 1"
    oldColor = white
      ifTrue: [newColor := black]
      ifFalse: [newColor := white].
```

```
logo valueAtPoint: x@y put: newColor]].           "Basic Step 2"
```

```
logo displayOn: gc
```

Variant

Specifying the New Color by Its Encoded Number

1. To get the current color number of a pixel, send an `atPoint:` message to the image. The argument is a `Point` indicating the coordinates of the pixel in the image. The number that identifies the pixel color in the image's palette is returned.
2. To change the color of a pixel, send an `atPoint:put:` message to the image. The first argument is the location of the pixel and the second argument is a color number that exists in the image's palette.

```
| gc logo oldColor newColor |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
```

```
"Change each black pixel to white, and vice versa."
```

```
0 to: logo height - 1 do: [:y |
  0 to: logo width - 1 do: [:x |
    oldColor := logo atPoint: x@y.           "Variant Step 1"
    oldColor = 1
      ifTrue: [newColor := 0]
      ifFalse: [newColor := 1].
```

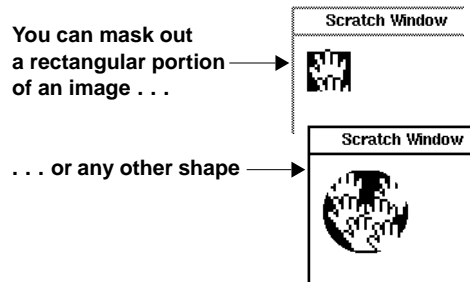
```
logo atPoint: x@y put: newColor]].           "Variant Step 2"
```

```
logo displayOn: gc
```

See Also

- “Creating a Color” on page 686
- “Changing an Image’s Color Palette” on page 696
- “Changing the Policy for Rendering Colors” on page 698

Masking Part of an Image



Strategy

Sometimes an image contains extraneous material that needs to be removed. The basic steps show how to copy a rectangular portion.

When the desired portion of an image is not rectangular, you can create a Mask whose shape matches the desired portion. The mask is then used as a kind of stencil through which the image is displayed onto a graphics context.

Basic Steps

1. Create a display surface containing the image by sending an `asRetainedMedium` message to the image.
2. Send a `completeContentsOfArea:` message to the display surface. The argument is a rectangle that defines the desired portion of the image. The copied portion is returned as an image.

```
| gc logo subImage pixmap copyRect |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
logo := LogoExample logo magnifiedBy: 2@2.
```

```
pixmap := logo asRetainedMedium.                                "Basic Step 1"
copyRect := 0@0 extent: (logo width @ logo height / 2) rounded.
```

```
subImage := pixmap completeContentsOfArea: copyRect.           "Basic Step 2"
subImage displayOn: gc at: 10@10.
```

Variant

Masking a Nonrectangular Portion

1. Create a display surface on which the image has been displayed by sending `asRetainedMedium` to the image.
2. Create the desired mask by sending an `extent: message` to the `Mask` class. The argument is a `Point` indicating the size of the mask. You can display the desired shape or shapes on the `Mask` as with a window or other display surface (in the example, a solid oval is displayed). The shapes on the mask define the regions through which the image will be visible.
3. Send a `copyArea:from:sourceOffset:destinationOffset: message` to the graphics context of the destination display surface (in the example, the scratch window). The `copyArea` argument is the mask. The `from` argument is the graphics context of the source display surface (the pixmap containing the logo). The `sourceOffset` argument is a `Point` indicating the origin of the mask when placed over the source display surface. The `destinationOffset` argument is the origin of the subimage when displayed on the destination display surface.

```

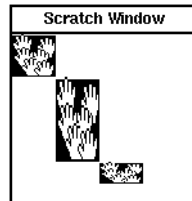
| gc logo pixmap ovalMask |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
logo := LogoExample logo magnifiedBy: 2@2.
pixmap := logo asRetainedMedium.                                     "Variant Step 1"

ovalMask := Mask extent: 66@66.                                     "Variant Step 2"
ovalMask graphicsContext
  displayWedgeBoundedBy: ovalMask bounds
  startAngle: 0
  sweepAngle: 360.

gc copyArea: ovalMask                                             "Variant Step 3"
  from: pixmap graphicsContext
  sourceOffset: 0@0
  destinationOffset: 10@10.

```

Expanding or Shrinking an Image



Strategy

You can get a copy of an image that has been magnified or shrunk in the x dimension, the y dimension, or both, as shown in the basic steps.

Basic Steps

1. To get an expanded copy of an image, send a `magnifiedBy:` message to the image. The argument is a `Point` whose `x` value is multiplied by the width of the image to derive the width of the expanded version; similarly, the `y` value controls the height of the expanded version.
2. To shrink an image, send a `shrunkBy:` message to the image. The argument is a point that is used as a divisor to reduce the width and height in the shrunk version.

```
| gc logo bigLogo tinyLogo |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
logo := LogoExample logo.
```

```
bigLogo := logo magnifiedBy: 1@2.
```

"Basic Step 1"

```
tinyLogo := logo shrunkBy: 1@2.
```

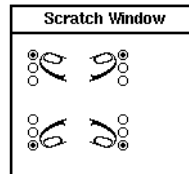
"Basic Step 2"

```
logo displayOn: gc.
```

```
bigLogo displayOn: gc at: logo extent.
```

```
tinyLogo displayOn: gc at: logo extent + bigLogo extent.
```

Flopping an Image



Strategy

Sometimes you need a mirror copy of an image. The basic steps show how to get a reflected copy in which the imaginary mirror is aligned with the x axis, the y axis, or both. This process of rotating an image about the x axis or the y axis is known as *flopping an image*, from the photographic process in which a negative is flopped onto its backside to produce a mirror image.

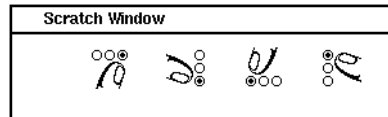
Basic Steps

1. To flop an image about the x axis, send a `reflectedInX` message to the image.
2. To flop an image about the y axis, send a `reflectedInY` message.
3. To flop an image about both axes, send a `reflectedInX` message followed by a `reflectedInY` message.

```
| gc helpImage |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
helpImage := VisualLauncher helpIcon asImage.
```

```
helpImage
  displayOn: gc at: 10@10.
helpImage reflectedInX
  displayOn: gc at: 60@10.
helpImage reflectedInY
  displayOn: gc at: 10@60.
helpImage reflectedInX reflectedInY
  displayOn: gc at: 60@60.
```

Rotating an Image



Strategy

You can rotate an image about the z axis in 90-degree increments, as shown in the basic step.

Each rotated copy uses time and memory resources. For a series of rotations, you can reduce the resources required by reusing the same scratch image for each subsequent copy, as shown in the variant. The scratch image must be of the same size as the unrotated image, so this technique works only when all images in the series are the same size.

Basic Step

1. Send a `rotatedByQuadrants:` message to the image. The argument is an integer indicating how many 90-degree rotations you want. A rotated copy of the image is returned.

```
| gc helpImage rotatedImage |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
helpImage := VisualLauncher helpIcon asImage.

rotatedImage := helpImage rotatedByQuadrants: 1.           "Basic Step"

helpImage
  displayOn: gc at: 10@10.
rotatedImage
  displayOn: gc at: 60@10.
```

Variant

Reusing the Rotated Image

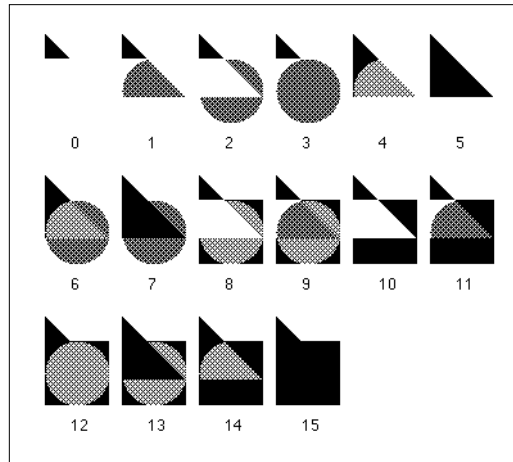
1. Create a scratch image the same size as the image that is to be rotated by sending a `copyEmpty` message to the original image.
2. Send a `rotateByQuadrants:to:` message to the image to be copied. The first argument is the number of quadrants to rotate the image. The second argument is the scratch image.

```
| gc helpImage scratchImage |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
helpImage := VisualLauncher helpIcon asImage.
```

```
scratchImage := helpImage copyEmpty. "Variant Step 1"
```

```
1 to: 4 do: [:quads |  
  helpImage rotateByQuadrants: quads to: scratchImage. "Variant Step 2"  
  scratchImage displayOn: gc at: (60 * quads) @ 10]
```

Layering Two Images



Strategy

You can achieve a variety of layering effects by combining two images and applying a filtering algorithm to the overlapping portions. VisualWorks provides 16 built-in algorithms, called *combination rules*. The rules are numbered 0 through 15, and the more commonly used rules have names. Thus, sending an erase message to the RasterOp class returns the combination rule for erasing shared pixels from the combined image. Combining two images involves copying a region from one image (the source) onto the other image (the destination), applying the combination rule.

Raster operations work correctly only on monochrome screens that have the most commonly used polarity characteristics. On color screens and on monochrome screens of the opposite polarity, the effects will be unpredictable. Because of this, only the RasterOp over rule is portable across screen types.

Basic Steps

1. To preserve the destination image in its unchanged state, make a copy on which to merge the source image.

2. Send a copy:from:in:rule: message to the destination image (in the example, triangle). The copy argument is a rectangle identifying the region in the destination image to be merged with the source image (the lower part of the triangle). The from argument is the origin of the rectangle within the source image (the origin of the circle, because we want to copy the entire circle). The in argument is the source image. The rule argument is an integer identifying a combination rule (which can be derived by sending and, over, erase, reverse, under, or reverseUnder to the RasterOp class).

```
| gc triangle circle scratch |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

triangle := Pixmap extent: 50@100.
triangle graphicsContext
  displayPolygon: (Array
    with: 0@0
    with: 0@50
    with: 50@50).
triangle := triangle asImage.

circle := Pixmap extent: 50@50.
circle graphicsContext
  displayDotOfDiameter: 50
  at: 25@25.
circle := circle asImage.

0 to: 15 do: [ :rule |
  scratch := triangle copy,                                     "Basic Step 1"
  scratch                                       "Basic Step 2"
    copy: (0@20 extent: 50@50)
    from: 0@0
    in: circle
    rule: rule.

  scratch displayOn: gc at: (50 * rule \\ 400) @ (50 * rule // 400 * 100)]
```

Caching an Image

Strategy

A display surface such as a `Pixmap` usually can be displayed on another display surface (such as a window) more quickly than an equivalent `Image`. However, an `Image` has greater longevity because it does not require a resource from the operating system, and thus it survives when you quit and restart VisualWorks. A `CachedImage` combines the longevity of an `Image` with the displaying speed of a display surface. Whenever its display surface is unavailable, as when it has been destroyed by a save-and-restart operation, it is recreated from the image automatically. This relieves your application from having to recreate such display surfaces manually. The images used by VisualWorks, such as the insertion point in the text editor, use cached images in this way.

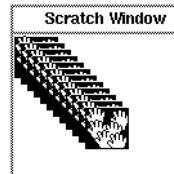
Note that a `CachedImage` must be treated like a display surface, not an image. For example, you cannot rotate a `CachedImage`.

Basic Step

- Create a `CachedImage` by sending an `on:` message to that class. The argument is the image that is to be cached.

```
| gc logo |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
  
logo := CachedImage on: LogoExample logo.           "Basic Step"  
logo displayOn: gc.
```

Animating an Image



Strategy

Animating an image, or any graphic object, consists of creating a loop in which the object is drawn and erased at successive locations along a path. All subclasses of `VisualComponent`, including `Image`, can animate themselves in this way, as shown in the basic step.

The first technique is limited to a single object, so it is not useful when multiple objects are being animated, nor when the object has multiple phases, such as a walking robot. This technique also suffers from a phenomenon known as *flashing*, which results when the new location overlaps the previous location—the overlapping pixels are first erased (unnecessarily) and then redrawn.

For smoother animation, a technique called *double buffering* is used. With double buffering, when the new location overlaps the old location, only the pixels that need to be erased or drawn are affected. Double buffering is also useful when multiple objects are being animated together, because an entire scene can be assembled on a hidden `Pixmap` and then substituted for the current scene all at once. Double buffering tends to be slower, especially in a medium to large window. The variant demonstrates a technique for double buffering during animation.

Basic Step

- Send a `follow:while:on:` message to the image. The `follow` argument is a block in which the origin of the image is shifted to the next location in the path. The `while` argument is a block that provides a test for ending the display loop.

The on argument is the graphics context of the display surface on which the animation is to take place.

```
| gc logo origin jump |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
logo := LogoExample logo.  
origin := 0@0.  
jump := 3@3.
```

```
logo "Basic Step"  
  follow: [origin := origin + jump]  
  while: [origin x < 400]  
  on: gc.
```

Variant

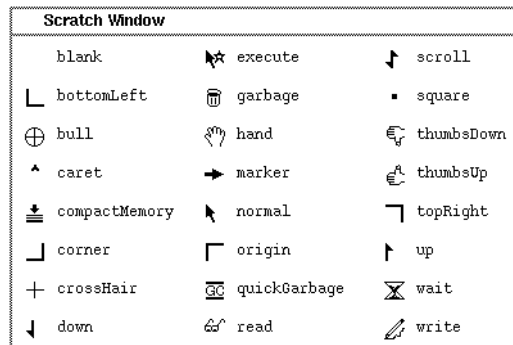
Animating with Double Buffering

1. Create a Pixmap of the same size as the window on which the animation is to take place by sending an extent: message to the Pixmap class. The argument is a rectangle with the window's dimensions, which can be derived by sending a clippingBounds message to the window's graphics context.
2. Create a loop in which the erase-and-display operations occur.
3. Inside the loop, begin by moving the origin of each object to be animated.
4. Still inside the loop, erase the Pixmap by sending a clear message to it.
5. Still inside the loop, display each animated object in its new location.
6. Still inside the loop, display the Pixmap on the window.

