

Customizing Your Editing Environment

HyperLook lets you customize your stack editing environment in a number of ways. You can define new types of stacks, new types of object, and you can customize the stack editor.

This is very useful, for example, if you want to impose a company style on all stacks and objects.

Stack Editor Properties

Selecting **Editor Info** from the **Edit** menu shows the editor properties for the stack. Each stack has its own editor properties. This means that you can configure each stack differently.

Figure 122

The editor properties of a stack.

Stack Properties

Stack: Untitled-#2

Files:

Client: [7]

Objects: 1 Card and 1 Card object, 1 BackGround and 3 BackGround objects, 3 Stack objects

Grid: 5 **Border:** 10

☒ **Edit Invisible**

Select Color: [Color Picker]

Resize Color: [Color Picker]

Drag Color: [Color Picker]

Center Color: [Color Picker]

Script **Help** **Reset** **Apply** **Done**

Stack Editor Properties Explained

The **Grid** numeric field lets you set the grid size which is used when dragging and resizing objects in the stack.

The **Border** numeric field defines how big the resize corners of selected objects are. The wider the resize corners, the easier it is to resize objects.

The **Edit Invisible** checkbox specifies whether invisible objects can be selected and edited.

The **Select Color**, **Resize Color**, **Drag Color**, and **Center Color** color selectors let you choose various colors which are used by the editor to draw the selection borders etc.

Creating Your Own Untitled Stack

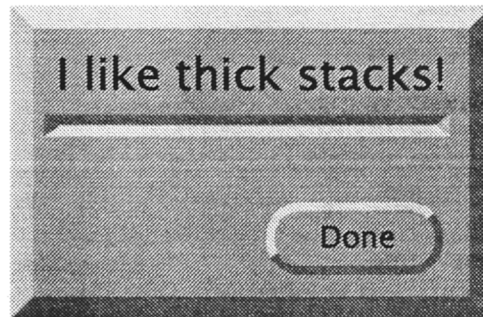
HyperLook lets you re-define the **Untitled** stack. This means that you can create your own style of stacks.

Create a stack to your own specification, then save it using **Save As ...** from the **Stack** menu, with the name **Untitled**.

Now when you press the **New** button in the **OpenStack** stack a new copy of *your* **Untitled** stack will be used!

Figure 123

Creating your own style of stack.



You can define several generic stacks and use them as templates when creating new stacks. Let's say you've created a stack called **Template**. To create a copy of this stack, use the **OpenStack** dialog (see "Creating a New Stack" on page 5-5), type **Template** in the **Name** field and press **New**.

Creating Your Own Object Warehouse

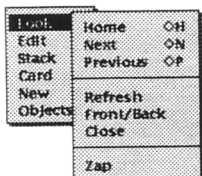
When designing your own systems with HyperLook, you may want to create your own set of object templates. To create your object warehouse, create a stack with template objects. Each card of the stack should hold a set of related objects.

The names of the template objects should reflect what they do, since the name will be used as a menu item (see "Object Name Property" on page 6-4 on how to set the object name). Objects that have a name which starts with "#" are not considered as template objects.

Copy the **Install** button from the **Warehouse** stack and paste it into your warehouse stack. Now you are ready to install the template objects in to the **New** menu.

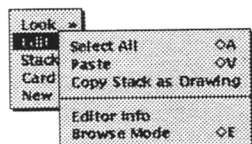
Edit Mode Menus

Some of the menus explained here provide different choices depending on whether *no* object is selected, *one* object is selected or *several* objects are selected.



Look

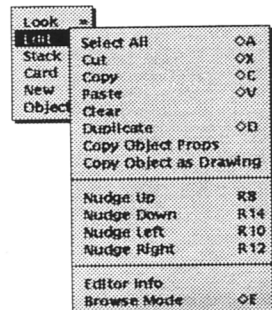
The **Look** menu is explained in "The Stack Menu" on page 3-36.



Edit

Select All

Select all the object on the stack, current background and current card (see "Selecting Objects" on page 5-15).



Cut

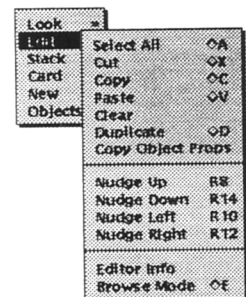
Copy

Paste

Clear

Duplicate

Clipboard editing operations. Only **Paste** is available when no objects are selected (see "Copying and Pasting Objects" on page 5-20).



Copy Object Props

Copy the properties of an object to the clipboard (see "Copying and Pasting Object Properties" on page 5-22).

Copy Object as Drawing

Copy Stack as Drawing

Copy an object (if one object is selected) or copy the stack to the clipboard as a drawing (see "Copy as Drawing" on page 5-23).

Nudge Up
Nudge Down
Nudge Left
Nudge Right

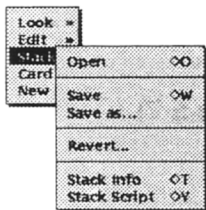
Nudge the selected objects in the indicated direction (see “Moving Objects” on page 5-16).

Editor Info

Show the editor properties of the stack (see “Stack Editor Properties” on page 5-33).

Browse Mode

Switch the stack out of edit mode (see “Edit Mode” on page 5-3).



Stack

Open

Show the OpenStack dialog (see “Opening Stacks” on page 3-8).

Save

Save as...

Save the stack (see “Saving Stacks” on page 5-6).

Revert

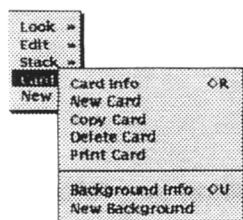
Revert the stack to the previously saved version (see “Reverting Stacks” on page 5-6).

Stack Info

Show the stack properties (see “Stack Properties” on page 5-8).

Stack Script

Show the stack’s script (see “Scripting” on page 7-1).



Card

Card Info

Show the properties of the current card (see “Card Properties Explained” on page 5-26).

New Card

Create a new card (see “Adding and Deleting Cards” on page 5-25).

Copy Card

Copy the current card to the clipboard (see “Copying Cards to the Clipboard” on page 5-26).

Delete Card

Delete the current card (see “Adding and Deleting Cards” on page 5-25).

Print Card

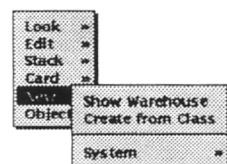
Print the current card (see “Printing Cards” on page 5-7).

Background Info

Show the properties of the background of the current card. See “BackGround Properties” on page 5-27.

New Background

Give the current card a new background (see “Using BackGrounds” on page 5-26).



New

Show Warehouse

Show the object warehouse (see “Using the Object Warehouse” on page 5-13).

Create from Class

Create an object from a resource class (see “Scripting” on page 7-1).

System

Create a new system object (see “Creating New Objects” on page 5-12).

More submenus can be added to the **New** menu (see “Using the Object Warehouse” on page 5-13).

Object(s)

Info

Show the object properties. This menu item is only available if a single object is selected (see “Object Properties” on page 6-1).

Script

Show the script of the selected object. This menu item is only available if a single object is selected (see “Scripting” on page 7-1).

To Front

To Back

Send objects to the front or to the back (see “Front to Back Order” on page 5-18).

Align Center

Align Left

Align Right

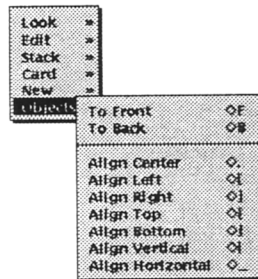
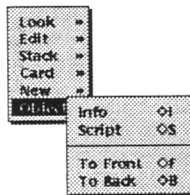
Align Top

Align Bottom

Align Vertical

Align Horizontal

Align multiple objects. This menu item is only available when more than one object is selected (see “Aligning Objects” on page 5-18).







HyperLook

Object Properties

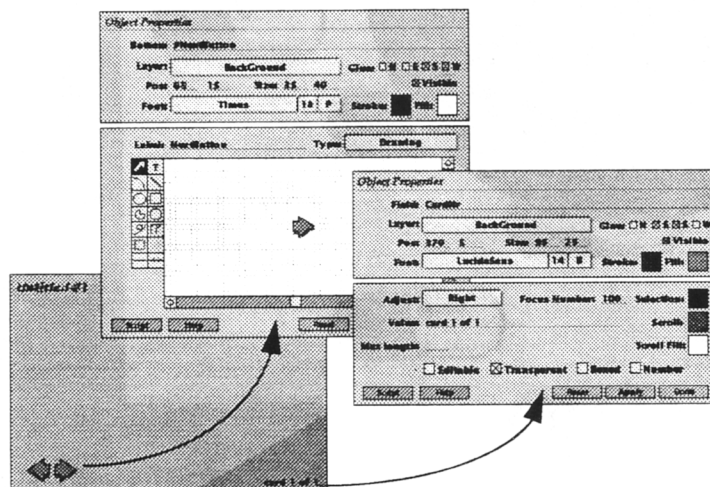
6

This chapter explains the properties of the HyperLook user interface objects. In Chapter 5, "Editing Stacks" you found out how to create and edit objects interactively.