```
| gc buffer logo windowSize origin1 origin2 jump bufferGC |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
windowSize := gc clippingBounds extent.  
logo := LogoExample logo.  
origin1 := 0@0.  
origin2 := 360@0.  
jump := 5.
```

buffer := Pixmap extent: windowSize. bufferGC := buffer graphicsContext.	"Variant Step 1"
80 timesRepeat: [origin1 := origin1 + jump. origin2 := (origin2 x - jump) @ (origin2 y + jump).	"Variant Step 2" "Variant Step 3"
"Clear the buffer, then assemble the next scene." buffer clear. logo displayOn: bufferGC at: origin1. logo displayOn: bufferGC at: origin2.	"Variant Step 4" "Variant Step 5"
"Display the next scene." buffer displayOn: gc]	"Variant Step 6"

Creating a Cursor



Strategy

The cursor is familiar to you as the pictorial object that represents the mouse pointer. A variety of built-in cursors are available to indicate various kinds of application activity, such as an hourglass for when the user must wait for processing to be completed. The basic step shows how to access one of the built-in cursors. The first variant shows a technique for displaying all of the cursors with their names, so you can choose an appropriate one.

The second variant shows how to create a cursor from scratch. The technique requires creating an image that defines the appearance of the cursor. In addition, a second image is used as a mask to define the opaque areas in the first image. For example, the familiar arrow cursor has a mask that is arrow-shaped but slightly larger than the arrow, so that the arrow has a one-pixel border of opacity.

A cursor has a control point or hot spot, which is the single pixel that defines the cursor's location on the screen. For the arrow cursor, for example, the control point is at the tip of the arrow. When creating a new cursor, you must also specify its control point, as shown in the second variant.

Basic Step

- The `Cursor` class provides methods for accessing the built-in cursors. Send one of those messages to the `Cursor` class to

access the corresponding cursor. (To see the available cursors, try the first variant below.)

```
| gc cursor |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

cursor := Cursor wait.                                     "Basic Step"
cursor displayOn: gc at: 10@10.
```

Variants

V1. Displaying the Available Cursors

1. Get a list of the methods for accessing built-in cursors by sending a `listAtCategoryNamed: message` to the `Cursor` class organization. The argument is `#constants` (the name of the protocol containing the methods).
2. Create a loop in which each cursor is displayed along with its name. (Because you probably won't use this code in an application, it won't be described in detail.)

```
| gc cursorNames index topLeftMargin columnWidth rowHeight x y cursor |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
index := 0.
topLeftMargin := 10.
columnWidth := 140.
rowHeight := 30.

cursorNames := Cursor class organization                    "Variant Step 1"
listAtCategoryNamed: #constants.

cursorNames do: [ :cName |                                  "Variant Step 2"
  x := topLeftMargin + (columnWidth * (index//8)).
  y := topLeftMargin + (rowHeight * (index\8)).
  cursor := Cursor perform: cName.

  cursor displayOn: gc at: x@y.
  cName displayOn: gc at: (x + 25) @ (y + 10).
  index := index + 1]
```

V2. Creating a New Cursor

Online example: CursorExample

1. Create an image that provides the pictorial element in the cursor. If you use the Image Editor to create the image, you must convert its palette to a color-based palette rather than a coverage-based one. To do so, edit the resource method that defines the image, substituting `MappedPalette whiteBlack` (or another two-color palette) for the default `CoveragePalette monoMaskPalette`.
2. Create a coverage-based image that defines the opaque portion of the first image. The Image Editor can be used to create this image. Typically, it is the same shape as the image from step 1, but completely darkened and one pixel larger on each side.
3. Create the cursor by sending an `image:mask:hotSpot:name:` message to the `Cursor` class. The `image` argument is the color-based image that you created in step 1. The `mask` argument is the coverage-based image from step 2. The `hotSpot` argument is a point indicating which pixel in the image is the control point. The `name` argument is a string containing a descriptive name for the cursor. The name is of little importance, but it is displayed when you inspect a cursor.

```
| cursor colorImage maskImage |
colorImage := CursorExample townCrierForCursor.           "V2 Step 1"
maskImage := CursorExample shadow.                         "V2 Step 2"

cursor := Cursor                                           "V2 Step 3"
  image: colorImage
  mask: maskImage
  hotSpot: 8@8
  name: 'townCrier'.
cursor showWhile: [(Delay forSeconds: 3) wait].
```

Changing the Current Cursor

Strategy

By default, an arrow cursor is displayed and moves in response to mouse movements. You can display a different cursor as a way of indicating that your application is processing information (reading or writing data, for example). Changing the cursor is also a means of indicating to the user of your application that a certain kind of input is expected—for example, the `crossHair` cursor is typically used to indicate that a drawing operation is expected. The basic steps show how to display a different cursor and then revert to the normal cursor when appropriate.

In the example, the `SketchController1` causes the cursor to change to a cross-hair whenever it comes within the boundaries of the `SketchView1`.

Basic Step

Online example: `CustomView1Example` and `SketchController1`

- Send a `showWhile:` message to the cursor. The argument is a block containing the actions that are to take place while the cursor is in its changed state. After the actions in the block are finished, the cursor will return to normal automatically. (In the example, the controller changes the cursor for as long as it holds onto control.)

`controlLoop`

"Change the cursor to a cross-hair for drawing."

Cursor `crossHair` `showWhile:` [`super controlLoop`].

"Basic Step"

Creating an Icon

Strategy

Most often used to represent a collapsed window, an icon typically provides a pictorial clue to the nature of the window. The basic steps show how to create an icon.

Basic Steps

1. Create an Image containing the pictorial element for the icon. You can use the Image Editor to create the image and save it in a method (in the example, the image is returned by the `townCrier` method of the `CursorExample` class).
2. Create a Mask containing the image by sending an `asRetainedMedium` message to the image.
3. Create an icon by sending an `image: mask:` message to the `Icon` class. The argument is the mask from step 2.

```
| icon gc image mask |
```

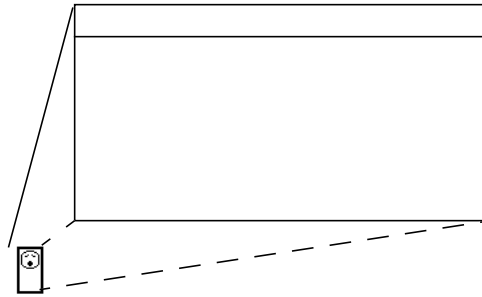
```
image := CursorExample townCrier.           "Basic Step 1"  
mask := image asRetainedMedium.           "Basic Step 2"
```

```
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
```

```
icon := Icon image: mask.                   "Basic Step 3"  
icon displayOn: gc at: 10@10.
```

Note that the `Icon` instance takes care of reconstructing the `Mask` instance when you quit and restart `VisualWorks`.

Associating an Icon with a Window



Strategy

The default icon that VisualWorks displays for each collapsed window may not be appropriate for your application's windows. The basic step shows how to associate a custom icon with a window. This is typically done in the method that creates the window.

Basic Step

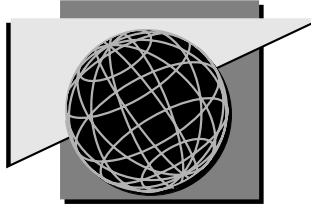
- Send an `icon:` message to the window. The argument is the icon that is to be displayed when the window is collapsed.

```
| icon window mask |
mask := CursorExample townCrier asRetainedMedium.
icon := Icon image: mask.
window := ApplicationWindow new.
```

```
window icon: icon.
```

"Basic Step"

```
window open.
(Delay forSeconds: 1) wait.
window collapse.
```

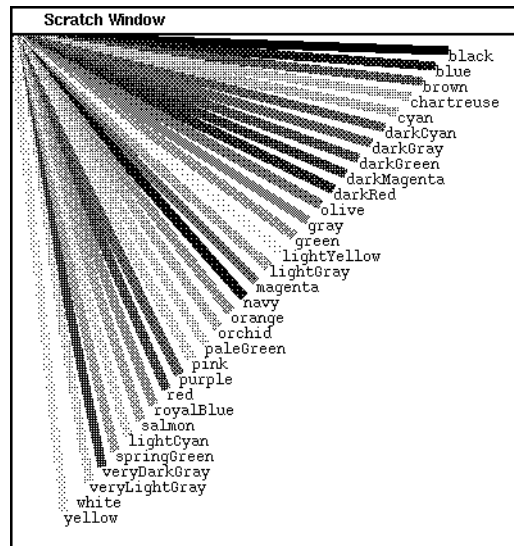


Chapter 31

Color

Creating a Color	686
Creating a Coverage	690
Creating a Tiled Pattern	692
Applying a Color or Pattern	694
Changing an Image's Color Palette	696
Changing the Policy for Rendering Colors	698

Creating a Color



Strategy

The Properties Tool enables you to choose and apply a color for a window or widget. When the color of a graphic element needs to be changed dynamically, you can create a color programmatically. A set of predefined colors is provided by the `ColorValue` class, as demonstrated in the basic step. The first variant shows a technique for displaying all of the color constants along with their names.

When a predefined color will not suffice, you can create a color by specifying its percentages of red, green, and blue (the primary colors in the world of computer monitors), as shown in the second variant. This approach enables you to create fine gradations of color and lends itself to algorithmic generation of color, in which numeric values are represented as colors.

For some applications, the red-green-blue or RGB approach to creating a color does not suffice. This is especially so when three-dimensional shading effects are involved, because it is not easy to darken or lighten a color when you cannot manipulate a black or white component. For this reason you can also

create a color using hue, saturation, and brightness (known as HSB color), as shown in the third variant.

Basic Step

- Send a cyan message to the `ColorValue` class, or another message identifying one of the predefined color constants.

```
| gc color |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

color := ColorValue cyan.                                     "Basic Step"

gc paint: color.
gc displayDotOfDiameter: 400 at: 200@200.
```

Variants

V1. Displaying the Predefined Colors

1. Get the list of color constants by sending a `constantNames` message to the `ColorValue` class.
2. Create a loop in which a thick line is drawn in each color, with the color name displayed at the end of the line. (Because this is not likely to be a loop that you will use in an application, it will not be described in detail.)

```
| gc endPoint colors |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
gc lineWidth: 7.
endPoint := 350@0.

colors := ColorValue constantNames.                          "V1 Step 1"

colors do: [ :c |                                           "V1 Step 2"
  endPoint := endPoint + (-10@12).
  gc paint: (ColorValue perform: c).
  gc displayLineFrom: 0@0 to: endPoint.
  gc paint: ColorValue black.
  c asString displayOn: gc at: endPoint + (0@8)]
```

V2. Creating a Color from Red, Green, and Blue

- Send a `red:green:blue:` message to the `ColorValue` class. All arguments are numbers between zero and one, representing the intensity of their respective colors. (In the example, the intensity of green is varied while the red and blue intensities remain at zero.)

```
| gc origin |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
origin := 0@0.

1 to: 0 by: -0.01 do: [ :grn |
  gc paint: (ColorValue red: 0.0 green: grn blue: 0.0).          "V2 Step"
  origin := origin + 4.
  gc displayRectangle: (origin extent: 400 - origin)]
```

V3. Creating a Color from Hue, Saturation, and Brightness

- Send a `hue:saturation:brightness:` message to the `ColorValue` class. The hue argument is a number from 0 to 1, where 0 is red, 0.333 is green, 0.667 is blue, and 1 is red again. The saturation argument is a number from 0 to 1, representing minimum vividness (white) to full color; a more saturated color makes an object appear closer to the viewer. The brightness argument is a number from 0 to 1, representing minimum brightness (black) to full color; varying the brightness is useful for representing shadows.

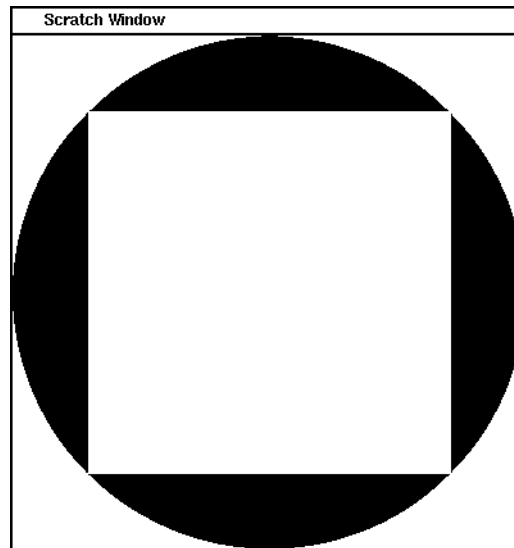
```
| gc r x y |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
r := 50.
gc lineWidth: 2.

gc translation: 150@150.
0 to: 1 by: 0.005 do: [ :i |
  x := (i * Float pi) cos * r.
  y := (i * Float pi) sin * r / 2.
  gc paint: (ColorValue hue: 0.0 saturation: 0.5 brightness: i).    "V3 Step"
  gc displayLineFrom: x@y to: 0@-100 ].
```

```
gc translation: 200@200.  
0 to: 1 by: 0.005 do: [:i |  
  x := (i * Float pi) cos * r.  
  y := (i * Float pi) sin * r / 2.  
  gc paint: (ColorValue hue: 0.0 saturation: 0.75 brightness: i).  
  gc displayLineFrom: x@y to: 0@-100 ].
```

```
gc translation: 250@250.  
0 to: 1 by: 0.005 do: [:i |  
  x := (i * Float pi) cos * r.  
  y := (i * Float pi) sin * r / 2.  
  gc paint: (ColorValue hue: 0.0 saturation: 1.0 brightness: i).  
  gc displayLineFrom: x@y to: 0@-100 ]
```

Creating a Coverage



Strategy

In a window or Pixmap, each pixel can be assigned a different color. In a Mask, each pixel is assigned a level of opaqueness—that is, 0 (transparent) or 1 (opaque). The mask is then used as a stencil through which a graphic image is projected onto another display surface. Each opaque pixel in the mask causes the corresponding pixel in the image to be displayed. (This works the opposite way from a physical stencil, in which the opaque regions block paint.) A CoverageValue is used to represent the level of opaqueness associated with a pixel, as shown in the basic step.

Basic Step

- Send a coverage: message to the CoverageValue class. The argument is 0 (transparent) or 1 (opaque). As an alternative, you can also send a transparent or opaque message to the CoverageValue class.

```
| gc mask |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
```

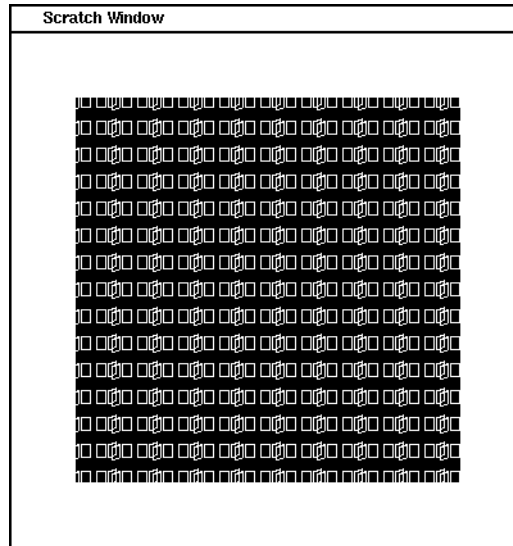

mask := Mask extent: 400@400.

mask graphicsContext
displayDotOfDiameter: 400 at: 200@200.

mask graphicsContext
paint: (CoverageValue coverage: 0); "Basic Step"
displayRectangle: (59@59 extent: 283@283).

mask displayOn: gc at: 0@0.

Creating a Tiled Pattern



Strategy

A Pattern is created by filling a space with a single graphic image that is repeated over and over, like tiles covering a floor. A pattern can be used in the same situations in which you would use a solid color. The basic steps show how to create a tile and from it, a pattern.

By default, the first tile in the pattern is displayed at the origin of the display surface. You can shift that first tile, and with it the entire pattern. This shift, known as the *tile phase*, is sometimes helpful for aligning the edges of the tiles with the edges of the graphic object that is being painted, as in the variant.

Basic Steps

1. Create the graphic image that will serve as the repeating tile in the pattern. You can also use a window, Pixmap, or Mask as the tile.
2. Send an `asPattern` message to the tile.

```
| gc tile |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
  
tile := Image parcPlaceDigitalkLogo shrunkenBy: 4@4.           "Basic Step 1"  
tile := tile asPattern.                                         "Basic Step 2"  
  
gc paint: tile.  
gc displayRectangle: (50@50 extent: 300@300).
```

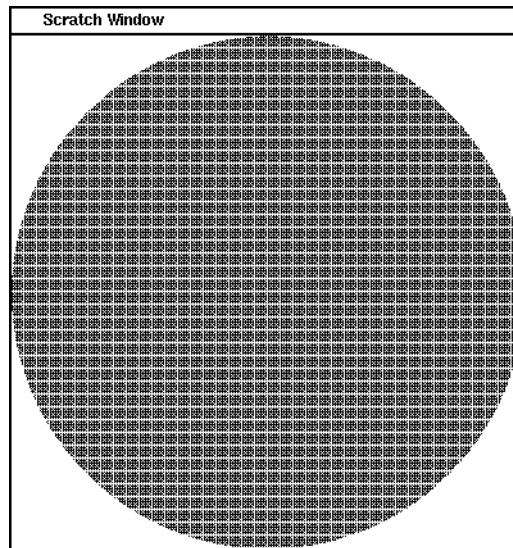
Variant

Adjusting a Pattern's Tile Phase

- Send a `tilePhase:` message to the graphics context of the display surface on which the patterned object is to be displayed. The argument is a point that defines the origin of the first tile in the pattern. (As in the example, the tile phase is often the same as the origin of the painted object, which aligns the tiles with the top and left edges of the object.)

```
| gc tile |  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
tile := Image parcPlaceDigitalkLogo shrunkenBy: 4@4.  
tile := tile asPattern.  
gc paint: tile.  
  
gc tilePhase: 50@50.                                           "Variant Step"  
  
gc displayRectangle: (50@50 extent: 300@300).
```

Applying a Color or Pattern



Strategy

The Properties Tool is the best means of applying color to a widget or a window. For graphic images, the color is defined when the image is created. For colorless graphic objects such as geometrics, the basic step shows how to arrange for a particular paint to be used.

Basic Step

- Send a paint: message to the graphics context of the display surface on which the object is to be displayed. The argument is a color, a pattern, or in the case of a Mask, a coverage.

```
| gc tile |  
tile := Pixmap extent: 10@10.  
gc := tile graphicsContext.
```

```
"Tile background"  
gc paint: ColorValue chartreuse.  
gc displayRectangle: (0@0 extent: 10@10).
```

"Basic Step"

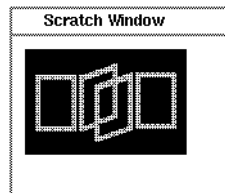
```
"Tile foreground"  
gc paint: ColorValue red.  
gc displayDotOfDiameter: 10 at: 4@4.
```

```
"Patterned circle"  
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.  
gc paint: tile asPattern. "Basic Step"  
gc displayDotOfDiameter: 400 at: 200@200.
```

See Also

- “Coloring a Widget” on page 75
- “Coloring a Window” on page 94
- “Coloring a Geometric” on page 649
- “Coloring Pixels in an Image” on page 664

Changing an Image's Color Palette



Strategy

In a graphic image, each pixel is associated with a color in the image's palette of colors. One way of changing the color scheme is to substitute a different palette. By placing the desired new color at the old color's position in the palette, you can effectively change the color of every pixel in the image that had the old color. The basic steps show how to create a color palette and install it in the image. The new palette must have the same number of color entries as the old palette. In the example, every pixel that was previously white is converted to yellow by changing the white entry in the palette to yellow.

When a color palette differs from the palette used by the display surface, a temporary image is created so VisualWorks can simulate the desired colors when necessary. This step can take a significant amount of time. To display an image quickly, convert it to use the default palette that is used by display surfaces, as shown in the variant. This is especially helpful when the image is to be displayed more than once.

Basic Steps

1. Create an array containing the new colors. To access the existing palette's array of colors, send a `palette` message to the image, and then send a `colors` message to the resulting palette. You can then modify the color array as desired (in the example, we locate the white entry and substitute the color yellow).
2. Create a new palette by sending a `withColors:` message to the `MappedPalette` class. The argument is the new or modified color array.

3. Install the new palette by sending a palette: message to the image. The argument is the new palette.

```
| gc palette image colors whiteIndex |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
image := InputFieldSpec paletteIcon asImage.
```

colors := image palette colors. "Basic Step 1"
whiteIndex := colors indexOf: ColorValue white.
colors at: whiteIndex put: ColorValue yellow.

palette := MappedPalette withColors: colors. "Basic Step 2"

image := image palette: palette. "Basic Step 3"
image displayOn: gc at: 10@10.

Variant

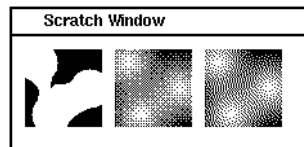
Converting an Image to Use the Default Palette

- Send a convertToPalette: message to the image. The argument is the default color palette, which can be accessed by sending a default message to the Screen class and then sending a colorPalette message to the resulting screen. (For a coverage-based image, send a coveragePalette message instead of colorPalette.)

```
| gc image |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
image := Image parcPlaceDigitalkLogo magnifiedBy: 2@2.
```

image := image convertToPalette: Screen default colorPalette. "Variant Step"
image displayOn: gc at: 10@10.

Changing the Policy for Rendering Colors



Strategy

When a graphic image contains a color that does not exist in the screen's palette, a neighboring color is used instead. VisualWorks provides three different renderers for deciding which color to substitute for a missing color. Of the three, `NearestPaint` is the fastest and is the default on color screens. `OrderedDither` is the default on monochrome and gray-scale screens. `ErrorDiffusion` uses a more sophisticated color-blending algorithm than `OrderedDither` does, but it tends to be slower.

The basic step shows how to convert a graphic image of three colored spotlights to the screen's palette using a specified renderer. Converting the image in this way makes it possible to display the converted image multiple times, instead of leaving it to the display surface to perform the conversion each time the original image is displayed on it.

The image can be converted to a palette other than the screen's palette. In the first variant, the color image is converted to a monochrome palette. This technique is useful when you want to show what the image looks like on a screen that has a limited palette.

Graphic objects other than images do not have their own color, so the rendering is performed by the graphics context of the display surface. The second variant shows how to install the desired renderer in the graphics context. The example paints a series of 4-pixel squares successively, each time shifting the color from red toward green.

Basic Step

- Send a `convertForGraphicsDevice:renderedBy:` message to the image. The first argument is typically `Screen default`. The second argument is `NearestPaint` when you want to use the nearest

available paint from the screen's palette, OrderedDither when you want to synthesize the nonexistent color by blending neighboring colors, and ErrorDiffusion when you want to use a more sophisticated (but often slower) blending algorithm.

"THIS EXAMPLE IS INTENDED TO BE USED ON
A COLOR SCREEN. IT CAN TAKE SEVERAL MINUTES."

```
| gc r g b im |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
im := Image
    extent: 60@60
    depth: 15
    palette: (FixedPalette
        redShift: 10 redMask: 31
        greenShift: 5 greenMask: 31
        blueShift: 0 blueMask: 31).
0 to: 59 do: [:x |
    0 to: 59 do: [:y |
        r := 1 - ((x@y - (10@10)) r / 30) max: 0.
        g := 1 - ((x@y - (20@50)) r / 30) max: 0.
        b := 1 - ((x@y - (50@30)) r / 30) max: 0.
        im atPoint: x@y put: (im palette
            indexOfPaintNearest: (ColorValue red: r green: g blue: b))].
```

```
(im convertForGraphicsDevice: Screen default
    renderedBy: NearestPaint new)
displayOn: gc at: 10@10. "Basic Step"
```

```
(im convertForGraphicsDevice: Screen default
    renderedBy: OrderedDither new)
displayOn: gc at: 80@10. "Basic Step"
```

```
(im convertForGraphicsDevice: Screen default
    renderedBy: ErrorDiffusion new)
displayOn: gc at: 150@10. "Basic Step"
```

Variants

V1. Converting an Image to a Specific Palette

- Send a `convertToPalette:renderedBy:` message to the image. The first argument is the desired palette (in the example, a monochrome palette). The second argument is the desired renderer (a `NearestPaint`, an `OrderedDither`, or an `ErrorDiffusion`).

```
| gc r g b im |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.
im := Image
    extent: 60@60
    depth: 15 palette: (FixedPalette
        redShift: 10 redMask: 31
        greenShift: 5 greenMask: 31
        blueShift: 0 blueMask: 31).
0 to: 59 do: [:x |
    0 to: 59 do: [:y |
        r := 1 - ((x@y - (10@10)) r / 30) max: 0.
        g := 1 - ((x@y - (20@50)) r / 30) max: 0.
        b := 1 - ((x@y - (50@30)) r / 30) max: 0.
        im atPoint: x@y put: (im palette
            indexOfPaintNearest: (ColorValue brightness: 1-((1-r)*(1-g)*(1-
b)))))]].
```

```
(im convertToPalette: MappedPalette whiteBlack
    renderedBy: NearestPaint new) "V1 Step"
displayOn: gc at: 10@10.
```

```
(im convertToPalette: MappedPalette whiteBlack
    renderedBy: OrderedDither new) "V1 Step"
displayOn: gc at: 80@10.
```

```
(im convertToPalette: MappedPalette whiteBlack
    renderedBy: ErrorDiffusion new) "V1 Step"
displayOn: gc at: 150@10.
```

V2. Setting the Rendering Policy for Nonimage Graphics

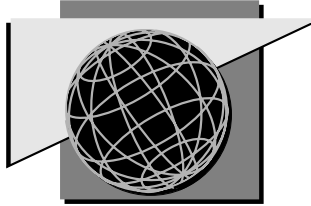
1. Install a paint policy in the graphics context of the display surface by sending a `paintPolicy:` message to the graphics context. The argument is a `PaintPolicy`, typically a new instance.
2. Set the rendering algorithm by sending a `paintRenderer:` message to the paint policy. The argument is a `NearestPaint` or an `OrderedDither` (but not an `ErrorDiffusion`, which is only used with images). By default, a new `OrderedDither` has an order of 6, which means it synthesizes 65 (2 to the sixth, plus 1) intermediate color values between each pair of neighboring colors in the palette. You can set the order by sending an `order:` message to the `OrderedDither` class to create an instance; the argument is the desired order number.