In this chapter you will find out how to change an object's properties so that it looks just the way you want it to. You have the freedom to change object properties such as, color, font, shape and size. Every HyperLook object has properties which can be changed.

Figure 124

Every object has properties.



Before reading this chapter, you should be familiar with Chapter 3, "Using HyperLook" and Chapter 5, "Editing Stacks".

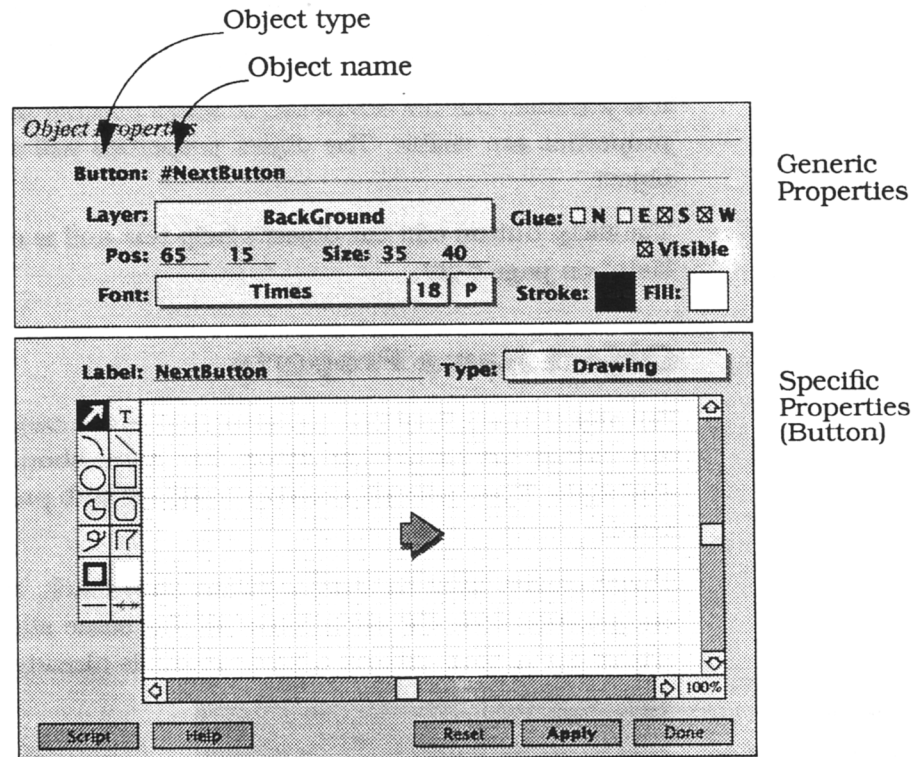
Generic Properties



When editing a stack you can edit the properties of an object by selecting the object and then choosing **Info** from the stack's **Object** menu. You can also double click on the selected object or press the **Props** function key on the keyboard, while the object is selected. This shows the object's property stack.

Figure 125

The properties of a Button object.



The property stack is divided into two sections. The top half contains properties that apply to every object in the system (generic properties). The bottom half contains the properties that apply only to the type of object which you have selected (specific properties).

The generic properties are explained next. After that you'll find out about the specific properties of each type of object.

Editing Properties

When editing properties, no changes are made to the selected object until you press the **Apply** button. The properties are then applied to the object and the object is given the new values.

You can throw away changes made to the properties by pressing **Reset**. When you have finished editing the properties press **Done**. Remember to press **Apply** if you want to make the changes permanent. You will be warned if you press **Done** without first pressing **Apply**.

The **Script** button shows the object's script. See Chapter 7, "Scripting".

It is possible, but not advisable, to select another object while an object's properties are visible. The object properties will still apply to the first object.

The **Help** button edit the object's help text and is explained in "Defining Help" on page 5-29.

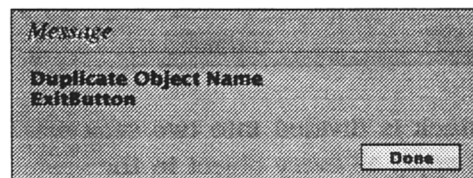
Object Name Property

The object name field (the top most field in the **ObjectProps** stack), lets you change the name of an object. The name should contain only letters and digits. This makes it easier to interface with programming languages like C.

Object names must be unique within each stack. You cannot define two components with the same name on the same stack. A warning is displayed if you have chosen a name which is already in use.

Figure 126

Somebody tried to create a second object called **ExitButton**.



Layer Property

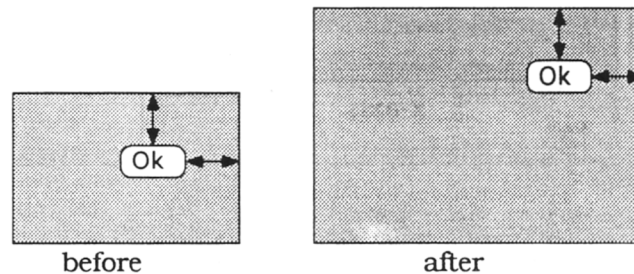
The **Layer** pulldown lets you select the layer of the object. You can select **Stack**, **Background** or **Card**. See “Cards and BackGrounds” on page 5-24 on how these layers affect the object's behavior.

Glue Property

The **Glue** checkboxes¹ let you specify the way an object behaves when the stack is resized (see “Making a Stack Resizable” on page 5-11 on how to make a stack resizable). You can define how the object is *glued* to the edges of the stack.

Figure 127

An object glued to the north and to the east edge of the stack.

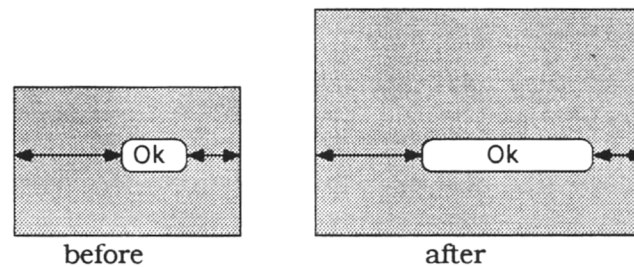


For example, an object which is glued to the north edge and to the east edge of the stack will remain in the same position relative to the north east corner of the stack.

If an object is glued to opposing edges, i.e. the west and the east edge, it will be stretched in order to satisfy the glue constraints.

Figure 128

An object glued to the east and to the west side of the stack.



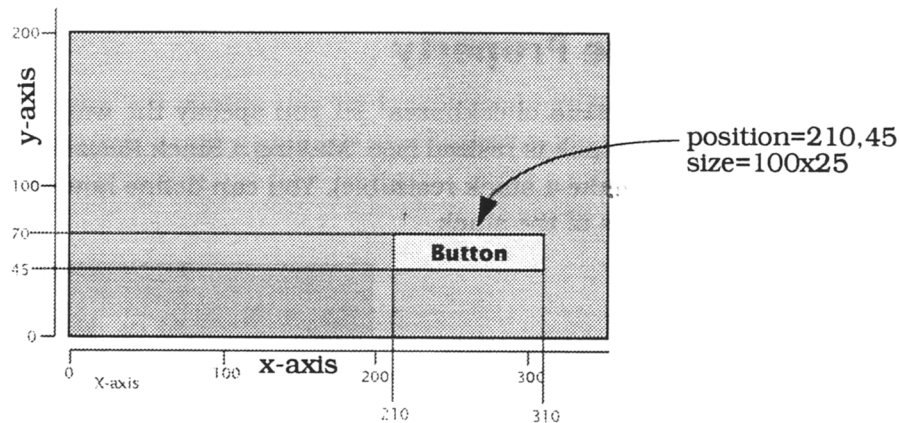
1. **N** = north, **E** = east, **S** = south, **W** = west

Position and Size Properties

The **Pos** and **Size** numeric fields let you specify the exact position and size of the object. See “Moving Objects” on page 5-16 and “Sizing Objects” on page 5-17 on how to change these parameters interactively.

Figure 129

The coordinate system of a stack.



The coordinates are relative to the bottom left corner of the stack.

Visible Property

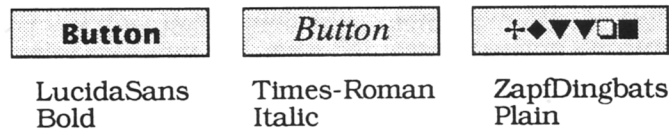
The **Visible** checkbox lets you select whether an object is visible or not. It is sometimes convenient to make an object invisible. The user can not interact with an invisible object in any way, until you make it visible again!