```
| gc |
gc := (ExamplesBrowser prepareScratchWindow) graphicsContext.

gc paintPolicy: (PaintPolicy new imageRenderer: OrderedDither new).
                                                                    "V2 Step 1"
gc paintPolicy paintRenderer: NearestPaint new.                    "V2 Step 2"
0 to: 60 by: 4 do: [:i |
  0 to: 60 by: 4 do: [:j |
    gc paint: (ColorValue red: i/60 green: j/60 blue: 0).
    gc displayRectangle: (i@j+(10@10) extent: 4@4)].

gc paintPolicy paintRenderer: (OrderedDither order: 1).            "V2 Step 2"
0 to: 60 by: 4 do: [:i |
  0 to: 60 by: 4 do: [:j |
    gc paint: (ColorValue red: i/60 green: j/60 blue: 0).
    gc displayRectangle: (i@j+(80@10) extent: 4@4)].

gc paintPolicy paintRenderer: (OrderedDither order: 6).            "V2 Step 2"
0 to: 60 by: 4 do: [:i |
  0 to: 60 by: 4 do: [:j |
    gc paint: (ColorValue red: i/60 green: j/60 blue: 0).
    gc displayRectangle: (i@j+(150@10) extent: 4@4)].
```



Chapter 32

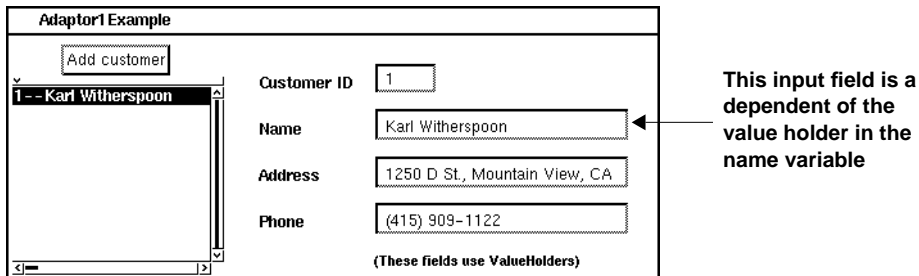
Adapting Domain Models to Widgets

Setting up Simple Value Models (ValueHolder)	704
Adapting Part of a Domain Model (AspectAdaptor)	706
Synchronizing Updates (Buffering)	710
Adapting a Collection (SelectionInList)	713
Adapting a Collection Element	715
Creating a Custom Adaptor (PluggableAdaptor)	717

See Also

- “Changing the Range Dynamically” on page 270

Setting up Simple Value Models (ValueHolder)



Strategy

A widget that presents data (such as an input field) relies on an auxiliary object called a *value model* to manage the data it presents. That is, instead of holding onto the data directly, a data widget delegates this task to its value model. Thus, when a data widget accepts input from a user, it sends this data to its value model for storage. When a data widget needs to update its display, it asks its value model for the data to be displayed.

A value model provides a uniform set of messages for accessing the data to be presented. This enables all data widgets to store and refresh their data in a standard way—by using a value message to get the data from the value model and a value: message to send the data to the value model for storage. Other objects, such as the application model, can also send these messages to a value model to obtain or change a widget's data programmatically.

A data widget is a *dependent* of its value model in that the widget depends on its value model to notify it when the relevant data has changed. The widget responds to such notification by asking the value model for the new data and displaying it. This keeps the widget's display synchronized with changes made programmatically to the data.

A ValueHolder is the most basic type of value model. As its name implies, a value holder holds onto the relevant data in an instance variable. A value holder is most useful for widgets that accept temporary pieces of information that the interface must hold onto until they can be further processed.

The basic step shows a common way to create a value holder. The variants demonstrate special-purpose ways.

Basic Step

Online example: Adaptor1Example

- Send an `asValue` message to the data object that is to be contained (in the example, the number 0 is asked to return a value holder containing itself).

```
initializeID
  accountID := 0 asValue.                                "Basic Step"
  accountID onChangeSend: #changedID to: self.
```

Variants

V1. Creating a Value Holder on an Empty String

Online example: Adaptor1Example

- Send a `newString` message to the `ValueHolder` class. This is equivalent to the expression `String new asValue`, and the choice of which to use is a matter of personal preference.

```
initializeName
  name := ValueHolder newString.                        "V1 Step"
  name onChangeSend: #changedName to: self.
```

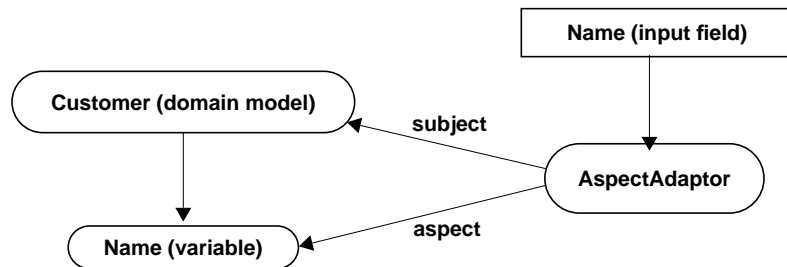
V2. Creating a Value Holder on a Boolean Value (False)

- Send a `newBoolean` message to the `ValueHolder` class. This is equivalent to the expression `false asValue`.

V3. Creating a Value Holder on a Decimal Number (0.0)

- Send a `newFraction` message to the `ValueHolder` class. This is equivalent to the expression `0.0 asValue`.

Adapting Part of a Domain Model (AspectAdaptor)



Strategy

Data widgets are commonly used for presenting data that is held by some object in the application, such as a domain model. In such cases, the appropriate value model is an `AspectAdaptor`, which is conceptually a pointer to the remote data. An aspect adaptor has a *subject*, which is the relevant domain model, and an *aspect*, which is the name of the instance variable that holds the relevant data. An aspect adaptor translates the value and value: messages it receives into accessor and mutator messages that are understood by its subject.

Subject channel: You can set up an aspect adaptor so that it obtains its subject from a value holder, called a *subject channel*. A subject channel provides indirect access to the subject, allowing you to programmatically introduce a new subject for an existing aspect adaptor. This is useful when you want particular data widget to display data held by successive domain model instances. The basic steps show how to create an `AspectAdaptor` with a subject channel. Variant 1 shows the slight modification required to address a subject directly.

Unusual accessors: By default, an aspect adaptor assumes that the domain model has accessing messages for getting and setting the value of the variable, and that those messages can be derived from the variable's name—in the example, the domain model is a `Customer1Example`, and it provides `name` and `name:` methods for accessing the value of its `name` variable. Variant 2 shows how to customize the accessing messages for a domain model that has unusual names for its accessors.

Programmatic changes to data: By default, an aspect adaptor notices programmatic changes to the data only upon receiving a value: message. However, a domain model normally changes its data without sending any messages to the aspect adaptor. Variant 3 shows how to arrange for such changes to be noticed by the aspect adaptor (and hence its dependents).

Basic Steps

Online example: Adaptor2Example

This example shows how to set up an aspect adaptor with a subject channel—a value holder from which the aspect adaptor will obtain its subject.

1. In an initialize method in the application model, initialize an instance variable (selectedCustomer) with a value holder that holds the domain model (a Customer1Example).
2. In an aspect method (accountID), send a subjectChannel: message to the AspectAdaptor class. The argument is the value holder you created in step 1.
3. Tell the aspect adaptor which aspect of the domain model to monitor by sending a forAspect: message to the adaptor. The argument is a Symbol, typically the name of the desired instance variable (accountID) in the domain model.

initialize

```
customers := SelectionInList new.
customers selectionIndexHolder
    onChangeSend: #changedCustomer to self.
```

```
selectedCustomer := Customer1Example new asValue      "Basic Step 1"
```

accountID

```
| adaptor |
adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
```

"Basic Step 2"

```
adaptor forAspect: #accountID.
```

"Basic Step 3"

```
adaptor onChangeSend: #redisplayList to: self.
^adaptor
```

Variants

V1. Addressing the Subject Directly

- Send a `subject: message` to the `AspectAdaptor` class (instead of a `subjectChannel: message`). The argument is a domain model (instead of a value holder containing the domain model).

V2. Accommodating Unusual Accessors

Use these steps when a domain model provides accessor methods whose names are different from the instance variable they access—for example, when the aspect is an instance variable called `income`, and its accessors are `getIncome` and `putIncome`.

Online example: `Adaptor2Example`

- After creating the adaptor, send an `accessWith:assignWith: message` to it. The first argument is the name of the domain model's method that accesses the desired value. The second argument is the name of the method that assigns a new value. (In the example, the message names are `address` and `address:`, and they access an instance variable named `address`, so the same effect can be achieved via `forAspect: #address`).

```
address
| adaptor |
adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
^adaptor                                "V2 Step"
    accessWith: #address
    assignWith: #address:
```

V3. Monitoring Programmatic Changes

Online example: `Adaptor2Example` and `Customer1Example`

1. Send a `subjectSendsUpdates: message` to the adaptor with the argument is `true`. This causes the adaptor to register itself as a dependent of the subject (in the example, a `Customer1Example`).
2. In the domain model class (`Customer1Example`), edit every method that alters the data value directly (that is, without

going through the adaptor), so that it sends a changed: message to self. The argument is the aspect that has been changed (#phoneNumber). This causes the dependent adaptor to be notified when the domain model makes the relevant change.

phoneNumber

```
| adaptor |
adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
adaptor forAspect: #phoneNumber.

adaptor subjectSendsUpdates: true.                                "V3 Step 1"

^adaptor
```

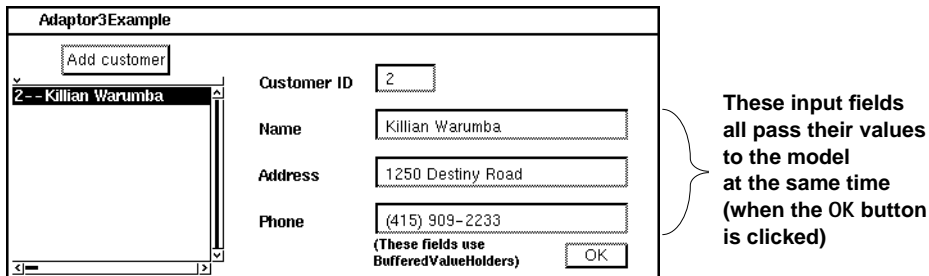
formatUSPhoneNumber

```
| rawPhone rawSize areaCode prefix suffix separator |
rawPhone := self phoneNumber select: [ :ch | ch isDigit].
rawSize := rawPhone size.
areaCode := ".
prefix := ".
suffix := ".
separator := '-'.

rawSize > 0 ifTrue: [areaCode :=
    rawPhone copyFrom: 1 to: (3 min: rawSize)].
rawSize > 3 ifTrue: [prefix :=
    separator,
    (rawPhone copyFrom: (4 min: rawSize) to: (6 min: rawSize))].
rawSize > 6 ifTrue: [suffix :=
    separator,
    (rawPhone copyFrom: (7 min: rawSize) to: (rawSize min: 10))].

self phoneNumber: areaCode, prefix, suffix.
self changed: #phoneNumber.                                "V3 Step 2"
```

Synchronizing Updates (Buffering)



Strategy

Frequently, it is useful to delay updating a particular widget's value until other widgets in the same series are ready to be updated. This is especially true in applications that make use of a database, because a row is updated only after all changes to the fields in that row have been made. Using `BufferedValueHolders` enables you to arrange a *trigger channel* that is monitored by all of the widgets in the series.

The trigger channel is a value holder that contains `true` or `false`—putting `true` in the trigger channel causes all of the dependent adaptors to update the model. Putting `false` in the trigger channel discards the buffered values, canceling the update.

`Adaptor3Example` provides an `OK` button that you press after you enter customer information in the input fields. When you press the `OK` button, the values are assigned to the selected customer.

The basic steps show how to connect a series of buffered value holders to arrange for a simultaneous update. The variant shows how to discard the buffered values. In the example, this is done when a new customer is selected.

Basic Steps

Online example: `Adaptor3Example`

1. In the application model, create an instance variable (in the example, `updateTrigger`) to contain the `true/false` value that triggers updates.
2. Create an accessing method for the variable.

```

updateTrigger                                     "Basic Step 2"
  ^updateTrigger

```

3. Initialize the variable to a value holder containing false.

```

initialize
  customers := SelectionInList new.
  customers selectionIndexHolder onChangeSend: #changedCustomer to:
  self.

  selectedCustomer := Customer1Example new asValue.

  updateTrigger := false asValue.                                     "Basic Step 3"

```

4. For each widget in the series, place the widget's value model in a BufferedValueHolder by sending a subject:triggerChannel: message to the BufferedValueHolder class. The first argument is the widget's value model (in the example, an AspectAdaptor). The second argument is the trigger channel (updateTrigger). This is typically done when the widget's value model is initialized. Note that the buffered value holder does not replace the widget's value model—rather, it contains that value model.

```

accountID
  | adaptor bufferedAdaptor |
  adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
  adaptor forAspect: #accountID.

  bufferedAdaptor := BufferedValueHolder                                     "Basic Step 4"
    subject: adaptor
    triggerChannel: self updateTrigger.

  ^bufferedAdaptor

```

5. Provide a button, a menu command, or other device with which the user can indicate that the series of values have all been edited as much as necessary (in the example, completion is indicated using an OK button that triggers an accept action).

6. In the action method (accept), send a value: message to the trigger channel (updateTrigger). The argument is true.

```

accept
  self updateTrigger value: true.                                     "Basic Step 6"

  self redisplayList.

```

Variant

Discarding the Buffered Values

Online example: Adaptor3Example

- Send a value: message to the trigger channel (updateChannel). The argument is false. This is typically done after confirming that the user wants to abandon the edited data.

```

changedCustomer
| chosenCustomer selector |
  chosenCustomer := self customers selection.

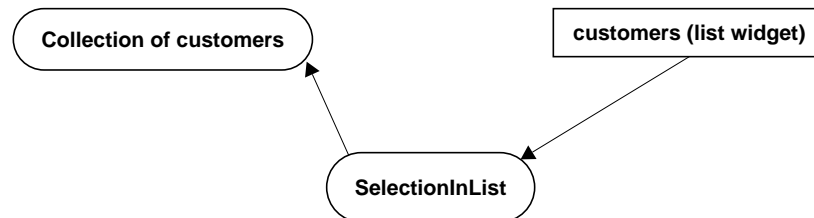
  chosenCustomer isNil
    ifTrue: [
      self selectedCustomer value: Customer1Example new.
      selector := #disable]
    ifFalse: [
      self selectedCustomer value: chosenCustomer.
      selector := #enable].