Font Property

The **Font** pulldowns let you change the font, font size and font style¹ of the object. See “Changing the System Fonts” on page 3-31 to see how to change the list of fonts.

Figure 130

Buttons with different fonts.

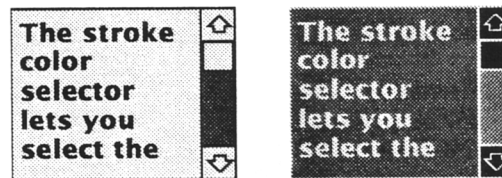


Color Properties

The **Stroke** color selector lets you select the stroke (line and text) color of the object. The **Fill** color selector lets you select the fill (interior) color of the object.

Figure 131

Changing the color properties.



See “The Color Pallet” on page 3-20 on how to change the default set of colors.

These are the generic object properties. They apply to all object types. Let’s now look at specific object properties for different types of object.

1. **P** = plain, **B** = bold, **I** = italic, **BI** = bold and italic

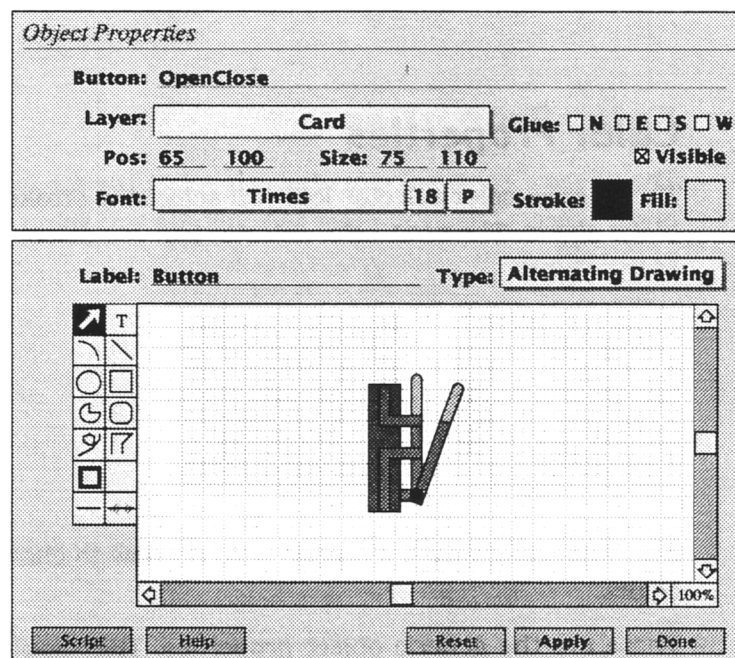
Button Properties

Buttons do something when you press them and they are used a lot in HyperLook. Lots of different types of button are provided, but you can also draw your own with the graphics editor.

Buttons are activated by pressing them with the left mouse button. The button's action is executed **when the mouse is released** inside the perimeter of the button.

Figure 132

The properties of a Button object.



We're only interested in the specific button properties, so concentrate on the bottom half of Figure 132.

The **Type** pulldown lets you select the type of button. You can choose from **Rounded Button**, **Square Button**, **Transparent Button**, **Checkbox**, **Small Checkbox**, **Drawing** and **Alternating Drawing**. The button types are discussed below.

Push Buttons

Rounded push buttons and square push buttons are the most common types of button. When the mouse is clicked in the button, it highlights. When the mouse is released, the button returns to its normal state.

Figure 133

Examples of rounded and square buttons.



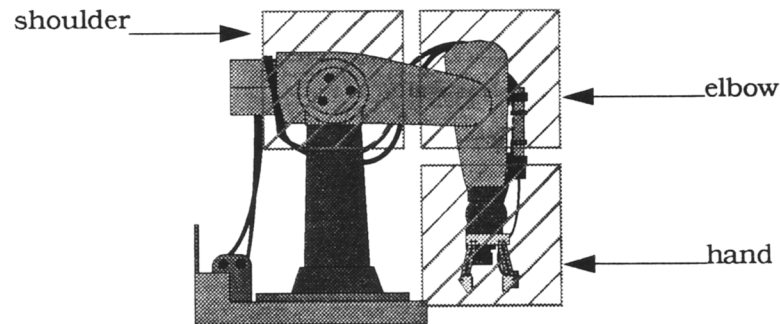
Fill in the **Label** field to specify the text displayed in the button.

Transparent Buttons

Transparent buttons are used to detect mouse clicks in an area. You can overlay them on a drawing of some sort to detect a click in a certain part of the drawing.

Figure 134

Transparent buttons on top of a drawing.



Transparent buttons act as normal buttons but they have no visual appearance, so you can't see them. You can specify a label for a transparent button if you want to, but you won't see it!

Checkboxes

A checkbox button is used to indicate a Boolean state, on or off. They come in two sizes, big or small.

Figure 135
Examples of
checkboxes.

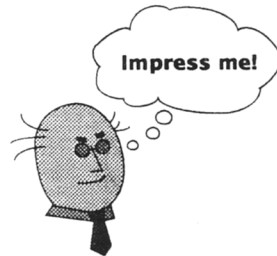


The value of a checkbox is **0** for false (not highlighted) and **1** for true (highlighted). Fill in the **Label** field to specify the text printed next to the text box.

Drawing Buttons

A drawing button is most frequently used to display a drawing in a stack. Pasting a drawing from a graphics editor into a stack automatically creates a drawing button containing the drawing (see “Pasting Text and Graphics” on page 5-22).

Figure 136
A drawing button.



The graphics editor lets you create or edit the drawing for the button. Note that you sometimes need to adjust the size of the button before you can see the whole drawing.

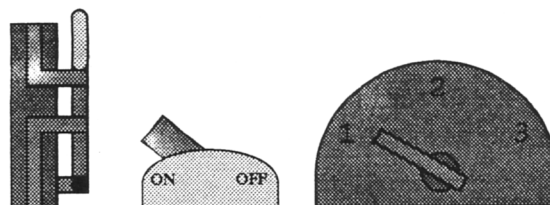
Alternating Drawing Buttons

An alternating drawing button contains multiple drawings, which are used to display different states of the button. There is one drawing for each state of the button, and each time you press the button the drawing changes from one to the next.

In this way it is easy to create graphically pleasing buttons like switches and levers.

Figure 137

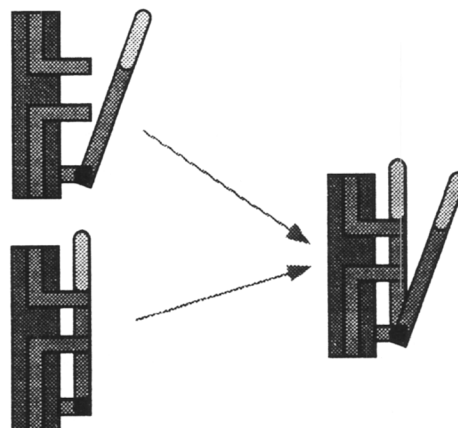
Examples of alternating drawing buttons.



To specify the states for the button you need to draw each state in the graphics editor on the button properties stack. Each state should be a single drawing object, so group objects together so that you have one object per state. Place all the states on top of each other starting with the initial state at the bottom of the pile.

Figure 138

Button states drawn on top of each other.



Now you have given HyperLook all the information it needs to create an alternating drawing button.

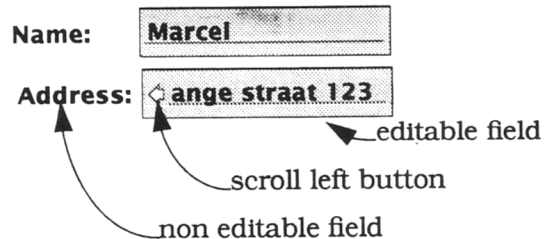
Note that the button will redraw between states using the fill color of the button as the background of the drawing. Make sure that the fill color matches the background color of the window!

Field Properties

A field is a single line of text which can scroll horizontally. It is used for labels or to let the user enter a single line of text.

You usually finish editing a text field by hitting the **Return** or **Esc** key, then the action associated with the field is then executed.

Figure 139
Examples of field objects.



See “Editing Text” on page 3-14 on how to edit text using keyboard short cuts. You can copy and paste text to and from fields using the clipboard, see “The System Properties” on page 3-31. See “Text Properties” on page 6-15 if you want to use a multi line text editor.

Figure 140
The property stack of a Field object.

Object Properties			
Field: #9			
Layer:	Card	Glue:	<input type="checkbox"/> N <input type="checkbox"/> E <input type="checkbox"/> S <input type="checkbox"/> W
Pos:	160 35	Size:	100 30
Font:	LucidaSans	14 B	<input checked="" type="checkbox"/> Visible
Stroke:		Fill:	
Adjust:	Left	Focus Number:	100
Value:		Selection:	
Max length:		Scroll:	
Scroll Fill:			
<input checked="" type="checkbox"/> Editable <input type="checkbox"/> Transparent <input type="checkbox"/> Boxed <input type="checkbox"/> Number			
Script	Help	Reset	Apply Done

Field Properties Explained

We're only interested in the specific properties of field objects, so concentrate on the bottom half of Figure 140.

The **Value** field lets you enter the text to be displayed in the field. You can do this even if the text field itself is not editable.

The **Transparent** checkbox lets you choose whether the text field is transparent or opaque. A transparent field can not be edited.

The **Boxed** checkbox lets you choose whether a box is drawn around the text field. The border color of the box is the same as the text color.

The **Selection**, **Scroll** and **Scroll Fill** color selectors let you change the colors of selected text and scroll buttons.

The **Editable** checkbox determines whether the user can type into the field directly. If the field is not editable it is usually used for display only.

The **Adjust** pulldown lets you select how the field is adjusted. You can choose **Left**, **Center** or **Right**.

The **Max length** field lets you limit the length of the text field to a maximum number of characters. If you don't enter a value here, then the maximum length is 65000 characters.

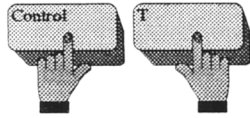
Numeric Fields

The **Number** checkbox is used to specify whether or not a field is numeric. A numeric field restricts the user to entering numbers. Numbers can start with a “-” and may contain a decimal point.

Controlling the Input Focus

The object which is receiving typed characters is said to hold the input focus. It's useful to be able to control the input focus so that user of your application can enter text in a logical sequence.

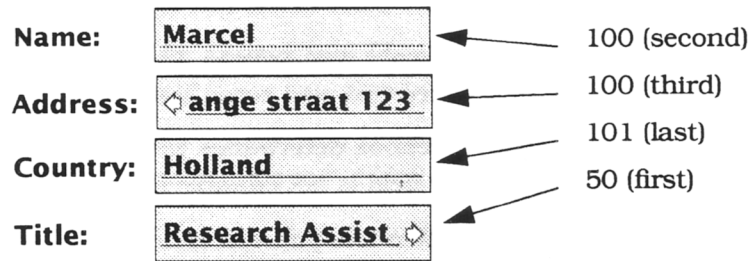
The **Focus Number** field lets you specify the focus number of the field. This number is used to establish some ordering between text objects when more than one editable text object is visible. The order is used to decide the sequence in which input focus is passed around between objects.



When editing text in a stack, typing **Control-T** forwards the input focus to next editable text object in the stack. You can also force the input focus to go to a particular text object by clicking on it. This applies to any object which can accept typed characters.

Figure 141

The focus number affects the order. The order is: Title, Name, Address, Country.



The object with the *smallest* focus number gets the initial focus, that is when the stack or card is opened. After that, the focus is forwarded to the text object with the next higher focus number.

If there is more than one text object with the same focus number, then the front to back ordering is used.

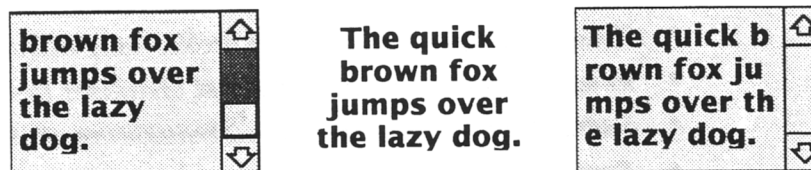
The default input focus number is 100. This number is chosen so that there are plenty of lower and higher input focus numbers. Leaving the input focus number field empty means that the object does not compete for the input focus. It means that you have to click on it to give it the input focus.

Text Properties

Text objects are used for multi-line text editing. They provide editing facilities that allow editing of text up to 65000 characters long.

Figure 142

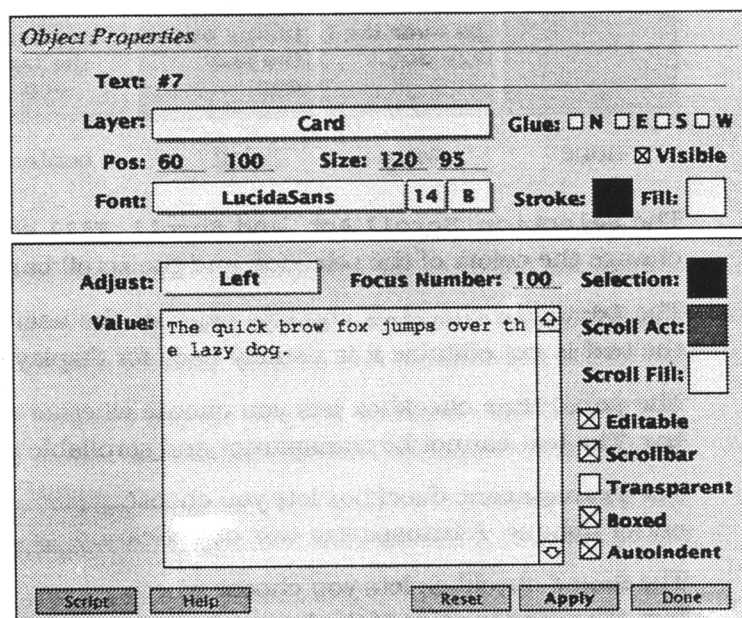
Some examples of text objects.



In this section we're concerned with text properties, not text editing. See "Editing Text" on page 3-14 to see how to edit text using keyboard short cuts. See "The Clipboard" on page 3-18 to see how to copy and paste text.

Figure 143

The properties of a Text object.



Text Properties Explained

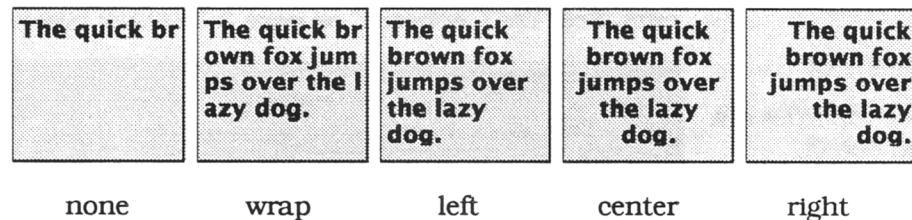
We're only interested in the specific properties of text objects, so concentrate on the bottom half of Figure 143.

The **Value** field lets you enter the text for the text object. You can do this even if the text object is not editable.

The **Adjust** field lets you select the way the text is formatted. You can choose from:

- **None.** No formatting, lines are broken only where you type a newline.
- **Wrap.** Lines are broken at the last character that fits.
- **Left.** Lines are broken at word boundaries. The text is left justified.
- **Center.** Lines are broken at word boundaries. The text is centered.
- **Right.** Lines are broken at word boundaries. The text is right justified.

Figure 144
Different text
adjustments.



The **Selection**, **Scroll Act** and **Scroll Fill** color selectors let you change the colors of the selection and the scroll bar.

The **Editable** checkbox controls whether the user can edit the text. If the text is not editable it is usually used for display only.

The **Scrollbar** checkbox lets you choose whether the text has a scrollbar. The text cannot be transparent and scrollable at the same time.

The **Transparent** checkbox lets you choose whether the text is transparent or opaque. A transparent text object cannot be editable.

The **Boxed** checkbox lets you choose whether a box is drawn around the text. The border color of the box is the same as the text color.

The **AutoIndent** checkbox lets you select whether the text is in *auto indent* mode. If the text is in *auto indent* mode, typing a newline will indent the new line the same as the previous line. This feature is useful for entering program text.

See "Controlling the Input Focus" on page 6-13 for how to use the **Focus Number** field to control the input focus.