  "Discard changes that were not OK'd."
  self updateTrigger value: false.                                     "Variant Step"

  "Enable/disable selection-sensitive widgets."
  #( #accountID #name #address #phoneNumber #ok)
  do: [ :componentName
    (self builder componentAt: componentName)
    perform: selector].

```

Adapting a Collection (SelectionInList)



Strategy

A list or notebook widget is designed to work with a SelectionInList, which contains a value holder for holding the collection to be displayed. When the domain model supplies a simple collection, you can set up a SelectionInList to adapt to it, as shown in the basic step.

Basic Step

Online examples: Adaptor4Example and Customer2Example

In this example, the application model is Adaptor4Example, and the domain model is a Customer2Example, which holds an OrderedCollection of customers.

- Put the domain model's collection in a SelectionInList by sending an `adapt:aspect:list:selection:` message to the SelectionInList class. The `adapt:` argument is the domain model (in the example, `collectionModel`). The `aspect:` argument is typically the name of the domain model's collection variable. The `list:` argument is the name of the domain model's method that returns the collection. The `selection:` argument is the name of the domain model's method that sets the selection in the collection.

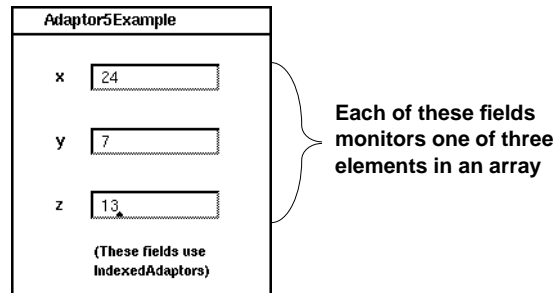
initialize

```

collectionModel := Customer2Example new.
customers := SelectionInList                                "Basic Step"
  adapt: collectionModel
  aspect: #customers
  list: #customers
  selection: #selectedCustomer:.
  
```

```
customers selectionIndexHolder  
  onChangeSend: #changedCustomer to: self.  
  
selectedCustomer := Customer1Example new asValue.
```

Adapting a Collection Element



Strategy

Sometimes a widget is used to display a single element in a collection. This situation arises when an array or other collection is used to gather a set of related attributes that would normally be separate instance variables in a domain model.

In the example, the domain model has a vector that consists of an array of three numbers, representing the *x*, *y*, and *z* axes. If the vector were an instance of a hypothetical *Vector* class and could respond to vectorlike accessing messages, you could use *AspectAdaptors* to hold its three axis values. Because the vector is a simple array that cannot respond to such messages, you must use *IndexedAdaptors*.

An *IndexedAdaptor* has a subject (the collection) or subject channel (when the collection is in a value holder) and an index number (the position of the desired element in the collection).

Basic Steps

Online example: *Adaptor5Example*

1. Send a `subjectChannel:` message to the *IndexedAdaptor* class, with the value holder containing the collection as the argument. If the collection is not contained in a value holder, send a `subject:` message instead, with the collection as the argument.
2. Send a `forIndex:` message to the adaptor. The argument is the position of the desired element in the collection.

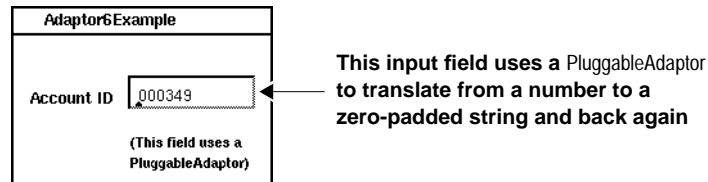
xAxis

| adaptor |

adaptor := IndexedAdaptor subjectChannel: self vector. "Basic Step 1"

^adaptor forIndex: 1 "Basic Step 2"

Creating a Custom Adaptor (PluggableAdaptor)



Strategy

Occasionally it is convenient to use a custom adaptor that performs a block of actions each time its value is accessed or changed. A `PluggableAdaptor` provides this flexibility. In the example, a `PluggableAdaptor` is used to translate an integer such as 342 into a string containing prefixed zeroes ('000342'), saving the user the trouble of entering the leading zeroes.

A `PluggableAdaptor` takes three blocks, which enable it to perform custom actions at three junctures in the flow of communications between the widget and the domain model:

- The `getBlock`: controls what happens when a value is fetched from the model by the widget. In the example, the block translates the model's account number into a zero-padded string.
- The `putBlock`: controls what happens when a value is sent to the model by the widget. In the example, the zero-padded string is converted back into a number.
- The `updateBlock`: controls when the widget updates itself based on an update message sent by the model. When the widget is the only source of changes to the data value, this block can simply return false. When the data value can be changed by other objects, the block performs a test to determine whether the widget should refetch the data value. Typically this test uses the update block's second argument, the `aspect`, to find out whether the aspect that it cares about has changed.

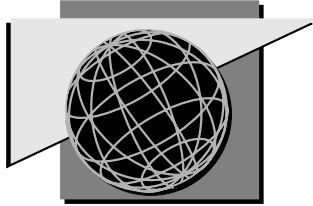
Basic Steps

Online example: Adaptor6Example

1. Create the custom adaptor by sending an `on:` message to the `PluggableAdaptor` class. The argument is the domain model.
2. Send a `getBlock:putBlock:updateBlock:` message to the adaptor. The first block takes one argument: the domain model. The second block takes two arguments: the model and the value to be assigned. The third block takes three arguments: the model, the aspect of the model that was changed, and a parameter that corresponds to the argument of a `changed:with:` message, when applicable.

initialize

```
accountID := 1.
paddedID := PluggableAdaptor on: self.           "Basic Step 1"
paddedID                                       "Basic Step 2"
  getBlock: [ :model |
    | paddedString |
    paddedString := model accountID printString.
    6 - paddedString size
      timesRepeat: [paddedString := '0', paddedString].
    paddedString]
  putBlock: [ :model :value |
    model accountID: value asNumber]
  updateBlock: [ :model :aspect :parameter | false]
```



Index

Symbols

- 431, 518
\$ 530
& 38
* 432
** 436
+ 431
, 510, 545
/ 432
// 432
:= 4
< 439
<= 439
<Control>-click xvii

' 532

A

abbreviating a string 551
abs message 450
absolute number 450
accelerator
 See also shortcut key
access date of a file 596
accessing method 10
accessors
 adapting 706
accessWith:assignWith: message 708
action
 overriding an inherited 304
action button
 adding 164
 uses for 164
action method 47
Action property 48
active window, accessing 99
adapt:aspect:list:selection: message 713
adaptor
 aspect of a model 706
 buffered 710
 custom 717
 on a collection 713
 on a collection element 715
 value holder 704

- Adaptor1Example 705
- Adaptor2Example 707, 708
- Adaptor3Example 710, 712
- Adaptor4Example 713
- Adaptor5Example 715
- Adaptor6Example 718
- add command
 - in System Browser 29, 31
- add: message 497
- add:before: message 498
- add:beforeIndex: message 498
- addAll: message 498
- addAll:beforeIndex: message 499
- addAllFirst: message 499
- addDays: message 471
- addFirst: message 498
- adding
 - dates 471
 - elements to a collection 497
 - numbers 431
 - times 482
- addItemLabel:value: message 245
- addTime: message 482
- after: message 513
- Align command 65
- Align->Distribute command 66
- aligning
 - text 561
 - text in a field 124
 - widgets 65
- allBold message 574
- and: message 38
- animating
 - a graphic image 675
- anyButtonPressed message 414
- appending
 - text to a file 599
- appendStream message 599
- application
 - closing windows 101
 - designing 42
 - grouping its classes 29
 - inheriting capabilities 302
 - integrating graphics 652
 - nesting 305
 - starting 82
- application model
 - and custom controller 399
 - combined with domain model 46
 - connecting to interface 47
 - description 43
- ApplicationDialogController 407
- ApplicationModel 302
 - creating a subclass 27
- ApplicationWindow 82
- applyColorDrop: message 360
- applyColorEnter: message 352
- applyColorExit: message 352, 353
- applyColorOver: message 352
- applyMoreColorEnter: message 354
- applyMoreColorExit: message 354
- applyMoreColorOver: message 354
- arc
 - displaying 640
- arcCos message 443
- arcSin message 442
- arcTan message 443
- Arrange->Align command 65
- Arrange->Bring To Front command 74
- Arrange->Equalize command 56
- array
 - creating 492
 - description 490
 - expanding 499
 - literal 493
 - removing an element 504
 - See also collection
- as geometric object 637

-
- asArray message 522
 - asCharacter message 451
 - asComposedText message 556, 569, 607
 - asDays message 465
 - asDouble message 449
 - asFilename message 592, 598
 - asFiller message 637
 - asFixedPoint: message 428, 449
 - asFloat message 449
 - asImage message 660
 - asList message 522
 - asLowercase message 537, 571
 - asPattern message 586, 692
 - aspect adaptor 122, 706
 - Aspect property 48
 - aspectAt:put: message 299
 - asPoint message 451
 - asRational message 449
 - asRetainedMedium message 663, 666
 - asSeconds message 466, 480
 - asSet message 523
 - assigning to a variable 4
 - Association 490
 - association
 - removing from dictionary 503
 - associationAt: message 17
 - asString message 595
 - asStroker message 632
 - asText message 557
 - asUppercase message 537, 571
 - asValue message 705
 - at: message 511
 - at:ifAbsent: message 512
 - at:put: message 497, 505
 - atAllPut: message 506
 - atEnd message 618
 - atNameKey
 - message 261
 - atPoint: message 665
 - atPoint:put: message 665
- B**
- background: message 95
 - backgroundColor: message 76, 257
 - base
 - of a number 429
 - baseline
 - in text 580
 - baseline: message 577
 - before: message 513
 - beginSubMenuLabeled: message 231
 - beInvisible message 70
 - beMaster message 105
 - beOff message 255
 - beOn message 255
 - bePartner message 106
 - beSlave message 105
 - between:and: message 440
 - beTwoDimensional message 274
 - beVisible message 70
 - Bezier curve 634
 - binary file
 - See also BOSS
 - binary message 6
 - BinaryObjectStorage 614
 - binaryReaderBlockForVersion:format:
 - method 624
 - binaryRepresentationVersion
 - method 624
 - binding
 - in a notebook 322
 - blanking a subcanvas 312
 - blueButton 412
 - blueButtonPressed message 413
 - bold emphasis 572
 - boolean
 - in a field 126

Border property 67
bordering a widget 67
BOSS
 appending 615
 converting old data 624
 custom representations 626
 omitting source code 622
 random access 619
 reading a file 617
 sequential access 618
 skipping the initial scan 617
 storing a class 621
 storing a collection 615
 storing objects 614
bounded widget 58
boundingBox:startAngle:sweepAngle:
 message 641
bounds message 59
box
 around widgets 156
branch
 creating 33
browseAllCallsOn: message 17
BufferedValueHolder 710
buffering
 model updates 710
builder
 window 85
bulletin boards xx
button
 default 165
 graphic label 167
 highlighting 168
 See also action button
 See also check box
 See also menu button
 See also radio button
ButtonExample 160, 162, 164, 167
buttons, mouse, see mouse buttons

bypassing a dependency 78
byte array
 creating an image 660
 in a field 126

C
caching
 a graphic image 674
Can Tab property 74
canBeWritten message 599, 611
canvas
 opening 82
 polling vs. event-driven 392
 tab order 74
cap style of a line 645
capacity message 495
capturing
 a graphic image 659
cascading messages 9
case
 in a string search 544
 in text 571
 of a character 535
 of a string 537
category
 of classes 29
 of messages 31
ceiling message 434
center:radius: message 640
centered message 561
century
 seconds in 480
chaining messages 7
change validation callback 132
Change Validation property 133
changed: message 384, 709
changed:with: message 382, 653, 718
changeRequest message 102

- character
 - categories of 534
 - comparing 540
 - composed 531
 - counting in text 564
 - creating 451, 530
 - line ends 553
 - nondisplaying 530
- CharacterAttributes 576, 578
- characterAttributes: message 577, 579, 583
- check box
 - adding 162
 - in menu 254
 - uses for 162
- choose:fromList:values:lines:cancel: message 289
- choose:labels:values:default: message 282
- circle 640
 - See also ellipse
- class
 - BOSS representation 626
 - category 29
 - changing BOSSed instances 624
 - creating 26
 - grouping 29
 - in a BOSS file 621
 - moving to different category 30
 - naming conventions 26
- class category
 - adding 29
 - definition 29
 - removing 29
 - renaming 30
 - reordering 30
- class instance variable
 - declaring 14
 - removing 17
 - usage 11
- class var refs command 16
- class variable
 - declaring 14
 - removing 16
 - undeclared 19
 - usage 11
- clear all command 88
- clearAll message 193
- click xvii
 - See also mouse
- client: message 312
- client:spec: message 311, 335
- clientData message 360
- clippingBounds message 676
- clock
 - elapsed seconds 481
- close message 598
- closeRequest message 101
- closing a window 101
- code
 - timing 484
- collapse message 103
- collapsing a window 103
- collect: message 37, 526
- collection
 - adapting 713
 - adapting an element 715
 - adding elements 497
 - capacity 495
 - combining 510
 - comparing 517
 - copying elements 508
 - counting occurrences 513
 - creating 491
 - finding an element 36
 - finding elements 511
 - in a BOSS file 615
 - inserting an element 498

- iterating 35
- looping 524
- removing elements 500
- replacing elements 505
- See also array
- See also dictionary
- See also list
- See also list widget
- See also set
- selecting elements 36
- size 495
- sorting 519
- subtracting a subset 518
- testing for emptiness 495
- transforming 37
- type conversion 522
- types 490
- collection class
 - creating 27
- color
 - applying 694
 - box widget 157
 - creating 686
 - dithering 698
 - in a menu 257
 - label widget 116
 - of a geometric object 649
 - palette 696
 - predefined 686
 - rendering policy 698
- color emphasis 585
- color: message 257
- ColorDDEExample 344, 351, 354, 356, 360, 367
- colorDrag: message 345, 360
- ColorExample 76
- coloring
 - a graphic image 664
 - a widget 75
 - a window 94
 - text 585
- colorLayerEnter: message 356
- colorLayerExit: message 356
- colorLayerOver: message 356
- colorPalette message 697
- colors message 696
- ColorValue 686
- colorWantToDrag: message 345
- column selecting
 - dataset 209
- column widths
 - dataset 207
 - table 219
- combining
 - collections 510
 - graphic images 672
 - strings 545
- combo box
 - creating 146
- ComboBoxExample 146
- ComboConversionExample 148
- comma
 - for combining collections 510
 - for combining strings 545
- comparing
 - characters 540
 - collections 517
 - dates 473
 - files or directories 605
 - numbers 438
 - strings 540
 - texts 568
- completeContentsOfArea: message 666
- component
 - See also widget
- componentAt: message 55
- composed character 531
- composeDiacritical: message 531

-
- ComposedText 113, 607
 - See also text
 - composite view
 - connecting to controller 386
 - compositionWidth: message 559, 570
 - compressing
 - a string 551
 - concatenating
 - See also combining
 - conditional expression 33
 - conditional looping 35
 - conditional test 38
 - ConfigurableDropSource 372
 - ConfigurableDropTarget 341, 348
 - confirm: message 280
 - confirmer dialog 280
 - connecting
 - controller to model 399
 - controller to view 400
 - field to field 143
 - view to controller 385
 - constantNames message 687
 - constants
 - numeric 458
 - constraining window size 87
 - construct: message 593
 - contents
 - of a file or directory 597
 - contents message 617
 - contentsOfEntireFile message 597, 605
 - contextApplication: message 346
 - contextWidget: message 345
 - contextWindow: message 345
 - continuousAccept: message 137
 - contractTo: message 551
 - control loop 405
 - controlActivity method 406, 412, 413, 416, 417
 - controlInitialize method 407
 - Controller 393, 405
 - controller
 - activity 405
 - connecting to model 399
 - connecting to view 400
 - connecting view 385
 - consuming events 406
 - creating 395
 - cursor control 408
 - cursor location 419
 - dual-architecture 393
 - event methods 405
 - handing off control 411
 - initializing 401, 407
 - keyboard input 416
 - loading event-driven code 393
 - mouse input 412
 - polling vs. event-driven 392
 - pop-up menu 409
 - routing menu messages 410
 - taking control 402
 - terminating 408
 - yielding control 403
 - controller: message 386, 401
 - ControllerWithMenu 409
 - controlLoop method 408
 - ControlManager 392
 - controlPoints: message 634
 - controlTerminate method 408
 - conventions
 - screen xvi
 - typographic xiv-xvi
 - convertForGraphicsDevice:renderedBy: message 698
 - converting
 - collection type 522
 - number types 449
 - old BOSS data 624
 - convertToPalette

Index

- message 697
- convertToPalette:renderedBy:
 - message 700
- copy:from:in:rule: message 673
- copyArea:from:sourceOffset:destination
 - Offset: message 667
- copyEmpty message 671
- copyFrom:to: message 509, 547, 569
- copying
 - a file or directory 603
 - elements in a collection 508
 - text 569
- copyReplaceAll:with: message 550
- copyReplaceFrom:to:with: message 549
- copyTo: message 603
- copyUpTo: message 547
- copyWith: message 499
- copyWithout: message 504
- corner: message 638
- cos message 442
- coverage
 - as a paint 690
- coverage: message 690
- CoveragePalette 680
- coveragePalette message 697
- CoverageValue 690
- CR
 - as line end 553
- CUARadioButtonView 401
- current
 - time 478
- current window
 - See also active window
- cursor
 - changing 681
 - changing via controller 408
 - creating 678
 - predefined 678
 - sensing location 419

- cursorPoint message 420
- cursorPointFor: message 420
- curve
 - Bezier 634
 - spline 634
- custom
 - adaptor 717
 - controller 395
 - dialog 296
 - font 576
 - view 376
- Customer1Example 20, 24, 26, 708
- Customer2Example 713
- CustomerExample 25
- CustomView1Example 376, 379, 389, 395, 652, 681
- CustomView2Example 396

D

- damage
 - rectangle 387
 - window 93
- data
 - as fields in a file 609
 - formatting 129
- data types 125
- dataset
 - adding a row 210
 - connecting data 212
- dataset columns
 - labels 213
 - order 207
 - scrolling 207
 - selecting 209
 - widths 207
- dataset widget
 - adding 204
 - label colors 214

- row marker 211
- selecting columns 209
- splitting column labels 213
- Dataset1Example 205
- Dataset2Example 210
- Dataset3Example 212
- Dataset4Example 213
- date
 - adding 471
 - comparing 473
 - creating 462
 - day information 465
 - days left in year 469
 - days since 1901 465
 - formatting 475
 - in a field 125
 - leap year 469
 - month information 467
 - previous day of week 472
 - seconds since 1901 466
 - subtracting 471
 - year information 469
- dateAndTimeNow message 483
- dates message 596
- day
 - hours in 481
 - in a date 465
 - of year 468
 - seconds in 480
- dayOfMonth message 465
- days
 - in each month 467
- daysInMonth message 467
- daysInYear message 469
- daysLeftInYear message 469
- declaring a variable 13
- default
 - button 165
 - font 575, 588
 - message 23
 - palette 697
- defaultController method 401
- defaultControllerClass method 385, 400
- defaultDirectory message 594
- degreesToRadians message 450
- delete message 602
- deleting
 - a file or directory 602
- denominator message 428
- dependency
 - adding and removing 78
 - bypassing 78
- DependencyExample 79
- dependent window
 - See also slave window
- desiresFocus message 395
- detect: message 36
- detect:ifNone: message 515
- diacritical mark 531
- dialog
 - custom 296
 - file name 286
 - list 289
 - multiple-choice 282
 - textual 284
 - warning 278
 - yes-no 280
- DialogExample 278, 283, 297
- dictionary
 - description 490
 - removing an association 503
 - See also collection
- dimensions, accessing window 88
- directory
 - characteristics 594
 - comparing 605
 - contents 597
 - copying or moving 603

- creating 592
- dates 596
- default 594
- deleting 602
- distinguishing from file 596
- parent 595
- directoryContents message 597, 606
- disable message 72, 248
- disabling
 - menu items 248
 - widgets 72
- display message 93
- display surface
 - capturing as an image 660
- displayArcBoundedBy:startAngle:sweep
Angle: message 641
- displayBox: message 88
- displayDotOfDiameter:at: message 641
- displaying
 - a graphic image 662
 - a line 631
 - a point 630
 - a polygon 637
 - in a view 380
 - text 558
- displayLineFrom:to: message 630, 631
- displayOn: message 558, 632
- displayOn: method 380
- displayOn:at: message 662
- displayPolygon: message 637
- displayPolyLine: message 632
- displayWedgeBoundedBy:startAngle:sweepAngle: message 641
- Distribute command 66
- distributing widgets 66
- dithering color 698
- divider 154
 - in a menu 250
- dividing
 - numbers 432
- division 428
- divisor, greatest common 453
- do: message 35, 524
- documentation, see VisualWorks documentation
- doDragDrop message 346
- dollar sign
 - in creating a character 530
- domain model
 - adapting an aspect 706
 - combined with application model 46
 - connecting to view 378
 - creating 45
 - description 45
 - updating view 382
- Double
 - See also floating-point
- double-click xvii
- double-clicking 392
- drag and drop 339-373
 - adding 343-349
 - defining custom pointers 371
 - dragging multiple selections 347
 - drag-ok method 343
 - drag-start method 343
 - dropping data on a list item 361