List Properties

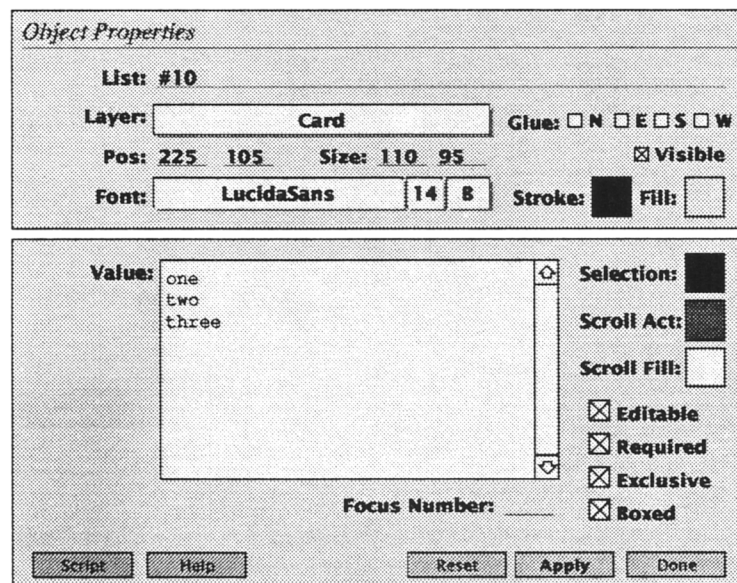
List objects are used to select items from a list. List items are useful if you want to present a long list of items, for example, fonts.

Figure 145
Some examples of List objects.



List objects are either exclusive or nonexclusive. An exclusive list object can only have one item selected at any time. A nonexclusive list object can have many items selected. See “Scrolling Lists” on page 3-16 on how to use List objects with keyboard short cuts.

Figure 146
The properties of a List object.



List Properties Explained

We're only interested in the specific properties of list objects, so concentrate on the bottom half of Figure 146.

Section "Generic Properties" on page 6-3 explains how to change the generic properties of a List object.

The **Value** field lets you specify the items in the list. You can specify any number of items separated by newlines.

The **Selection**, **Scroll Act** and **Scroll Fill** color selectors let you change the colors of the selection and the scroll bar.

The **Editable** checkbox determines whether the user can change the selection by clicking on items.

The **Required** checkbox indicates whether at least one item must be selected or not.

The **Exclusive** checkbox lets you choose whether the list is exclusive or nonexclusive. An exclusive list can have at most one item selected.

The **Boxed** checkbox lets you choose whether a box is drawn around the list. The border color of the box is the same color as the items.

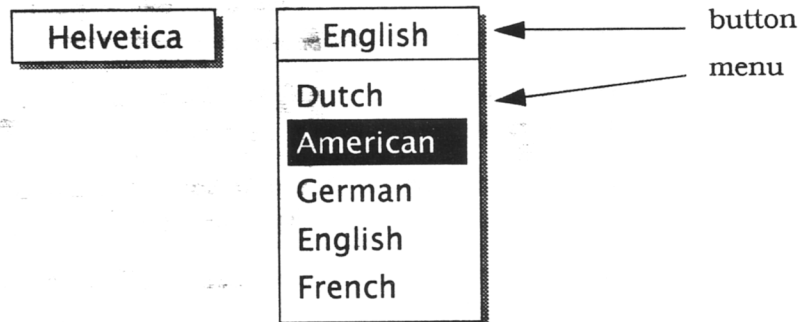
See "Controlling the Input Focus" on page 6-13 on how to use the **Focus Number** field to control the input focus.

PullDown Properties

PullDown objects are buttons with a pulldown menu. Pulldowns are displayed as a square button, usually with a shadow.

Figure 147

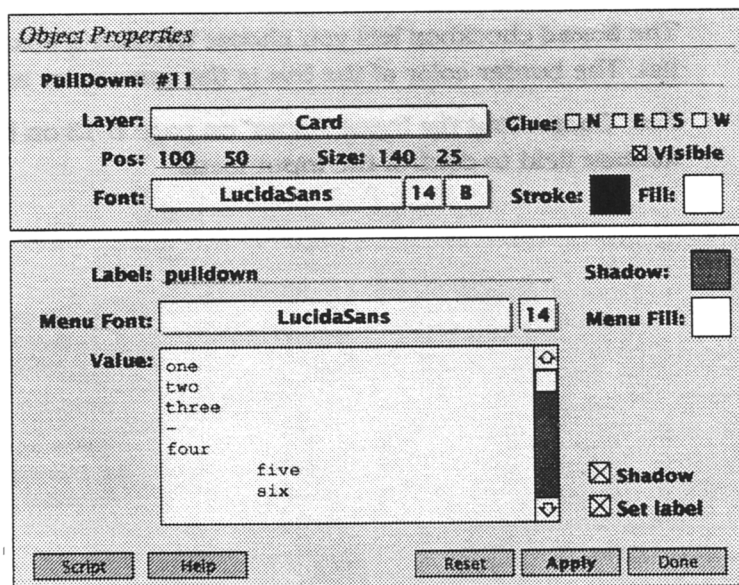
Some examples of PullDown objects.



PullDown objects are used to select from a menu. A good example of a pulldown object is the font selector pulldown in the object properties stack.

Figure 148

The properties of a PullDown object.



PullDown Properties Explained

We're only interested in the specific properties of PullDown objects, so concentrate on the bottom half of Figure 148.

The **Value** field lets you specify the items in the pulldown menu. You can specify any number of items (one line per item). A single "-" item will put a horizontal line in the menu. See also "Pull Right Menus in PullDown Objects" later in this section.

The **Label** field lets you enter the text displayed in the pulldown button. The **Set Label** checkbox indicates whether the label of the pulldown button should change when a selection is made from the menu. If Set Label is true, the label will display the most recent selection.

The **Shadow**, and **Menu Fill** color selectors let you change the colors of the shadow and the menu.

The **Menu Font** pulldowns let you select the font which is used in the menu.

The **Shadow** checkbox determines whether the pulldown button has a shadow to distinguish it from ordinary square buttons.

Pull Right Menus in PullDown Objects

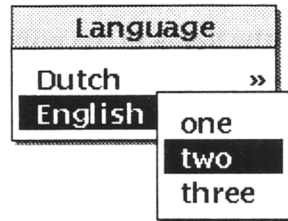
You can specify pull right menus for PullDown objects by indenting the items in the **Value** field accordingly. Items which are indented by equal amounts are assumed to be part of the same menu. For example:

```
Dutch
  een
  twee
  drie
English
  one
  two
  three
```

This results in three menus. The top level pulldown menu contains two items, **Dutch** and **English**. Each item has a pull right menu with three items.

Figure 149

An example of a pull right menu in a PullDown object.

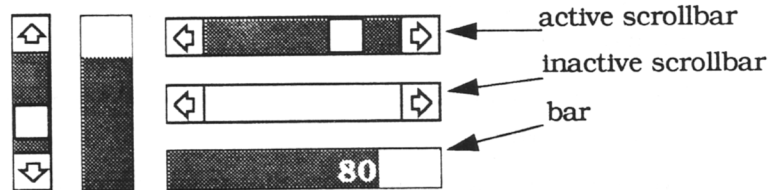


Slider Properties

Slider objects let you select a numeric value. Sliders come in two flavors: *bar* and *scrollbar*. Bar sliders are usually used to select a value from a range. Scrollbars are used to scroll data such as text or graphics.

Figure 150

Some examples of Slider objects.



See “Moving Around” on page 4-8 to see how to use scrollbars.

Figure 151

The properties of a Slider object.

Object Properties

Slider: #12

Layer: Card Glue: ☐ N ☐ E ☐ S ☐ W

Pos: 100 80 Size: 130 20 ☒ Visible

Font: LucidaSans 14 B Stroke: Fill:

Min: 10 Value: 80 Max: 100 Value:

Type: ScrollBar

☒ Editable ☐ Vertical ☒ Show value ☒ Boxed

Script Help Reset Apply Done

Slider Properties Explained

We’re only interested in the specific properties of slider objects, so concentrate on the bottom half of Figure 151.

The **Min**, **Max**, and **Value** numeric fields let you specify the minimum, maximum and current value of the slider itself. If the minimum value is greater or equal to the maximum, or if the current value is outside the legal range, the slider is made inactive.

The **Value** color selector lets you change the color of the bar or scrollbar.

The **Type** pulldown lets you select the type of slider, either **Bar** or **ScrollBar**.

The **Editable** checkbox determines whether the user can change the value of the slider.

The **Vertical** checkbox indicates whether the slider moves vertically or horizontally.

The **Show Value** checkbox lets you choose whether a **Bar** slider shows its current value.

The **Boxed** checkbox lets you choose whether a box is drawn around the Slider.

ColorSelect Properties

ColorSelect objects let you select a color from a popup color pallet. The currently selected color is displayed in the color rectangle. See “The Color Pallet” on page 3-20 on how to change the colors on the popup color pallet.

Figure 152

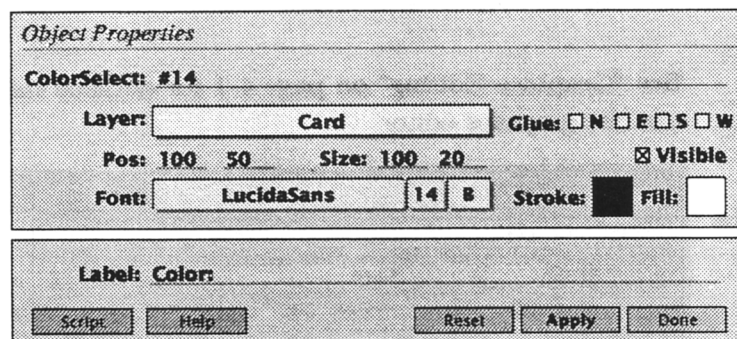
Some examples of ColorSelect objects.

Line Color:  Fill Color: 

An example of how to use a popup color pallet is described in “Color Selectors” on page 3-17.

Figure 153

The properties of a ColorSelect object.



The screenshot shows the 'Object Properties' dialog box. The 'ColorSelect: #14' section is active. It contains the following fields and controls:

- Layer:** A text field containing 'Card'.
- Glue:** A set of checkboxes for 'N', 'E', 'S', and 'W', all of which are currently unchecked.
- Pos:** Two text fields containing '100' and '50'.
- Size:** Two text fields containing '100' and '20'.
- Visible:** A checked checkbox.
- Font:** A text field containing 'LucidaSans', followed by two small buttons labeled '14' and 'B'.
- Stroke:** A color selection rectangle showing black.
- Fill:** A color selection rectangle showing white.

Below the 'ColorSelect' section is a 'Label: Color:' section with four buttons: 'Script', 'Help', 'Reset', and 'Apply'. The 'Done' button is located at the bottom right of the dialog box.

ColorSelect Properties Explained

We're only interested in the specific properties of ColorSelect objects, so concentrate on the bottom half of Figure 153.

ColorSelect objects are really very simple. All you can change is the **Label**. The size of the color rectangle is controlled by changing the size of the object.

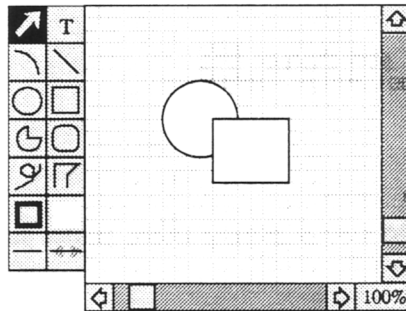
To change the colors appearing in the popup menu, use the ColorPallet stack and modify the specific entries as described in “The Color Pallet” on page 3-20.

DrawTool Properties

The DrawTool object is a complete graphics editor in the shape of a HyperLook object. It lets you incorporate the graphics editing features in your own application.

Figure 154

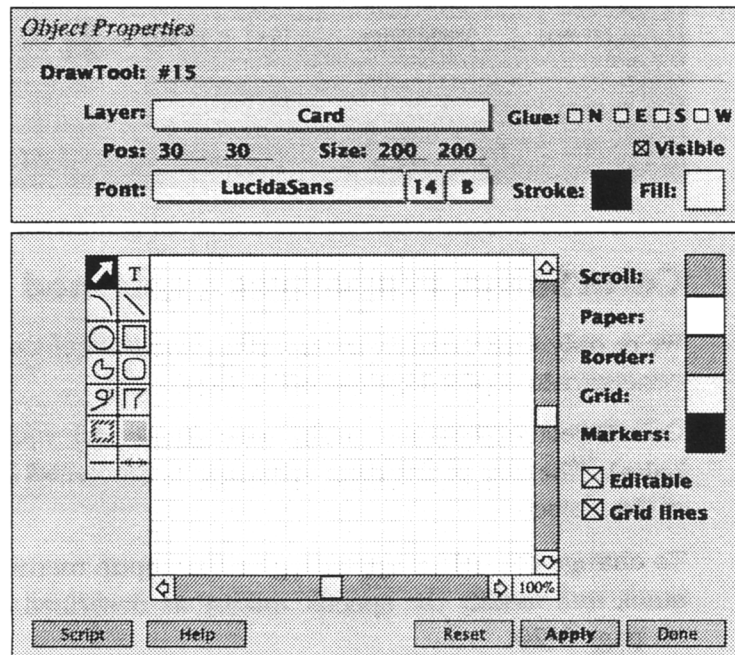
An example of a DrawTool object.



See "Graphics Editing" on page 4-1 for detailed information on how to use the graphics editor.

Figure 155

The properties of a DrawTool object.



DrawTool Properties Explained

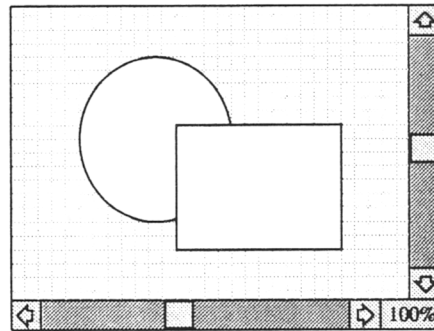
We're only interested in the specific properties of DrawTool objects, so concentrate on the bottom half of Figure 155.

The graphics editor on the properties stack lets you edit the drawing of the DrawTool object. You can do this even if the object is not editable.

The **Editable** checkbox determines whether the user can change the drawing in the DrawTool object. A non editable DrawTool object does not have the tool pallet and its menu is restricted.

Figure 156

An example of a non editable DrawTool object.



The **GridLines** checkbox lets you choose whether the grid lines are drawn or not.

The **Scroll** color selector lets you choose the color of the scrollbars.

The **Paper**, **Grid** and **Border** color selectors let you choose the color of the paper, the grid lines, and the borders of the paper.

The **Markers** color selector determines the color of the markers used to indicate the selected object.



HyperLook

Scripting

This chapter explains how to program HyperLook objects. Programming a HyperLook object is called scripting. The language that is used is PostScript.

Writing scripts is useful if you want to define bits of program which are executed when, for example, a button is pressed. But you are not restricted to simple programs alone, you can redefine most behaviors of an object by writing scripts. You can even create new classes of object.

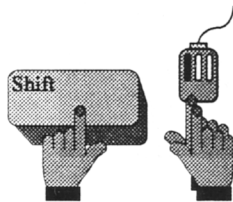
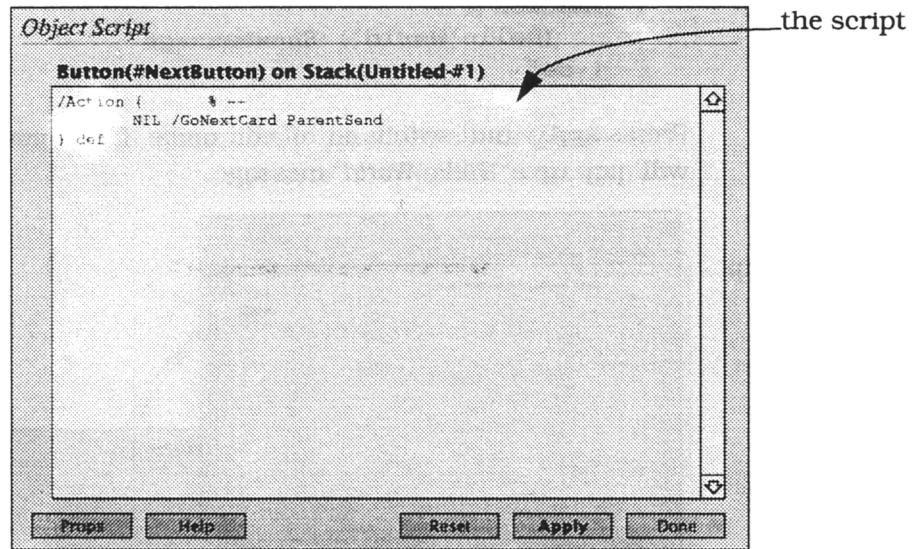


Editing Scripts

To edit the script of an object, switch the stack to edit mode and select the object. Now select **Script** from the **Object** menu. The script property stack is shown.

Figure 157

The Script properties of a button.



The script of an object can also be shown by double clicking the object while holding down the **Shift** key.

The scripts properties of Stacks, BackGrounds, Cards and objects are shown by pressing the **Script** button on the appropriate property stack.

When you are finished editing the script, press the **Apply** button. This evaluates the script and assigns it to the object. You can throw away any changes that you have made by pressing **Reset**. Press **Done** when you are finished.

Pressing the **Props** button shows the object properties of the object. These are explained in Chapter 5, "Editing Stacks" and Chapter 6, "Object Properties". The **Help** button is explained in "Defining Help" on page 5-29.