 - effect symbol. See effect symbol
 - examining dragged data 365
 - framework classes 341
 - implementation example 341
 - providing visual feedback 350
 - responding to a drop 359
 - using modifier keys 366
 - See also drop source
 - See also drop target
- Drag OK property 343
- Drag Start property 343
- DragDropContext 341, 351

-
- DragDropData 341, 345
 - DragDropManager 341, 346, 348, 359
 - drop method 359
 - Drop property 348, 359
 - drop source 340, 343
 - adding 343–347
 - setting up 340
 - drop target 340, 348
 - adding 348–349
 - button 354
 - changing button label 354
 - examining dragged data 365
 - list item 356, 361
 - messages 350
 - providing visual feedback 350–358, 371
 - responding to a drop 359
 - setting up 340
 - tracking a specific list item 356
 - using modifier keys 366
 - dropFinalVowels message 551
 - DropSource 341, 346
 - dummy
 - See also placeholder
- E**
- edit all command 30
 - in System Browser 32
 - edit message 601
 - editing
 - a list 187
 - editor
 - opening on a file 601
 - editor widget
 - accessing selection 174
 - adding 172
 - connecting to list 200
 - connecting to notebook 336
 - Editor1Example 173, 174, 200
 - Editor2Example 93, 310
 - editors
 - image 659
 - effect symbol 351, 360
 - adding 372
 - creating custom symbol 373
 - defining 371
 - types of 351
 - See also target emphasis
 - electronic bulletin boards xx
 - electronic mail xx
 - elements
 - adapting 715
 - adding to collection 497
 - ellipse 640
 - grouping widgets 158
 - Ellipse property 158
 - EllipticalArc 641
 - embedded canvas
 - See also subcanvas
 - emphasizeAllWith: message 573, 586
 - emphasizeFrom:to:with: message 572, 578, 585
 - enable message 72, 248
 - endSubMenu message 231
 - enterEvent: method 406
 - Entry property 348, 350
 - enumerating
 - See also looping
 - enumerating a collection 35
 - equality
 - of numbers 438
 - Equalize command 56
 - error
 - compilation 19
 - ErrorDiffusion 699
 - even message 445
 - event

Index

- consumption 406
- methods 405
- user-input 392
- window 106
- Event Driven property 394
- event-driven controller
 - See controller
- events.st 393
- exists message 594
- Exit property 348, 350
- exit validation callback 132
- Exit Validation property 133
- exitEvent: method 406
- exp message 444
- expand message 103
- expandedMenu message 140
- expanding
 - graphic images 668
 - windows 103
- exponential
 - of a number 444
- extent: message 638
- extent:depth:palette: message 660
- extent:depth:palette:bits:pad:
 - message 660
- F**
- factorial 453
- factoring
 - numbers 453
- family of a font 582
- family: message 582
- fax support xxi
- field
 - aligning 124
 - connecting to a field 143
 - creating 122
 - dialog 284
 - filtering and validating 132
 - formatting numbers 129
 - highlighting 150
 - in a text file 609
 - insertion point 150
 - menu 139
 - menu of entries 146
 - read-only 124
 - size restriction 124
 - type restriction 125
 - widget, connecting to a slider 267
- FieldConnectionExample 143, 144
- FieldMenuExample 139
- FieldSelectionExample 150
- FieldTypeExample 126, 131
- FieldValidation1Example 136
- FieldValidation2Example 137
- FieldValidInputExample 133, 134
- file
 - binary
 - See also BOSS
 - characteristics 594
 - comparing 605
 - contents 597
 - copying or moving 603
 - creating 592
 - dates 596
 - deleting 602
 - distinguishing from directory 596
 - fields of textual data 609
 - opening an editor 601
 - parts of name 595
 - permissions 611
 - printing 607
 - size 594
 - storing text 598
- file name
 - dialog 286
- file out

- vs. a BOSS file 621
 - Filename 592
 - fileSize message 594
 - filling wrapper for geometrics 637
 - fill-in-the-blank dialog 284
 - filtering
 - field input 132
 - finalization
 - upon window closing 102
 - finding
 - elements in a collection 511
 - See also searching
 - findString:ignoreCase:useWildcards:
 - message 544
 - findString:startingAt:ifAbsent:
 - message 543
 - findString:startingAt:ignoreCase:useWildcards: message 566
 - first message 514
 - firstDayOfMonth message 468
 - firstIndent: message 562
 - fixed command 57
 - fixed size command 87
 - fixed-point number
 - creating 428
 - definition 426
 - in a field 126
 - fixedWidth emphasis 582
 - floating-point number
 - creating 427
 - definition 426
 - floor message 434
 - flopping an image 669
 - follow:while:on: message 675
 - font
 - changing widget's 68
 - creating 576
 - default 588
 - default for platform 575
 - family 582
 - in a text 572
 - label 116
 - menu 587
 - name 582
 - platform 589
 - size 578
 - font mixing 116
 - Font1Example 68
 - Font2Example 589
 - FontDescription 576, 583
 - fonts xiv-xvi
 - forAspect: message 707
 - foregroundColor: message 76
 - forgetInterval: message 619
 - forIndex: message 715
 - Format property 126, 129
 - formatting
 - a date 475
 - displayed data 129
 - numeric field 129
 - fraction
 - creating 428
 - definition 426
 - from:to: message 632
 - fromDays: message 464
 - fromGenerator:seededWith:
 - message 455
 - fromSeconds: message 479
 - fromUser message 659
- G**
- gcd: message 453
 - geometric
 - arc 640
 - Bezier curve 634
 - color 649
 - ellipse 640

- integrating in application 652
- line 631
- point 630
- polygon 637
- rendering color 701
- spline 634
- getBlock:putBlock:updateBlock:
 message 718
- global variable
 - removing 17
 - usage 10
- globalCursorPoint message 100, 421
- globalPoint message 421
- graphic image
 - animating 675
 - caching 674
 - capturing 659
 - coloring 664
 - converting to display surface 663
 - creating 658
 - displaying 662
 - expanding and shrinking 668
 - flopping 669
 - in menu 254
 - layering 672
 - masking 666
 - palette 696
 - rotating 670
- graphic label 111, 118
 - in a button 167
- graphics
 - See also geometric
- graphics context 558
- GraphicsAttributes 650
- GraphicsAttributesWrapper 650
- GraphicsContext 380
- graphicsContext message 631
- graying out
 - See also disabling

- greater than 440
- greatest common divisor 453
- grid
 - in lines of text 580
- grid: message 271
- gridForFont:withLead: message 580,
 583
- grouping widgets 156

H

- handlerForMouseEvent: method 403
- hardcopy
 - See also printing
- hardcopy message 565
- hasAlt message 417
- hasCtrl message 417
- hasLock message 417
- hasMeta message 417
- hasShift message 417
- head message 595
- HelpBrowser example 168
- HideExample 70, 72
- hideItem: message 247
- hiding
 - widgets 70
 - windows 104
- hierarchic menus 231
- highlighting
 - button 168
 - in a field 150
 - in a list 196
 - See also selection
- hiliteSelection: message 168
- holder
 - See also value holder
- hour
 - minutes in 481
- hours

- in day 481
- hsb color 688
- hue:saturation:brightness: message 688
- I**
- icon
 - creating 682
 - in a menu 254
 - linking to a window 683
 - predefined 682
 - window 108
- icon: message 108, 683
- iconify
 - See also collapsing
- ID property 54
- identity
 - comparing numbers 438
- IdentityDictionary 346
- ifTrue: message 33
- image
 - See also graphic image 658
- Image Editor 659
- image:mask:hotSpot:name:
 - message 680
- includes: message 513
- includesAssociation: message 513
- includesKey: message 513
- indenting
 - text 562
- IndexedAdaptor 715
- indexOf: message 512, 543
- indexOfSubCollection:startingAt:
 - message 514
- inequality
 - testing numbers 438
- inheriting
 - application capabilities 302
- initialize message 24, 123
- initialize method 48
- initializeDispatchTable method 417
- initializeMenu method 409
- Initially Disabled property 72
- Initially Invisible property 70
- input architecture 392
- input controller
 - See controller
- input field
 - See also field
- inserting
 - element in a collection 498
- insertion point
 - in a field 150
- inst var refs command 16
- install command 27
- instance
 - creating 22
 - initializing 24
 - used in drag and drop 341
- instance variable
 - declaring 14
 - removing 16
 - undeclared 19
 - usage 11
- integer
 - creating 426
 - definition 426
 - small-large boundary 459
- integrating
 - graphics in an application 652
 - view in interface 389
- interface
 - connecting to models 47
 - integrating a custom view 389
 - painting 43
 - reusing 308
 - swapping at runtime 310
- interface builder 43

interface component
 See also widget
interfaceSpecFor: message 311
interval
 iterating 34
invalidate message 174, 653
invalidateRectangle 388
invalidateRectangle: message 388
invalidateRectangle:repairNow:
 message 388
invalidating a view 387
invisible, making a widget 70
isAlphabetic message 534
isAlphaNumeric message 534
isComposed message 536
isControlActive method 403
isControlWanted method 403
isDigit message 535
isDirectory message 596
isEmpty message 495
isInteger message 445
isLetter message 534
isLowercase message 535
isSeparator message 535
isVowel message 534
isZero message 446
italic emphasis 572
iterating
 See also looping
iterating a collection 35

J

join style of a line 647
justified message 561
justify
 See also aligning

K

key: message 345
keyAtValue:ifAbsent: message 512
keyboard
 sensing input 416
keyboard activity, responding to 392
keyboard shortcut 252
KeyboardEvent 416
keyboardEvent message 417
keyboardHook: message 136
keyboardPeek message 417
keyboardPressed message 417
KeyboardProcessor 395, 411
keyboardProcessor message 411
keyPressedEvent: message 416
keyPressedEvent: method 418
keysAndValuesDo: message 526
keysDo: message 526
keyValue message 417
keyword message 5

L

label
 aligning table labels 223
 table columns 223
 table rows 223
 window 92
Label property 92
label widget
 changing at runtime 113
 color 116
 creating 110
 font 116
 graphic 111
 multi-line 110
 registry of labels 118
label: message 92, 114
labelAt:put: message 119

- labelImage: message 254
- labelString
 - message 69
- labelString: message 113, 354
- large emphasis 572, 578
- LargeNegativeInteger 426
- LargePositiveInteger 426
- last message 514
- lastIndexOf: message 512
- Launcher
 - creating 294
- layering
 - graphic images 672
- Layout->Be Bounded command 58
- Layout->Relative command 56, 60
- Layout->Unbounded command 58
- lcm: message 453
- leap message 469
- least common multiple 453
- leftFlush message 561
- length
 - of a string 539
 - of text line 559
- less than 439
- LF, as line end 553
- line
 - cap style 645
 - displaying 631
 - join style 647
 - See also divider
 - thickness 644
- line end characters 553
- line message 250
- line spacing in text 580
- LineExample 154, 156, 158
- lineGrid: message 577
- LineSegment 631
- lineWidth: message 644
- lining up
 - See also aligning
- list
 - creating 492
 - description 490
 - See also collection
- list dialog 289
- list widget
 - adding 184
 - connecting to editor 200
 - connecting two lists 198
 - controlling textual representation 185
 - editing the list 187
 - highlighting style 196
 - menu 194
 - multiple selections 189
 - selection 191
- List1Example 185, 187, 190, 191, 194, 196, 198
- List2Example 303, 306, 308, 310
- listAtCategoryNamed: message 679
- listPlatformFonts message 589
- literal
 - array 493
- ln message 444
- Locale object 129
- log message 444
- log: message 444
- logarithmic functions 444
- LogoExample 110, 111, 113, 114, 116, 118
- LookPreferences 76
- loop
 - creating 34
- looping 34-37
 - through a collection 524
- lowercase
 - See also case

M

- magnifiedBy: message 668
- mail
 - electronic xx
- makeDirectory message 593
- makeUnwritable message 611
- makeWritable message 611
- map message 104
- MappedPalette 680, 696
- marker
 - in a slider 273
- mask
 - creating 666
 - setting opacity 690
- master window 105
- master windows 292
- max size command 87
- max: message 441
- maximum
 - of two numbers 441
- maximumSize: message 87
- maxVal message 459
- Menu 410
- menu
 - as list in dialog 289
 - check box in 254
 - color 257
 - combined with field 146
 - creating 226
 - disabling an item 248
 - divider 250
 - icon 254
 - in a field 139
 - in a Launcher 294
 - in list widget 194
 - modifying at runtime 243
 - of commands 227
 - of fonts 587
 - of values 228
 - pop-up 240
 - routing messages 410
 - shortcut key 252
 - submenu 231
 - via controller 409
- menu bar
 - adding 98, 233
 - creating 233
- Menu Bar property 98, 233
- menu button
 - adding 236
- Menu Editor 226, 259
- menu item
 - label 254
- Menu property 48
- menuAt
 - message 261
- menuAt: message 247
- MenuBuilder 227
- MenuCommandExample 227, 231, 233, 238, 240, 248, 250, 252
- MenuEditorExample 260, 261
- menuHolder: message 410
- menuItemLabeled: message 246
- menuItems message 261
- MenuModifyExample 245, 246, 247, 257
- MenuSwapExample 243
- MenuValueExample 228, 229, 235, 236, 241, 254, 255
- merging graphic images 672
- Message
 - constructing 4
- message
 - binary 6
 - cascading 9
 - chaining 7
 - definition 4

- keyword 5
 - message category
 - See also protocol
 - message expression 4
 - complex 7
 - MessageSend 626
 - method
 - copying to other class 32
 - creating 18
 - grouping 31
 - instance vs. class 18
 - moving to different protocol 32
 - subdividing 18
 - millisecondClockValue message 454, 481
 - millisecondsToRun: message 484
 - min size command 87
 - min: message 441
 - minimum
 - of two numbers 441
 - minimumSize: message 87
 - minor keys in notebook 328
 - minutes
 - in hour 481
 - minVal message 459
 - model
 - See also application model
 - See also domain model
 - model: message 399, 654
 - model: method 379
 - modification date of a file 596
 - modulo division 432
 - monoMaskPalette message 680
 - month
 - days in 467
 - in a date 467
 - monthIndex message 467
 - monthName message 467
 - mouse
 - sensing input 412
 - waiting for click 421
 - mouse action, responding to 392
 - mouse buttons xvi
 - <Operate> button xvi
 - <Select> button xvi
 - <Window> button xvi
 - one-button mouse xvii
 - three-button mouse xvi
 - two-button mouse xvii
- mouse operations xvii
 - <Control>click xvii
 - <Meta>-click xvii
 - <Shift>-click xvii
 - click xvii
 - double-click xvii
- MouseEvent 419
 - MouseMovedEvent 419
 - mouseMovedEvent: method 414
 - move to command
 - in System Browser 30, 32
 - moveBy: message 63
 - MoveExample 63
 - moveTo: message 64, 91, 603
 - moving
 - a file or directory 603
 - a widget 63
 - a window 90
 - Multi Select property 189, 347
 - multi-line label 110
 - multiple, least common 453
 - multiple-choice dialog 282
 - multiplying
 - numbers 432
 - MultiSelectionInList 190, 347, 490
- N**
- name

Index

- of a font 582
- name key 261
- name: message 583
- nameKey message 261
- natural log 444
- NearestPaint 698
- negated message 448
- negative message 447
- nesting
 - applications 305
- new message 22
- new: message 492
- new:withAll: message 492
- newBoolean message 705
- newBounds: message 59
- newDay:monthNumber:year:
message 463
- newDay:year: message 463
- newFraction message 705
- newString message 705
- newWithDefaultAttributes
message 576, 579, 582
- nextIndexOf:from:to: message 512
- nextPut: message 614, 615
- nextPutAll: message 545, 598, 615
- nextPutClasses: message 621
- noButtonPressed message 415
- NoController 386
- noMenu message 142
- nonBlueButtonPressed message 415
- nontextual collection
 - displaying in list 185
- normalSelection message 197
- notational conventions xiv-xvi
- notebook widget
 - adding 316
 - binding appearance 322
 - changing the page 334
 - connecting to editor 336
 - index tabs 324
 - minor keys 328, 331
 - secondary tabs 328
 - starting page 326
 - tab selection 319
- Notebook1Example 316, 320, 326
- Notebook2Example 320, 328
- Notebook3Example 322, 324, 331
- Notebook4Example 334
- Notebook5Example 337
- now message 478
- number
 - adding 431
 - comparing 438
 - constants 458
 - creating 426
 - dividing 432
 - exponential 444
 - factoring 453
 - field formatting 129
 - in a field 125
 - logarithmic functions 444