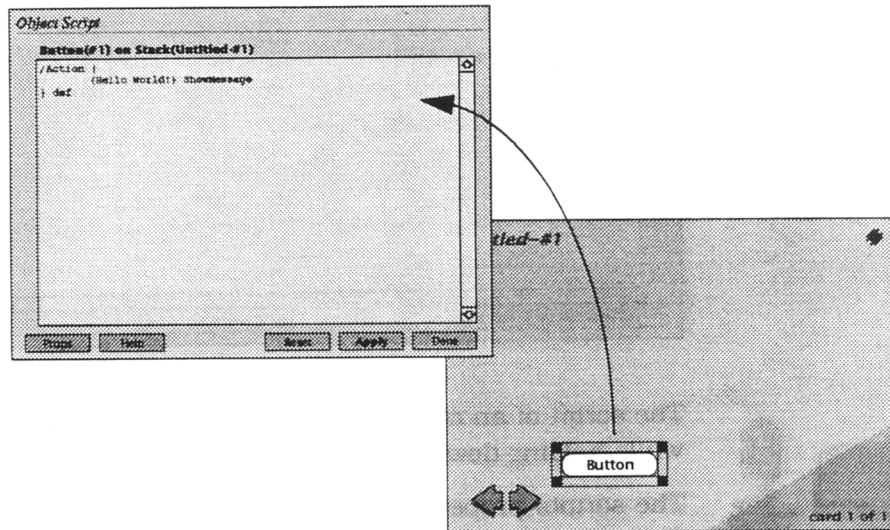
An Example of a Script

Now try writing your own script. Create a new button and get its script properties. Enter the following script (see “Editing Text” on page 3-14 for some useful editing hints):

```
/Action {
  (Hello World!) ShowMessage
} def
```

Press **Apply** and switch out of edit mode. If you now press the button it will pop up a “Hello World” message.

Figure 158
Editing the script
of a button.



A script should contain only PostScript variable and function definitions. It should not contain code which gets executed when the script is applied. See “The OnOpen and OnClose Methods” on page 7-25 on how to perform initializations.

This is because when a stack is loaded from disk for the first time, its script evaluated in a strange environment. Not all operations are guaranteed to work in that state. Please read on, later on in this chapter more information about writing scripts is given. And there are lots of examples too!

The PostScript Language

The programming language used in scripts is called PostScript. If you want to learn more about PostScript, read the Adobe PostScript manuals and the NeWS® manuals (see the end of this section for details)¹.

It is beyond the scope of this manual to give a comprehensive introduction to the PostScript language. It assumes that you are familiar with PostScript.

But don't panic if you don't know PostScript. Just read on and copy the examples that are given. First, a little bit of background on the language itself.

Postfix Notation

The most striking aspect of the PostScript language is its notation. PostScript uses postfix notation. That means that the arguments of an operator come *before* the operator name. For example, C uses *infix* notation:

$$1 + 2$$

In PostScript this is written differently:

$$1\ 2\ \text{add}$$

Postfix notation is not hard to learn. You have to get used to it, but once you understand the principles behind it, it's very easy.

Stack Operations

Most PostScript functions use the *operand stack*. This is a stack of things like temporary values and arguments to function. Take the following expression:

$$6 - (2 + 3)$$

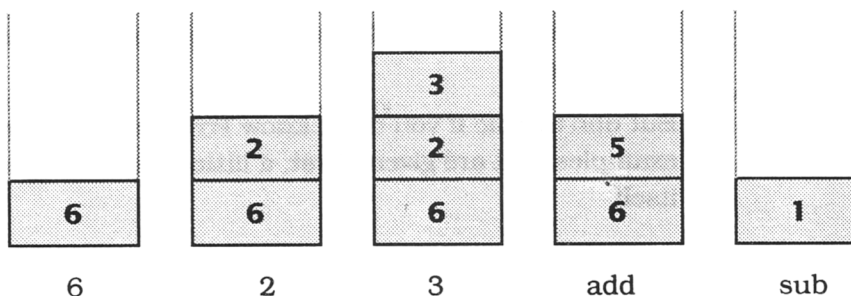
1. NeWS is a registered trademark of Sun Microsystems Incorporated.

The PostScript equivalent of this expression is as follows:

6 2 3 add sub

To understand this you could imagine the numbers being pushed on the stack as shown in Figure 159.

Figure 159
The PostScript operand stack.



You never need to use parenthesis to group sub expressions in PostScript. All PostScript expressions are unambiguous (even without parenthesis).

PostScript provides lots of operators which let you manipulate the items on the stack. For example, **pop** discards the top item of the stack, **dup** duplicates the top item of the stack, and **exch** swaps the top two items of the stack.

Notation

A brief note on the notation of variables and functions is useful because you will see it used a lot in examples that follow.

A PostScript function is defined as follows (note that the '%' sign starts a comment):

```
/functionname {    % arguments -- results
    ... Postscript Code ...
} def
```


In the case of multiple arguments or results, they are specified in bottom to top order as they occur on the operand stack. A PostScript variable is defined as follows:

```
/variablename initialvalue def
```

More Information on PostScript

There are four sources of information (apart from this manual) that you may find useful when writing HyperLook scripts.

- **The PostScript Language Reference Manual**¹. It explains the PostScript language and all the PostScript operators. The PostScript interpreter used by OpenWindows is almost 100% compatible with Adobe PostScript (level 1).
- **The NeWS programmer's manual**. This manual lists the differences between Adobe PostScript and the PostScript used by OpenWindows. It also describes the many enhancements and utilities included in to support interactive programming.
- **The Hyperlook documentation browser**. This on-line tool lets you access the documentation on all the PostScript functions that HyperLook provides (see "The Documentation Browser" on page 7-10).

1. ISBN: 0-201-10174-2



Classes

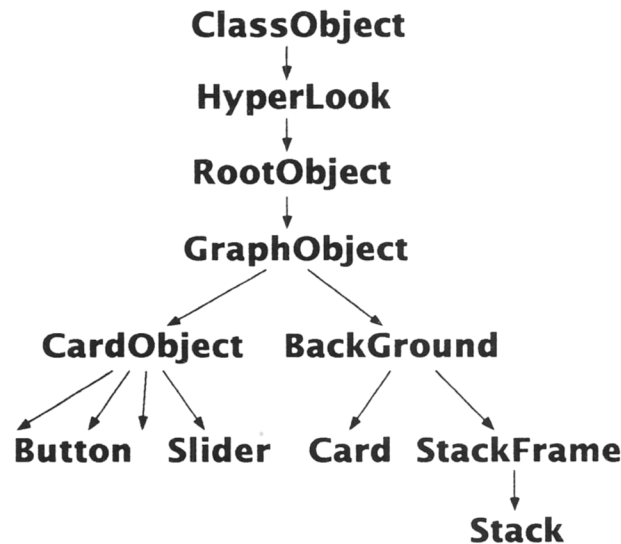
A *class* defines the behavior of an object. All objects in the HyperLook system are implemented using classes. HyperLook is an *object oriented* system.

A class contains *variables* and *methods*. Methods are functions which are defined in the context of the class. A method can always access other methods and variables of the class.

An *instance* is an actual object created from the class. Take for example the system stack, which contains multiple instances of the Button class. A script defines methods and variables that apply only to a single instance.

Each class is part of the class hierarchy. Figure 160 shows part of the HyperLook class hierarchy. The class hierarchy defines how methods and variables are inherited.

Figure 160
Part of the
HyperLook class
tree.



A button, for example, inherits its variables and methods from the classes `ClassObject`, `HyperLook`, `RootObject`, `GraphObject`, `CardObject` and `Button`.

Super Classes and Sub Classes

The parent of a class is called a *super* class. For example, class **CardObject** is a super class of class **Button**.

The child of a class is called a *sub* class. For example, class **BackGround** has two sub classes: class **Card** and class **StackFrame**.