 - maximum 441
 - minimum 441
 - multiplying 432
 - raised to a power 436
 - random 454
 - range checking 440
 - root 436
 - rounding 434
 - See also fixed-point
 - See also floating-point
 - See also fraction
 - See also integer
 - See also radix notation 429
 - See also scientific notation 429
 - sign conversion 447
 - subtracting 431
 - testing 445

trigonometric functions 442
type conversion 449
NumberPrintPolicy 131
numerator message 428

O

object
 in a field 126
 See also instance
object file
 See also BOSS
occurrencesOf: message 513
odd message 446
onChangeSend:to: message 79
online documentation, see VisualWorks
 documentation
onNew: message 614
onOld: message 615, 617
onOldNoScan: message 617
opaque message 690
open message 83
openDialogInterface: message 297
openFor:interface: message 298
openIn: message 90
opening
 canvases 83
 specs 83
 windows 82
openOnMenu: message 294
openWithExtent: message 86
openWithSpec: message 83
or: message 38
order
 of tabbing 74
order of execution 8
OrderedCollection 490
OrderedDither 698
Over property 348, 350

P

page
 in a notebook 326
paint
 applying 694
 See also color
 See also coverage
 See also pattern
paint: message 649, 694
PaintPolicy 701
paintPolicy: message 701
paintRenderer: message 701
palette
 default 697
 of an image 696
palette message 696
pane
 See also view
ParagraphEditor 403, 417
parsing order of messages 8
partner windows 106
password
 in a field 125
Pattern 585
pattern
 applying 694
 creating 692
 tile phase 693
patterned text 585
performer: message 410
permissions
 on a file or directory 611
persistence
 See also BOSS
phase
 of a tiled pattern 693
pi message 459
pixelSize: message 579

placeholder
 button action 165
platform fonts, listing 589
PluggableAdaptor 127, 717
point
 creating 451, 630
 displaying 630
pointer shape
 See effect symbol
pointer-type class
 creating 27
PointExample 614, 624
policy
 for rendering color 698
policy colors, in Color Tool 76
polling controller
 See controller
polygon
 displaying 637
PolyLine 631
Polyline 637
pool dictionary
 declaring 15
 removing 17
 usage 12
pop-up menu 409
 creating 240
position: message 619
positioning a widget 60
positive message 447
postBuildWith: method 136, 143, 168
power
 of a number 436
pref size command 86
PreferredCustomerExample 25
previous day of week 472
Print property 148
printFormat: message 475
printing

 a text file 607
 text 565
printString message 451
printStringRadix: message 452
printTextFile message 607
protocol
 adding 31
 creating 31
 private vs. public 31
 removing 31
 renaming 32
 reordering 32

Q

quotation mark
 in creating a string 532

R

radiansToDegrees message 450
radio button
 adding 160
 uses for 160
radix notation 429
random access
 in a BOSS file 619
random number
 generating 454
range
 in a slider 270
 of numbers 440
rangeStart: message 271
rangeStop: message 271
RasterOp 672
rational number 449
Read property 148
read stream 610
readAppendStream message 615

-
- readFromString: message 462, 478
 - read-only
 - fields 124
 - sliders 266
 - readStream message 617
 - receiver 4
 - receiver:selector:arguments:
 - message 626
 - reciprocal message 450
 - Rectangle 637
 - red:green:blue: message 688
 - redButton 412
 - redButtonPressed message 413
 - redisplaying a view 387
 - reducedMenu message 140
 - references
 - to a variable 16
 - reflectedInX message 669
 - reflectedInY message 669
 - refreshing a window 93
 - registering an interest 78
 - registry
 - labels 118
 - reject: message 36, 515
 - relative sizing of widget 57
 - remainder division 432
 - remove command
 - in System Browser 29, 31
 - remove: message 500
 - remove:ifAbsent: message 501
 - removeAll: message 501
 - removeAllSuchThat: message 503
 - removeFirst message 502
 - removeFirst: message 502
 - removeFrom:to: message 502
 - removeItem: message 246
 - removeKey: message 17, 503
 - removeKey:ifAbsent: message 503
 - removeLast message 502
 - removeLast: message 502
 - removing a variable 16
 - rename as command
 - in System Browser 30, 32
 - renameTo: message 604
 - rendering color 698
 - repeating a block 34
 - replaceAll:with: message 507
 - replaceFrom:to:with: message 567
 - replaceFrom:to:with:startingAt:
 - message 507
 - replaceSelectionWith: message 174
 - replacing
 - elements in a collection 505
 - part of a text 567
 - representBinaryOn: method 626
 - request: message 284
 - requestActivationFor: message 411
 - requestFileName: message 286
 - resetSelections message 174
 - resetViews message 588
 - respondsToArithmetic message 445
 - restIndent: message 562
 - retractInterestsFor: message 79
 - return
 - from a method 20
 - reuse techniques 18, 301-314
 - reusing
 - interface 308
 - reverse message 521
 - reverseDo: message 525
 - rgb color 688
 - rightFlush message 561
 - root
 - of a number 436
 - rotateByQuadrants:to: message 671
 - rotatedByQuadrants: message 670
 - rotating
 - a graphic image 670

- See also flopping
- rounded message 434
- rounding numbers 434
- roundTo: message 435
- row selector
 - dataset 211
- S**
- sameAs: message 541
- sameCharacters: message 542
- scale 428
- scanning
 - fields in a file 609
- ScheduledControllers 99
- scientific notation 429
- scope of variables 10
- Screen 100, 589
 - default palette 697
- screen capture
 - See also graphic image 659
- screen conventions xvi
- screen coordinates
 - of cursor 421
- scroll bars
 - adding and removing 96
- searching
 - a string 543
 - in a text 566
- secondary tabs in notebook 328
- seconds
 - creating time 478
 - in a date 466
 - in century 480
 - in day 480
 - in minute 480
 - since clock reset 481
- select: message 36, 515
- selectAll message 193

- selection
 - editor widget 174
 - list 191
- selection: message 327
- selectionBackgroundColor: message 76
- selectionForegroundColor: message 76
- selectionIndex: message 326
- selectionIndexes: message 193
- SelectionInList 184, 205, 317, 344, 490, 713
- SelectionInTable 218
- SelectionInTable class 216
- selections message 347
- selections: message 193
- sensing input
 - See also controller
- sensor
 - cursor location 419
- sensor message 417, 420
- separator
 - characters 535
- sequential access
 - in a BOSS file 618
- serif emphasis 572, 582
- set
 - creating 492
 - description 490
 - See also collection
- setDefaultQuery: message 576, 579, 582
- setDefaultTimeZone: message 487
- setDefaultTo: message 588
- setMarkerLength: message 273
- setting up a dependency 78
- setToEnd message 615
- setValue: message 80
- shortcut key
 - in a menu 252
- shortcutKeyCharacter: message 252

- shortening a string 551
- showDropFeedbackIn:allowScrolling:
 - message 357
- showWhile: message 681
- shrinking graphic images 668
- shrunkedBy: message 668
- sign
 - of a number 447
- sign message 447
- SimpleDialog 297
 - creating a subclass 27
- sin message 442
- size
 - of a file 594
 - of a string 539
 - of field input 124
 - of font 578
 - of text 564
- Size as Default property 165
- size message 495, 539
- Size1Example 57, 61
- Size2Example 58, 62
- Size3Example 59
- sizing
 - widgets 56
 - windows 86
- Sketch 376, 382, 395, 652
- SketchController1 376, 385, 396, 400, 403, 406, 408, 409, 413, 417, 420, 681
- SketchController2 396, 397, 403, 405, 407, 413, 418, 420
- SketchView1 376, 379, 380, 382, 385, 388, 389, 395, 400, 652, 681
- SketchView2 396
- slave window 105
- slider widget
 - adding 264
 - connecting to field 267
 - marker length 273
 - modifying range 270
 - read-only 266
 - two-dimensional 274
- Slider1Example 264, 267, 270
- Slider2Example 123, 268, 273, 274
- small emphasis 572, 578
- SmallInteger 426
- sort message 519
- SortedCollection 490
- sorting
 - a collection 519
- sortWith: message 520
- source code
 - omitting in BOSS file 622
- source: message 84
- sourceData message 360
- sourceMode: message 622
- spacing a group of widgets 66
- spacing widgets 66
- spacing, of lines in text 580
- spawn command 19
- spec
 - opening 83
- special symbols xiv-xvi
- spellAgainst: message 542
- Spline 634
- sqrt message 436
- squared message 436
- start:end:controlPoint1:controlPoint2:
 - message 635
- stencil
 - See also mask
- stream
 - closing 598
 - creating 598
 - for reading a text file 609
 - positioning 615
- strictlyPositive message 447

strikeout emphasis 572
string
 abbreviating 551
 changing its case 537
 combining 545
 comparing 540
 converting to text 556
 creating 451, 532
 getting a substring 547
 in a field 125
 length and width 539
 removing a substring 549
 replacing a substring 549
 searching 543
StringPrintPolicy 131
strokedSelection message 197
stroking wrapper for geometrics 632
style
 See also text style
style of text
 See also font
styleNamed: message 68, 557, 575
styleNamed:put: message 587
subcanvas
 accessing embedded widget 313
 in a notebook 334
subcanvas widget
 adding 302
 blinking 312
Subcanvas1Example 303
Subcanvas2Example 306, 308
Subcanvas3Example 310, 312, 313
subclass
 creating 26
subject
 of an adaptor 706
subject channel
 of an adaptor 706
subject: message 708, 715

subject:triggerChannel: message 711
subjectChannel: message 707, 715
subjectSendsUpdates: message 708
submenu
 creating 231
submenu message 261
substituting a menu 243
substring
 extracting 547
subtractDate: message 471
subtractDays: message 471
subtracting
 dates 471
 numbers 431
 time 482
subtractTime: message 482
super message 25
support, technical xix
 electronic bulletin boards xx
 electronic mail xx
 fax xxi
 telephone xxi
 World Wide Web xx
swapping interfaces 310
symbol
 in a field 125
symbols used in documentation xiv-xvi
synchronizing
 updates in model 710
System font 588
systemDefault text style 575

T

tab stops in text 562
tabbing order 74
table
 updating 221
table widget

- adding 216
- aligning labels 223
- connecting to input field 221
- labeling 223
- Table1Example 218
- Table2Example 221
- Table3Example 223
- TableInterface class 216
- tail message 595
- takeKeyboardFocus message 150, 151
- tan message 442
- target emphasis 356, 361
- technical support xix
 - electronic mail xx
 - electronic bulletin boards xx
 - fax support xxi
 - telephone support xxi
 - World Wide Web xx
- telephone support xxi
- templatesMenuForMenuBar message 229
- temporary variable
 - declaring 13
 - removing 16
 - undeclared 19
 - usage 10
- testing
 - equality of numbers 438
- text
 - adding emphasis 572
 - aligning 561
 - boldfacing 572
 - changing case 571
 - color 585
 - comparing 568
 - copying a subtext 569
 - creating 556
 - custom font 576
 - default font 575
 - displaying 558
 - editor
 - See also editor widget
 - font family 582
 - font size 578
 - in a field 125
 - indents and tabs 562
 - line length 559
 - line spacing 580
 - patterned 585
 - printing 565
 - replacing a subtext 567
 - searching 566
 - size 564
 - storing in file 598
 - vs. string 556
 - word wrapping 560
- text style
 - adding 587
 - creating 576
 - default 588
 - emphasis 572
 - See also font
 - tab stops 562
- text: message 571
- TextAttributes 68, 556, 575, 578, 583, 587, 588
 - See also text style
- textStyle message 563
- textStyle: message 69, 563, 575, 577, 579, 583
- textual dialog 284
- thickness
 - box widget 157
 - of a line 644
- tile
 - in a pattern 692
 - phase 693
- tilePhase: message 693

Time 454
time
 adding 482
 creating 478
 current 478
 in a field 125
 seconds 480
 subtracting 482
 zones 486
time stamp 483
 in a field 125
timeDifference:DST:at:from:to:startDay:
 message 487
timesRepeat: message 34
TimestampPrintPolicy 131
TimeZone 486
timing code 484
title
 See also window label
to:by:do: message 35
to:do: message 34
today message 462
tooldd.st 341
topComponent message 99, 411
totalSeconds message 480
transparent message 690
trigonometric functions 442
true-false dialog 280
truncated message 434
truncateTo: message 435
truncating
 a string 551
two-dimensional slider 274
type
 of field input 125
Type property 125, 129
TypeConverter 127, 131
typographic conventions xiv-xvi

U

UI

 See also interface
UIBuilder 84
 See also builder
unbounded widget 58, 62
Undeclared 13
undeclared variables 13
underline emphasis 572
unhideItem: message 247
unity message 458
unmap message 104
update:with: message 653
update:with: method 383
updating
 a table 221
 a view 382
 an aspect adaptor 707
 buffered 710
uppercase
 See also case
useHorizontalScrollBar message 97
user interface
 See also interface
user-input event 392
useTabs: message 563
useVerticalScrollBar message 97

V

validating
 field input 132
Validation properties 133
value holder 122, 704
 dependency 78
valueAtPoint: message 664
valueAtPoint:put: message 664
valueNowOrOnUnwindDo: message 600,
 614

- variable
 - accessing 10
 - class instance 11
 - declaring 13
 - default value 13
 - naming 13
 - pool dictionary 12
 - removing 16
 - See also class variable
 - See also global variable
 - See also instance variable
 - See also temporary variable
 - types 10
 - undeclared 13
 - variable-byte class
 - creating 27
 - VariableSizeTextAttributes 578
 - version
 - of a BOSS file 624
 - vertices: message 637
 - view
 - connecting to controller 385, 400
 - connecting to model 378
 - creating 376
 - displaying 380
 - integrating in interface 389
 - invalidating 387
 - redisplaying 387
 - updating 382
 - with no controller 386
 - viewHasCursor message 421
 - viewHasCursorWithEvent: message 421
 - visibility of a widget 70
 - visual component
 - See also widget
 - visualAt:put: message 118
 - visuals message 119
 - VisualWorks documentation
 - online xix
 - Database Cookbook xix
 - Database Quick Start Guides xix
 - International User's Guide xix
 - VisualWorks Cookbook xix
 - VisualWorks DLL and C Connect Reference xix
 - printed
 - Database Connect User's Guide xviii
 - Database Tools Tutorial and Cookbook xviii
 - Installation Guide xviii
 - International User's Guide xviii
 - Object Reference xviii
 - Release Notes xviii
 - Tutorial xviii
 - User's Guide xviii
- ## W
- waitButton message 100, 421
 - waitClickButton message 421
 - waiting
 - for mouse click 421
 - waitNoButton message 421
 - warn: message 278
 - warn:for: message 292
 - warning dialog 278
 - weekday message 465
 - weekDayToStartDST message 486
 - whileTrue: message 35
 - white space characters 535
 - whiteBlack message 680
 - widget
 - accessing programmatically 54
 - aligning 65
 - border 67
 - bounded vs. unbounded 58
 - color 75
 - dependency 78

- disable 72
- distribute 66
- drop source. See drop source
- drop target. See drop target
- embedded in subcanvas 313
- font 68
- grouping visually 156
- hiding 70
- positioning 60
- separating groups 154
- sizing 56
- spacing 66
- tab order 74
- unbounded 62
- widget message 55
- WidgetWrapper 54, 411
- width
 - of a string 539
 - of dataset columns 207
 - of table columns 219
- wildcard
 - in a string search 544
- window
 - accessing dimensions 88
 - active 99
 - at a location 100
 - clearing size constraints 88
 - closing 101
 - color 94
 - Event Driven property 394
 - events 106
 - expanding and collapsing 103
 - from a builder 85
 - hiding 104
 - icon 108, 682, 683
 - label 92
 - master 292
 - menu bar 98, 233
 - moving 90

- opening 82
- refreshing 93
- scroll bars 96
- sizing 86
- slave 105
- window message 85
- windowAt: message 100
- #windowSpec 83
- with:do: message 527
- with:with:with:with: message 492
- withAll: message 493
- withColors: message 696
- withCRs message 553
- withText:style: message 557
- wordWrap: message 560, 570
- working directory 594
- World Wide Web xx
- wrapper 54
- wrapping text 560
- write stream 609, 614
- WriteStream 545
- writeStream message 598

Y

- year
 - counting days 469
 - day in 468
 - days left 469
 - in a date 469
- yellowButton 412
- yellowButtonPressed message 413
- yes-no dialog 280
- yielding control 403

Z

- zero 458
- zone, time 486