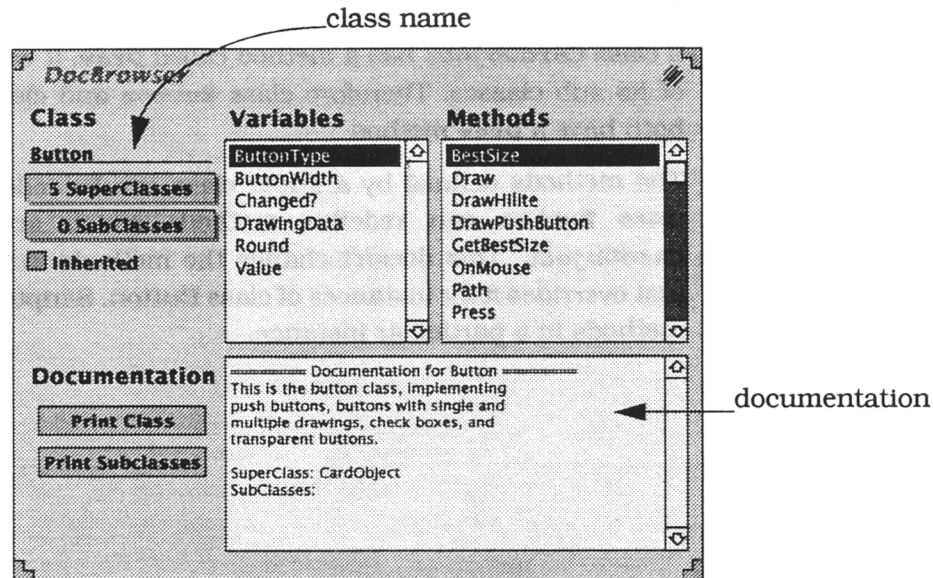
A super class's methods are inherited by its sub classes. This means that if class **CardObject** has a method called **Draw**, it will be inherited by all of its sub classes. Therefore class **Button** and class **Slider** therefore both have a **Draw** method

The methods defined by a super class can be redefined. For example, class **Button** can redefine methods which are defined in class **CardObject**. This doesn't change the methods in class **CardObject**, it just overrides it for instances of class **Button**. Scripts often redefine class methods in a particular instance.

The Documentation Browser

HyperLook provides a tool which lets you examine the class hierarchy and the methods and variables of the HyperLook classes. It is called the documentation browser. Select **DocBrowser** from the **Tools** menu in the system stack to show the documentation browser.

Figure 161
The documentation browser.



The documentation browser lets you examine the documentation of methods and variables of HyperLook classes. Every HyperLook class method and variable has some documentation describing briefly what it does.

Using the Documentation Browser

To view the documentation on a class you must enter the name of the class in the **Class** field and type return. The methods and variables of the class are then displayed in the **Variables** and **Methods** lists.

The **Inherited** checkbox determines which methods and variables are shown. If the checkbox is checked, all methods and variables that are inherited from the super classes are displayed. If it is not checked, only those methods and variables which are defined by the class itself are displayed.

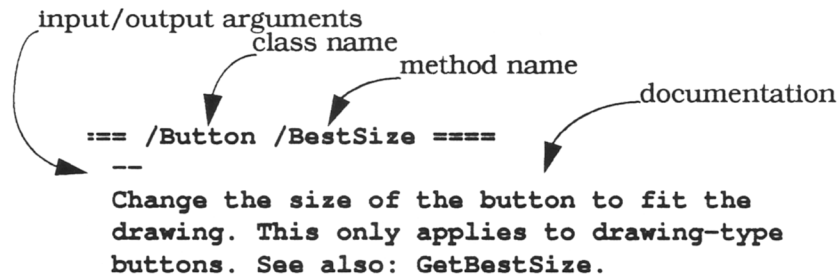
You can also select classes from the **SuperClasses** and **SubClasses** pulldown items. The **SuperClasses** pulldown contains all the super classes of the current class. The **SubClasses** pulldown contains all the sub classes.

Printing Class Documentation

You can print the class documentation by pressing **Print Class** ("Specifying a Printer" on page 3-32). This prints an alphabetic listing of the variables and methods of the current class. There are many pages of source documentation associated with each class, the printouts can take up a lot of paper!

Figure 162

An example of some documentation for the method **BestSize** of class **Button**.



```

input/output arguments
class name
method name
documentation
==== /Button /BestSize ====
--
Change the size of the button to fit the
drawing. This only applies to drawing-type
buttons. See also: GetBestSize.

```

The header sometimes contains the words *persistent* or *instance*, these indicate the type of methods or variable. Don't worry about them, they are only significant for advanced programmers.

Pressing **Print SubClasses** prints documentation for the current class and all its subclasses. The **Inherited** checkbox is turned off to save trees.

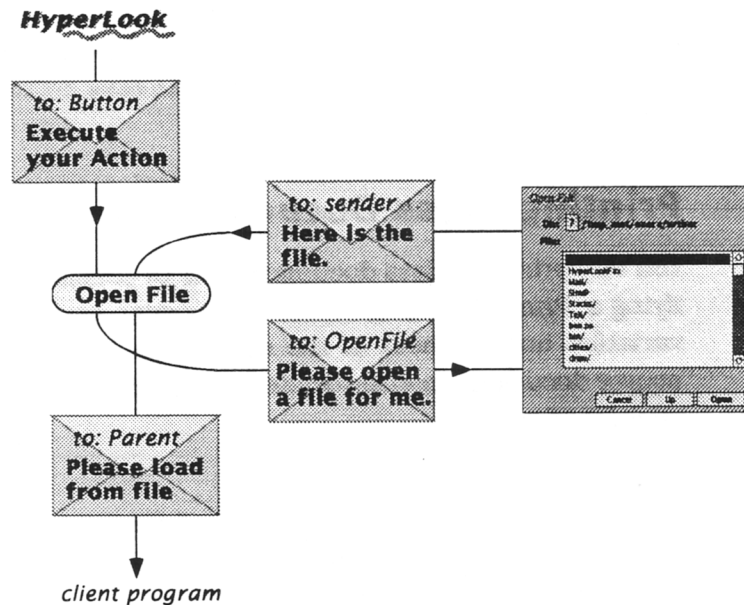
To print the class documentation of *all* HyperLook classes, select the **HyperLook** class and press **Print SubClasses**.

Messages

HyperLook uses message passing to communicate between objects and between objects and clients. Figure 163 shows the flow of messages when a button is pressed to open a file.

Figure 163

Communication between a Button, a stack and a client.



The HyperLook system sends a message to the **Open File** button when it is pressed. The button responds by asking the OpenFile stack to open a file, and the resulting file name is send back to the button. The button then passes the message onto its parent.

Messages and Methods

Every message has a name. For example, the HyperLook system sends a **Draw** message to a button when it needs updating. The button has a *method* called **Draw** which draws the stack.

The methods of an object are defined in its *class* or in its script. You can define new methods by editing the script of the object.



Most programming languages let you write functions which can be accessed from anywhere, they are defined globally. Object oriented systems let you define functions which are known only in the scope of an object.

This not only means that objects can have different methods with the same names, but also that a message can mean different things to different objects. For example, the **Draw** method of a button draws a button while the **Draw** method of a slider draws a slider.

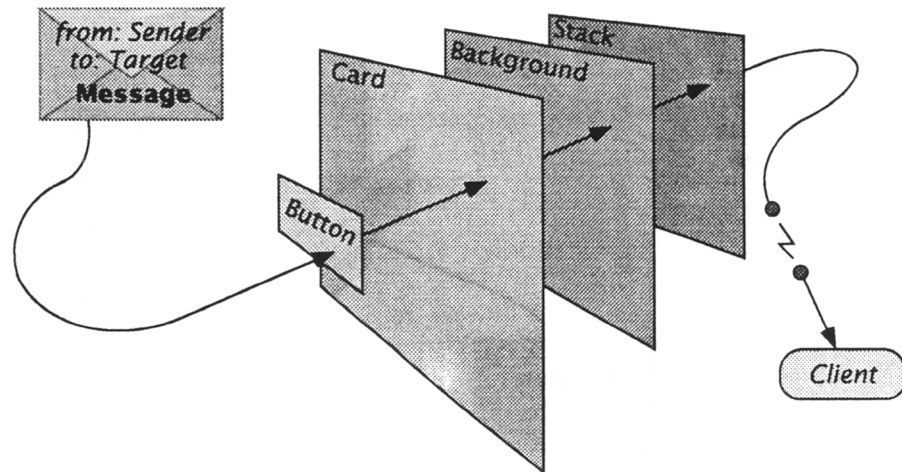
Sending a message to an object really means calling a function in the *context* of the object. The *context* of an object is defined by its class¹.

The Message Hierarchy

What happens when a message is sent to an object and the object does not define the method? In some systems this would cause an error! Not in HyperLook.

Figure 164

The HyperLook message hierarchy.



When a message is not defined by an object, it is passed on to the *parent* of the object. If the parent of the object does not define the message, it is sent on to its parent, and so on. Eventually the end of the chain is reached and the message is either sent to a client program or it is lost.

1. The *class* of an object and the *type* of an object really mean the same thing.

This path is traversed to find the receiver of a message:

1. Component (Button, Slider, etc.)
2. Card
3. Background
4. Stack
5. Client (only when connected)

The message is delivered to the first object in the chain that understands the message. If the message is not delivered, it is discarded, or it is sent to the client. You can also send messages directly to a card, background, stack or client.

A message is sent from the **Sender** to the **Target**. The **Target** is the object which was identified by the sender as the initial target of the message. If the target can't handle the message, it is passed on to its parent until it's handled by the **Receiver**.

Figure 165

A message is send by the **Sender**, it is aimed at the **Target** and it is handled by the **Receiver**.

